# Apache HBase

BAS Academy

# Agenda

- About Hbase
- Basic Concepts
- Hbase Architecture
- Shell Commands
- DDL and DML
- Hands On

# About HBase

# HBase

- Hbase is distributed column oriented database built on top of HDFS
- HBase store its files on HDFS (HFiles and WAL)
- HBase supports massively parallelized processing via MapReduce
- HBase  supports Auto-Sharding

**Auto-Sharding:**

- Tables are dynamically distributed by the system when they become too large
- HBase tables are distributed on the cluster via regions, and regions are automatically split and re-distributed as your data grows.

- HBase is very much a distributed database
- HBase is really more a "Data Store" than "Data Base" because it lacks many of the features you find in an RDBMS, such as typed columns, secondary indexes, triggers, and advanced query languages

# Features of HBase

- It provides real time CRUD operations (Create, Retrieve, Update, Delete) unlike HDFS
- Horizontally scalable and have automatic failover mechanism
- It provides data replication across clusters.

- Type of NoSQL DB (Not Only SQL)
  - Does not provide SQL based access
  - Does not adhere to relational model for storage

# Hbase Vs HDFS

| HDFS | HBase |
| --- | --- |
| HDFS is a distributed file system suitable for storing large files. | HBase is a database built on top of the HDFS. |
| HDFS does not support fast individual record lookups. | HBase provides fast lookups for larger tables. |
| It provides high latency batch processing | It provides low latency access to single rows from billions of records (Random access). |
| It provides only sequential access of data. | HBase internally uses Hash tables and provides random access, and it stores the data in indexed HDFS files for faster lookups. |

# Hbase Vs RDBMS

| HBase | RDBMS |
| --- | --- |
| HBase is schema-less, it doesn't have the concept of fixed columns schema; defines only column families. | An RDBMS is governed by its schema, which describes the whole structure of tables. |
| It is built for wide tables. HBase is horizontally scalable. | It is thin and built for small tables. Hard to scale. |
| No transactions are there in HBase. | RDBMS is transactional. |
| It has de-normalized data. | It will have normalized data. |
| It is good for semi-structured as well as structured data. | It is good for structured data. |

- **A note on the `NULL` value**
  - In RDBMS `NULL` cells need to be set and occupy space
  - In HBase, `NULL` cells or columns are simply not stored

# When to Use HBase

Following are the scenarios when you should use HBase:

You are using the concept of variable schema.

You are using it for random selects and range scans by key.

When to use HBase

You have enough data and hundreds of millions or billions of rows.

You have sufficient commodity hardware with a minimum of five nodes.

You need to carefully evaluate HBase for mixed workloads.

# Basic Concepts

# Storage Mechanism in HBase

- Table is a collection of rows.
- Row is a collection of column families.
- Rows are referenced by unique key.
- Column family is a collection of columns.
- Column is a collection of key value pairs.

COLUMN FAMILIES

| Row key | personal data | | professional data | |
|---|---|---|---|---|
| empid | name | city | designation | salary |
| 1 | raju | hyderabad | manager | 50,000 |
| 2 | ravi | chennai | sr.engineer | 30,000 |
| 3 | rajesh | delhi | jr.engineer | 25,000 |

- Column exist only when inserted. Nulls are free
- Data is stored in cells
- For each cell multiple versions are kept (3 by default)
- Versions are stored in decreasing timestamp order

- **Value = Table+RowKey+Family+Column+Timestamp**

Cell Coordinates= Key

Value

| Row key | Column Family | Column Qualifier | Timestamp | Value |
|---|---|---|---|---|
| Smithj | Address | city | 1391813876369 | Nashville |

# Hbase Cells

**An example - Logical representation of how values are stored**

| Row Key | Time stamp | Name Family | | Address Family | |
|---|---|---|---|---|---|
| | | first_name | last_name | number | address |
| row1 | t1 | **Bob** | **Smith** | | |
| | t5 | | | 10 | First Lane |
| | t10 | | | 30 | Other Lane |
| | t15 | | | **7** | **Last Street** |
| row2 | t20 | **Mary** | Tompson | | |
| | t22 | | | 77 | **One Street** |
| | t30 | | **Thompson** | | |

# Hbase Architecture

# HBase Components

**Region:**

- Regions are contiguous ranges of rows stored together

**Region Server:**

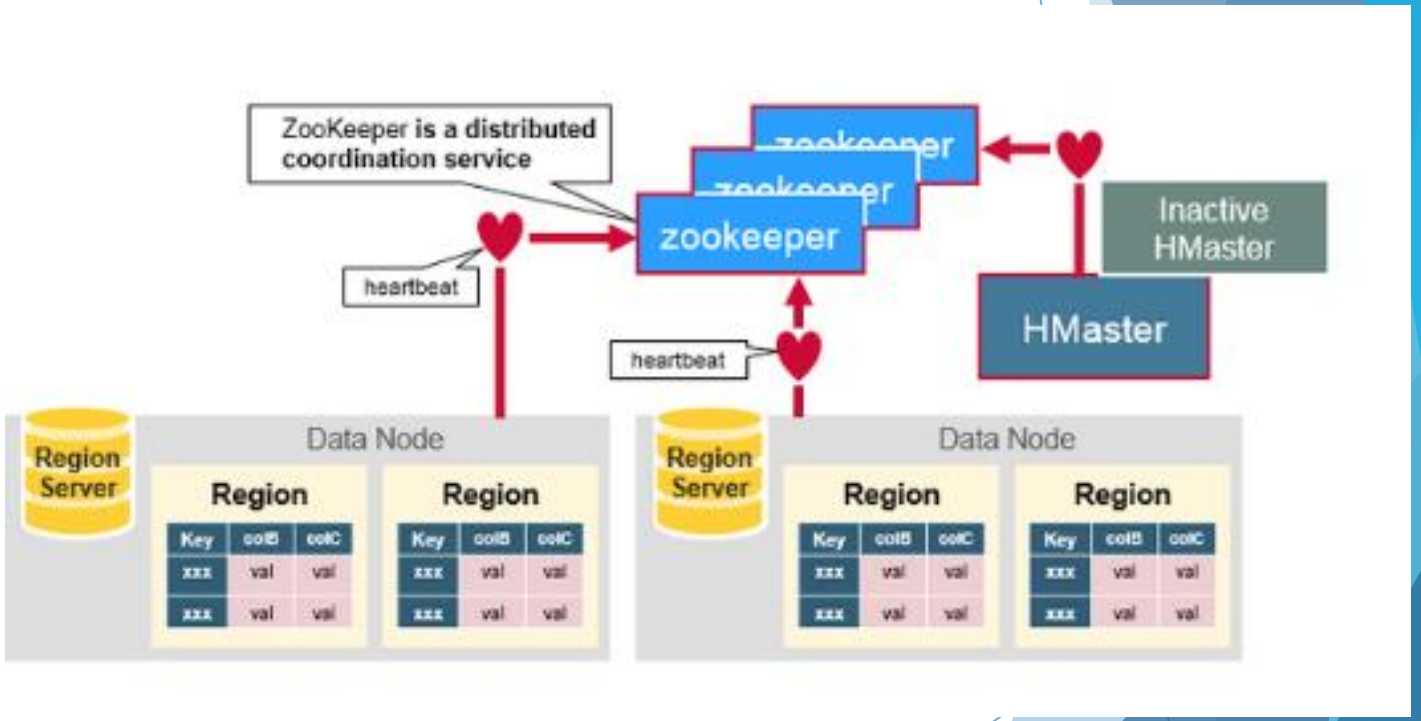
- Regions are assigned to the nodes in the cluster, called Region Servers

**HMaster:**

- Responsible for managing regions and schema management( adding/removing tables)

**Zookeeper:**

- ZooKeeper for does coordination service between client and HMaster



# Region Server Components

**MemStore:**

- It stores new data which has not yet been written to disk
- When in-memory data exceeds maximum value it is flushed into Hfile
- There is one MemStore per column family

**HFile**

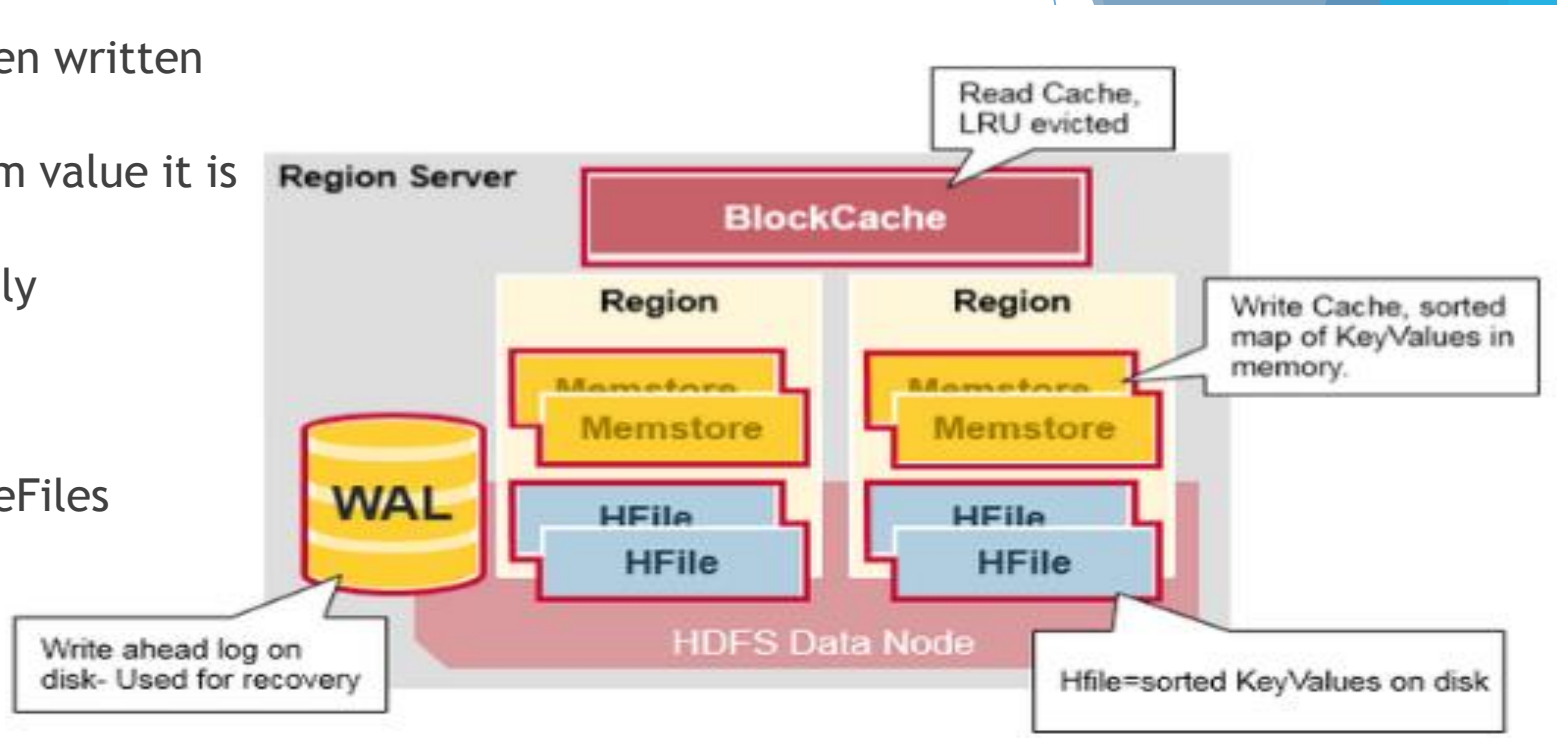
- Data is stored in files called Hfiles/StoreFiles
- Hfile is a key-value map

**WAL: (Write Ahead Log)**

- The WAL is used to store new data that hasn't yet been persisted to permanent storage
- It can be re-played incase of server failure

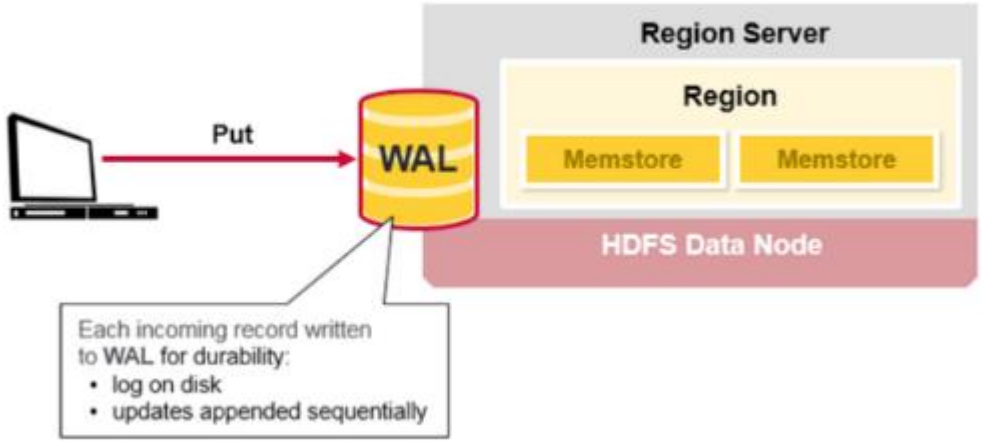
**BlockCache:**

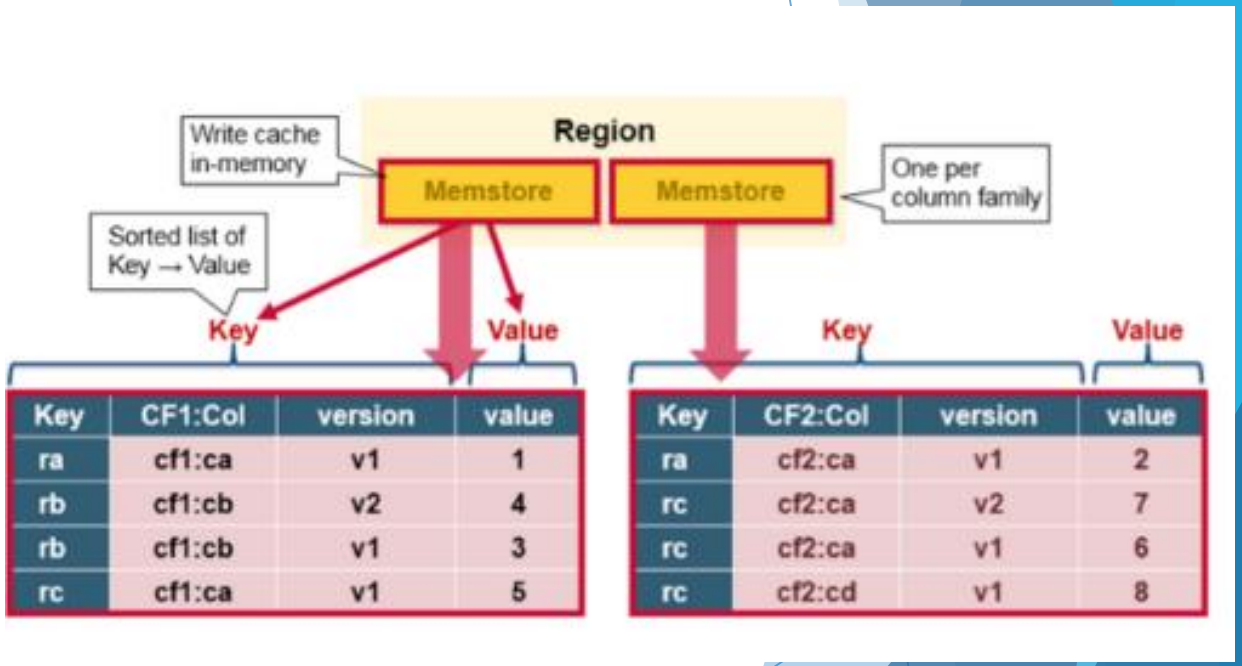
- It stores frequently read data in memory



# Hbase Write

- The first step is to write the data to the write-ahead log, the WAL.

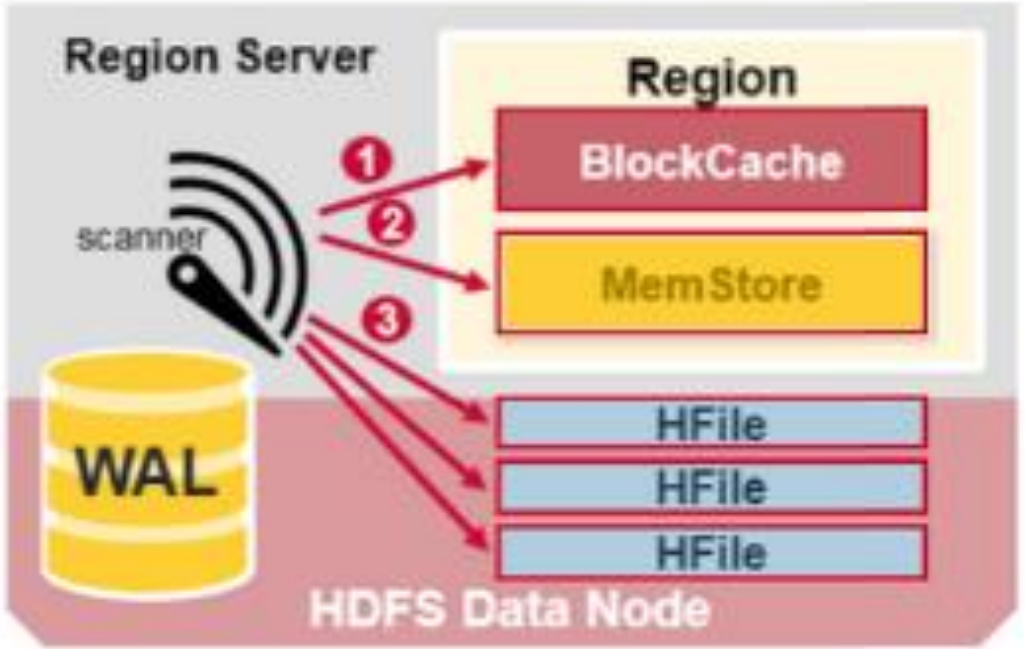


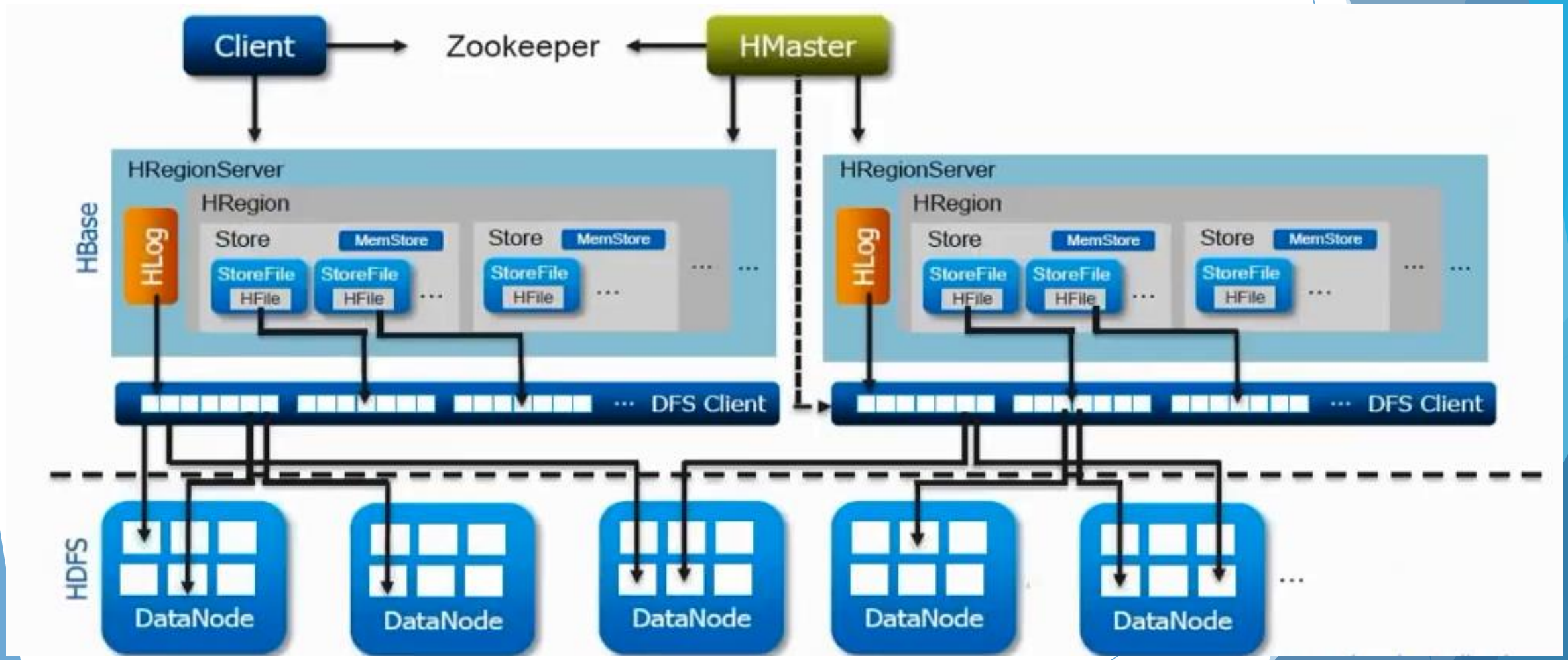

- Once the data is written to the WAL, it is placed in the MemStore.
- Then, the put request acknowledgement returns to the client.

# HBase Read

1. First the scanner looks for the Row KeyValues in the Block cache
2. Next the scanner looks in the MemStore
3. If all row cells not in MemStore or blockCache, look in HFiles



# Hbase Architecture



# Hbase Data Compaction

- Hbase performs periodic compaction to control the  number of Hfiles and keep the cluster well balanced

Data compaction can be done in 2 ways

- Minor Compaction - Smaller Hfiles are merged into larger Hfiles
- Major Compaction – Merge all the files within a column family into a single file

What happens when data is deleted

- HFiles are immutable
- A delete marker (also known as tombstone marker ) is written to indicate that a given key is deleted
- During the read process, data marked as deleted is skipped
- Compactions finalize the deletion process

# Where are Hfiles Stored?

- HBase has a root directory set to “/hbase” in HDFS
- **Every table has its own directory**

/hbase
- .logs
- .oldlogs
- .hbase.id
- .hbase.version
- /example-table

- /example-table
  - .tableinfo
  - .tmp
  - “...Key1...”
    - .oldlogs
    - .regioninfo
    - .tmp
    - colfam1/
- colfam1/
  - “....column-key1...”

# HFile Example

- Below is one of the key/value pair stored in the HFile which represents a cell in a table.

  K: row-550/colfam1:50/1309812287166/Put/vlen=3 V: 501

  where

  `row key` is `row-550`
  `column family` is `colfam1`
  `column family identifier (aka column)` is `50`
  `time stamp` is `1309812287166`
  `value` stored is `501`.

The dump of a HFile (which stores a lot of key/value pairs) looks like below in the same order

K: row-550/colfam1:50/1309812287166/Put/vlen=2 V: 50
**K: row-550/colfam1:51/1309813948222/Put/vlen=2 V: 51**
K: row-551/colfam1:30/1309812287200/Put/vlen=2 V: 51
K: row-552/colfam1:31/1309813948256/Put/vlen=2 V: 52
K: row-552/colfam1:49/1309813948280/Put/vlen=2 V: 52
**K: row-552/colfam1:51/1309813948290/Put/vlen=2 V: 52**

# Tall-Narrow vs. Flat-Wide Tables

**Tall-Narrow Tables**

- Few columns
- Many rows

**Flat-Wide Tables**

- Many columns
- Few rows

- It is recommended to go for Tall-Narrow Tables

– Tall-Narrow: each row represents a single blog, multiple rows will represent a single user

Tall

| Row id | entry:blog |
|---|---|
| userid1_t1 | Blah blah blah... |
| userid1_t2 | Interesting stuff... |
| ... | ... |
| userid1_tN | Important news... |
| .... | ... |
| .... | ... |
| userid3_500 | Blah blah blah... |
| userid3_501 | Blah blah blah... |

Narrow

Flat-Wide: each row represents a single user

Flat

| Row id | entry:t1 | entry:t2 | ... | entry:tN |
|---|---|---|---|---|
| userid1 | Blah blah blah... | Important news.. | ... | Interesting stuff... |
| userid2 | Interesting stuff... | Blah blah blah... | ... | Important news.. |
| userid3 | Important news... | Interesting stuff... | ... | Blah blah blah... |

# Shell Commands

# Hbase Shell

HBase Shell – JRuby IRB (Interactive Ruby Shell)

**HBase Shell supports various commands**

- General
  - status, version
- Data Definition Language (DDL)
  - alter, create, describe, disable, drop, enable, exists, is_disabled, is_enabled, list
- Data Manipulation Language (DML)
  - count, delete, deleteall, get, get_counter, incr, put, scan, truncate
- Cluster administration
  - balancer, close_region, compact, flush, major_compact, move, split, unassign, zk_dump, add_peer, disable_peer, enable_peer, remove_peer, start_replication, stop_replication

**$ <hbase_install>/bin/hbase shell**

HBase Shell; enter 'help<RETURN>' for list of supported commands.
Type "exit<RETURN>" to leave the HBase Shell

Type 'help' to get a listing of commands

- $ help "command" (quotes are required)
  - > help "get"

# Naming Convention

- **Quote all names**
  - Table and column names
  - Single quotes for text
    - hbase> get 't1', 'myRowId'
  - Double quotes for binary
    - Use hexadecimal representation of that binary value
    - hbase> get 't1', "key\x03\x3f\xcd"
- **Uses ruby hashes to specify parameters**
  - {'key1' => 'value1', 'key2' => 'value2', …}
  - Example:

```
hbase> get 'UserTable', 'userId1', {COLUMN => 'address:str'}
```

# DDL and DML

# DDL and DML

- **Let's walk through an example**
  1. Create a table
     - Define column families
  2. Populate table with data records
     - Multiple records
  3. Access data
     - Count, get and scan
  4. Edit data
  5. Delete records
  6. Drop table

# Create Table

- **Create table called 'Blog' with the following schema**
  - 2 families
    - 'info' with 3 columns: 'title', 'author', and 'date'
    - 'content' with 1 column family: 'post'

| Blog | | |
|---|---|---|
| Family: | info: | Columns: title, author, date |
| | content: | Columns: post |

```
hbase> create 'Blog', {NAME=>'info'}, {NAME=>'content'}
0 row(s) in 1.3580 seconds
```

# Create Data

- **Put command format:**

```
hbase> put 'table', 'row_id', 'family:column', 'value'
```

```
# insert row 1
put 'Blog', 'Matt-001', 'info:title', 'Elephant'
put 'Blog', 'Matt-001', 'info:author', 'Matt'
put 'Blog', 'Matt-001', 'info:date', '2009.05.06'
put 'Blog', 'Matt-001', 'content:post', 'Do elephants like monkeys?'
…
```

- **Populate data with multiple records**

| Row Id | info:title | info:author | info:date | content:post |
|---|---|---|---|---|
| Matt-001 | Elephant | Matt | 2009.05.06 | Do elephants like monkeys? |
| Matt-002 | Monkey | Matt | 2011.02.14 | Do monkeys like elephants? |
| Bob-003 | Dog | Bob | 1995.10.20 | People own dogs! |
| Michelle-004 | Cat | Michelle | 1990.07.06 | I have a cat! |
| John-005 | Mouse | John | 2012.01.15 | Mickey mouse. |

# Access Data

## Access Data

- count: display the total number of records
- get: retrieve a single row
- scan: retrieve a range of rows

# Access Data - Count

- **Count is simple**
  - hbase> count 'table_name'

```
hbase> count 'Blog'
Current count: 1, row: Bob-003
Current count: 2, row: John-005
Current count: 3, row: Matt-001
Current count: 4, row: Matt-002
Current count: 5, row: Michelle-004
```

# Access Data - Get

## Select single row with 'get' command

- hbase> get 'table', 'row_id'
  - Returns an entire row
- Requires table name and row id
- Optional: timestamp or time-range, and versions

```
hbase> get 'Blog', 'unknownRowId'
COLUMN                CELL
0 row(s) in 0.0250 seconds
```

Row Id doesn't exist

```
hbase> get 'Blog', 'Michelle-004'
COLUMN                    CELL
content:post     timestamp=1326061625690, value=I have a cat!
info:author      timestamp=1326061625630, value=Michelle
info:date        timestamp=1326061625653, value=1990.07.06
info:title       timestamp=1326061625608, value=Cat
4 row(s) in 0.0420 seconds
```

Returns ALL the columns, displays 1 column per row!!!

# Access Data - Scan

- **Scan entire table or a portion of it**
- **Load entire row or explicitly retrieve column families, columns or specific cells**
- **To scan an entire table**
  - hbase> scan 'table_name'
- **Limit the number of results**
  - hbase> scan 'table_name', {LIMIT=>1}

Scan the entire table, grab ALL the columns

```
hbase(main):014:0> scan 'Blog'
ROW                    COLUMN+CELL
Bob-003 column=content:post, timestamp=1326061625569,
                             value=People own dogs!
Bob-003 column=info:author, timestamp=1326061625518, value=Bob
Bob-003 column=info:date, timestamp=1326061625546,
                             value=1995.10.20
Bob-003 column=info:title, timestamp=1326061625499, value=Dog
John-005       column=content:post, timestamp=1326061625820,
                             value=Mickey mouse.
John-005       column=info:author, timestamp=1326061625758,
                             value=John
…
Michelle-004   column=info:author, timestamp=1326061625630,
                             value=Michelle
Michelle-004   column=info:date, timestamp=1326071670471,
                             value=1990.07.08
Michelle-004   column=info:title, timestamp=1326061625608,
                             value=Cat
5 row(s) in 0.0670 seconds
```

# Access Data – Scan…

## Scan a range

- hbase> scan 'Blog', {STARTROW=>'startRow', STOPROW=>'stopRow'}
- Start row is inclusive, stop row is exclusive
- Can provide just start row or just stop row

Stop row is exclusive, row ids that start with John will not be included

```
hbase> scan 'Blog', {STOPROW=>'John'}
ROW      COLUMN+CELL
Bob-003  column=content:post, timestamp=1326061625569,
                              value=People own dogs!
Bob-003  column=info:author, timestamp=1326061625518,
                              value=Bob
Bob-003  column=info:date, timestamp=1326061625546,
                              value=1995.10.20
Bob-003       column=info:title, timestamp=1326061625499,
                              value=Dog
1 row(s) in 0.0410 seconds
```

# Edit Data

- **Put command inserts a new value if row id doesn't exist**
- **Put updates the value if the row does exist**
- **But does it really update?**
  - Inserts a new version for the cell
  - Only the latest version is selected by default
  - N versions are kept per cell
    - configured per family at creation:

```
hbase> create 'table', {NAME => 'family', VERSIONS => 7}
```

    - 3 versions are kept by default

# Delete Records

- **Delete cell by providing table, row id and column coordinates**
  - delete 'table', 'rowId', 'column'
  - Deletes all the versions of that cell

```
hbase> get 'Blog', 'Bob-003', 'info:date'
COLUMN        CELL
info:date     timestamp=1326061625546, value=1995.10.20
1 row(s) in 0.0200 seconds

hbase> delete 'Blog', 'Bob-003', 'info:date'
0 row(s) in 0.0180 seconds

hbase> get 'Blog', 'Bob-003', 'info:date'
COLUMN        CELL
0 row(s) in 0.0170 seconds
```

# Drop Table

- **Must disable before dropping**
  - puts the table "offline" so schema based operations can be performed
  - `hbase> disable 'table_name'`
  - `hbase> drop 'table_name'`
- **For a large table it may take a long time....**

```
hbase> list
TABLE
Blog
1 row(s) in 0.0120 seconds

hbase> disable 'Blog'
0 row(s) in 2.0510 seconds

hbase> drop 'Blog'
0 row(s) in 0.0940 seconds

hbase> list
TABLE
0 row(s) in 0.0200 seconds
```

Take the table offline for schema modifications

# Hands On

# Thank You

Keerthiga Barathan