HBase Architecture

Cluster Architecture

In this chapter you will learn

- **The HBase terminology**
- **How Regions work**
- **Understand minor and major compactions**
- **Understand how crash recovery works**
- **How the Block Cache works**
- **How a RegionServer splits a region**

Hbase Data model

Table: HBase organizes data into tables . Table names are Strings and composed of characters that are safe for use in a file system path .

Row: Within a table, data is stored according to its row . Rows are identified uniquely by their *row key* . Row keys do not have a data type and are always treated as a *byte[ ]* (byte array) .

Column Family:

- Data within a row is grouped by column family .
- Column families also impact the physical arrangement of data stored in HBase . For this reason, they must be defined up front and are not easily modified .
- Every row in a table has the same column families, although a row need not store data in all its families .
- Column families are Strings and composed of characters that are safe for use in a file system path .

Hbase Data model

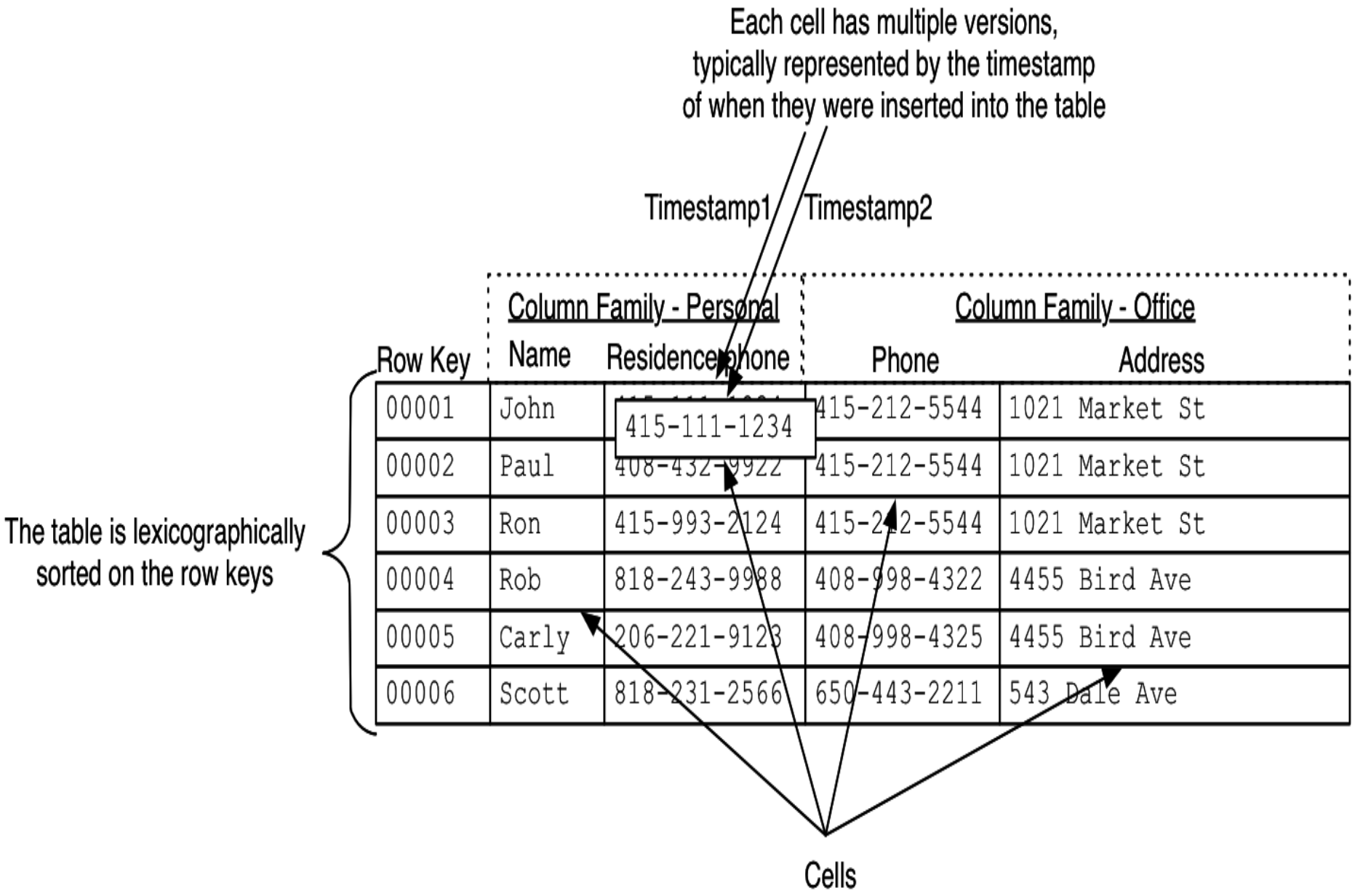
Column Qualifier: Data within a column family is addressed via its column qualifier, or simply, column . Column qualifiers need not be specified in advance . Column qualifiers need not be consistent between rows . Like row keys, column qualifiers do not have a data type and are always treated as a byte[ ] .

Cell: A combination of row key, column family, and column qualifier uniquely identifies a cell . The data stored in a cell is referred to as that cell's value . Values also do not have a data type and are always treated as a byte[ ] .

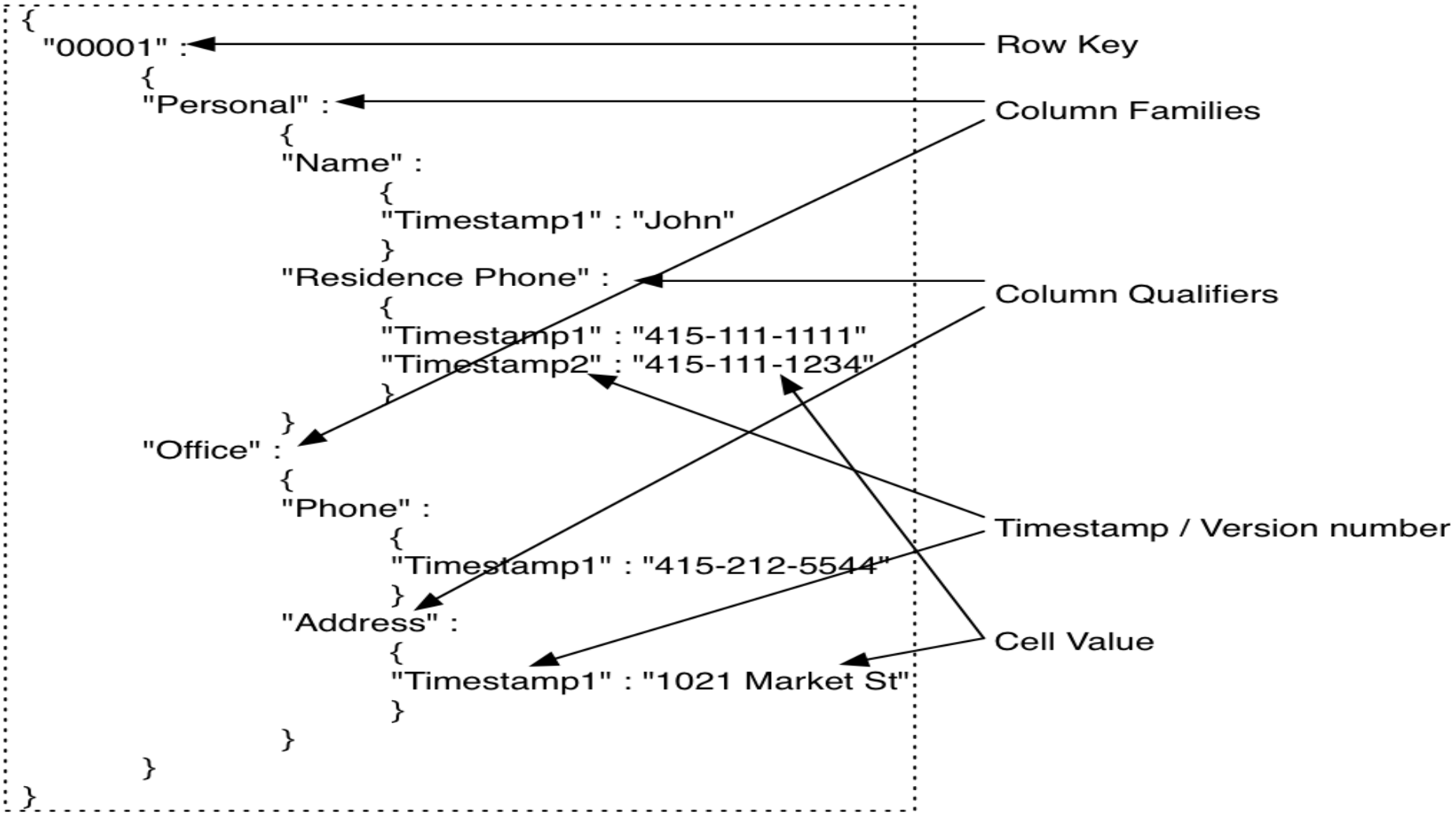
Timestamp:

- Values within a cell are versioned . Versions are identified by their version number, which by default is the timestamp of when the cell was written .
- If a timestamp is not specified during a write, the current timestamp is used . If the timestamp is not specified for a read, the latest one is returned .
- The number of cell value versions retained by HBase is configured for each column family .
- The default number of cell versions is three .

Data ina Hbase Table



Tree Representation



Key-Value Representation

00001 → { Personal : { Name : { Timestamp1 : John }, Residence Phone : { Timestamp1 : 415-111-1234 } },
{ Office : { Phone : { Timestamp1 : 415-212-5544 }, Address : { Timestamp1 : 1021 Market St } } }

00001 , Personal → { Name : { Timestamp1 : John }, Residence Phone : { Timestamp1 : 415-111-1234 } }

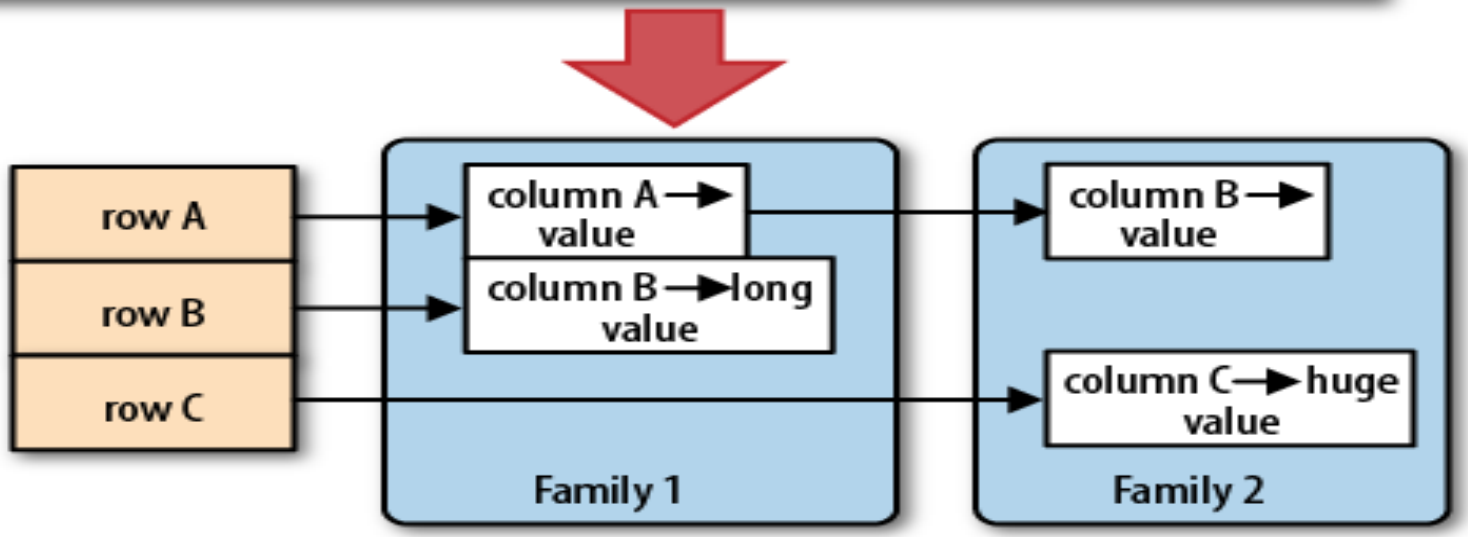
00001, Personal:Residence Phone → { {Timestamp1 : 415-111-1111 } , { Timestamp2 : 415-111-1234))

00001, Personal:Residence Phone , Timestamp1 → { 415-111-1111)

00001, Personal:Residence Phone , Timestamp2 → { 415-111-1234)

Relational Vs NoSQL

	column A (int)	column B (varchar)	column C (boolean)	column D (date)
row A				
row B				
row C			NULL?	
row D				

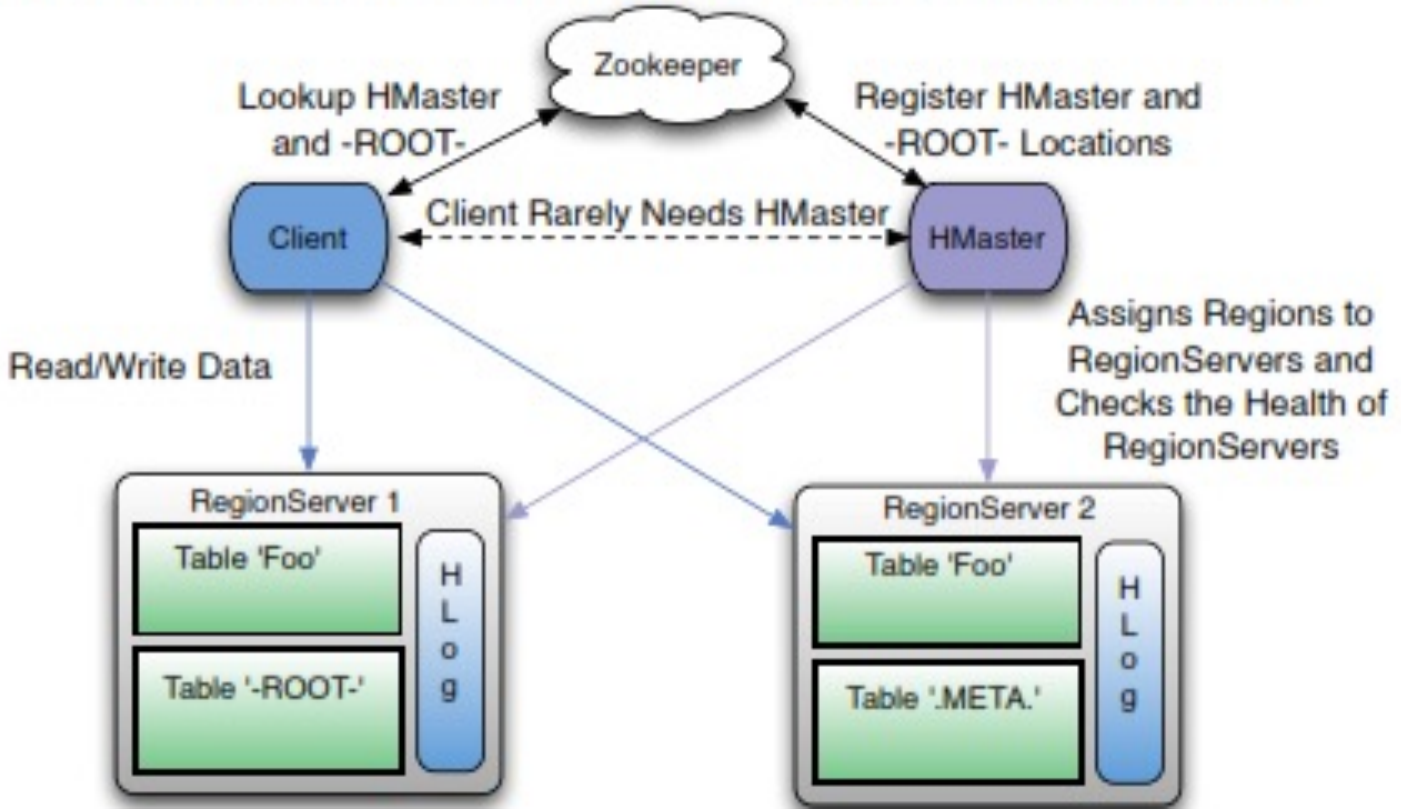


Major Components of an HBase Cluster

- **Zookeeper**
  - A centralized service used to maintain configuration information and service for HBase
- **Catalog Tables**
  - Keep track of the locations of RegionServers and Regions
- **Master**
  - Monitors all RegionServer instances in the cluster
  - The interface for all metadata changes
- **RegionServer**
  - Responsible for serving and managing regions
- **Region**
  - A set of rows belonging to a table

Zookeeper

- **Zookeeper Service**
  - Stores global information about the cluster
  - Provides synchronization and detects Master node failure
  - Holds the location of the -ROOT- table and the Master



Catalog Tables

- **-ROOT- Catalog Table**
  - A table that lists the location of the .META. table(s)
  - The following is an example of: scan '-ROOT-'!

ROW	COLUMN+CELL
.META.,,1	column=info:regioninfo, timestamp=1309921036780, value=REGION => {NAME => '.META.,,1', STARTKEY => '', END KEY => '', ENCODED => 1028785192, TABLE => {{NAME => '.META.', IS_META => 'true', FAMILIES => [{NAME =>'info', BLOOMFILTER => 'NONE', REPLICATION_SCOPE => '0', VERSIONS => '10', COMPRESSION => 'NONE', TTL=> '2147483647', BLOCKSIZE => '8192', IN_MEMORY => 'true', BLOCKCACHE => 'true'}]}}
.META.,,1	column=info:server, timestamp=1322688307364, value=localhost.localdomain:55537
.META.,,1	column=info:serverstartcode, timestamp=1322688307364, value=1322688297223

Catalog Tables (cont.)

- **.META. Catalog Table**
  - A table that lists all the regions and their locations
  - Very large clusters may have multiple .META. Tables
  - The following is an example of: scan '.META.'!

ROW	COLUMN+CELL
USER	column=info:regioninfo,timestamp=1319211648984, value=REGION => {NAME =>'USER,,1319211648881.8413c02a2cc 65f57a97faabe025f. 72ec65f57a97faabe025f.', STARTKEY=> '', ENDKEY => '', ENCODED =>8413c02a2cc72ec65f57a97faabe025f, TABLE=> {{NAME => 'USER', FAMILIES =>[{NAME => 'INFO', BLOOMFILTER => 'NONE', REPLICATION_SCOPE => '0',COMPRESSION => 'NONE', VERSIONS => '1',TTL => '2147483647', BLOCKSIZE => '65536', IN_MEMORY => 'false', BLOCKCACHE => 'true'}]}}
USER	column=info:server,timestamp=1322688307710, value=localhost.localdomain: 5553765f57a97faabe025f.
USER	column=info:server,timestamp=1322688307710, value=localhost.localdomain: 5553765f57a97faabe025f.

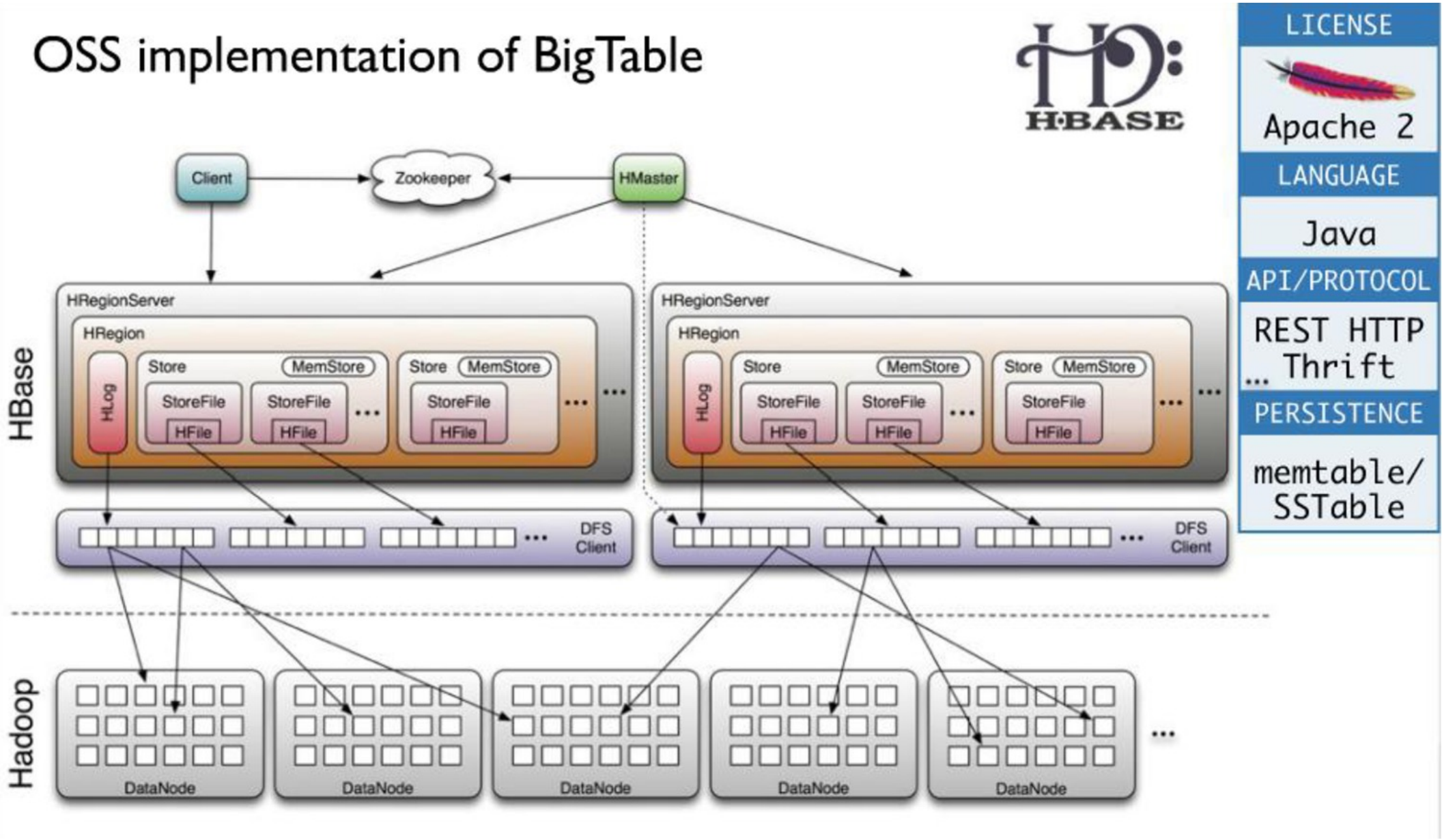
HMaster

- **Responsible for coordinating the slaves (HRegionServers)**
- **Assigns regions, detects failures of HRegionServers**
- **Handles schema changes**
- **Master runs several background threads**
  - LoadBalancer Periodically reassigns Regions in the cluster
  - CatalogJanitor periodically checks and cleans up the .META. Table
- **Can have multiple Masters**
  - Upon startup all compete to run the cluster
  - If the active Master loses its lease in Zookeeper then the remaining Masters compete for the Master role

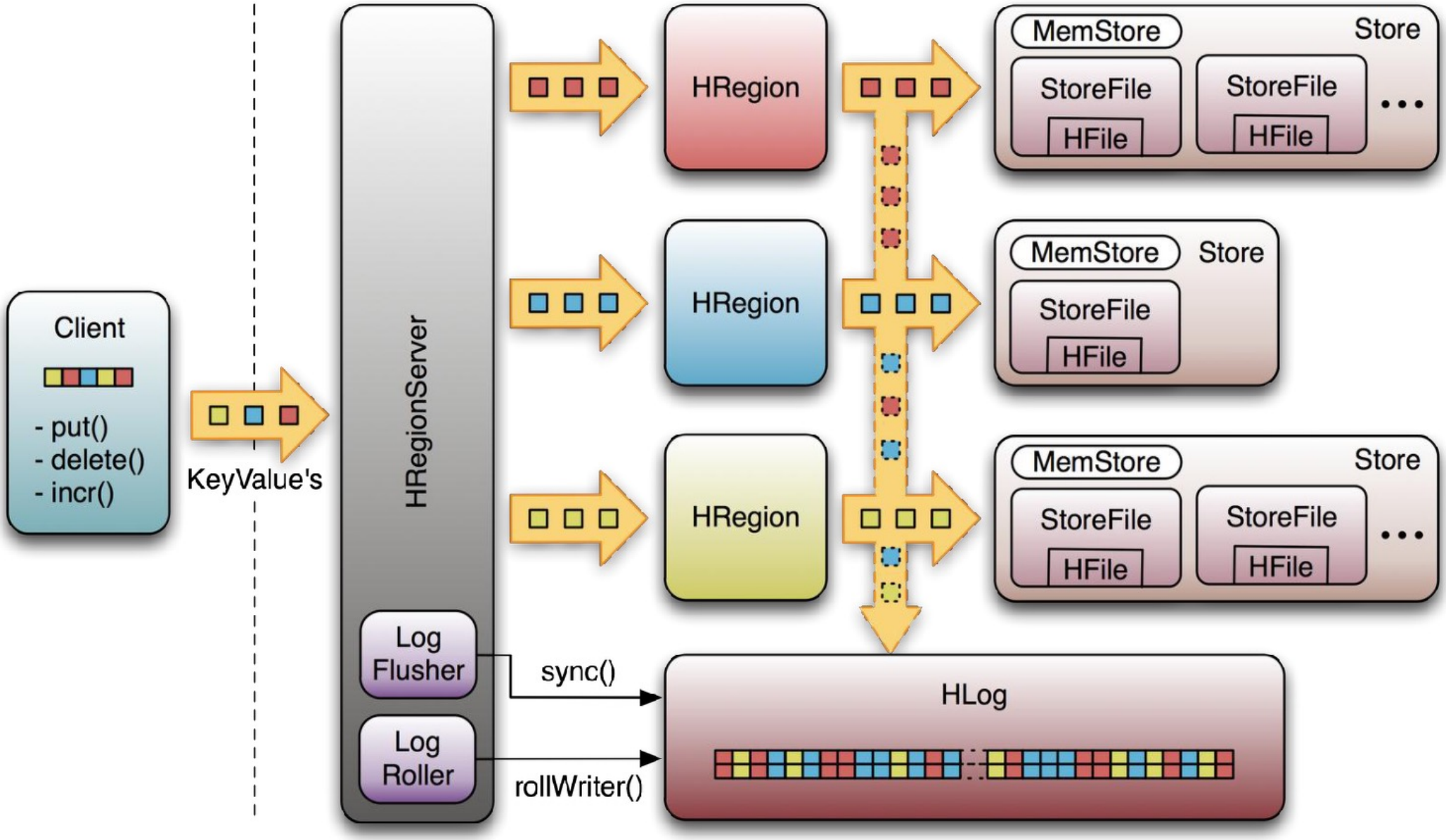
RegionServers

- **Serve data for reads and writes of rows contained in Regions**
- **Also will split a Region that has become too large**
- **Interface Methods exposed by HRegionRegionInterface**
  - Data Methods
    - Get, put, delete, next, etc.
  - Region Methods
    - splitRegion, compactRegion, etc.
- **Runs several background threads**
  - CompactSplitThread checks for splits and handle minor compactions
  - MajorCompactionChecker checks for major compactions
  - MemStoreFlusher periodically flushes in-memory writes in the MemStore to StoreFiles.
  - LogRoller periodically checks the RegionServer's HLog

Hbase Architecture

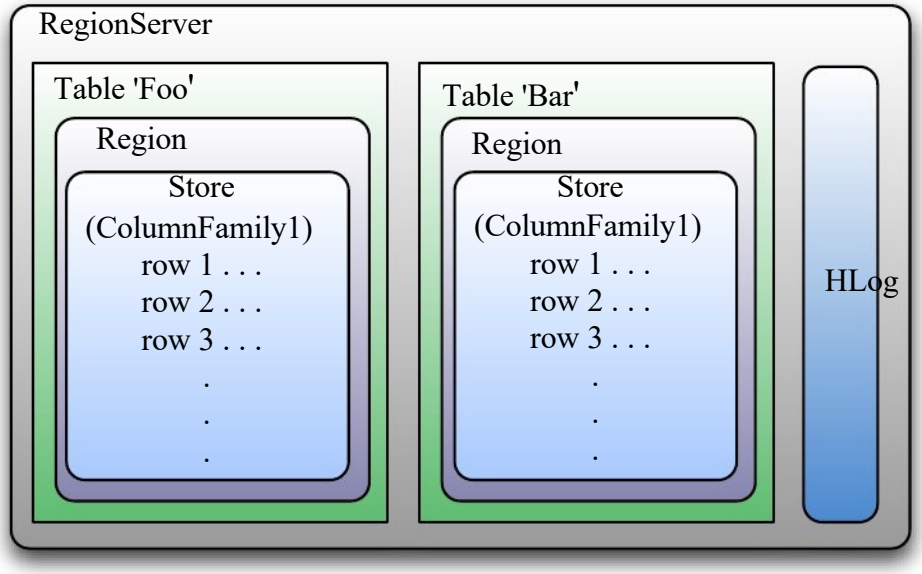


Hbase Architecture



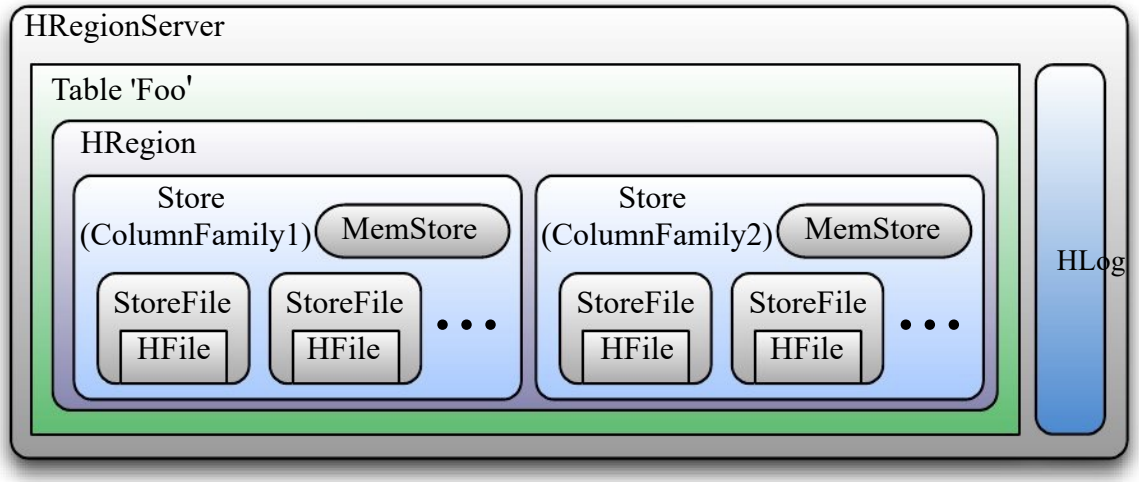
Regions

- **Holds a subset of a table's rows, like a partition**
- **A table may have one or more Regions**
  - Comprised of a Store per Column Family
- **New Regions are automatically created as tables grow**



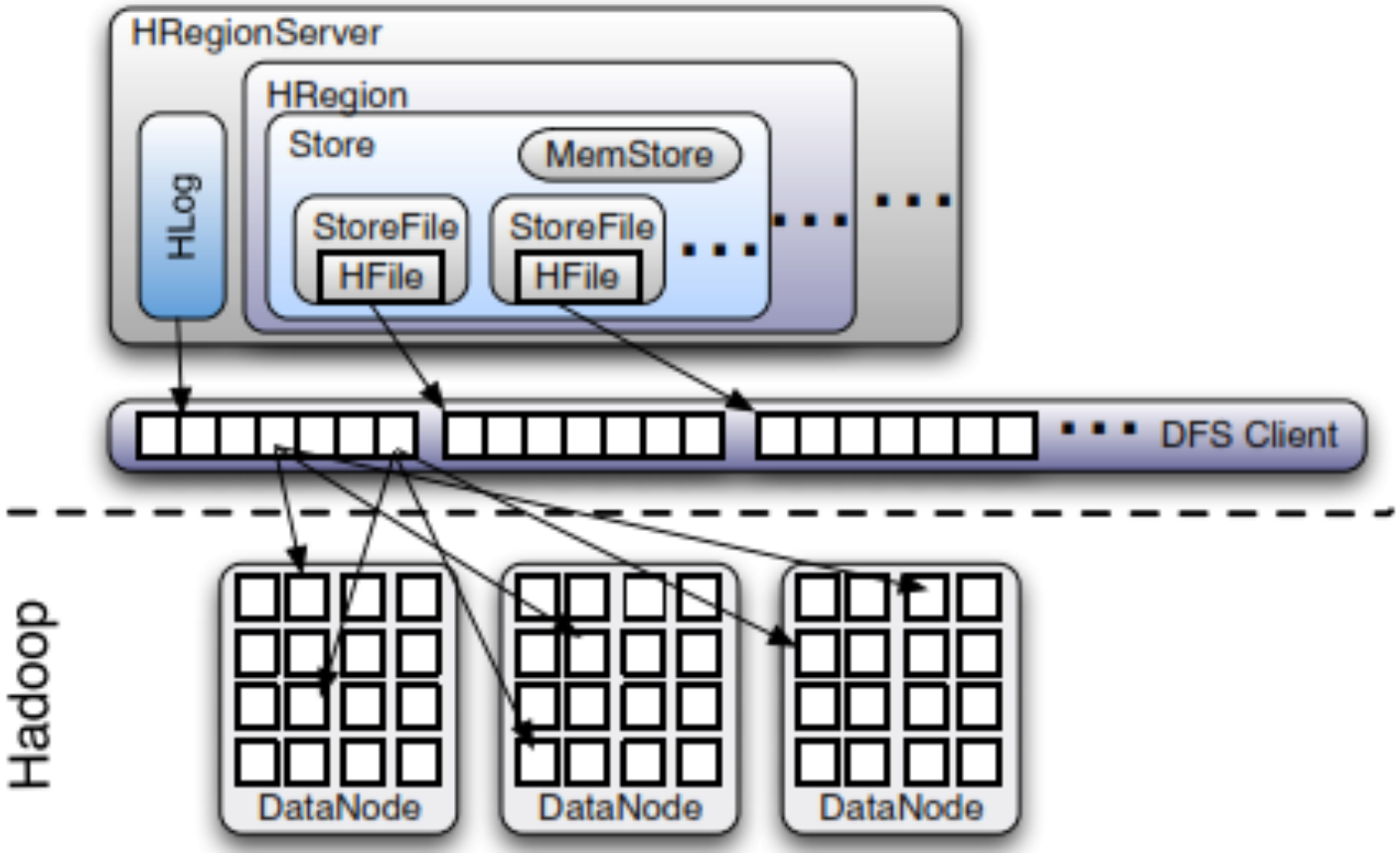
Region Store

- **A Store hosts a MemStore and 0 or more StoreFiles (HFiles)**
- **Store corresponds to a column family for a table for a given region**
- **MemStore**
  - Holds in-memory modifications to the Store
  - Flush writes to disk and clears the MemStore
- **StoreFile (HFile)**
  - Actual data storage



Region Files & HDFS

- **Files are divided up into smaller blocks when stored in HDFS**
- **Block replication is handled by HDFS**

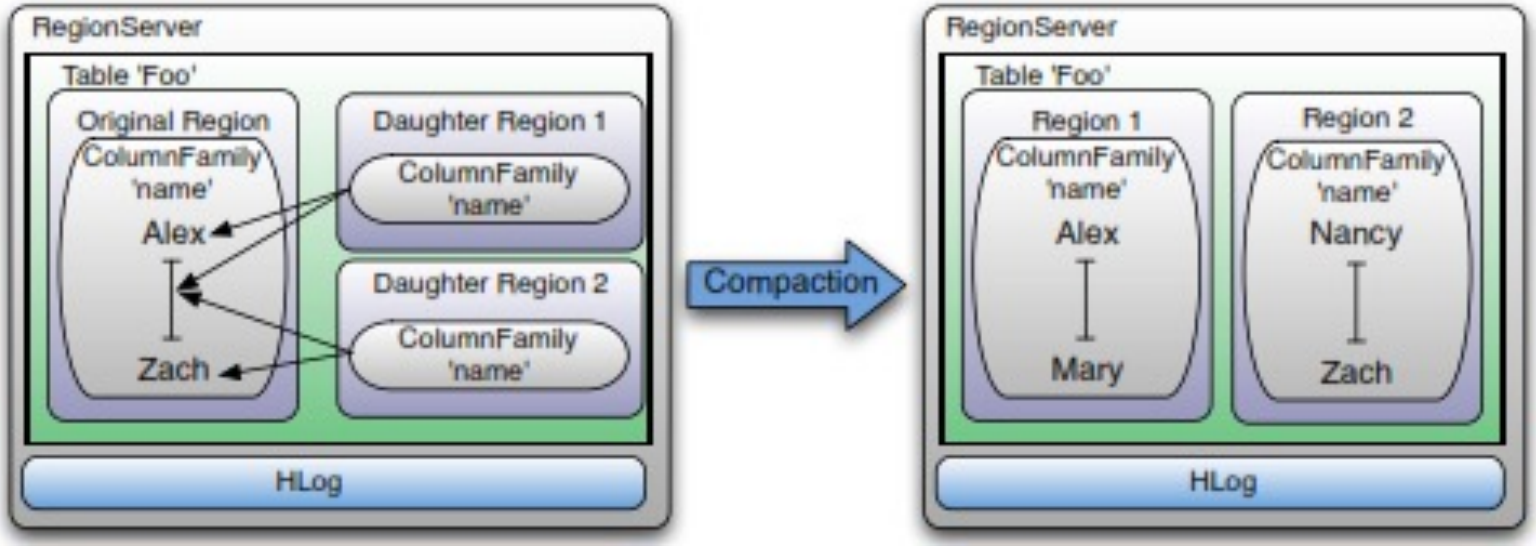


Split Regions

- **When Regions get too big (256MB) they are automatically split**
  - HBase create two “daughter” reference files
  - Daughter files only contain the key where region was split
  - Reference are used by HalfHFileReader class to read original regions
  - At Major compaction the original data files are rewritten into separate files in the new region directory
  - Small reference files and original region are removed

Region Splits (cont.)

- **RegionServer handles the region splits**
  1. Offlines the original region
  2. Adds daughter regions to .META.
  3. Opens daughter regions on parent RegionServer
     - Daughter regions reference the original regions until compaction (discussed later)
  4. Reports the split to the Master



Region Size

- **Region Size**
  - Basic element of availability and distribution
    - i.e., you do not want high number of regions for small amount of data
  - High Region count (e.g., > 3000) can impact performance
  - Low Region count prevents parallel scalability
- **Load Balancer**
  - Automatically moves regions around to balance cluster load
  - Period when this runs is configurable
- **Data Distribution**
  - Data is distributed across Regions based on Row Keys

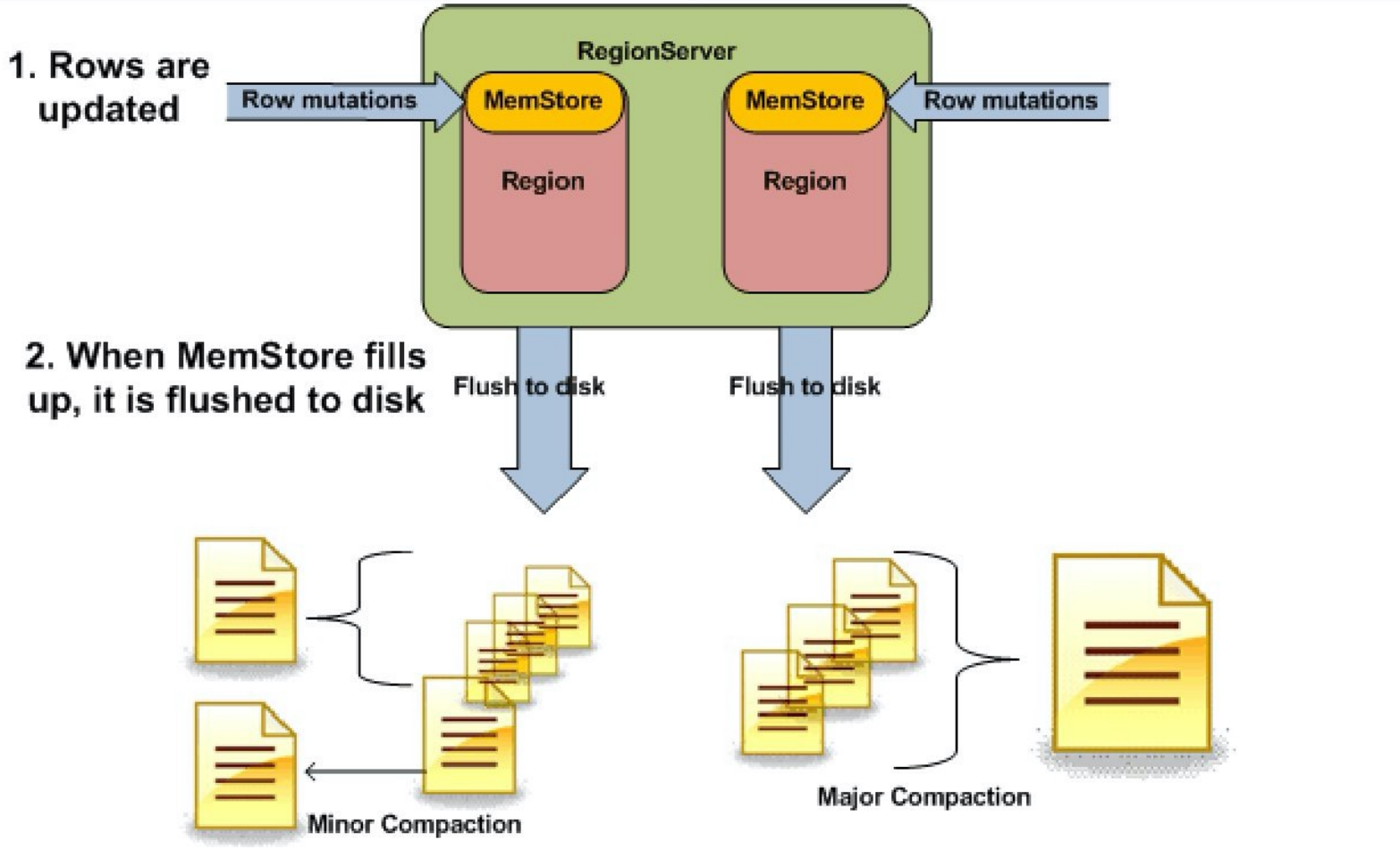
Data Storage

- **Data is initially written to the Region's MemStore**
  - The HLog is required for crash recovery if the MemStore is lost
- **When the MemStore gets to a certain size, it will flush to an immutable file (Store file)**
- **Eventually these store files will be aggregated and cleaned up during a compaction**

Compactions

- **Minor Compaction**
  - Combines some Store files into a single file
  - Runs after three Store files have accumulated
  - Can be configured with hbase.hstore.compactionThreshold
    - Larger number of compactions will take longer but will be fewer
  - MemStore cannot flush to disk during compaction
    - If MemStore runs out of memory, clients will hang/timeout
- **Major Compaction**
  - Reads all the Store files and writes to a single Store
  - Deleted rows and expired versions are removed
  - Happens once daily
  - Can be configured with hbase.hregion.majorcompaction
  - Heavyweight operation – run when load is low

Minor and Major Compaction



Cluster Architecture

Components of an HBase Cluster

Regions

Flushes and Compactions

Reading and Writing to HBase

Bloom Filters and Block Cache

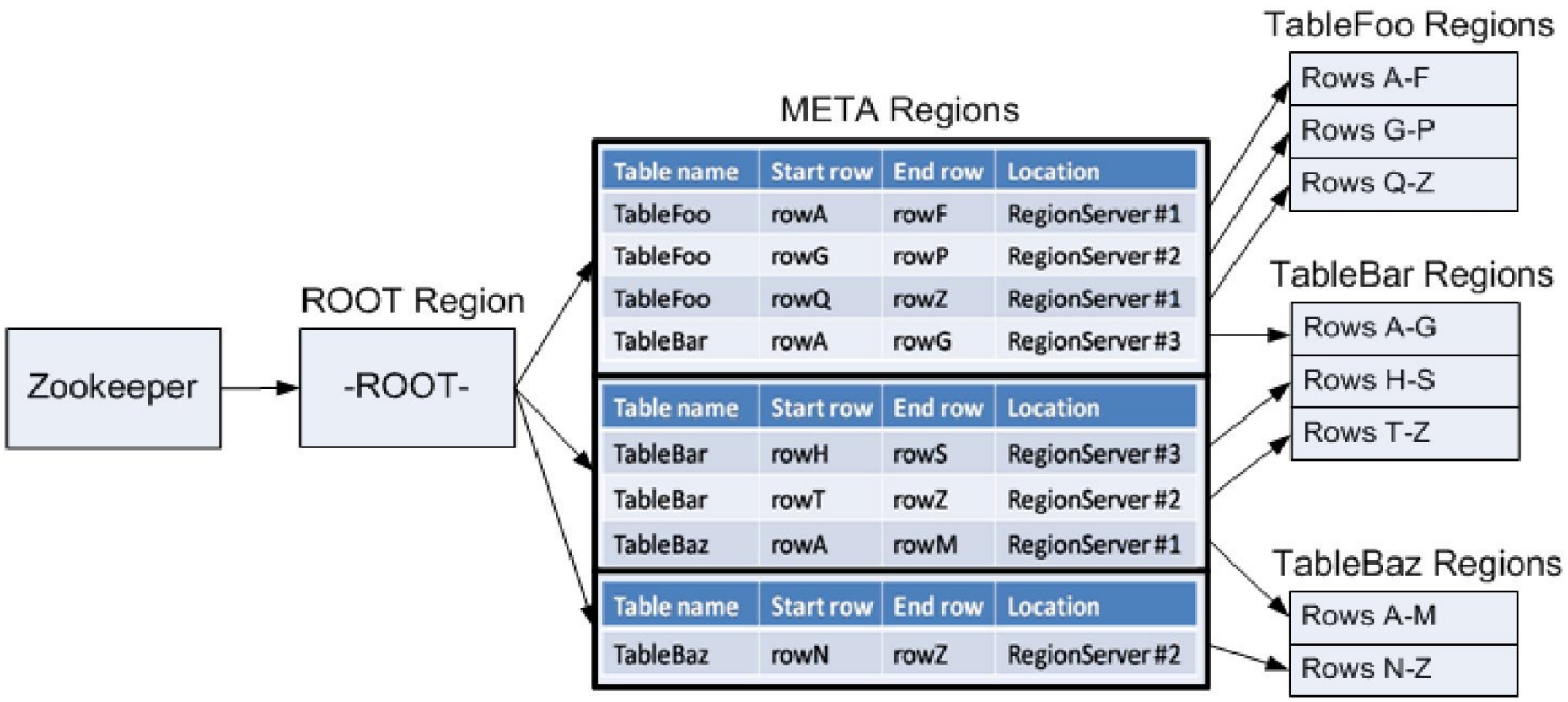
WAL and Crash Recovery

Conclusion

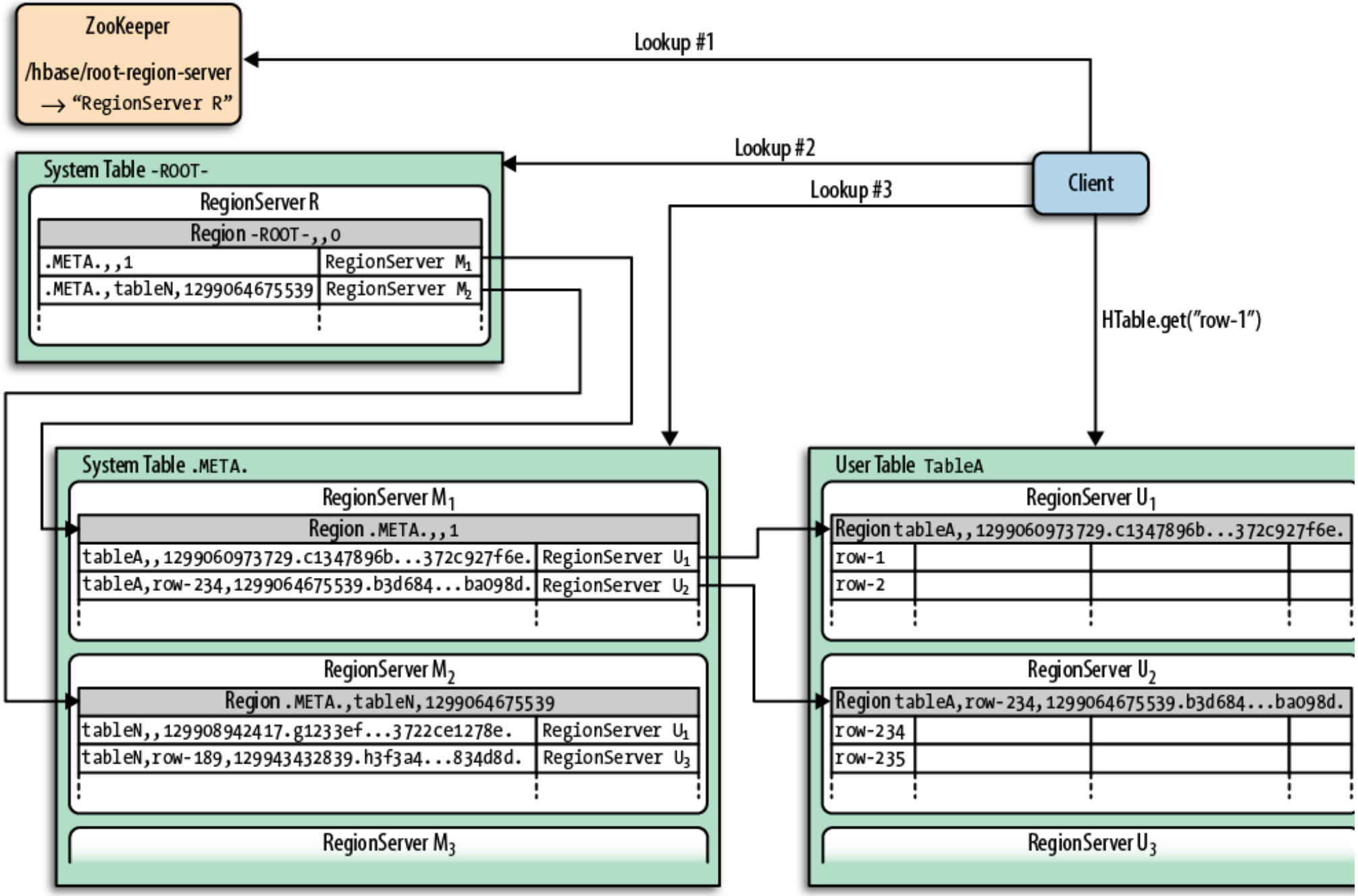
Client Reading and Writing

- **HTable Client**
  - Finds RegionServers that serve the row range of interest
  - Queries Zookeeper to find the location of HMaster and -ROOT-
  - Queries -ROOT- table to find the location of .META. table
  - Queries the .META. table to find the RegionServer hosting the Region of interest
  - Client then directly contacts the RegionServer and issues Read or Write request

Finding a Row



Row Key Lookups



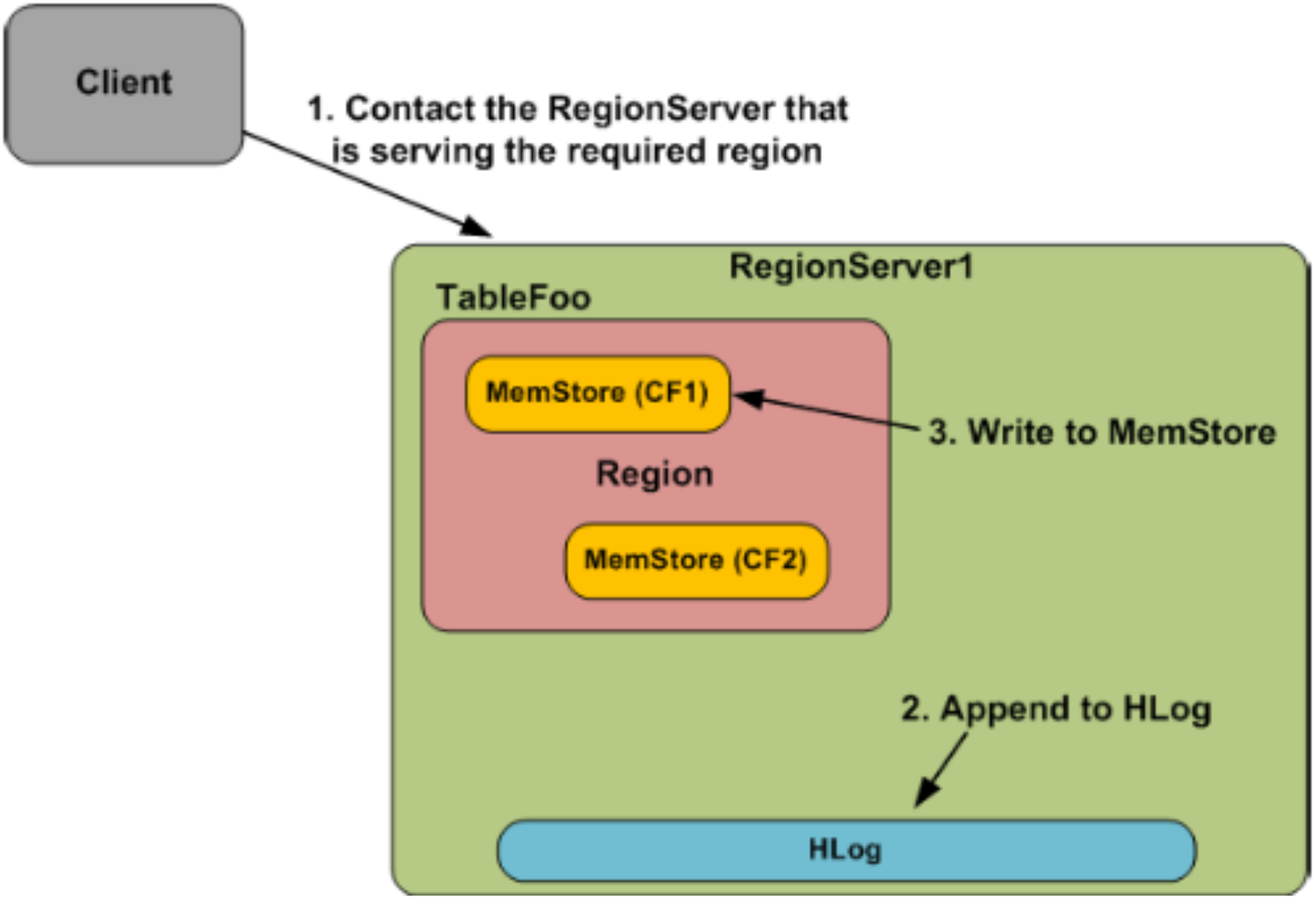
Caching Metadata

- **Client caches information from ZooKeeper**
  - Location of META and previously used Regions
- **The cache can become stale**
  - Regions may have moved
  - RegionServers can be unavailable
- **If Client fails to find a Region**
  - Re-reads .META. -> -ROOT- -> Zookeeper
  - Hbase.client.retries.number to set retry limit
    - Default is 10

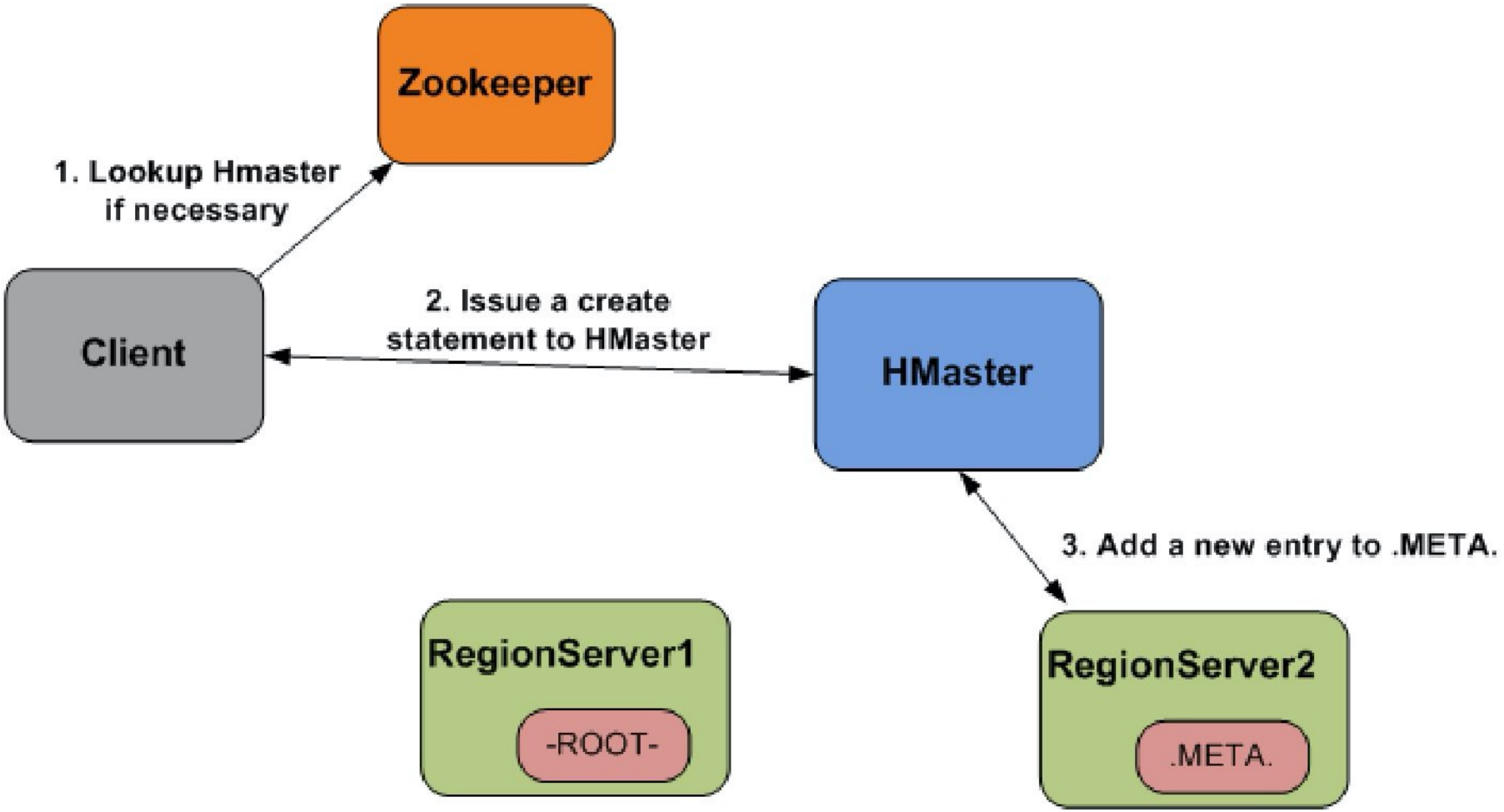
Writing Data to Storage

- **Client issues a HTable.put(Put) request to the HRegionServer**
- **Details are handed to the appropriate HRegion instance**
- **If the client flag Put.writeToWAL(boolean) is set then write to MemStore**
  - If MemStore is full then request a flush to disk
  - Flush is served by a separate HRegionServer thread
  - Flushed data is written to an HFile in HDFS
  - Sequence number is saved to keep track of persisted data

Modifying a Row in a Table



Creating a Table in HBase



Locating a Row with Bloom Filters

- **Brute force approach:**
  - RegionServer reads MemStore and *all* store files
- **HBase supports Bloom filters**
  - Eliminates the need to read every store file
  - Uses a probabilistic data structure to possibly skip one or more store files
  - Allows RegionServer to skip files that do not contain the row

Bloom Filters

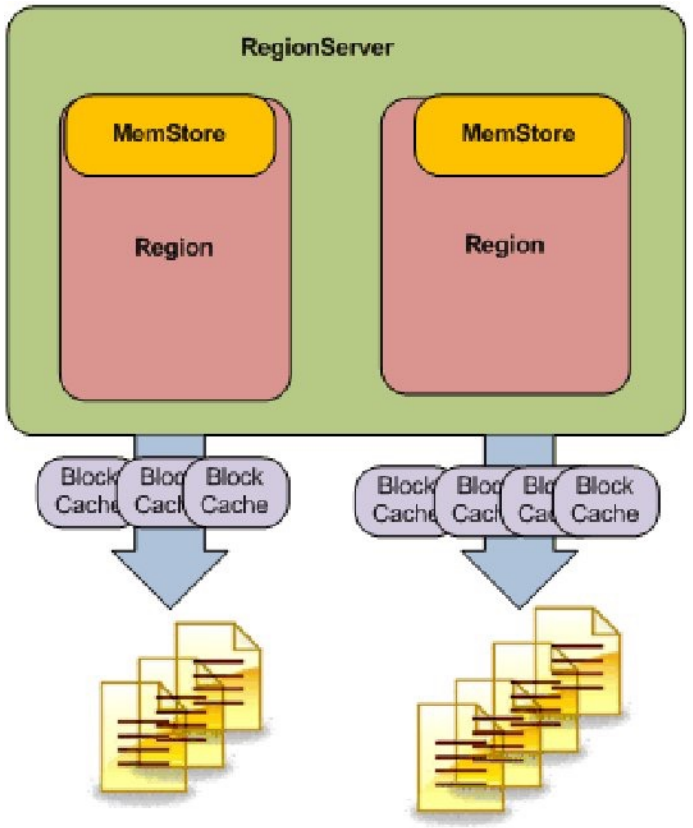
- **Use Cases**
  - Access patterns with lots of misses during reads
  - Speed up reads by cutting down internal lookups
  - Update all of the rows regularly
    - Majority of the rows will have a piece of data
  - Update in batches
    - Each row is written into only a few files at a time
- **Storage**
  - Stored in the meta data of each HFile
  - Never needs to be updated as HFiles are immutable
  - Filters add minimal overhead to storage

Bloom Filters (cont.)

- **Access**
  - When HFile is opened the bloom filter is loaded into memory
  - Used to determine if a given key is in that store file
  - Can be scoped on a row key or column key level
    - Column key need more space as it stores many row keys
- **Applying Bloom Filters**
  - Enabled on a per-ColumnFamily basis
  - HColumnDescriptor.setBloomFilterType(NONE | ROW | ROWCOL)
    - NONE = Default
    - ROW = Hash of the row added to bloom on each insert
    - ROWCOL = Hash of the row + column family + column family

The Block Cache

- Blocks read from the storage files are cached internally in configurable caches
- Each Store file has a block-level cache to avoid reading data from disk
- Block is normally 64K
- Default is 20% of RegionServer Heap
- MemStore is write cache, Block Cache is read cache
- Similar to a “buffer pool”
- Lease-Recently Used (LRU) policy to evict blocks from cache

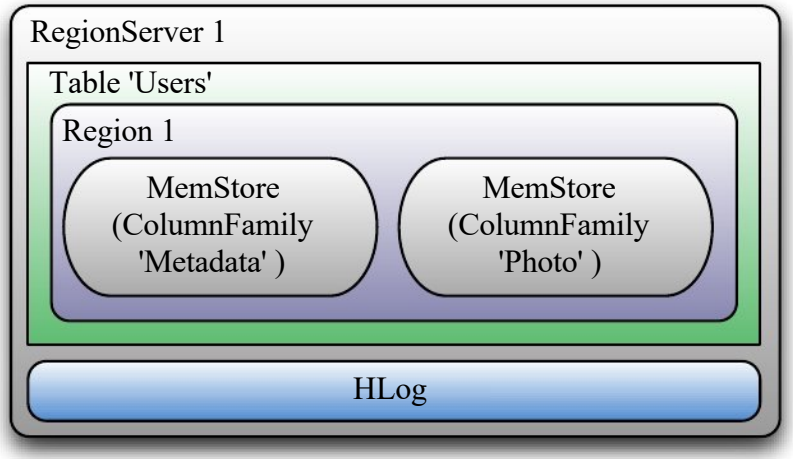


HBase File types

- **HBase has two file types:**
  - HLog
    - Used for the Write Ahead Log (WAL)
    - Standard Hadoop SequenceFile
    - Stores HLogKeys as well as actual data
    - HLogKeys are used to replay not yet persisted data after a server crash
  - HFile
    - Used for actual data storage

Write Ahead Log (WAL)

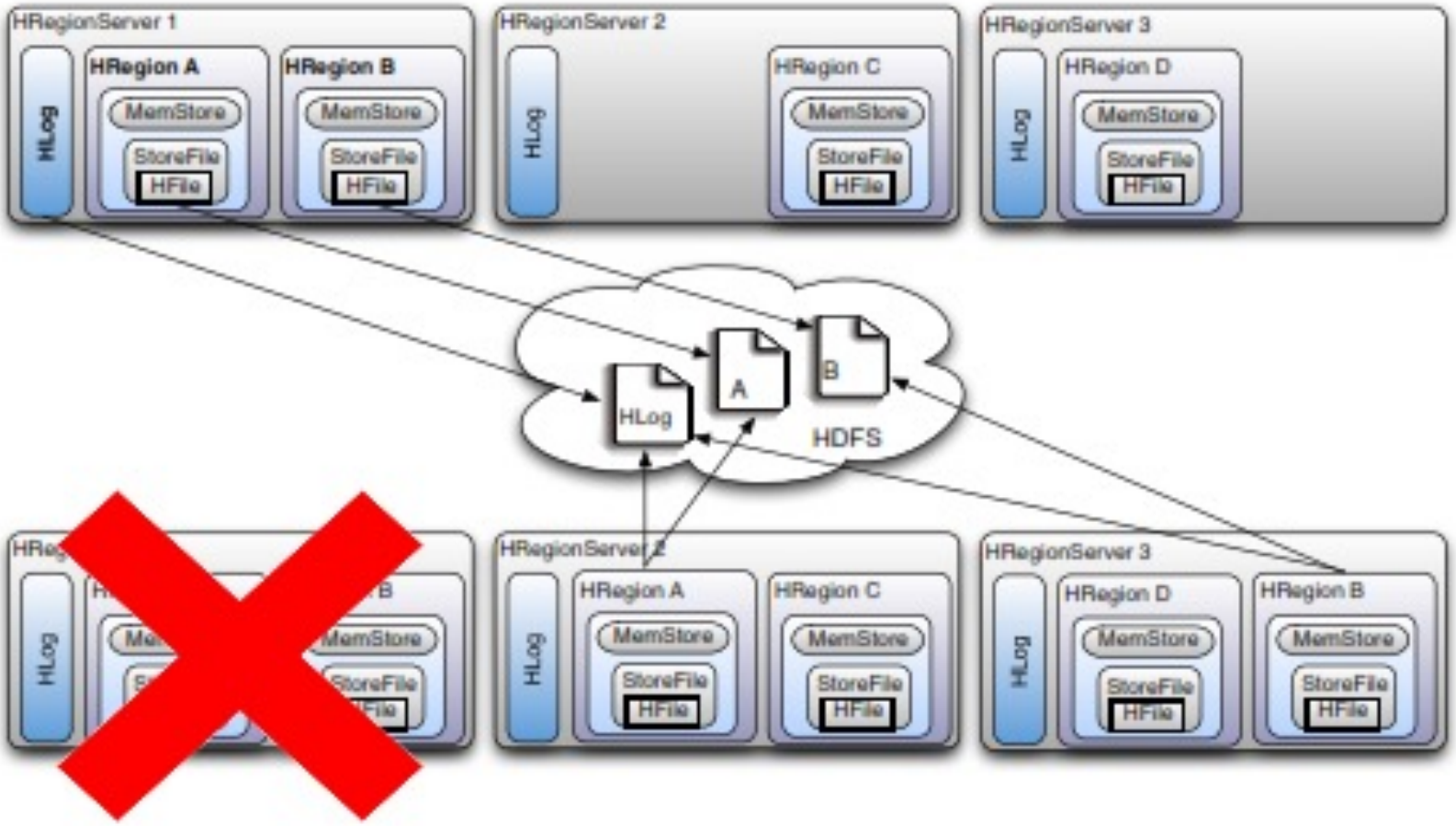
- **RegionServer update (Puts, Deletes)**
  - Added to WAL before written to MemStore
  - Ensures durable writes
- **HLog is the WAL implementation**
  - One HLog instance per RegionServer
  - Stored in HDFS in /hbase/.logs with subdirectories per region



RegionServer Crash

- **MemStore is in memory and its data is lost**
  - Recent changes would still be in MemStore
  - HLog is used to reapply the recent changes
- **Region StoreFiles (HFiles) are stored in HDFS**
  - HDFS replicates storage blocks automatically by default
- **Master notices that Regions are not being served**
  - Master splits the Hlog into a per-region log
  - Reassigns Regions to another RegionServer
  - RegionServer automatically applies the log

Region Server Failover



DDL Statements

create - Creates a table.
list - Lists all the tables in HBase.
disable - Disables a table.
is_disabled - Verifies whether a table is disabled.
enable - Enables a table.
is_enabled - Verifies whether a table is enabled.
describe - Provides the description of a table.
alter - Alters a table.
exists - Verifies whether a table exists.
drop - Drops a table from HBase.
drop_all - Drops the tables matching the ‘regex’ given in the command.

DML Statements

put - Puts a cell value at a specified column in a specified row in a particular table.
get - Fetches the contents of row or a cell.
delete - Deletes a cell value in a table.
deleteall - Deletes all the cells in a given row.
scan - Scans and returns the table data.
count - Counts and returns the number of rows in a table.
truncate - Disables, drops, and recreates a specified table.

Sample data

```
create 'emp','personalData','professionalData'
put 'emp','1','personalData:name','ravi‘
put 'emp','1', 'personalData:city','hyderabad'
put 'emp','1', 'professionalData:desig','manager‘
put 'emp',‘2','personalData:name',‘kiran‘
put 'emp',‘2', 'personalData:city',‘banglore',
put 'emp',‘2', 'professionData:desig',‘Analyst‘
```

Sample data contd..

list
scan ' emp '
get 'emp', '1'
get 'emp', '1','personalData'
get 'emp', '1', {COLUMN => 'personalData:name’}
hdfs dfs -ls /hbase/data/default
deleting a cell
delete 'emp', '1', 'personalData:city'
alter 'emp', 'delete' ⇒ 'professionalData'
disable 'emp'
enable 'emp’
describe 'emp'