Getting Started with HBase Application development

Sridhar Reddy sreddy@mapr.com

March 17th, 2015

Objectives of this session

- What is HBase?
  - Why do we need NoSQL / HBase?
  - Overview of HBase & HBase data model
  - HBase Architecture and data flow
  - Hbase Use Cases
- How to get started
  - Demo/Lab using HBase Shell using MapR Sandbox
- Developing Applications using HBase Java API
  - Demo/Lab to perform CRUD operations using put, get, scan, delete

Agenda

- Why do we need NoSQL / HBase?
- Overview of HBase & HBase data model
- HBase Architecture and data flow
- HBase Use Cases
- Demo/Lab using HBase Shell
  - Create tables and CRUD operations using MapR Sandbox
- HBase Java API to perform CRUD operations
  - Demo HBase Java API using Eclipse & MapR Sandbox

Prerequisite for Hands-On-Labs

- Install a one-node MapR Sandbox on your laptop
- Install and configure Eclipse to develop HBase applications using Java API
- MapR Client is optional

Hbase_Tutorial_3_2015.pdf

GettingStartedWithHBase_3_17_2015.pdf

exercises.zip &

solutions.zip

Why do we need NoSQL / HBase?

Relational Database is **typed** and **structured before stored**:

- Entities map to tables, normalized
- **Transactions** handle concurrency , consistency
- Structured Query Language
  - Joins tables to bring back data

Customer Table

Customer ID	First Name	Last Name	Address	City	State	Zip Code
1	John	Doe	1 Junction Avenue	San Jose	CA	95134
2	Albert	Einstein	2 Physics Place	New York	NY	10003
3	Ben	Franklin	1776 Freedom Way	Philadelphia	PA	19106
4	Mark	Twain	47 Huckleberry Drive	Hannibal	MO	63401
5	Abe	Lincoln	16 Springfield Road	Springfield	IL	62561

Customer-Order Junction Table

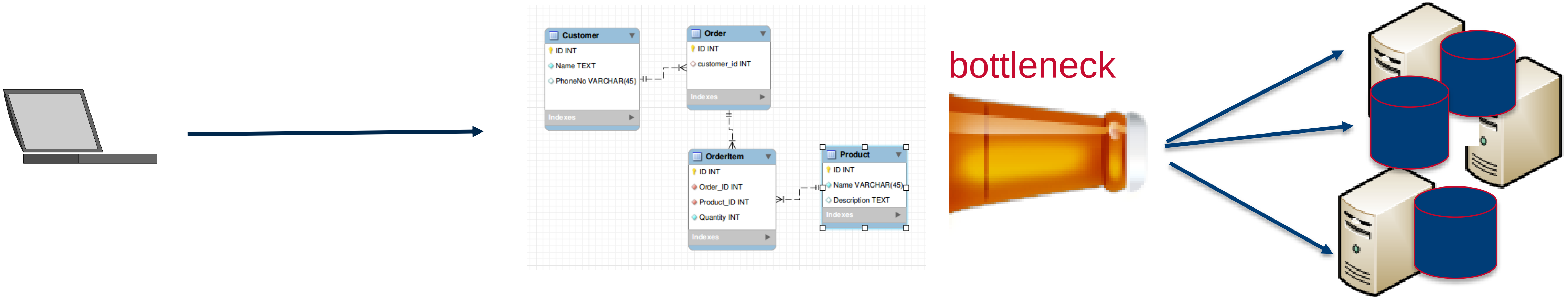
Customer ID	Order Number
1	1001
1	1002
2	1003
3	1004
4	1005
5	1006
5	1007

Order Table

Order Number	Item	Quantity	Unit Price	Total Price
1001	Hamilton Beach Coffee Maker	1	54.99	$54.99
1001	Kindle Fire 7"	1	$139.00	$139.00
1001	Divergent (paperback)	1	$5.49	$5.49
1001	Seagate Desktop HDD 4TB	1	$156.95	$156.95
1002	In Paradise: A Novel	1	$21.49	$21.49
1003	Astonish Me: A Novel	1	$16.41	$16.41
1003	Updike	1	$21.77	$21.77
1003	Sous Chef: 24 Hours on the Line	1	$18.85	$18.85
1003	Secrecy (paperback)	1	$12.71	$12.71
1004	Happy (from Despicable Me 2)	1	$1.29	$1.29
1004	Let It Go	1	$1.29	$1.29
1005	Dark Horse	1	$1.29	$1.29
1005	All of Me (Album Version)	1	$1.29	$1.29
1005	The Man	1	$1.29	$1.29
1005	Frozen (Two-Disc Blu-ray)	1	$19.96	$19.96
1007	Anchorman 2: The Legend Continues	1	$16.96	$16.96

What changed to bring on NoSQL? Big data

Horizonal scale : partition or shard tables across cluster



Distributed Joins, Transactions are Expensive

- **Cons** of the Relational Model:
  - Does not **scale** horizontally:
    - Sharding is **difficult** to manage
    - Distributed join, transactions **do not scale** across shards

NoSQL Landscape

Key Value Store
- Couchbase
- Riak
- Citrusleaf
- Redis
- BerkeleyDB
- Membrain
- ...

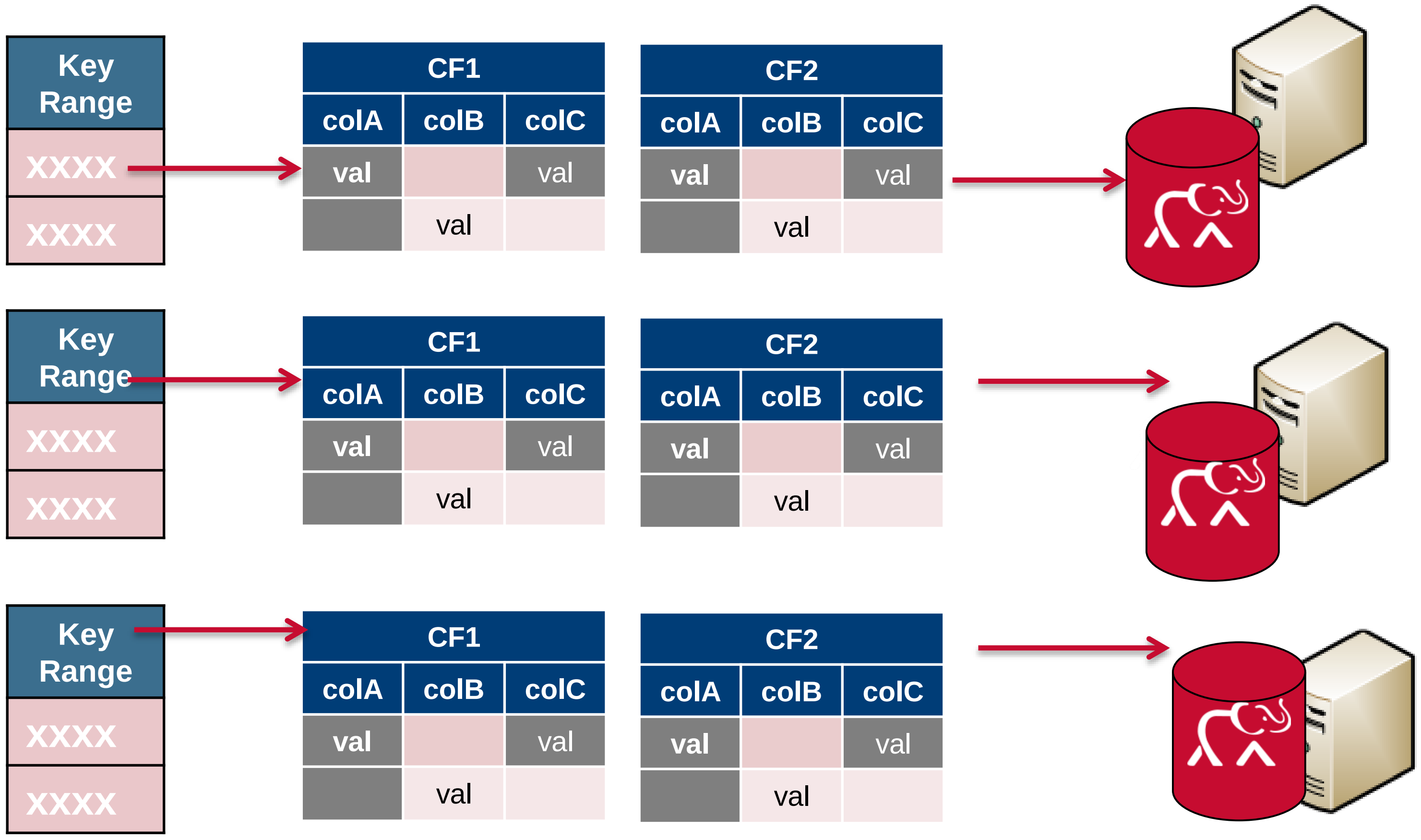
Document
- MongoDB
- CouchDB
- RavenDB
- Couchbase
- ...

Wide Column
- **HBase**
- **MapR-DB**
- Hypertable
- Cassandra
- ...

Graph
- OrientDB
- DEX
- Neo4j
- GraphBase
- ...

Hbase designed for Distribution, Scale, Speed

HBase is a Distributed Database



Data is automatically distributed across the cluster.

- Table is indexed by row key
- **Key range** is used for **horizontal partitioning**
- **Table splits happen automatically as the data grows**

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	

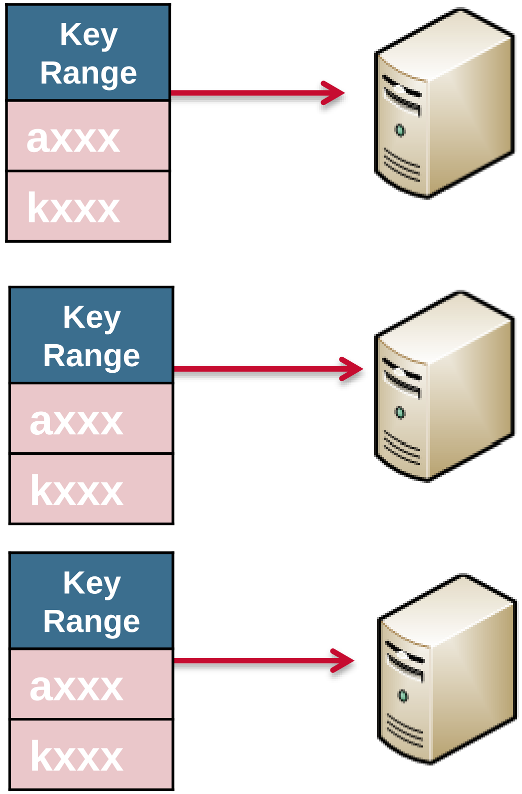
Data is accessed and stored together by RowKey

Similar Data is grouped & stored in Column Families

- share common properties:
  - **Number of versions**
  - **Time to Live (TTL)**
  - **Compression [lz4, lzf, Zlib]**
  - **In memory option …**

HBase designed for Distribution

- Distributed data stored and accessed together:
  - **Key range** is used for **horizontal partitioning**
- Pros
  - **scalable** handles data volume and velocity
  - **Fast Writes and Reads by Key**
- Cons
  - **No** joins
  - Need to **know** how data will be **queried in advance** to do good schema design to achieve best performance



HBase Data Model

Row Keys: identify the rows in an HBase table

Columns are grouped into column families

	Row Key	CF1			CF2				…
		colA	colB	colC	colA	colB	colC	colD	
R1	axxx	val		val	val			val	
	…								
	gxxx	val			val	val	val		
R2	hxxx	val	val	val	val	val	val	val	
	…								
	jxxx	val							
R3	kxxx	val		val	val			val	
	…								
	rxxx	val	val	val	val	val	val		

HBase Data Storage - Cells

- Data is stored in **Key value** format
- Value for each cell is specified by complete coordinates:
- (Row key, ColumnFamily, Column Qualifier, timestamp) => Value
  – RowKey:CF:Col:Version:Value
  – smithj:data:city:1391813876369:nashville

Cell Coordinates= Key **Value**

Column Key

Row key	Column Family	Column Qualifier	Timestamp	Value
Smithj	data	city	1391813876369	nashville

Logical Data Model vs Physical Data Storage

Logical Model

RowKey	CF1		CF2	
	colA	colB	colA	colC
ra	1		2	
rxxxx				
rxxx				

- Data is stored in Key Value format
- Key Value is stored for each Cell
- Column families data are stored in separate files

Key

Value

Row Key	CF1:Col	version	value
ra	cf1:cola	1	1

Physical Storage

Key

Value

row Key	CF2:Col	version	value
ra	cf2:cola	1	2

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

- Version
  - **each** put, delete adds new cell, new version
  - A **long**
    - by **default** the current **time** in milliseconds if no version specified
  - **Last 3** versions are stored by **default**
    - **Configurable** by column family
  - You can **delete** specific cell **versions**
  - When a cell exceeds the **maximum** number of versions, the extra records are removed

Key	CF1:Col	version	value
ra	cf1:cola	3	3
ra	cf1:cola	2	2
ra	cf1:cola	1	1

Table Physical View

Physically data is stored per Column family as a sorted map

- Ordered by **row key, column qualifier** in ascending order
- Ordered by **timestamp** in descending order

Sorted by Row key and Column

Row key	Column qualifier	Cell value	Timestamp **(long)**
Row1	CF1:colA	value3	*time7*
Row1	CF1:colA	value2	*time5*
Row1	CF1:colA	value1	*time1*
Row10	CF1:colA	value1	*time4*
Row 10	CF1:colB	value1	*time4*

Sorted in descending order

Logical Data Model vs Physical Data Storage

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

Logical Model

Column families are stored separately
Row keys, Qualifiers are sorted **lexicographically**

Physical Storage

Key Value

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 Value

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

SortedMap<Key, Value>

Table
Map of Rows

Map of CF

Map of columns

Map of cells

SortedMap<RowKey,
SortedMap< ColumnFamily,
SortedMap< ColumnName,
SortedMap < version, 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 Table SortedMap<Key, Value>

<ra,<cf1, <ca, <v1, 1>>
<cf2, <ca, <v1, 2>>>
<rb,<cf1, <cb, <v2, 4>
<v1, 3>>>
<rc,<cf1, <ca, <v1, 5>>
<cf2, <ca, <v2, 7>>
<ca, <v1, 6>>
<cd, <v1, 8>>>

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

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):**

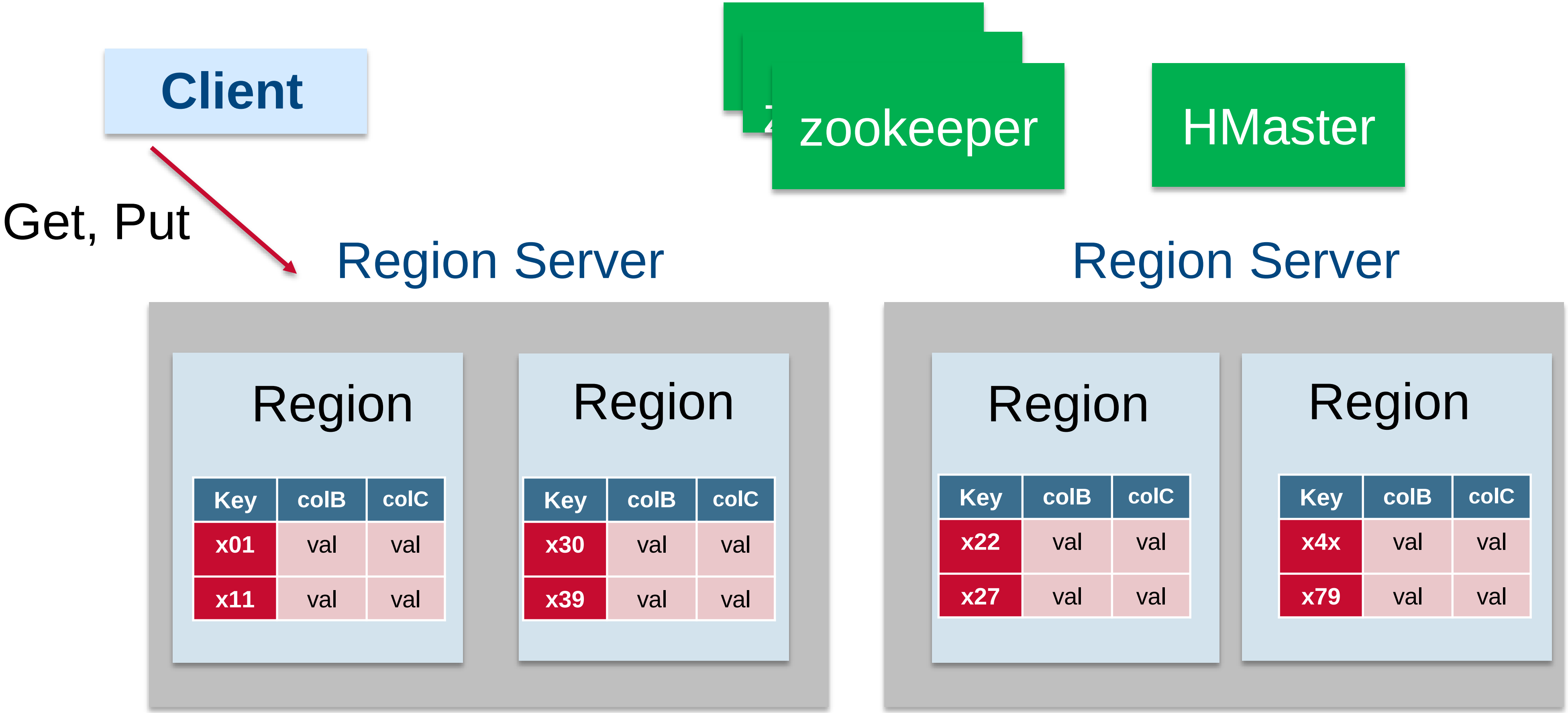
put	Inserts data into rows (both create and update)
get	Accesses data from one row
scan	Accesses data from a range of rows
delete	Delete a row or a range of rows or columns

HBase Architecture

Data flow for Writes, Reads
Designed to Scale

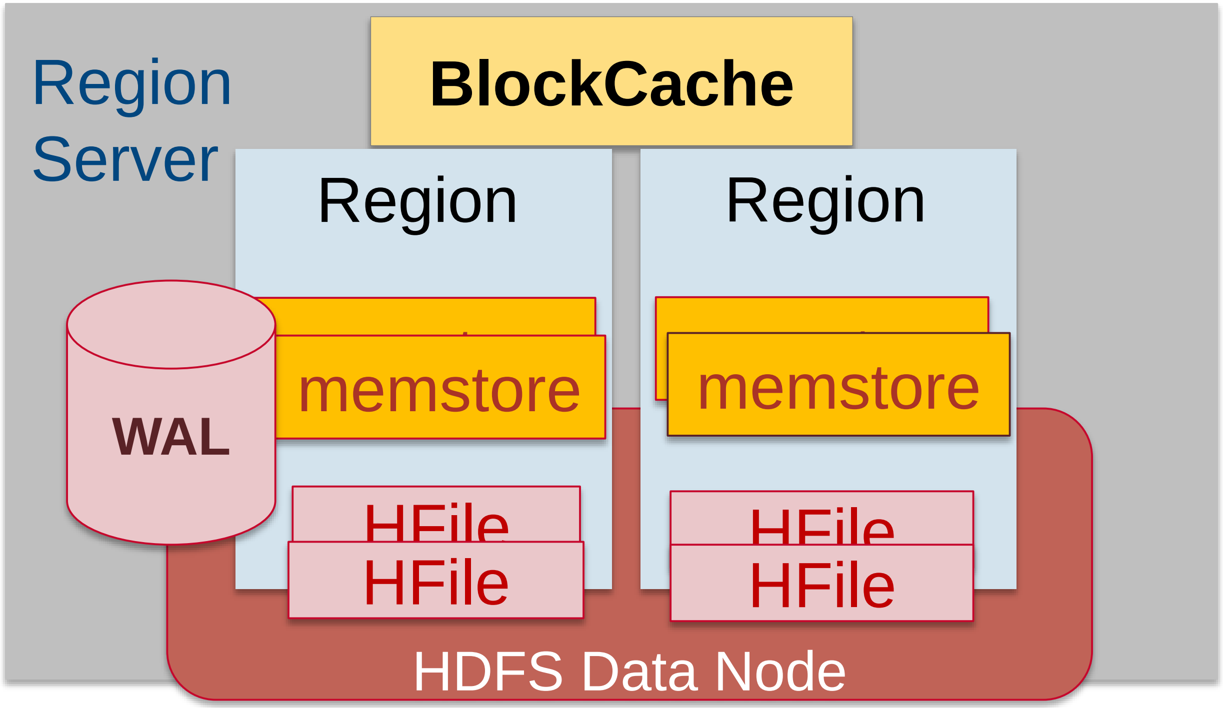
What is a Region?

- Tables are partitioned into **key ranges (regions)**
- Region servers serve data for reads and writes
  - For the **range of keys** it is responsible for

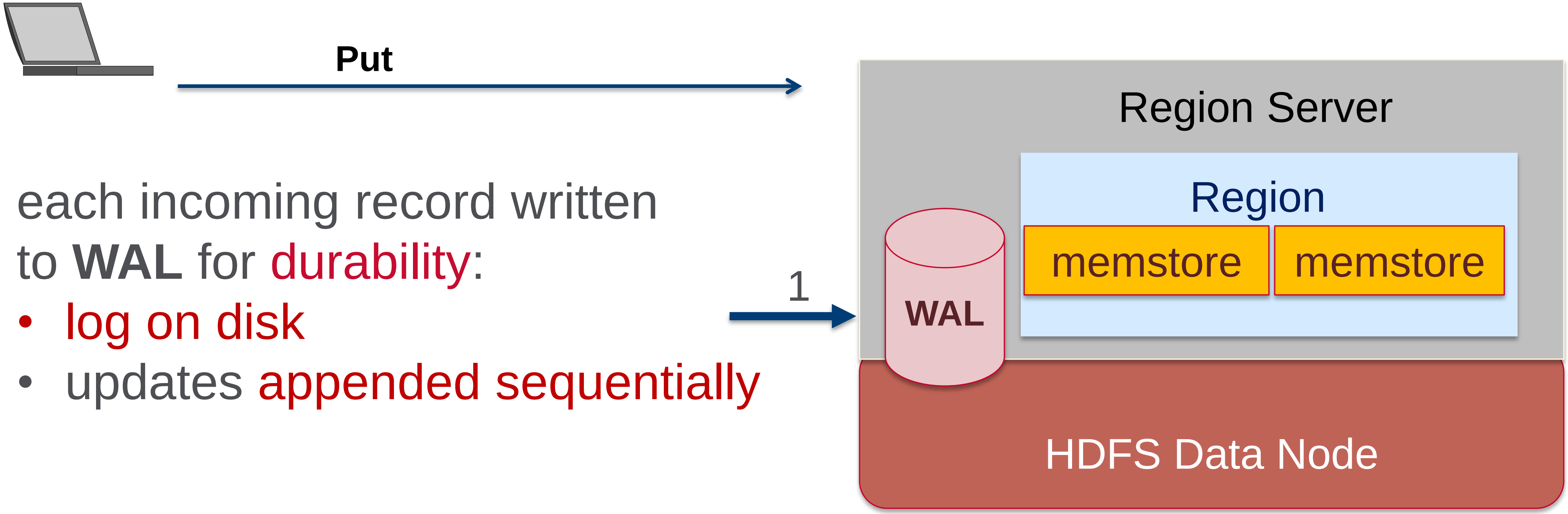


Region Server Components

- WAL: write ahead log on disk (commit log), Used for recovery
- BlockCache: Read Cache, **L**east **R**ecently **U**sed evicted
- MemStore: Write Cache, sorted keyValue updates.
- Hfile: sorted KeyValues on disk



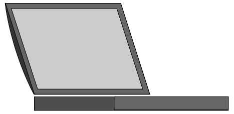
HBase Write Steps



each incoming record written to **WAL** for durability:

- log on disk
- updates appended sequentially

HBase Write Steps

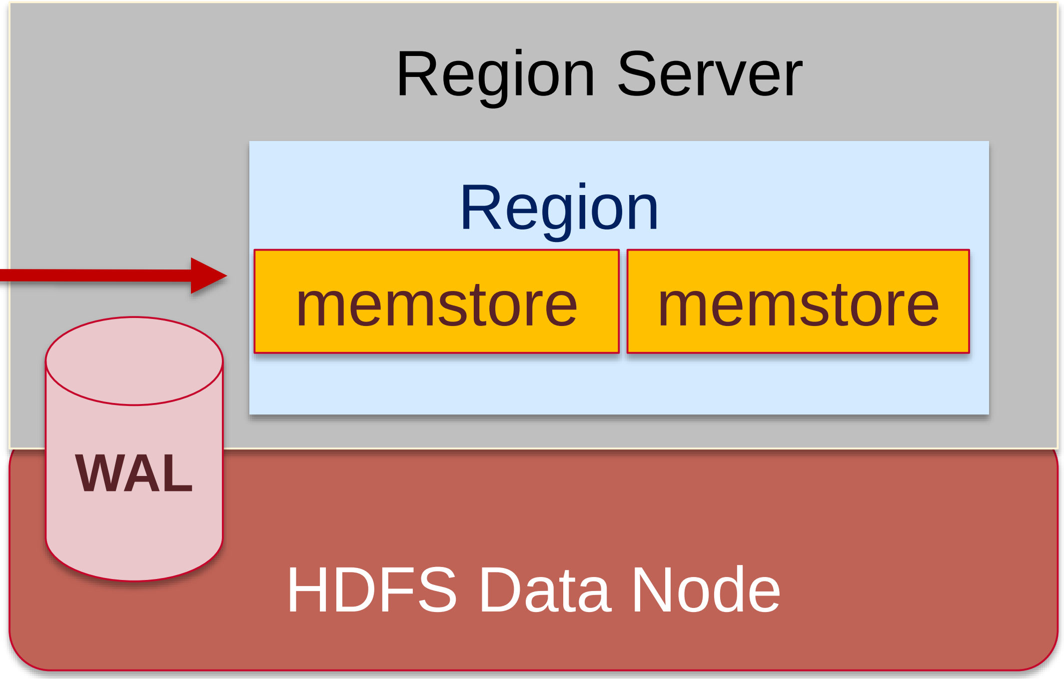


Put

Next updates are written to the **Memstore**:

- write cache
- in-memory
- sorted **list of KeyValue** updates

2



Ack

Updates **quickly sorted in memory** are available to queries after put returns

HBase Memstore

- **in-memory**
- **sorted list of Key → Value**
- **One per column family**
- Updates **quickly sorted in memory**

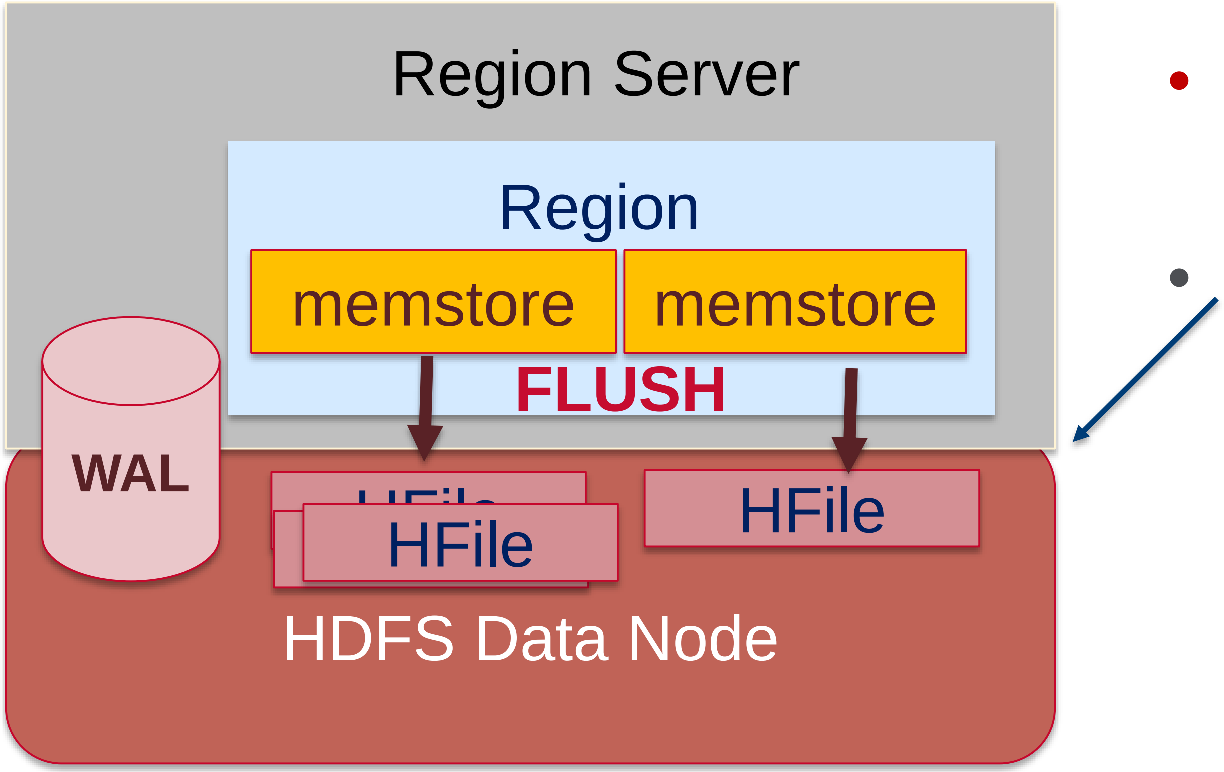
Key — Value

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 Region Flush



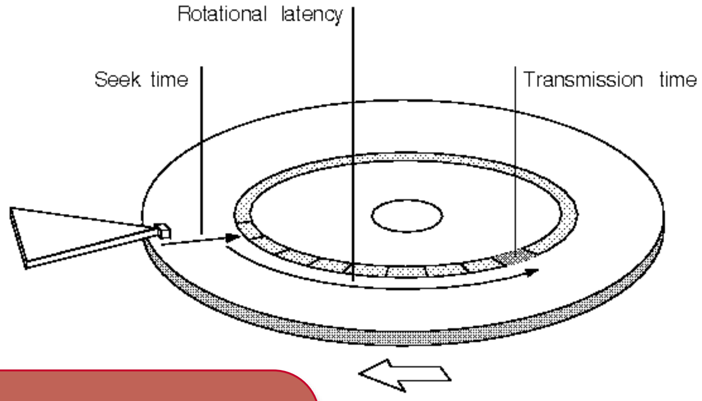
When **1 Memstore** is **full:**

- **all memstores** in region **flushed to new Hfiles** on **disk**
- Hfile: sorted list of key → values On disk

HBase HFile

- On disk **sorted** list of key → values
- **One per column family**
- **Flushed quickly to file**
  - **Sequential write**

Sequential write



HFile | HFile

HDFS Data Node

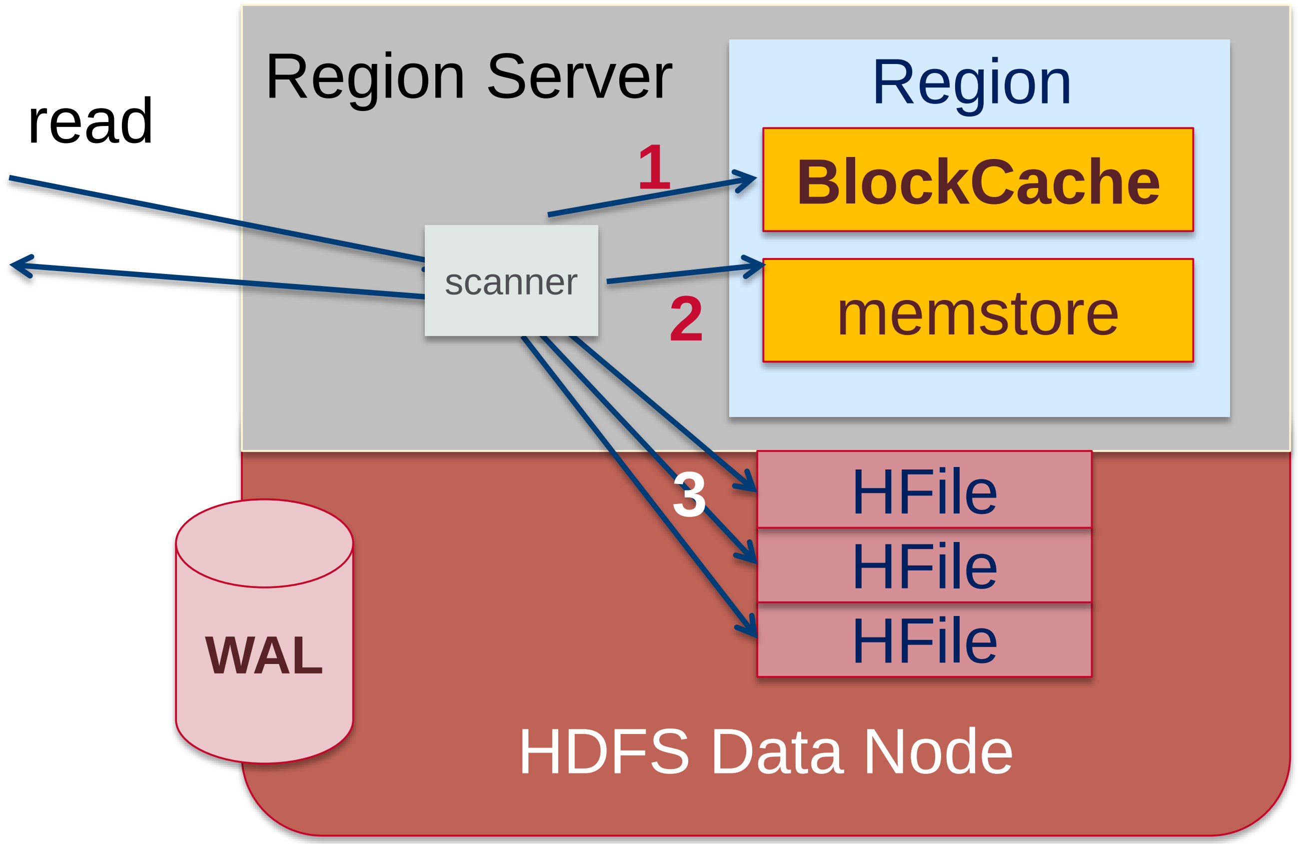
Key | **Value**

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 Read Merge from Memory and Files

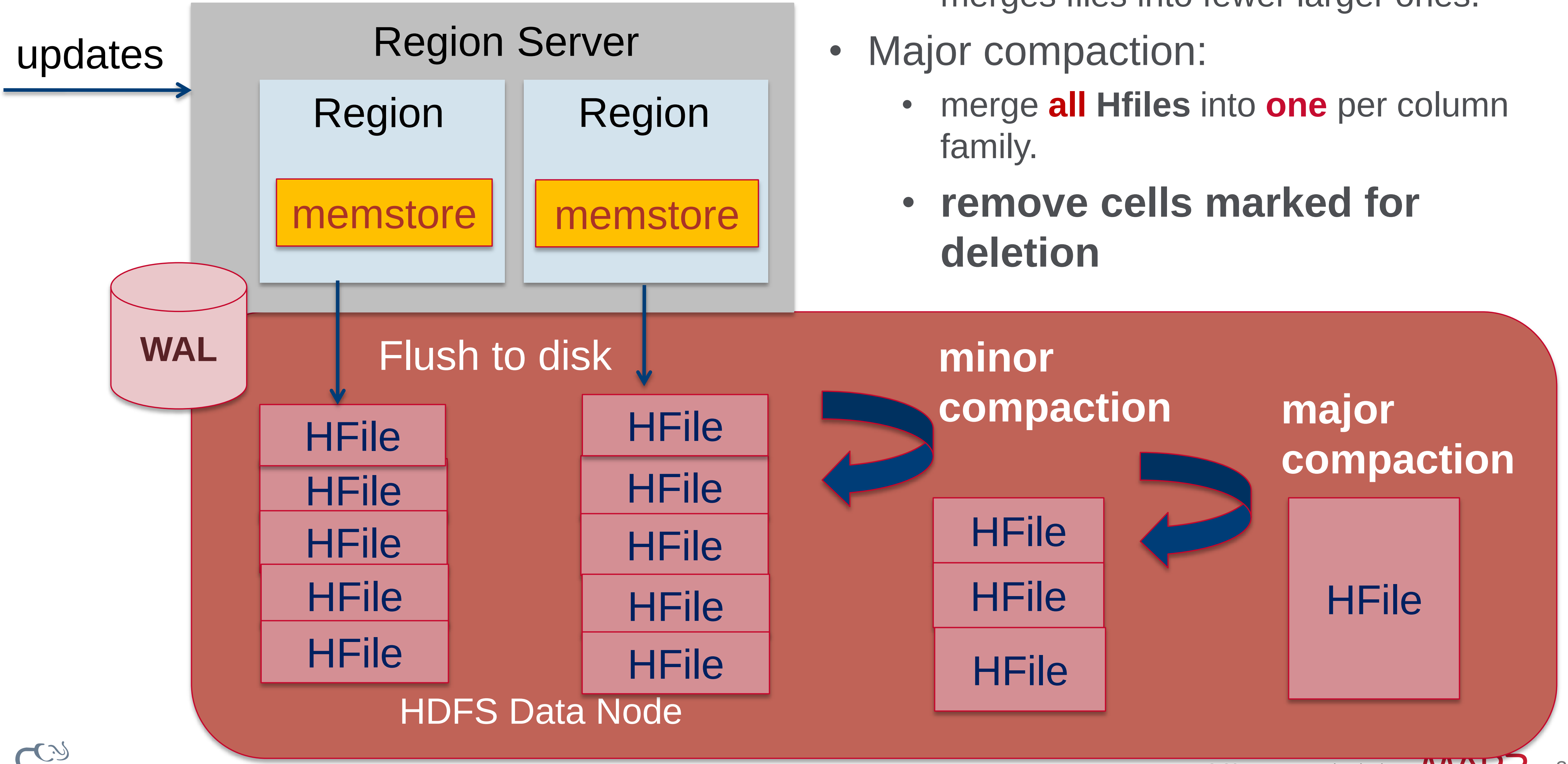


Get or Scan searches for Row Cell KeyValues:

1. Block Cache ((Memory)
2. Memstore (Memory)
3. Load HFiles from Disk into Block Cache based on indexes and bloomfilters

- MemStore creates multiple **small store files** over time when **flushing.**
- When a get/scan comes in, multiple files have to be examined

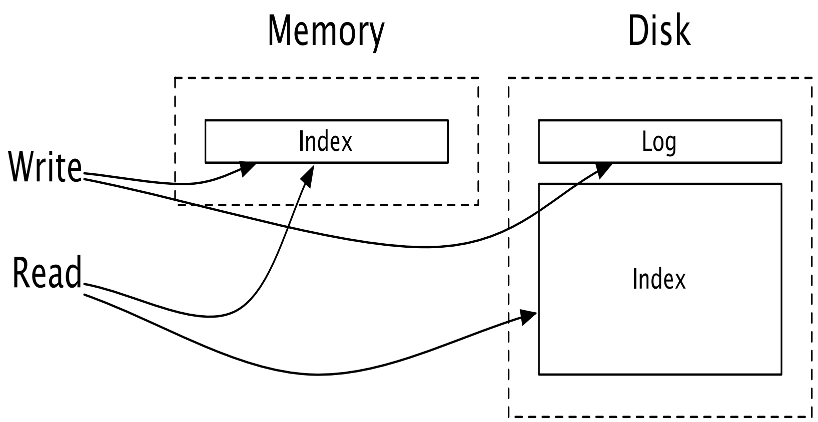
HBase Compaction



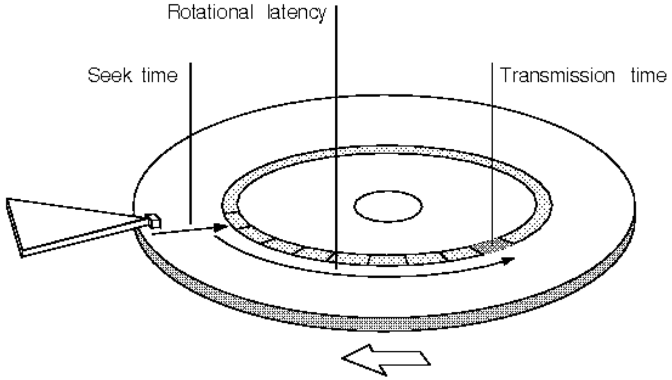
- **minor compaction:**
  - merges files into fewer larger ones.
- Major compaction:
  - merge **all Hfiles** into **one** per column family.
  - **remove cells marked for deletion**

HBase Background: Log-Structured Merge Trees

- Traditional Databases use B+ trees:
  - **expensive** to update
- HBase: Log Structured Merge Trees
  - **Sequential writes**
    - Writes go to **memory** And WAL
    - **Sorted memstore flushes** to **disk**
  - **Sequential Reads**
    - From **memory, index, sorted disk**



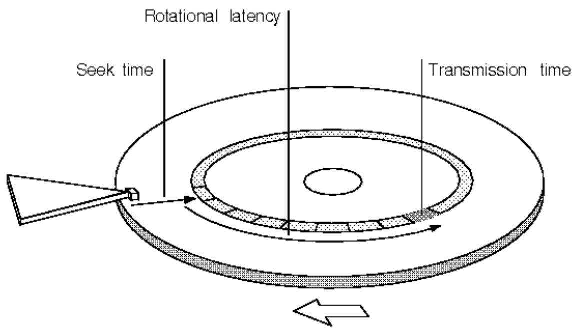
predictable disk seeks



Data Model for Fast Writes, Reads

- Predictable disk lay out
- Minimize disk seek
- Get, Put by row key: **fast** access
- Scan by row key range: stored **sorted**, **efficient sequential** access for **key range**

Minimize disk seek



Get key

Scan start key,
Stop key

Region1 Key Range
ra
rx

Region

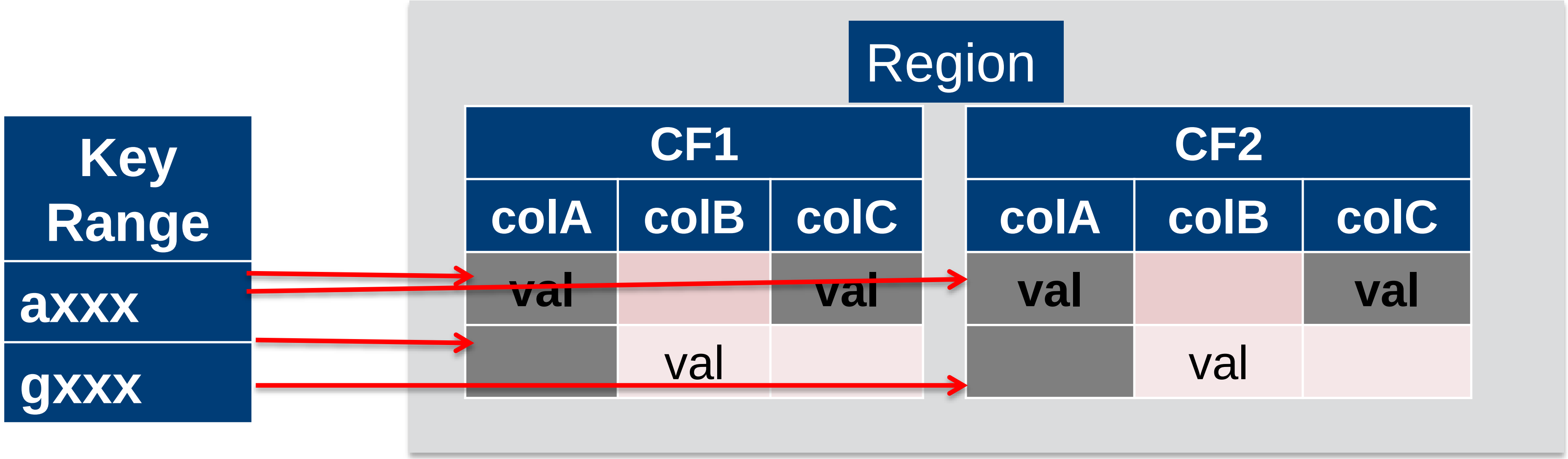
Key	CF1:Col	version	value
ra	cf1:ca	v1	1
rb	cf1:cb	v2	4
rb	cf1:cb	v1	3
rc	cf1:ca	v1	5

Region Server

R1

S3

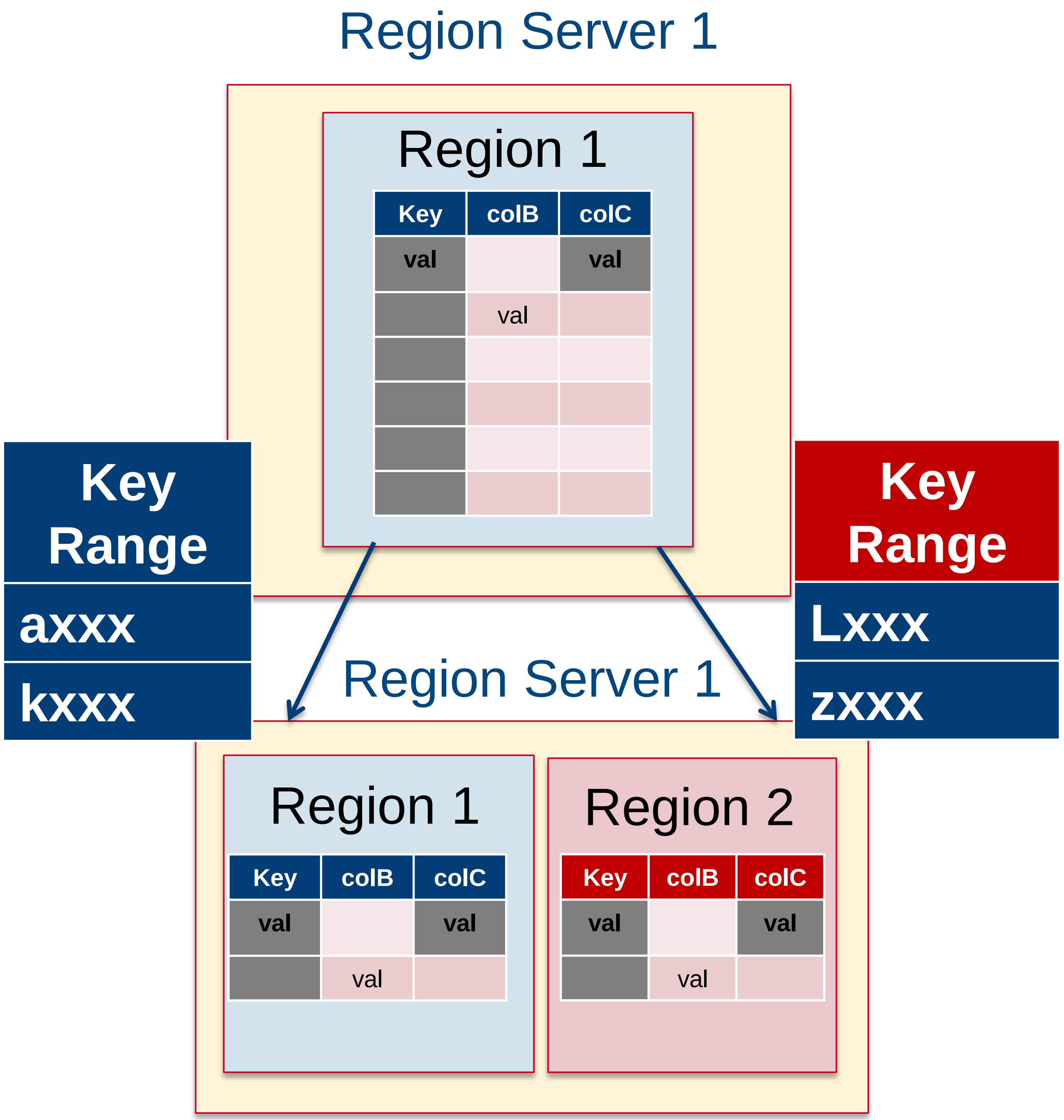
Region = contiguous keys



- **Regions** fundamental **partitioning/sharding** object.
- By **default**, on **table creation 1 region** is created that holds the entire key range.
- When region becomes **too large**, **splits** into **two** child regions.
- Typical region size is a few GB, sometimes even 10G or 20G

Region Split

- The RegionServer splits a region
- daughter regions
  - each with ½ of the regions **keys**.
  - opened in parallel on **same server**
- reports the split to the Master



HBase Use Cases

3 Main Use Case Categories

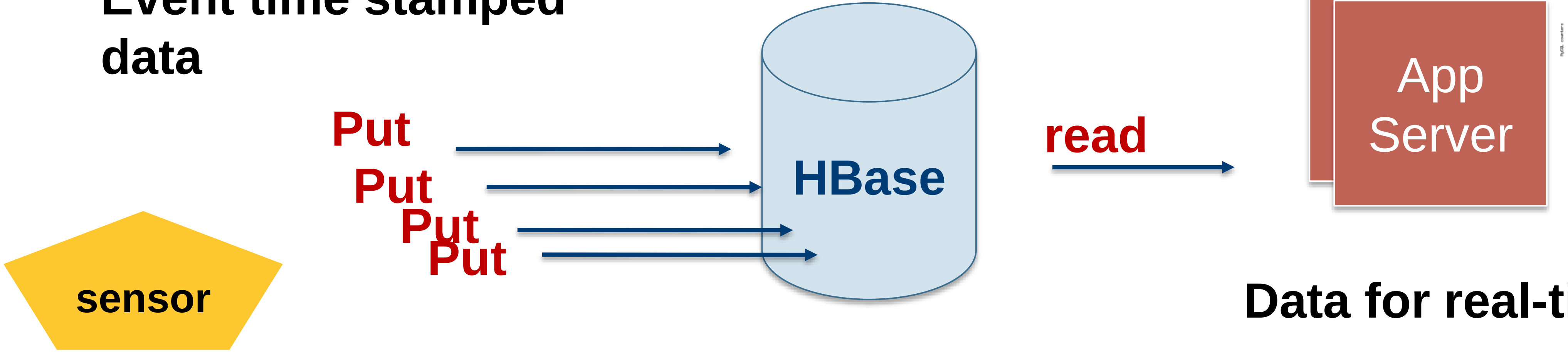
- Capturing Incremental data --Time Series Data
  - Hi Volume, Velocity Writes
- Information Exchange, Messaging
  - Hi Volume, Velocity Write/Read
- Content Serving, Web Application Backend
  - Hi Volume, Velocity Reads

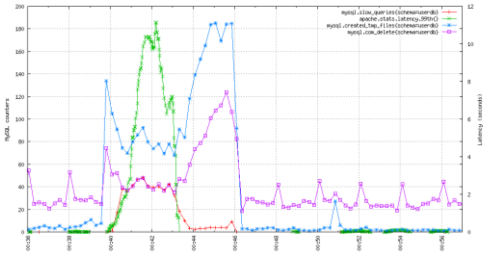
3 Main Use Case Categories

- Time Series Data, Stuff with a Time Stamp
  - Sensor, System Metrics, Events, log files
  - Stock Ticker, User Activity
  - Hi Volume, Velocity Writes

OpenTSDB

Event time stamped data

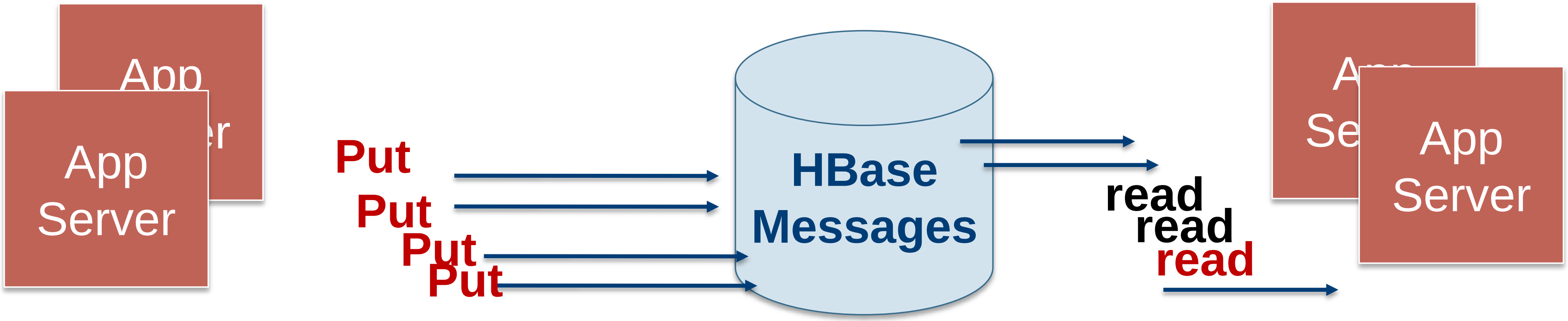




Data for real-time monitoring.

3 Main Use Case Categories

- Information Exchange
  - email, Chat, Inbox: **Facebook**
  - Hi Volume, Velocity Write/Read

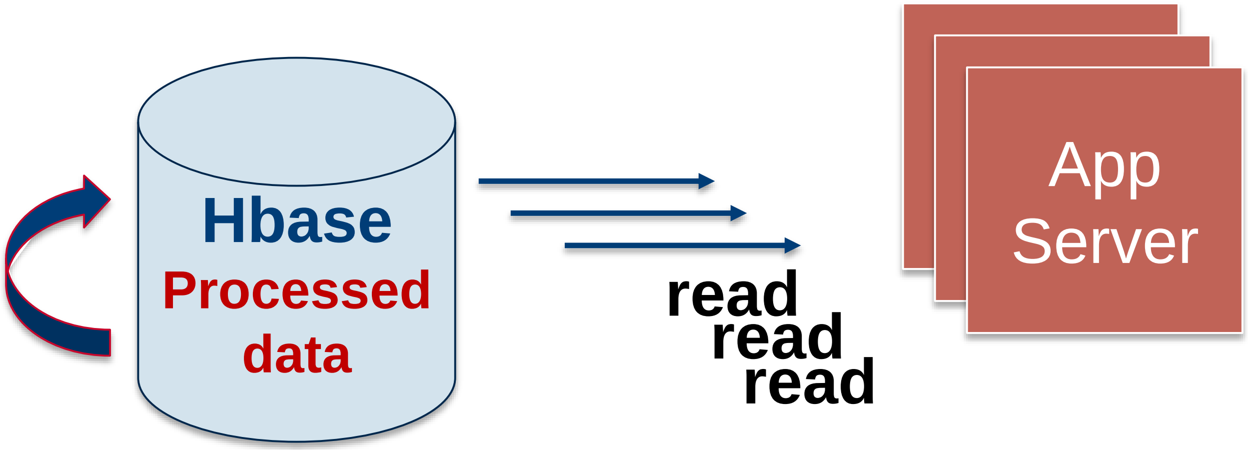


https://www.facebook.com/UsingHbase

3 Main Use Case Categories

- Content Serving, Web Application Backend
  - Online Catalog: Gap, World Library Catalog.
  - Search Index: ebay
  - Online Pre-Computed View: Groupon, Pinterest
  - Hi Volume, Velocity Reads

Bulk Import
Pre-Computed
Materialized View



Agenda

- Why do we need NoSQL / HBase?
- Overview of HBase & HBase data model
- HBase Architecture and data flow
- HBase Use Cases
- Demo/Lab using HBase Shell
  - Create tables and CRUD operations using MapR Sandbox
- HBase Java API to perform CRUD operations
  - Demo HBase Java API using Eclipse & MapR Sandbox

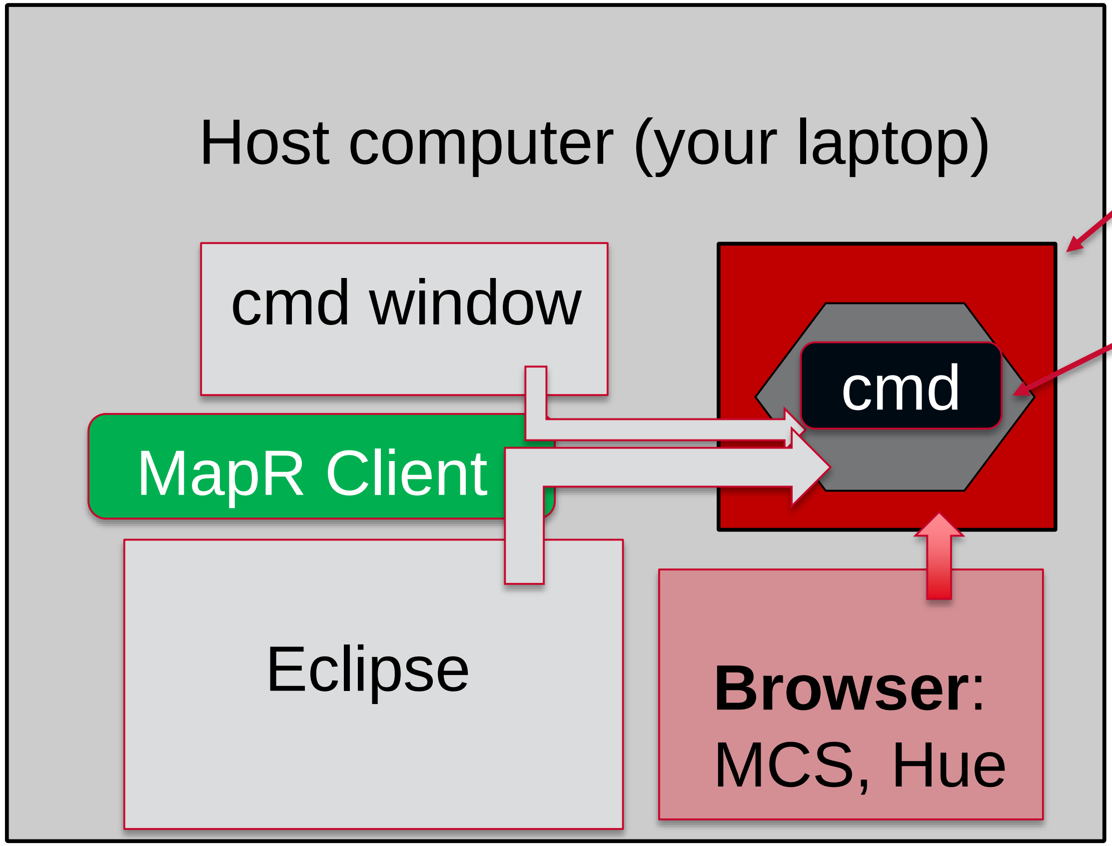
Prerequisite for Hands-On-Labs
Install MapR Sandbox

- Install a one-node MapR Sandbox on your laptop
- Install and configure Eclipse to develop HBase applications using Java API
- MapR Client is optional

Open   Hbase_Tutorial_3_2015.pdf

What is MapR Sandbox and how to use it

- MapR Sandbox is a fully functional single-node Hadoop cluster running on a virtual machine



Vmware or VirtualBox

MapR Sandbox

You can directly login and use terminal window to run Hadoop commands

Lab Exercise

See **Lab_Hbase_Shell.pdf**

Start MapR Sandbox and log into the cluster

[user: mapr,   passwd: mapr]

Use the HBase shell

>Hbase shell

hbase> ***help***

hbase> ***create '/user/mapr/mytable', {NAME =>'cf1'}***

hbase> ***put '/user/mapr/mytable', 'row1', 'cf1:col1', 'datacf1c1v1'***

hbase> ***get '/user/mapr/mytable', 'row1'***

hbase> ***scan '/user/mapr/mytable'***

hbase> ***describe '/user/mapr/mytable'***

Agenda

- Why do we need NoSQL / HBase?
- Overview of HBase & HBase data model
- HBase Architecture and data flow
- HBase Use Cases
- Demo/Lab using HBase Shell
  - Create tables and CRUD operations using MapR Sandbox
- HBase Java API to perform CRUD operations
  - Demo / Lab using Eclipse, HBase Java API & MapR Sandbox

HBase
Java API fundamentals to perform CRUD operations

Shoppingcart Application Requirements

- Need to create Tables: Shoppingcart & Inventory
- Perform **CRUD** operations on these tables
  - Create, Read, Update, and Delete items from these tables

Inventory & Shoppingcart Tables

Perform checkout operation for Mike

Inventory Table

	CF “stock "
	quantity
Pens	**10**
Notepads	**21**
Erasers	10
Pencils	40

Shoppingcart Table

	CF “items"		
	pens	notepads	erasers
Mike	**1**	**2**	**3**
John	3	4	5
Mary	1	2	5
Adam	5	4	0

Java API Fundamentals

- **CRUD operations**
  - **Get, Put, Delete, Scan, checkAndPut, checkAndDelete, Increment**
  - KeyValue, Result, Scan – ResultScanner,
  - Batch Operations

CRUD Operations Follow A Pattern (mostly)

- **common pattern**
  - Instantiate object for an operation: `Put put = new Put(key)`
  - Add attributes to specify what to insert: `put.add(…)`
  - invoke operation with HTable: `myTable.put(put)`

```
// Insert value1 into rowKey in columnFamily:columnName1
Put put = new Put(rowKey);
put.add(columnFamily, columnName1, value1);
myTable.put(put);
```

Shopping Cart Table

Shoppingcart Table

	CF “items"		
	erasers	notepads	pens
Mike	3	2	1

Physical Storage

Key	CF:COL	ts	value
Mike	items:erasers	1391813876369	3
Mike	items:notepads	1391813876369	2
Mike	items:pens	1391813876369	1

Put Operation

Key	CF:COL	ts	value
Mike	items:erasers	1391813876369	3
Mike	items:notepads	1391813876369	2
Mike	items:pens	1391813876369	1

adding multiple column values to a row

```
byte [] tableName = Bytes.toBytes("/path/Shopping");
byte [] itemsCF = Bytes.toBytes(“items");
byte [] penCol = Bytes.toBytes (“pens”);
byte [] noteCol = Bytes.toBytes (“notes”);
byte [] eraserCol = Bytes.toBytes (“erasers”);
HTableInterface table = new HTable(hbaseConfig, tableName);

Put put = new Put(“Mike”);
put.add(itemsCF, penCol, Bytes.toBytes(1));
put.add(itemsCF, noteCol, Bytes.toBytes(2));
put.add(itemsCF, eraserCol, Bytes.toBytes(3));

table.put(put);
```

Get Example

Key	CF:COL	ts	value
Mike	items:erasers	1391813876369	3
Mike	items:notepads	1391813876369	2
Mike	**items:pens**	**1391813876369**	**1**

```
byte [] tableName = Bytes.toBytes("/user/user01/shoppingcart");
byte [] itemsCF = Bytes.toBytes(“items");
byte [] penCol = Bytes.toBytes (“pens”);
HTableInterface table = new HTable(hbaseConfig, tableName);

    Get get = new Get(“Mike”);

    get.addColumn(itemsCF, penCol);

    Result result = myTable.get(get);

    byte[] val = result.getValue(itemsCF, penCol);

    System.out.println("Value: " + Bytes.toLong(val));    //prints 1
```

Result Class

- A **Result** instance **wraps** data from a **row returned** from a **get** or a **scan** operation. Result wraps KeyValues

	Items:erasers	Items:notepads	Items:pens
Adam	0	4	5

- **Result** toString() looks like this :

  keyvalues={Adam/items:erasers/1391813876369/Put/vlen=8/ts=0,
  Adam/items:notepads/1391813876369/Put/vlen=8/ts=0,
  Adam/items:pens/1391813876369/Put/vlen=8/ts=0}

- The **Result** object provides methods to return **values**

```
byte[] b = result.getValue(columnFamilyName,columnName1);
```

http://hbase.apache.org/0.94/apidocs/org/apache/hadoop/hbase/client/Result.html

KeyValue – The Fundamental HBase Type

- A KeyValue instance is a cell instance
  - Contains **Key (cell coordinates)** and the **Value** (**data**)
- Cell coordinates: Row key, Column family, Column qualifier, Timestamp
- **KeyValue** toString() looks like this :

  Adam/items:erasers/1391813876369/Put/vlen=8/

Key =Cell Coordinates

Value

Row key	Column Family	Column Qualifier	Timestamp	Value
Adam	items	erasers	1391813876369	0

Bytes class

http://hbase.apache.org/0.94/apidocs/org/apache/hadoop/hbase/util/Bytes.html

- org.apache.hadoop.hbase.util.Bytes
- Provides methods to convert Java types to and from byte[] arrays
- Support for
  - String, boolean, short, int, long, double, and float

```
byte[]  bytesTable = Bytes.toBytes("Shopping");
String  table = Bytes.toString(bytesTable);

byte[] amountBytes = Bytes.toBytes(1000l);
long   amount = Bytes.toLong(amount);
```

Scan Operation – Example

```
byte[] startRow=Bytes.toBytes("Adam");
byte[] stopRow=Bytes.toBytes("N");

Scan s = new Scan(startRow, stopRow);

scan.addFamily(columnFamily);

ResultScanner rs = myTable.getScanner(s);
```

ResultScanner - Example

Resultscanner provides **iterator-like** functionality

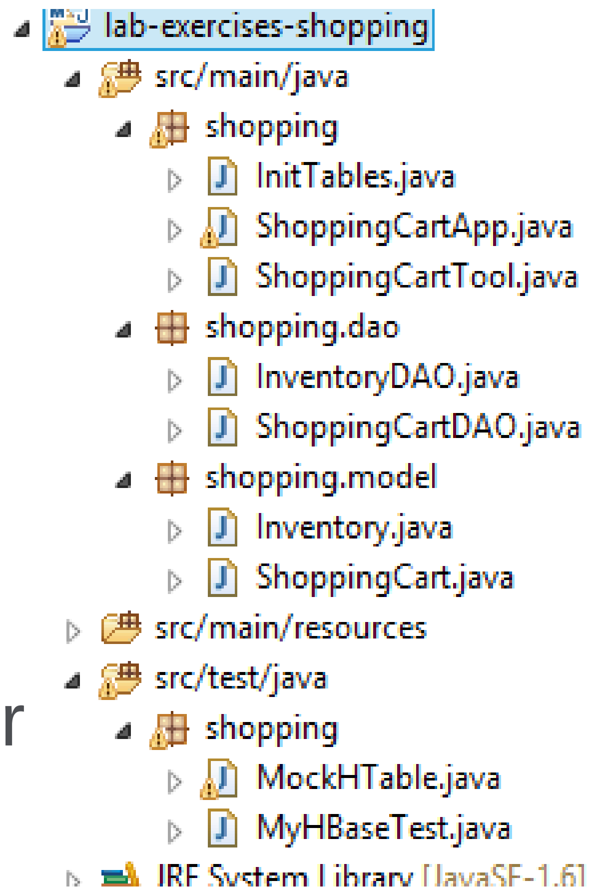
```
Scan scan = new Scan();
scan.addFamily(columnFamily);
ResultScanner scanner = myTable.getScanner(scan);
try {
    for (Result res : scanner) {
        System.out.println(res);
    }
} catch (Exception e) {
    System.out.println(e);
} finally {
    scanner.close();
}
```

Calls scanner.next()

Always put in finally block

Lab Exercise Program Structure

- **`ShoppingCartApp`**– main class
- **`InventoryDAO`** – A DAO for the Inventory CRUD functionality
- **`ShoppingCartDAO`** – A DAO for the Inventory CRUD functionality
- **`Inventory`** – A Java object that holds data for a single Inventory row
- **`ShoppingCart`** – A Java object that holds data for a single Inventory row
- **MockHtable** – in memory test hbase table, allows to run code, debug without hbase running on a cluster or vm.

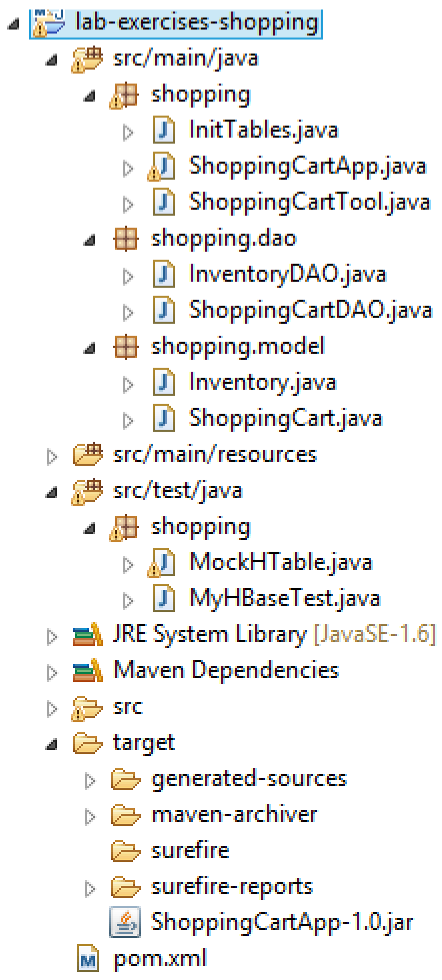


Lab Exercise

- See Lab_3_Java_API.pdf
  - Import the project “**lab-exercises-shopping**” into Eclipse
  - Setup creates Inventory and Shoppingcart Tables and inserts data
  - Use Get, Put, Scan, and Delete operations

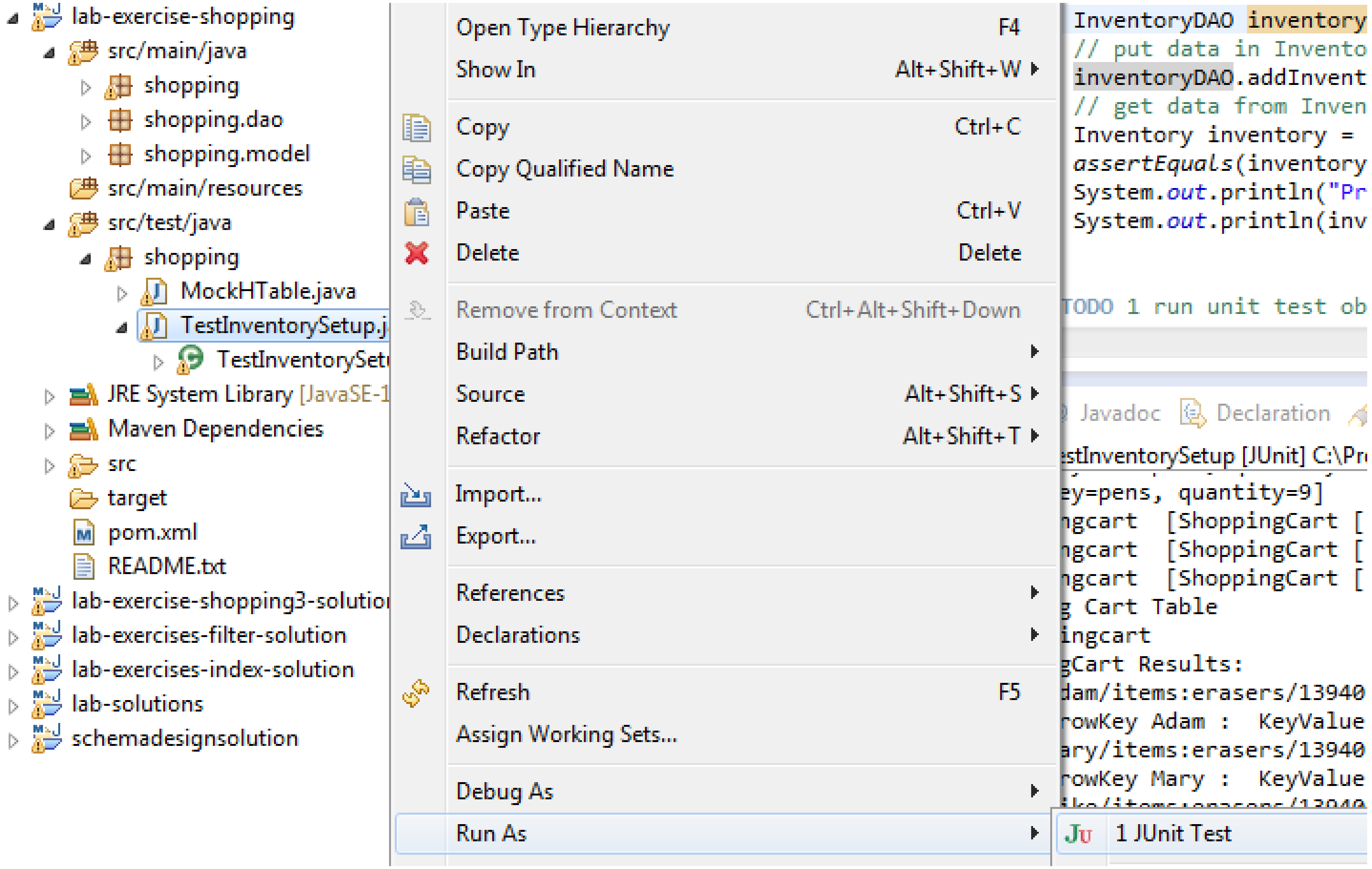
Lab: Import, build

- **Download** the code
- **Import** Maven project lab-exercises-shopping into Eclipse
- **Build** : Run As -> Maven Install



Lab: run TestInventorySetup JUnit

- **Select Test Class, Then Run As -> JUnit Test**
- **Uses MockHTable** **https://gist.github.com/agaoglu/613217**



Summary

- Why do we need NoSQL / HBase?
- Overview of HBase & HBase data model
- HBase Architecture and data flow
- HBase Use Cases
- Demo/Lab using HBase Shell
  - Create tables and CRUD operations using MapR Sandbox
- HBase Java API to perform CRUD operations
  - Demo / Lab using Eclipse, HBase Java API & MapR Sandbox

References

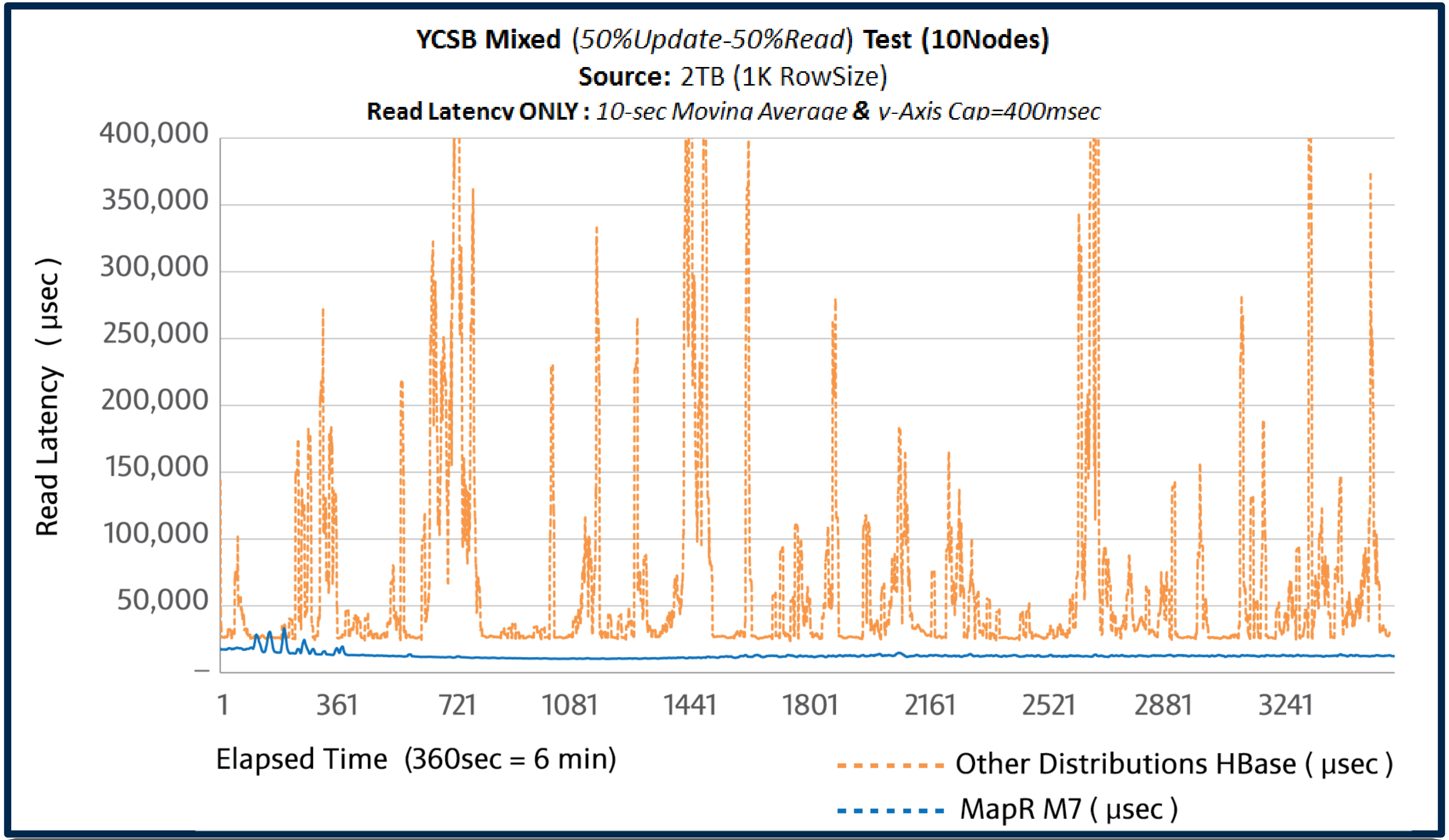
- http://hbase.apache.org/
- http://hbase.apache.org/0.94/apidocs/
- http://hbase.apache.org/0.94/apidocs/org/apache/hadoop/hbase/client/package-summary.html
- http://hbase.apache.org/book/book.html

- http://doc.mapr.com/display/MapR/MapR+Overview
- http://doc.mapr.com/display/MapR/M7+-+Native+Storage+for+MapR+Tables
- http://doc.mapr.com/display/MapR/MapR+Sandbox+for+Hadoop
- http://doc.mapr.com/display/MapR/Migrating+Between+Apache+HBase+Tables+and+MapR+Tables

MapR-DB: Fast, Integrated NoSQL

Operational Analytics	Resilience/Business Continuity	Performance/Scale
Hadoop-enabled architecture • Operational/architectural simplicity, analytics on live data • Hadoop-scale (petabytes)	**Proven production readiness** • Enterprise-grade HA/DR • Consistent snapshots • Integrated security	**Consistent performance at any scale** • High throughput (100M data points per second ingestion) • Consistent low latency, no compaction delays, even at 95th and 99th percentiles (< 10ms in YCSB)

Mapr-DB Performance: Consistent, Low Latency



YCSB Mixed *(50% Updated - 50% Read)*, **Test** *(10 Nodes)*, **Source:** *2TB (1K RowSize)*, **Read latency ONLY:** *10-sec Moving Average and y-Axis Cap=400msec*

Q&A

Engage with us!

@mapr	Twitter	Facebook	maprtech
mapr-technologies	LinkedIn	Google+	MapR
sreddy@mapr.com	Email	YouTube	maprtech