# Getting Started with HBase Application development

Sridhar Reddy sreddy@mapr.com
Carol McDonald cmcdonald@mapr.com

October 15th, 2014 http://answers.mapr.com/

# Objectives of this session

- What is HBase?
  - Why do we need NoSQL / HBase?
  - Overview of HBase & HBase data model
  - HBase Architecture and data flow
- How to get started
  - Demo/Lab using HBase Shell using MapR Sandbox
- Design considerations when migrating from RDBMS to HBase
- Developing Applications using HBase Java API
  - Demo/Lab to perform CRUD operations using put, get, scan, delete
  - How to work around transactions

# 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

**http://tinyurl.com/hbase-Oct-15-2014**

MapRSandboxInstallGuide.pdf

Hbase_Tutorial_LabDoc.pdf

# http://tinyurl.com/hbase-Oct-15-2014

| | |
|---|---|
| exercises.zip | archive |
| GettingStartedWithHBase_HadoopWorld.pdf | document |
| Hbase_Tutorial_LabDoc.pdf | document |
| MapRClientInstallGuide.pdf | document |
| MapRSandboxInstallGuide.pdf | document |
| README.pdf | document |
| README.txt | document |
| solutions.zip | archive |

# Why do we need NoSQL / HBase?

## Relational database model

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

- Entities map to tables, normalized
- **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**

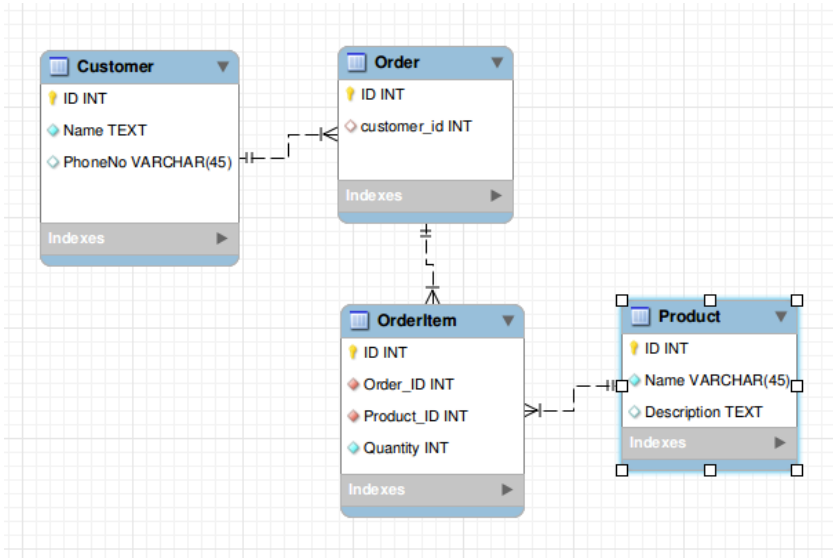
| Customer ID | Order Number |
|---|---|
| 1 | 1001 |
| 1 | 1002 |
| 2 | 1003 |
| 3 | 1004 |
| 4 | 1005 |
| 5 | 1006 |
| 5 | 1007 |

**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 |

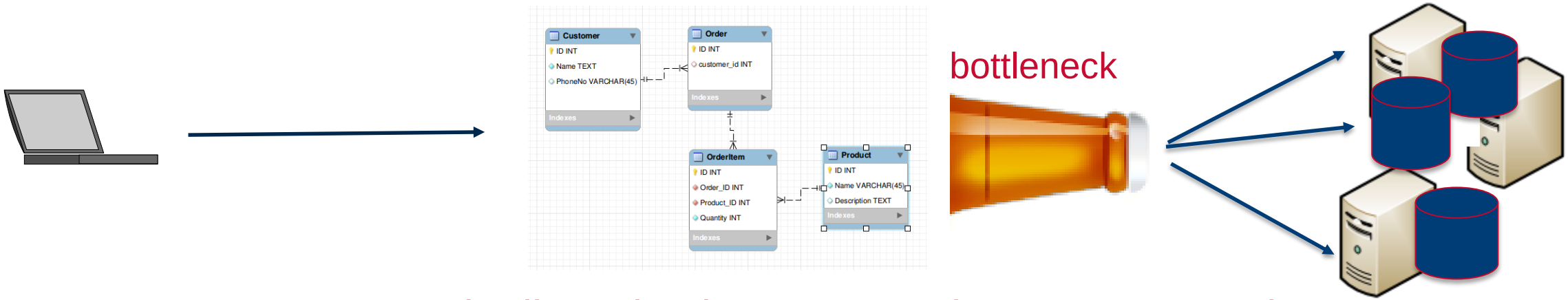
# Why do we need NoSQL / HBase?
## Relational Model

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



# 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
- ...

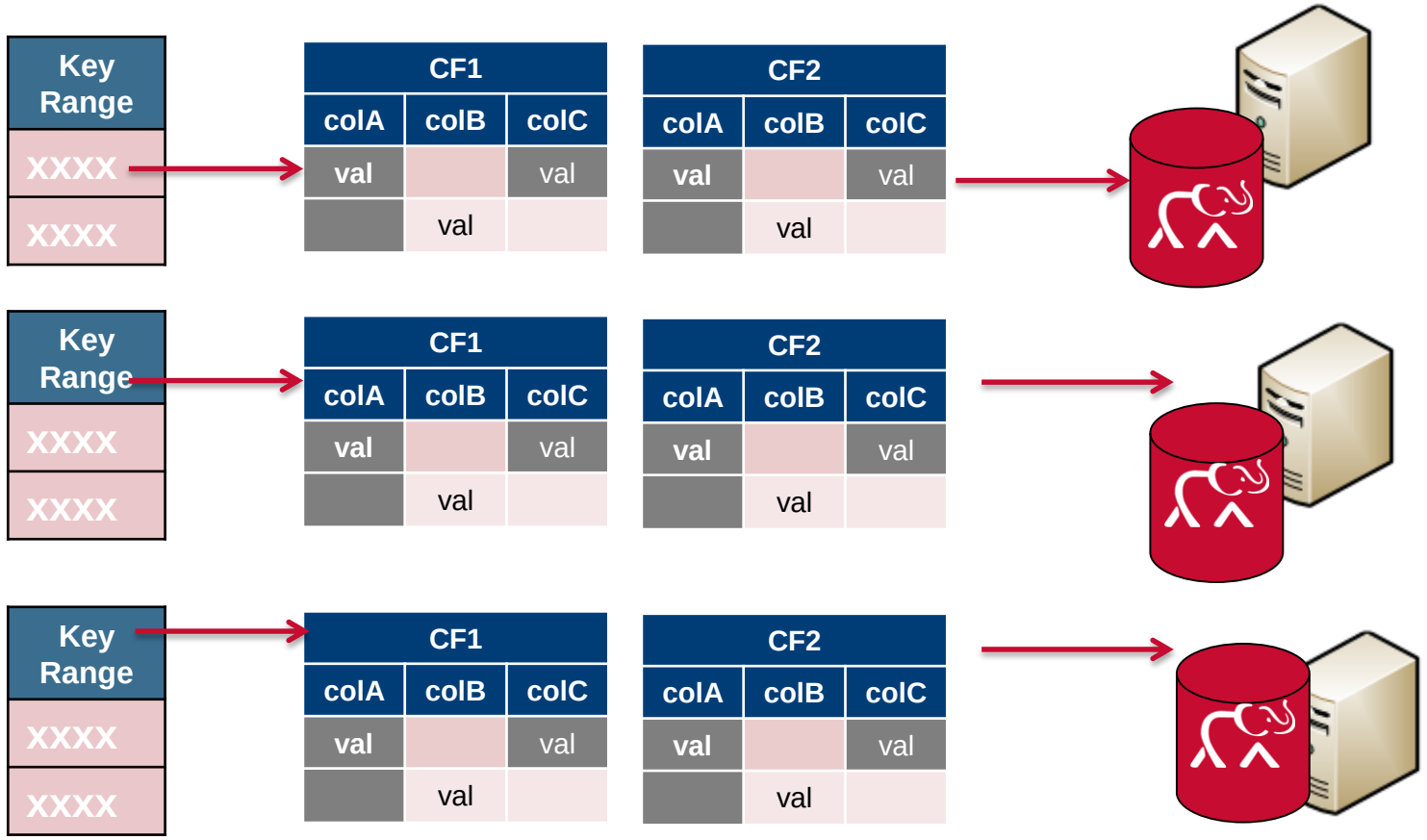
**Wide Column**
- **HBase**
- **MapR-DB**
- Hypertable
- Cassandra
- ...

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

# Hbase designed for Distribution, Scale, Speed

# HBase is a Distributed Database

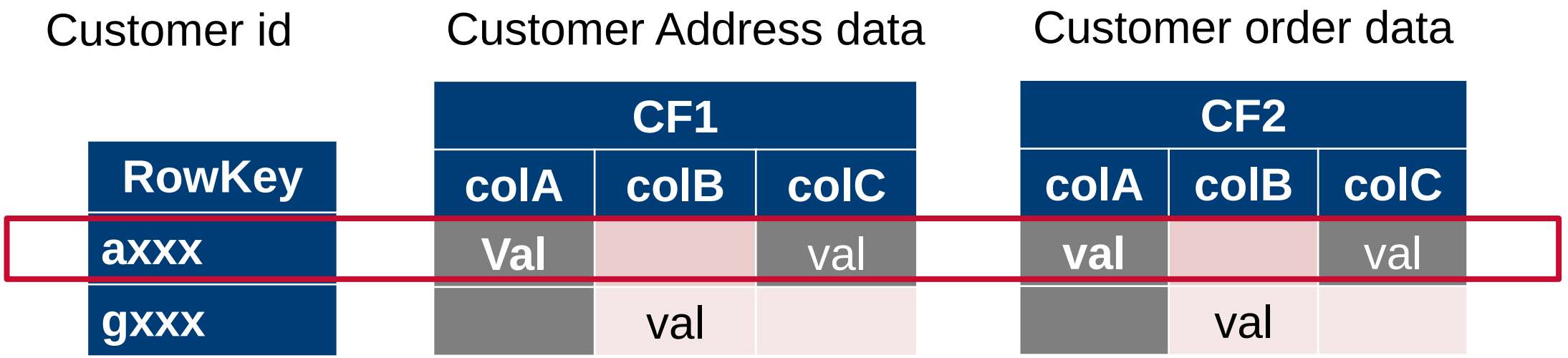
Put, Get by Key




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

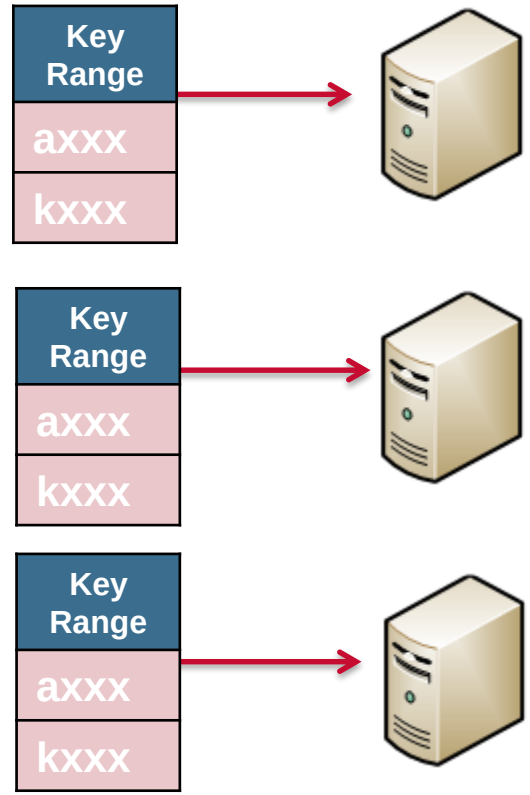



Data is accessed and stored together by RowKey

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

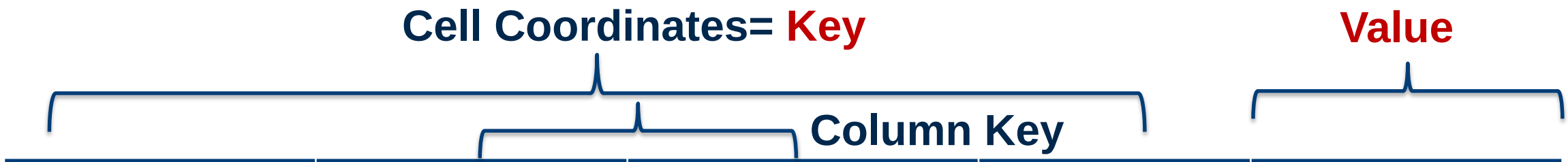
Row Keys: identify the rows in an HBase table

Columns are grouped into column families

| | Row Key | CF1 | | | CF2 | | | | … |
|---|---|---|---|---|---|---|---|---|---|
| | | colA | colB | colC | colA | colB | colC | colD | |
| R1 | **a**xxx | val | | val | val | | | val | |
| | … | | | | | | | | |
| | **g**xxx | val | | | val | val | val | | |
| R2 | **h**xxx | val | val | val | val | val | val | val | |
| | … | | | | | | | | |
| | **j**xxx | val | | | | | | | |
| R3 | **k**xxx | val | | val | val | | | val | |
| | … | | | | | | | | |
| | **r**xxx | 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




| 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

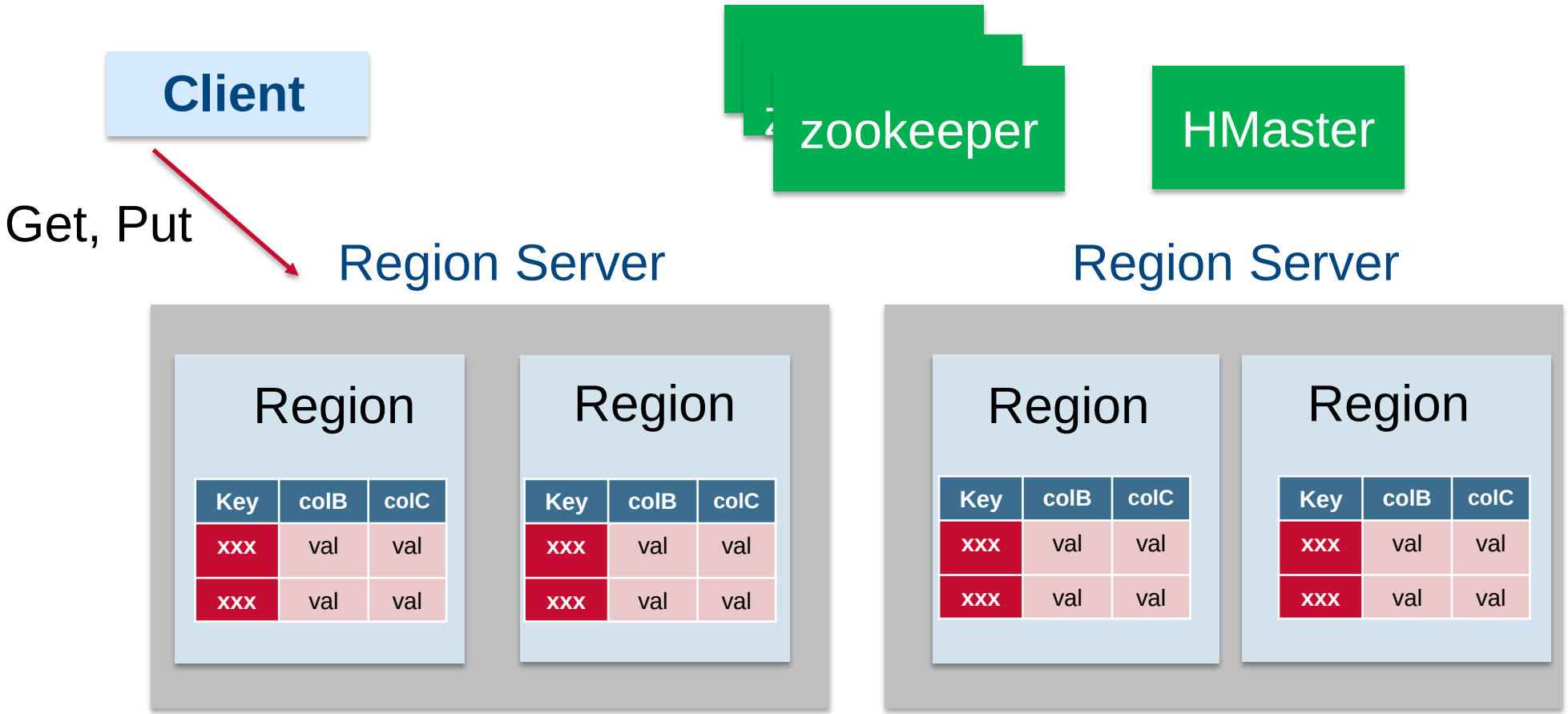
## Data flow for Writes, Reads

## Designed to Scale



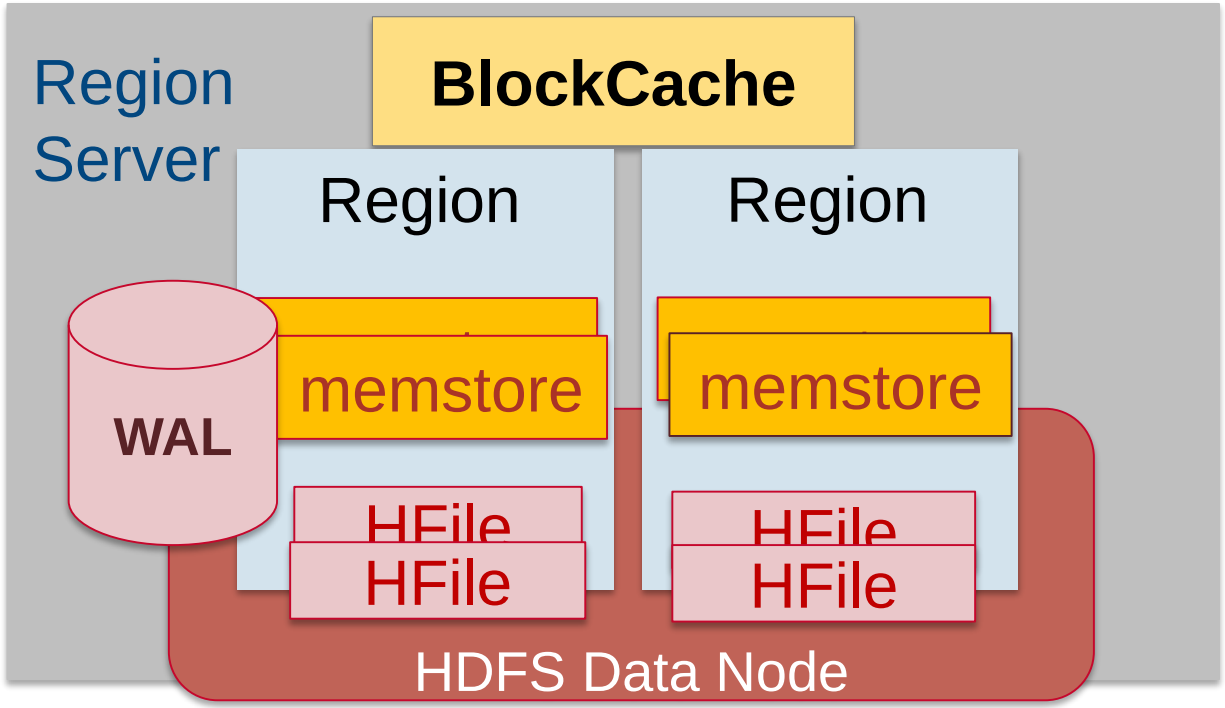
# What is a Region?

- 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

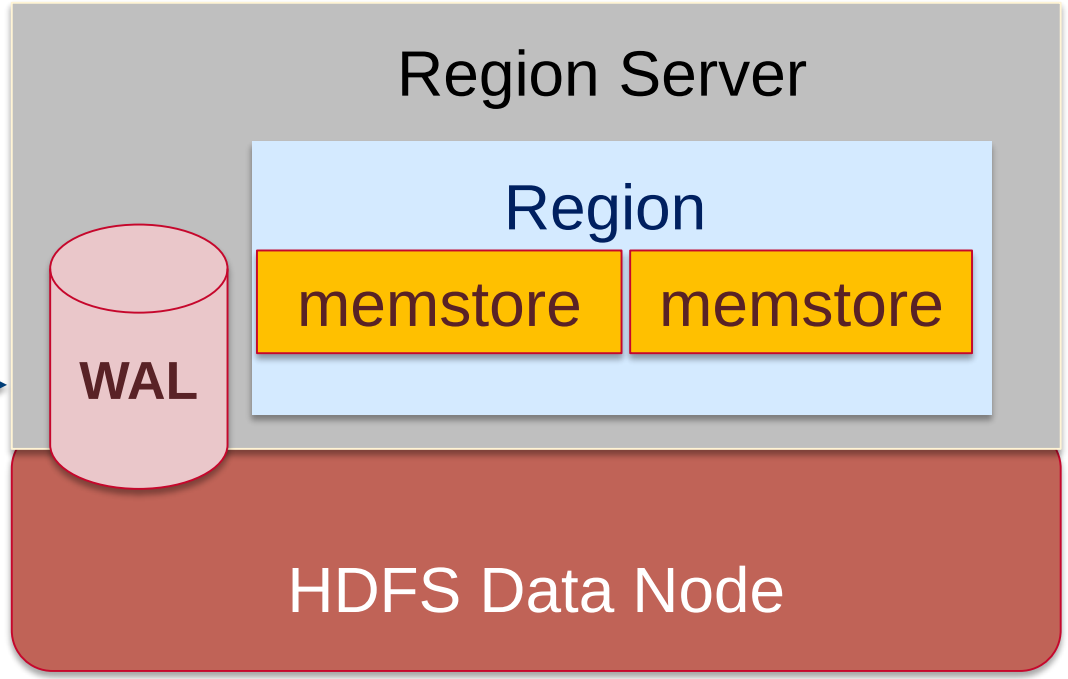


# HBase Write Steps

**Put**

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

- log on disk
- updates appended sequentially

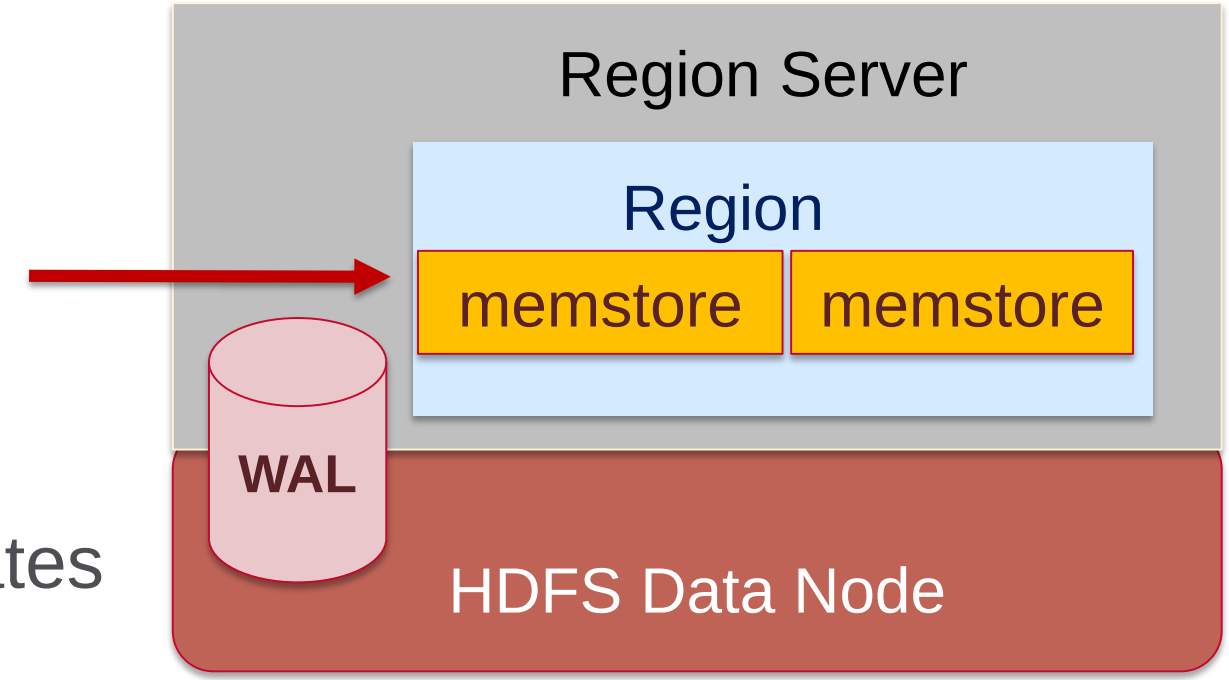


# HBase Write Steps

**Put**

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

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




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**

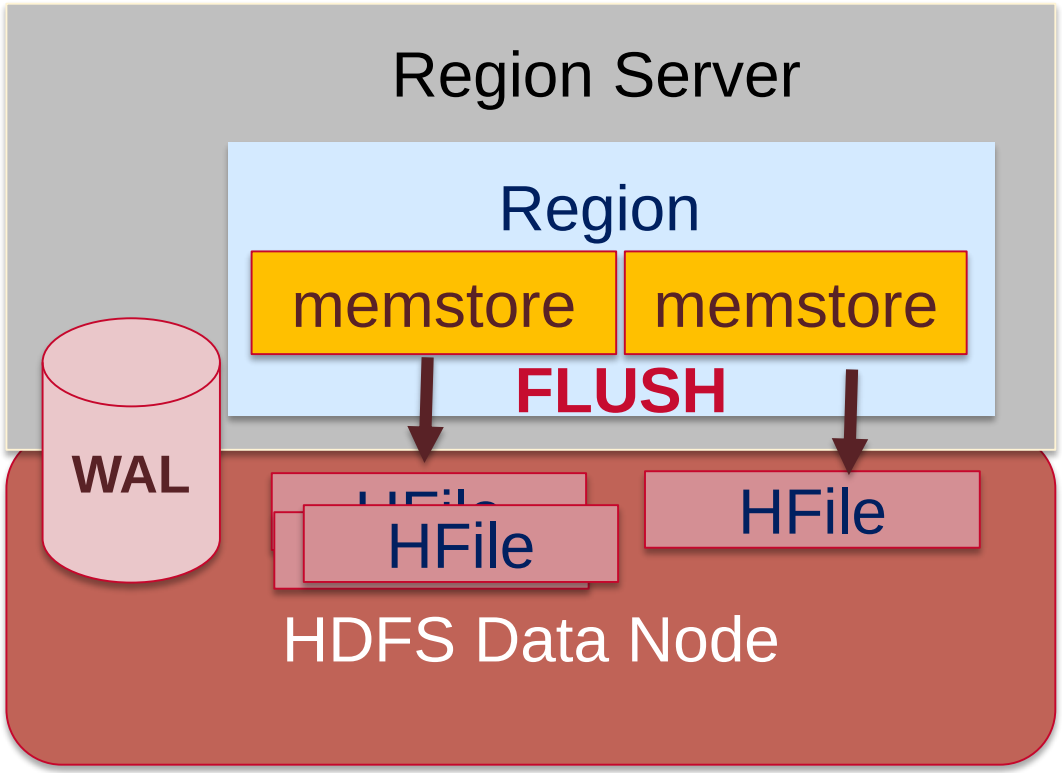

**Key** **Value**

| 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



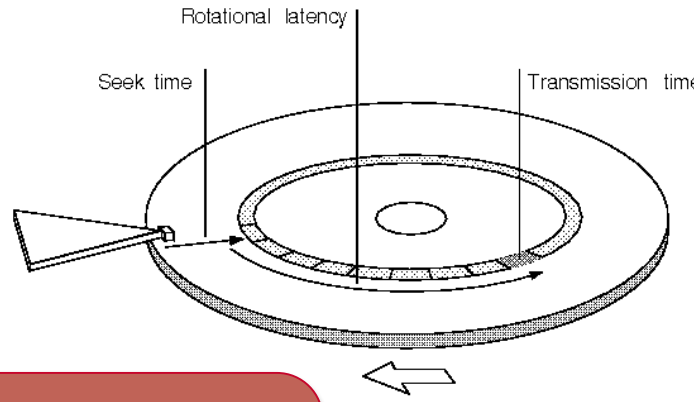

When **1 Memstore** is **full:**

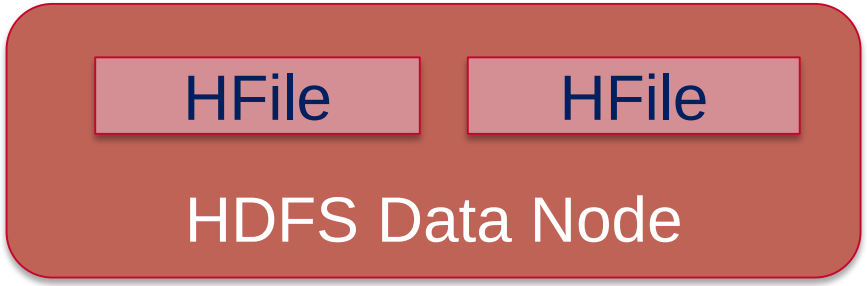
- **all memstores** in region **flushed to new Hfiles** on **disk**
- Hfile: sorted list of key → values On disk

# HBase HFile

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

**Sequential write**




**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**

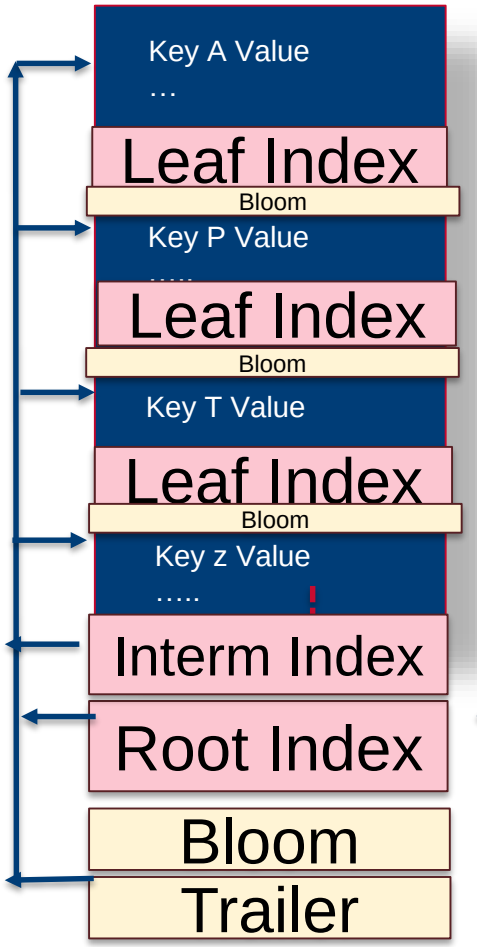
| Key | CF2:Col | version | value |
|---|---|---|---|
| ra | cf2:ca | v1 | 2 |
| rc | cf2:ca | v2 | 7 |
| rc | cf2:ca | v1 | 6 |
| rc | cf2:cd | v1 | 8 |

# HBase HFile Structure

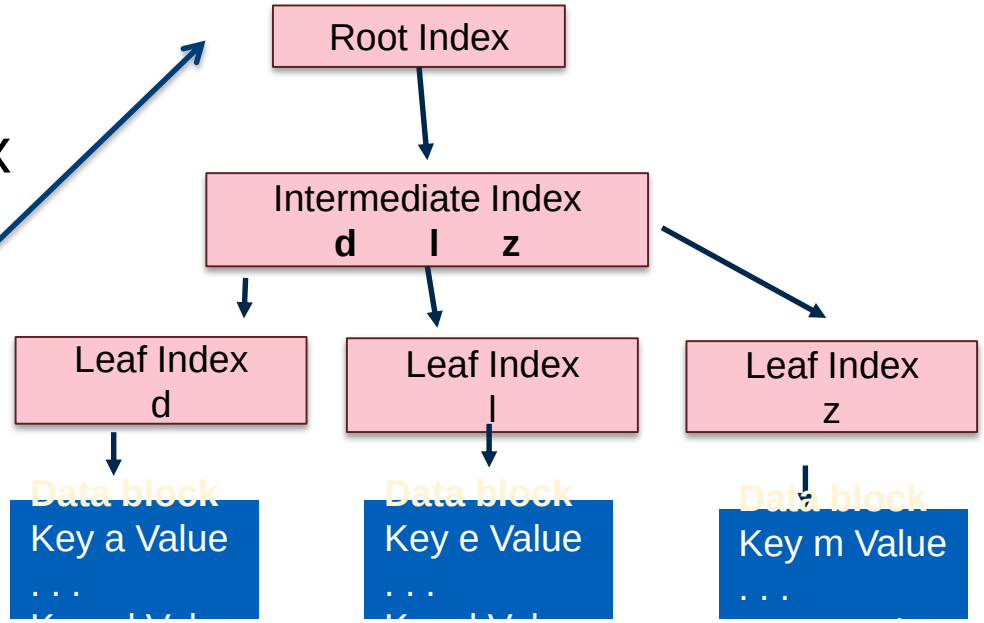
- Memstore flushes to an Hfile

Key-value pairs are stored in increasing order

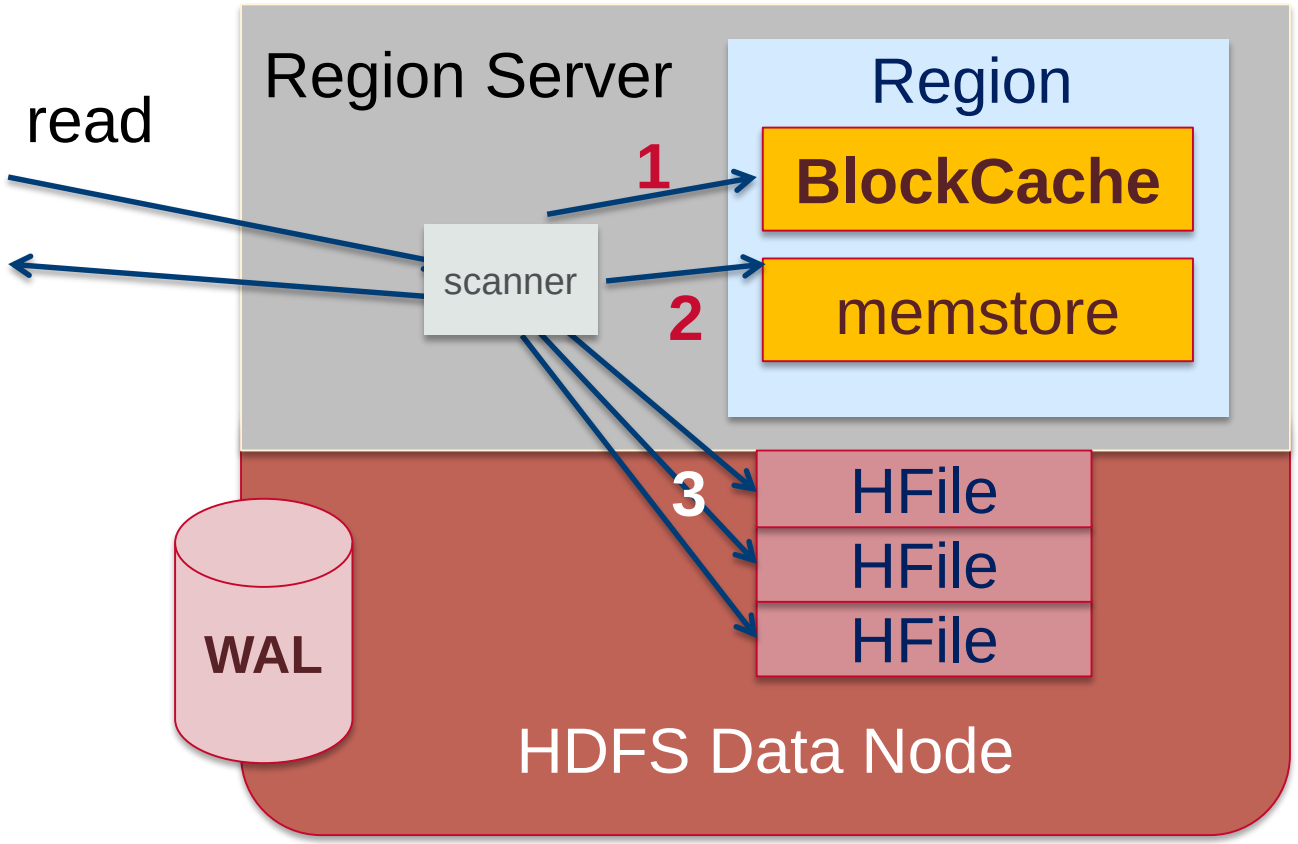
Index points to row **keys location**




B+-tree: leaf index , root index



# HBase Read Merge from Memory and Files

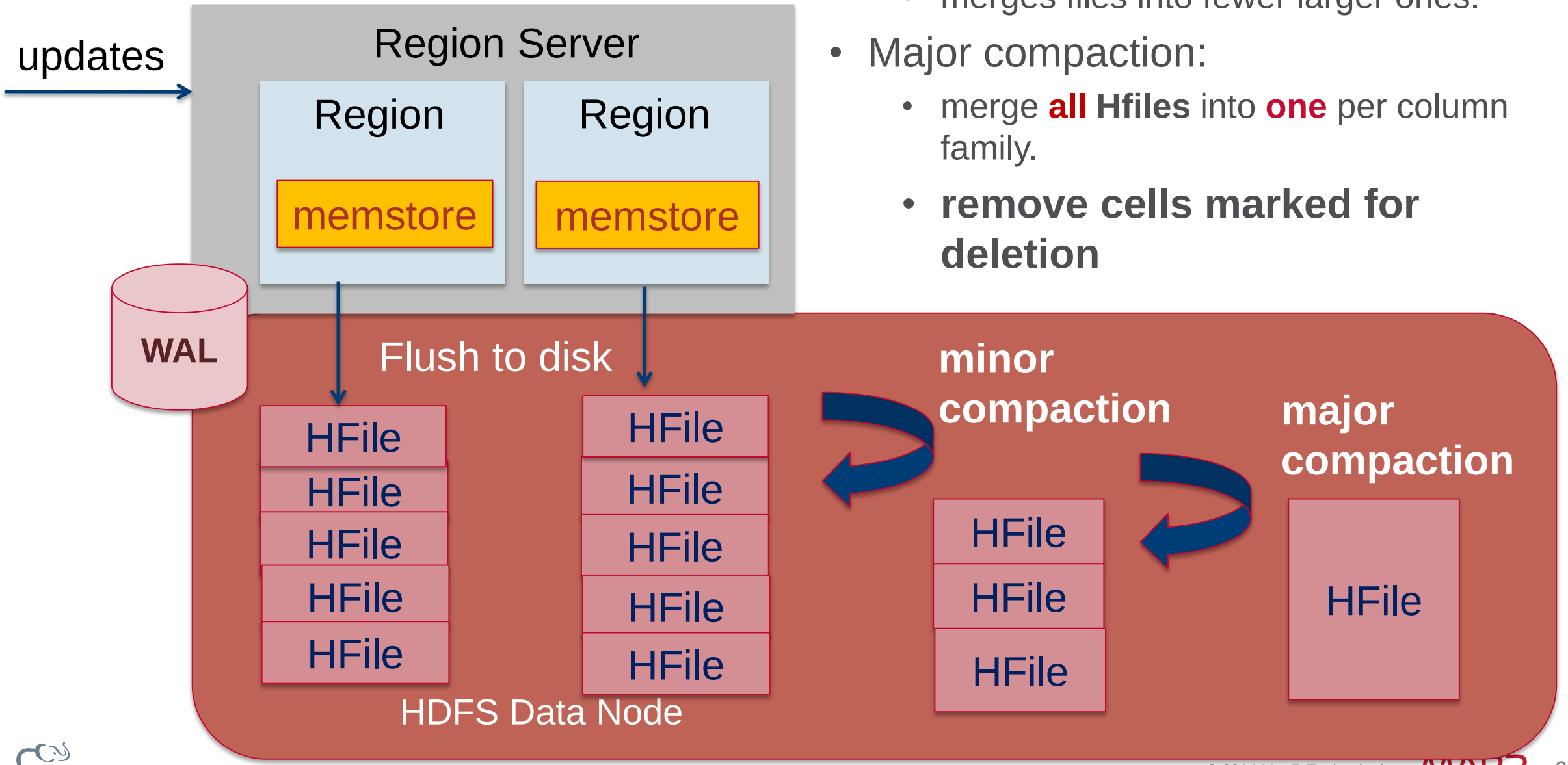



Get or Scan searches for Row Cell KeyValues:

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

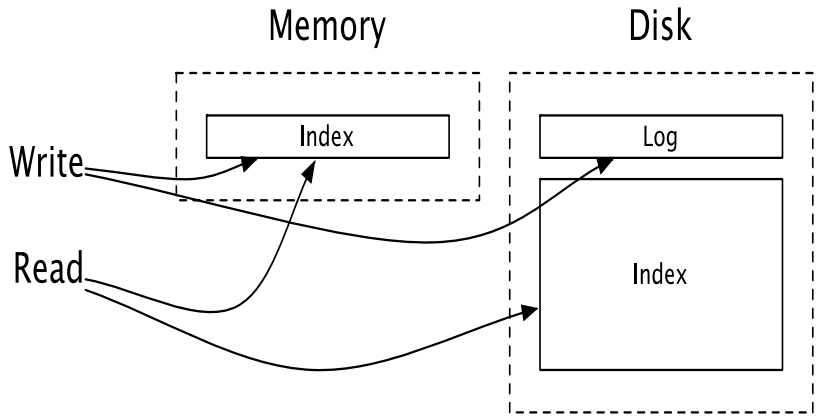



- **minor compaction:**
  - merges files into fewer larger ones.
- Major compaction:
  - merge **all Hfiles** into **one** per column family.
  - **remove cells marked for deletion**

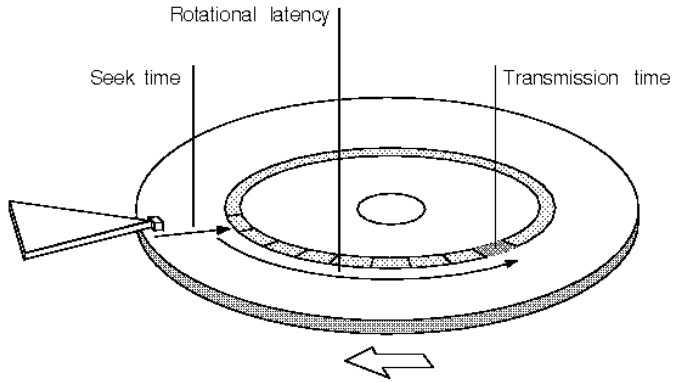
# HBase Background: Log-Structured Merge Trees

- 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**



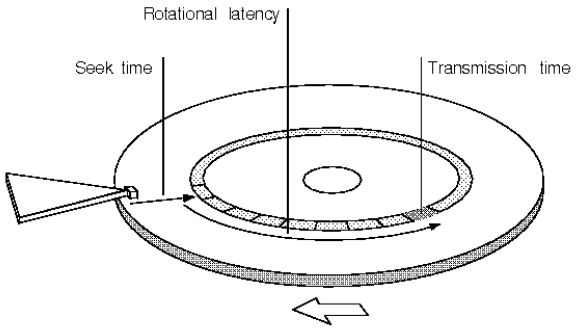

**predictable disk seeks**



# 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 →

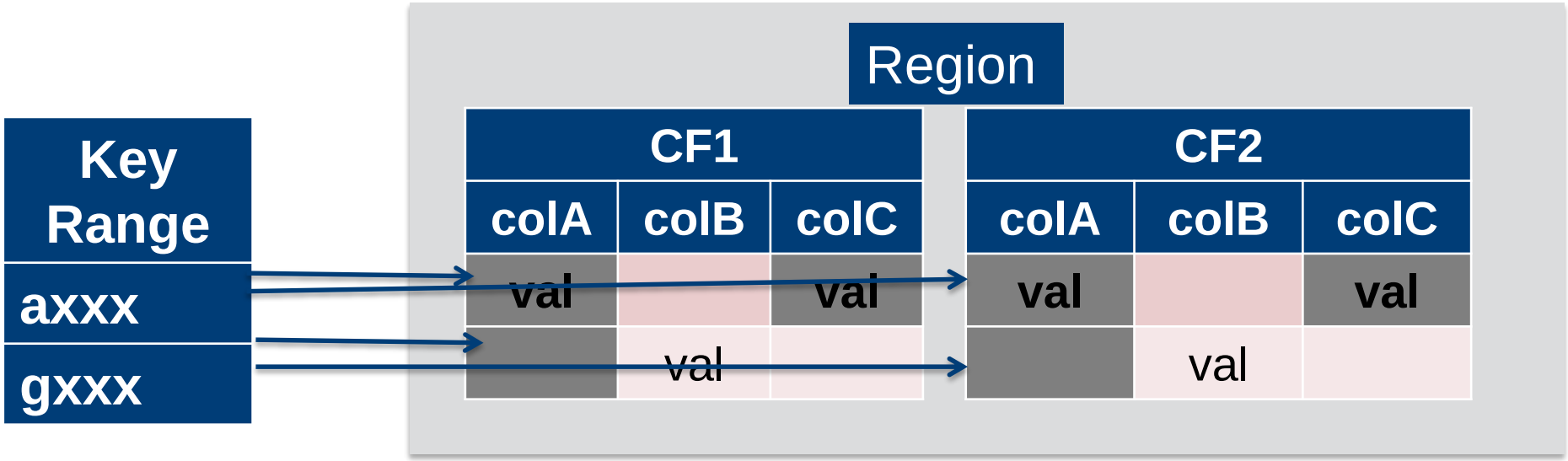
| Region1 Key Range |
|---|
| ra |
| rx |

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

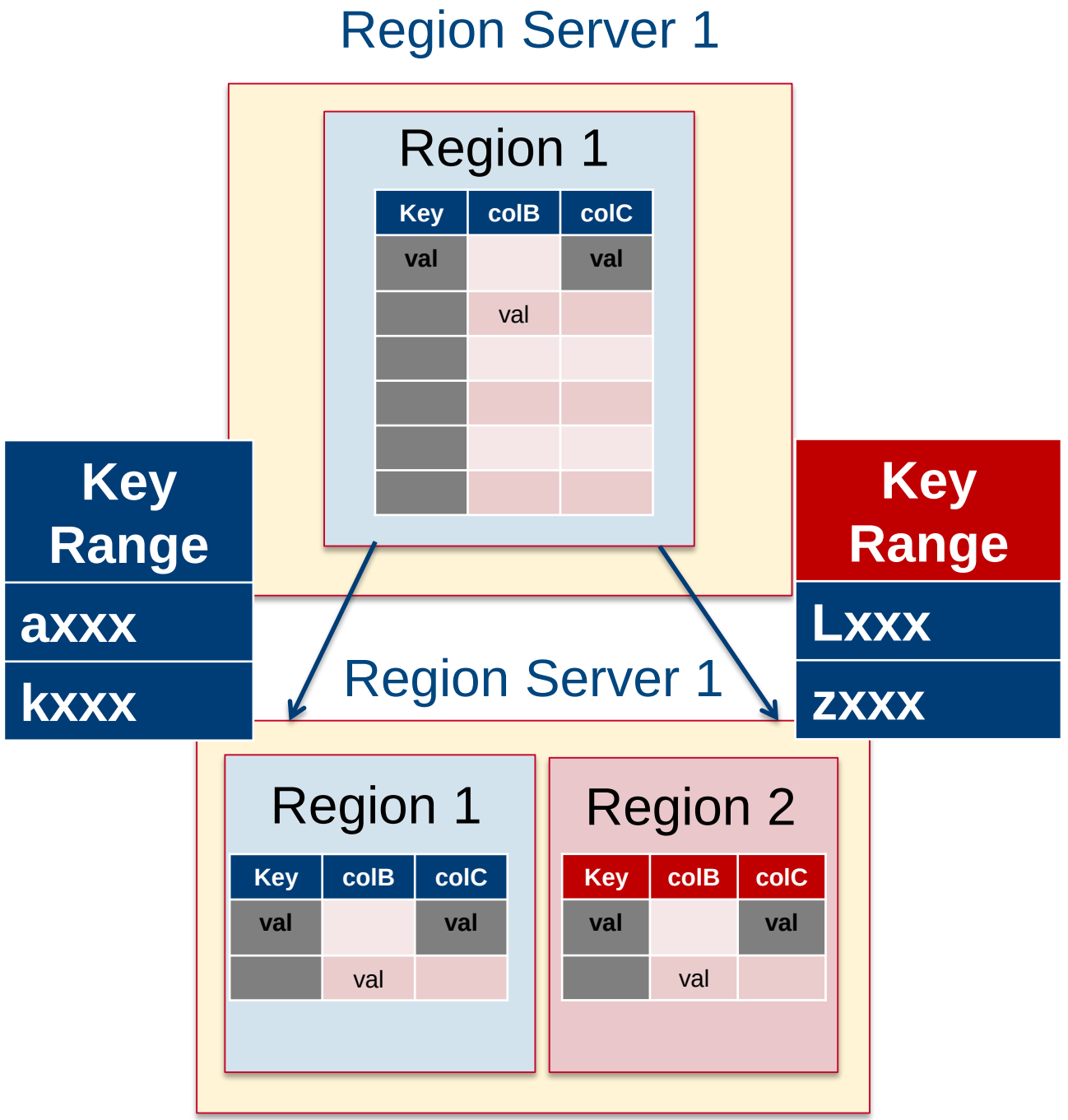
# Region = contiguous keys




- **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



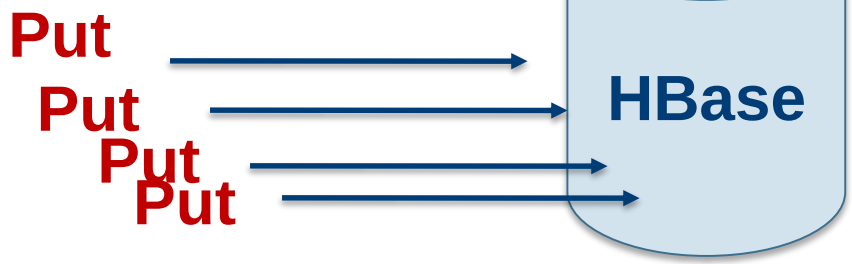
# HBase Use Cases

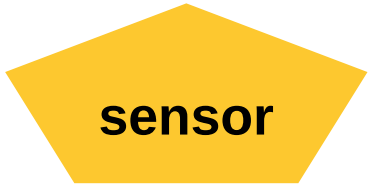
# 3 Main Use Case Categories

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

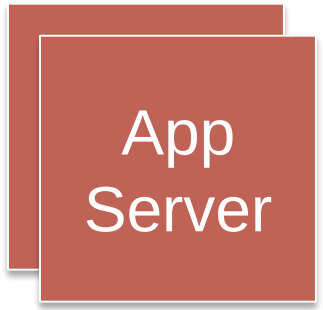
OpenTSDB

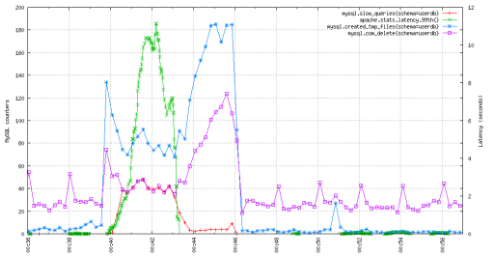
**Event time stamped data**





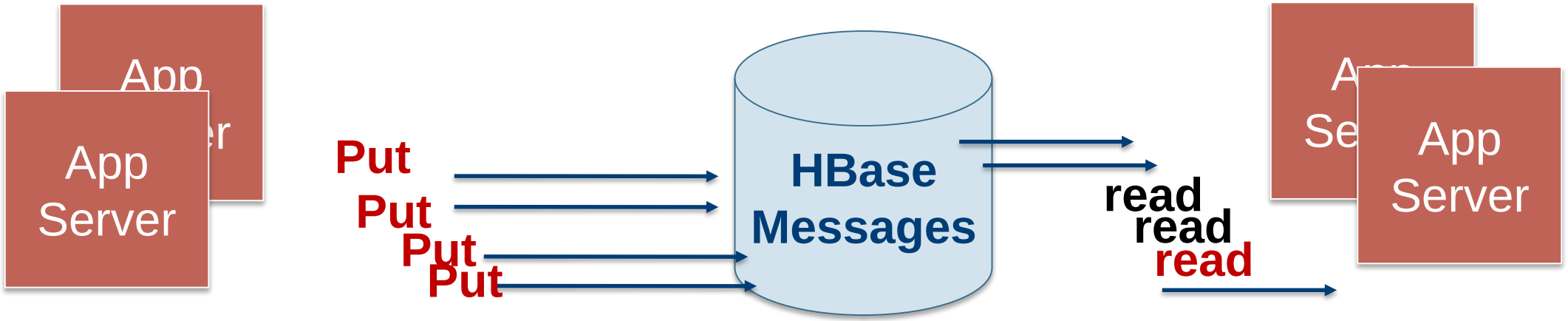


**Data for real-time monitoring.**

# 3 Main Use Case Categories

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

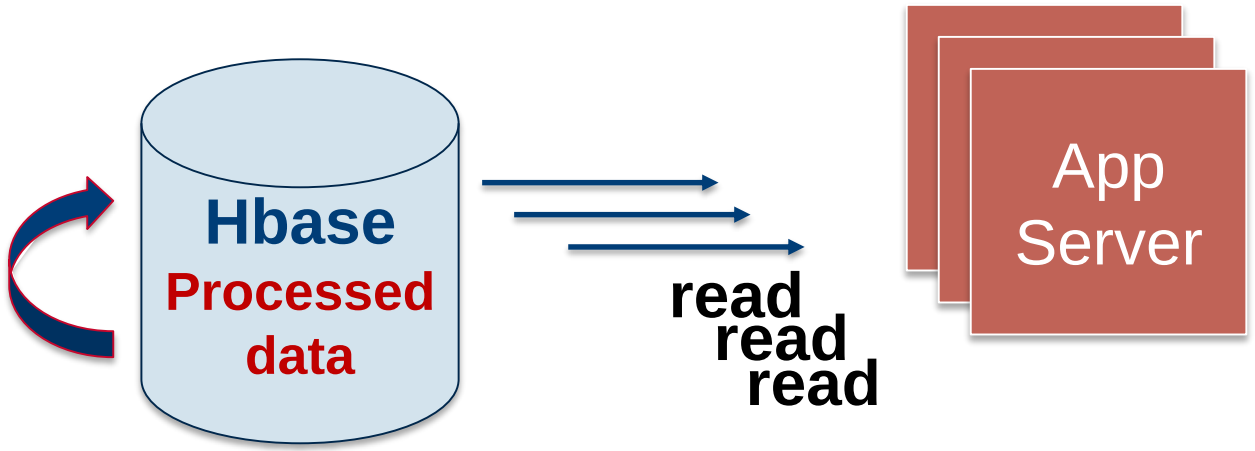



https://www.facebook.com/UsingHbase

# 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
- Demo/Lab using HBase Shell
  - Create tables and CRUD operations using MapR Sandbox
- Design considerations when migrating from RDBMS to HBase
- HBase Java API to perform CRUD operations
  - Demo / Lab using Eclipse, HBase Java API & MapR Sandbox
- How to work around transactions

# Prerequisite for Hands-On-Labs
# Install MapR Sandbox

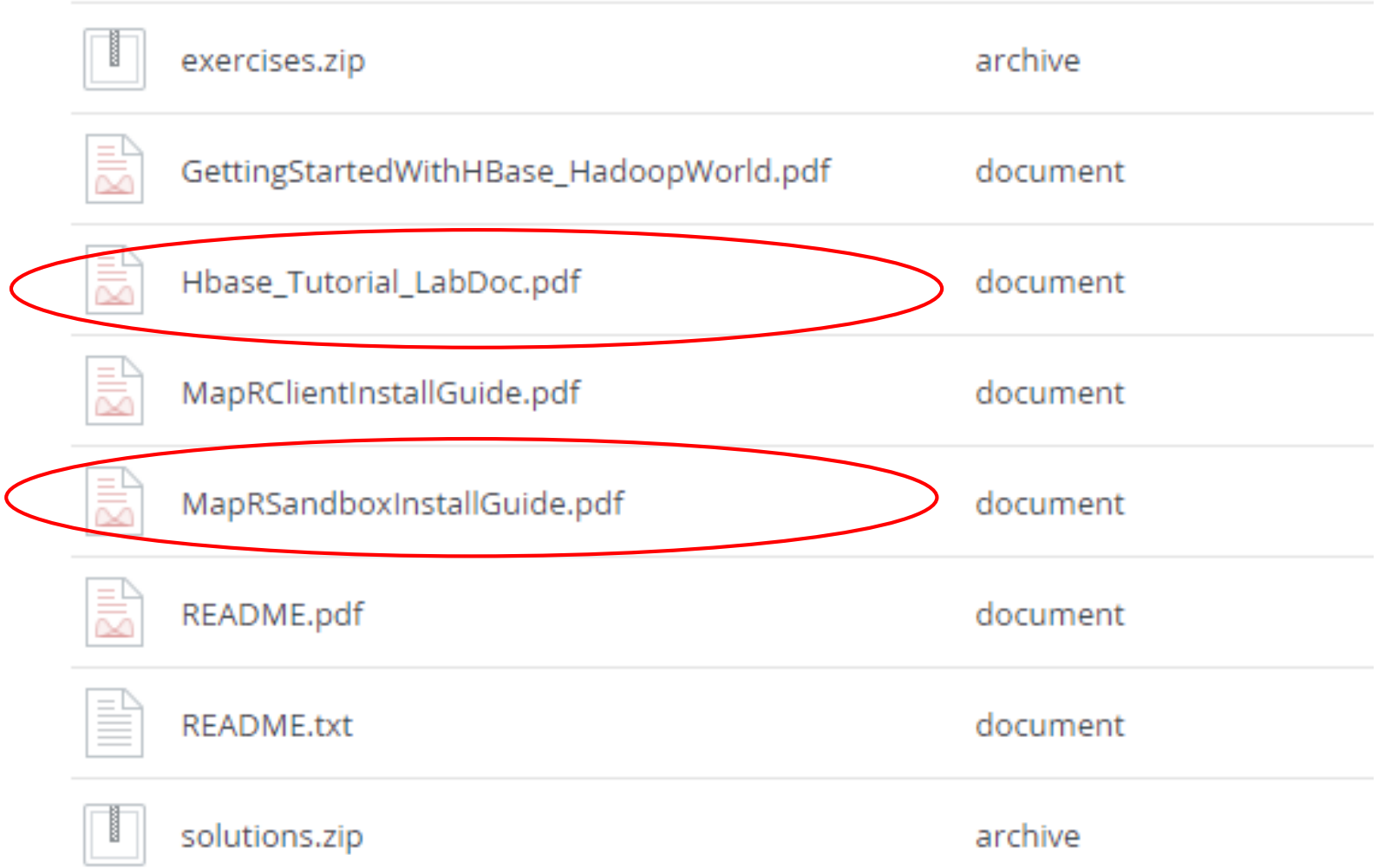
- Install a one-node MapR Sandbox on your laptop
- Install and configure Eclipse to develop HBase applications using Java API
- MapR Client is optional

**http://tinyurl.com/hbase-Oct-15-2014**

MapRSandboxInstallGuide.pdf

Hbase_Tutorial_LabDoc.pdf

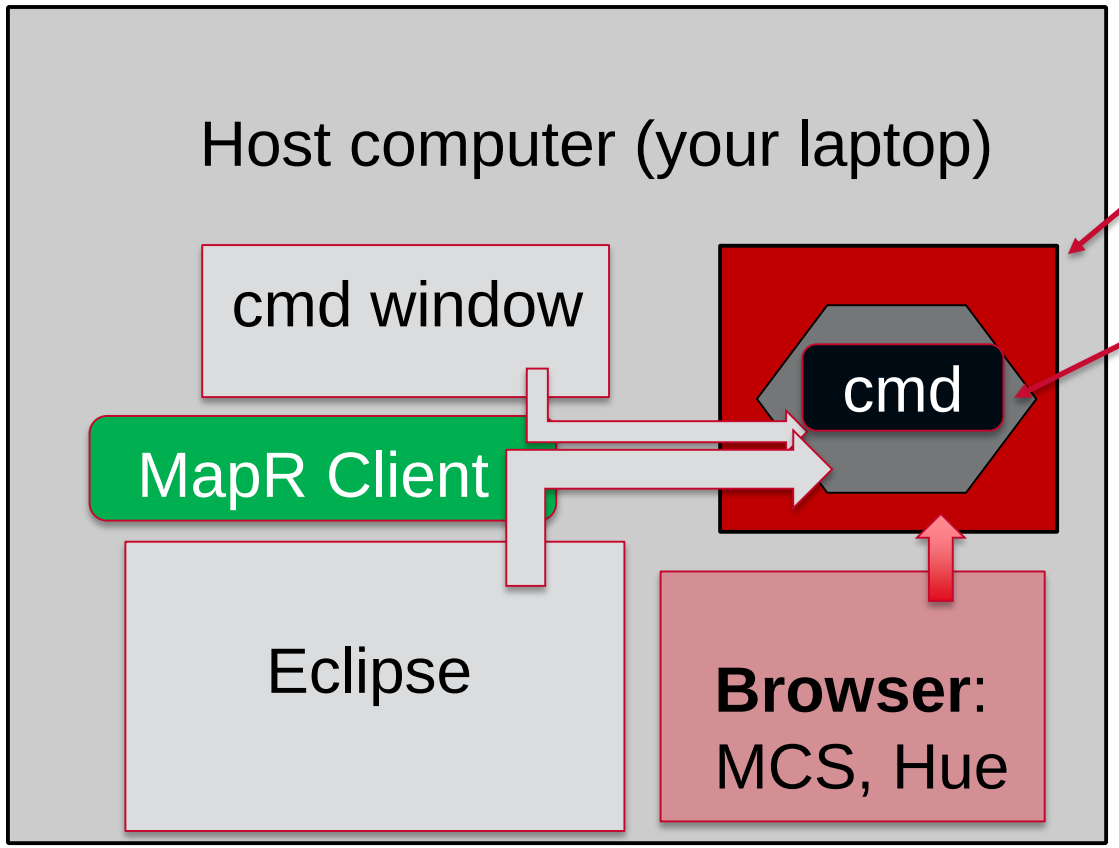
# http://tinyurl.com/hbase-Oct-15-2014



| | |
|---|---|
| exercises.zip | archive |
| GettingStartedWithHBase_HadoopWorld.pdf | document |
| Hbase_Tutorial_LabDoc.pdf | document |
| MapRClientInstallGuide.pdf | document |
| MapRSandboxInstallGuide.pdf | document |
| README.pdf | document |
| README.txt | document |
| solutions.zip | archive |

# 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

# Software components to install

- Here are all the software components you need to install to get MapR Sandbox working on Windows
  1. JDK 7
  2. MVMware player (or) Oracle VirtualBox (prerequisite)
  3. MapR Sandbox
  4. MapR Client (optional)
  5. Eclipse (for Java developers only)
- MapRSandboxInstallGuide.pdf will cover how to download, install and configure all these components on your laptop

If you have not already done it, please install it **now!!**

# Cluster for those that have not installed MapR Sandbox

- CLDB Nodes:

  https://ec2-54-176-71-123.us-west-1.compute.amazonaws.com:8443
  https://ec2-54-176-33-167.us-west-1.compute.amazonaws.com:8443

- Login nodes:

| | | |
|---|---|---|
| Rows 1 & 2: | 54.176.71.123 | ip-10-197-54-142 |
| Rows 3 & 4: | 54.176.33.167 | ip-10-199-46-212 |
| Rows 5 & 6: | 54.193.226.196 | ip-10-198-76-134 |
| Rows 7 & 8: | 54.177.2.5 | ip-10-197-8-213 |
| Last Rows: | 54.193.140.172 | ip-10-198-79-15 |

Username/passwd: user02 … user50 **mapr**

# 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'***

# Schema Design Guidelines

- HBase tables ≠ Relational tables!
  - HBase Design for Access Paterns

# Use Case Example: Record Stock Trade Information in a Table


- Trade data:

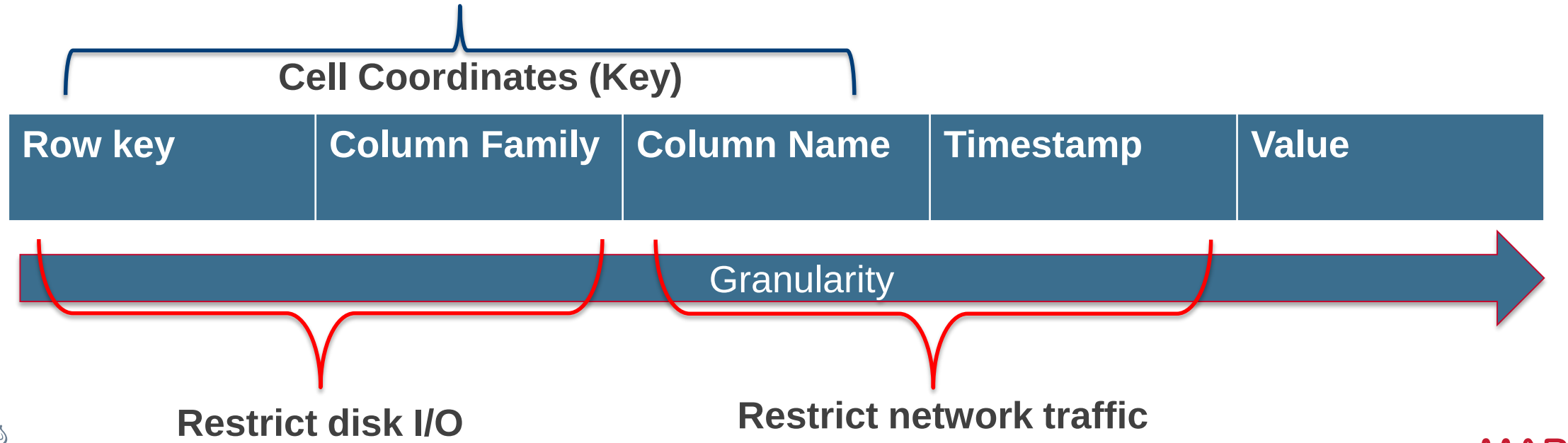
**Trade**

- timestamp
- stock symbol
- price per share
- volume of trade

- Example
  - 1381396363000 (epoch timestamp with millisecond granularity)
  - AMZN
  - $304.66
  - 1333 shares

# Intelligent keys

- Only the row keys are indexed
- **Compose the key** with attributes used for **searching**
  - Composite key : 2 or more **identifying** attributes
  - Like **multi-column index design** in RDB



# Composite Keys

## Use composite rowkey: attributes used for searching

- Include **multiple elements** in the rowkey
  - Use a separator or fixed length
- **Example** rowkey format:

| SYMBOL | + | DATE (YYYYMMDD) |
|---|---|---|

  - Ex: GOOG_20131012
- `Get` operations require complete row key.
- **Scans** can use **partial** keys.
  - Ex: “GOOG” or "GOOG_2014"

# Consider Access Patterns for Application

**How will data be retrieved?**

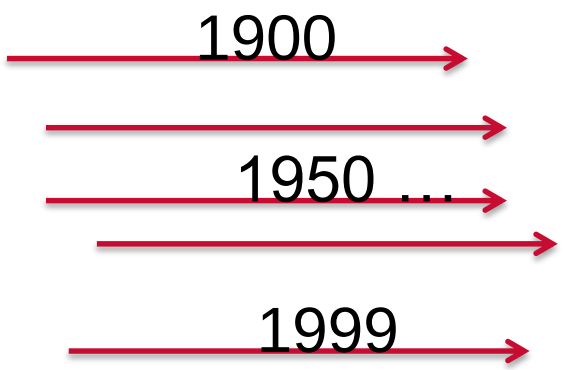
- By date? By hour? By companyId?
  - ***Rowkey design***
- **What if the Date/Timestamp is leftmost ?**

| Key | | | | |
|---|---|---|---|---|
| 1391813876369_AMZN | | | | |
| 1391813876370_AMZN | | | | |
| 1391813876371_GOOG | | | | |

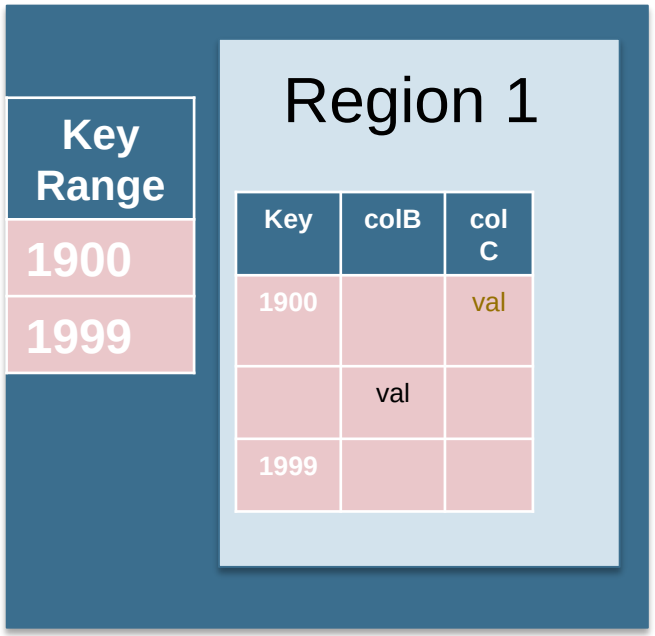
# Hot-Spotting and Region Splits

- If rowkeys are ***written*** in sequential order then writes go to only one server
  - Split when full

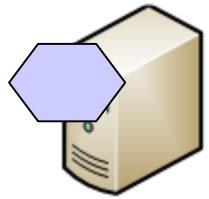
Sequential key, like a timestamp




Region Server 1

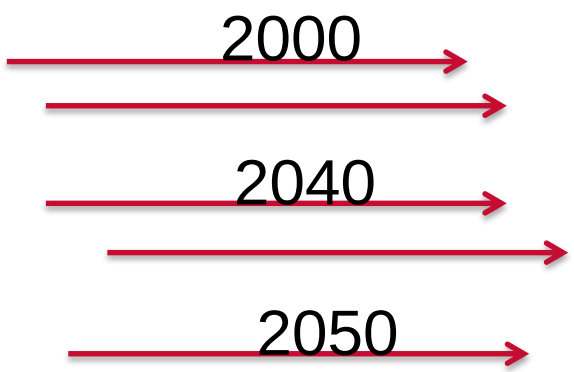


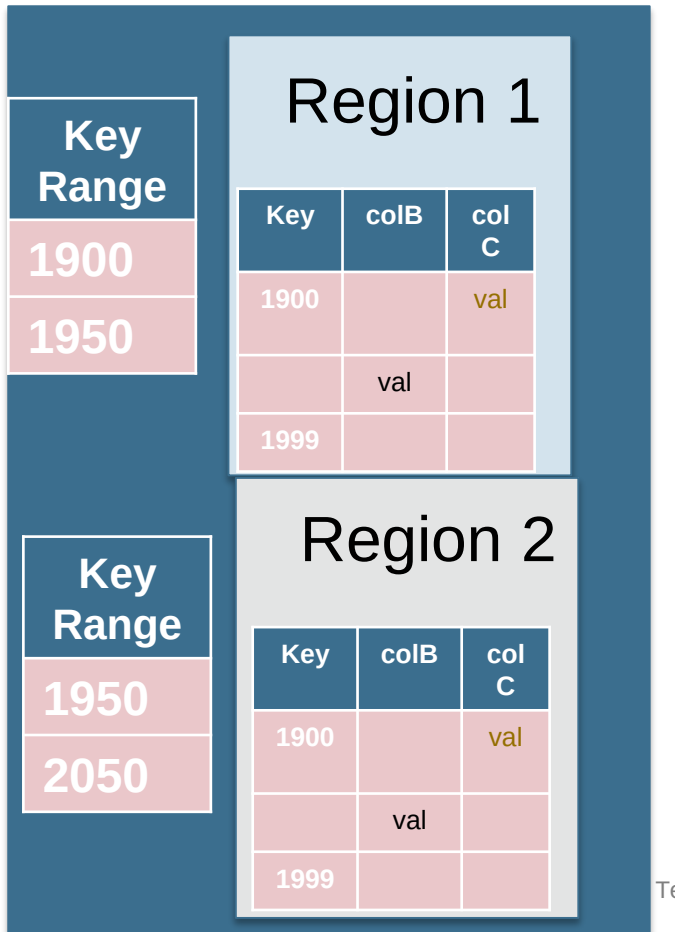

File Server 1

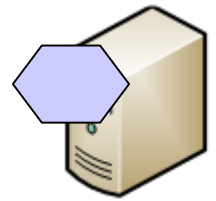


# Hot-Spotting and Region Splits

- Regions split as the table grows.
  - RegionServer Creates two new regions, each with half of the original regions keys.
- Sequential writes will go to new region

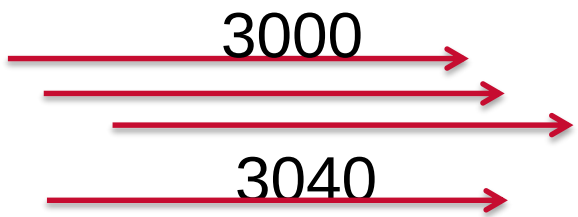


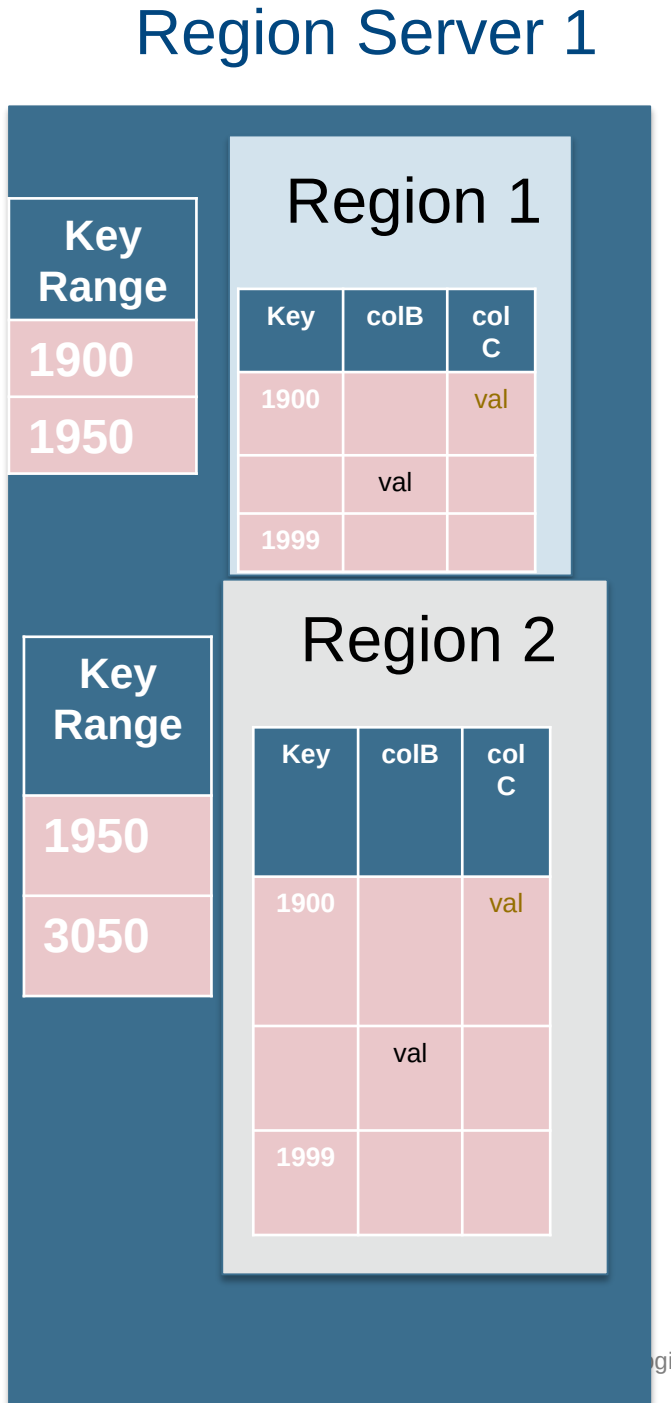



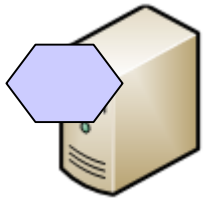
# Hot-Spotting and Region Splits

Sequential writes will go to new region



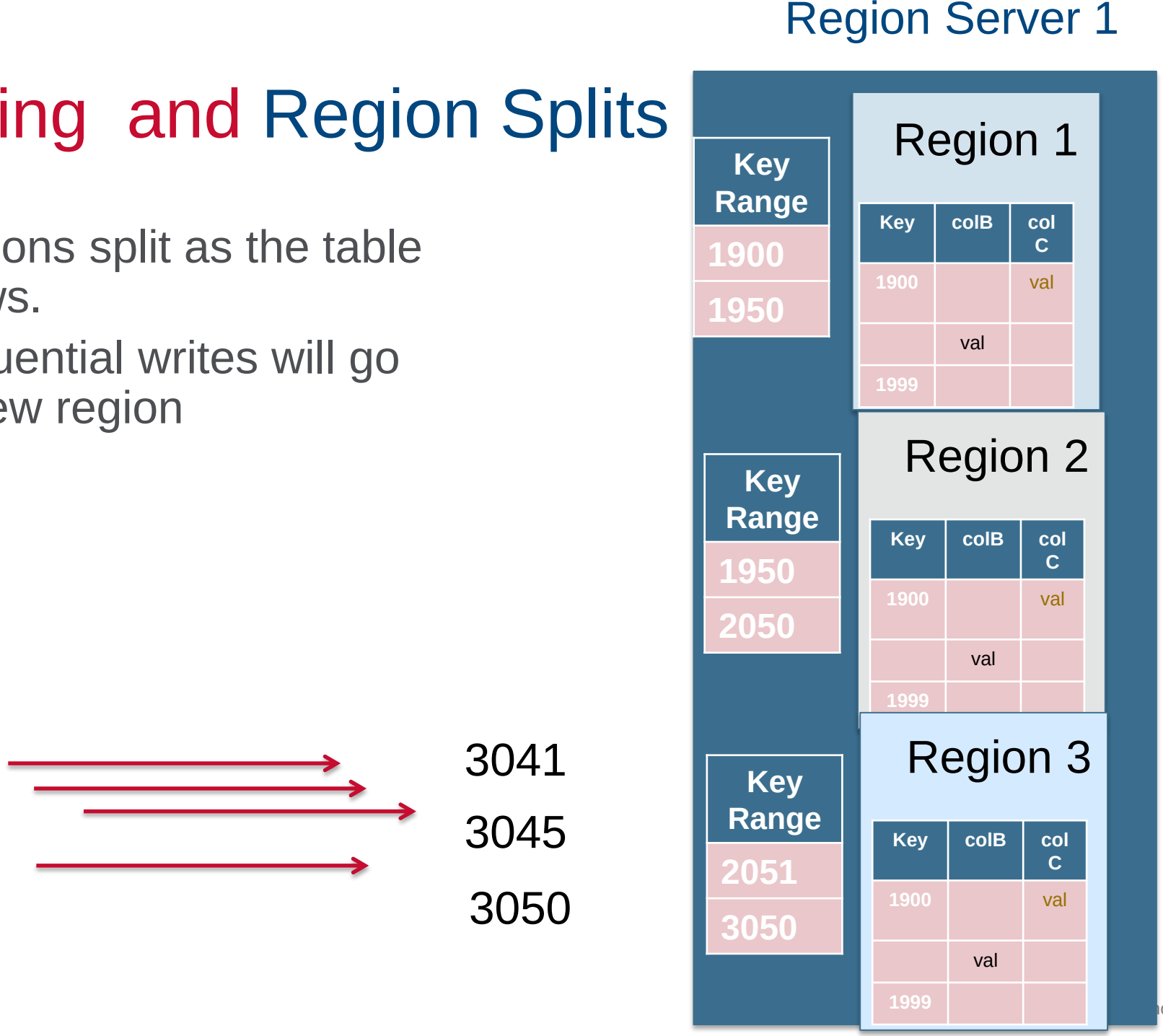

File Server 1



# Hot-Spotting and Region Splits

Regions split as the table grows.

Sequential writes will go to new region

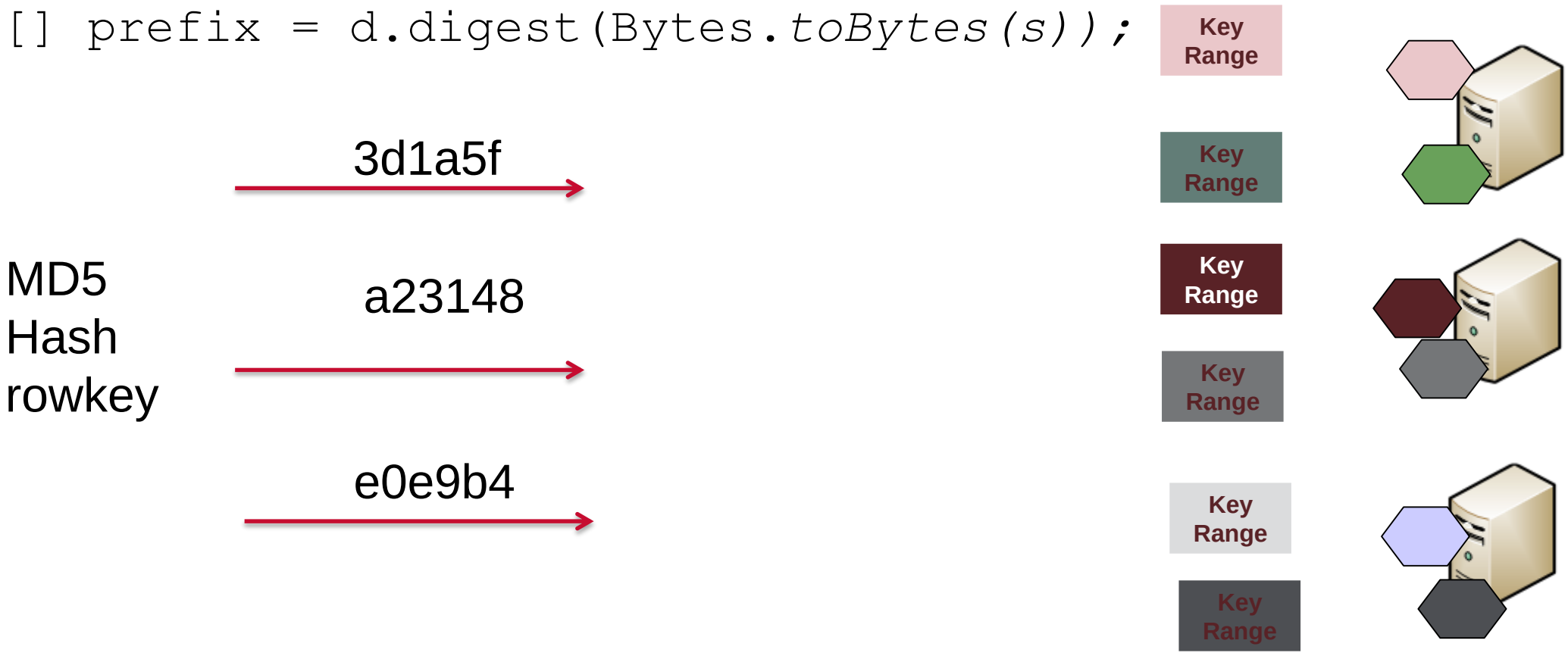


# Random keys

Random writes will go to different regions

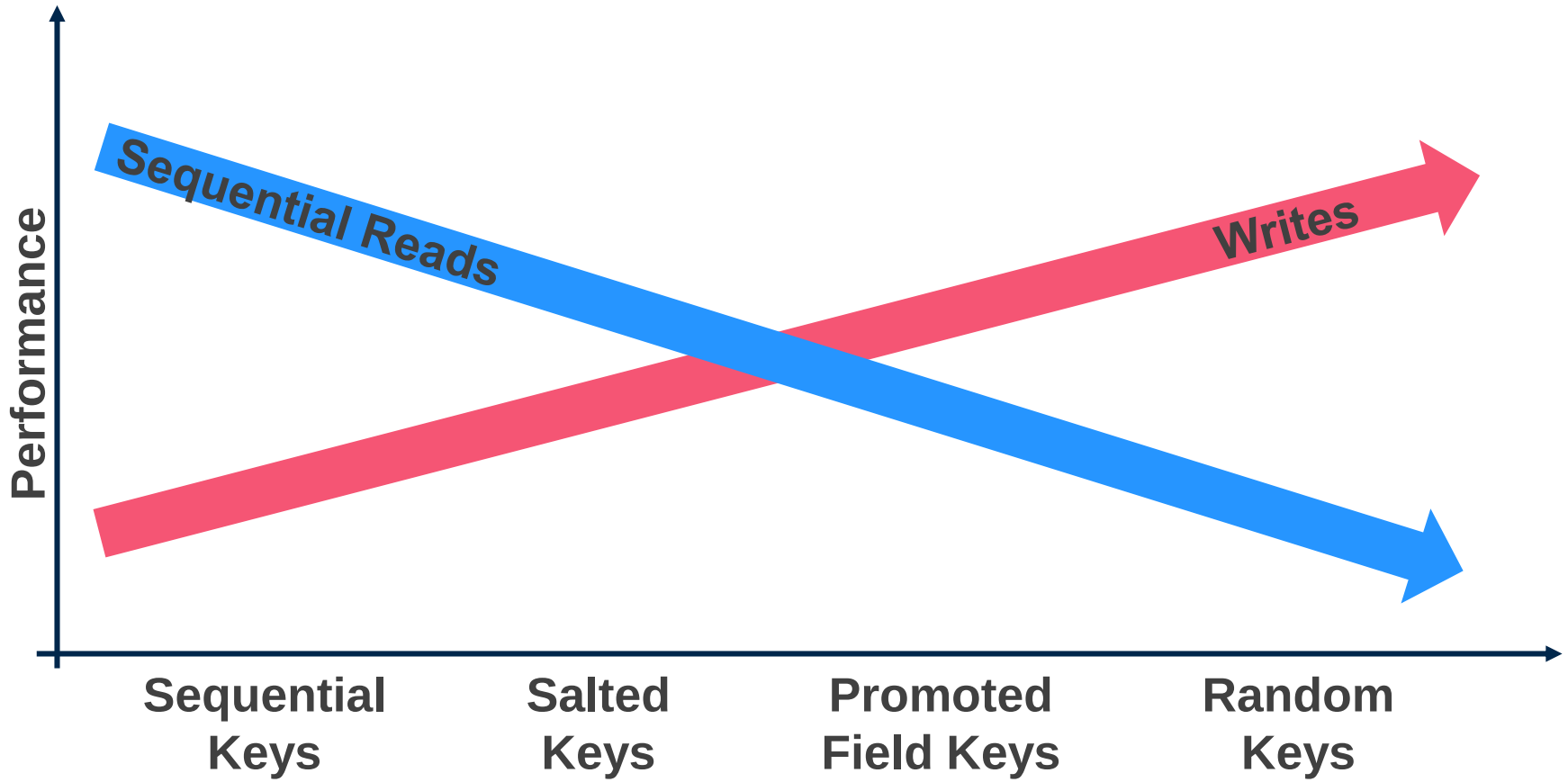
If table was pre-split or **big enough** to have split

```
d = MessageDigest.getInstance("MD5");
byte[] prefix = d.digest(Bytes.toBytes(s));
```



# Sequential vs. Random keys

**Random** is better for **writing** , but **sequential** is better for **scanning** row keys



# Prefix, Promote a field key

amzn_1999

amzn_2003

amzn_2005

cisc_1998

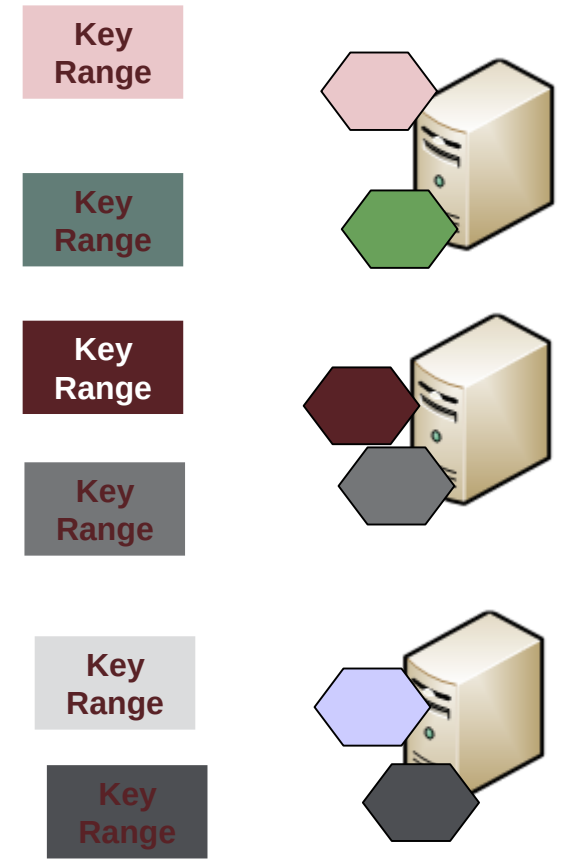
cisc_2002

cisc_2010

goog_1990

goog_2020

goog_2030



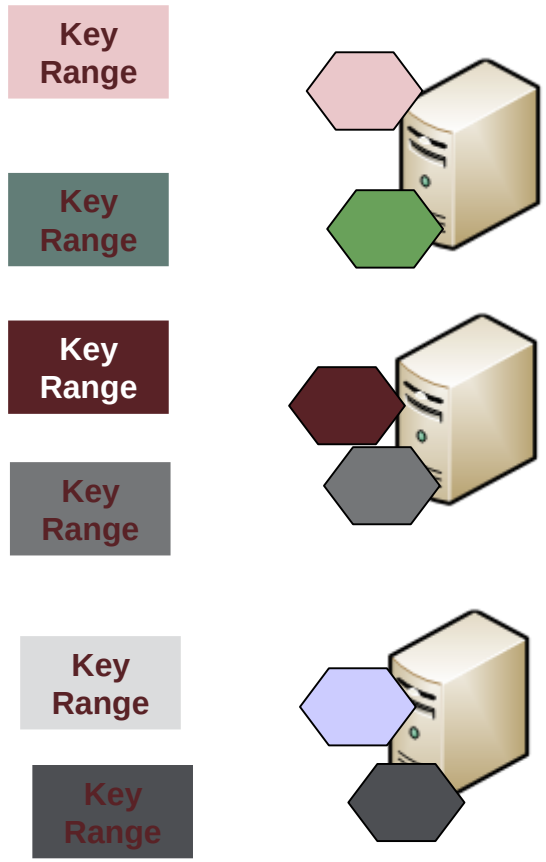
# Prefix with a Hashed field key

prefix the rowkey with a (shortened) hash:

```
byte[] hash = d.digest(Bytes.toBytes(fieldkey));
Bytes.putBytes(rowkey, 0, hash, 0, length);
```

MD5 Hash **prefix** rowkey

1d1a5f_1999
3d1a5f_2007
a23148_2003
b33148_2006
e0e9b4_2000
g0e8b4_2004



# Consider Access Patterns for Application

## How will data be retrieved?

- Which trade data needs fastest access (or most frequent)?
  - ***Rowkey ordering***
- What if you want to retrieve the stocks by symbol and date?
  - **Scan by row key prefix Increasing time: PREFIX_TIMESTAMP**

| SYMBOL | + | timestamp |
|---|---|---|

| Key | | | | |
|---|---|---|---|---|
| AMZN_1391813876369 | | | | |
| AMZN_1391813876370 | | | | |
| GOOG_1391813876371 | | | | |

- **What if you usually want to retrieve the most recent?**

# Last In First Out Access: Use Reverse-Timestamp

- Row keys are sorted in increasing order
- For fast access to **most-recent** writes:
  - **design composite** rowkey with **reverse-timestamp** that **decreases** over time.
  - **Scan by row key prefix Decreasing: [MAXTIME–TIMESTAMP]**
    - **Ex:** `Long.MAX_VALUE-date.getTime()`

| SYMBOL | + | Reverse timestamp |
|---|---|---|

| Key | | | | |
|---|---|---|---|---|
| AMZN_98618600666 | | | | |
| AMZN_98618600777 | | | | |
| GOOG_98618608888 | | | | |

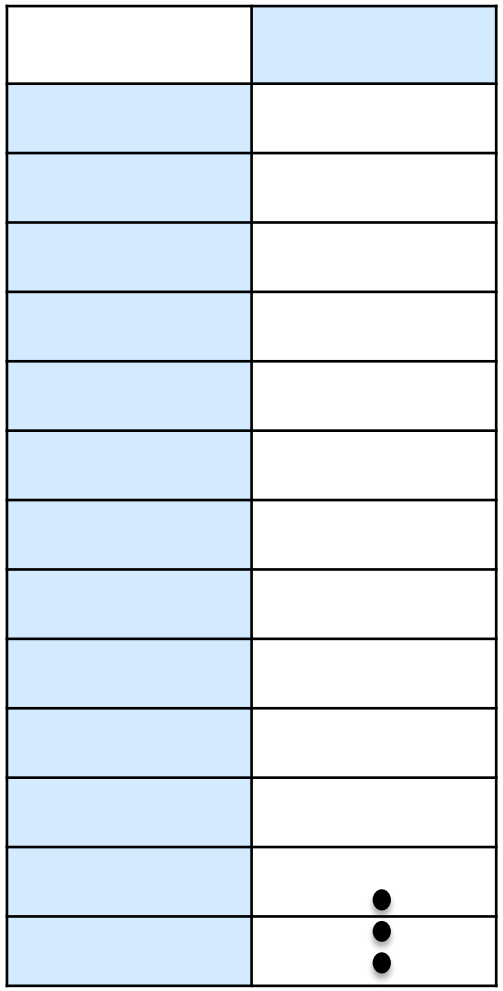
# Consider Access Patterns for  Application

**How will data be retrieved?**

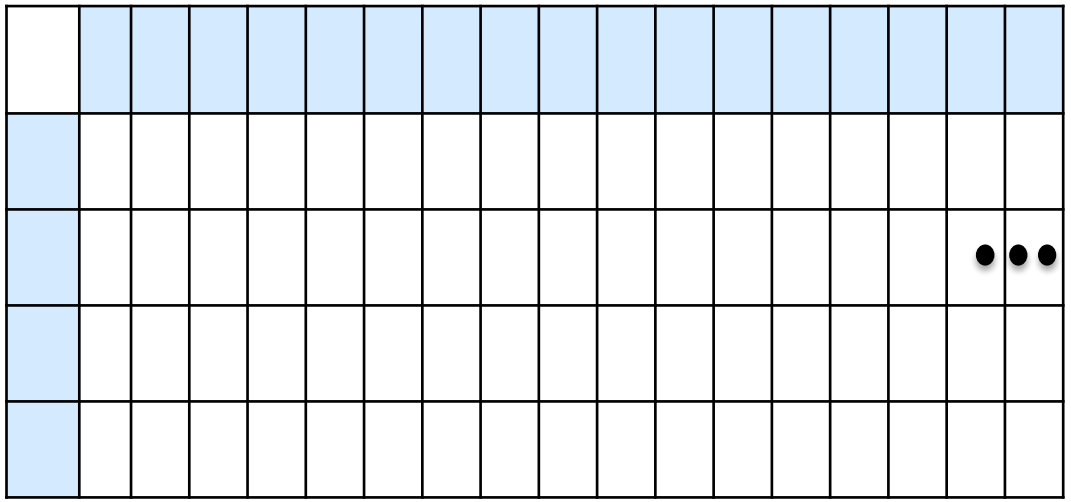
- What are the needs for atomicity of transactions?
  - ***Column design***
  - **More** Values  in a **single row**
    - Works well to get or **update multiple values**

# Rowkey design influences shape of Tables: Tall or Flat

Tall Narrow



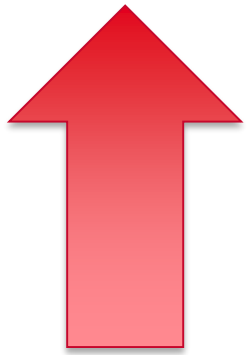
Flat Wide



# Tall Table for Stock Trades

Rowkey format: **SYMBOL** | **+** | **Reverse timestamp**

Ex: AMZN_98618600888



| rowkey | CF: CF1 | |
|---|---|---|
| | CF1:price | CF1:vol |
| … | … | … |
| AMZN_98618600666 | 12.34 | 2000 |
| AMZN_98618600777 | 12.41 | 50 |
| AMZN_98618600888 | 12.37 | 10000 |
| … | … | … |
| CSCO_98618600777 | 23.01 | 1000 |
| … | | |

# Consider Access Patterns for Application

**How will data be retrieved?**

- Are Price and Volume data typically accessed together, or are they unrelated?
  - ***Column family structure***
- Column Families
  - group data that will be read and stored together
  - Can set attributes:
    - # Min/Max versions, compression, in-memory, Time-To-Live
- Columns
  - Column names are dynamic, not pre-defined
  - every row does not need to have same columns

# Wide Table for Stock Trades

Rowkey format: **SYMBOL** + **Reverse timestamp rounded to the hour**

Ex: AMZN_20131020

- Separate price & volume data into column families
- Segregate time into buckets:
  - Time rounded to the **hour** in the rowkey
  - Time in column name represents **seconds since the timestamp** in the key
  - One **row** stores a **bucket** of measurements for the hour

| rowkey | CF price | | | | CF vol | | | |
|---|---|---|---|---|---|---|---|---|
| | p:10 | p:1000 | ... | p:2000 | v:10 | v:1000 | ... | v:2000 |
| AMZN_986186006 | 12.37 | 13 | | 12.34 | 10000 | | | 2000 |
| … | | | | | | | | |
| CSCO_986186070 | | | | | 1000 | | | |

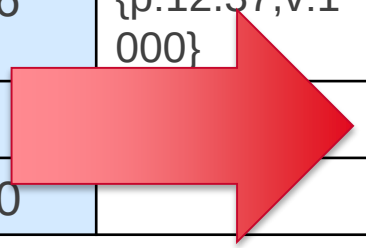
# Wide Table for Stock Trades

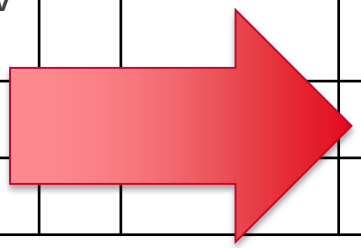
Rowkey format: **SYMBOL** + **Reverse timestamp rounded to the hour**

Ex: AMZN_20131020

- Segregate time into buckets:
  - Time rounded to the **hour** in the rowkey
  - Time in column name represents **seconds since the timestamp** in the key
  - One **row** stores a **bucket** of measurements for the hour

| rowkey | CF1 | | | CF stats | | |
|---|---|---|---|---|---|---|
| | CF1:10 | CF1:1000 | … | Day Hi | … | Day LO |
| AMZN_986186006 | {p:12.37,v:1000} | {p:12.37,v:1000} | | | | |
| … | | | | | | |
| CSCO_986186070 | | | | | | |





Column names can be dynamic, every row does not need to have same columns

# Lesson: Schemas can be very flexible and can even change on the fly

# Consider Access Patterns for  Application

**How will data be retrieved?**

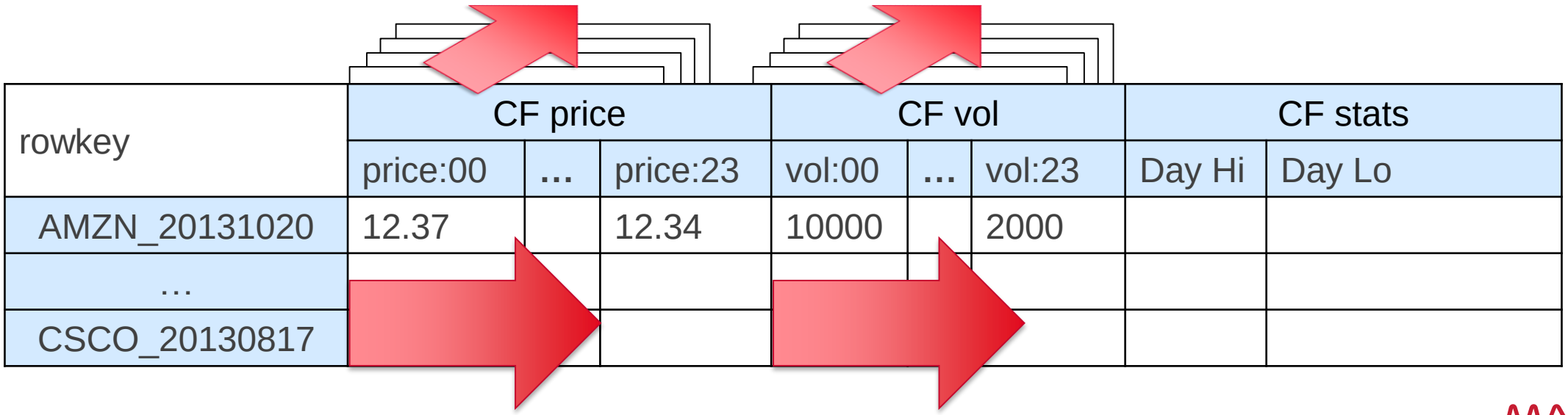
- Do all trades need to be saved forever?
  - ***TTL  Time to Live ,  CF can be set to expire cells***
- How many Versions?
  - ***Max Versions***
  - You can have many versions of data in a cell, default is 3

# Wide Table for Stock Trades

Rowkey format: **SYMBOL** + **date YYYYMMDD**

Ex: AMZN_20131020

- Separate price & volume data into column families
- Segregate time into buckets:
  - Date in the rowkey
  - Hour in the column name
  - Set Column Family to store Max Versions, timestamp in the version



| rowkey | CF price | | | CF vol | | | CF stats | |
|---|---|---|---|---|---|---|---|---|
| | price:00 | ... | price:23 | vol:00 | ... | vol:23 | Day Hi | Day Lo |
| AMZN_20131020 | 12.37 | | 12.34 | 10000 | | 2000 | | |
| ... | | | | | | | | |
| CSCO_20130817 | | | | | | | | |

# Flat-Wide Vs. Tall-Narrow Tables

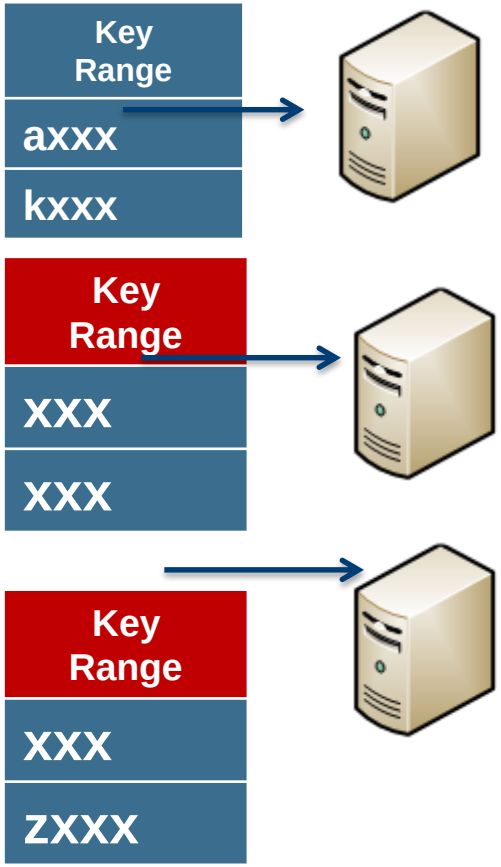
- Tall-Narrow provides better **query granularity**
  - **Finer grained** Row Key
  - Works well with **scan**

- Flat-Wide supports built-in row **atomicity**
  - **More** Values in a **single row**
    - Works well to **update multiple values** (row atomicity)
    - Works well to **get multiple associated values**

# Lesson: Have to know the queries to design in performance

# Comparing Relational schema to HBase

- HBase is **lower-level** than relational tables
  - Design is different
- Relational design
  - **Data** centric, focus on **entities** and **relations**
  - Query, joins
    - **New views** of data from different tables **easily created**
      - Does not scale across cluster
- HBase is designed for clustering:
  - Distributed data is stored and accessed together
  - **Query** centric, focus on how the data is **read**
  - Design for the **questions**



# Normalization


| ORDER_ID | ORDER_DATE | SHIP_STATUS | TOTAL |
|---|---|---|---|
| 123 | 2012-07-11 | SHIPPED | 39.45 |
| 124 | 2012-07-12 | BACKORDER | 29.37 |
| 125 | 2012-07-13 | SHIPPED | 42.47 |

| ORDER_ID | ITEM_ID | PRICE |
|---|---|---|
| 123 | 83924893 | 10.00 |
| 123 | 563344893 | 20.00 |
| 123 | 343978893 | 9.45 |
| 124 | 83924893 | 29.37 |
| 125 | 563344893 | 20.00 |
| 125 | 343978893 | 22.47 |

- Relational Databases are typically normalized
- Goal Normalization:
  - **eliminate redundant** data
  - Put **repeating** information in its **own** table

Normalized database :

- Causes joins
  - data has to be retrieved from more tables.
  - queries can take more time

# HBase Nested Entity

- A one-to-many relationship can be modeled as a single row
  - Embedded, Nested Entity
- **Order** one-to-many with **Line Items**
  - Row key: **parent id**
    - OrderId
  - column name : **child id** stored
    - line Item id

| OrderId | Data:date | item:**id1** | item:**id2** | item:**id3** |
|---|---|---|---|---|
| 123 | 20131010 | $10 | $20 | $9.45 |

# De-Normalization

Order and Items in **same table**

| OrderId | Data:date | item:**id1** | item:**id2** | item:**id3** |
|---|---|---|---|---|
| 123 | 20131010 | $10 | $20 | $9.45 |

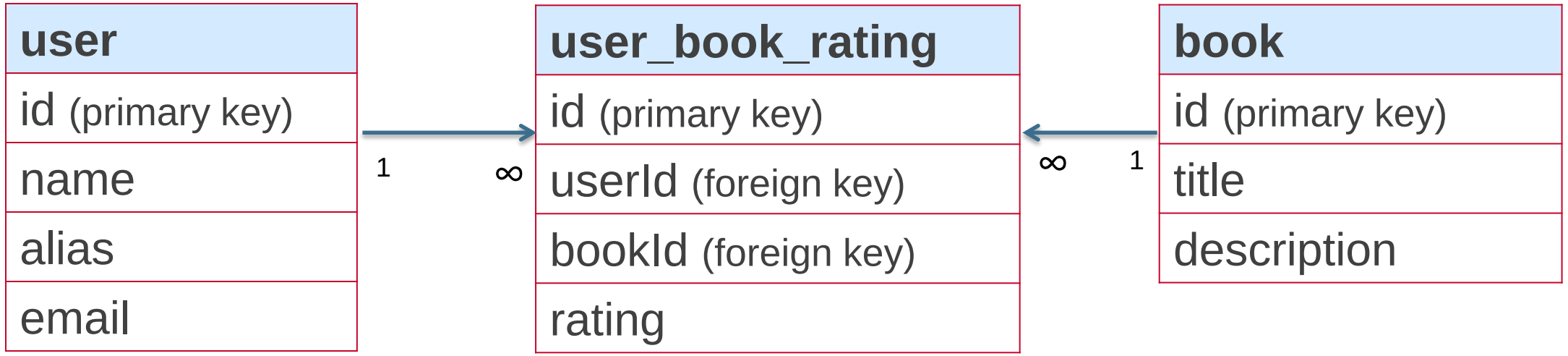
De-normalization:

- **store** data about an entity and related entities in the **same table**.

- Reads are faster across a cluster
  - **retrieve** data about entity and related entities in **one read**

# Many to Many Relationship RDBMS

Online book store




- Querys
- Get name for user x
- Get tiltle for book x
- Get books and  corresponding ratings for userId  x
- Get all userids and corresponding ratings for book y

# Many to Many Relationship HBase

User table  Column family rating, bookid is column name

| Key | **data**:fname | … | **rating:bookid1** | rating:bookid2 |
|---|---|---|---|---|
| **userid1** | | | 5 | 4 |

Book table  Column family rating, userid is column name

| Key | **data**:title | … | **rating:userid1** | rating:userid2 |
|---|---|---|---|---|
| **bookid1** | | | 5 | 4 |

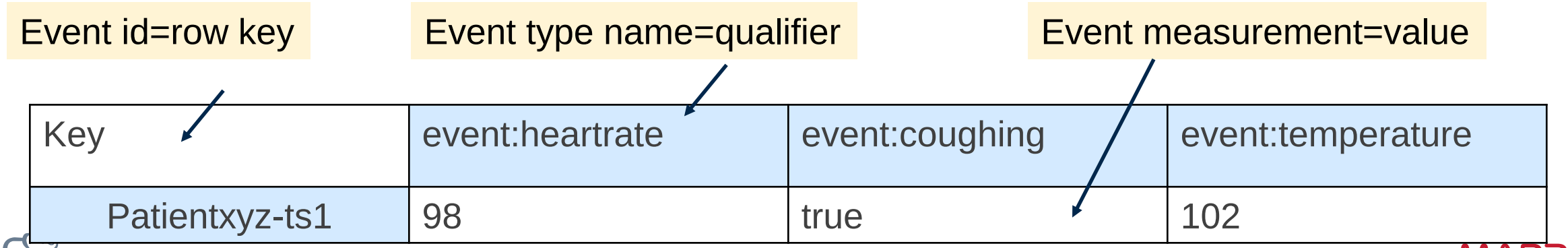
- Queries
- Get books and corresponding ratings for userId x
- Get all userids and corresponding ratings for book y

# Generic data: Event, Attributes, Values

- Event Id, Event name-value pairs , **schema-less**

```
patientXYZ-ts1, Temperature , "102"
patientXYZ-ts1, Coughing, "True"
patientXYY-ts2, Heart Rate, "98"
```

- This is the advantage of HBase
  - Define columns on the fly,
    - put **attribute name** in **column qualifier**



# Self Join Relationship  HBase

- **Example Twitter**
  - **User_x  follows  User_y**
  - **User_y followed by User_z**

- Querys
  - Get all users who Carol follows
  - Get all users following Carol

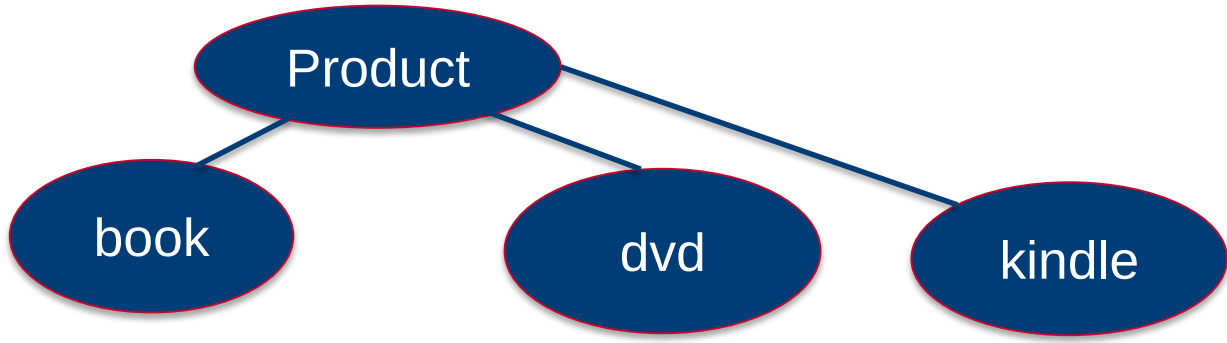
| Key | data:timestamp |
|---|---|
| **Carol:follows:SteveJobs** | |
| **Carol:followedby:BillyBob** | |

# Hierarchical Data

- tree like structure
- Use a flat-wide
  - Parents, children in columns


| Key | P:USA | P:TN | p:FL | c:TN | C:FL | C:Nashvl | C:Miami |
|---|---|---|---|---|---|---|---|
| USA | | | | state | state | | |
| TN | country | | | | | city | |
| FL | country | | | | | | city |
| Nashville | | state | | | | | |
| miami | | | state | | | | |

# Inheritance mapping




- Online Store Example  Product table
  - type is in row key for searching
  - Columns are not  the same for different types

| Key | price | title | details | model |
|---|---|---|---|---|
| Bok+id1 | 10 | Hbase | blah | |
| Dvd+Id2 | 15 | stones | blah | |
| Kin+Id3 | 100 | | blah | fire |

# Agenda

- Why do we need NoSQL / HBase?
- Overview of HBase & HBase data model
- HBase Architecture and data flow
- Demo/Lab using HBase Shell
  - Create tables and CRUD operations using MapR Sandbox
- Design considerations when migrating from RDBMS to HBase
- HBase Java API to perform CRUD operations
  - Demo / Lab using Eclipse, HBase Java API & MapR Sandbox
- How to work around transactions

# HBase
# Java API fundamentals to perform CRUD operations



MAPR®

# 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/139181387636 9/Put/vlen=8/ts=0,
Adam/items:notepads/139181387636 9/Put/vlen=8/ts=0,
Adam/items:pens/139181387636 9/Put/vlen=8/ts=0}

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

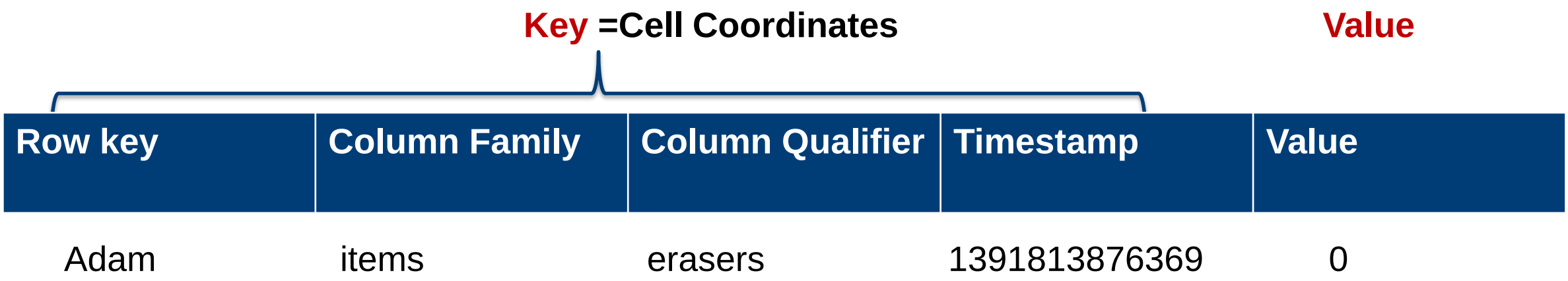
```
byte[] b = result.getValue(columnFamilyName,columnName1);
```

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/



# Bytes class

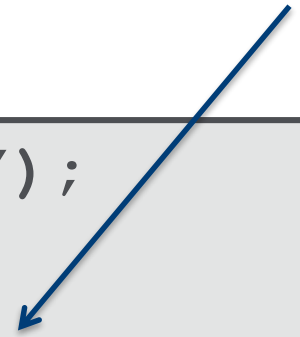
http://hbase.apache.org/0.94/apidocs/org/apache/hadoop/hbase/util/Bytes.html

- 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

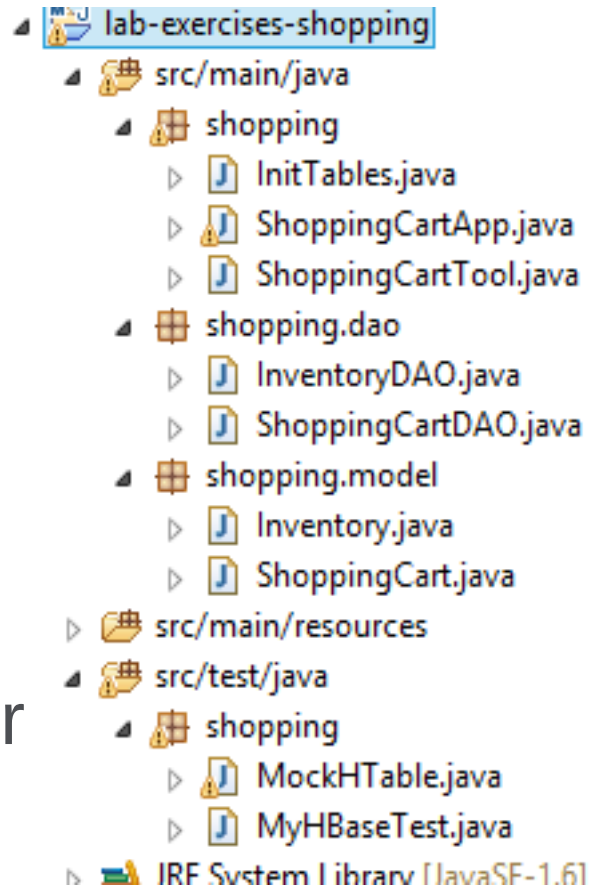
```
Scan scan = new Scan();
scan.addFamily(columnFamily);
ResultScanner scanner = myTable.getScanner(scan);
try {
    for (Result res : scanner) {
        System.out.println(res);
    }
} catch (Exception e) {
    System.out.println(e);
} finally {
    scanner.close();
}
```

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.

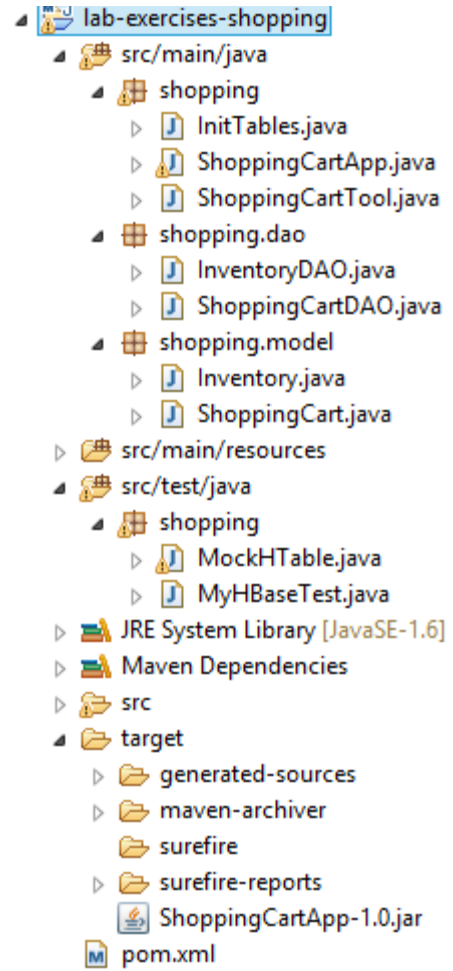


# Lab Exercise

- 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

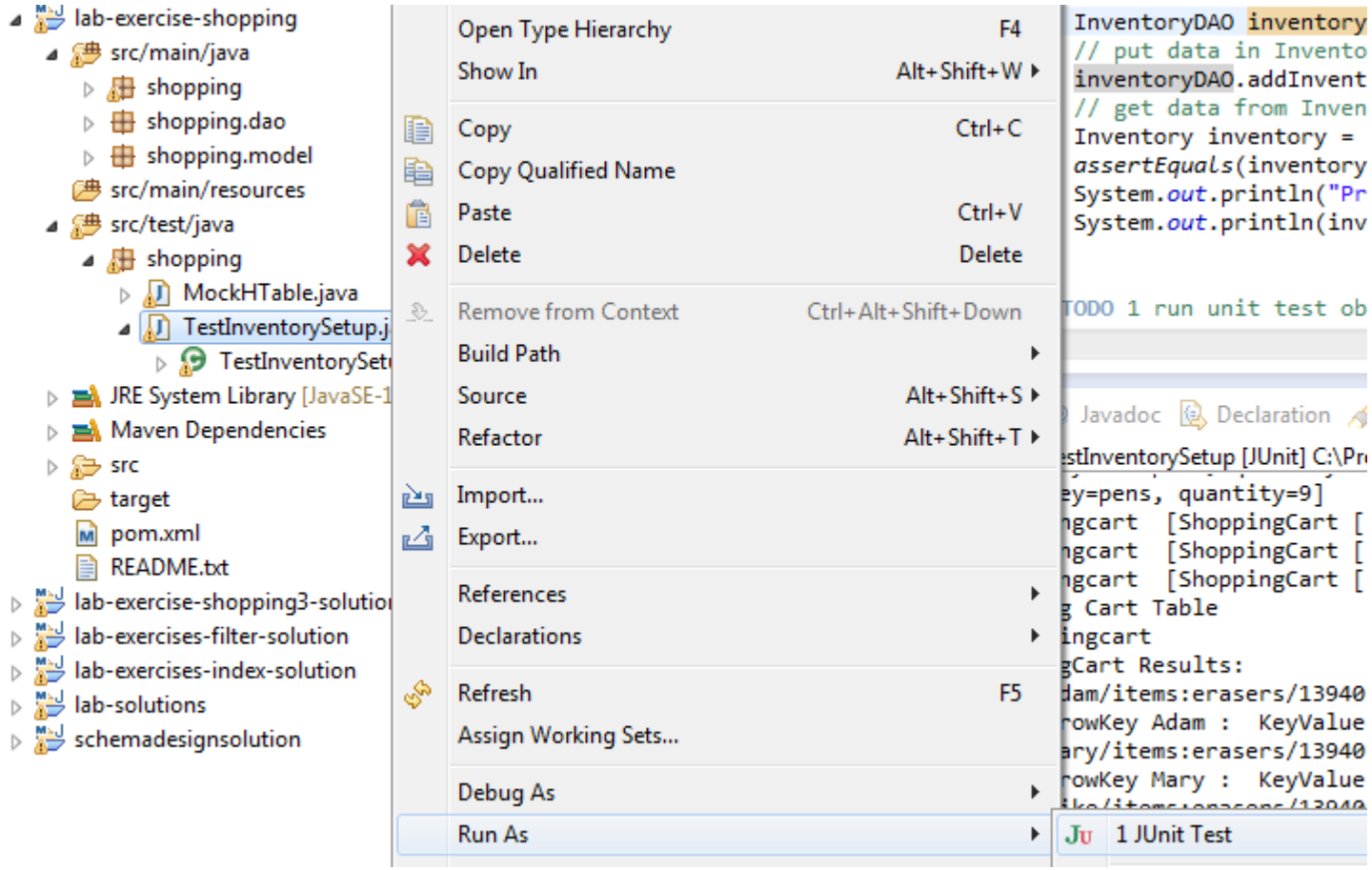
# Lab: Import, build

- **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**



# Shoppingcart Checkout functionality

# Inventory & Shoppingcart Tables

## How can we implement Checkout functionality?

### 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 |

# Inventory & Shoppingcart Tables

## after checkAndPut for Mike

### Inventory Table

| | CF "stock" | | |
|---|---|---|---|
| | **quantity** | **Mike** | **…** |
| Pens | ~~10~~ 9 | 1 | |
| Notepads | ~~21~~ 19 | 2 | |
| Erasers | ~~10~~ 7 | 3 | |
| Pencils | 50 | | |

### Shoppingcart Table

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

# checkAndPut Method

```
boolean checkAndPut(byte[] row, byte[] family,
    byte[] qualifier, byte[] value, Put put)

myTable.checkAndPut("pens", CF, COL, 10, put);
```

Check a Value

Put a row update

**Inventory Table before**

| | CF "stock" |
|---|---|
| | quantity |
| pens | 10 |

**Inventory Table after**

| | CF "stock" | |
|---|---|---|
| | quantity | Mike |
| pens | 9 | 1 |

# checkAndPut method

```
byte [] rowPens = Bytes.toBytes ("pens");
byte [] CF = Bytes.toBytes("stock");
byte [] COL = Bytes.toBytes ("quantity");
byte [] oldquantity= Bytes.toBytes(10);
byte [] amount= Bytes.toBytes(1);
byte [] newquantity= Bytes.toBytes(9);
Put put = new Put(rowPens);
put.add(CF, COL, newquantity);
put.add(CF, Bytes.toBytes("Mike"), amount);

boolean ret = myTable.checkAndPut(rowPens, CF,COL,
                                  oldquantity, put);
System.out.println("Put succeeded: " + ret);
```

Best practice

# What are the problems with this implementation?

- Conflicts: What happens when two checkout operations read the same inventory value and try to change it?
  - **One fails**
  - **Application developer has to take this into consideration and in such a case, get the latest value and try checkAndPut again …**
  - **Not efficient in a large scale system as this can happen quite a lot …**
- Is there a simpler way to do this in HBase?
  - **Yes, Use Counters – Increment**

# Counters – Overview

- Counters provide an **atomic** increment of a number

| key | stats: clicks |
|---|---|
| com.example/home | 1000 |

A counter is a long column value

- Common use cases
  - Clicks, Page hits, views
- Two types of counters:
  - **single column** counter with HTable method
  - **multiple columns** counter with an Increment object.

# Single Column Counter

- HTable incrementColumnValue method
  - Atomically increments a single column value
  - Provides atomic **read and modify**
  - Easy to use

| | cf: qualifier |
|---|---|
| row | amount |

```
public long incrementColumnValue(byte[] row,
                                 byte[] family,
                                 byte[] qualifier,
                                 long amount)
```

# HTable incrementColumnValue

```
long returnVal =
            myTable.incrementColumnValue(
            rowA,
            stats,
            clicks, // counter column name
            42L);
```

| key | stats: clicks |
|---|---|
| rowA | ~~1000~~ 1042 |

# Multiple Column Counter- Increment object

```
Increment increment1 = new Increment(rowKey);


increment1.addColumn(CF, qualifier1, amount1);
increment1.addColumn(CF, qualifier1, amount1);


Result result1 = table.increment(increment1);
```

Create Increment object with Row Key

Add columns to Increment

Call table increment

# Inventory Table Increment

| | [CF] stock | |
|---|---|---|
| | quantity | Mike |
| **pens** | ~~13~~ 12 | 1 |

```
Increment increment1 = new Increment(pens);
increment1.addColumn(stock, quantity, -1 );
increment1.addColumn(stock, like, 1 );
Result result1 = table.increment(increment1);
```

# Summary

- Why NoSQL and why Hbase
- Try these hands-on-labs on
  - MapR Sandbox or
  - MapR Developer Release (M3)
- Design considerations when migrating from RDBMS to Hbase
  - De-normalize data
  - Column Families & Columns
  - Versions
  - RowKey design
- Transactions using checkAndPut and Increment

# 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

# HBase Challenges

## Reliability

- Compactions disrupt operations
- Very slow crash recovery

## Business continuity

- Common hardware/software issues cause downtime
- Administration requires downtime
- No point-in-time recovery
- Complex backup process

## Performance

- Many bottlenecks result in low throughput
- Limited data locality
- Limited # of tables

## Manageability

- Compactions, splits and merges must be done manually (in reality)
- Basic operations like backup or table rename are complex

# Fast NoSQL with Zero Administration

**MapR-DB**

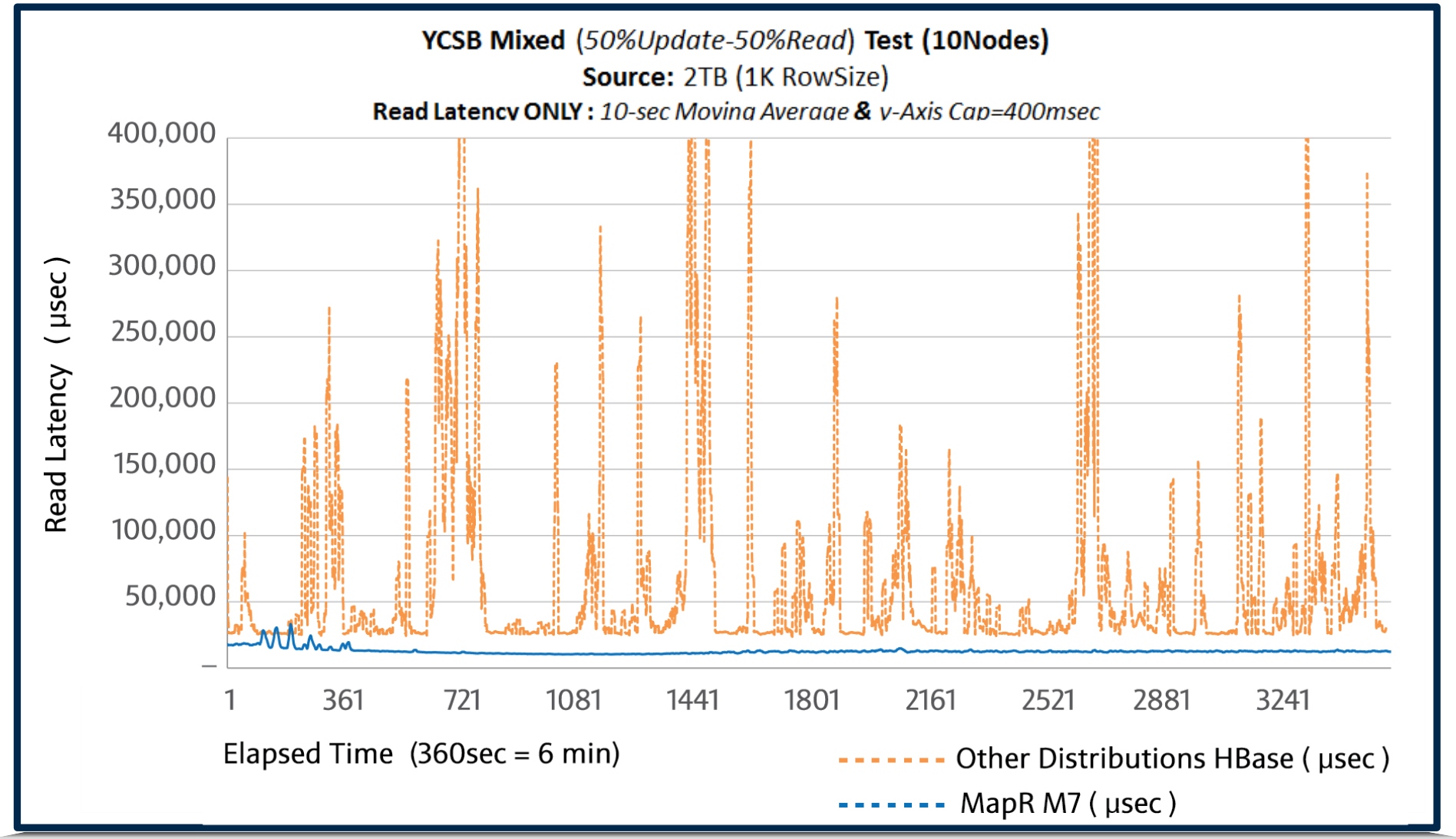
Performance

Reliability

Easy Administration

| Benefit | Features |
|---|---|
| **High Performance** | Over 1 million ops/sec with 10 node cluster |
| **Continuous Low Latency** | No I/O storms, no compactions |
| **24x7 Applications** | Instant recovery, online schema modification, snapshots, mirroring |
| **Zero Administration** | No processes to manage, automated splits, self-tuning |
| **High Scalability** | 1 trillion tables, billions of rows, millions of columns |
| **Low TCO** | Files and tables on one platform, more work with fewer nodes |

# 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*

# MapR Editions

## MapR Makes Its NoSQL Database Freely Available in Community Edition

| MapR Community Edition | MapR Enterprise Edition | MapR Enterprise Database Edition |
|---|---|---|
| **MapR-DB: Fastest in-Hadoop database** (No high availability(except Yarn), snapshot, mirroring) | Everything in *Community Edition, except MapR-DB* plus… | Everything in Enterprise Edition plus…. |
| Easy system-monitoring & management with MapR Control System | 99.999% high availability & self healing | MapR-DB Fastest in-Hadoop database with enterprise HA features [e.g. Mirroring, snapshot] |
| Real-time data flows with direct access NFS | Data protection with snapshots & mirroring | Consistent low latency with no compactions |
| World record performance | Multi-tenancy & job placement control | Zero database administration |
| | | Instant recovery, snapshots and mirroring for tables |

# Q&A

## Engage with us!

@mapr   maprtech

mapr-technologies   MapR

sreddy@mapr.com   maprtech