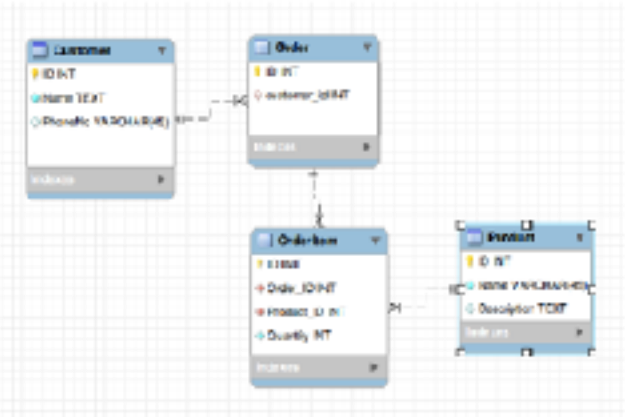
Why do we need NoSQL / HBase?

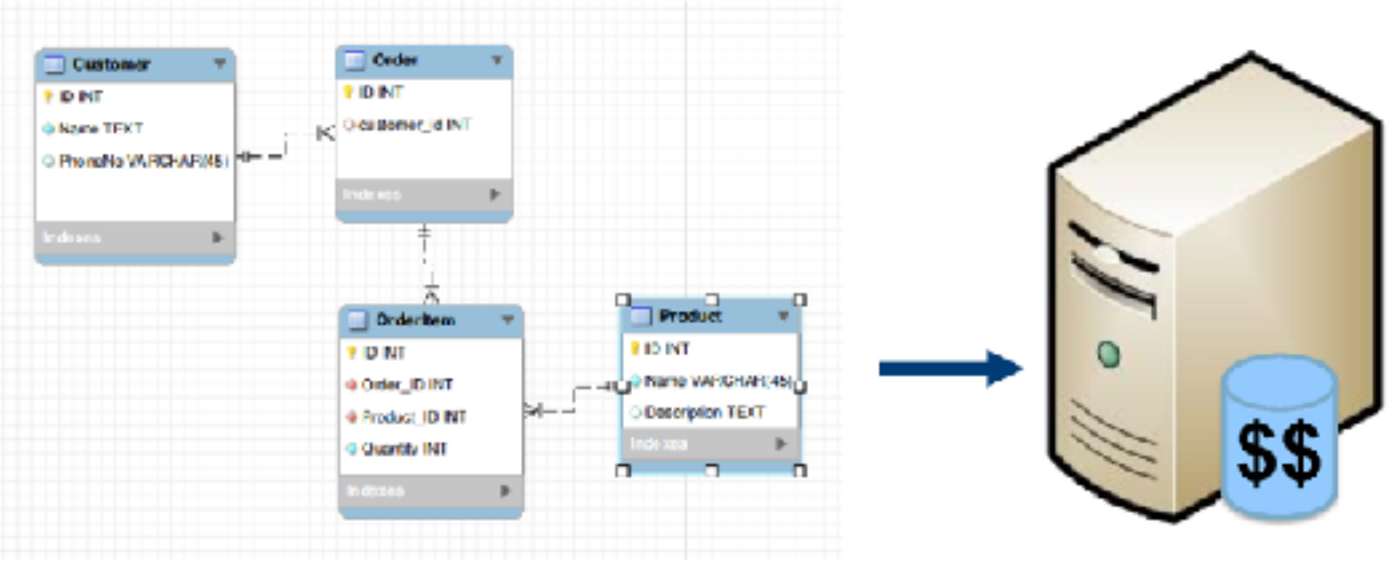
Relational Model

- Pros
  - **Standard** persistence model
  - **Transactions handle**
    - **concurrency , consistency**
  - efficient and robust structure for storing data



Relational Databses vs HBase - Scaling

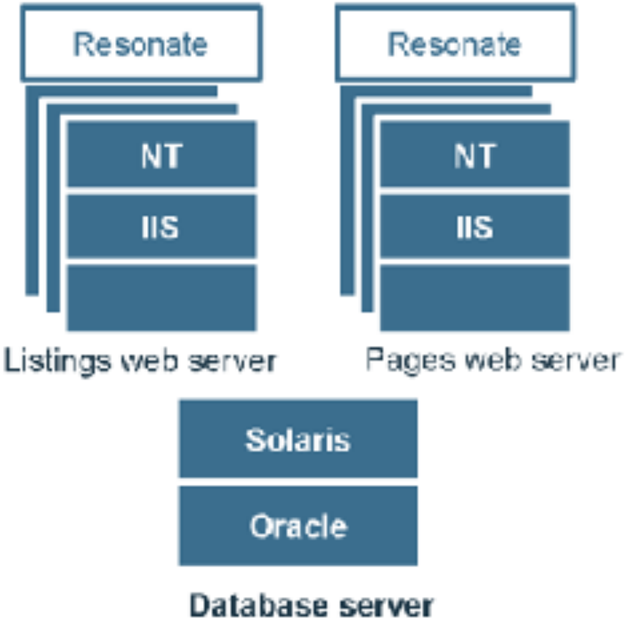
RDBMS - Scale UP approach



Vertical scale = **big box**

RDBMS Scaling Up Example - eBay

- Back End Oracle Database server **scaled vertically to a larger machine** (Sun E10000)



Resonate

NT

IIS

Listings web server

Resonate

NT

IIS

Pages web server

Solaris

Oracle

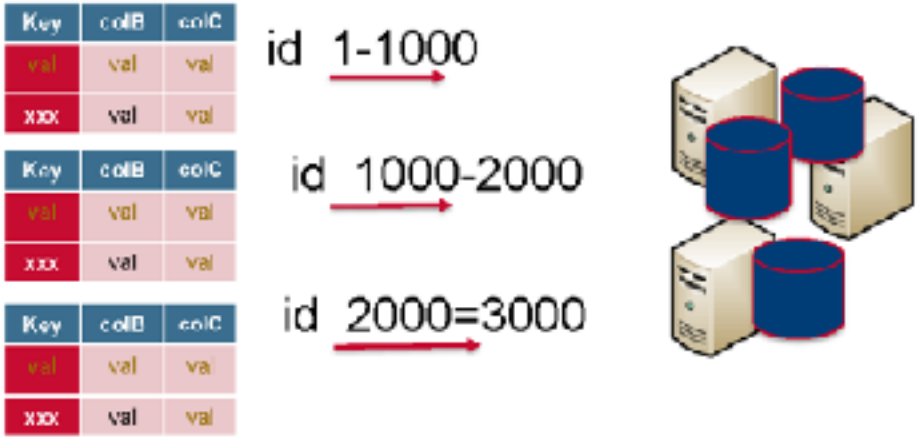
Database server

http://www.addsimplicity.com/downloads/eBaySDForum2006-11-29.pdf

What changed to bring on NoSQL?

Lots of data, the need to scale horizontally

Horizonal scale: split table by rows into partitions across a cluster



Key	colB	colC
val	val	val
xxx	val	val

id 1-1000

Key	colB	colC
val	val	val
xxx	val	val

id 1000-2000

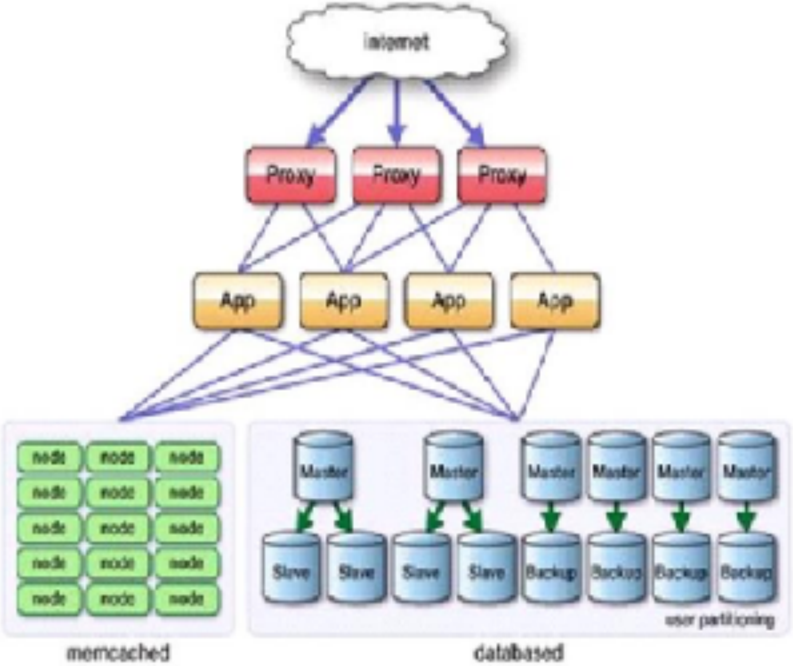
Key	colB	colC
val	val	val
xxx	val	val

id 2000=3000

- **Horizontal** scaling
  - Cheaper than vertical
  - parallel execution
- Relational databases were **not designed to do this** automatically

Facebook 2010

- 9000 memcache instances
- **4000 Shards mysql**

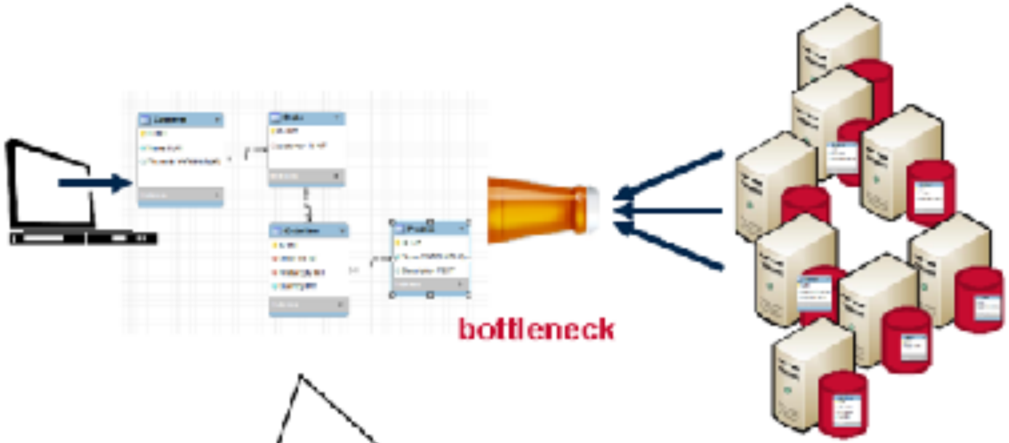


http://gigaom.com/2011/07/07/facebook-trapped-in-mysql-fate-worse-than-death/

Relational Databases vs. HBase – Data Storage Model

Storage Model

RDBMS



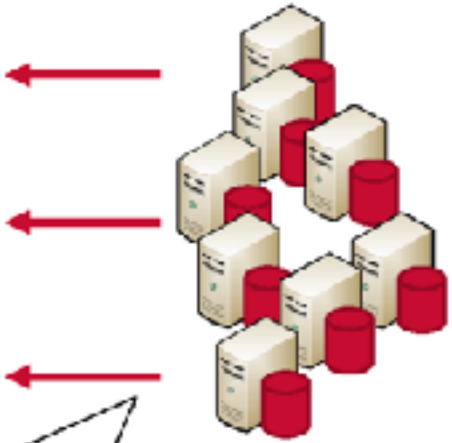
Distributed Joins, Transactions **do not scale**

HBase

Key	colB	colC
val	val	val
xxx	val	val

Key	colB	colC
val	val	val
xxx	val	val

Key	colB	colC
val	val	val
xxx	val	val



Data that is accessed together is stored together

Hbase designed for Distribution, Scale, Speed

Google

Google

Big Table

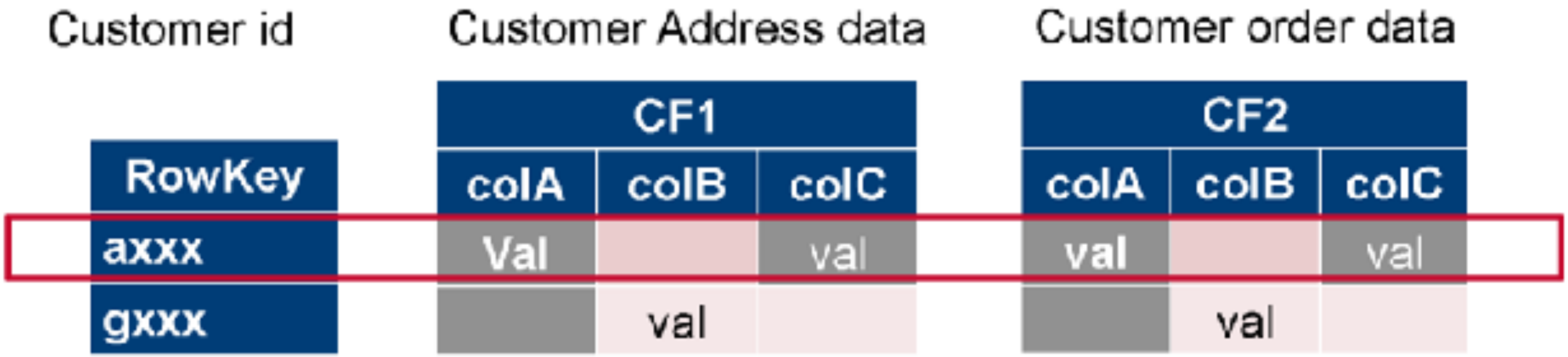
- Distributed Storage System
- Paper published in 2006.

Google File System

MapReduce

Runs on commodity hardware
Designed to Scale

HBase is a ColumnFamily oriented Database

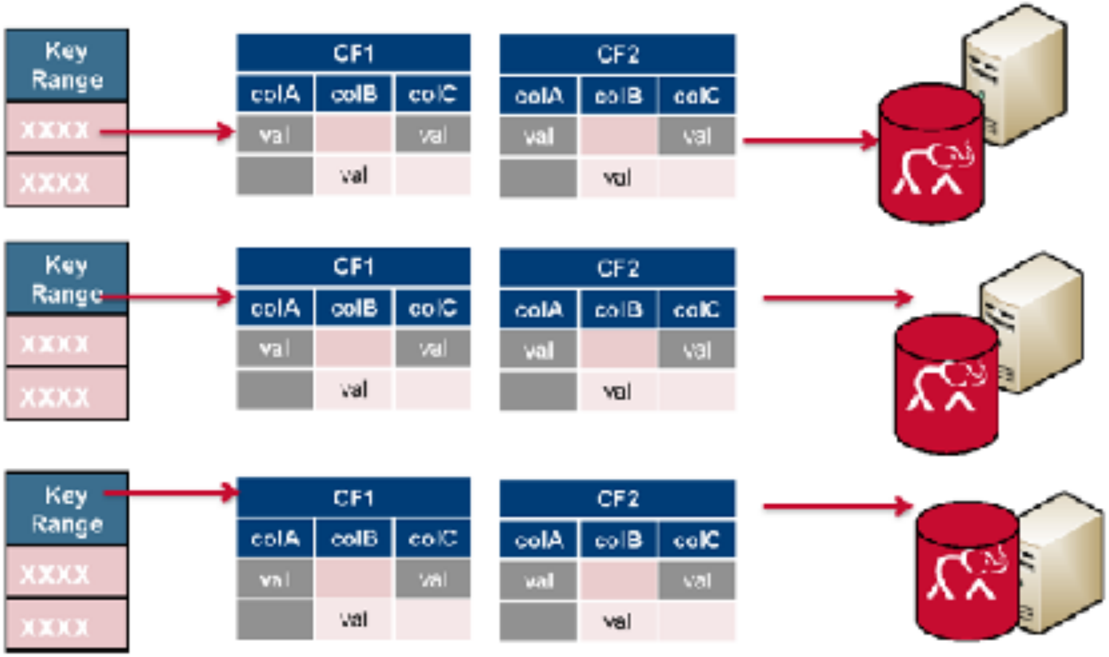


Customer id	Customer Address data			Customer order data		
	CF1			CF2		
RowKey	colA	colB	colC	colA	colB	colC
axxx	Val		val	val		val
gxxx		val			val	

Data is accessed and stored together:

- RowKey is the primary index
- Column Families group similar data by row key

HBase is a Distributed Database



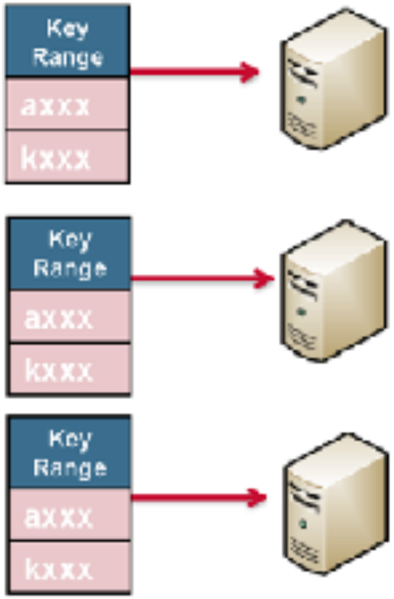
Data is automatically distributed across the cluster

- **Key range** is used for **horizontal partitioning**

Column Family Databases

distributed data stored and accessed together:

- Pros
  - **scalable**
  - **Fast Writes and Reads**
- Cons
  - **No** joins
  - No dynamic queries
  - Need to **know** how data will be **queried in advance**



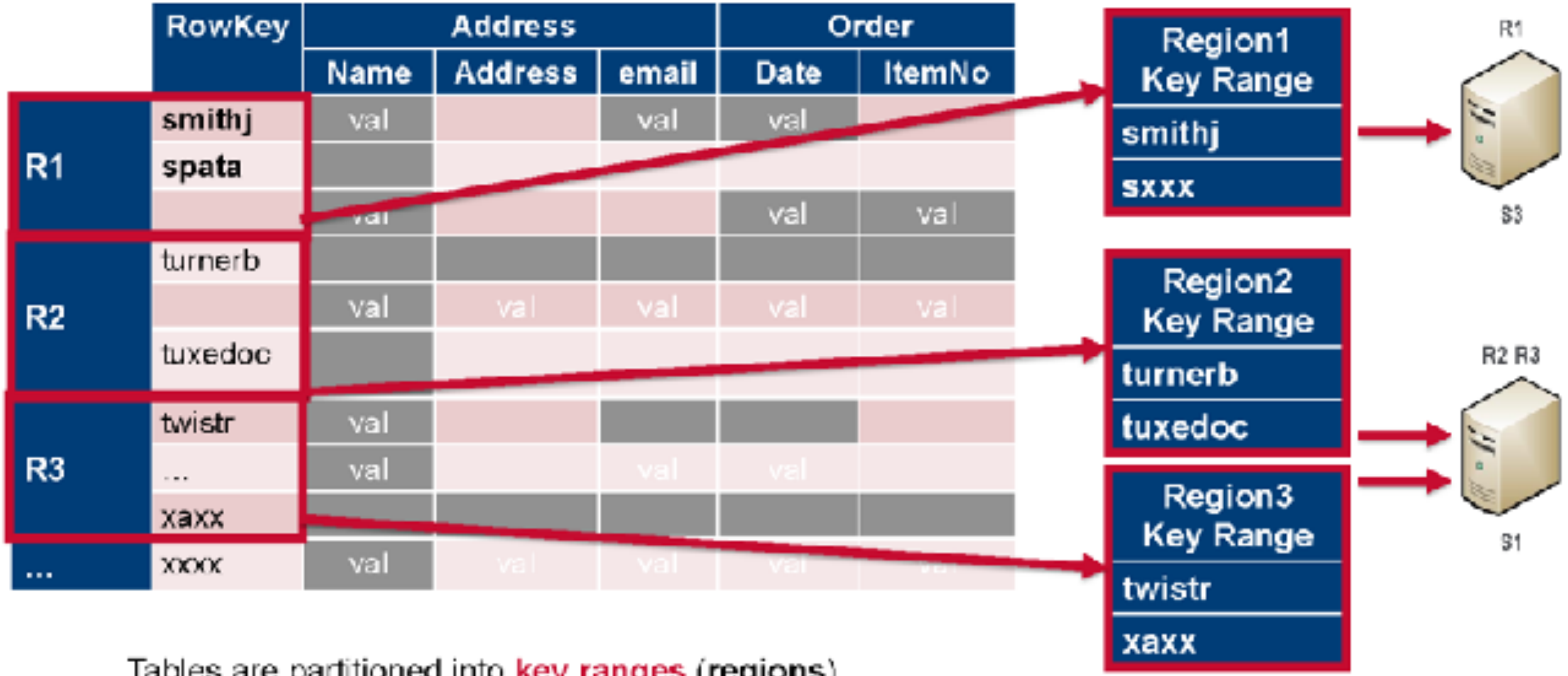
Hbase Data Model

HBase Data Model- Row Keys

RowKey	Address			Order				...
	street	city	state	Date	ItemNo	Ship Address	Cost	
smithj	val		val	val			val	
spata								
sxxxx	val			val	val	val		
...								
turnerb	val	val	val	val	val	val	val	
---								
	val							
twlstr	val		val	val			val	
---								
zaxx	val	val	val	val	val	val		
zxxx	val						val	

Row Keys: identify the rows in an HBase table.

Tables are split into Regions = contiguous keys



	RowKey	Address			Order	
		Name	Address	email	Date	ItemNo
R1	smithj	val		val	val	
	spata					
		val			val	val
R2	turnerb					
		val	val	val	val	val
	tuxedoc					
R3	twistr	val				
	...	val		val	val	
	xaxx					
...	xxxx	val	val	val	val	val

Region1 Key Range

smithj

sxxx

Region2 Key Range

turnerb

tuxedoc

Region3 Key Range

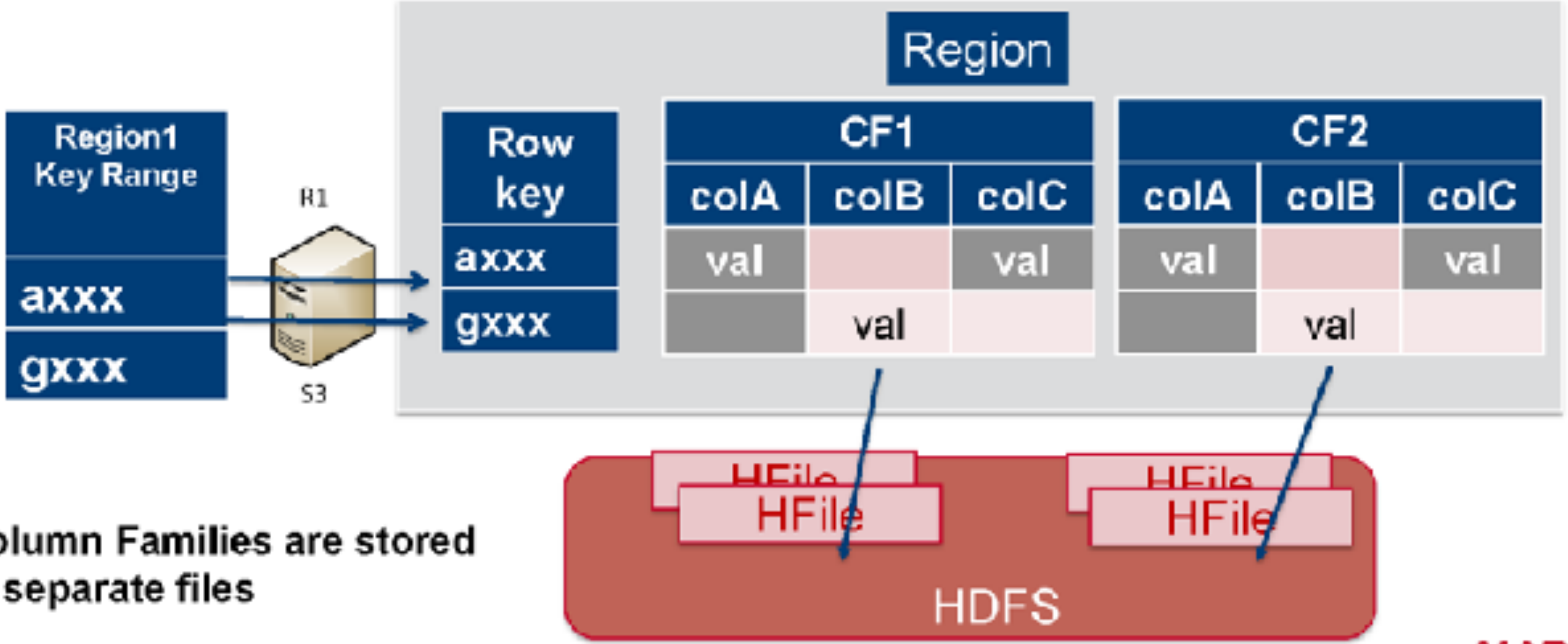
twistr

xaxx

Tables are partitioned into **key ranges** (**regions**)
Region= served by nodes (Region Servers)
Regions are spread across cluster

Column Families are stored Separately

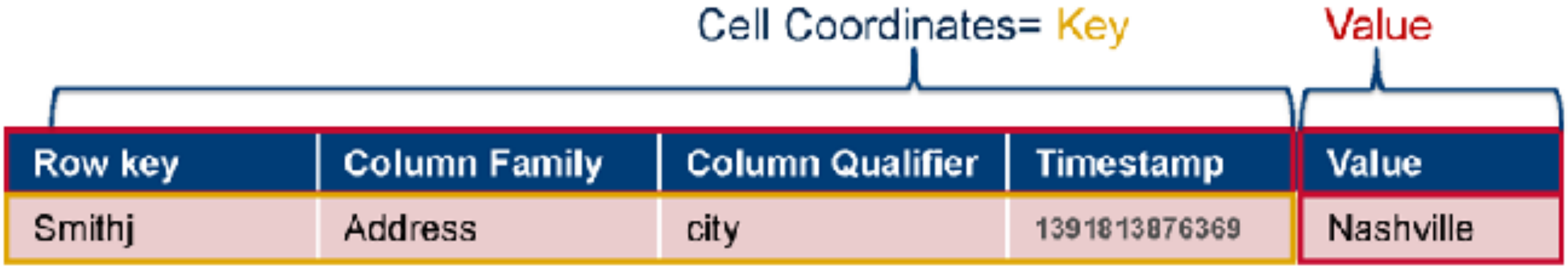
- column families are stored and can be accessed separately



Column Families are stored in separate files

HBase Data Model - Cells

- Data is stored in **Key Value** format
- Value for each **cell** is specified by complete coordinates:
  - (Row key, ColumnFamily, Column Qualifier, timestamp) => Value

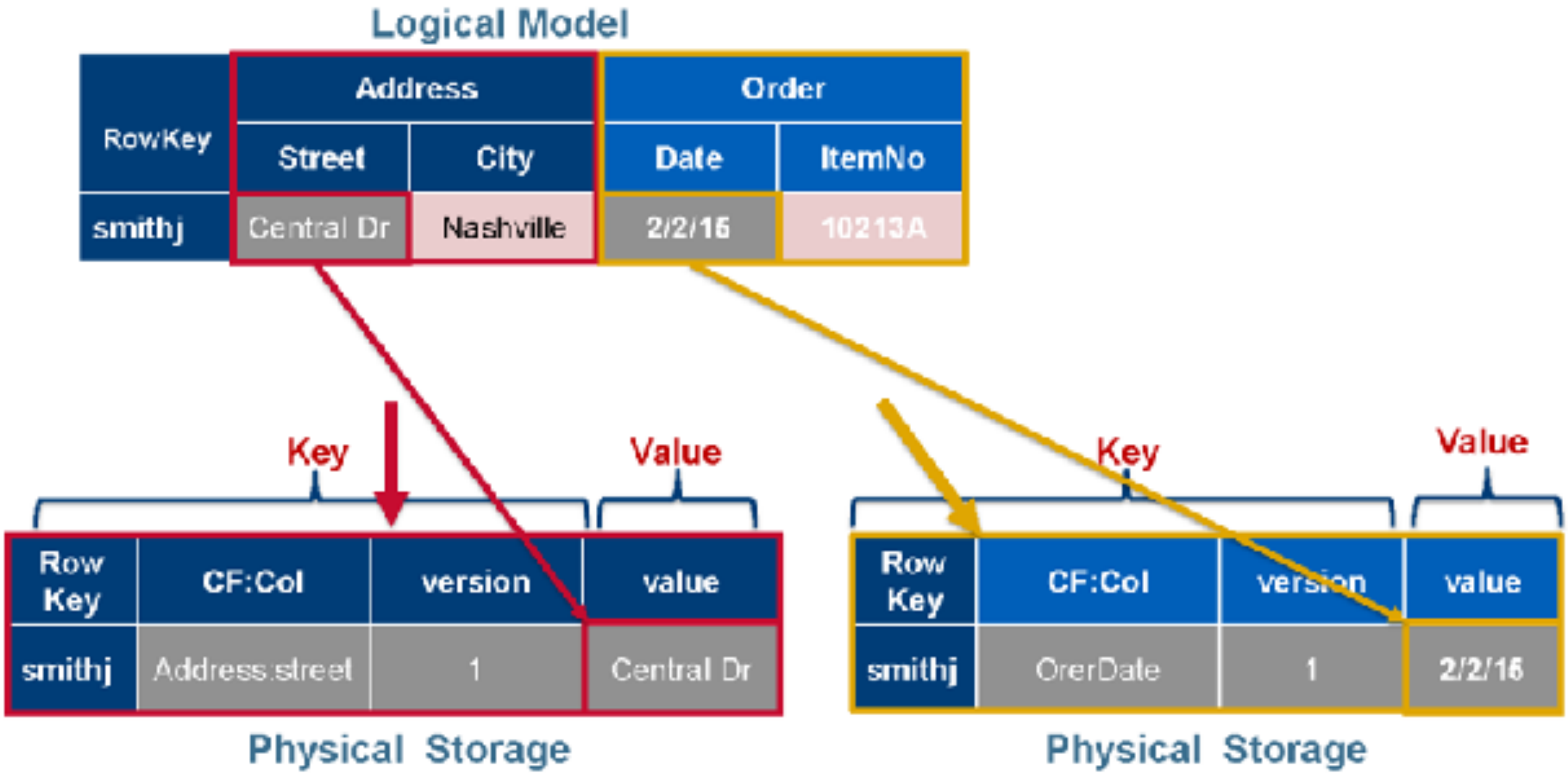


Cell Coordinates= Key

Value

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

Logical Data Model vs Physical Data Storage



Logical Model

RowKey	Address		Order	
	Street	City	Date	ItemNo
smithj	Central Dr	Nashville	2/2/16	10213A

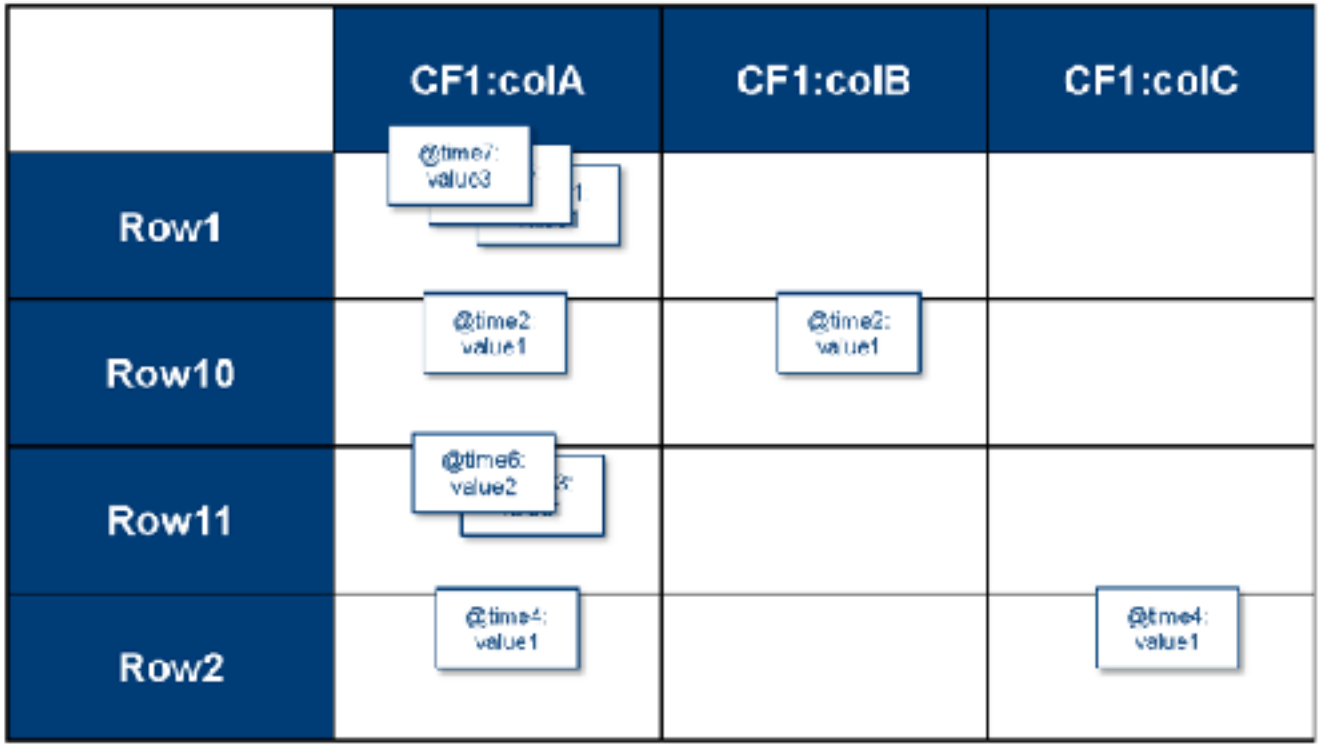
Row Key	CF:Col	version	value
smithj	Address:street	1	Central Dr

Physical Storage

Row Key	CF:Col	version	value
smithj	OrerDate	1	2/2/16

Physical Storage

Sparse Data with Cell Versions



	CF1:colA	CF1:colB	CF1:colC
Row1	@time7: value3		
Row10	@time2: value1	@time2: value1	
Row11	@time6: value2		
Row2	@time4: value1		@time4: value1

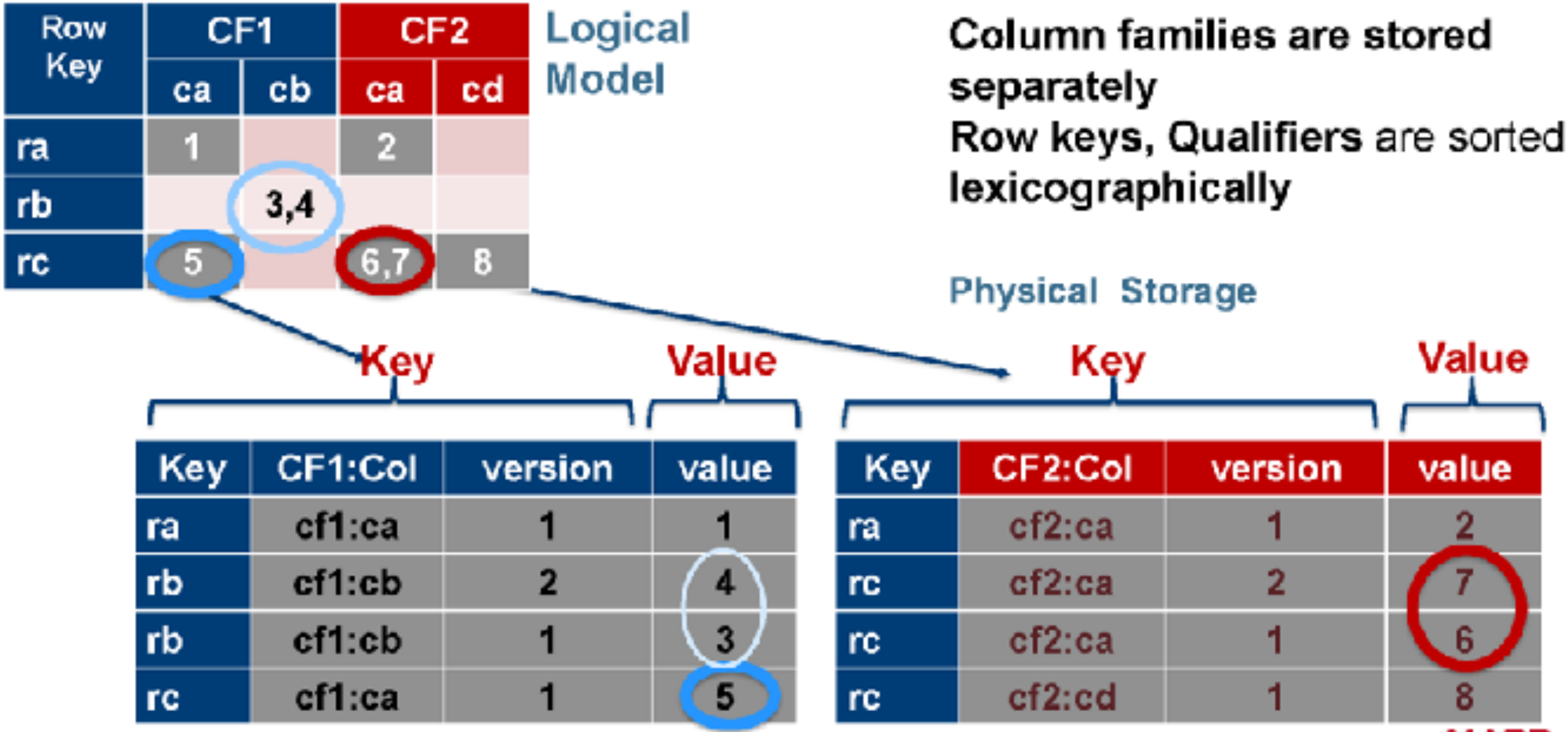
Versioned Data

Number of versions can be configured. Default number equal to 1

put, adds new cell

Key	CF:Col	version	value
smithj	Address:street	v3	19th Ave
smithj	Address:street	v2	Main St
smithj	Address:street	v1	⊗ Central Dr

Logical Data Model vs Physical Data Storage



Logical Model

Row Key	CF1		CF2	
	ca	cb	ca	cd
ra	1		2	
rb		3,4		
rc	5		6,7	8

Column families are stored separately

Row keys, Qualifiers are sorted lexicographically

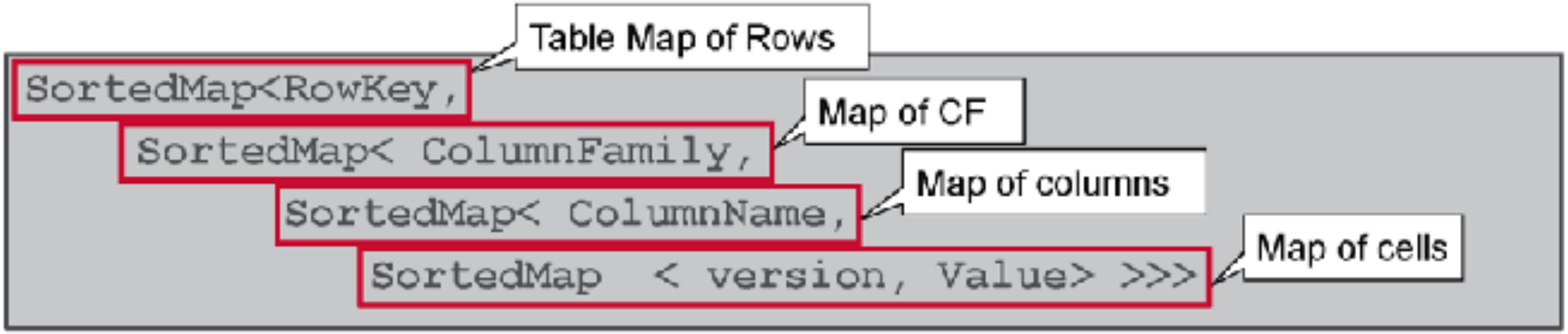
Physical Storage

Key	CF1:Col	version	value
ra	cf1:ca	1	1
rb	cf1:cb	2	4
rb	cf1:cb	1	3
rc	cf1:ca	1	5

Key	CF2:Col	version	value
ra	cf2:ca	1	2
rc	cf2:ca	2	7
rc	cf2:ca	1	6
rc	cf2:cd	1	8

HBase Table is a Sorted Map of Maps

In Java this is the table:



```
SortedMap<RowKey,
     SortedMap< ColumnFamily,
            SortedMap< ColumnName,
                SortedMap  < version, Value> >>>
```

Table Map of Rows

Map of CF

Map of columns

Map of cells

Key	CF:Col	version	value
smithj	Address:street	v2	Main St
smithj	Address:street	v1	Central Dr
spata	Address:street	v1	High Ave
turnerb	Address:street	v1	Cedar St

Key	CF:Col	version	value
smithj	Order:Date	v1	2/2/15
spata	Order:Date	v1	1/31/14
turnerb	OrderDate	v1	7/8/14

HBase Table - SortedMap

```
<smithj,<Address, <street, <v1, Central Dr>>
                  <street, <v2, Main St>>
             <Order <Date, <v1, 2/2/15>>>
<spata,<Address, <street, <v1,High Ave>>
              <Order <Date, <v1, 1/31/14>>>
<turnerb,<Address, <street, <v1,Cedar St>>
              <Order <Date, <v1, 7/8/14>>>
```

Key	CF:Col	version	value
smithj	Address:street	v2	Main St
smithj	Address:street	v1	Central Dr
spata	Address:street	v1	High Ave
turnerb	Address:street	v1	Cedar St

Key	CF:Col	version	value
smithj	Order:Date	v1	2/2/15
shawa	Order:Date	v1	1/31/14
turnerb	OrderDate	v1	7/8/14

Basic Table Operations

- **Create Table, define Column Families before data is imported**
  - but not the rows keys or number/names of columns
- Low level API, technically more demanding
- **Basic data access operations (CRUD):**

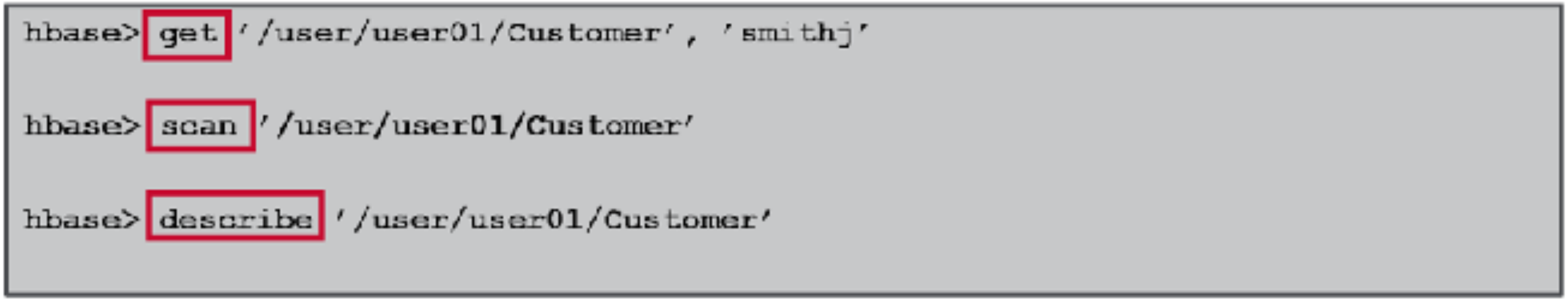
put	Inserts data into rows (both create and update)
get	Accesses data from one row
scan	Accesses data from a range of rows
delete	Delete a row or a range of rows or columns

Create HBase Table – Using HBase Shell

```
hbase> create '/user/user01/Customer', {NAME =>'Address'} , {NAME =>'Order'}
hbase> put '/user/user01/Customer', 'smithj', 'Address:street', ‘Central Dr’
hbase> put '/user/user01/Customer', 'smithj', 'Order:Date', ‘2/2/15’
hbase> put '/user/user01/Customer', 'spata', 'Address:city', ‘Columbus’
hbase> put '/user/user01/Customer', 'spata', 'Order:Date', '1/31/14'
```

Row Key	Address			Order	
	street	city	state	Date	ItemNo
smithj	Central Dr	Nashville	TN	2/2/15	10213A
spata	High Ave	Columbus	OH	1/31/14	23401V
turnerb	Cedar St	Seattle	WA	7/8/14	10938A

Create HBase Table – Using HBase Shell

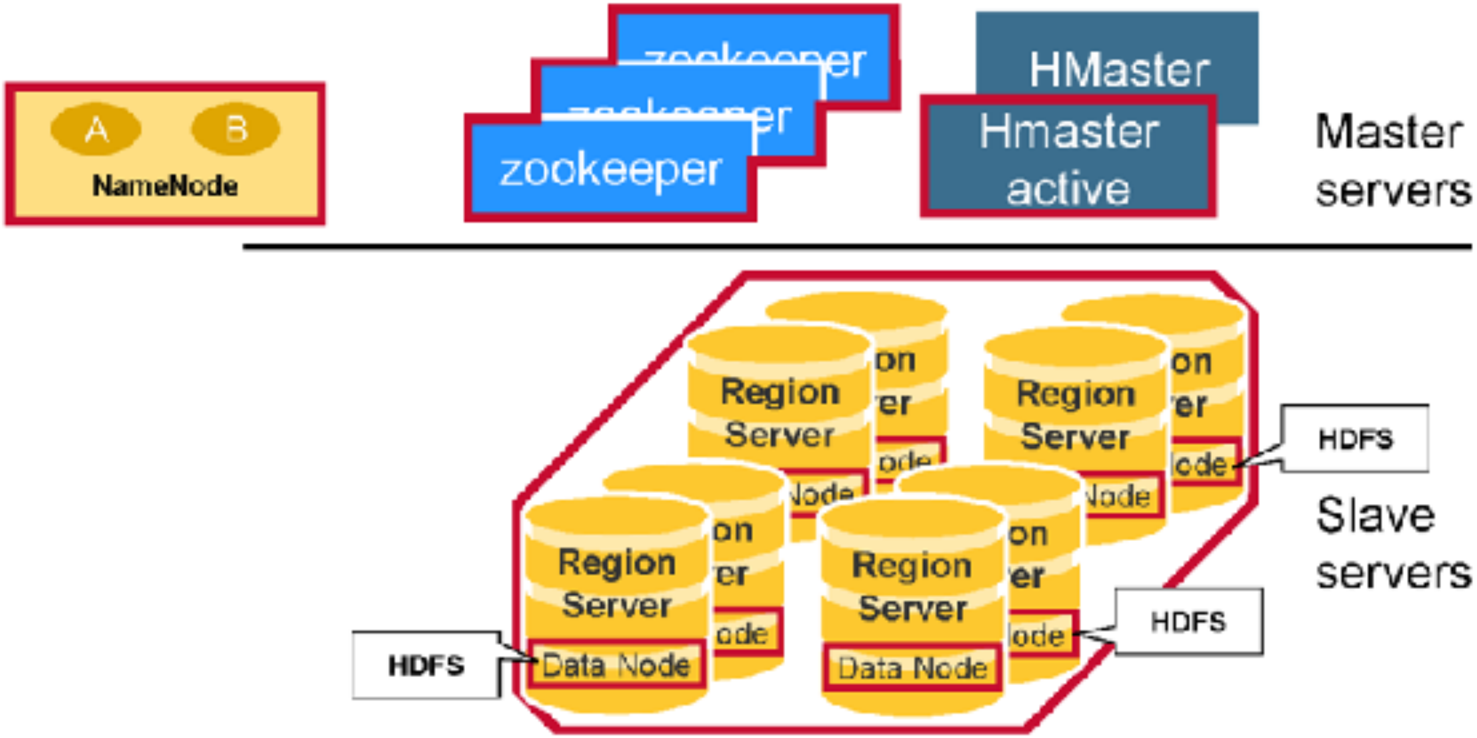


```
hbase> get '/user/user01/Customer', 'smithj'

hbase> scan '/user/user01/Customer'

hbase> describe '/user/user01/Customer'
```

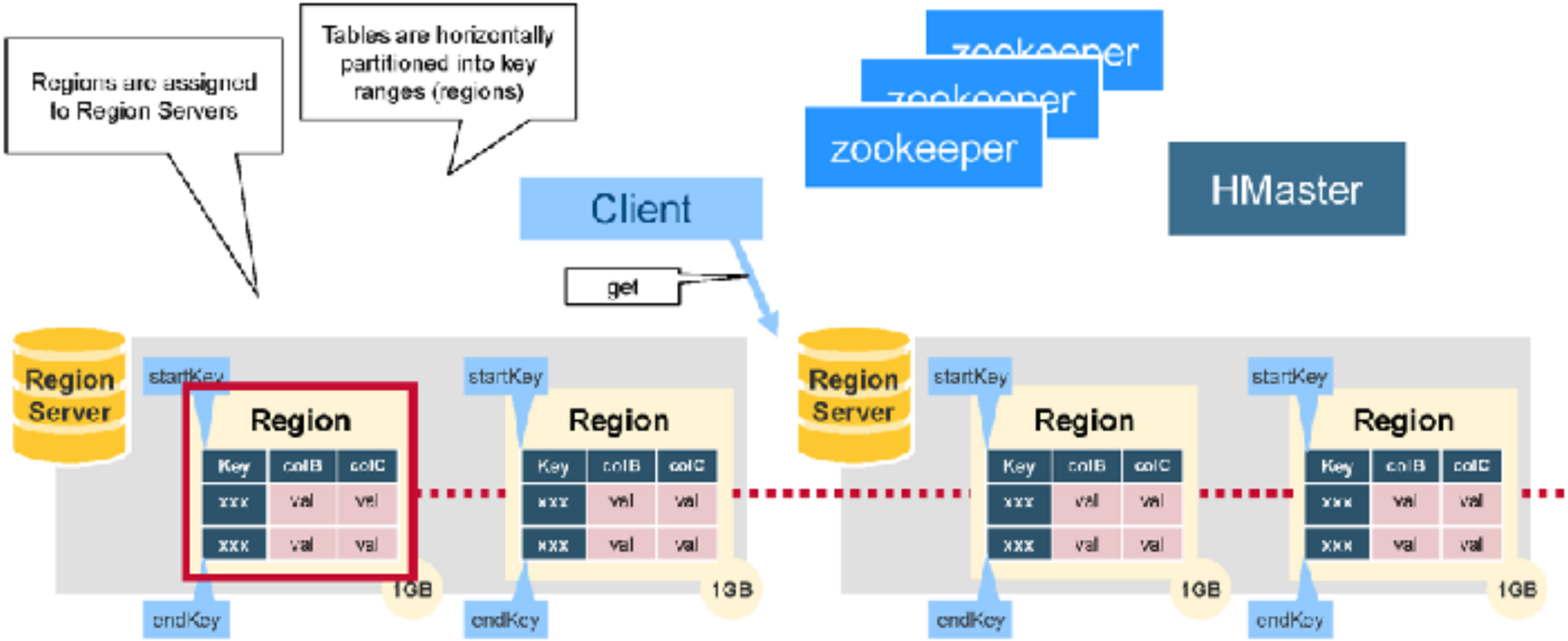
HBase Architecture

Data flow for Writes, Reads
Designed to Scale

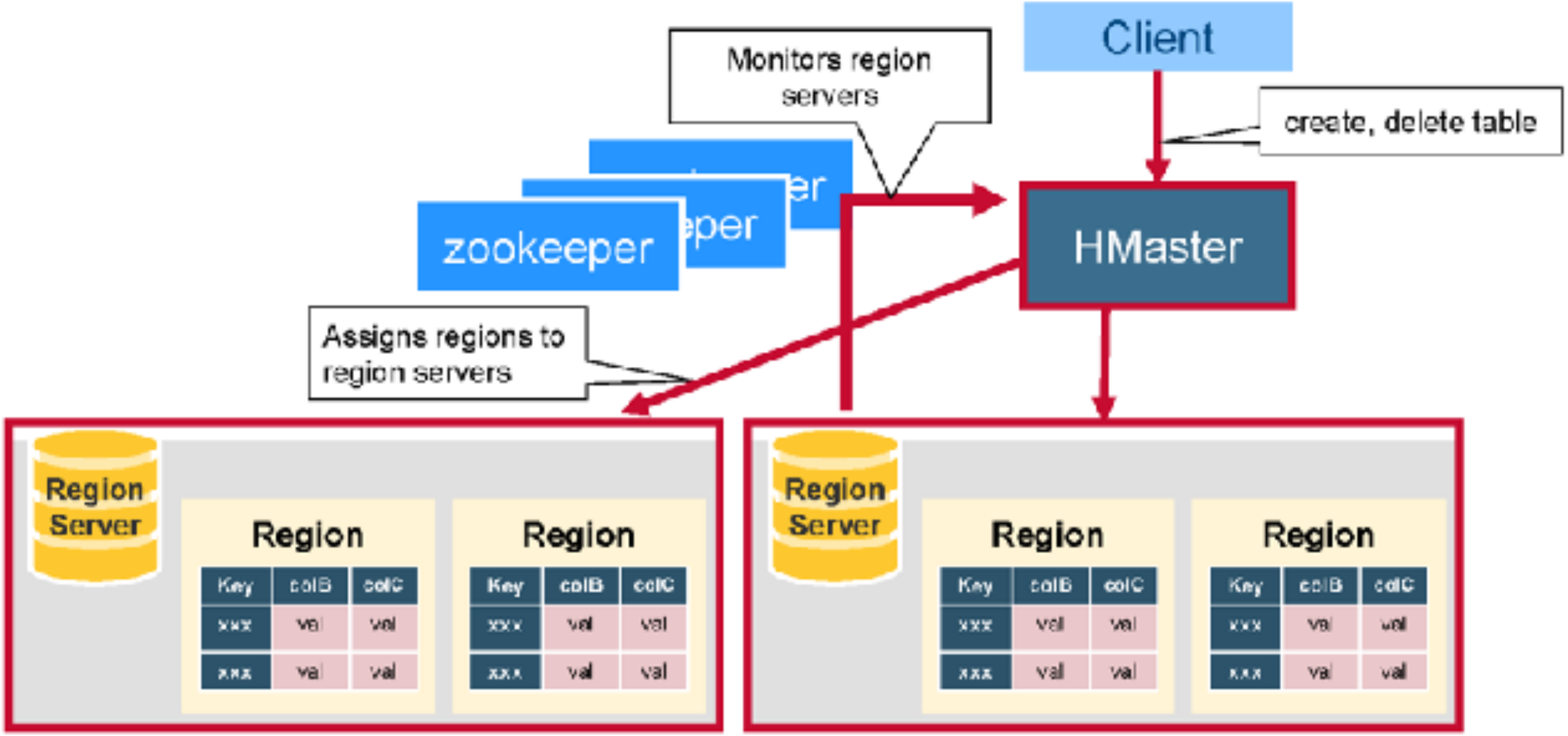
HBase Architectural Components



Regions


Regions are assigned to Region Servers
Tables are horizontally partitioned into key ranges (regions)
zookeeper
zookeeper
zookeeper
Client
HMaster
get
Region Server
startKey
Region
Key
colB
colC
xxx
val
val
xxx
val
val
1GB
endKey
startKey
Region
Key
colB
colC
xxx
val
val
xxx
val
val
1GB
endKey
Region Server
startKey
Region
Key
colB
colC
xxx
val
val
xxx
val
val
1GB
endKey
startKey
Region
Key
colB
colC
xxx
val
val
xxx
val
val
1GB
endKey

HBase HMaster

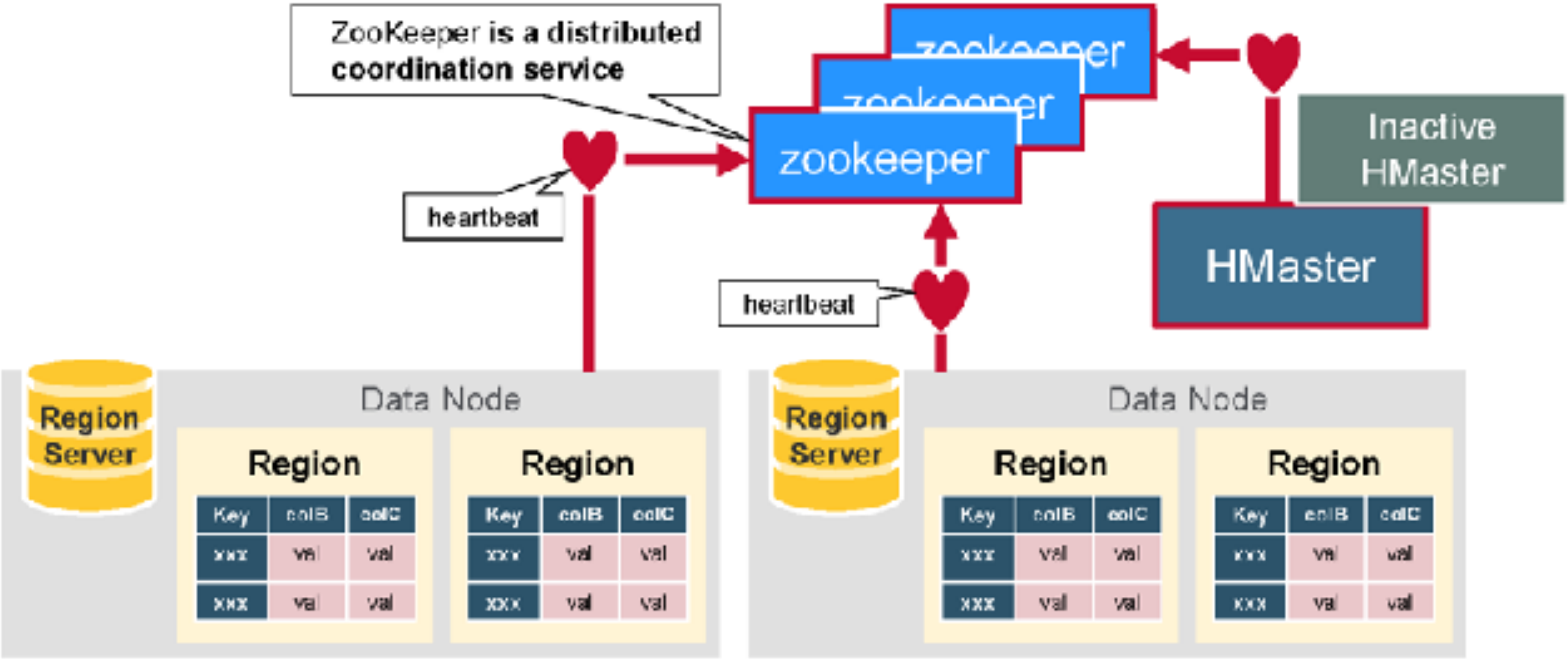


Monitors region servers

create, delete table

Assigns regions to region servers

Zookeeper The Coordinator



ZooKeeper is a distributed coordination service

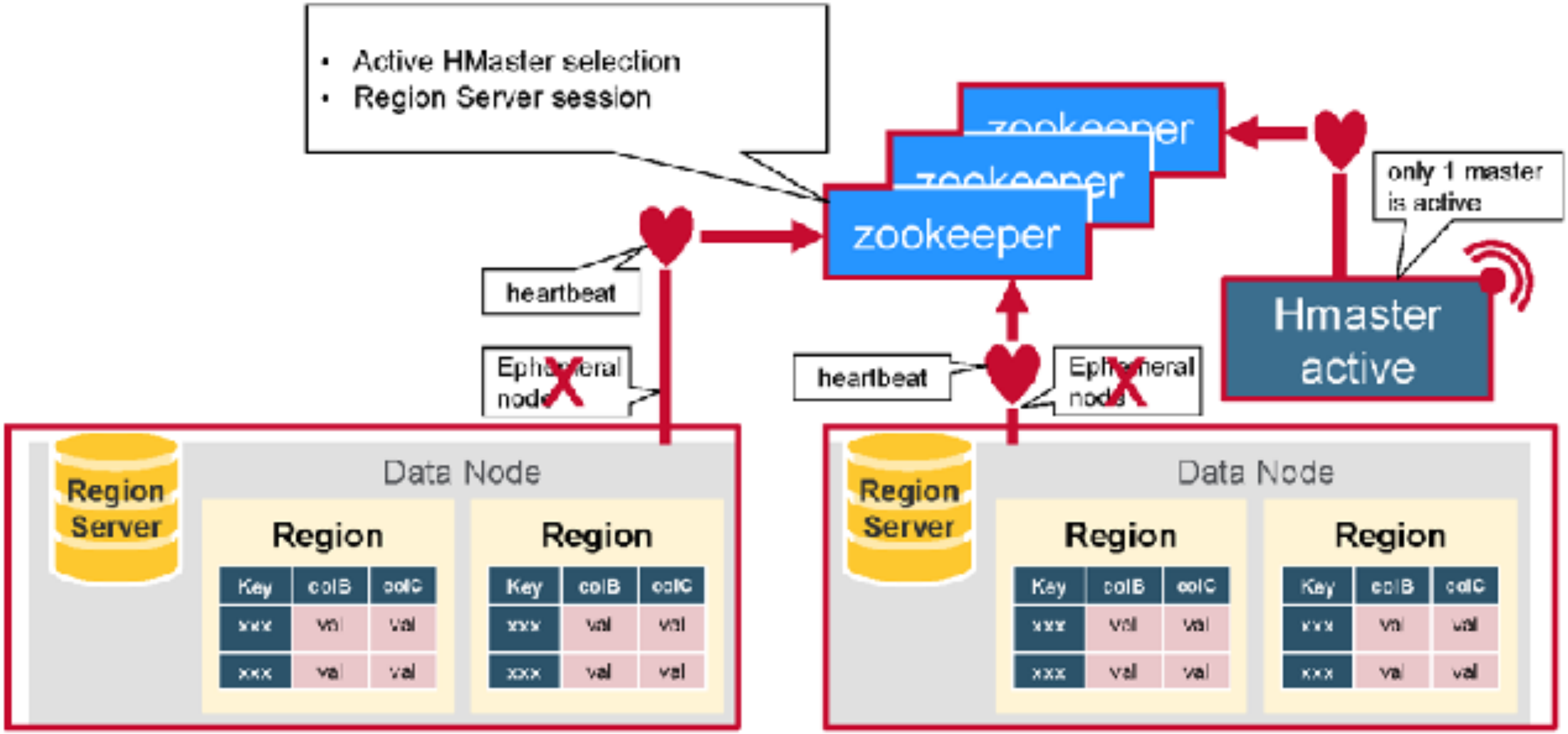
Key	colB	colC
xxx	val	val
xxx	val	val

Key	colB	colC
xxx	val	val
xxx	val	val

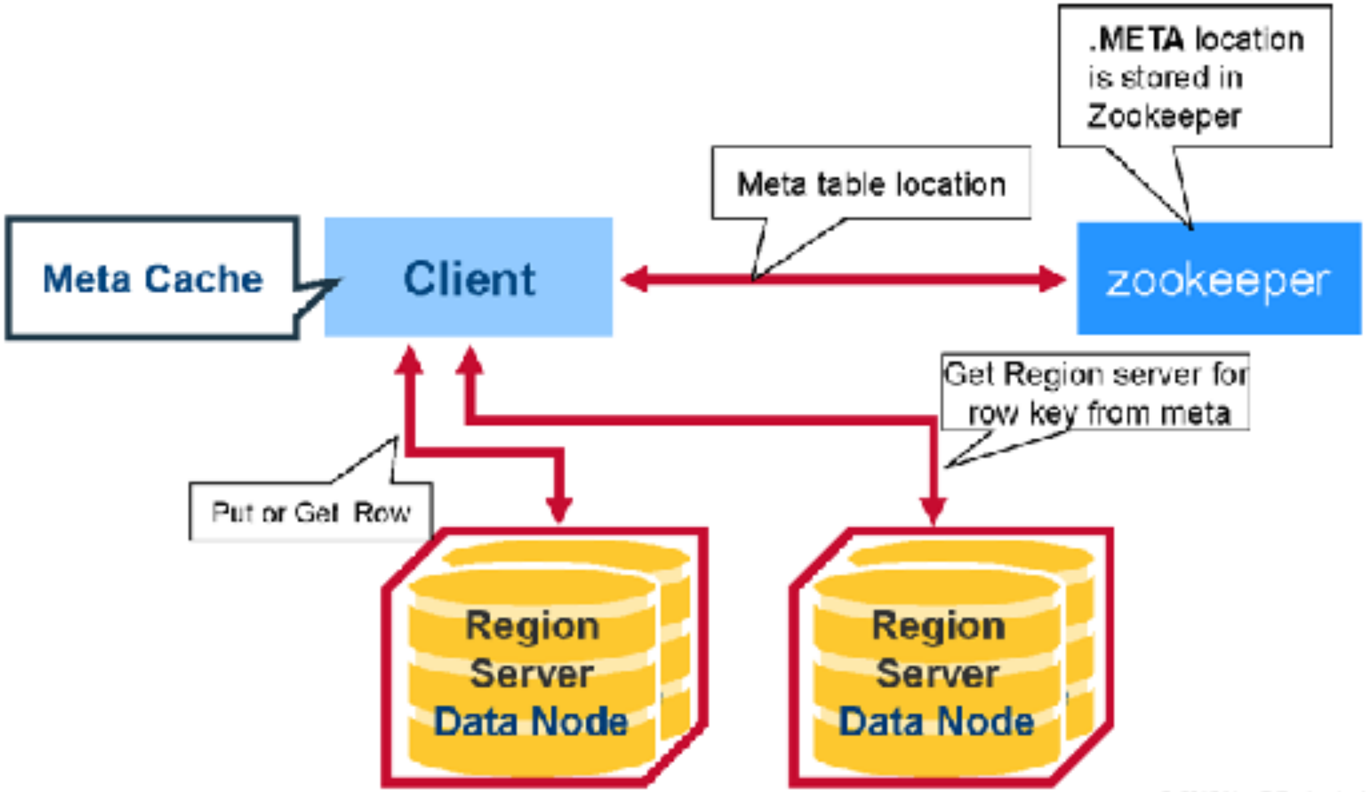
Key	colB	colC
xxx	val	val
xxx	val	val

Key	colB	colC
xxx	val	val
xxx	val	val

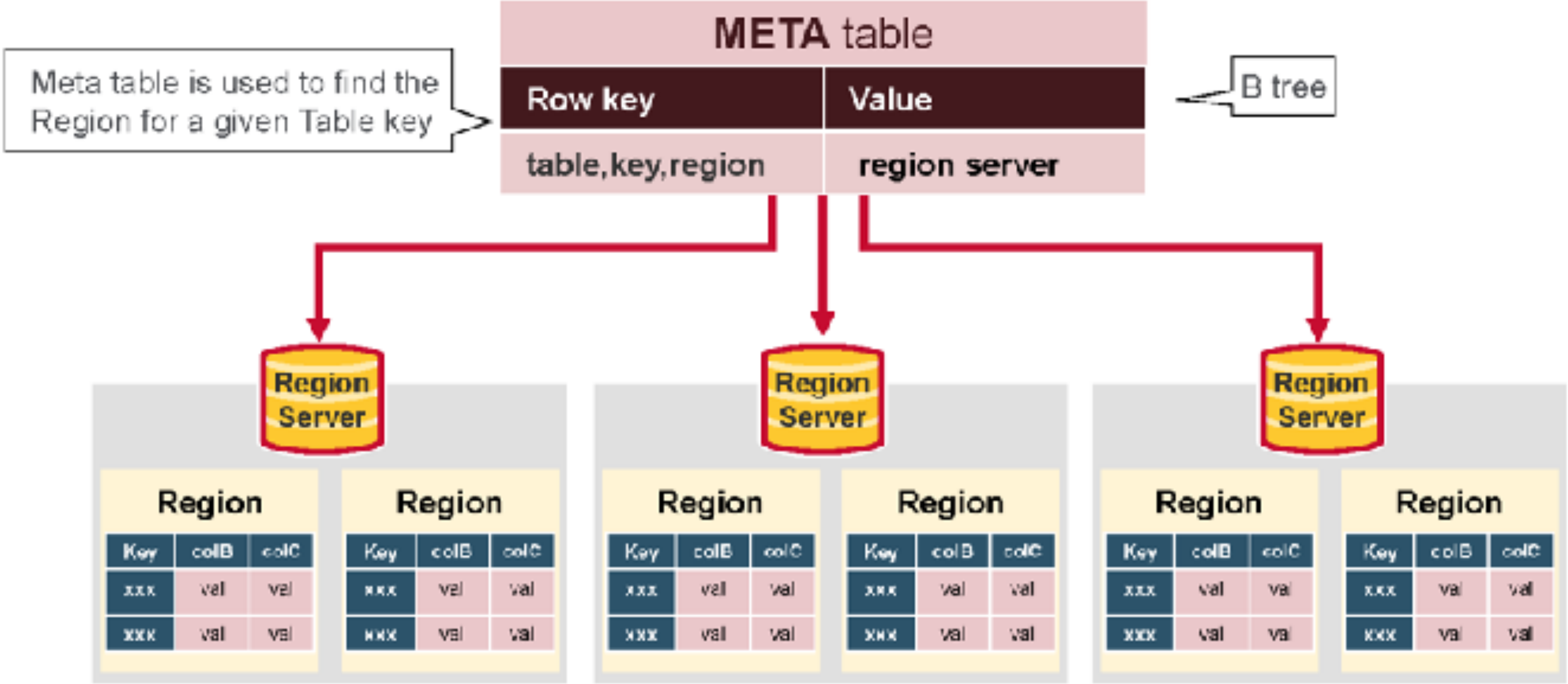
How the Components Work Together



HBase First Read or Write



HBase Meta Table



Meta table is used to find the Region for a given Table key

B tree

META table

Row key	Value
table,key,region	region server

Region

Key	colB	colC
xxx	val	val
xxx	val	val

Region

Key	colB	colC
xxx	val	val
xxx	val	val

Region

Key	colB	colC
xxx	val	val
xxx	val	val

Region

Key	colB	colC
xxx	val	val
xxx	val	val

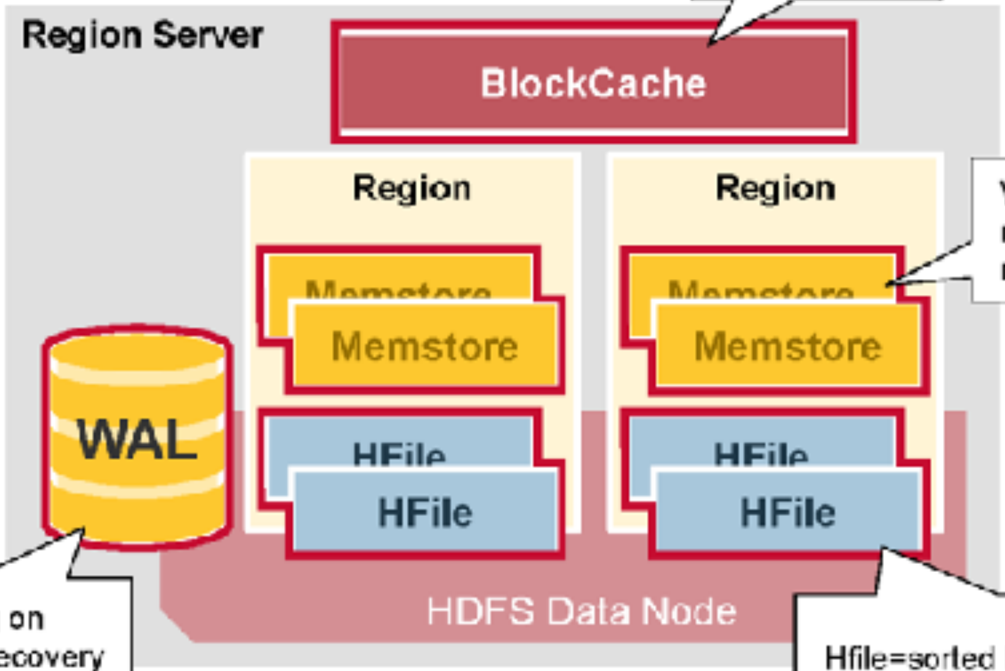
Region

Key	colB	colC
xxx	val	val
xxx	val	val

Region

Key	colB	colC
xxx	val	val
xxx	val	val

Region Server Components



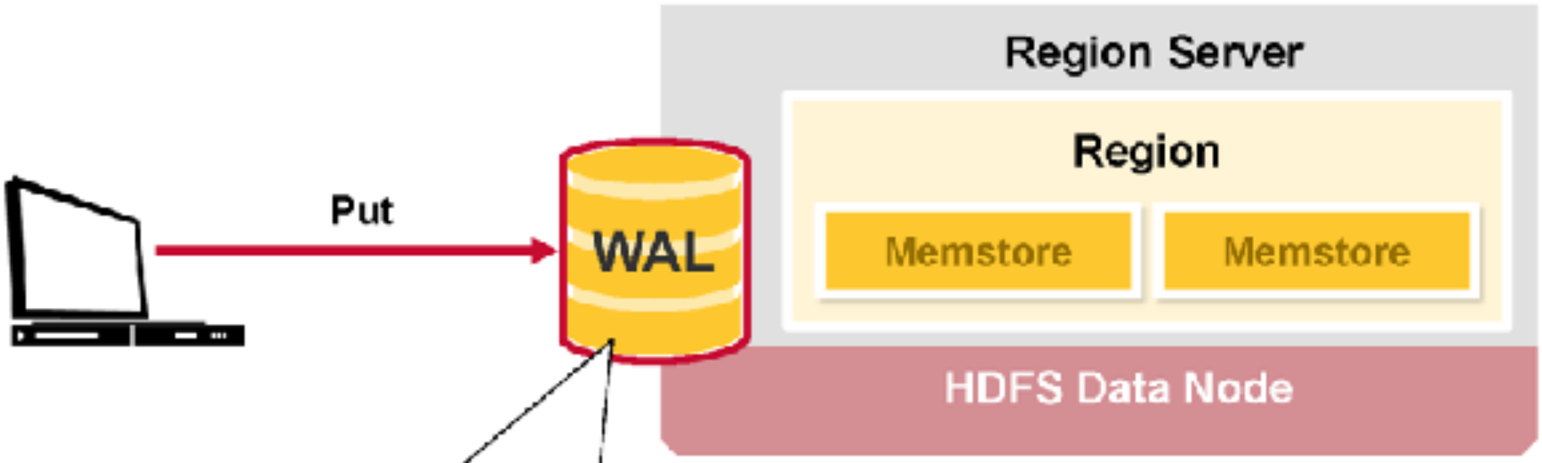
Read Cache, LRU evicted

Write Cache, sorted map of keyValues in memory.

Write ahead log on disk- Used for recovery

Hfile=sorted KeyValues on disk

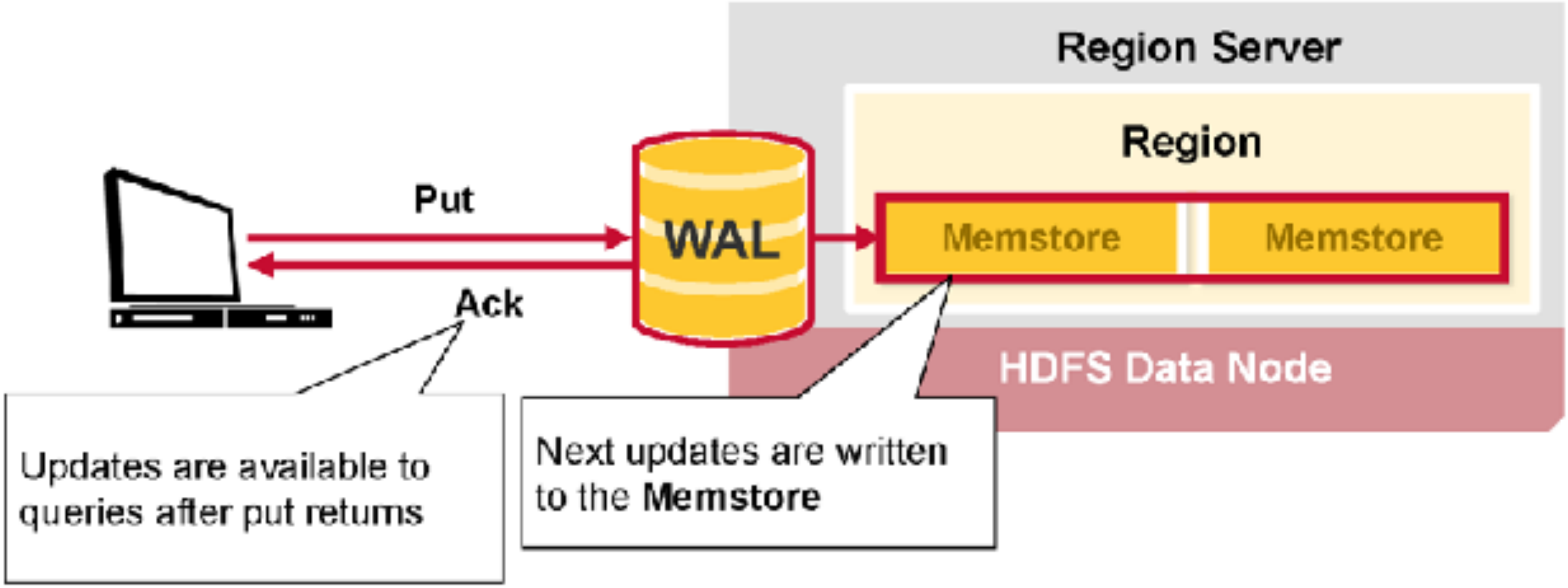
HBase Write Steps



Each incoming record written to WAL for durability:

- log on disk
- updates appended sequentially

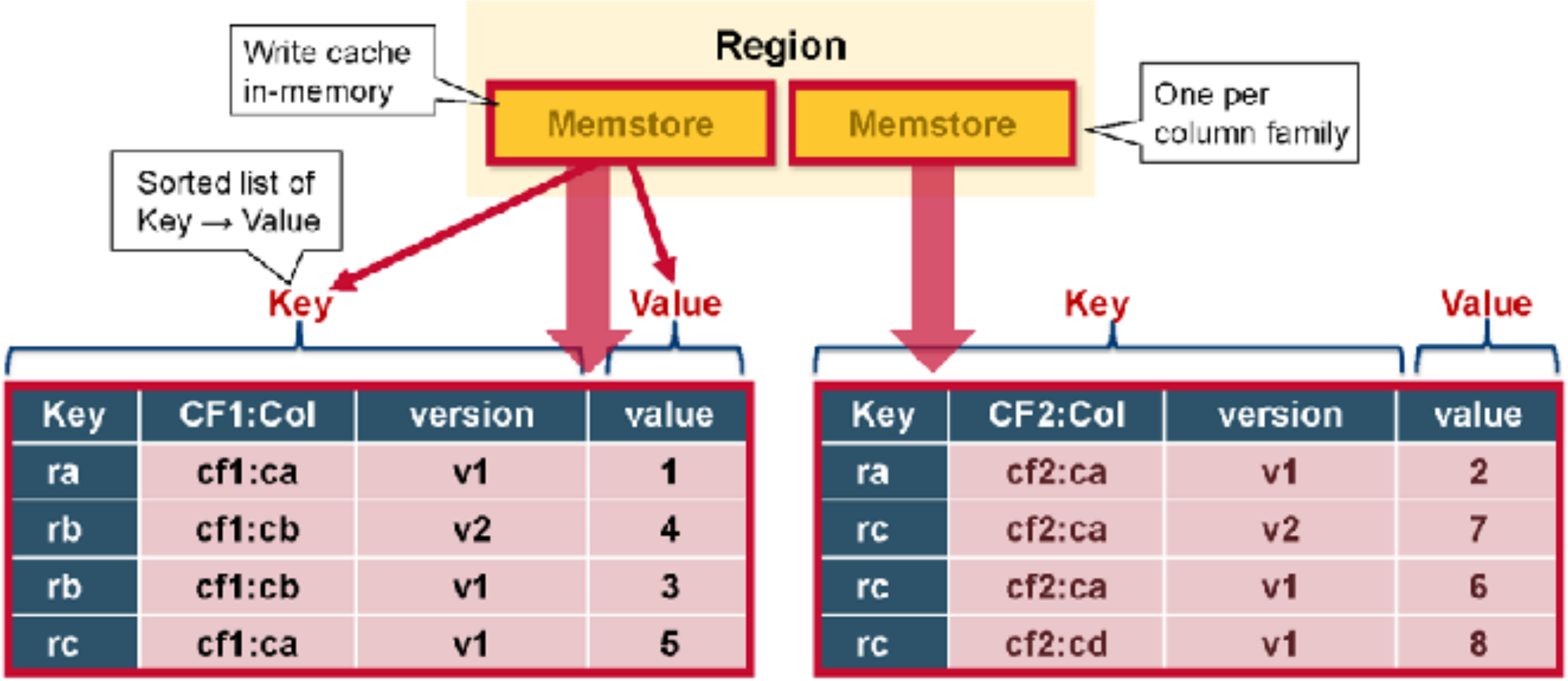
HBase Write Steps – (2)



Updates are available to queries after put returns

Next updates are written to the **Memstore**

HBase Memstore



Write cache in-memory

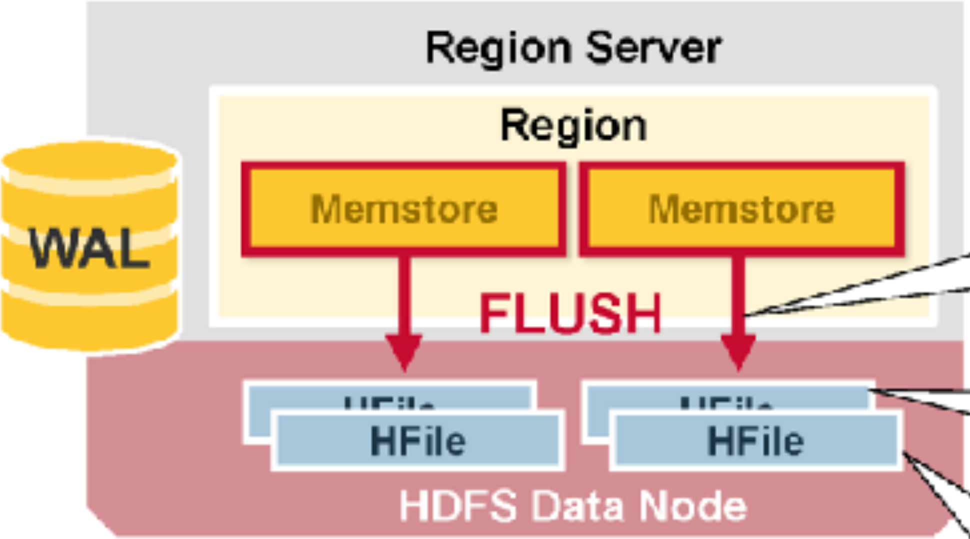
One per column family

Sorted list of Key → Value

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

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

HBase Region Flush

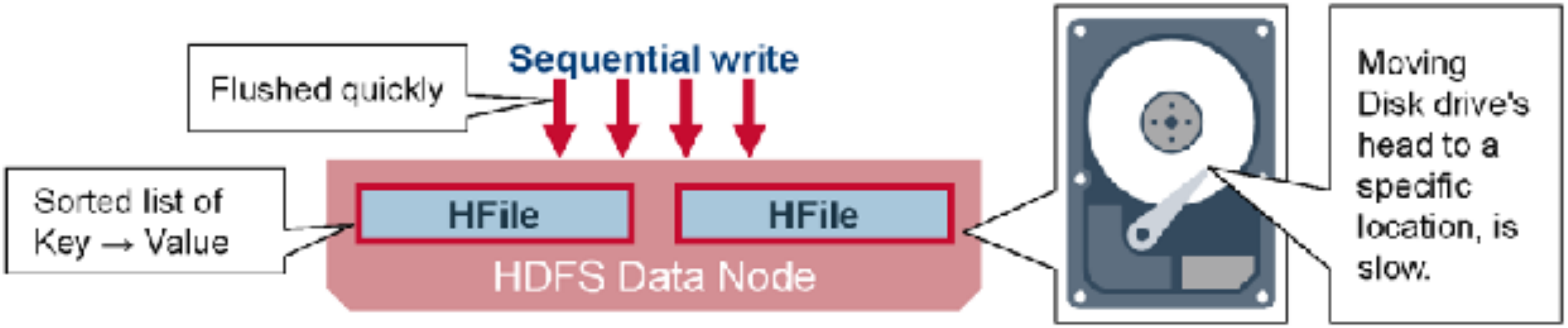


All memstores in region **flushed** to new HFiles on disk

HFile: sorted list of key → values on disk

Saves the last written sequence number so the system knows what was persisted so far.

HBase HFile



Sequential write

Flushed quickly

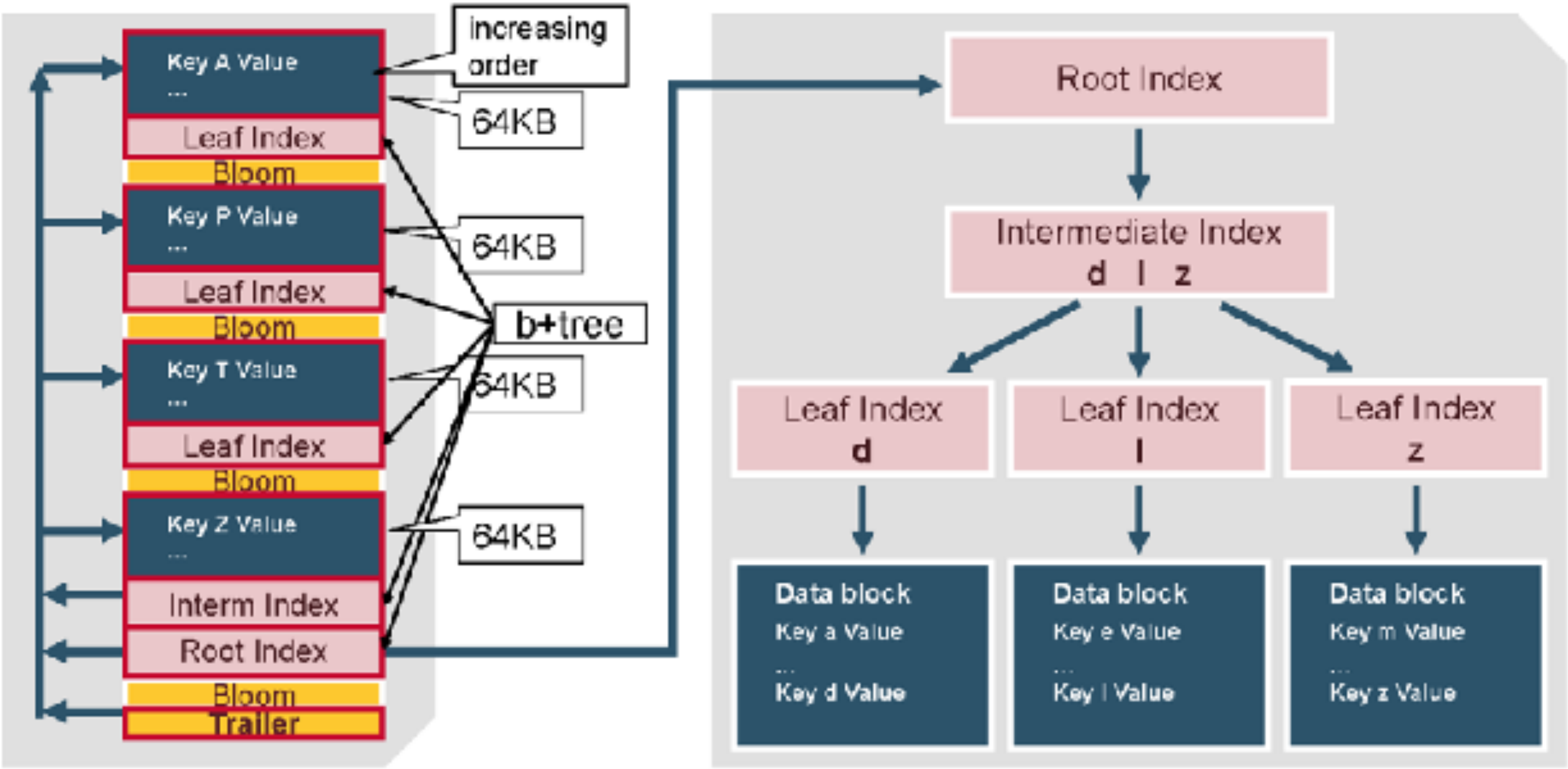
Sorted list of Key → Value

Moving Disk drive's head to a specific location, is slow.

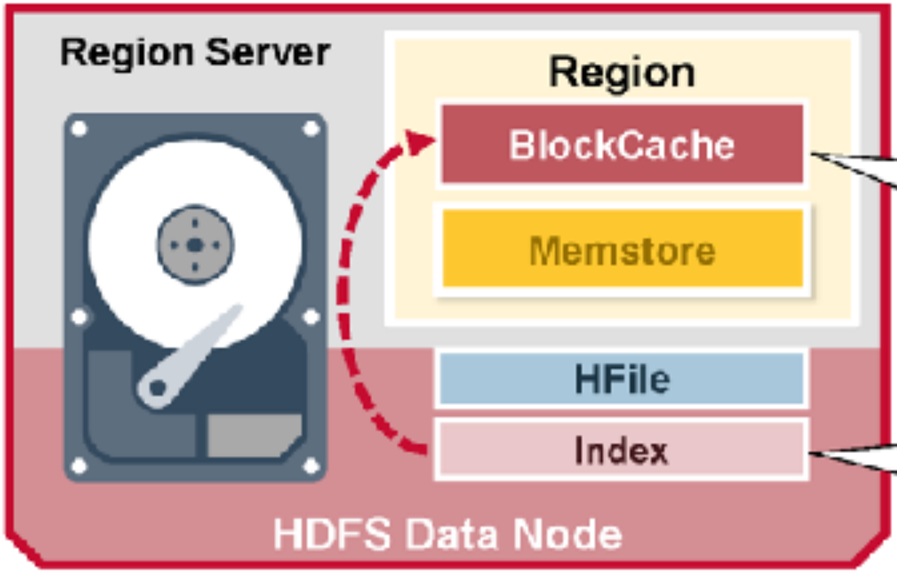
Key			Value
Key	**CF1:Col**	**version**	**value**
ra	cf1:ca	v1	1
rb	cf1:cb	v2	4
rb	cf1:cb	v1	3
rc	cf1:ca	v1	5

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

HBase HFile Structure



HFile Index



Lookups can be performed with a single disk seek

Hfile Index is kept in Block Cache (Memory)

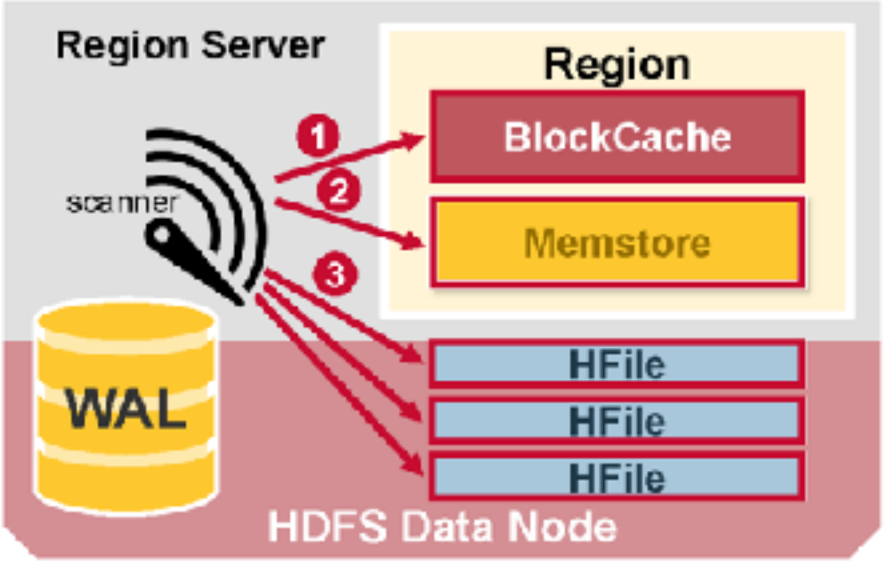
Index is loaded when the HFile is opened

HBase Read Merge

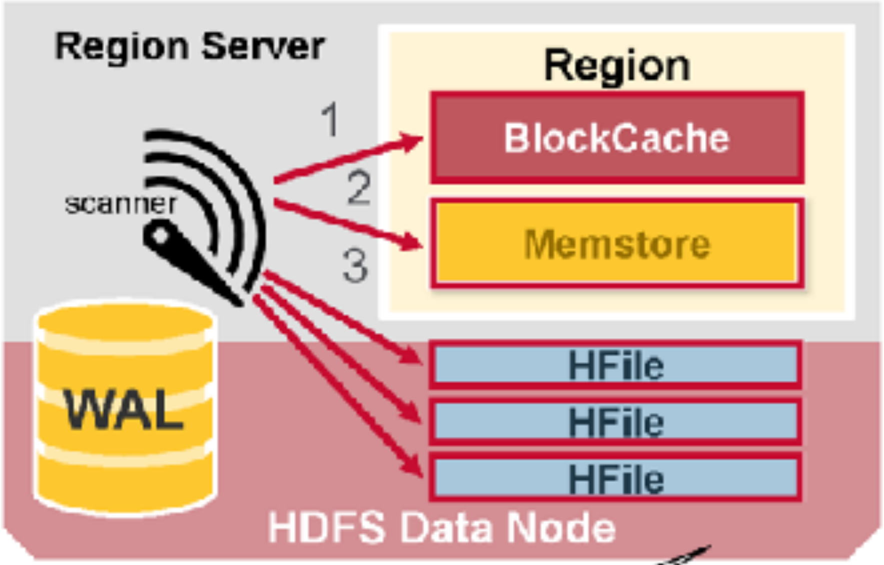
1 First the scanner looks for the Row KeyValues in the Block cache

2 Next the scanner looks in the Memstore

3 If all row cells not in memstore or blockCache, look in HFiles



HBase Read Merge

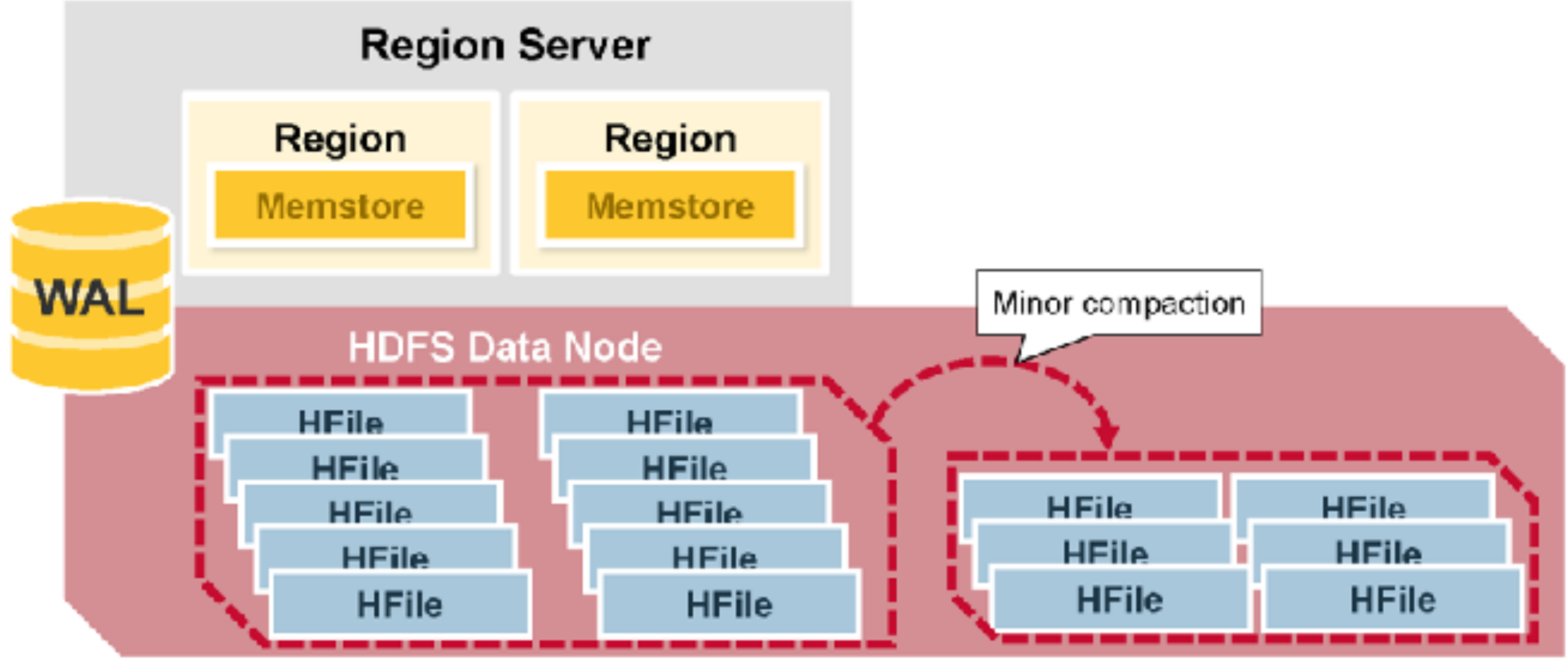


MemStore creates multiple **small store files** over time when **flushing.**

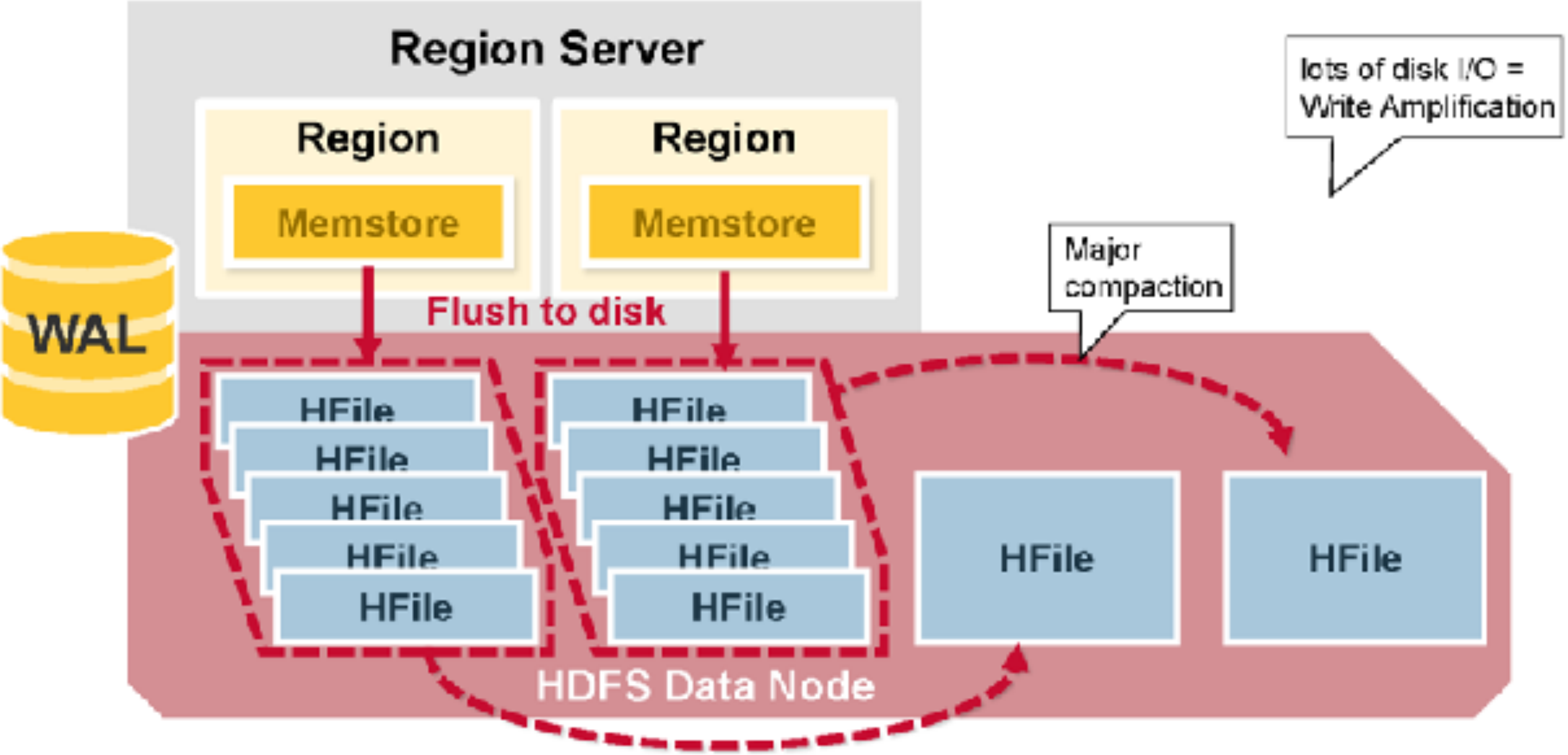
Read Amplification

- multiple files have to be examined

HBase Minor Compaction



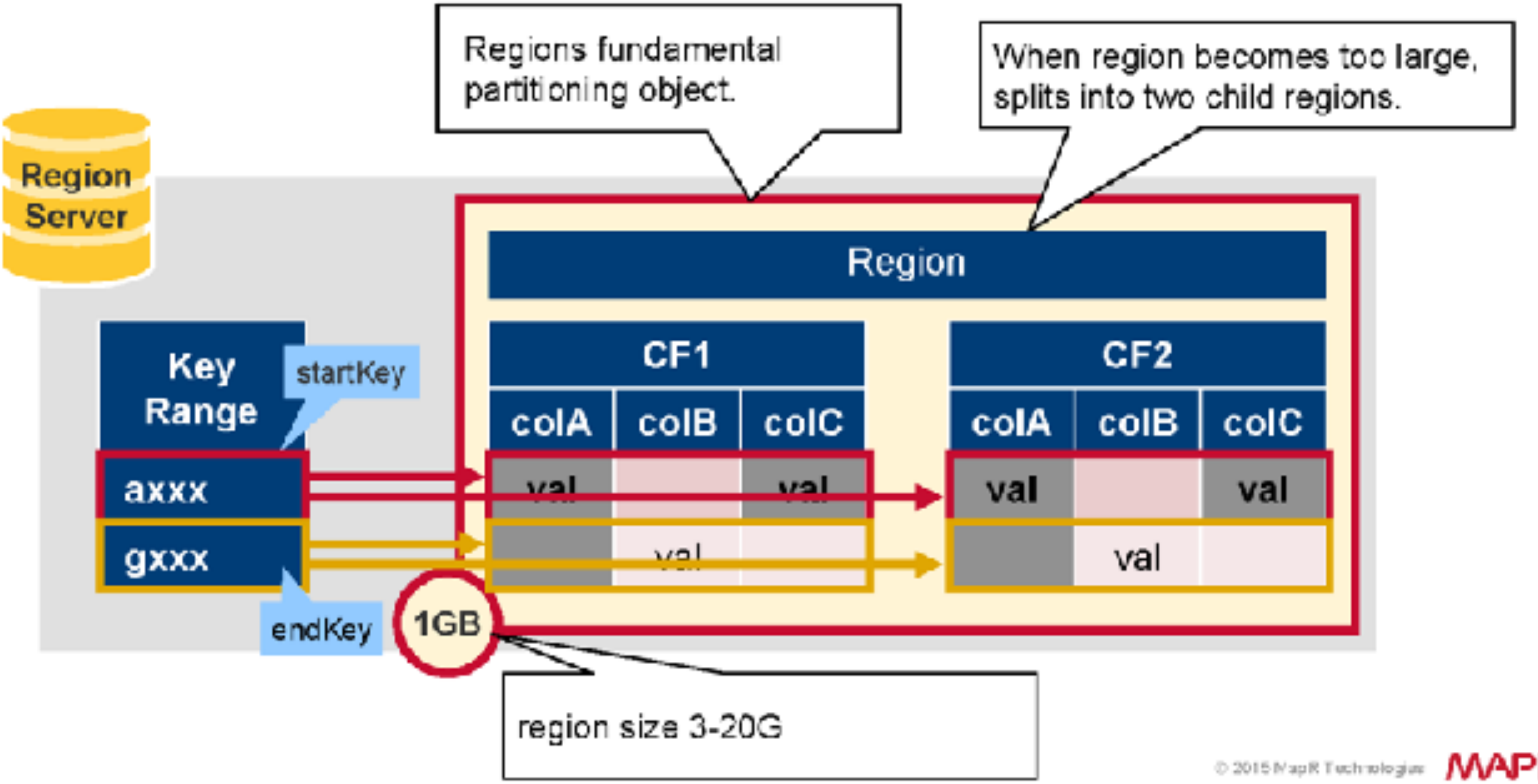
HBase Major Compaction



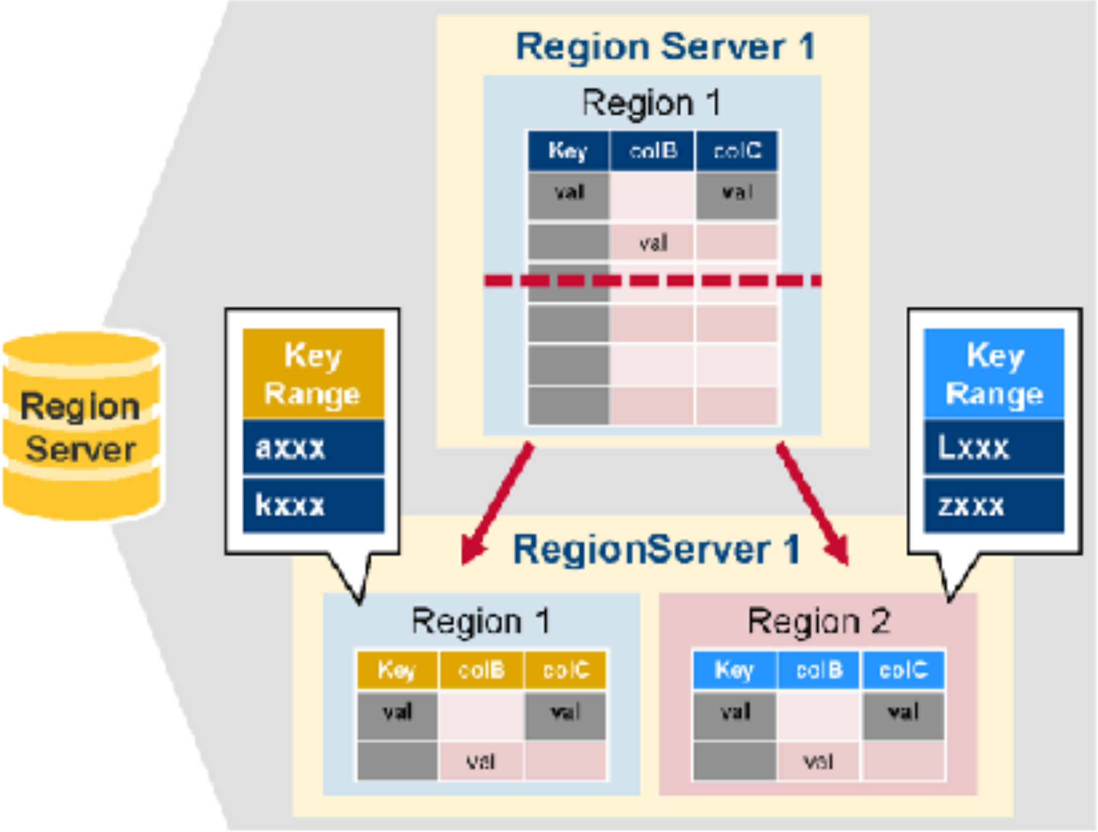
Major compaction

lots of disk I/O = Write Amplification

Region = contiguous keys

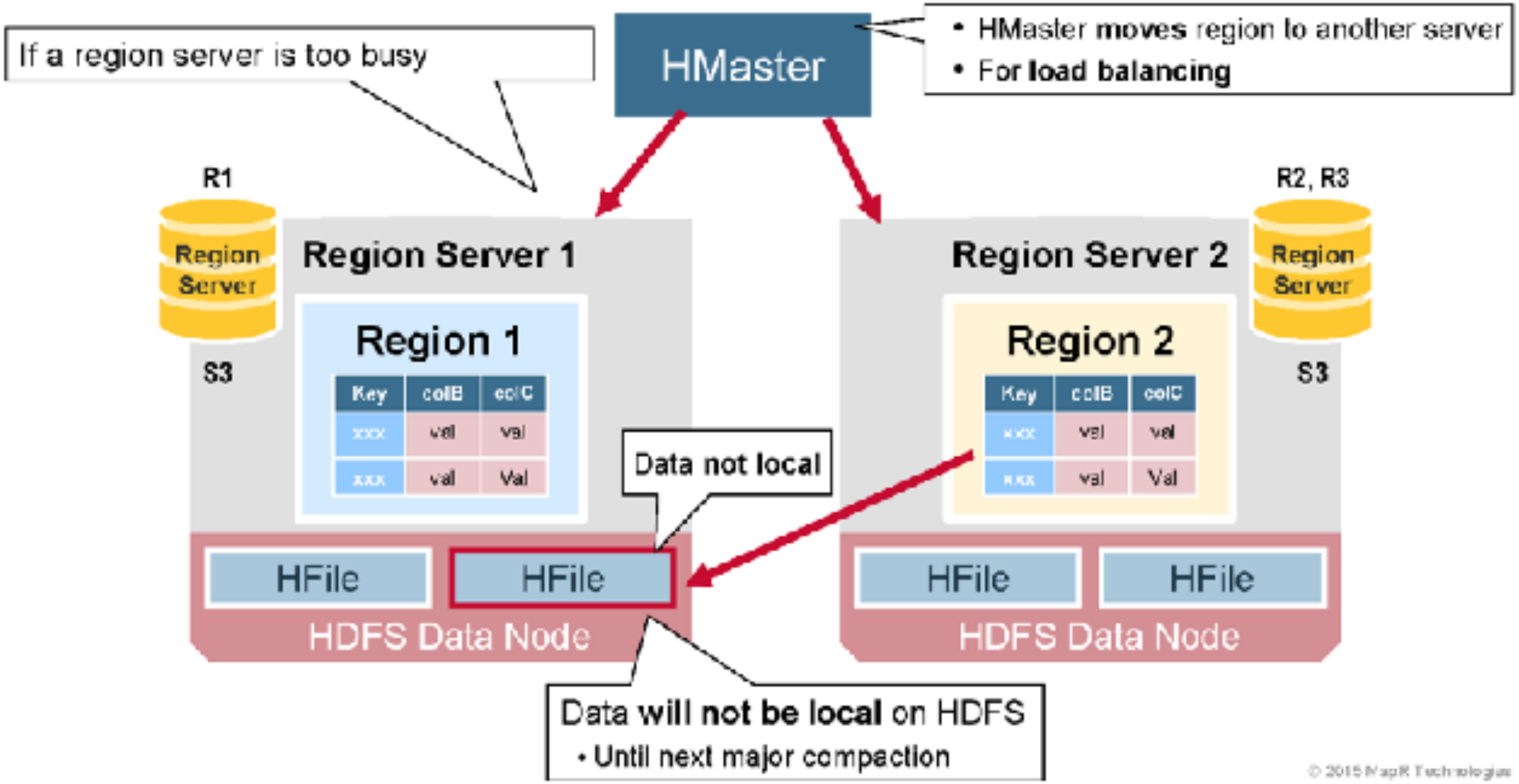


Region Split



when region size > hbase.hregion.max. filesize → split

Region Load Balancing



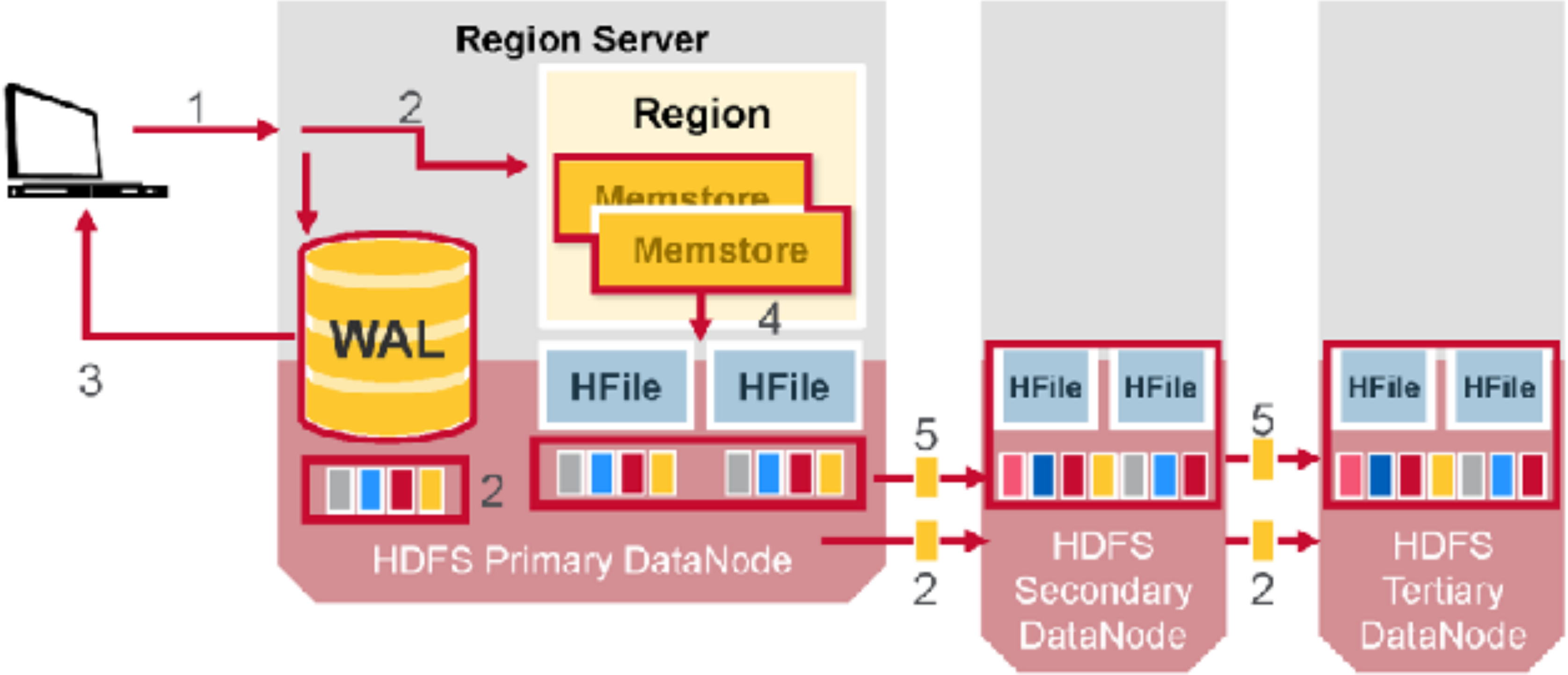
If a region server is too busy

- HMaster **moves** region to another server
- For **load balancing**

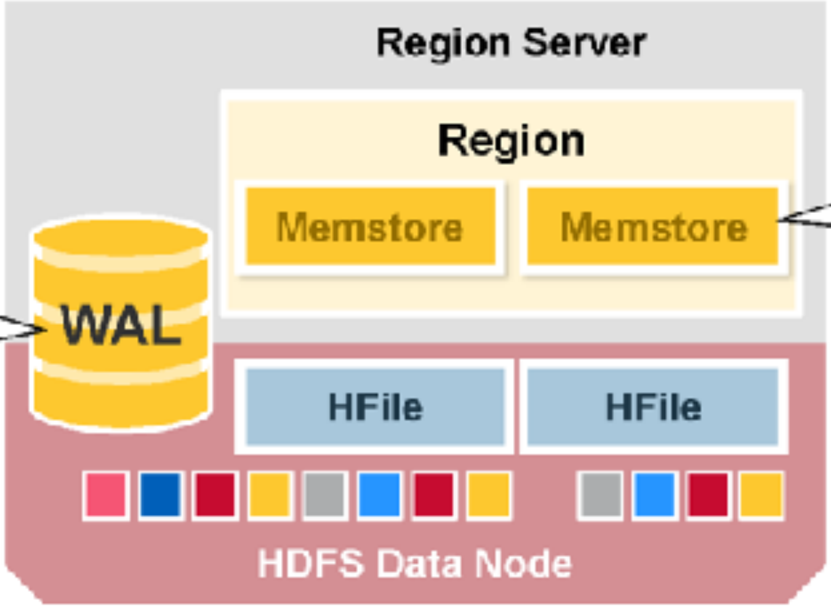
Data not local

Data **will not be local** on HDFS
- Until next major compaction

HDFS Data Replication



HDFS Data Replication – (2)

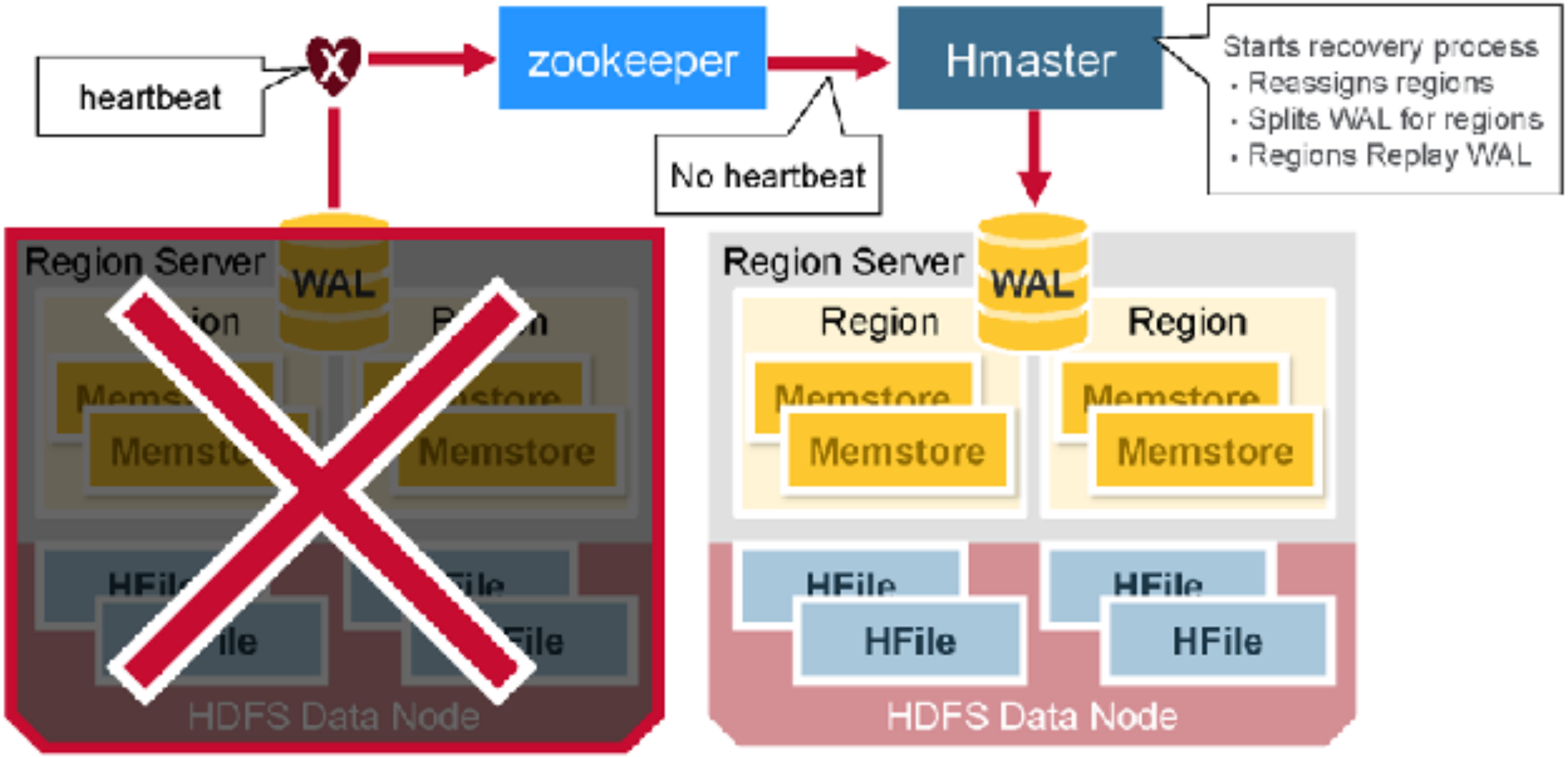


WAL is replicated for fault tolerance

Memstore is in memory, NOT replicated

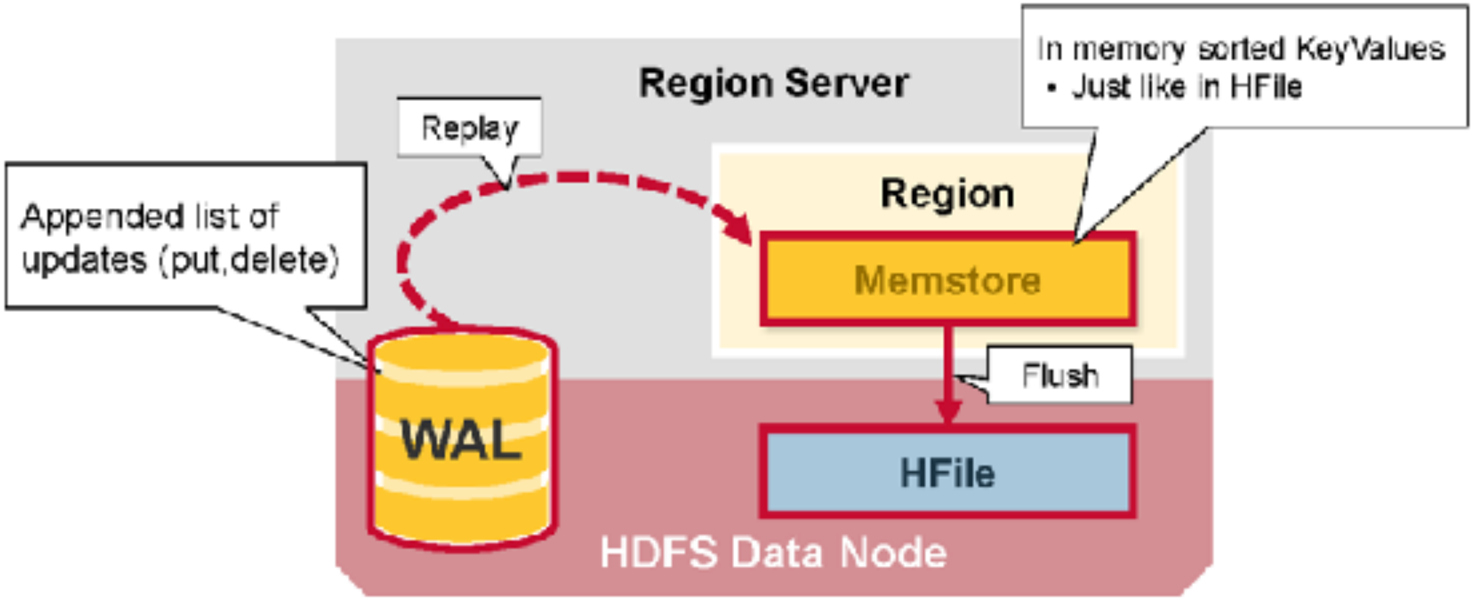
How does HBase recover updates not persisted to HFiles?

HBase Crash Recovery



Starts recovery process
- Reassigns regions
- Splits WAL for regions
- Regions Replay WAL

Data Recovery



Region Server

Replay

Appended list of updates (put,delete)

WAL

Region

Memstore

In memory sorted KeyValues
- Just like in HFile

Flush

HFile

HDFS Data Node

Apache HBase Architecture Benefits

Better than many NoSQL data store solutions, hence its popularity

- **Strong consistency model**
  - When a write returns, all readers will see same value
- **Scales automatically**
  - Regions **split** when data grows too large
  - Uses HDFS to **spread** and **replicate data**
- **Built-in recovery**
  - Using **Write Ahead Log** (similar to journaling on file system)
- **Integrated with Hadoop**
  - MapReduce on HBase is straightforward