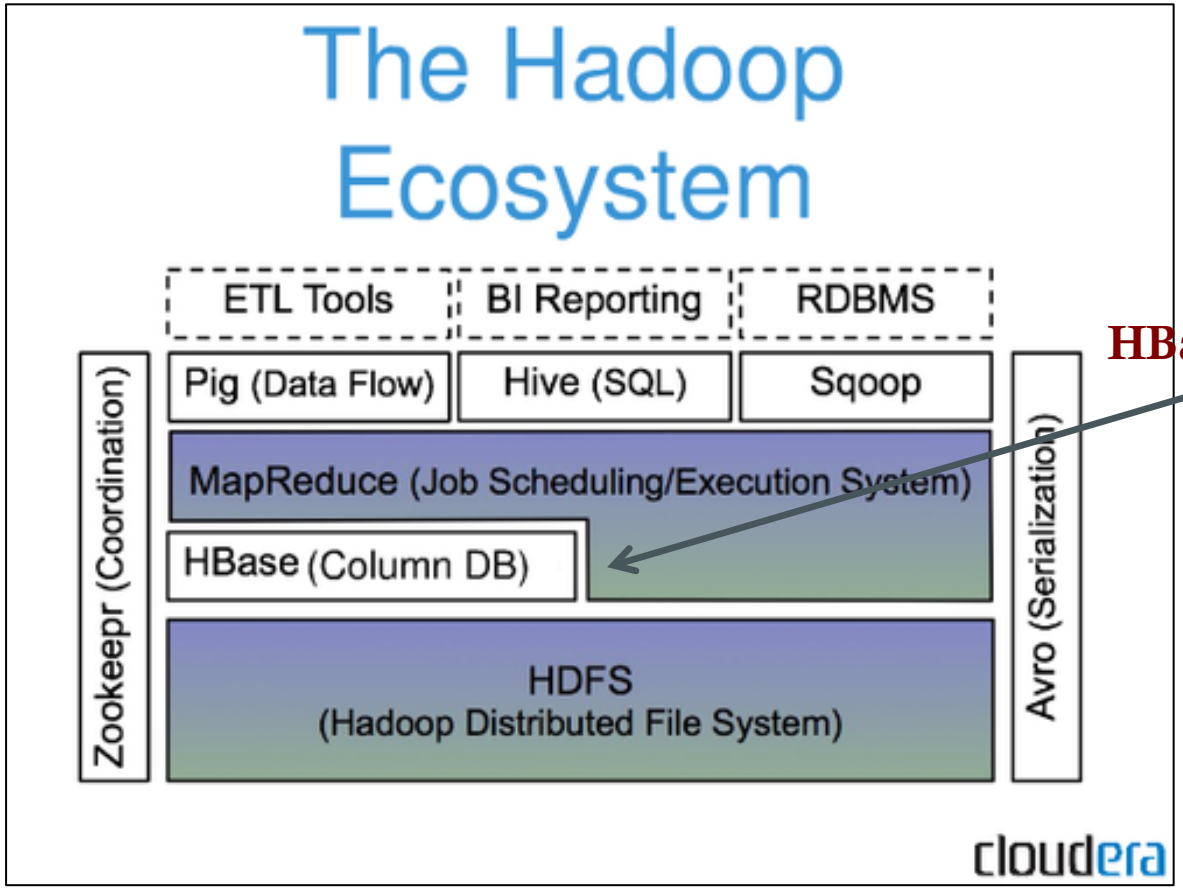
HBase: Overview

- **HBase is a distributed column-oriented data store built on top of HDFS**
- **HBase is an Apache open source project whose goal is to provide storage for the Hadoop Distributed Computing**
- **Data is logically organized into tables, rows and columns**

HBase: Part of Hadoop's Ecosystem



HBase is built on top of HDFS

HBase files are internally stored in HDFS

HBase Data Model

HBase

- HBase looks like a table, but it is **NOT a relational table**.
- It is a **sparse, distributed, multidimensional, sorted map**:

(RowKey, ColumnFamily, ColumnQualifier, Timestamp) → Value

edureka!

Row Key | Column Family

Row Key	Customers		Products	
Customer ID	Customer Name	City & Country	Product Name	Price
1	Sam Smith	California, US	MIke	$500
2	Arijit Singh	Goa, India	Speakers	$1000
3	Ellie Goulding	London, UK	Headphones	$800
4	Wiz Khalifa	North Dakota, US	Guitar	$2500

Column Qualifiers

Cell

Figure: HBase Table

Row Key

- Every row is identified by a single unique row key
- Data is sorted lexicographically by row key (not numeric sorting)
- Range scans are extremely fast due to this sorting.
- It is a byte array (byte[]) – everything converted to binary byte array

user_001
user_002
user_003

Best performance comes from designing the row key wisely.

Row Key

- **Any data type can be a row key:** String, Integer, Long, Timestamp, UUID, Composite keys, Hashes, Encoded binary keys
- All ultimately become byte[].

"1", "2", "10", "20"

"1", "10", "2", "20"

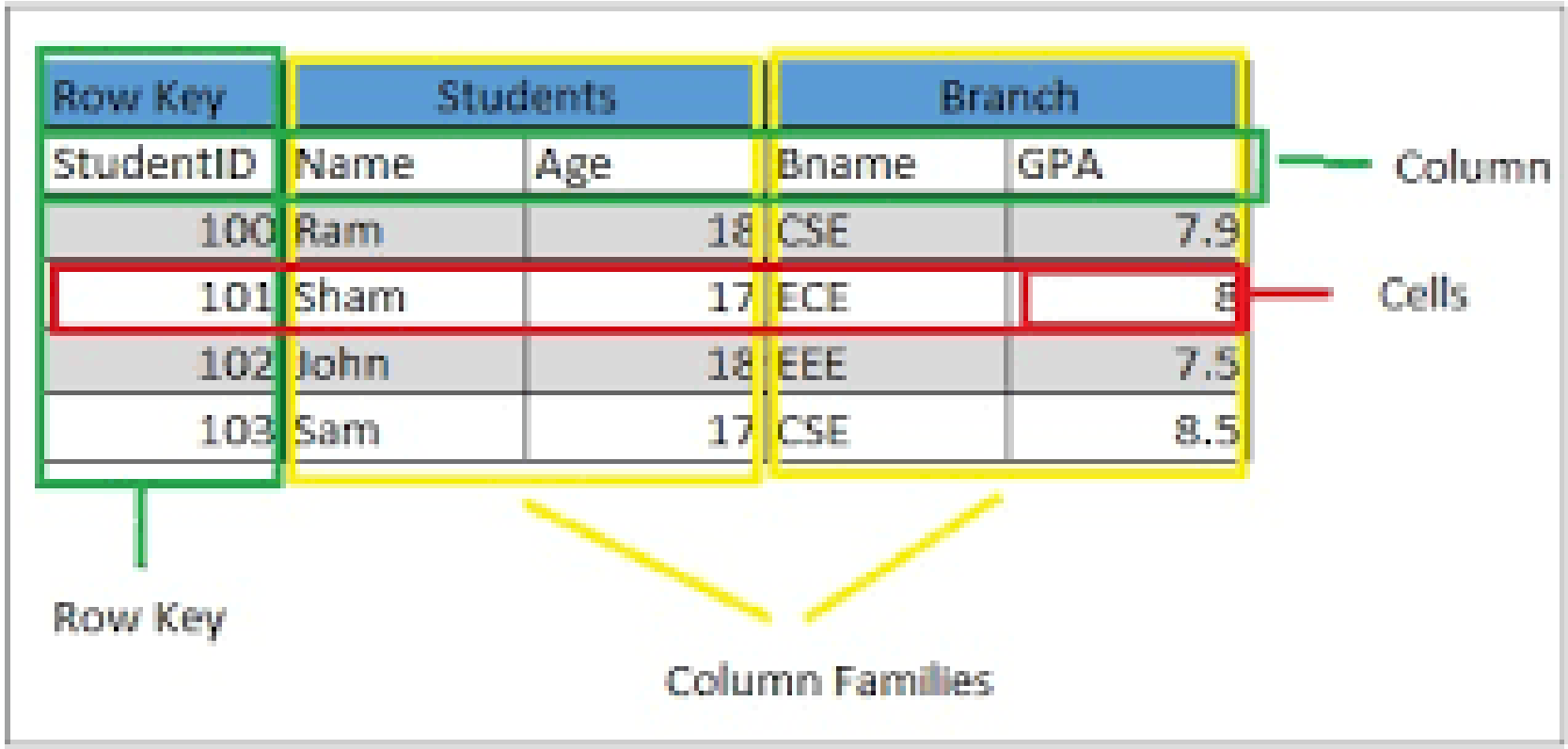
When numbers are stored as strings

1, 2, 10, 20

When stored using Bytes.toBytes(int) (fixed-width binary`)

Why Byte Arrays

- Flexibility: Any datatype can be encoded by the client.
- Performance: Raw byte comparison is extremely fast.
- No overhead: HBase doesn't interpret the value → client controls format.
- Efficient lexicographic sorting: Regions split easily based on these sorted byte arrays.


Row Key
Students
Branch
StudentID
Name
Age
Bname
GPA
100
Ram
18
CSE
7.9
101
Sham
17
ECE
8
102
John
18
EEE
7.5
103
Sam
17
CSE
8.5
Column
Cells
Row Key
Column Families

Column Family

- HBase groups columns into column families.
- CFs are fixed at schema definition; you cannot add/remove them frequently.
- Data in the same CF is stored together on disk (in the same HFile) – very important
- Every column belongs to a column family – cf:column

personal
orders
activity

The table schema defines column families, not individual columns.

Column Family

personal
orders
activity

The table schema defines column families, not individual columns.

Row: user_001
CF: personal → stored in one HFile
CF: orders → stored in another HFile
CF: activity → stored in another HFile

You can scan only one CF without touching data in others.

Families can have totally different sizes.

Column Family

You must define column families when creating the table.

create 'users', 'personal', 'orders', 'activity'

Column Qualifiers are dynamic

Column Families cannot be dynamically added often

Column Family

- **Reads:** All data in a CF is stored together → fast scan if you target only that CF.
- **Writes:** Each CF has its own: MemStore, Hfiles, BlockCache, Bloom filters
- Performance tuning is per-CF.
- **Compaction:** Compaction is done CF-wise.
  - Big CF = heavy compaction.

Column Family Settings

Setting	Meaning
TTL	Auto-delete old data after X seconds
Max versions	How many historical versions to keep
Compression	Snappy, GZip, LZ4 per CF
Block size	HFile block size (default 64 KB)
Bloom filters	Reduce disk seeks

Column Family Isolation

CF	Pattern	Why
personal	small, rarely updated	no compaction pressure
orders	large, frequently updated	separate compaction needed
activity	high-write time-series	can have short TTL

How many CFs

DO NOT make many column families

HBase best practice:

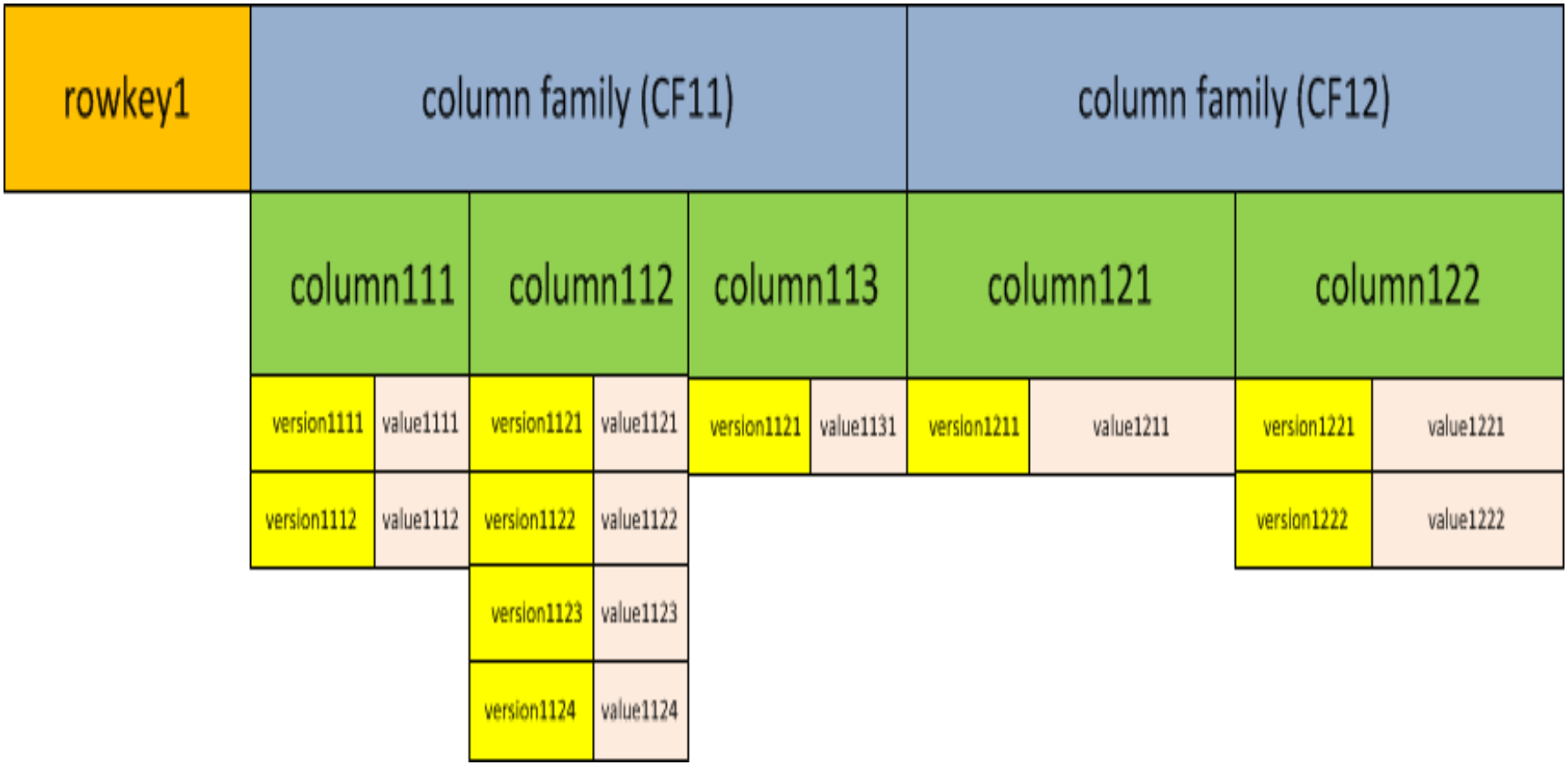
- **Maximum 2–3 CFs**
- Up to 5 possible (but risky)
- 1–2 is ideal

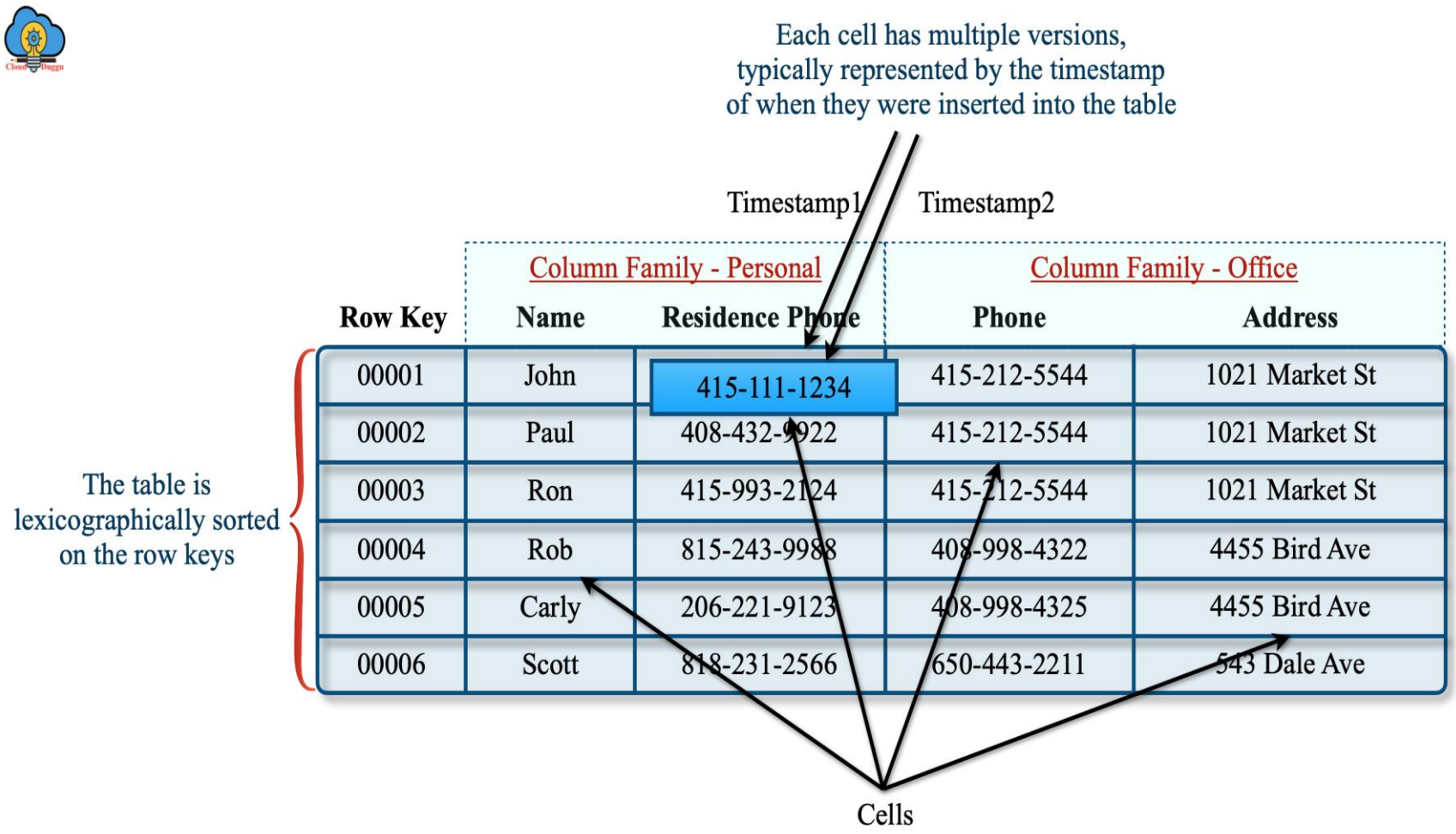
Each CF has:

- Separate MemStore
- Separate HFile set
- Separate WAL edits
- Separate flushes
- Separate compactions

With many CFs:

- RAM is wasted
- Region servers struggle
- Too many HFiles → compaction storms


rowkey1
column family (CF11)
column family (CF12)
column111
column112
column113
column121
column122
version1111
value1111
version1121
value1121
version1121
value1131
version1211
value1211
version1221
value1221
version1112
value1112
version1122
value1122
version1222
value1222
version1123
value1123
version1124
value1124


Each cell has multiple versions,
typically represented by the timestamp
of when they were inserted into the table
Timestamp1
Timestamp2
Column Family - Personal
Column Family - Office
Row Key
Name
Residence Phone
Phone
Address
00001
John
415-111-1234
415-212-5544
1021 Market St
00002
Paul
408-432-9922
415-212-5544
1021 Market St
00003
Ron
415-993-2124
415-212-5544
1021 Market St
00004
Rob
815-243-9988
408-998-4322
4455 Bird Ave
00005
Carly
206-221-9123
408-998-4325
4455 Bird Ave
00006
Scott
818-231-2566
650-443-2211
543 Dale Ave
The table is
lexicographically sorted
on the row keys
Cells

Column Qualifier

- Inside each column family, you can create unlimited dynamic columns called column qualifiers – it is just the column name
- These are NOT predefined; they can vary from row to row.

personal:name
personal:email
orders:2021-12-01
orders:2021-12-02

Gives HBase its schema-less flexibility within each family.

Column Qualifier

- Column qualifiers do NOT need to be defined in the schema.
- You can create new qualifiers at any time while inserting data.

```
put.addColumn(Bytes.toBytes("info"), Bytes.toBytes("new_column"),
Bytes.toBytes("value"));
```

No schema change needed!

! Unlimited columns
! Different rows can have different qualifiers
! HBase stores only what exists → sparse table

Column Qualifier

- Qualifiers Are Stored As Byte Arrays
- Just like row keys: Column qualifiers are stored internally as byte[] and are sorted lexicographically within the column family

"a", "ab", "abc", "b"

a
ab
abc
b

Column Qualifier

- Column Qualifiers Are Lightweight and Cheap
- Unlike column families (heavyweight), column qualifiers:
  - Require no server memory allocation
  - Do not create new HFiles
  - Do not create new MemStores
  - Do not affect WAL structure
  - Can be millions per row (common in time-series systems)
- So you can create qualifiers freely.

Column Qualifier

- Column Qualifiers Are Namespaced by the Column Family
- name in info has no relation to name in another CF.
- Qualifiers are only meaningful inside their family.

info:name
orders:2024-01-01
orders:2024-01-02

- A Single Row Can Have Millions of Qualifiers – time-stamped data can have that!

1700000000 → 74.5
1700000300 → 75.0
1700000600 → 74.8

Column Qualifier

- Qualifiers and Timestamps Form Unique Cells
- (row key, column family, column qualifier, timestamp) → value

Row Keys	Time_Stamp	Column 1 (C1)		Column 2 (C2)		
		C1:Col 1	C1:Col 2	C2:Col 3	C2:Col 4	C2:Col 5
Row1	Time stamp 1			Value 3	Value 4	Value 5
Row2	Time stamp 2	Value 6	Value 7	Value 8	Value 9	Value 10
Row3	Time stamp 3	Value 11	Value 12	Value 13		

Versions in HBase are *not* new rows.

They are stored inside the *same row*, under the *same row key*, but with different timestamps inside the same column family + qualifier.

Versions

- A “version” is identified by a **timestamp**:

(row key, column family, column qualifier, timestamp) → value

So, a **cell is not a single value**, it can be many values stored over time.

info:name @ ts=1700001000 → “Rasheed"
info:name @ ts=1700000000 → “Rasheed Ahmed"

Versions

- HBase stores multiple versions of each cell.
- Each value is stored with a timestamp.
- Latest version is returned by default.

(personal:name, ts=1699610000) → “BDA"
(personal:name, ts=1699500000) → “BDA Fall25 IBA"

Useful for time-based data, audit trails, and recovering old values

Versions

- HBase keeps **1 version** by default
- Meaning: only the latest timestamp is returned.
- You can change this in the CF definition:

```
alter 'users', NAME=>'info', VERSIONS=>3
```

- Now HBase stores the **3 latest versions** for each (row, cf, qual).

Versions

- **Server Timestamp :**If you do not supply a timestamp, HBase assigns **current system time** in milliseconds.
- **User Timestamp:** You can set your own timestamp:

```
put.addColumn(Bytes.toBytes("info"),
Bytes.toBytes("name"), 1234567890L, value);
```

Useful for:
Time-series data
CDC (change-data-capture)
Event logs
IoT measurements
Ordering values manually

Unix Epoch

The Unix epoch is:
00:00:00 (midnight) UTC on 1 January 1970

Unix-based systems count time as the number of seconds that have passed since 1 Jan 1970 (UTC).

Why 1970?
It was chosen by the early Unix designers as a convenient arbitrary starting date.

Examples
- `0` seconds = 1 Jan 1970, 00:00:00 UTC
- `1,000,000,000` seconds ≈ 9 Sep 2001
- Current timestamps like `1732435394` represent seconds since the epoch.

Versions

- Inside an HFile, values are stored sorted with most recent version first → fastest read.

  (row, cf, qualifier)
  ↓
  sorted by timestamp (descending)

- Analytics: "What was the value at time X?"
- Audit logs: "How did this record change?"
- Time-series: each timestamp = qualifier OR version
- Data recovery: read older values
- Machine learning: track drift over time
- **Versioning = built-in append-only log :**No need for separate logging tables.

Cell

- A cell is the smallest unit of data:

1 cell = (row key, column family, column qualifier, timestamp) → value

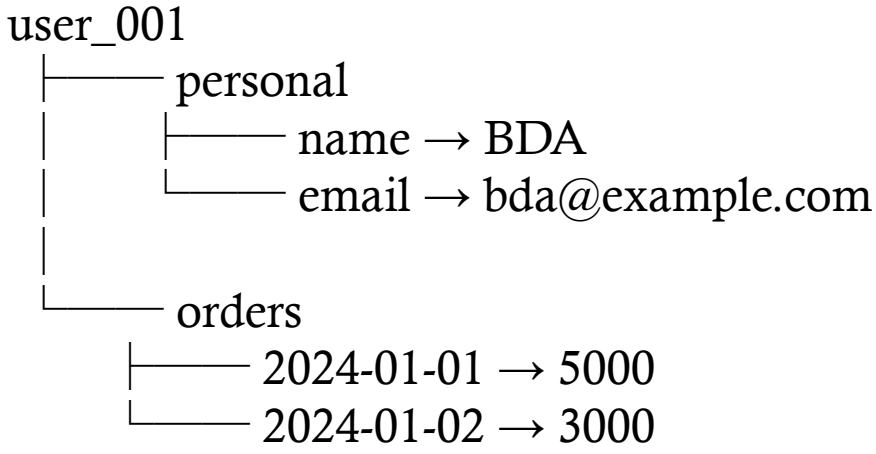
Row: user_001
Column: personal:name
Timestamp: 1699610000
Value: “BDA"

Useful for time-based data, audit trails, and recovering old values

HBase Internal Data Storage

How Data Stored Internally

- **RowKey → ColumnFamily → ColumnQualifier → Timestamp → Value**



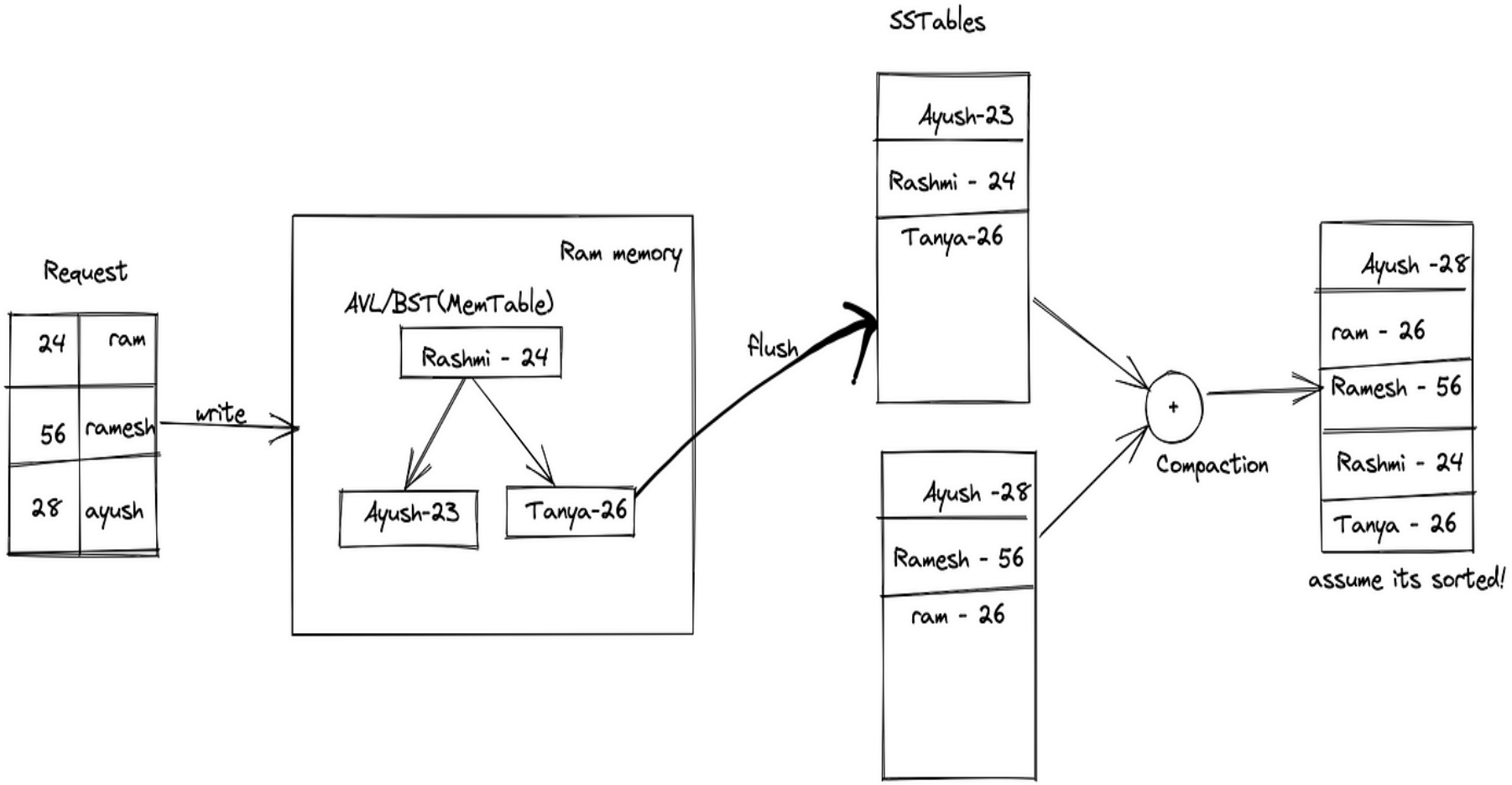
LSM Tree

- Log-Structured Merge Tree: disk-based data structure optimized for high write throughput.
- In-memory component (MemStore / MemTable)
  - All writes first go to RAM
  - They are also appended to WAL for durability
  - These writes stay sorted in memory
  - When the MemStore becomes full, it is flushed to disk as an immutable file.

LSM Tree

- On-disk component (SSTables / HFiles)
  - Flushed data becomes sorted immutable files on disk.
  - Over time, many such files accumulate.
  - System periodically merges and compacts them to reduce #files and delete overwritten/expired data.
- Why is it fast?
  - Writes are sequential (append to WAL, flush to disk).
  - Disk seeks are minimized.
  - Read efficiency is maintained using indexes, Bloom filters, and compactions.

LSMT


SSTables
Request
24 ram
56 ramesh
28 ayush
write
Ram memory
AVL/BST(MemTable)
Rashmi - 24
Ayush-23
Tanya-26
flush
Ayush-23
Rashmi - 24
Tanya-26
Ayush -28
Ramesh - 56
ram - 26
+
Compaction
Ayush -28
ram - 26
Ramesh - 56
Rashmi - 24
Tanya - 26
assume its sorted!

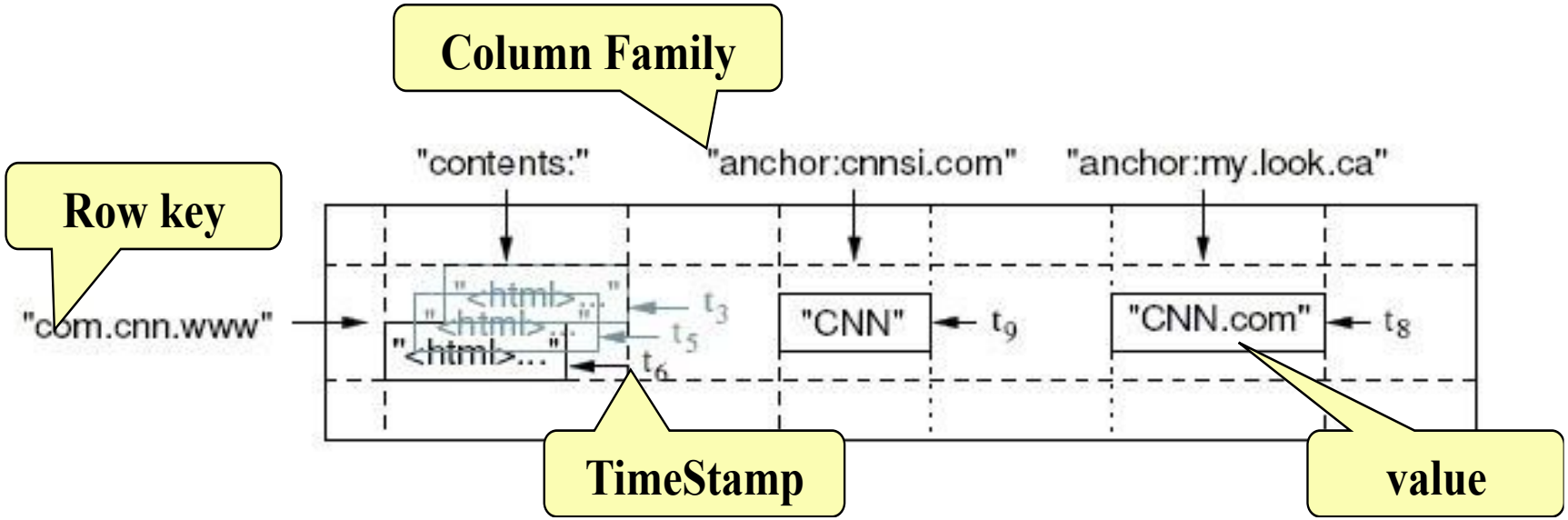
Major Benefits

- **Sparse:** Rows can have different columns without storage penalty.
- **Wide:** A row can have millions of columns (common for time-series).
- **Flexible schema:** Column qualifiers are dynamic.
- **High write throughput:** LSM-tree architecture is optimized for writes.
- **Scalable:** Tables automatically split into regions and spread across region servers.

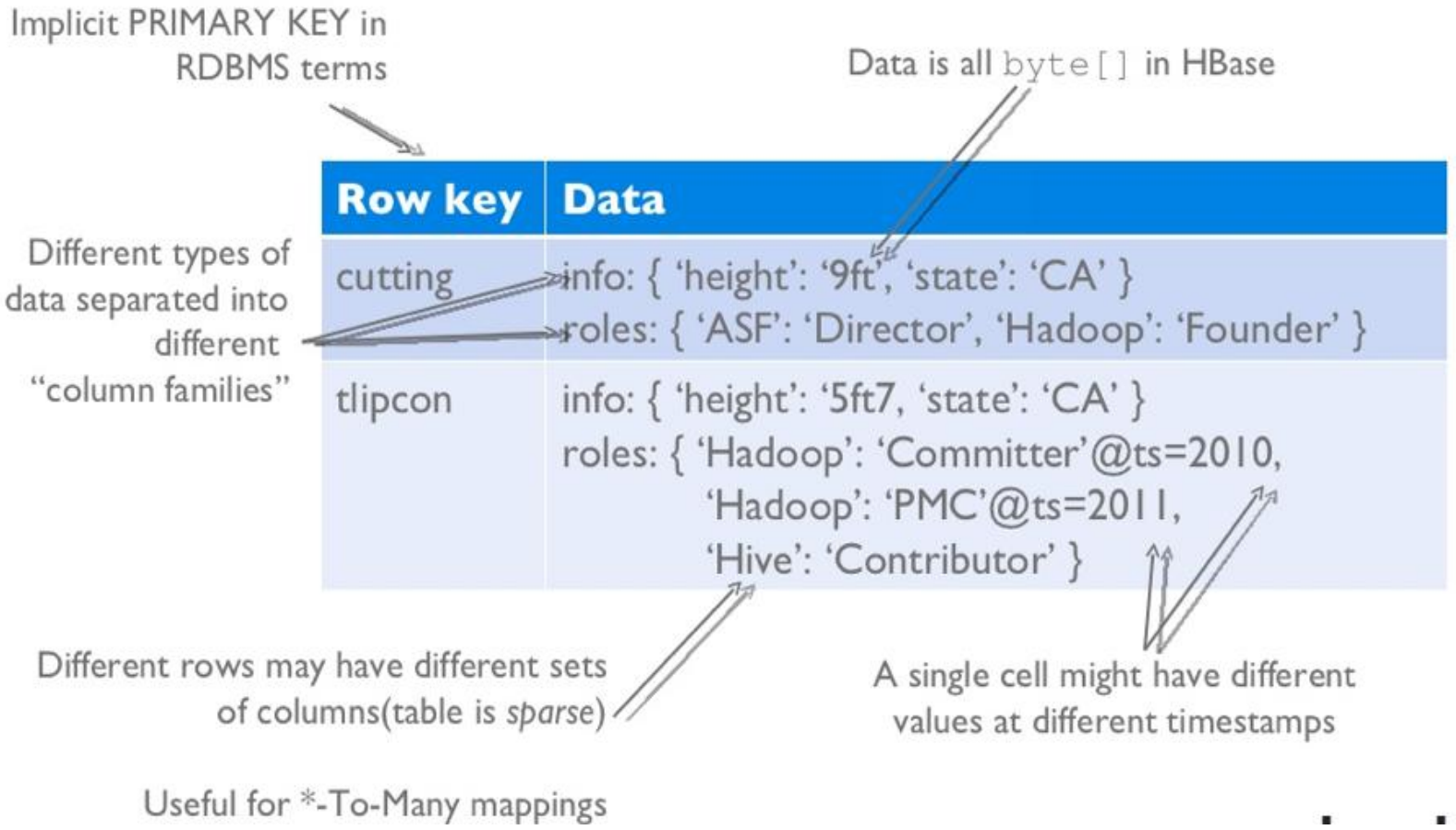
Back to Data Model

HBase Data Model

- **HBase is based on Google's Bigtable model**
  - **Key-Value pairs**



HBase Logical View



HBase: Keys and Column Families

Each record is divided into *Column Families*

Each row has a *Key*

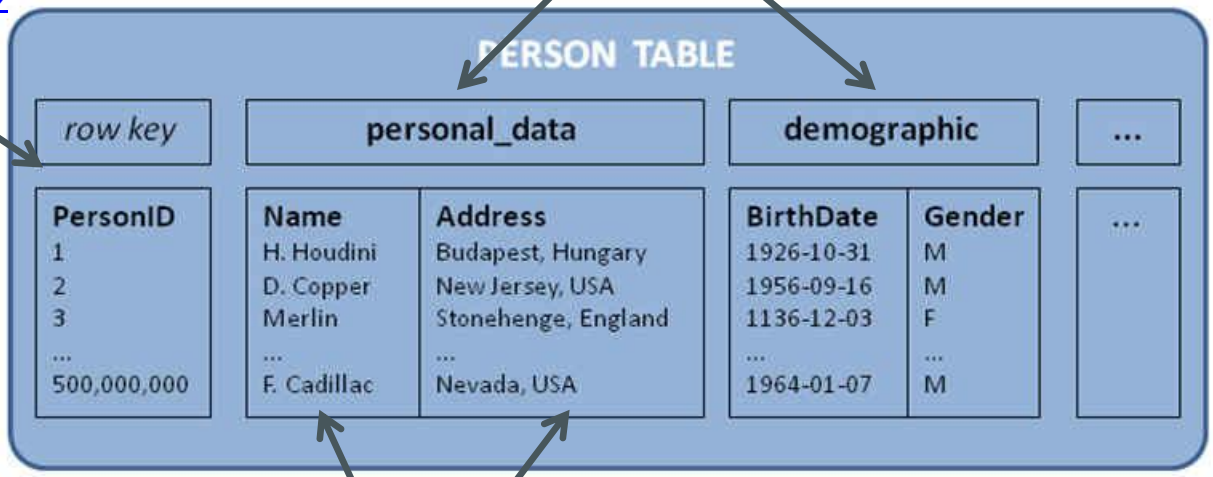


Figure 2 Census Data in Column Families

Each column family consists of one or more *Columns*

Column family named "anchor"

Column family named "Contents"

Column named "apache.com"

Row key	Time Stamp	Column "contents:"	Column "anchor:"	
"com.apache.www"	t12	"<html>…"		
	t11	"<html>…"		
	t10		"anchor:apache.com"	"APACHE"
"com.cnn.www"	t15		"anchor:cnnsi.com"	"CNN"
	t13		"anchor:my.look.ca"	"CNN.com"
	t6	"<html>…"		
	t5	"<html>…"		
	t3	"<html>…"		

Version number for each row

Row key	Time Stamp	Column "contents:"	Column "anchor:"	
"com.apache.www"	t12	"<html>…"		value
	t11	"<html>…"		
	t10		"anchor:apache.com"	"APACHE"
"com.cnn.www"	t15		"anchor:cnnsi.com"	"CNN"
	t13		"anchor:my.look.ca"	"CNN.com"
	t6	"<html>…"		
	t5	"<html>…"		
	t3	"<html>…"		

Notes on Data Model

- HBase schema consists of several ***Tables***
- Each table consists of a set of ***Column Families***
  - Columns are not part of the schema
- HBase has ***Dynamic Columns (don't predefine them – create on the fly)***
  - Because column names are encoded inside the cells
  - Different cells can have different columns

"Roles" column family has different columns in different cells

Notes on Data Model (Cont'd)

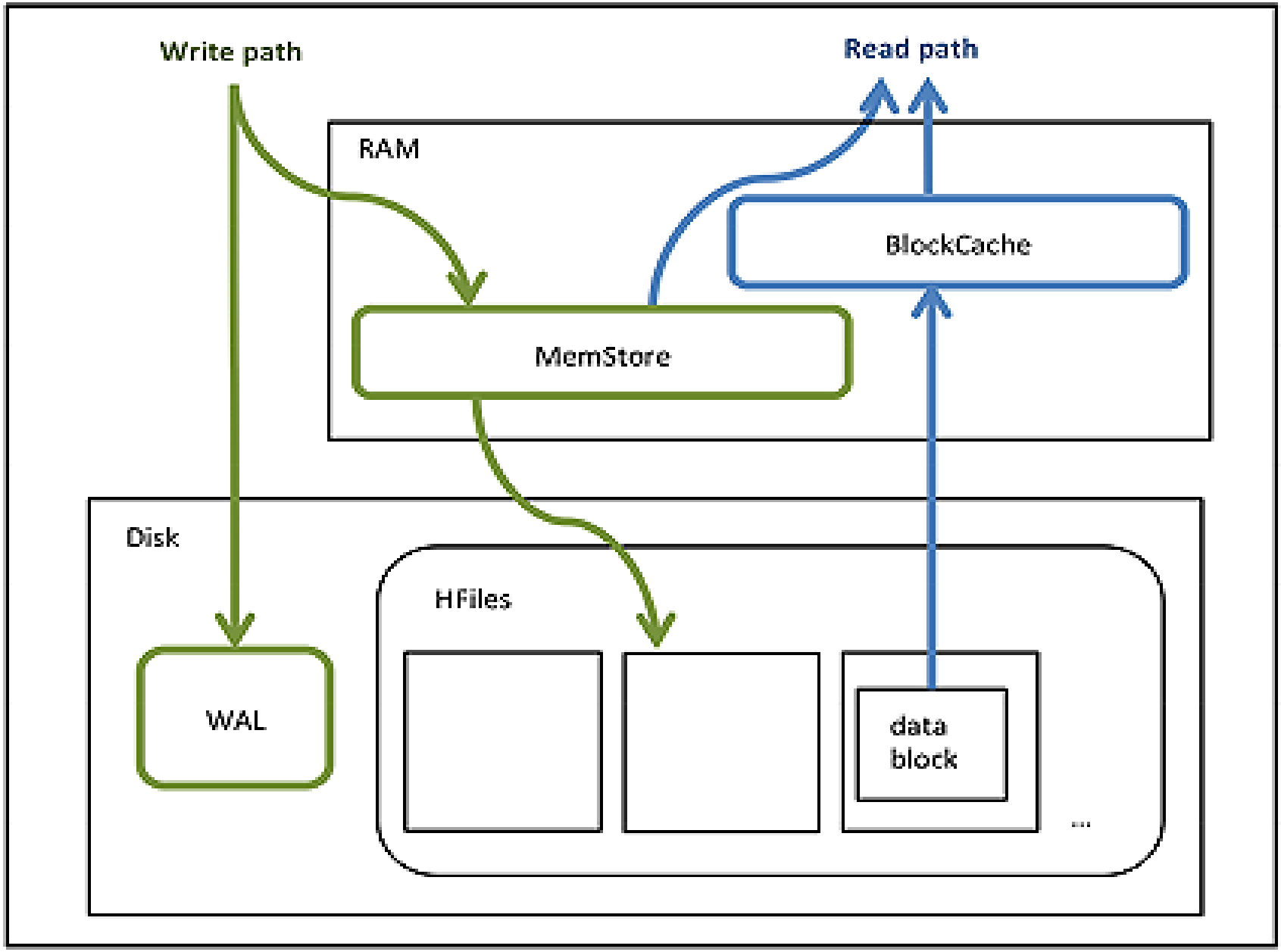
- The ***version number*** can be user-supplied
  - Even does not have to be inserted in increasing order
  - Version number are unique within each key
- Table can be very sparse
  - Many cells are empty
- ***Keys*** are indexed as the primary key

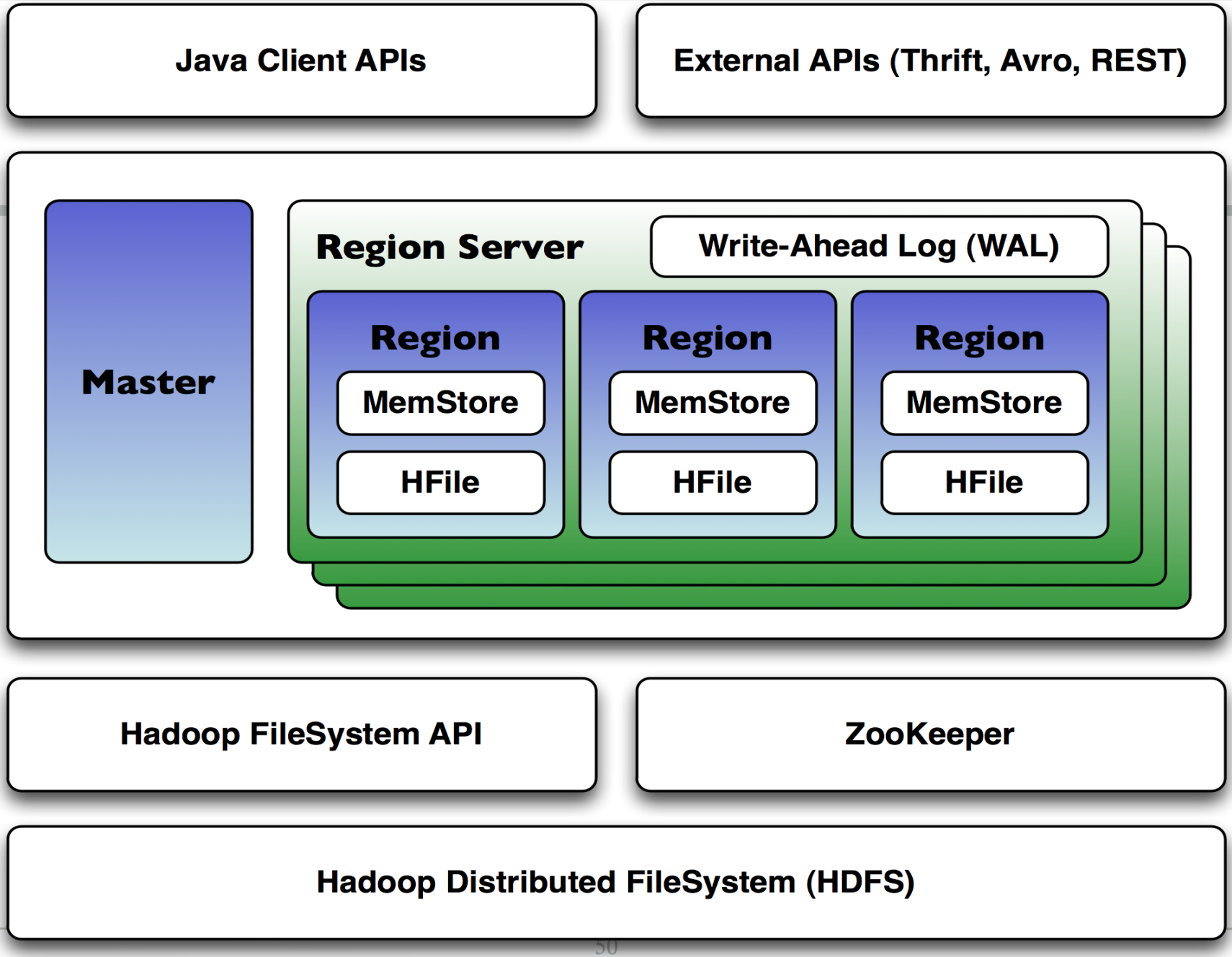
Has two columns
[cnnsi.com & my.look.ca]

Row Key	Time Stamp	ColumnFamily `contents`	ColumnFamily `anchor`
"com.cnn.www"	t9		`anchor:cnnsi.com` = "CNN"
"com.cnn.www"	t8		`anchor:my.look.ca` = "CNN.com"
"com.cnn.www"	t6	`contents:html` = "<html>..."	
"com.cnn.www"	t5	`contents:html` = "<html>..."	
"com.cnn.www"	t3	`contents:html` = "<html>..."	

HBase Physical Model

What files exist, how they are organized, and how rows/columns are stored.


Write path
Read path
RAM
BlockCache
MemStore
Disk
HFiles
WAL
data
block
...


Java Client APIs
External APIs (Thrift, Avro, REST)
Master
Region Server
Write-Ahead Log (WAL)
Region
MemStore
HFile
Region
MemStore
HFile
Region
MemStore
HFile
Hadoop FileSystem API
ZooKeeper
Hadoop Distributed FileSystem (HDFS)

Physical Model

- HBase Table → Regions → RegionServers → Hfiles
- Table: A logical table becomes multiple regions, each stored on some RegionServer.
- Regions: A region is a horizontal partition of a table, based on row key range.
  - Region 1: rows from “A” to “H”
  - Region 2: rows from “H” to “Q”
- Physically, each region has:
  - MemStore (in RAM)
  - Store (on disk)
  - HFiles
  - WAL (Write-Ahead Log)

Physical Model

- RegionServer
  - A RegionServer hosts many regions.
  - Each region has 1 or more stores (one store per column family).

Physical Model

- Column Families → Stores → MemStore + Hfiles
- **Column Family = Unit of Storage**
- For each column family, HBase creates a **Store** consisting of:
  - **MemStore** (in-memory)
  - **HFiles** (on HDFS)
- **Important:** All columns inside a CF (family) are stored together in the same physical files.

Physical Model

- **MemStore (In RAM)**
- When you write data:
  - Write goes to **WAL** (for crash recovery)
  - Data goes to **MemStore**
- When MemStore is full → data is flushed to disk into a new **HFile**

Physical Model

- MemStore = in-memory write buffer for each column family in a region.
- Every column family in a region has its own MemStore.
- Acts as a staging area for writes before flushing to disk (HFiles)
- Works together with WAL for durability.

Physical Model

- How Writes Go Through MemStore
- When a client does a Put:
  - Write goes to **WAL** (for crash recovery)
  - Write goes to **MemStore** (in RAM)
  - **ACK** is sent to client
  - When MemStore reaches a **threshold size**, data is **flushed to HFiles** on HDFS

Physical Model

- Example Column family cf1 in region r1: MemStore size threshold = 128 MB
- All writes stored in MemStore in sorted order
- When MemStore > 128 MB → flush → HFile created on HDFS
- MemStore cleared, starts accumulating new writes

Put row1 cf1:name = "Ali"
Put row2 cf1:email = "ali@example.com"
Put row3 cf1:age = 30

Physical Model

- **HFiles (on HDFS):** This is the actual physical file on disk.
- HFile =
  - Sorted by **row key**
  - Within row key, sorted by **column family**
  - Within CF, sorted by **column qualifier**
  - Then sorted by **timestamp**

RowKey →
ColumnFamily →
ColumnQualifier →
Timestamp (versioned cells)

Physical Model

- **WAL (Write-Ahead Log)**
  - Every write goes to WAL before hitting MemStore.
  - Stored in HDFS as HLog files.
  - Used to replay writes if RegionServer crashes.
- A file where every write (Put/Delete) is **first** written before the data actually goes to MemStore / disk.
- Purpose:
  - Prevent data loss
  - Recover after RegionServer crash

Physical Model

```
Put p = new Put(Bytes.toBytes("row1"));
p.addColumn(Bytes.toBytes("cf1"), Bytes.toBytes("name"),
Bytes.toBytes("Ali"));
table.put(p);
```

WAL:
[Timestamp] regionID: cf1:name row=row1 value="Ali"

MemStore:
row1 -> { cf1:name = "Ali" }`

- Client sends Put to RegionServer.
- RegionServer appends the mutation to WAL
- Only after WAL is safely written, the data goes into MemStore
- ACK is sent back to client.

Physical Model

- You inserted 1000 records. They are still in MemStore, not flushed to HFiles. Suddenly RegionServer crashes. Without WAL → data lost.
- With WAL → Recovery happens:
  - Master assigns region to another RegionServer.
  - New RS reads the WAL file associated with that region.
  - WAL replay
  - Rebuild MemStore → flush → HFiles.
  - All data restored.

Replay:
Apply cf1:name row=row1 value=Ali
Apply cf1:age row=row1 value=23
Apply cf1:name row=row2 value=Zara
…

HBase Physical Model

- Hey! Each Region of a table is divided by column families. Each column family inside that region has multiple HFiles. These HFiles store the actual data.
- Key & Version numbers are replicated with each column family
- Empty cells are not stored

HBase maintains a multi-level index on values: *<key, column family, column name, timestamp>*

Table 5.3. ColumnFamily `contents`

Row Key	Time Stamp	ColumnFamily "contents:"
"com.cnn.www"	t6	`contents:html` = "<html>..."
"com.cnn.www"	t5	`contents:html` = "<html>..."
"com.cnn.www"	t3	`contents:html` = "<html>..."

Table 5.2. ColumnFamily `anchor`

Row Key	Time Stamp	Column Family `anchor`
"com.cnn.www"	t9	`anchor:cnnsi.com` = "CNN"
"com.cnn.www"	t8	`anchor:my.look.ca` = "CNN.com"

Example

Row key	Data
cutting	info: { 'height': '9ft', 'state': 'CA' } roles: { 'ASF': 'Director', 'Hadoop': 'Founder' }
tlipcon	info: { 'height': '5ft7, 'state': 'CA' } roles: { 'Hadoop': 'Committer'@ts=2010, 'Hadoop': 'PMC'@ts=2011, 'Hive': 'Contributor' }

`info` Column Family

Row key	Column key	Timestamp	Cell value
cutting	info:height	1273516197868	9ft
cutting	info:state	1043871824184	CA
tlipcon	info:height	1273878447049	5ft7
tlipcon	info:state	1273616297446	CA

`roles` Column Family

Row key	Column key	Timestamp	Cell value
cutting	roles:ASF	1273871823022	Director
cutting	roles:Hadoop	1183746289103	Founder
tlipcon	roles:Hadoop	1300062064923	PMC
tlipcon	roles:Hadoop	1293388212294	Committer
tlipcon	roles:Hive	1273616297446	Contributor

Sorted on disk by Row key, Col key, descending timestamp

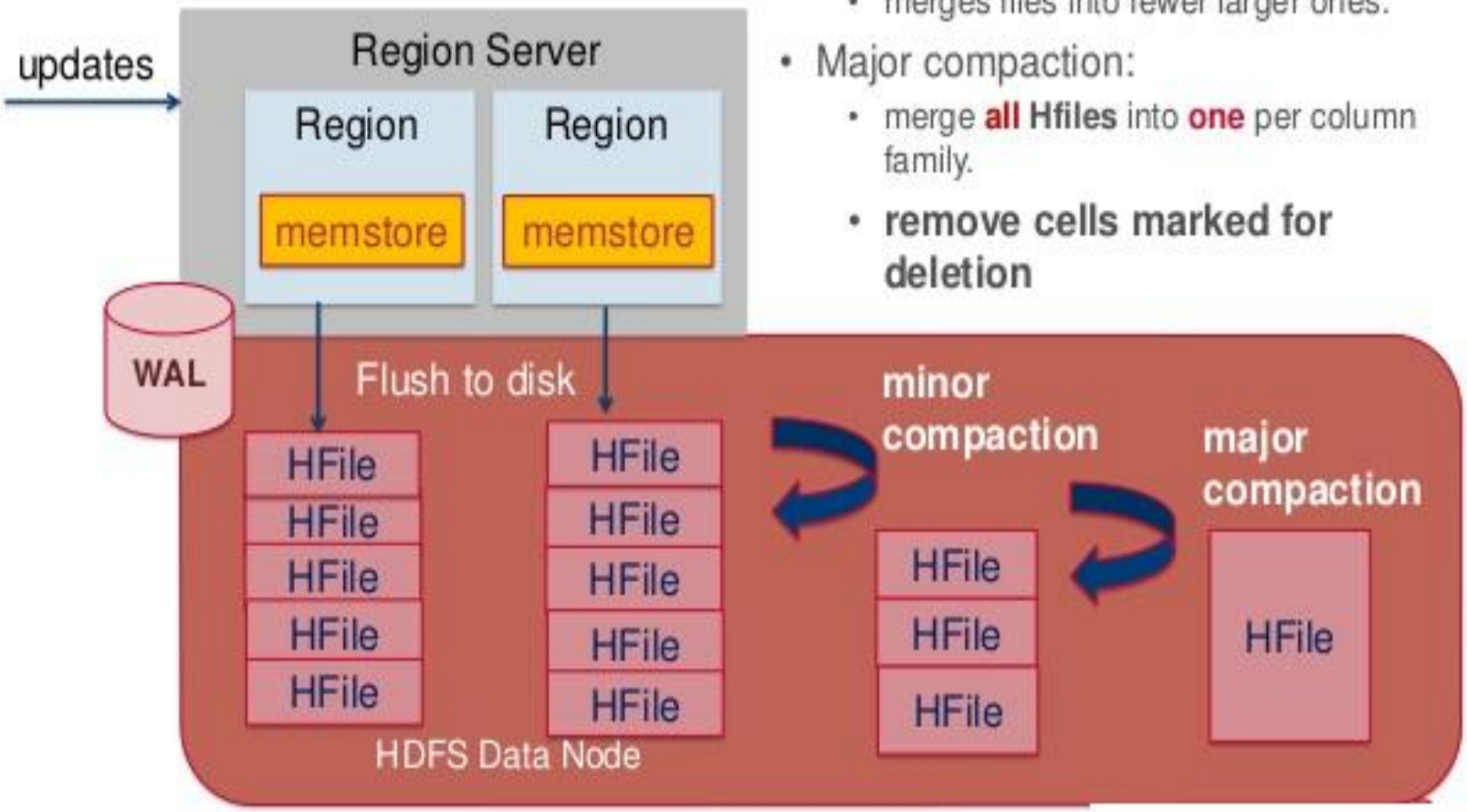
Milliseconds since unix epoch

cloudera

HBase Compaction

HBase Compaction

- **minor compaction:**
  - merges files into fewer larger ones.
- Major compaction:
  - merge **all Hfiles** into **one** per column family.
  - **remove cells marked for deletion**



Compaction: Motivation

- **Frequent small writes** → many HFiles in a column family
- **Reads** need to check **all HFiles** → slow
- **Deleted or old versions** still occupy space
- Compaction: **merging, cleaning, and re-sorting HFiles**.

Compaction

- Compaction = process of merging multiple HFiles into fewer, larger HFiles.
- Purpose:
  - Reduce the number of HFiles → faster reads
  - Delete expired/deleted versions → reclaim space
  - Organize sorted data efficiently on disk
  - HBase is write-optimized, not read-optimized.
  - Without compaction, frequent writes create many small HFiles, which slow down reads.

Compaction

HFiles before minor compaction: c1, c2, c3, c4
HFiles after minor compaction: merged_c1c2c3, c4

Merges a small number of HFiles into a larger Hfile

Keeps all old versions

Reduces the number of small files

Triggered automatically when the number of HFiles exceeds a threshold (default: 3)

Compaction

HFiles before major compaction: c1, c2, c3, c4
HFiles after major compaction: major_merged_c1c2c3c4

Major compaction is heavier → may impact performance

Usually done once per day or triggered manually

Removes deleted cells (tombstones)

Removes Expired versions (older than VERSIONS or TTL)

Creates a single large HFile per CF (per region)

HBase Compression

Column Families

- Different sets of columns may have different properties and access patterns
- Configurable by column family:
  - Compression (none, gzip, LZO)
  - Version retention policies
  - Cache priority
- CFs stored separately on disk: access one without wasting IO on the other.

!

Compression: Reducing disk usage and I/O by storing data in a compressed format inside HFiles.

- Examples: LZO, LZ4, Snappy, GZIP
- Applied per Column Family
- Done when writing HFiles (MemStore flush → HFile)

Important: Compression does not merge files or remove old versions — it just shrinks the size of the file.

Codec	Compression Ratio	Write Speed	Read Speed	CPU Usage	Notes / Use Case
LZO	Moderate (~30–50%)	Very fast	Very fast	Low	**write-heavy, low-latency workloads**.
LZ4	Moderate (~30–50%)	Very fast	Very fast	Low	Faster than LZO. **high-throughput, low-latency workloads**.
Snappy	Moderate (~30–50%)	Very fast	Very fast	Low	Google's codec. **write-heavy tables**.
GZIP / ZIP	High (~60–80%)	Slow	Slow	High	Pure Java **read-heavy, archival, or storage-optimized tables**.

Cache Priority

- Block Cache: stores data blocks that are read from Hfiles (64 KB) – hot data (read/expected to be read frequently) stays here – cold is evicted
- Mem Store: caches data that has been written but not yet flushed to disk – priority is about balancing write and flush – flush size is controlled (how much data stays)
- Change:
  - Block size
  - Eviction policy: LRU (least recently used): when the cache is full, the least recently accessed blocks are the first to be evicted

HBase Regions

- Each column family is partitioned horizontally into ***regions***
  - Regions are counterpart to HDFS blocks

Table 5.3. ColumnFamily `contents`

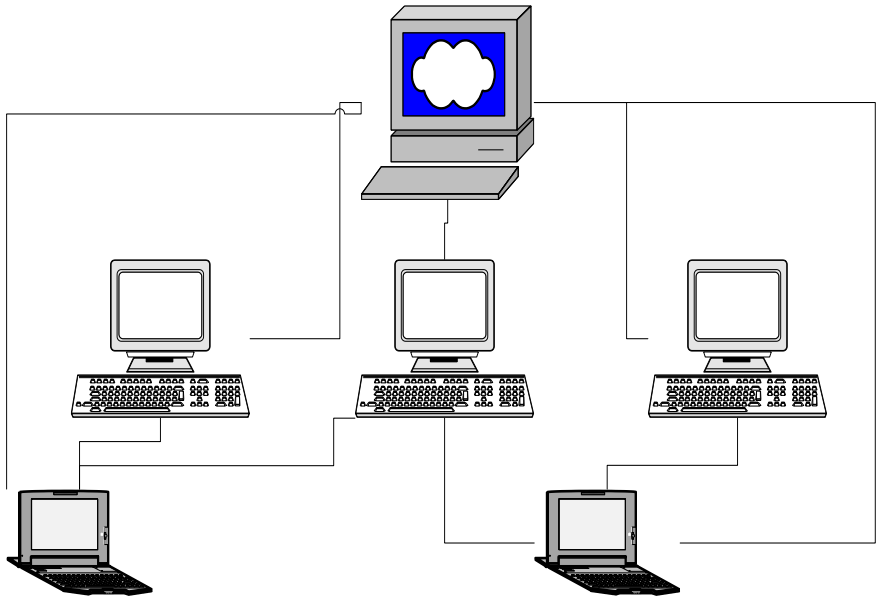
Row Key	Time Stamp	ColumnFamily "contents:"
"com.cnn.www"	t6	contents:html = "<html>..."
"com.cnn.www"	t5	contents:html = "<html>..."
"com.cnn.www"	t3	contents:html = "<html>..."

Each will be one region

HBase Architecture

Three Major Components

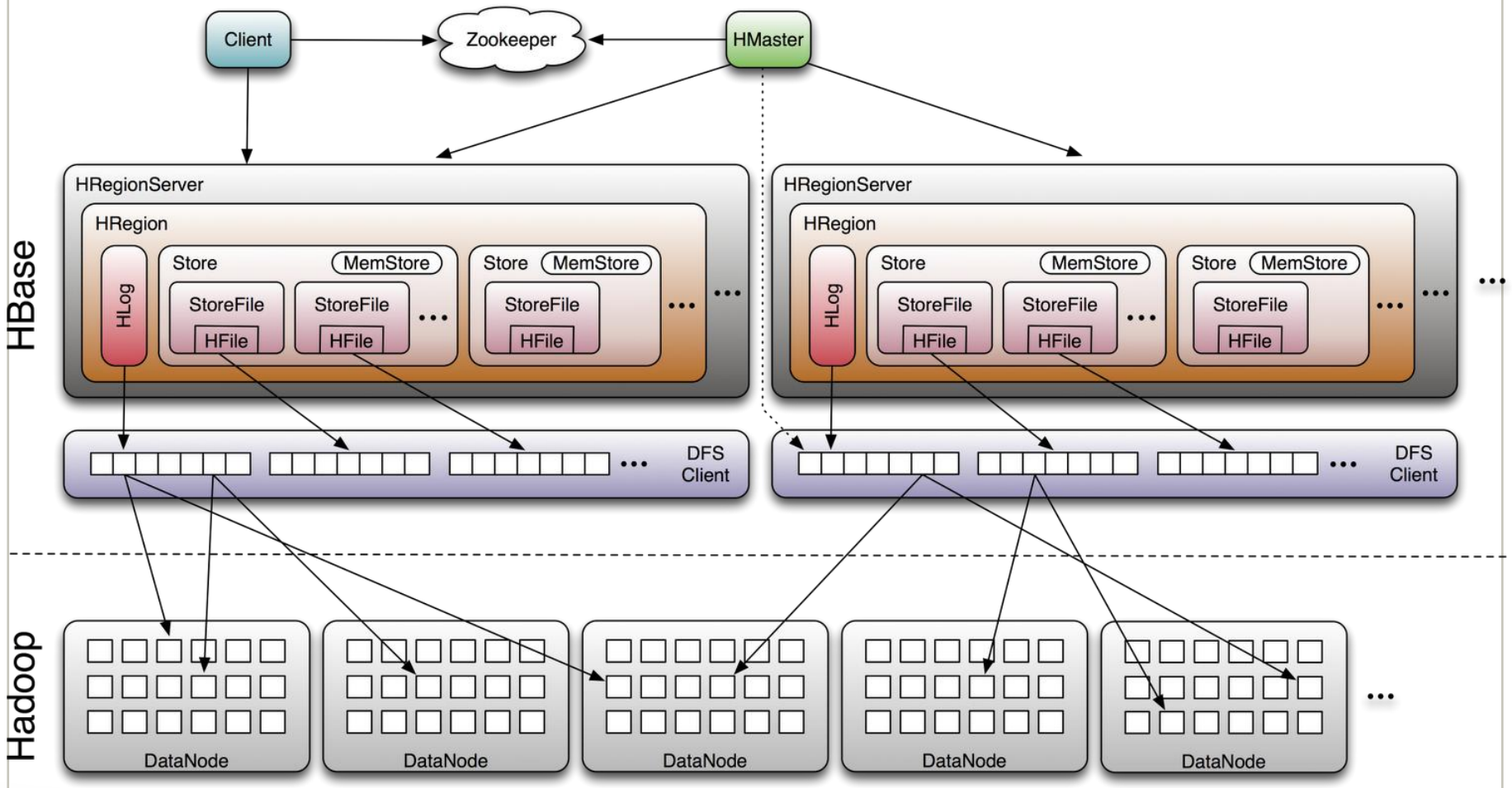
- The HBaseMaster
  - One master
- The HRegionServer
  - Many region servers
- The HBase client



HBase Components

- **Region**
  - A subset of a table's rows, like horizontal range partitioning
  - Automatically done
- **RegionServer (many slaves)**
  - Manages data regions
  - Serves data for reads and writes (***using a log***)
- **Master**
  - Responsible for coordinating the slaves
  - Assigns regions, detects failures
  - Admin functions

Big Picture


Client
Zookeeper
HMaster
HBase
HRegionServer
HRegion
HLog
Store
MemStore
StoreFile
HFile
DFS Client
Hadoop
DataNode

HBase vs. HDFS

- Both are distributed systems that scale to hundreds or thousands of nodes

- ***HDFS*** is good for batch processing (scans over big files)
  - Not good for record lookup
  - Not good for incremental addition of small batches
  - Not good for updates

HBase

- HBase is excellent for:
  - Fast random reads by row key
  - Millisecond-level lookup when you know the exact row key
  - High throughput for large datasets
- HBase is not good for:
  - Ad-hoc lookups (e.g., search by non-key columns)
  - Secondary indexes (not built-in like RDBMS)
  - Using scans over large ranges for random queries
    → these become slow and expensive.

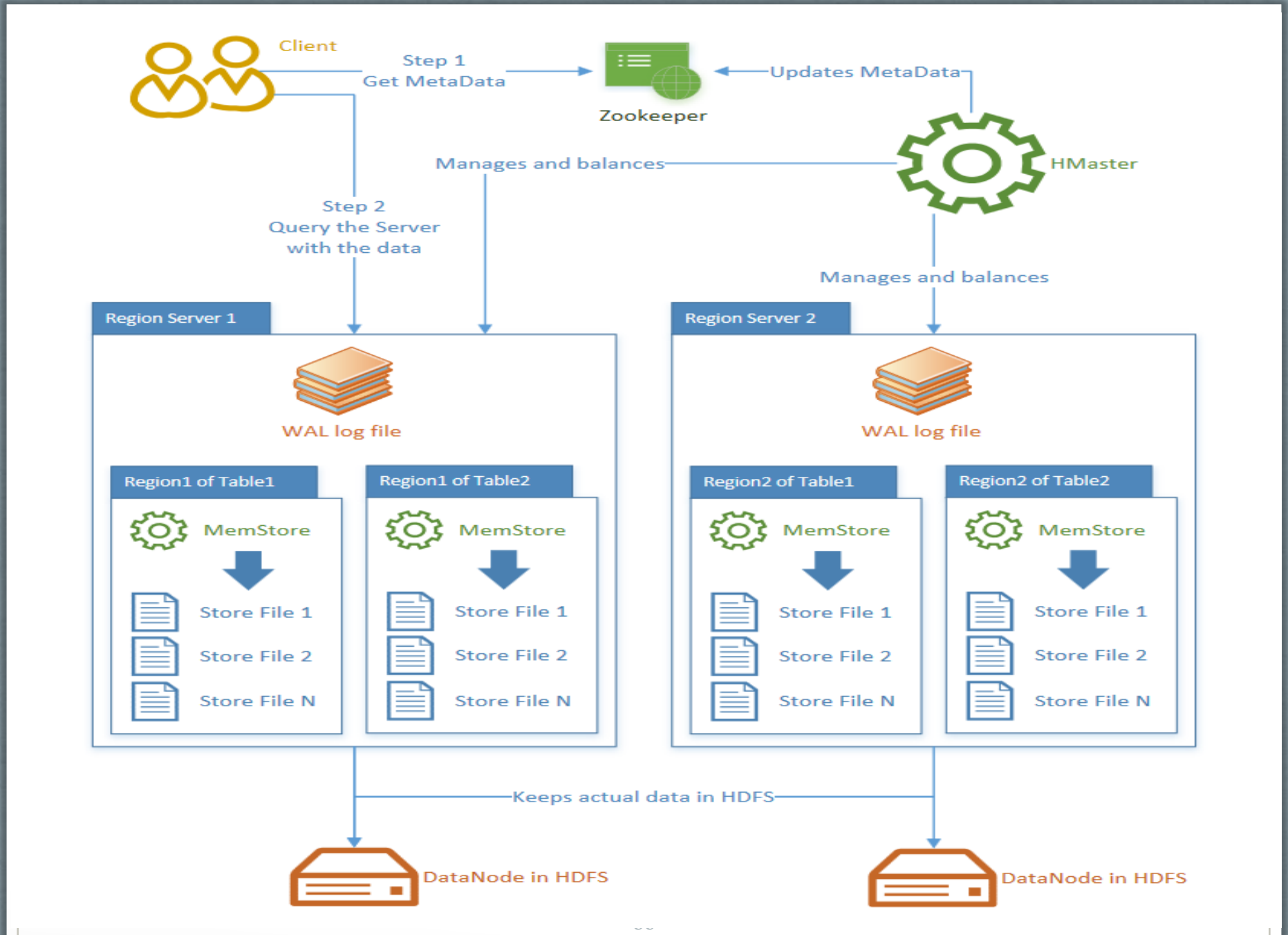
HBase is good for key-value pattern lookups, not for "query anything anywhere" lookup.

HBase

- HBase performance strongly benefits from **big sequential writes** (Bulk loads, streaming large volumes)
- **Why small writes are problematic:**
  - Each small write = RPC overhead
  - Frequent MemStore flushes → I/O pressure
  - Lots of tiny HFiles → compaction overhead

Buffer small updates and write in bulk batches.

HBase supports updates, but because they are *append-only writes*, the cost appears later during compaction.


Client
Step 1
Get MetaData
Zookeeper
Updates MetaData
HMaster
Manages and balances
Step 2
Query the Server
with the data
Manages and balances
Region Server 1
WAL log file
Region1 of Table1
MemStore
Store File 1
Store File 2
Store File N
Region1 of Table2
MemStore
Store File 1
Store File 2
Store File N
Region Server 2
WAL log file
Region2 of Table1
MemStore
Store File 1
Store File 2
Store File N
Region2 of Table2
MemStore
Store File 1
Store File 2
Store File N
Keeps actual data in HDFS
DataNode in HDFS
DataNode in HDFS

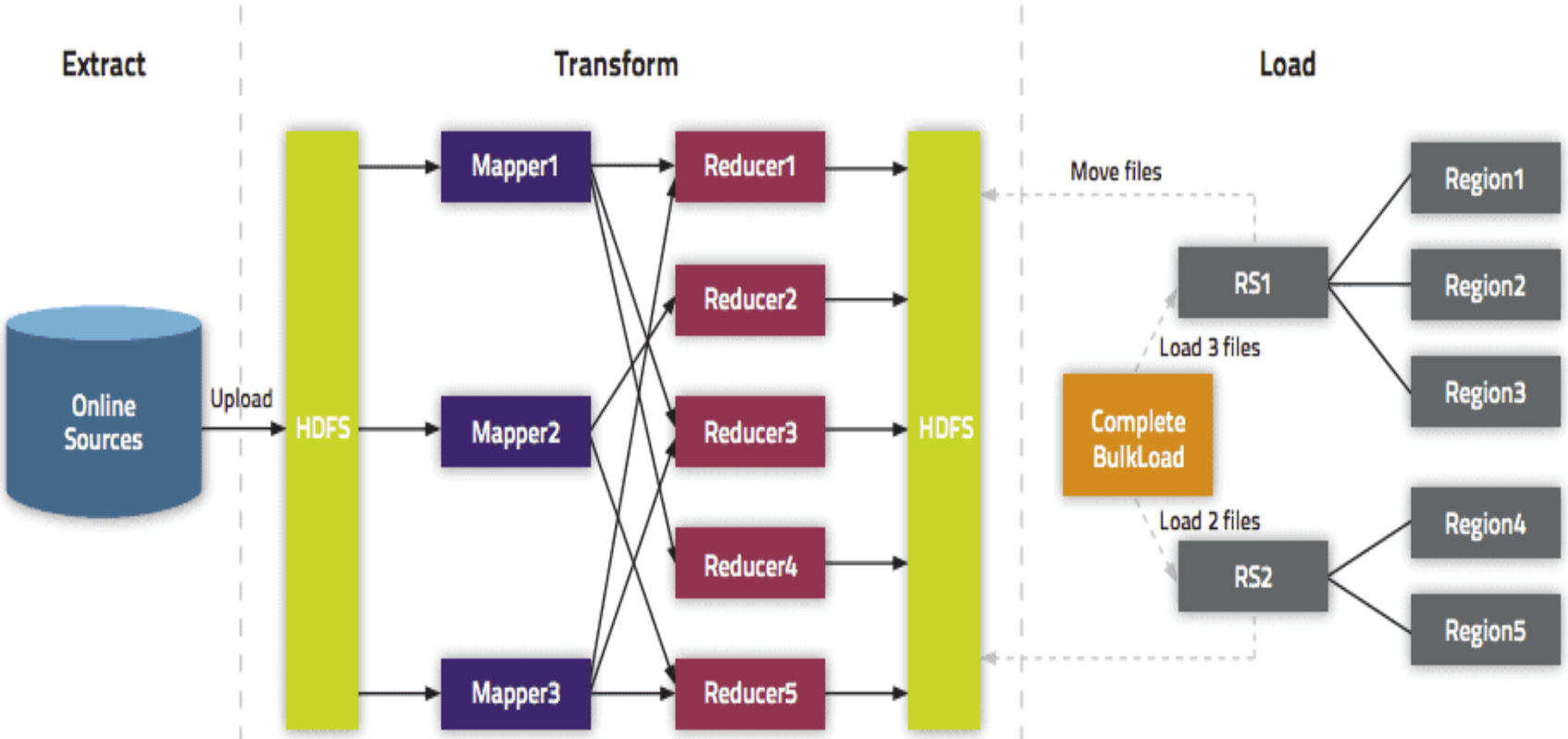
HBase Write Strategy

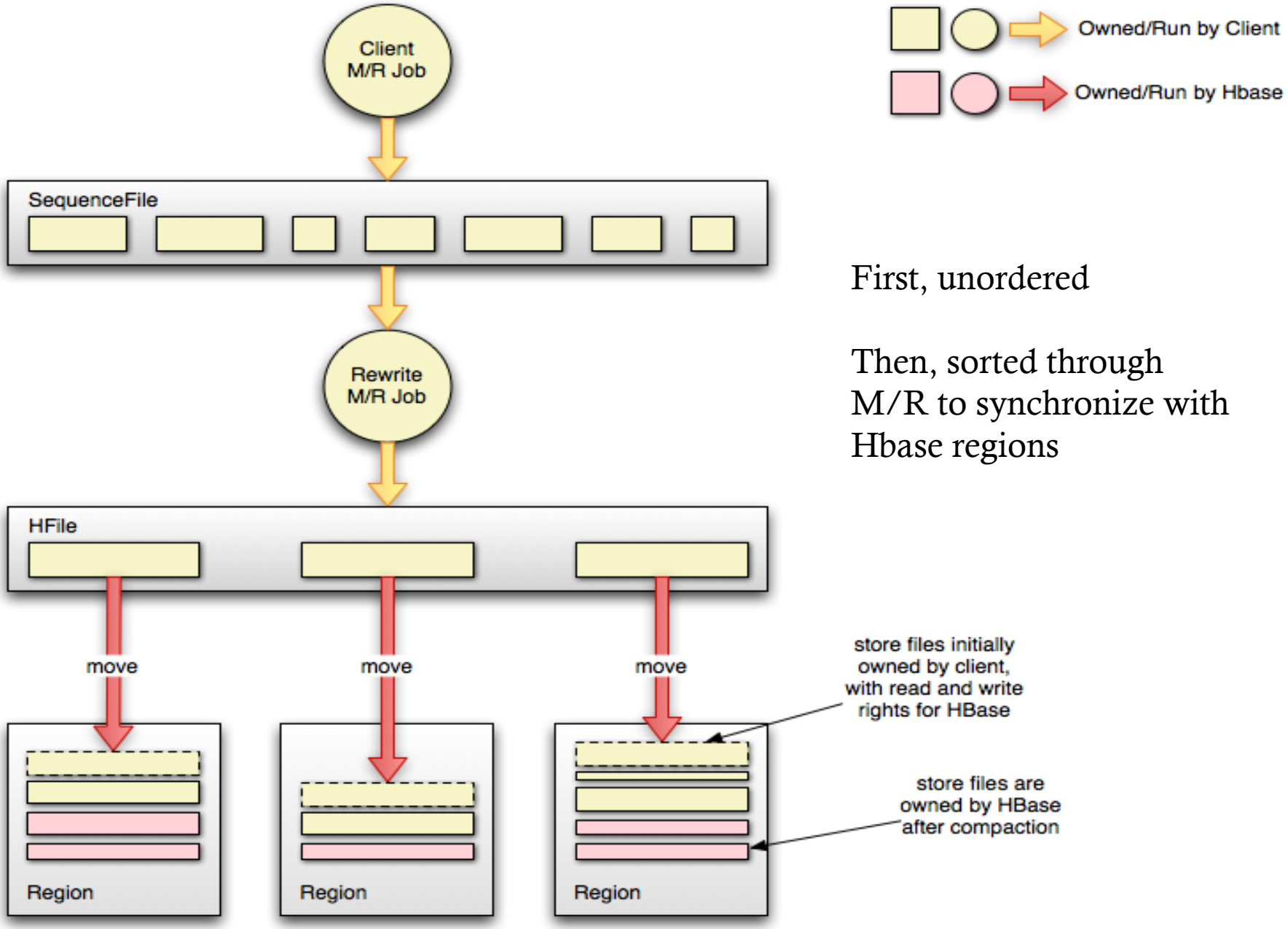
- Random Writes:
  - Well-suited for random writes – b/c architecture based on **LSM-trees**.
  - Data written first **MemStore** - periodically flushed to **HFiles**.
  - Fast, distributed, and consistent random writes

HBase Write Strategy

- Bulk Load:
  - For high-throughput ingestion
  - Data prepared and sorted into HFiles offline
  - Then HFiles are directly loaded into HBase regions.
  - Good to import large datasets quickly (ETL pipelines)
  - Efficient

Bulk Loading



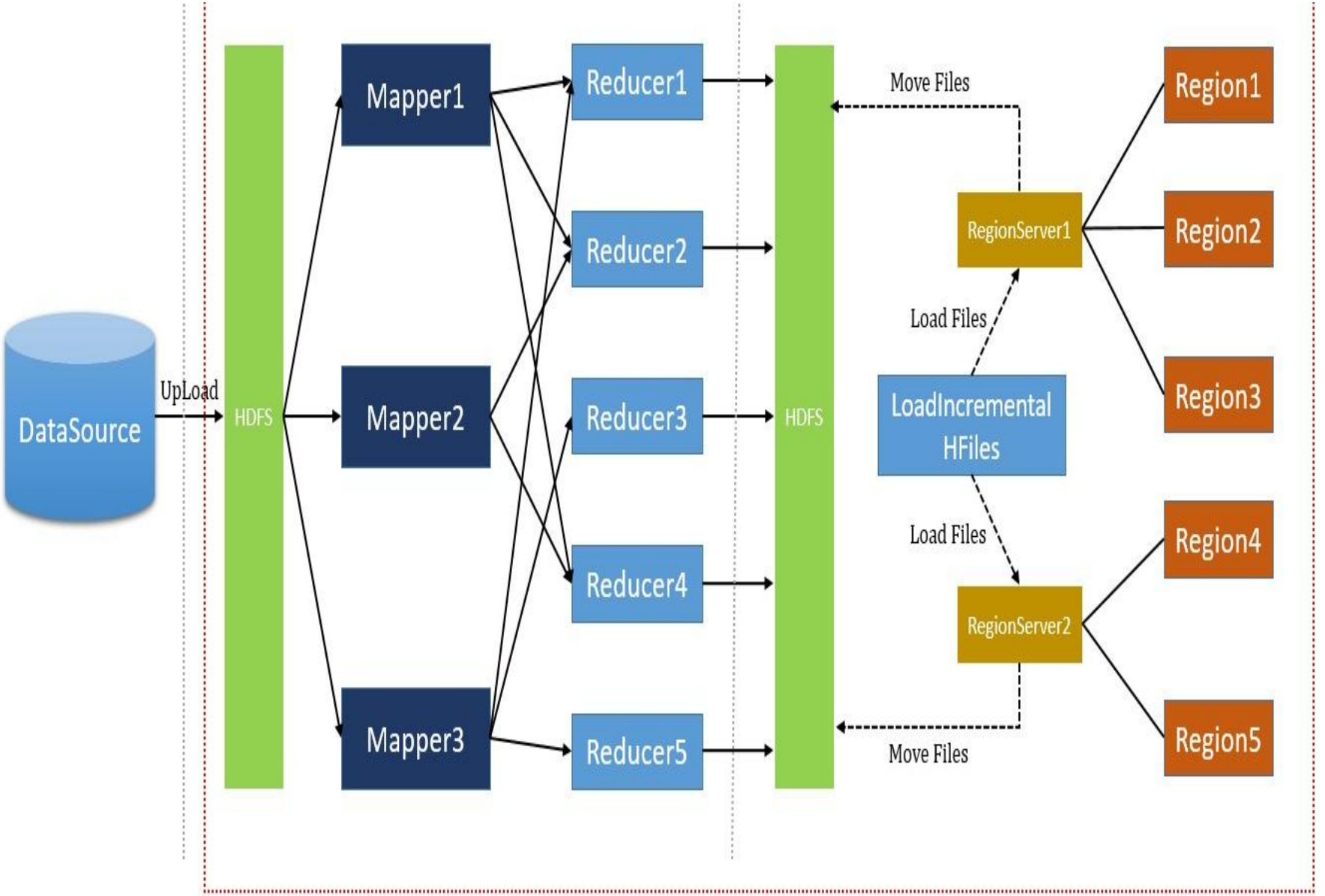


First, unordered

Then, sorted through M/R to synchronize with Hbase regions

HBase Write Strategy

- Incremental Load
- For smaller/time-sensitive updates
- Data streamed into HBase incrementally as events occur - real-time data processing!
- Each incremental write results in a new version


DataSource
UpLoad
HDFS
Mapper1
Mapper2
Mapper3
Reducer1
Reducer2
Reducer3
Reducer4
Reducer5
HDFS
Move Files
RegionServer1
Load Files
LoadIncremental HFiles
Load Files
RegionServer2
Move Files
Region1
Region2
Region3
Region4
Region5

HBase Read Strategy

- Get: Used to retrieve a single row based on its row key.
- Scan: Used for range queries to retrieve multiple rows with a specified range of row keys.

HBase Read Strategy

- Block Cache: keep frequently accessed data blocks in memory, reducing the need to read from disk.
- When data is read, HBase checks the block cache first;

HBase Read Strategy

- Bloom Filters: quickly determine if a specific row or column exists in a store file (HFile) without reading the entire file.

Hands On

Creating a Table

```
HBaseAdmin admin= new HBaseAdmin(config);
HColumnDescriptor []column;
column= new HColumnDescriptor[2];
column[0]=new HColumnDescriptor("columnFamily1:");
column[1]=new HColumnDescriptor("columnFamily2:");
HTableDescriptor desc= new HTableDescriptor(Bytes.toBytes("MyTable"));
desc.addFamily(column[0]);
desc.addFamily(column[1]);
admin.createTable(desc);
```

Operations On Regions: **Get()**

- Given a key → return corresponding record
- For each value return the highest version

```
Get get = new Get(Bytes.toBytes("row1"));
Result r = htable.get(get);
5.8.1.2. Default Get Example ue(Bytes.toBytes("cf"), Bytes.toBytes("attr"));  // returns current version of value
```

- Can control the number of versions you want

```
Get get = new Get(Bytes.toBytes("row1"));
get.setMaxVersions(3);  // will return last 3 versions of row
Result r = htable.get(get);
byte[] b = r.getValue(Bytes.toBytes("cf"), Bytes.toBytes("attr"));  // returns current version of value
List<KeyValue> kv = r.getColumn(Bytes.toBytes("cf"), Bytes.toBytes("attr"));  // returns all versions of
```

Operations On Regions: **Scan()**

```
HTable htable = ...      // instantiate HTable

Scan scan = new Scan();
scan.addColumn(Bytes.toBytes("cf"),Bytes.toBytes("attr"));
scan.setStartRow( Bytes.toBytes("row"));                   // start key is inclusive
scan.setStopRow( Bytes.toBytes("row" +  (char)0));  // stop key is exclusive
ResultScanner rs = htable.getScanner(scan);
try {
  for (Result r = rs.next(); r != null; r = rs.next()) {
  // process result...
} finally {
  rs.close();  // always close the ResultScanner!
}
```

Get()

Select value from table where key='com.apache.www' AND label='anchor:apache.com'

Row key	Time Stamp	Column "anchor:"	
"com.apache.www"	t12		
	t11		
	t10	"anchor:apache.com"	"APACHE"
"com.cnn.www"	t9	"anchor:cnnsi.com"	"CNN"
	t8	"anchor:my.look.ca"	"CNN.com"
	t6		
	t5		
	t3		

Scan()

Select value from table
where anchor='cnnsi.com'

Row key	Time Stamp	Column "anchor:"	
"com.apache.www"	t12		
	t11		
	t10	"anchor:apache.com"	"APACHE"
"com.cnn.www"	t9	"anchor:cnnsi.com"	"CNN"
	t8	"anchor:my.look.ca"	"CNN.com"
	t6		
	t5		
	t3		

Operations On Regions: **Put()**

- Insert a new record (with a new key), Or
- Insert a record for an existing key

Implicit version number (timestamp)

```
Put put = new Put(Bytes.toBytes(row));
put.add(Bytes.toBytes("cf"), Bytes.toBytes("attr1"), Bytes.toBytes( data));
htable.put(put);
```

Explicit version number

```
Put put = new Put( Bytes.toBytes(row));
long explicitTimeInMs = 555;  // just an example
put.add(Bytes.toBytes("cf"), Bytes.toBytes("attr1"), explicitTimeInMs, Bytes.toBytes(data));
htable.put(put);
```

Operations On Regions: **Delete()**

- Marking table cells as deleted
- **Multiple levels**
  - Can mark an entire column family as deleted
  - Can make all column families of a given row as deleted

- **All operations are logged by the RegionServers**
- **The log is flushed periodically**

HBase: **Joins**

- HBase does not support joins

- Can be done in the application layer
  - Using scan() and get() operations

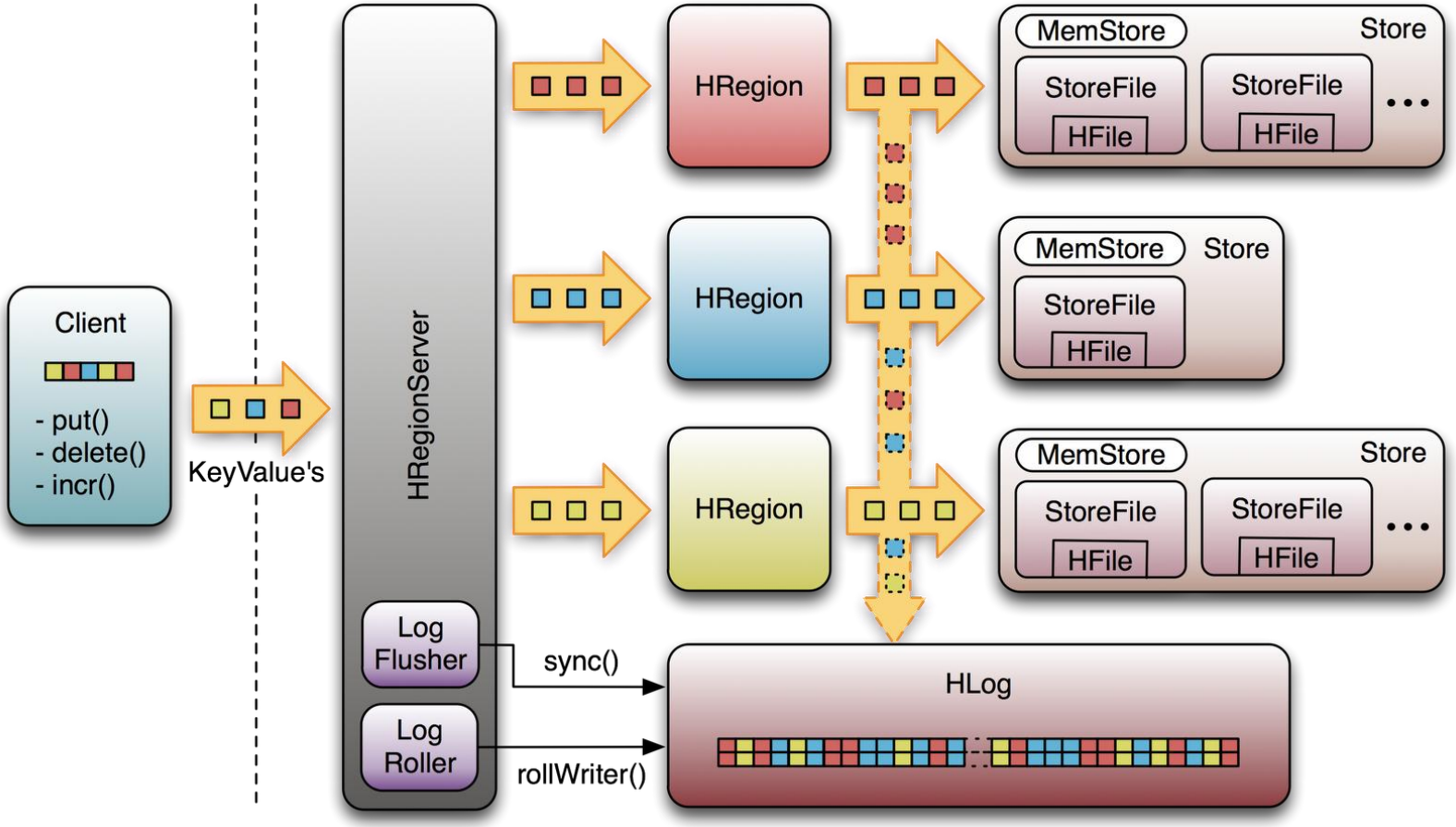
Altering a Table

```
Configuration config = HBaseConfiguration.create();
HBaseAdmin admin = new HBaseAdmin(conf);
String table = "myTable";

admin.disableTable(table);

HColumnDescriptor cf1 = ...;
admin.addColumn(table, cf1);      // adding new ColumnFamily
HColumnDescriptor cf2 = ...;
admin.modifyColumn(table, cf2);    // modifying existing ColumnFamily

admin.enableTable(table);
```

Disable the table before changing the schema

6.1. Schema Creation

Logging Operations



HBase Deployment