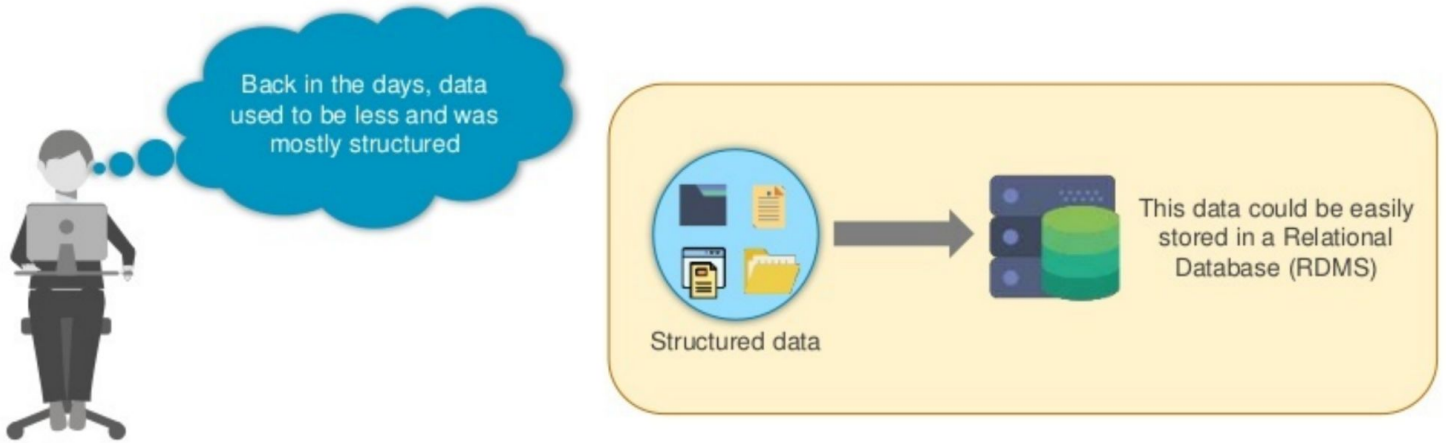
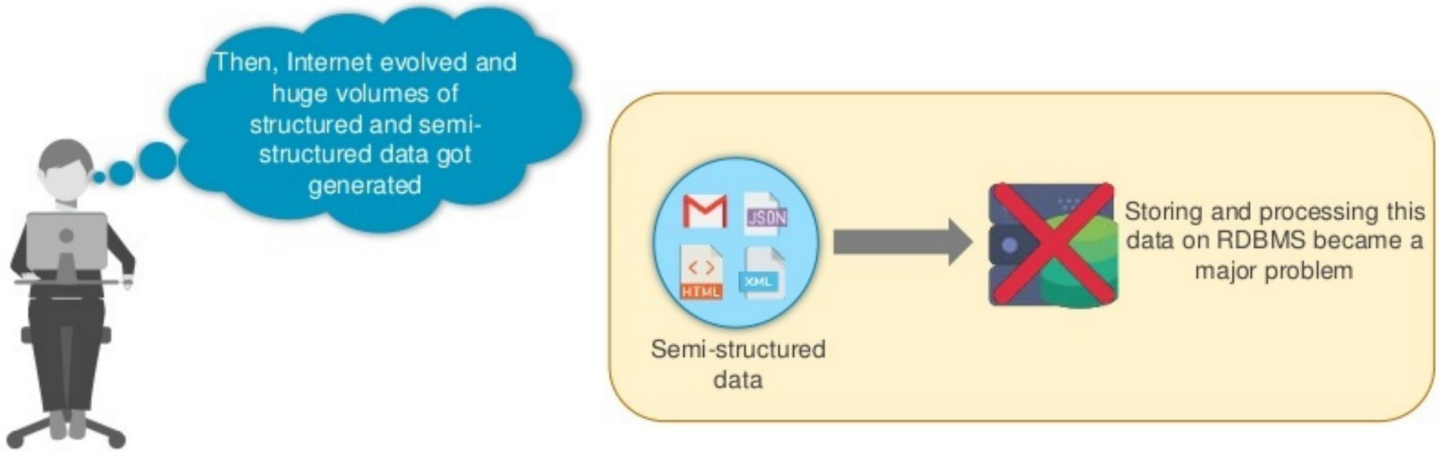
# HBase & Phoenix

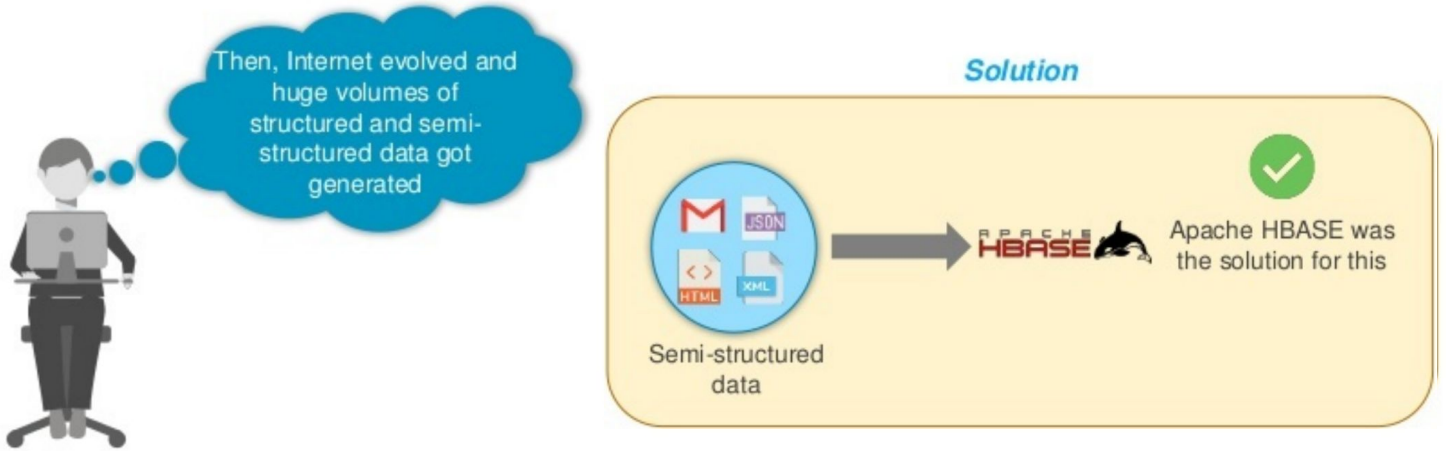
# Introduction



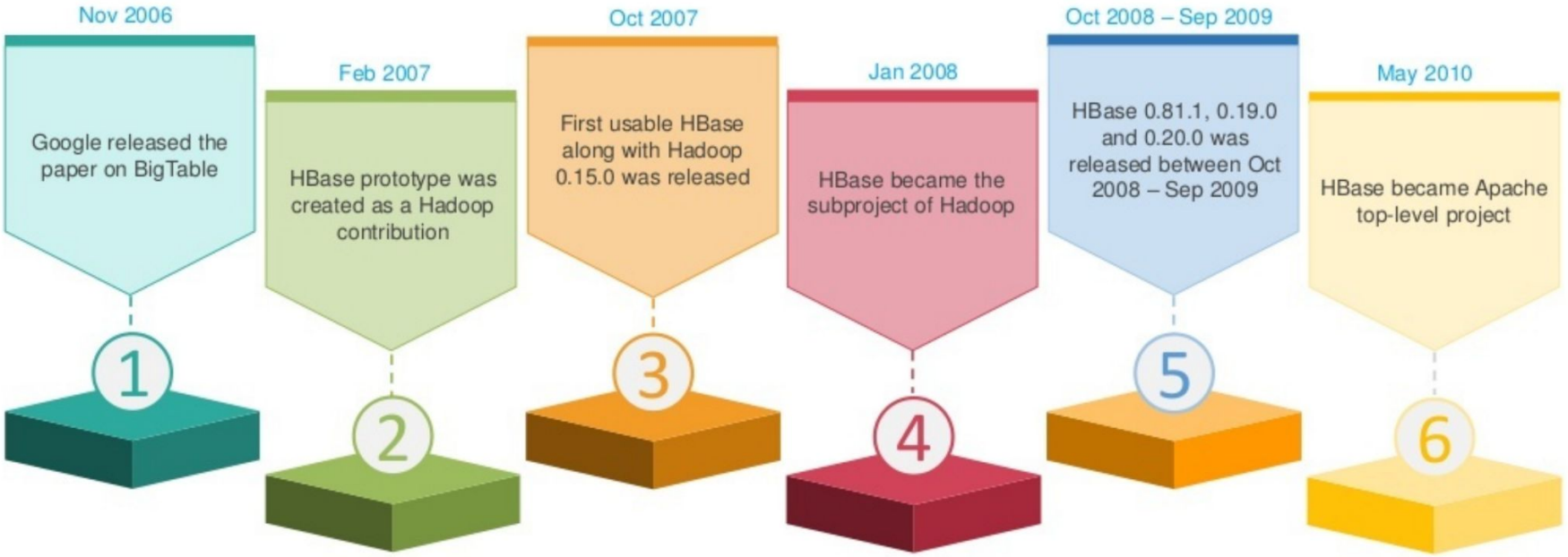
# Introduction



# Introduction



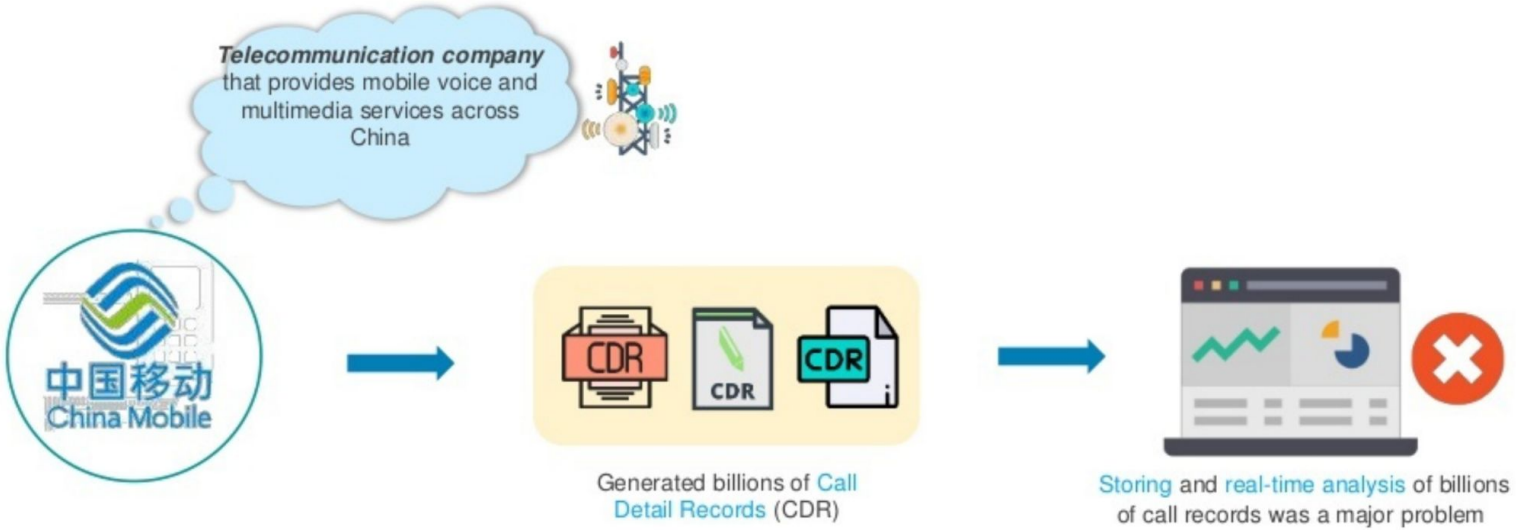
# Little bit history...



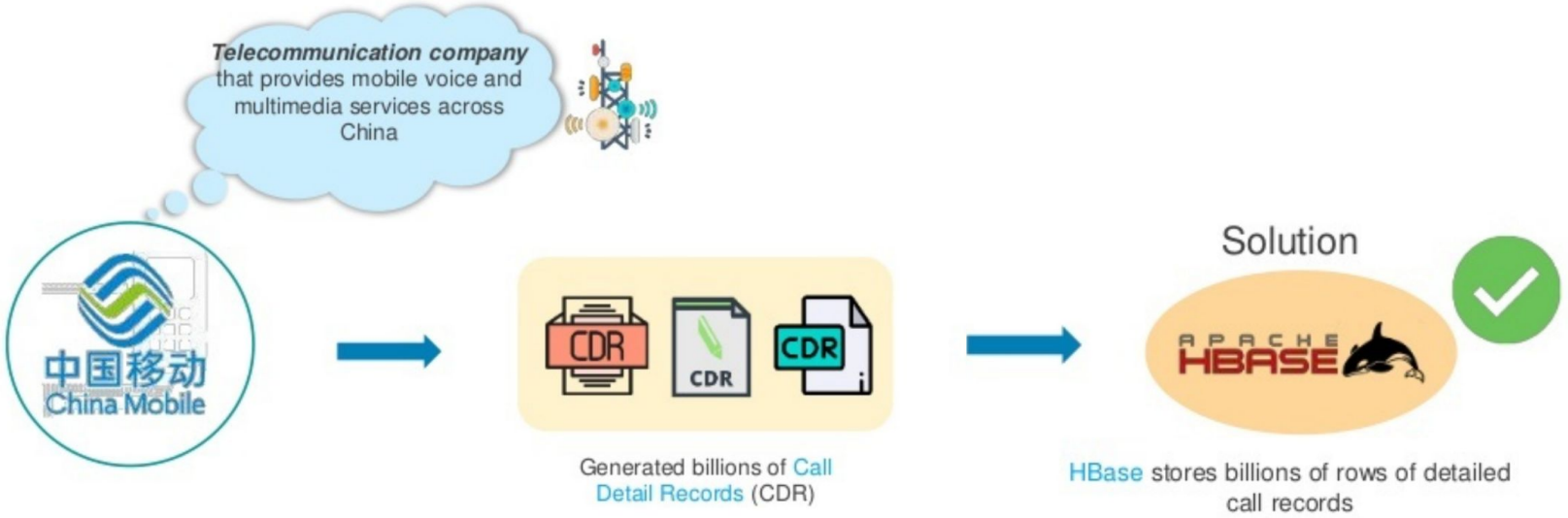
# What is HBase?

- Open source project built on top of Apache Hadoop
- NoSQL database
- Distributed, scalable datastore
- Column-family datastore

# HBase Use Case



# HBase Use Case



# HBase Use Case

**Time Series Data**
- Sensor, System metrics, Events, Log files
- User Activity
- High Volume, Velocity Writes

**Information Exchange**
- Email, Chat, Inbox
- High Volume, Velocity Read\Write

**Enterprise Application Backend**
- Online Catalog
- Search Index
- Pre-Computed View
- High Volume, Velocity Reads

# HBase Application

Medical

HBase is used for storing genome sequences

Storing disease history of people or an area

E-Commerce

HBase is used for storing logs about customer search history

Performs analytics and target advertisement for better business insights

Sports

HBase stores match details and history of each match

Uses this data for better prediction

# HBase vs RDBMS

| | RDBMS | HBase |
|---|---|---|
| Data layout | Row-oriented | Column-family-oriented |
| Transactions | Multi-row ACID | Single row only |
| Query language | SQL | get/put/scan/etc * |
| Security | Authentication/Authorization | Work in progress |
| Indexes | On arbitrary columns | Row-key only |
| Max data size | TBs | ~1PB |
| Read/write throughput limits | 1000s queries/second | Millions of queries/second |

# HDFS vs HBase

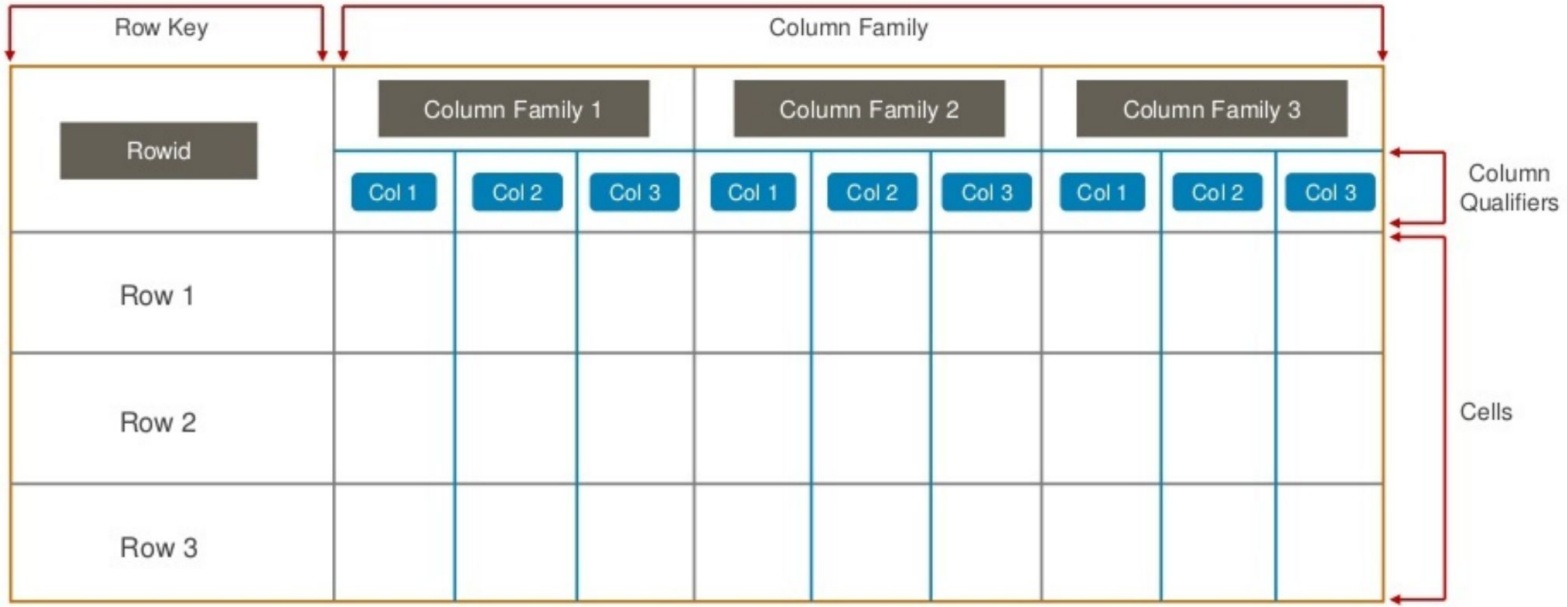
| | Plain HDFS/MR | HBase |
|---|---|---|
| Write pattern | Append-only | Random write, bulk incremental |
| Read pattern | Full table scan, partition table scan | Random read, small range scan, or table scan |
| Hive (SQL) performance | Very good | 4-5x slower |
| Structured storage | Do-it-yourself / TSV / SequenceFile / Avro / ? | Sparse column-family data model |
| Max data size | 30+ PB | ~1PB |

# HBase Features

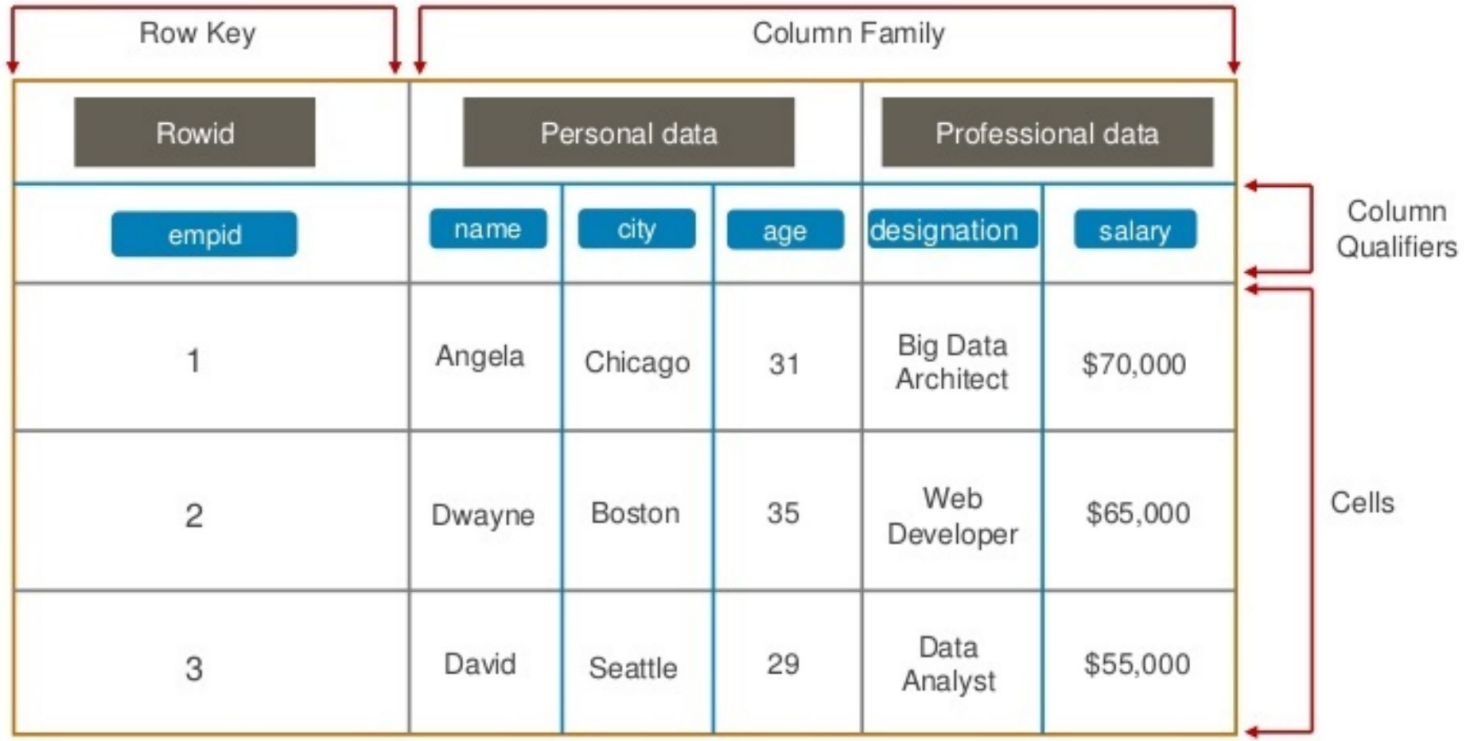
| Scalable | Automatic failure support | Consistent read and write | JAVA API for client access | Block cache and bloom filters |
|---|---|---|---|---|
|  |  |  |  |  |
| Data can be scaled across various nodes as it is stored in HDFS | Write Ahead Log across clusters which provides automatic support against failure | HBase provides consistent read and write of data | Provides easy to use JAVA API for clients | Supports block cache and bloom filters for high volume query optimization |

# HBase Data Model

# HBase Column Oriented



# HBase Column Oriented



# Data Model Overview

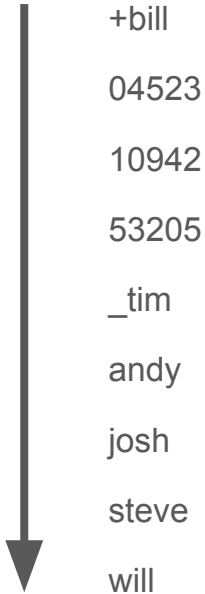
| Component | Description |
|---|---|
| *Table* | Data organized into tables |
| *RowKey* | Data stored in rows; Rows identified by RowKeys |
| *Region* | Rows are grouped in Regions |
| *Column Family* | Columns grouped into families |
| *Column Qualifier (Column)* | Indentifies the column |
| *Cell* | Combination of the row key, column family, column, timestamp; contains the value |
| *Version* | Values within in cell versioned by version number → timestamp |

# Data model: Rows

| RowKey | contacs | | | accounts | | … |
|---|---|---|---|---|---|---|
| | mobile | email | skype | UAH | USD | … |
| 084ab67e | VAL | | | VAL | | |
| 2333bbac | | VAL | VAL | | | |
| 342bbecc | VAL | | | | | |
| 4345235b | | | | | VAL | |
| 565c4f8f | VAL | | VAL | | VAL | |
| 675555ab | VAL | VAL | VAL | VAL | VAL | |
| 9745c563 | | VAL | | VAL | | |
| a89d3211 | VAL | VAL | | VAL | VAL | |
| f091e589 | | | VAL | VAL | VAL | |

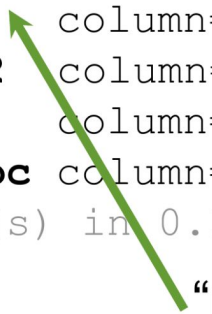
# Data model: Rows order

Rows are sorted in lexicographical order



# Rows ordered lexicographically

```
hbase(main):001:0> scan 'table1'
ROW       COLUMN+CELL
row-1     column=cf1:, timestamp=1297073325971 ...
row-10    column=cf1:, timestamp=1297073337383 ...
row-11    column=cf1:, timestamp=1297073340493 ...
row-2     column=cf1:, timestamp=1297073329851 ...
row-22    column=cf1:, timestamp=1297073344482 ...
row-3     column=cf1:, timestamp=1297073333504 ...
row-abc   column=cf1:, timestamp=1297073349875 ...
7 row(s) in 0.1100 seconds
```



"**row-10**" comes before "**row-2**".
How to fix?

Pad "row-2" with a "0".
i.e., "row-02"

# Data model: Regions

| RowKey | contacs | | | accounts | | … | |
|---|---|---|---|---|---|---|---|
| | mobile | email | skype | UAH | USD | … | |
| 084ab67e | VAL | | | VAL | | | R1 |
| 2333bbac | | VAL | VAL | | | | |
| … | VAL | | | | | | |
| 4345235b | | | | | VAL | | R2 |
| … | VAL | | VAL | | VAL | | |
| 675555ab | VAL | VAL | VAL | VAL | VAL | | |
| 9745c563 | | VAL | | VAL | | | R3 |
| … | VAL | VAL | | VAL | VAL | | |
| f091e589 | | | VAL | VAL | VAL | | |

RowKeys ranges → Regions

# Data model: Column Family

| RowKey | contacs | | | accounts | |
|---|---|---|---|---|---|
| | mobile | email | skype | UAH | USD |
| 084ab67e | VAL | | | VAL | |
| 2333bbac | | VAL | VAL | | |
| 342bbecc | VAL | | | | |
| 4345235b | | | | | VAL |
| 565c4f8f | VAL | | VAL | | VAL |
| 675555ab | VAL | VAL | VAL | VAL | VAL |
| 9745c563 | | VAL | | VAL | |

# Data model: Column Family

- Column Families are part of the table schema and defined on the table creation
- Columns are grouped into column families
- Column Families are stored in separate HFiles at HDFS
- Data is grouped to Column Families by common attribute

# Data model: Columns

| RowKey | contacs | | | accounts | |
|---|---|---|---|---|---|
| | mobile | email | skype | UAH | USD |
| 084ab67e | 977685798 | user123@gmail.com | user123 | 2875 | 10 |
| ... | ... | ... | ... | ... | ... |

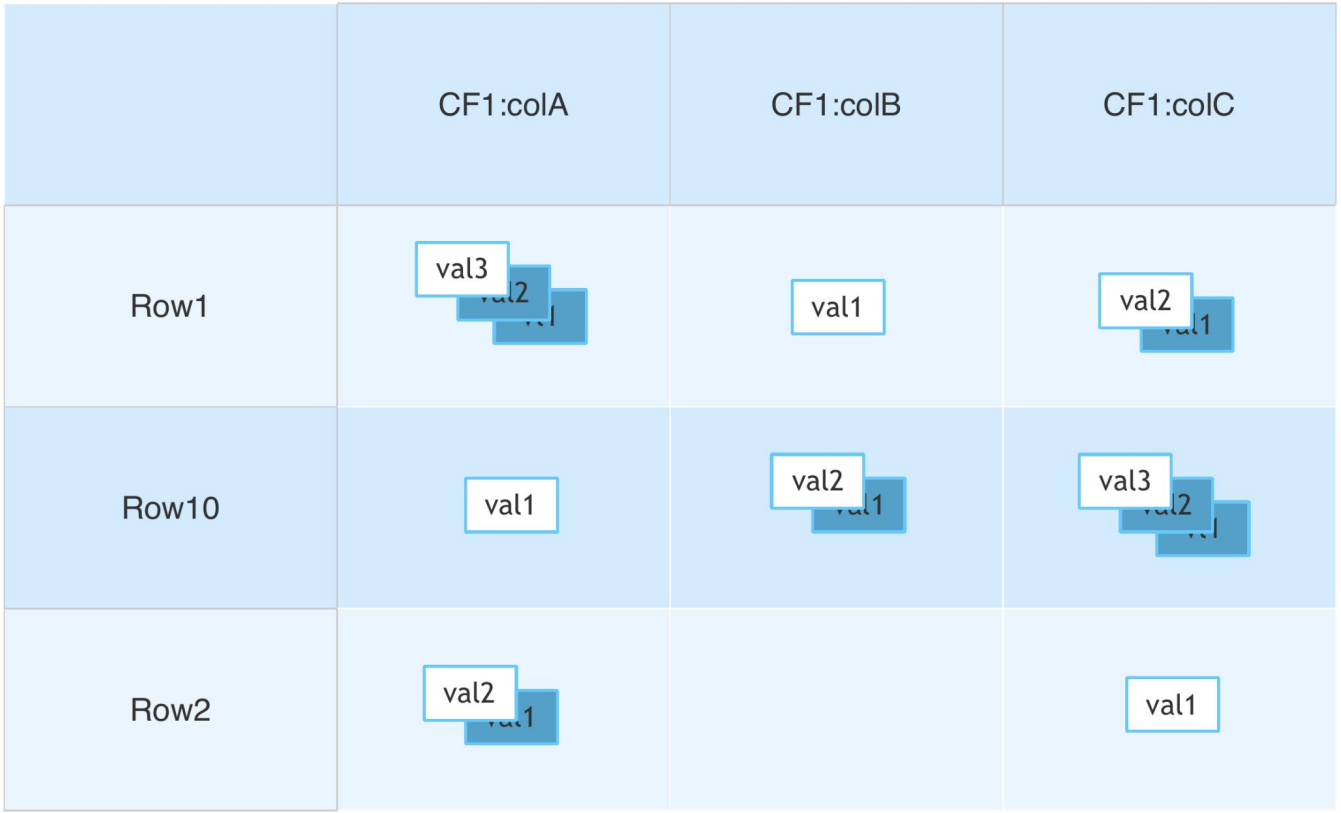
# Data model: Cells

| Key | | | | Value |
|---|---|---|---|---|
| RowKey | Column Family | Column Qualifier | Version | |
| 084ab67e | contacs | mobile | 1454767653075 | 977685798 |

# Data model: Cells

- Data is stored in KeyValue format
- Value for each cell is specified by complete coordinates: RowKey, Column Family, Column Qualifier, Version

# Data model: Versions



# Data Model: Version (Cell Timestamps)

- All cells are created with a timestamp
- Column family defines how many versions of a cell to keep
- Updates always create a new cell
- Deletes create a tombstone
- Queries can include an “as-of ” timestamp to return point-in-time values

# HBase General Commands

# Status

## `Syntax:status`

This command will give details about the system status like a number of servers present in the cluster, active server count, and average load value. You can also pass any particular parameters depending on how detailed status you want to know about the system. The parameters can be '**summary**', **'simple'**, or **'detailed'**, the default parameter provided is "summary".

# Table help

```
Syntax:table_help
```

This command guides

What and how to use table-referenced commands

It will provide different HBase shell command usages and its syntaxes

Here in the screen shot above, its shows the syntax to "create" and "get_table" command with its usage. We can manipulate the table via these commands once the table gets created in HBase.

It will give table manipulations commands like put, get and all other commands information.

# Table help

`Syntax:table_help`

This command guides

What and how to use table-referenced commands

It will provide different HBase shell command usages and its syntaxes

It will give table manipulations commands like put, get and all other commands information.

# Tables Managements commands

These commands will allow programmers to create tables and table schemas with rows and column families.

The following are Table Management commands

- Create
- List
- Describe
- Disable
- Disable_all
- Enable
- Enable_all
- Drop
- Drop_all
- Show_filters
- Alter
- Alter_status

# Table Create

```
Syntax: create <tablename>, <columnfamilyname>
Example: create 'education' ,'guru99'
```

The above example explains how to create a table in HBase with the specified name given according to the dictionary or specifications as per column family. In addition to this we can also pass some table-scope attributes as well into it

# Create table

```
create 'user_accounts', {NAME=>'contacts',VERSIONS=>1},
{NAME=>'accounts'}
```

# Table List

```
Syntax:list
```

- "List" command will display all the tables that are present or created in HBase
- The output showing in above screen shot is currently showing the existing tables in HBase
- Here in this screenshot, it shows that there are total 8 tables present inside HBase
- We can filter output values from tables by passing optional regular expression parameters

# Table Describe

```
Syntax:describe <table name>
Example: describe 'education'
```

- This command describes the named table.
- It will give more information about column families present in the mentioned table
- In our case, it gives the description about table "education."
- It will give information about table name with column families, associated filters, versions and some more details.

# Table disable

```
Syntax:disable <tablename>
Example: disable 'education'
```

- This command will start disabling the named table
- If table needs to be deleted or dropped, it has to disable first

# Table Enable

```
Syntax:enable <tablename>
Example: enable 'education'
```

- This command will start enabling the named table
- Whichever table is disabled, to retrieve back to its previous state we use this command
- If a table is disabled in the first instance and not deleted or dropped, and if we want to re-use the disabled table then we have to enable it by using this command.

# Table drop

```
Syntax:drop <table name>
Example: drop 'education'
```

- To delete the table present in HBase, first we have to disable it
- To drop the table present in HBase, first we have to disable it
- So either table to drop or delete first the table should be disable using disable command

# Table is_enabled

```
Syntax:is_enabled <table name>
Example: is_enabled 'education'
```

- This command will verify whether the named table is enabled or not. Usually, there is a little confusion between "enable" and "is_enabled" command action, which we clear here

# Table alter

```
Syntax:alter <tablename>, NAME=><column familyname>,
VERSIONS=>5
```

- This command alters the column family schema. To understand what exactly it does, we have explained it here with an example:

```
alter 'education', NAME='guru99_1', VERSIONS=>5
```

To change or add the 'guru99_1' column family in table 'education' from current value to keep a maximum of 5 cell VERSIONS,

- "education" is table name created with column name "guru99" previously
- Here with the help of an alter command we are trying to change the column family schema to guru99_1 from guru99

```
alter 'education', 'guru99_1', {NAME => 'guru99_2',
IN_MEMORY => true}, {NAME => 'guru99_3', VERSIONS => 5}
```

You can also operate the alter command on several column families as well. For example, we will define two new column to our existing table "education".

- We can change more than one column schemas at a time using this command
- guru99_2 and guru99_3 as shown in above screenshot are the two new column names that we have defined for the table education
- We can see the way of using this command in the previous screen shot

```
alter 'education', 'delete' =>' guru99_1'
```

In this command, we are trying to delete the column space name guru99_1 that we previously created in the first step

# Data manipulation commands

These commands will work on the table related to data manipulations such as putting data into a table, retrieving data from a table and deleting schema, etc.

The commands come under these are

- Put
- Get
- Delete
- Delete all
- Truncate
- Scan

# Put

```
Syntax: put <'tablename'>,<'rowname'>,<'columnvalue'>,<'value'>
Example: put 'guru99', 'r1', 'c1', 'value', 10
Snippets:
create 'guru99', {NAME=>'Edu'}
put 'guru99', 'r1', 'Edu:c1', 'value', 10
put 'guru99', 'r1', 'Edu:c1', 'value', 15
put 'guru99', 'r1', 'Edu:c1', 'value', 30
```

This command is used for following things

- It will put a cell 'value' at defined or specified table or row or column.
- It will optionally coordinate time stamp.

# Insert/Update

```
put 'user_accounts',
'user3455','contacts:mobile','977685798'

put 'user_accounts',
'user3455','contacts:email','user@mail.com',2
```

There is no update command. Just reinsert row

# Get

Syntax: **get <'tablename'>, <'rowname'>, {< Additional parameters>}**

Example:

- get 'guru99', 'r1', {COLUMN => 'c1'}
- get 'guru99', 'r1'

# Scan

Syntax: **scan <'tablename'>, {Optional parameters}**

Example: scan 'guru99'

This command scans entire table and displays the table contents.

# Read

```
get 'user_accounts', 'user3455'

get 'user_accounts', 'user3455', 'contacts:mobile'

get 'user_accounts', 'user3455', {COLUMN =>
'contacts:email', TIMESTAMP => 2}

scan ‘user_accounts’

scan 'user_accounts',{STARTROW=>'a',STOPROW=>'u'}
```

# Delete

Syntax: **delete <'tablename'>,<'row name'>,<'column name'>**

Example: delete 'guru99', 'r1', 'c1'

The above execution will delete row r1 from column family c1 in table "guru99."

# deleteall

Syntax: **deleteall <'tablename'>, <'rowname'>**

Example: deleteall 'guru99', 'r1', 'c1'

This will delete all the rows and columns present in the table. Optionally we can mention column names in that.

# Delete

```
delete 'user_accounts', 'user3455','contacts:mobile'

delete 'user_accounts', 'user3455','contacts:mobile',
1459690212356

deleteall 'user_accounts', 'user3455'
```

# Truncate

Syntax: **truncate <tablename>**

Example: deleteall 'guru99', 'r1', 'c1'

After truncate of an hbase table, the schema will present but not the records. This command performs 3 functions; those are listed below

- Disables table if it already presents
- Drops table if it already presents
- Recreates the mentioned table

# Demo

```
create 'newtbl', 'knowledge'
describe 'newtbl'
put 'newtbl', 'r1', 'knowledge:sports', 'cricket'
put 'newtbl', 'r1', 'knowledge:science', 'chemistry'
put 'newtbl', 'r2', 'knowledge:economics', 'macro economics'
put 'newtbl', 'r2', 'knowledge:music', 'pop music'
scan 'newtbl'
disable 'newtbl'
scan 'newtbl'
alter 'newtbl', 'test_info'
enable 'newtbl'
describe 'newtbl'
get 'newtbl', 'r1'
put 'newtbl', 'r1', 'knowledge:economics', 'market economics'
get 'newtbl', 'r1'
```

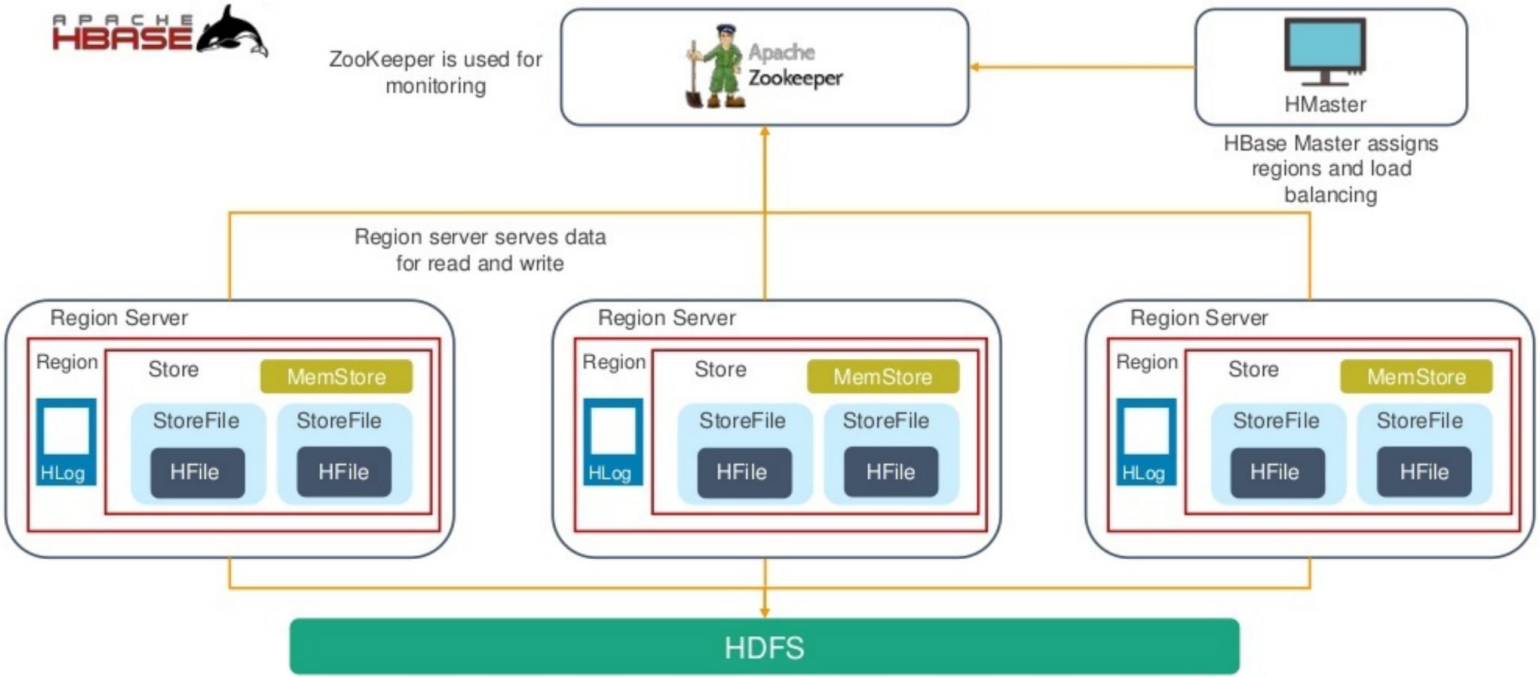
# Exercise: User Information

1. Create a table based on the side
2. Insert table data
3. Add an educational background column family, and educational backgrounds and titles to the user information table.
4. Query user names and addresses by user ID.

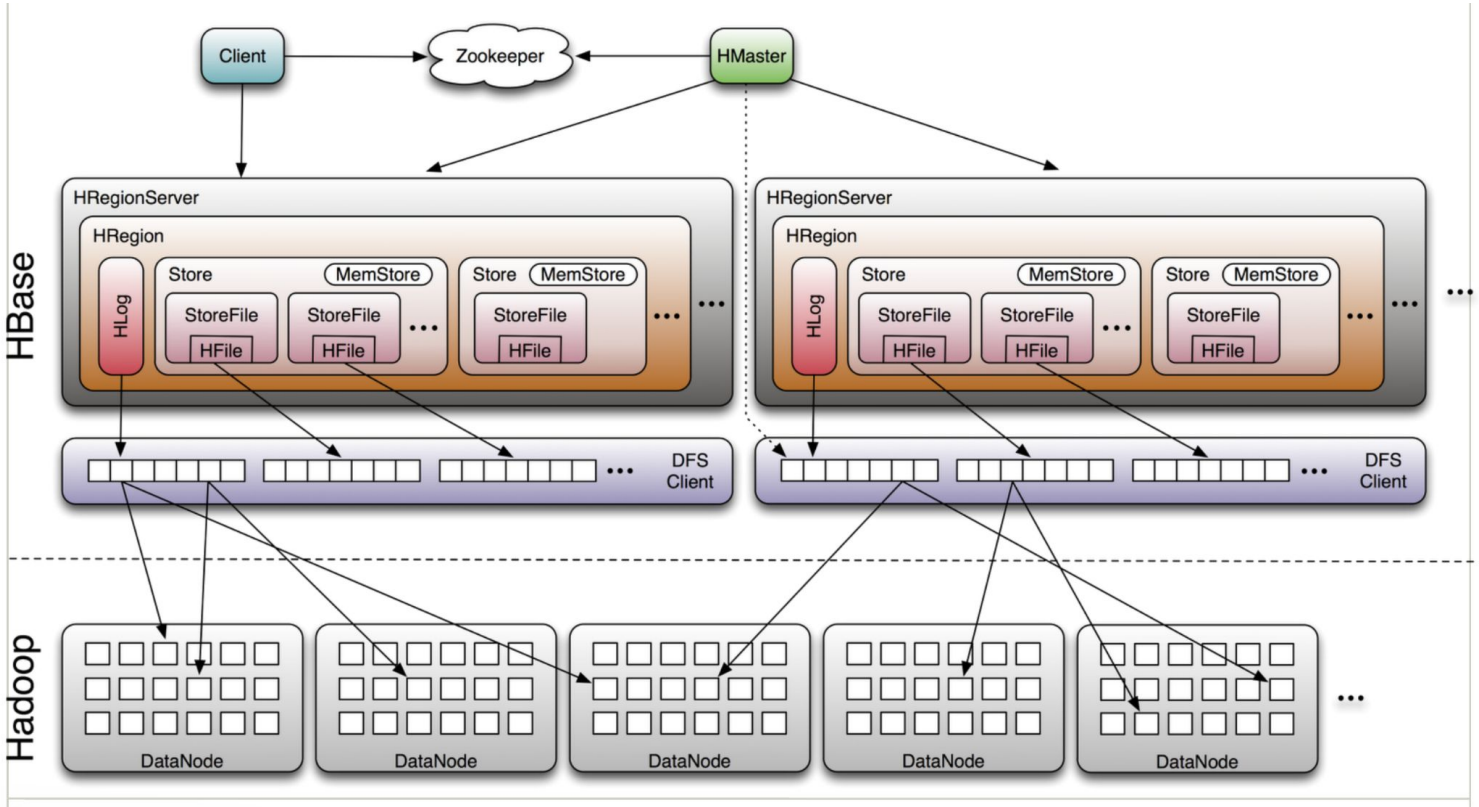
| SN | Name | Gender | Age | Address |
|---|---|---|---|---|
| 12005000201 | Tom | Male | 19 | Shenzhen, Guangdong |
| 12005000202 | Li Wanting | Female | 23 | Shijiazhuang, Hebei |
| 12005000203 | Wang Ming | Male | 26 | Ningbo, Zhejiang |
| 12005000204 | Li Gang | Male | 18 | Xiangyang, Hubei |
| 12005000205 | Zhao Enru | Female | 21 | Shangrao, Jiangxi |
| 12005000206 | Chen Long | Male | 32 | Zhuzhou, Hunan |
| 12005000207 | Zhou Wei | Female | 29 | Nanyang, Henan |
| 12005000208 | Yang Yiwen | Female | 30 | Kaixian, Chongqing |
| 12005000209 | Xu Bing | Male | 26 | Weinan, Shaanxi |
| 12005000210 | Xiao Kai | Male | 25 | Dalian, Liaoning |

# HBase Architecture

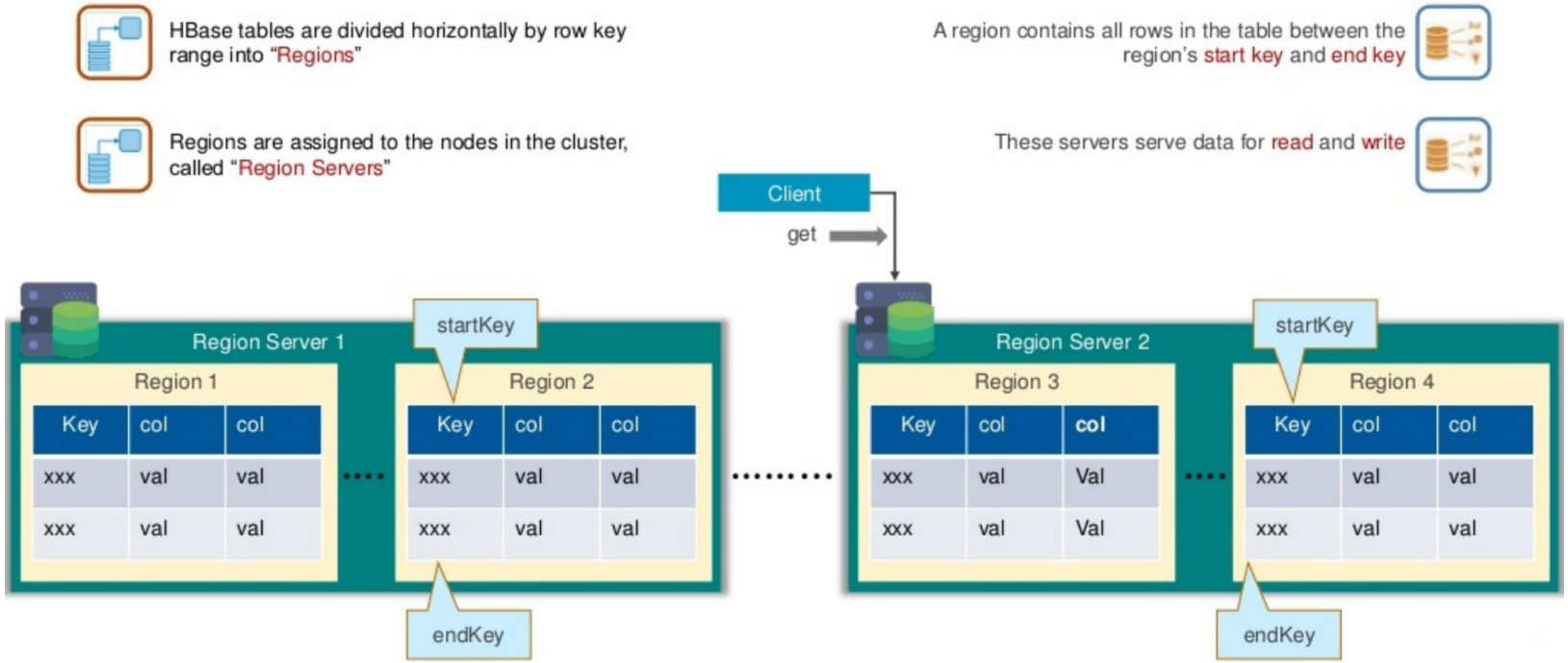
# HBase Architecture



# HBase Architecture



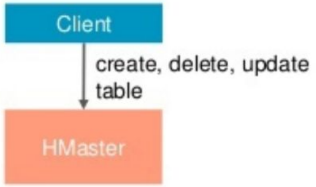
# HBase Components - Region

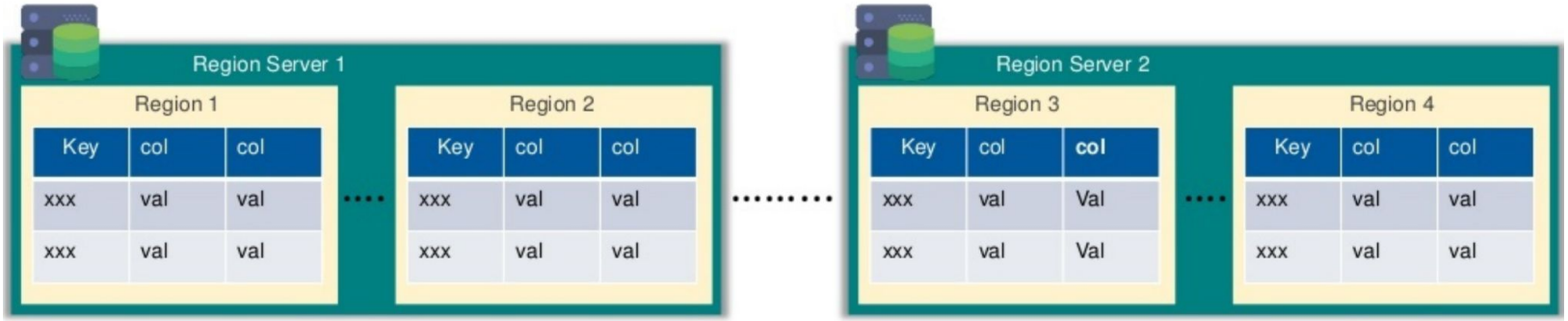


# HBase Components - HMaster

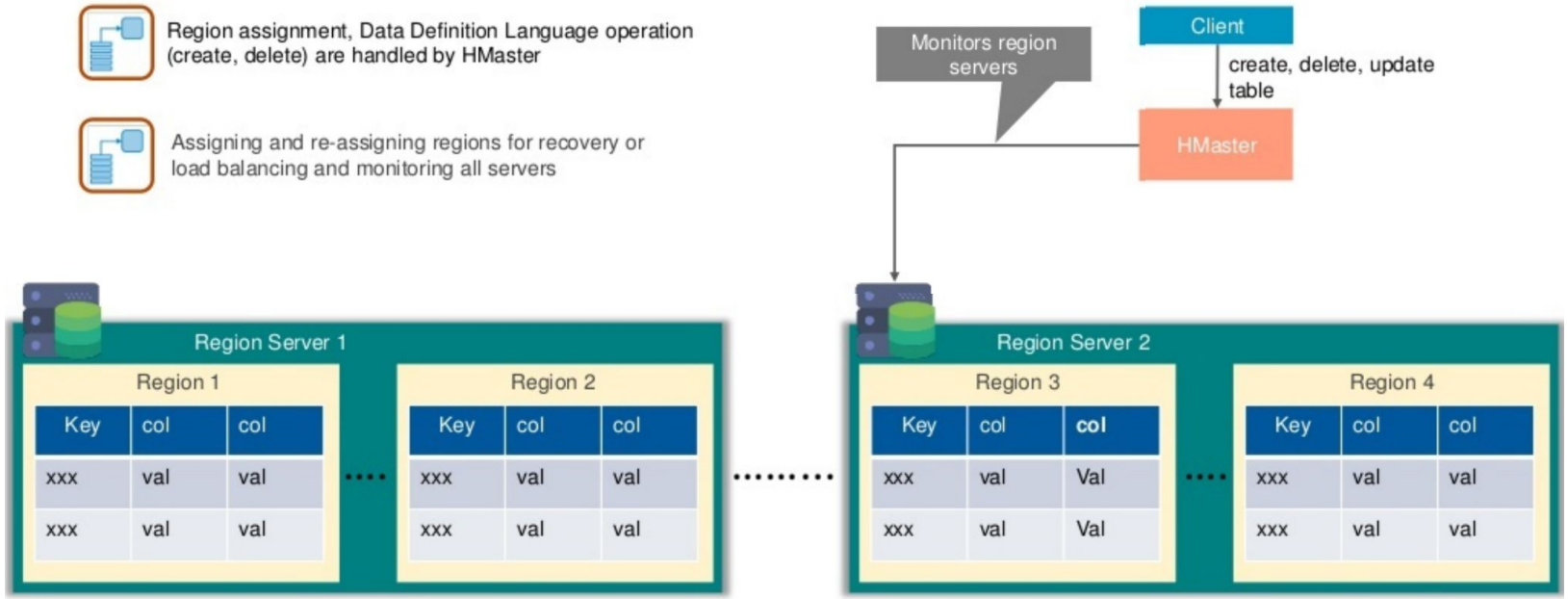
Region assignment, Data Definition Language operation (create, delete) are handled by HMaster

Assigning and re-assigning regions for recovery or load balancing and monitoring all servers

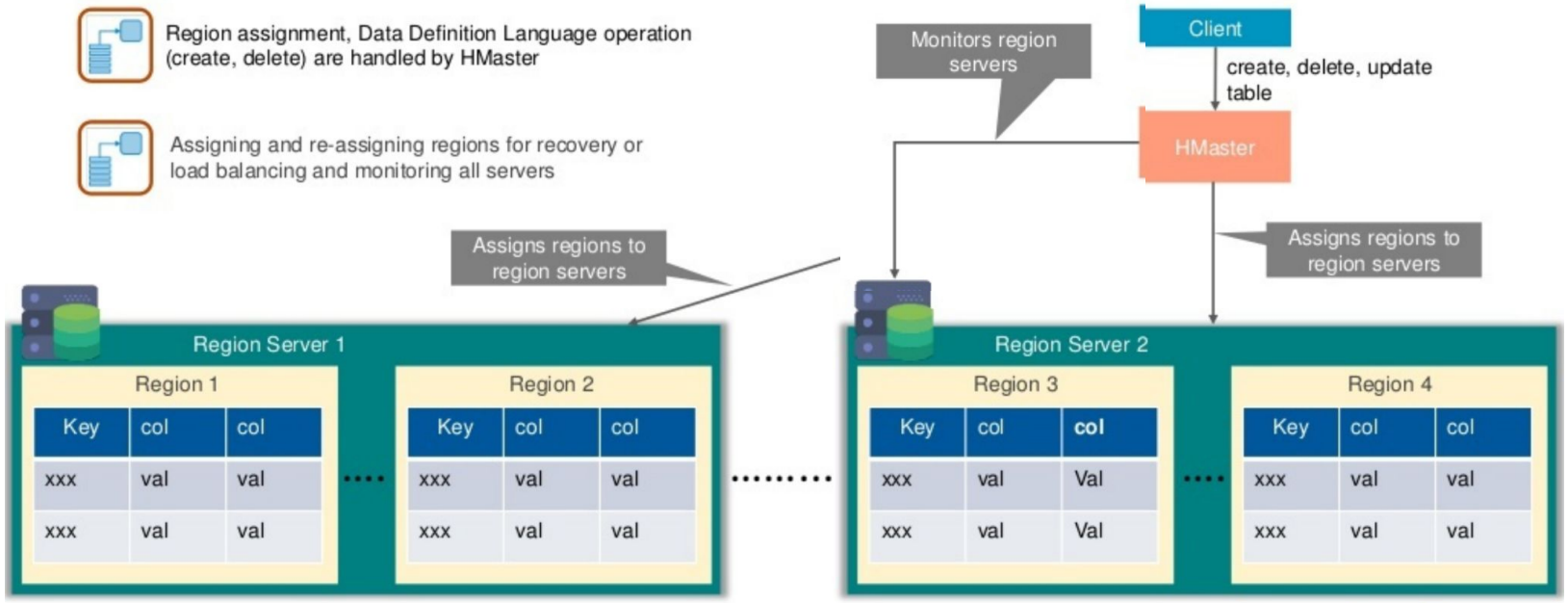


# HBase Components - HMaster



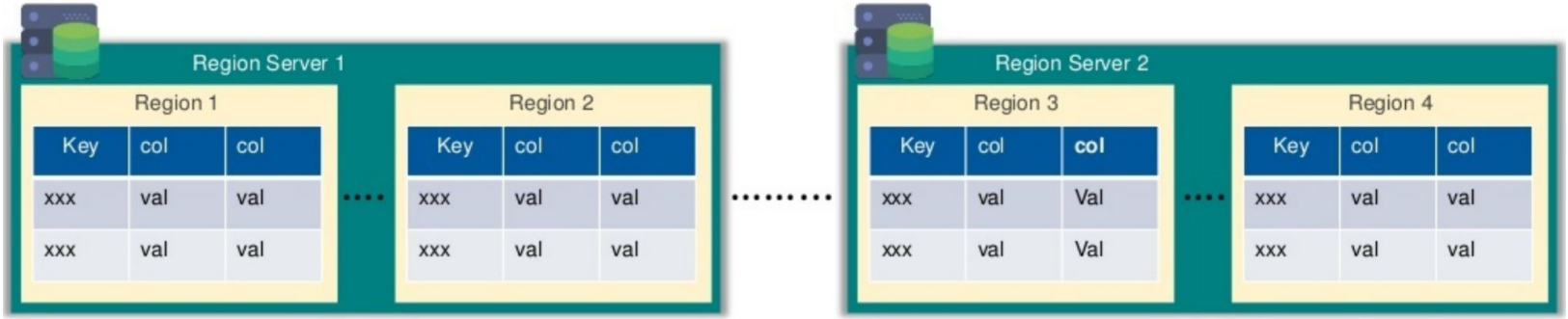
# HBase Components - HMaster



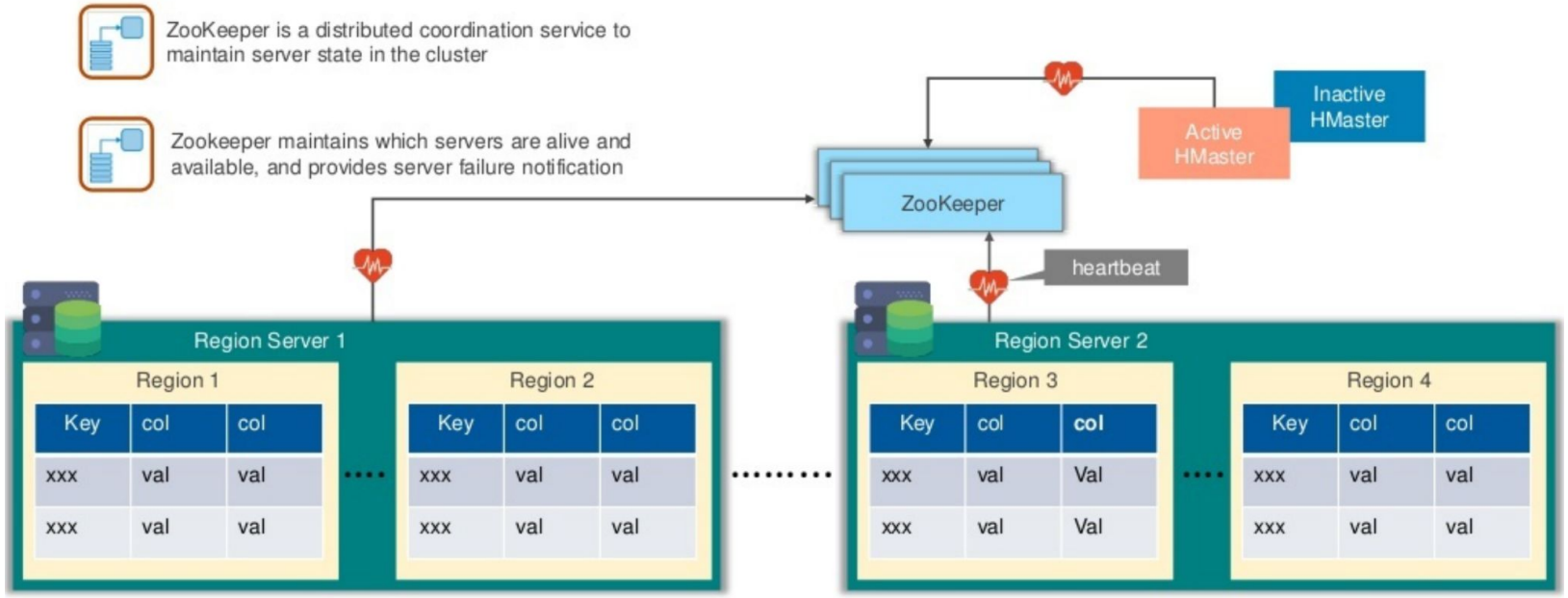
# HBase Components - Zookeeper

ZooKeeper is a distributed coordination service to maintain server state in the cluster

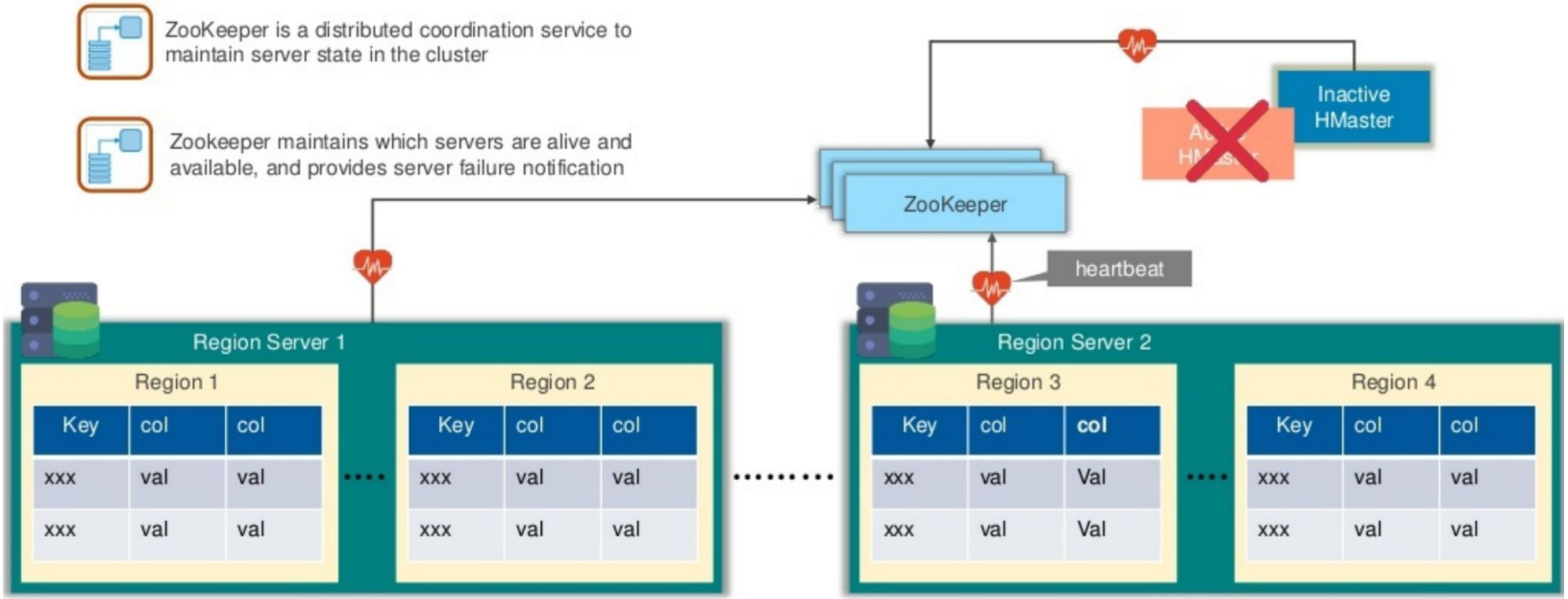
Zookeeper maintains which servers are alive and available, and provides server failure notification

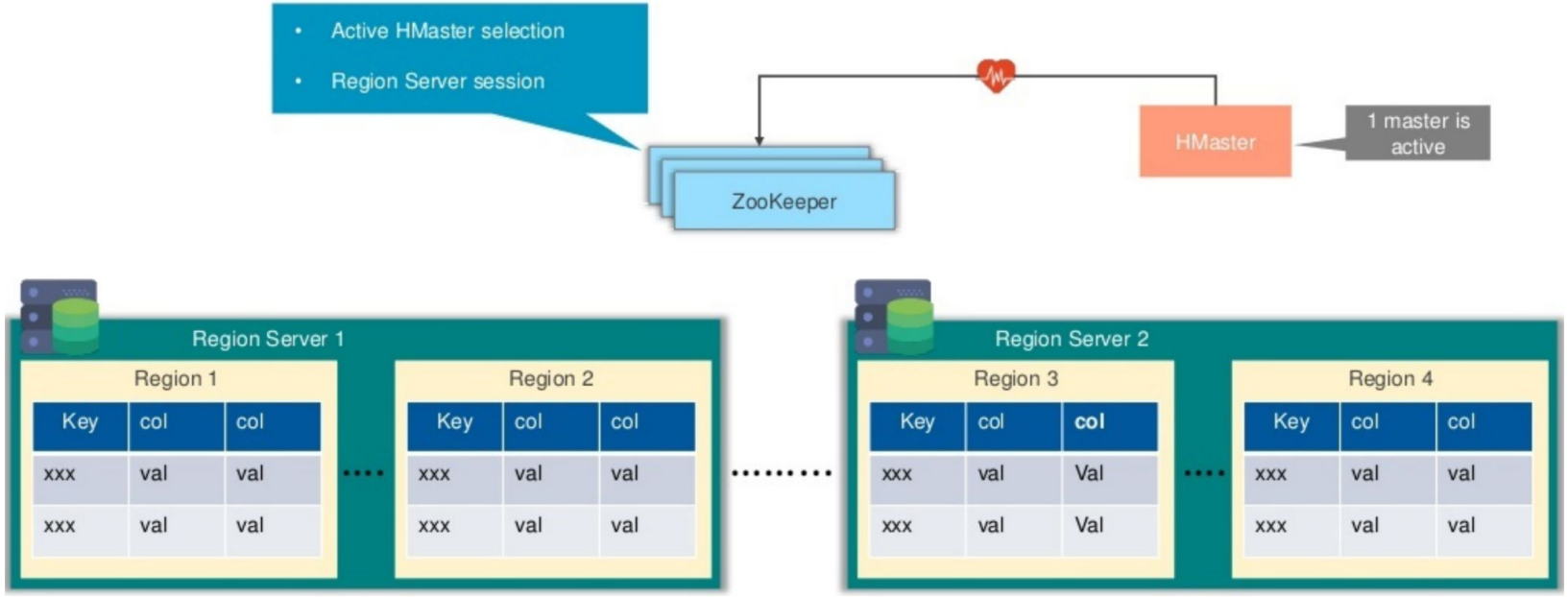
# HBase Components - Zookeeper



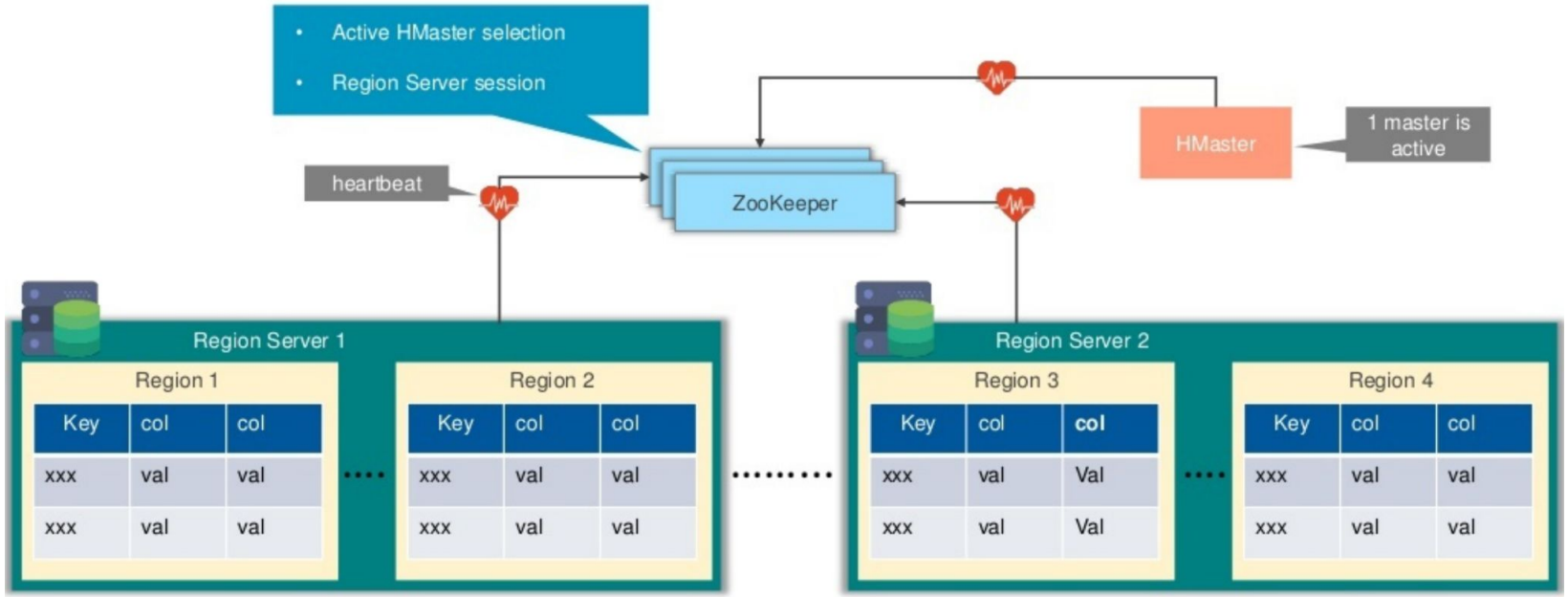
# HBase Components - Zookeeper



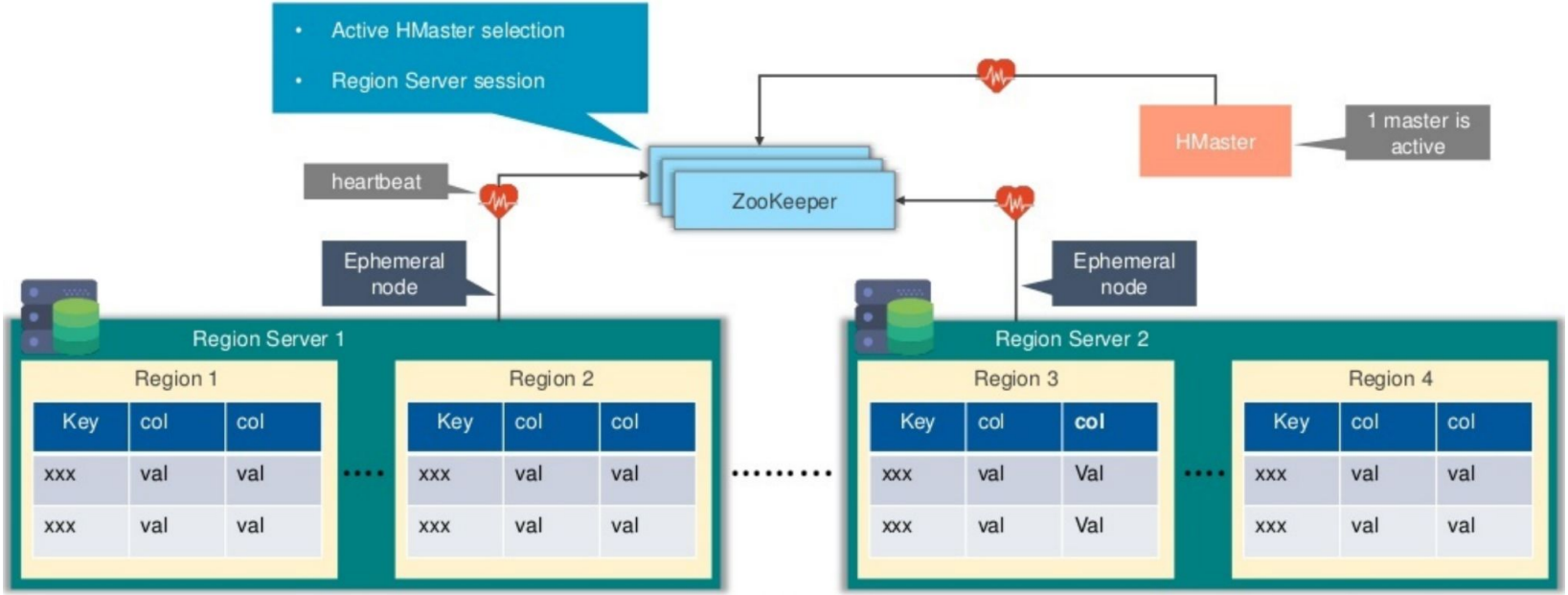
# HBase Components Work Together



# HBase Components Work Together



# HBase Components Work Together



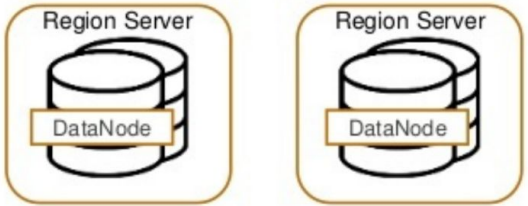
# HBase Read/Write

# HBase Read or Write

There is a special HBase Catalog table called the META table, which holds the location of the regions in the cluster

Here is what happens the first time a client reads or writes data to HBase

The client gets the Region Server that hosts the META table from ZooKeeper

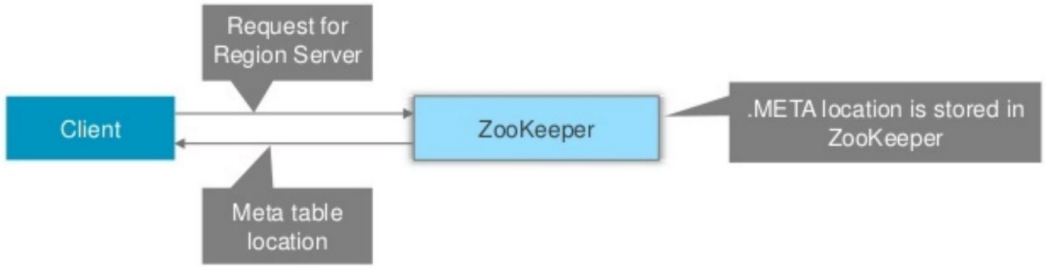



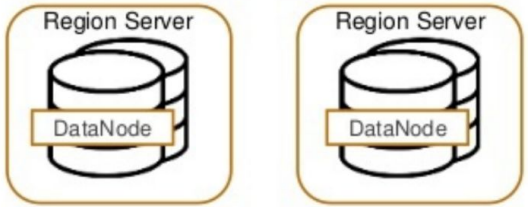
# HBase Read or Write

There is a special HBase Catalog table called the META table, which holds the location of the regions in the cluster

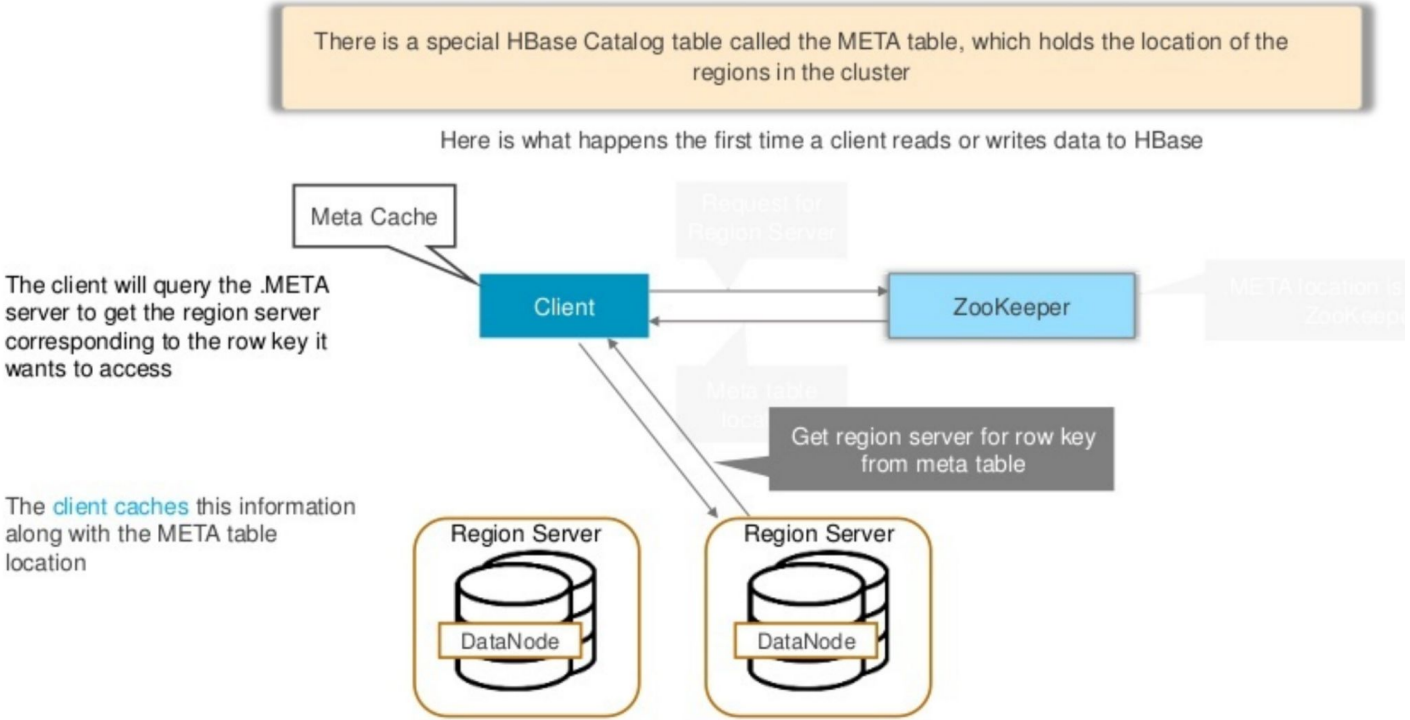
Here is what happens the first time a client reads or writes data to HBase

The client gets the Region Server that hosts the META table from ZooKeeper

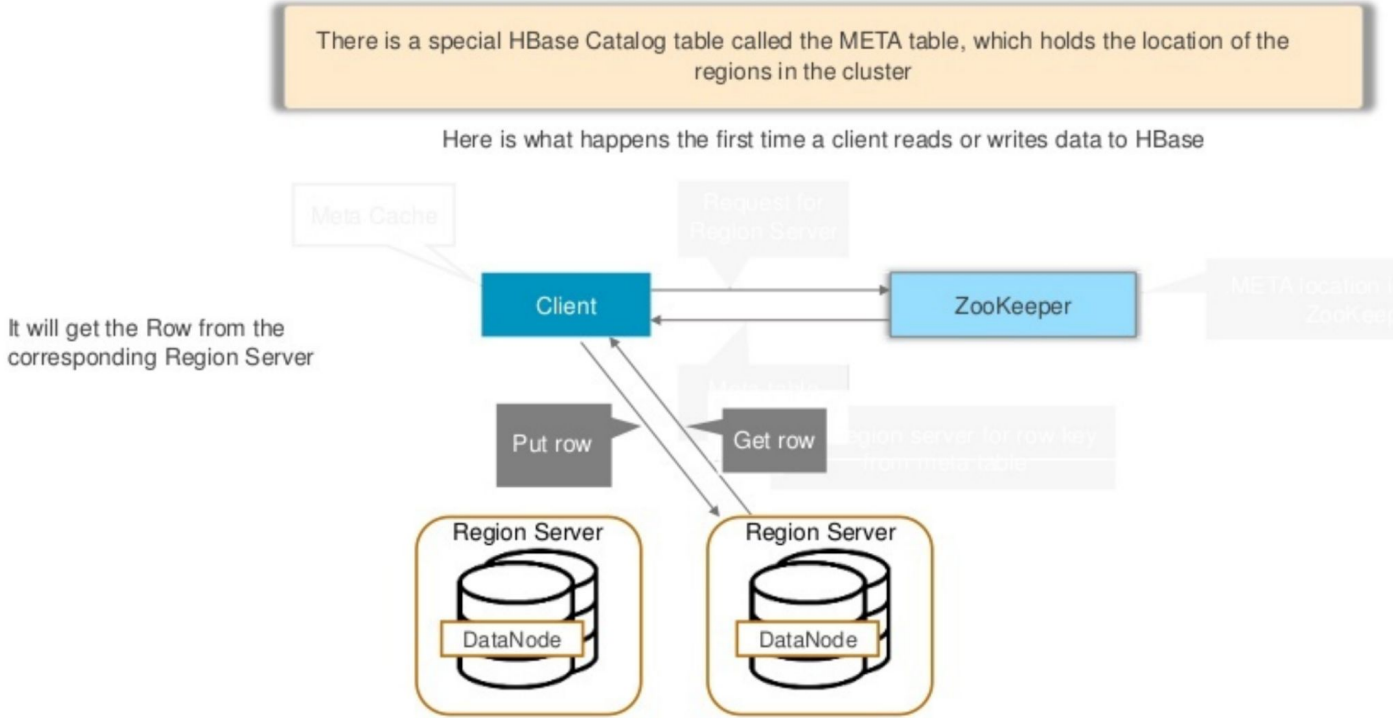


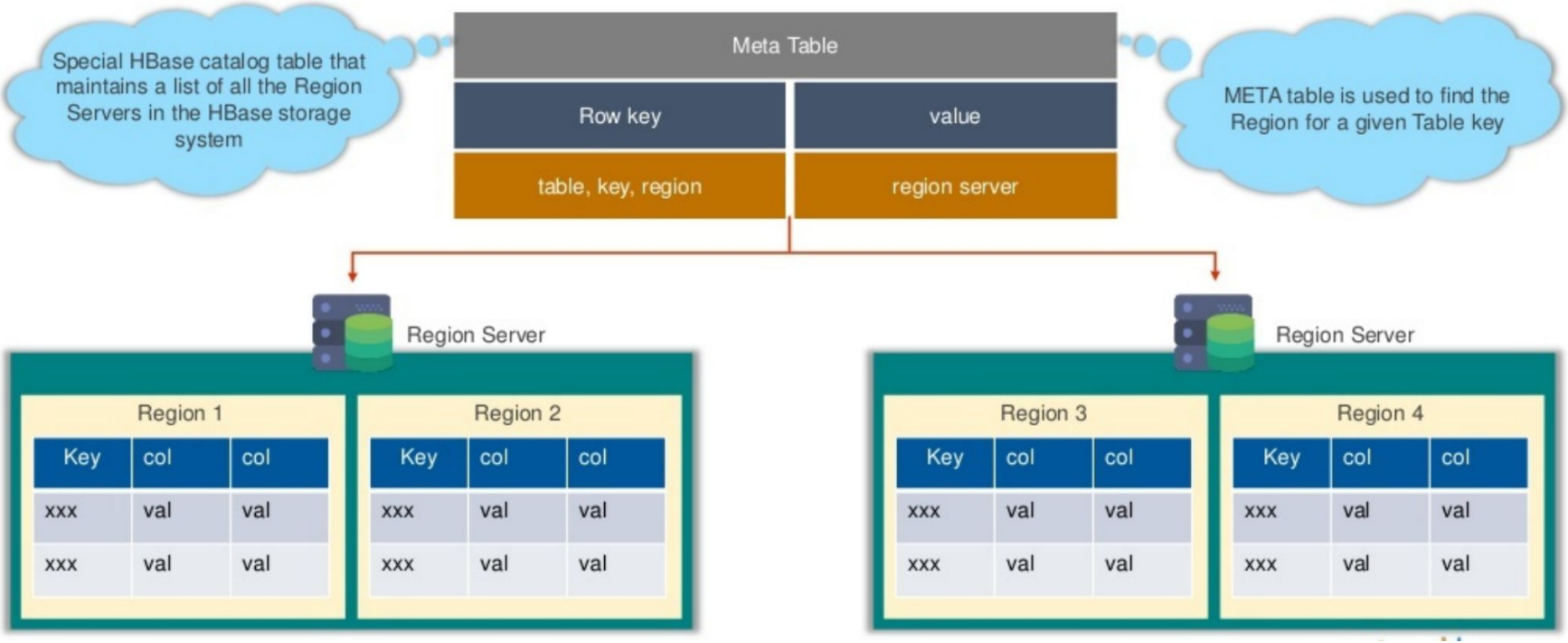
# HBase Read or Write



# HBase Read or Write

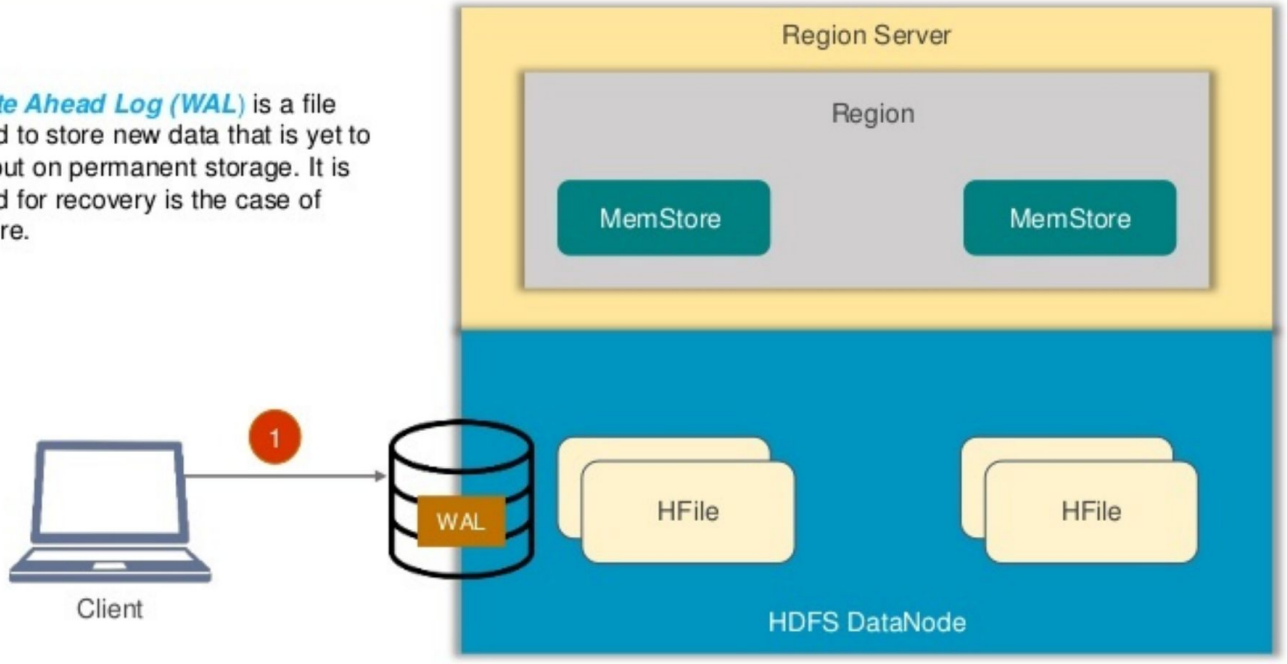


# HBase Meta Table



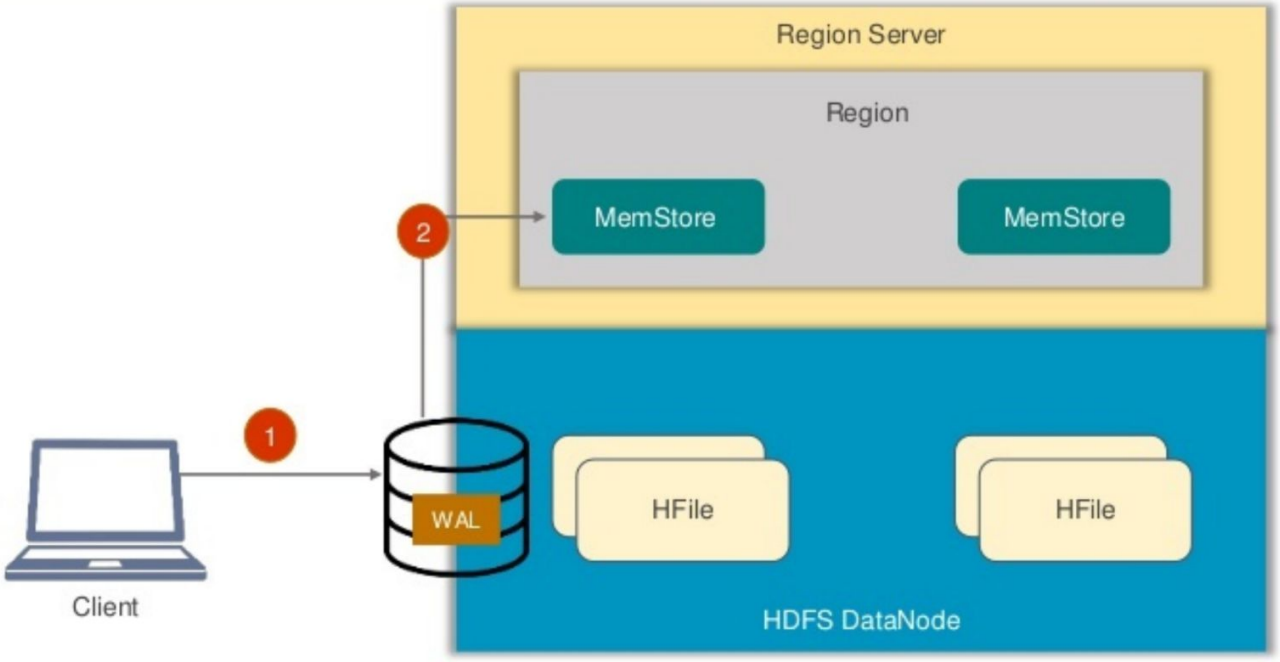
# HBase Write Mechanism

***Write Ahead Log (WAL)*** is a file used to store new data that is yet to be put on permanent storage. It is used for recovery is the case of failure.




1 When client issues a put request, it will write the data to the write-ahead log (WAL)

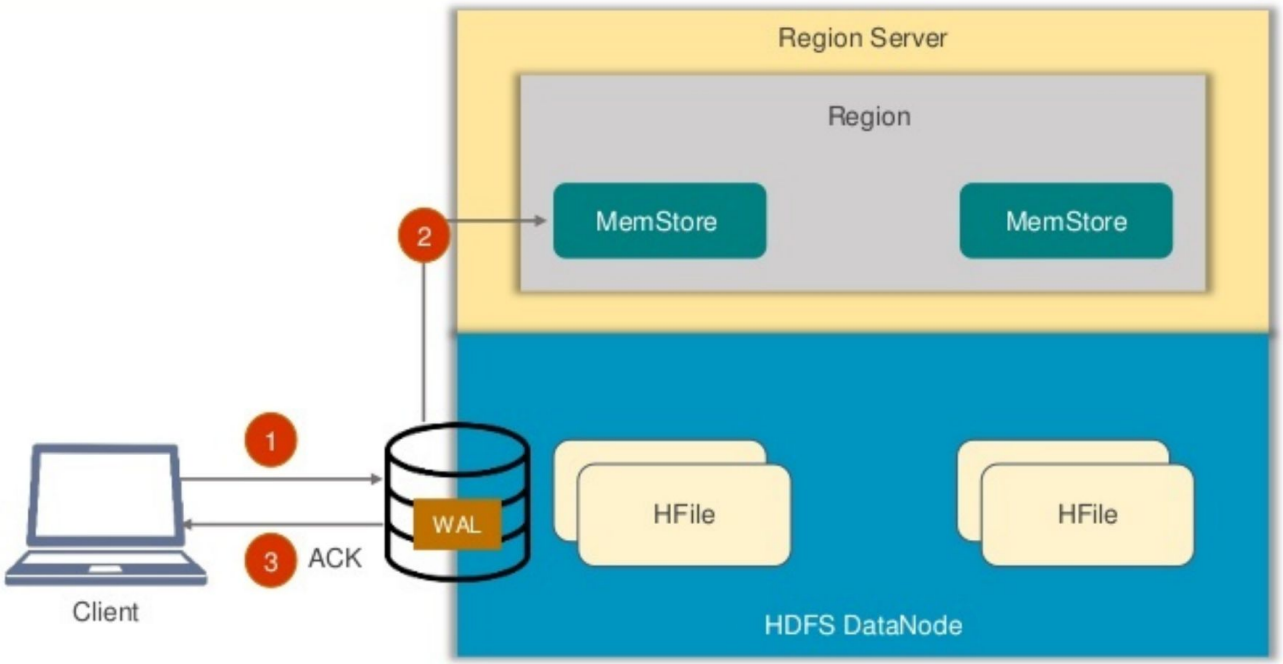
# HBase Write Mechanism




***MemStore*** is the write cache that stores new data that has not yet been written to disk. There is one MemStore per column family per region.

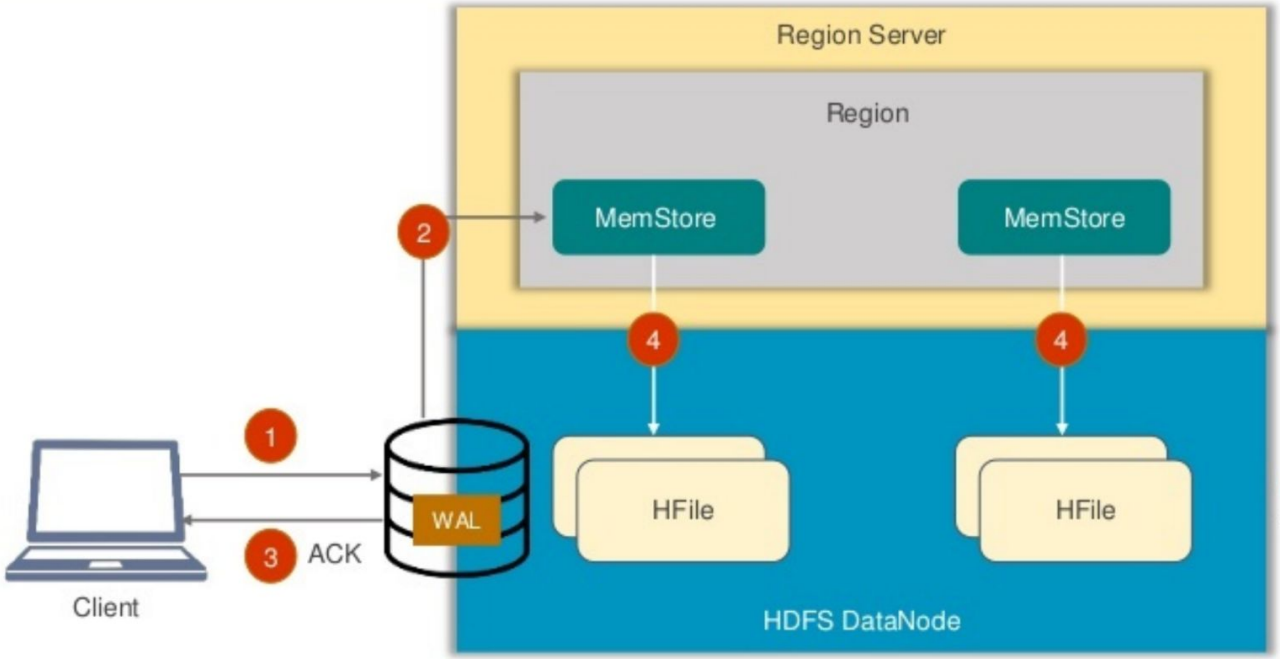
2 Once data is written to the WAL, it is then copied to the MemStore

# HBase Write Mechanism



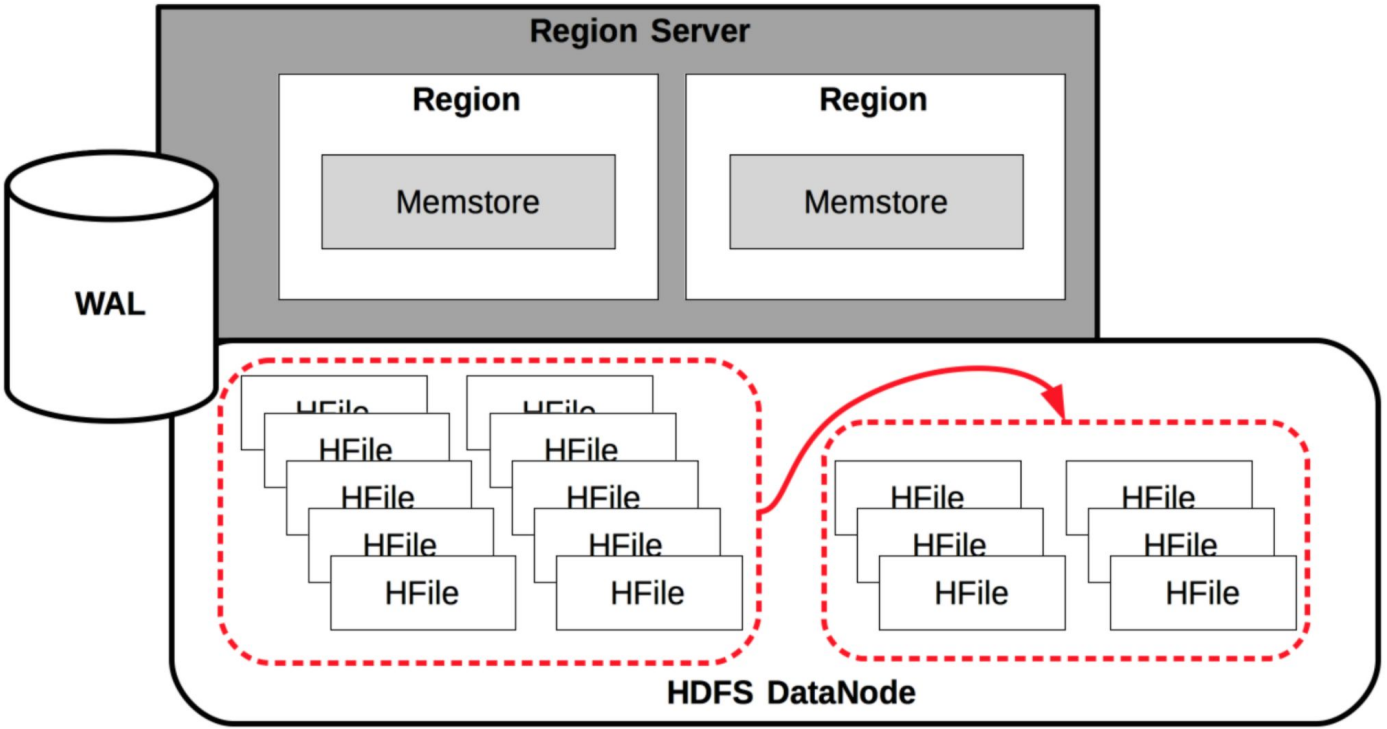

3 Once the data is placed in MemStore, the client then receives the acknowledgment

# HBase Write Mechanism

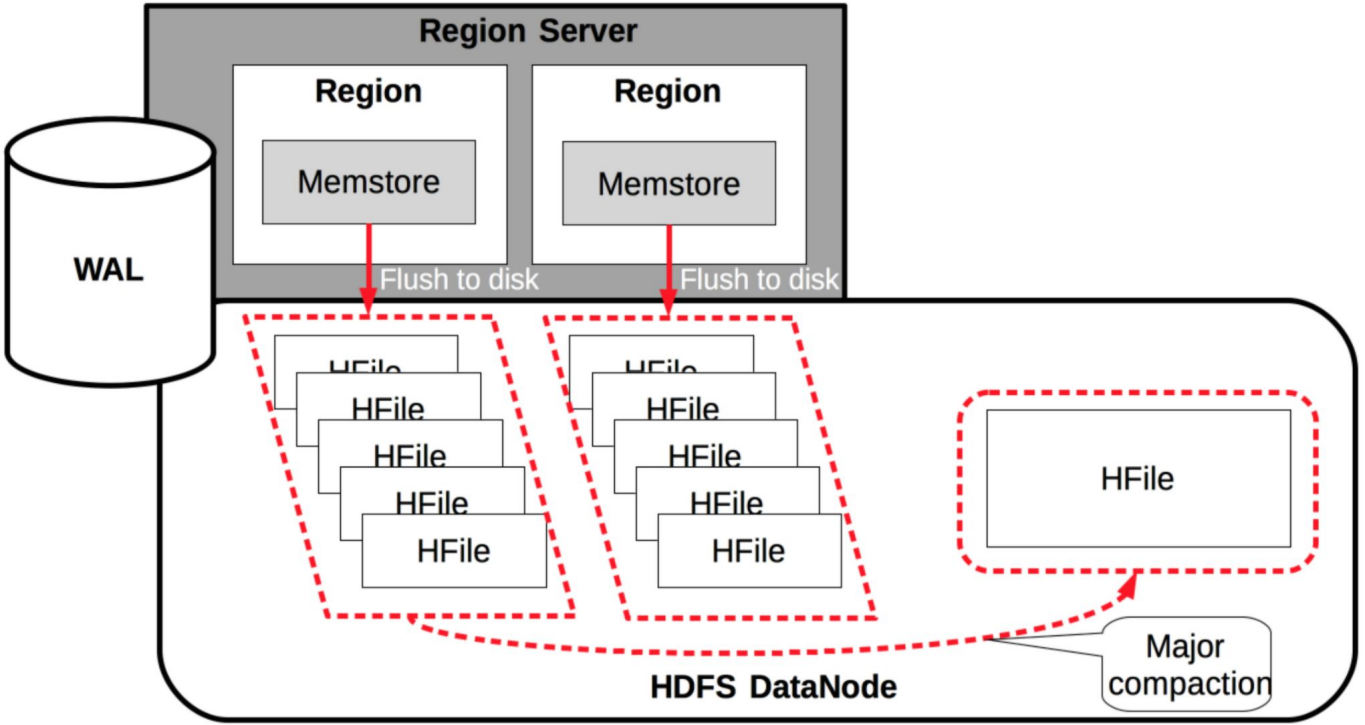



4 When the MemStore reaches the threshold, it dumps or commits the data into a HFile

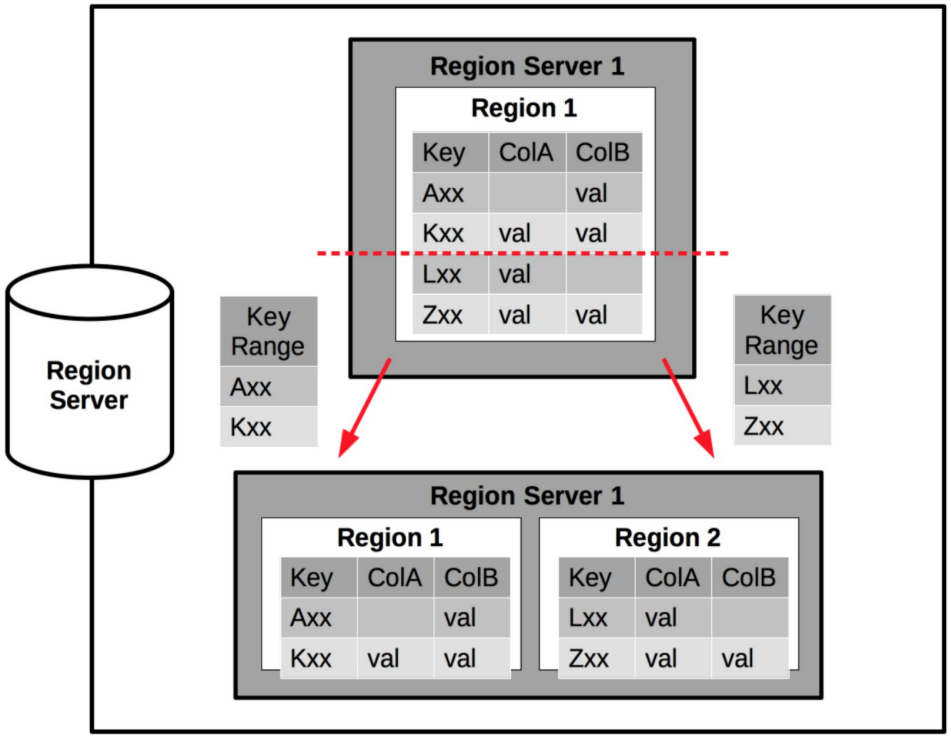
# Minor compaction



# Major Compaction

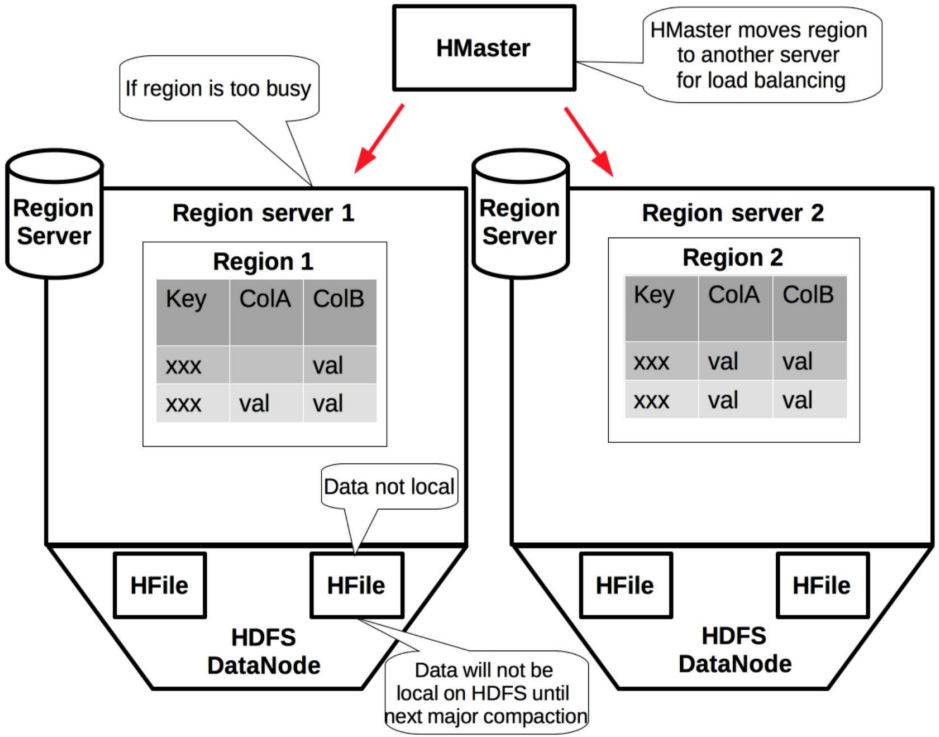


# Region Split




When region size > hbase.hregion.max.filesize -> split

# Region load balancing



# HBase Schema Design

# Elements of Schema Design

HBase schema design is QUERY based

1. Column families determination
2. RowKey design
3. Columns usage
4. Cell versions usage
5. Column family attribute: Compression, TimeToLive, Min/Max Versions, Im-Memory

# Column Families determination

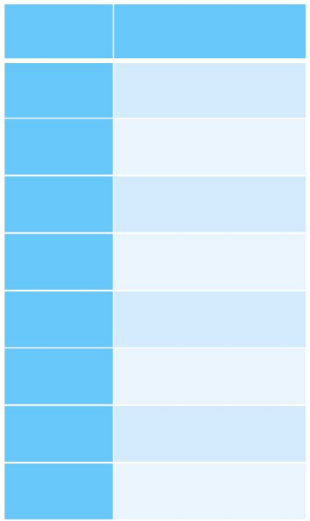
- Data, that accessed together should be stored together!
- Big number of column families may avoid performance. Optimal: ≤ 3
- Using compression may improve read performance and reduce store data size, but affect write performance

# RowKey design

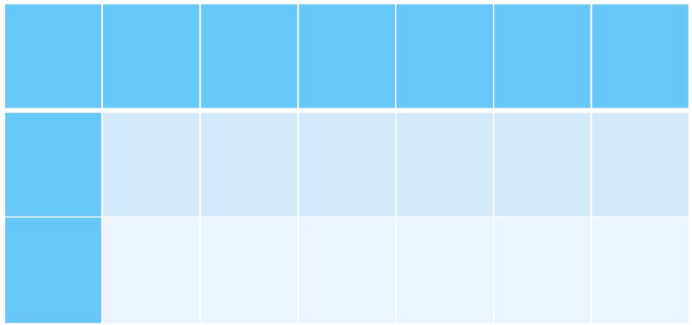
- Do not use sequential keys like timestamp
- Use hash for effective key distribution
- Use composite keys for effective scans

# Columns and Versions usage

Tall-Narrow Table



Flat-Wide Table



# Tall-Narrow Vs. Flat-Wide Tables

**Tall-Narrow provides better quality granularity**

- Finer grained RowKey
- Works well with Get

**Flat-Wide supports build-in row atomicity**

- More values in a single row
- Works well to update multiple values
- Works well to get multiple associated values

# Column Families properties

**Compression**
- LZO
- GZIP
- SNAPPY

**Time To Live (TTL)**
- Keep data for some time and then delete when TTL is passed

**Versioning**
- Keep fewer versions means less data in scans. Default now 1
- Combine MIN_VERSIONS with TTL to keep data older than TTL

**In-Memory setting**
- A setting to suggest that server keeps data in cache. Not guaranteed
- Use for small, high-access column families

# An exercise

How would you use HBase to create a webtable store snapshots of every webpage on the planet, over time?


APACHE
PHOENIX

# Phoenix - The Basics

**What is wrong with pure-HBase?**

- HBase is a powerful, flexible and extensible “engine”
- Too low level
- Have to write java code to do anything!

**Phoenix is relational layer over HBase**

- Also described as a SQL-Skin
- Looking more and more like a generic SQL engine

**Why not Hive / Spark SQL / other SQL-over-Hadoop**

- OTLP versus OLAP
- As fast as HBase, 1 ms query, 10K-1M qps

# Why SQL?

Accessing HBase data with Phoenix can be substantially easier than direct HBase API use

```
SELECT * FROM foo WHERE bar > 30
```

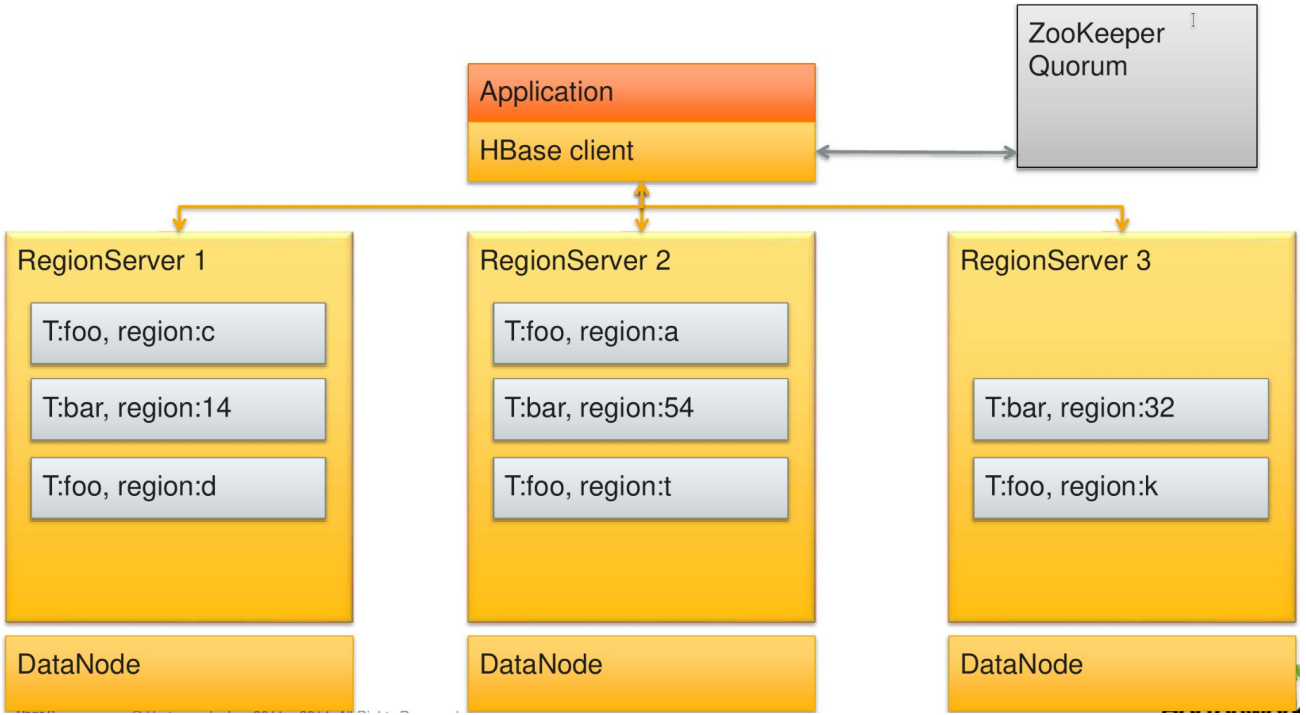
Versus

```
HTable t = new HTable("foo");
RegionScanner s = t.getScanner(new Scan(…,
    new ValueFilter(CompareOp.GT,
        new CustomTypedComparator(30)), …));
while ((Result r = s.next()) != null) {
     // blah blah blah Java Java Java
}
s.close();
t.close();
```

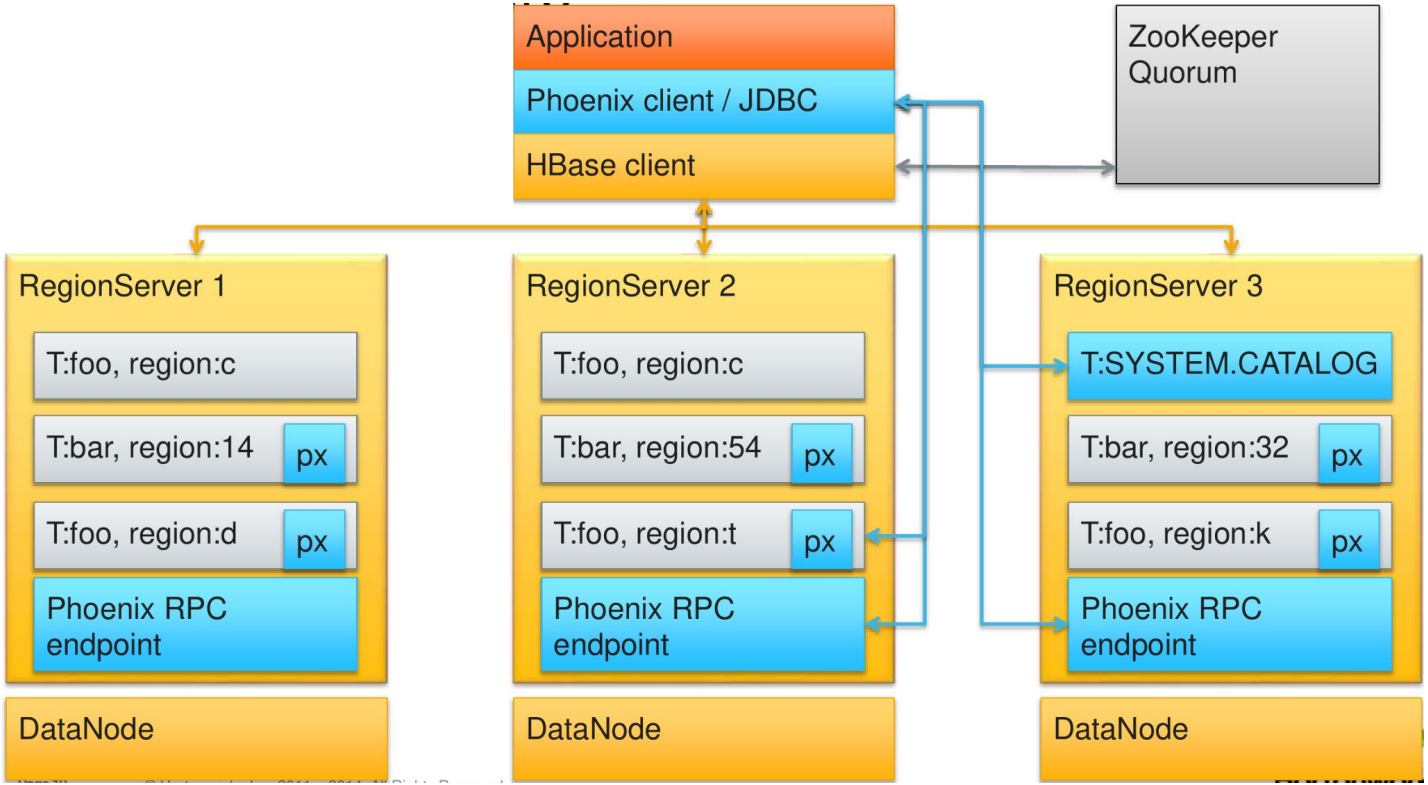
Your BI tool probably can't do this

*(And we didn't include error handling…)*

# HBase Architecture



# Phoenix Architecture

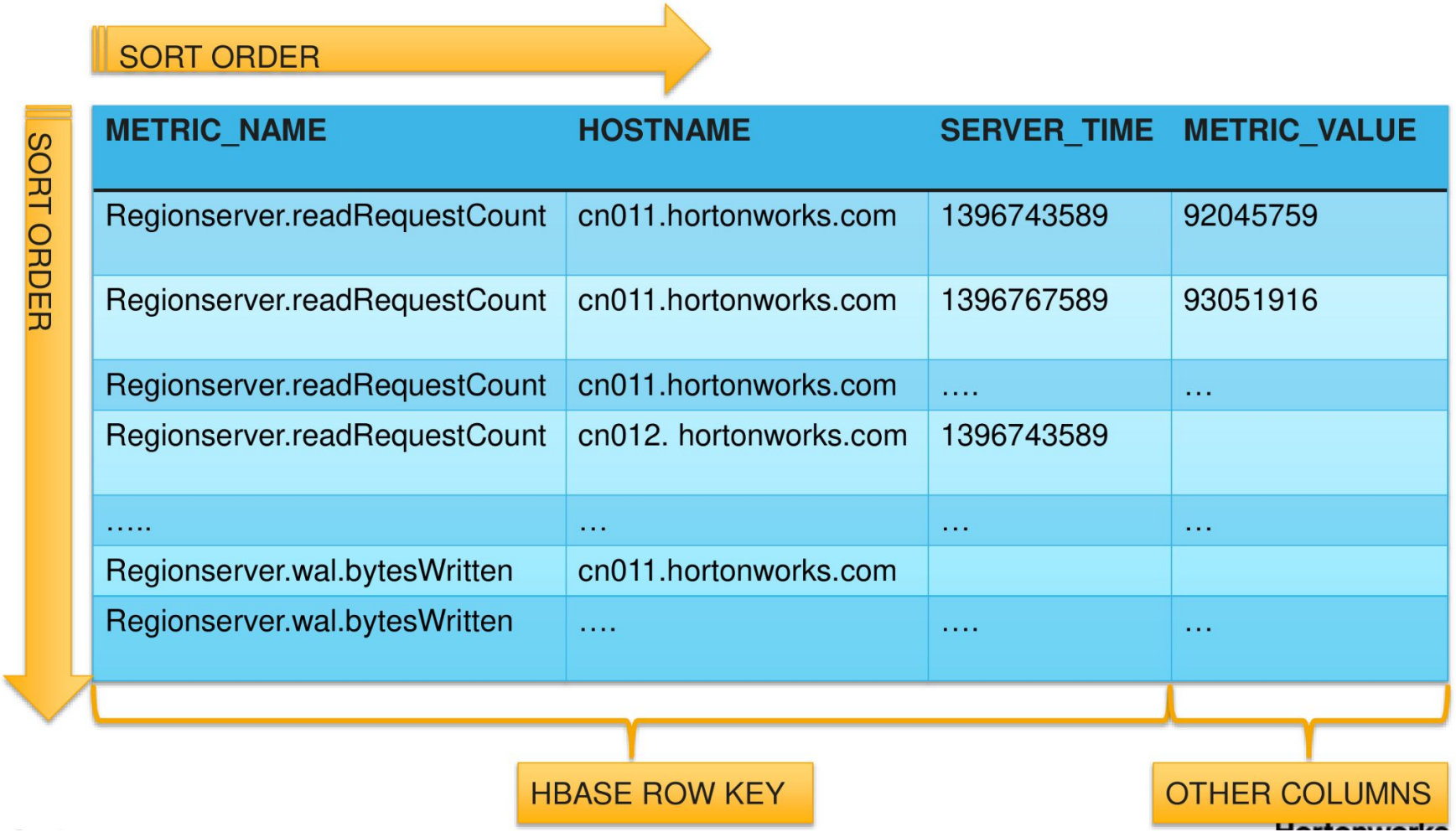


# Phoenix Goodies

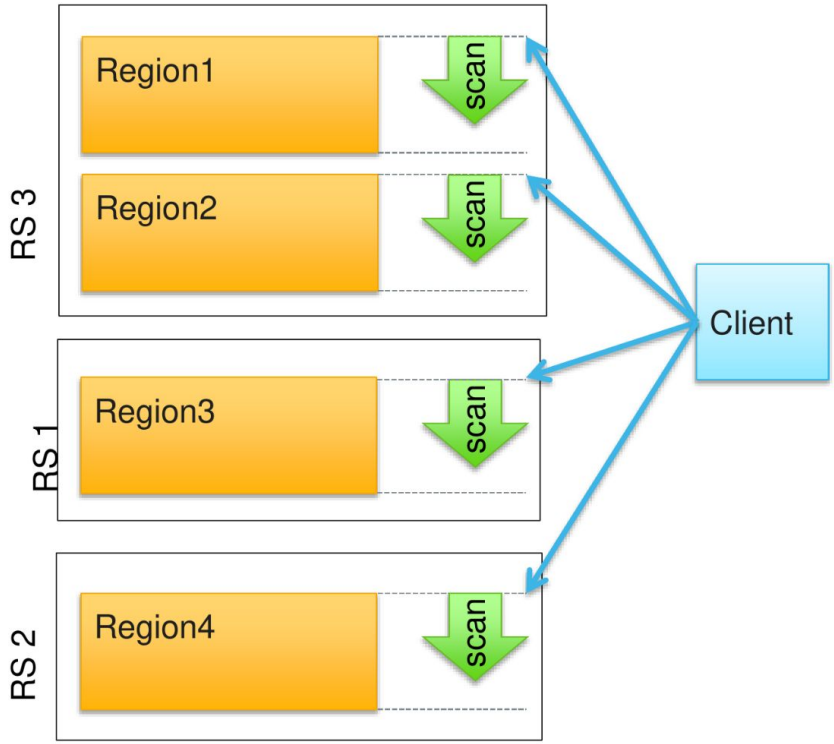
- SQL DataTypes
- Schemas / DDL / HBase table properties
- Composite Types (Composite Primary Key)
- Map existing HBase tables
- Write from HBase, read from Phoenix
- Salting
- Parallel Scan
- Skip scan
- Filter push down
- Statistics Collection / Guideposts

# DDL Example

```
CREATE TABLE IF NOT EXISTS METRIC_RECORD (
      METRIC_NAME VARCHAR,
      HOSTNAME VARCHAR,
      SERVER_TIME UNSIGNED_LONG NOT NULL
      METRIC_VALUE DOUBLE,
      ...
      CONSTRAINT pk PRIMARY KEY (METRIC_NAME, HOSTNAME,SERVER_TIME)
)
DATA_BLOCK_ENCODING='FAST_DIFF', TTL=604800,
COMPRESSION='SNAPPY'
SPLIT ON ('a', 'k', 'm');
```


SORT ORDER
SORT ORDER
METRIC_NAME
HOSTNAME
SERVER_TIME
METRIC_VALUE
Regionserver.readRequestCount
cn011.hortonworks.com
1396743589
92045759
Regionserver.readRequestCount
cn011.hortonworks.com
1396767589
93051916
Regionserver.readRequestCount
cn011.hortonworks.com
….
…
Regionserver.readRequestCount
cn012. hortonworks.com
1396743589
…..
…
…
…
Regionserver.wal.bytesWritten
cn011.hortonworks.com
Regionserver.wal.bytesWritten
….
….
…
HBASE ROW KEY
OTHER COLUMNS

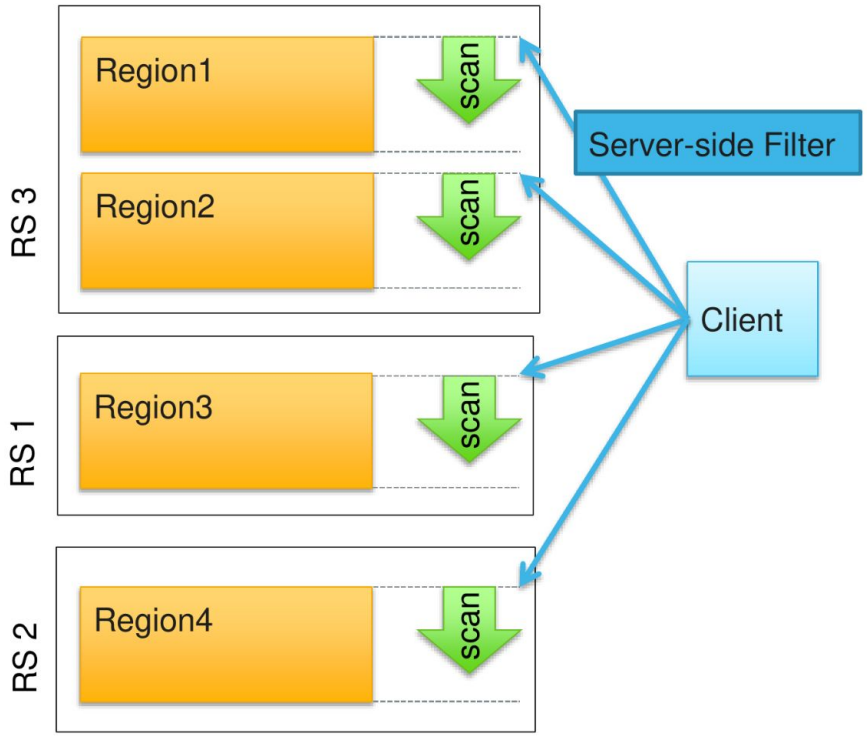
# Parallel Scan




```
SELECT * FROM METRIC_RECORD;
```

```
CLIENT 4-CHUNK PARALLEL 1-WAY
FULL SCAN OVER METRIC_RECORD
```

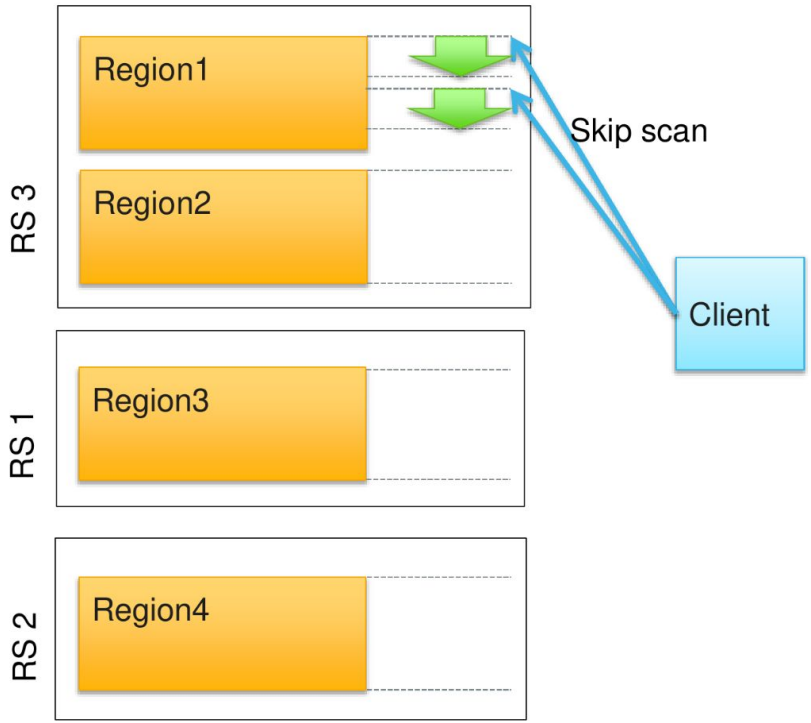
# Filter Push Down




```
SELECT * FROM METRIC_RECORD
WHERE SERVER_TIME > NOW() - 7;
```

```
CLIENT 4-CHUNK PARALLEL 1-WAY
FULL SCAN OVER METRIC_RECORD
    SERVER FILTER BY
    SERVER_TIME > DATE
    '2016-04-06 09:09:05.978'
```

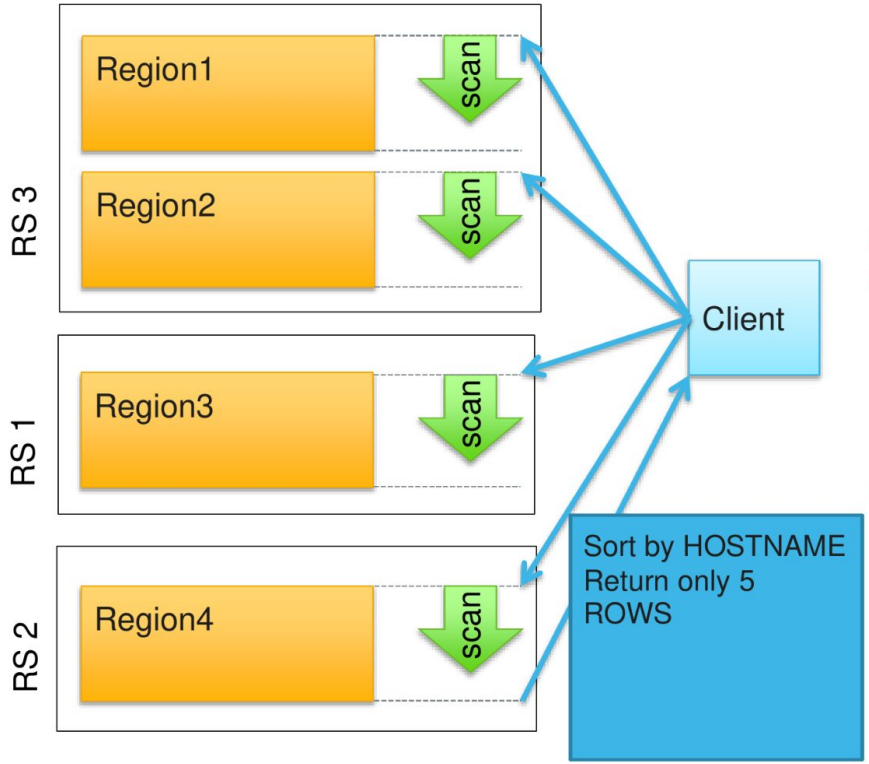
# Skip Scan




```
SELECT * FROM METRIC_RECORD
WHERE METRIC_NAME LIKE 'abc%'
  AND HOSTNAME in ('host1',
'host2');
```

```
CLIENT 1-CHUNK PARALLEL 1-WAY SKIP
SCAN ON 2 RANGES OVER
METRIC_RECORD ['abc','host1'] -
['abd','host2']
```

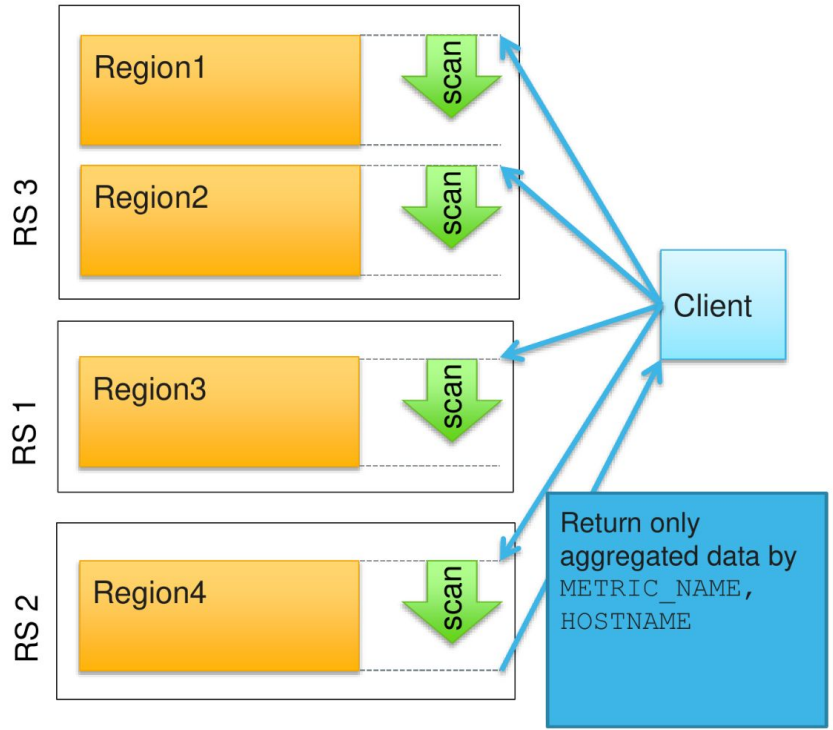
# TopN




```
SELECT * FROM METRIC_RECORD
WHERE SERVER_TIME > NOW() - 7
ORDER BY HOSTNAME LIMIT 5;

CLIENT 4-CHUNK PARALLEL 4-WAY FULL
SCAN OVER METRIC_RECORD
    SERVER FILTER BY SERVER_TIME > …
    SERVER TOP 5 ROWS SORTED BY
        [HOSTNAME]
CLIENT MERGE SORT
```

# Aggregation




```
SELECT METRIC_NAME, HOSTNAME,
AVG(METRIC_VALUE)
FROM METRIC_RECORD
WHERE SERVER_TIME > NOW() - 7
GROUP BY METRIC_NAME, HOSTNAME
ORDER BY METRIC_NAME, HOSTNAME;
```

```
CLIENT 4-CHUNK PARALLEL 1-WAY FULL
SCAN OVER METRIC_RECORD
    SERVER FILTER BY SERVER_TIME > …
    SERVER AGGREGATE INTO ORDERED
      DISTINCT ROWS BY
      [METRIC_NAME, HOSTNAME]
CLIENT MERGE SORT
```

# Joins and subqueries in Phoenix

**Grammar**

- Inner, Left, Right, Full outer join, Cross join
- Semi-join / Anti-join

**Algorithms**

- Hash-join, sort-merge join
- Hash-join table is computed and pushed to each regionserver from client

**Optimizations**

- Predicate push-down
- PK-to-FK join optimization
- Global index with missing columns
- Correlated query rewrite

# Joins and subqueries in Phoenix

- Phoenix can execute most of TPC-H queries!
- No nested loop join
- With Calcite support, more improvements soon
- No statistical Guided join selection yet
- Not very good at executing very big joins
  - No generic YARN / Tez execution layer
  - But Hive / Spark support for generic DAG execution

# Secondary Indexes

**HBase table is a sorted map**

- Everything in HBase is sorted in primary key order
- Full or partial scans in sort order is very efficient in HBase
- Sort data differently with secondary index dimensions

**Two types**

- Global index
- Local index

**Query**

- Indexes are “covered”
- Indexes are automatically selected from queries
- Only covered columns are returned from index without going back to data table

# Global and Local Index

**Global Index**

- A single instance for all table data in a
- different sort order
- A different HBase table per index
- Optimized for read-heavy use cases
- Can be one edit “behind” actual primary
- data
- Transactional tables indices have ACID
- guarantees
- Different consistency / durability for
- mutable / immutable tables

**Local Index**

- Multiple mini-instances per region
- Uses same HBase table, different cf
- Optimized for write-heavy use cases
- Atomic commit and visibility (coming soon)
- Queries have to ask all regions for relevant data from index

# Row Timestamps

A pseudo-column for HBase native timestamps (versions)

Enables setting and querying cell timestamps

Perfect for time-series use cases

- Combine with FIFO / Date Tiered Compaction policies
- And HBase scan file pruning based on min-max ts for very efficient scans

```
CREATE TABLE METRICS_TABLE (

CREATED_DATE NOT NULL DATE,

METRIC_ID NOT NULL CHAR(15), METRIC_VALUE LONG

CONSTRAINT PK PRIMARY KEY(CREATED_DATE ROW_TIMESTAMP,

METRIC_ID)) SALT_BUCKETS = 8;
```

# Salted Tables

- HBase tables can develop “hot spots” when writing data with monotonically increasing row keys
  - HBase RegionServers serve regions of data, which are ranges of lexiographically sorted rows
  - Region hosting is exclusive, load can fall all onto one server
- Phoenix can “salt” keys into N buckets such that writes  fan out N-ways even with monotonic primary keys
- For best results, N should approximate the number of  RegionServers in the HBase cluster

```
CREATE TABLE T (K VARCHAR PRIMARY
KEY, ...)
SALT_BUCKETS=32;
```

# Demo

```
CREATE TABLE IF NOT EXISTS us_population (
      state CHAR(2) NOT NULL,
      city VARCHAR NOT NULL,
      population BIGINT
      CONSTRAINT my_pk PRIMARY KEY (state, city)
);
```

# Demo

```
UPSERT INTO us_population(state, city, population) VALUES('NY','New York',8143197);
UPSERT INTO us_population(state, city, population) VALUES('CA','Los Angeles',3844829);
UPSERT INTO us_population(state, city, population) VALUES('IL','Chicago',2842518);
UPSERT INTO us_population(state, city, population) VALUES('TX','Houston',2016582);
UPSERT INTO us_population(state, city, population) VALUES('PA','Philadelphia',1463281);
UPSERT INTO us_population(state, city, population) VALUES('AZ','Phoenix',1461575);
UPSERT INTO us_population(state, city, population) VALUES('TX','San Antonio',1256509);
UPSERT INTO us_population(state, city, population) VALUES('CA','San Diego',1255540);
UPSERT INTO us_population(state, city, population) VALUES('TX','Dallas',1213825);
UPSERT INTO us_population(state, city, population) VALUES('CA','San Jose',912332);
```

# Demo

```
SELECT state as "State",count(city) as "City
Count",sum(population) as "Population Sum"

FROM us_population

GROUP BY state

ORDER BY sum(population) DESC;
```

# Complete SQL Reference

https://phoenix.apache.org/language/index.html

# Exercise: User Information

1. Create a table based on the side
2. Insert table data
3. Add an educational background column family, and educational backgrounds and titles to the user information table.
4. Query user names and addresses by user ID.
5. Query information by user name.

| SN | Name | Gender | Age | Address |
|---|---|---|---|---|
| 12005000201 | Tom | Male | 19 | Shenzhen, Guangdong |
| 12005000202 | Li Wanting | Female | 23 | Shijiazhuang, Hebei |
| 12005000203 | Wang Ming | Male | 26 | Ningbo, Zhejiang |
| 12005000204 | Li Gang | Male | 18 | Xiangyang, Hubei |
| 12005000205 | Zhao Enru | Female | 21 | Shangrao, Jiangxi |
| 12005000206 | Chen Long | Male | 32 | Zhuzhou, Hunan |
| 12005000207 | Zhou Wei | Female | 29 | Nanyang, Henan |
| 12005000208 | Yang Yiwen | Female | 30 | Kaixian, Chongqing |
| 12005000209 | Xu Bing | Male | 26 | Weinan, Shaanxi |
| 12005000210 | Xiao Kai | Male | 25 | Dalian, Liaoning |