



# DEV 3200 – Apache HBase Applications Design and Build Slide Guide

---

Winter 2017 – Version 5.1.0

## **For use with the following courses:**

- DEV 3200 – Apache HBase Applications Design and Build
- DEV 320 – Apache HBase Data Model and Architecture
- DEV 325 – Apache HBase Schema Design
- DEV 330 – Developing Apache HBase Applications: Basics
- DEV 335 – Developing Apache HBase Applications: Advanced
- DEV 340 – Bulk Loading, Security, and Performance

This Guide is protected under U.S. and international copyright laws, and is the exclusive property of MapR Technologies, Inc.

© 2017, MapR Technologies, Inc. All rights reserved. All other trademarks cited here are the property of their respective owners.





## **DEV 320 – Apache HBase Model and Architecture**

Lesson 1: Introduction to Apache HBase

Winter 2017, v5.1

Welcome to DEV 320 – Apache HBase Model and Architecture, Lesson 1: Introduction to Apache HBase.

## Learning Goals



## Learning Goals



- 1.1 Differentiate between RDBMS and HBase
- 1.2 Identify typical HBase use cases

By the end of this lesson, you will be able to:

- differentiate between an RDBMS and HBase,
- and identify typical HBase use cases.

## Learning Goals



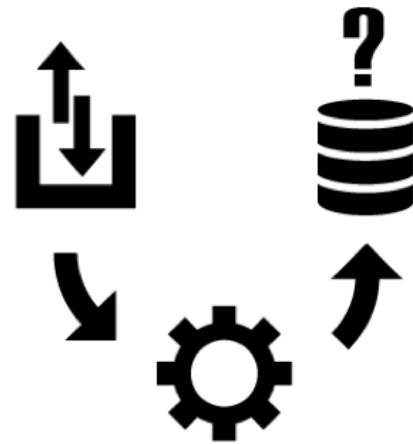
### 1.1 Differentiate between RDBMS and HBase

1.2 Identify typical HBase use cases

In this section, we will look at the differences between relational databases and HBase.

## What is Apache HBase?

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



What is Apache HBase? It is a NoSQL database that runs on top of Hadoop as a distributed scalable big data store.

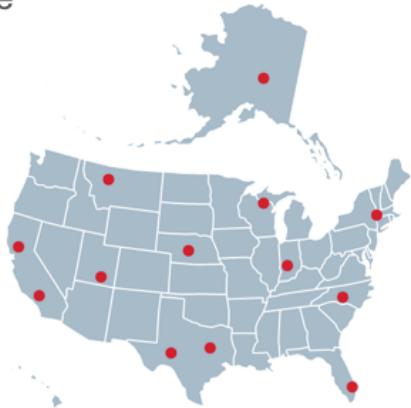
HBase is a column family oriented datastore. Column-oriented data bases save their data grouped by columns. The motivation for NoSQL databases came because of limitations of relational databases.

Before we deep dive into HBase, let us take a look at a scenario and see if a relational database can be used in this use case. We will then discuss how HBase compares to relational databases.

## Discussion - Scenario

Big Office Supply Company (BOSC) sells office supplies via different outlets:

- Stores around the country
- Website



An office supply company, the Big Office Supply Company, sells their products through different outlets. They have their own stores around the country, and an online shop. They want to expand their online presence.

## Discussion – Requirements

Track the following:

- Point of Sales
- History Data
- Social Media



They are interested in tracking point of sales from all the different channels to help manage their inventory. They want to track their shopper's purchase history data.

They also want to track social media to find out what is trending, if there are any likes, any negative comments in Facebook, Twitter, etc. and to increase their online presence. They want to add more targeted ad campaigns.

## Discussion – Requirements

Want to increase web presence

- Targeted web marketing campaigns
- Creating mobile apps



They are interested in increasing their web presence to create more targeted web marketing campaigns. They are also creating mobile applications and hope to gather some data from these apps to use for targeted marketing campaigns.

## Discussion – Questions

1. What does the data look like?
2. Is the data model likely to change?
3. Do you think this company has large volumes of data?
4. What kind of analysis would they be doing on this data?



When deciding what database best suits our data, here are some questions we may want to ask.

1. What does the data look like?
2. Is the data model likely to change?
3. Do you think this company has large volumes of data?
4. What kind of analysis would they be doing on this data?

## Discussion – Analysis

- Point of sales: What type of data? Is it going to change
- Shoppers' purchase history data: What type of data?
- Social media / mobile apps: what are some examples of data from social media?

01101010101000110010  
011011010000100010  
001010110101



Consider the point of sales data. Is this structured or unstructured? Is this data going to vary across the different points of sale?

What about purchaser history data?

What can you say about data from social media? Is this structured? Do you think the data that is collected from mobile apps will change? Do you think the data that is collected from mobile apps will change? Do you think the number of users will increase?

## Scenario – Analysis

1. Maybe large volumes
2. More or less same data from all sources
3. Definitely large volumes
4. Application constantly changing – so data changing
5. Simple tabular structure
6. No real structure
7. Number of users growing
8. Real-time analytics

point of  
sales/purchase  
history

mobile apps/social  
media

Discussion for the class: Which characteristics apply to which scenarios?

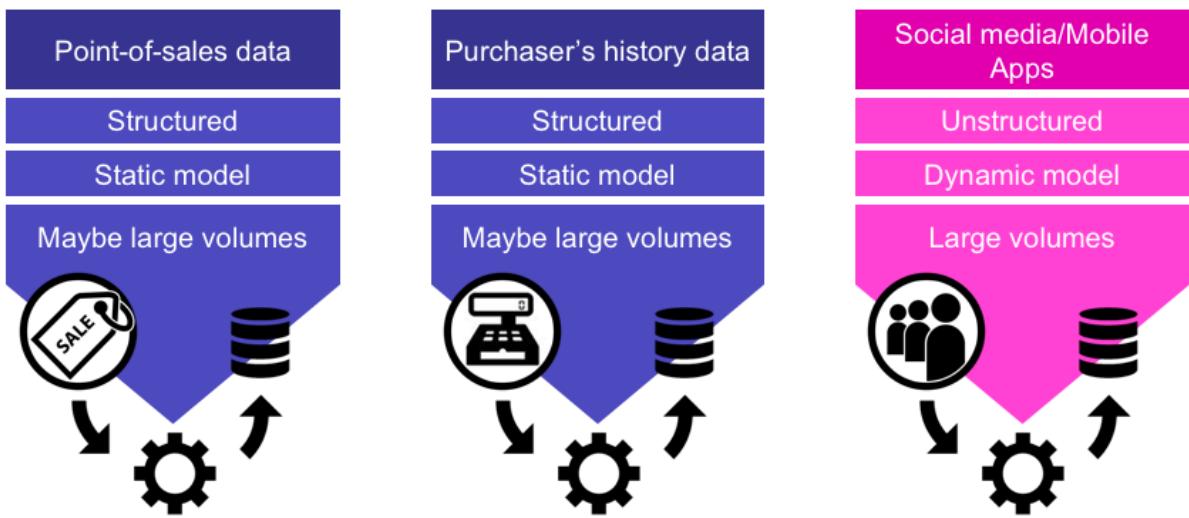
## Scenario – Analysis

1. Maybe large volumes
2. More or less same data from all sources
3. Definitely large volumes
4. Application constantly changing – so data changing
5. Simple tabular structure
6. No real structure
7. Number of users growing
8. Real-time analytics

point of  
sales/purchase  
history  
**1,2,5**

mobile apps/social  
media  
**3,4,6,7,8**

## Conclusion



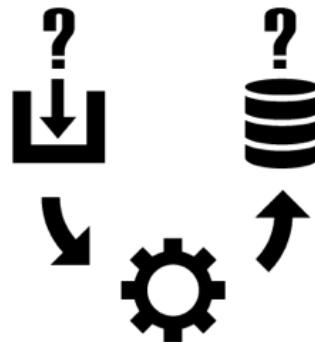
Point-of-sales data and purchasers' history data is most probably going to be structured and a reasonably static schema. Volume of data can grow, but maybe not exponentially.

A relational database can be used for this. However, we also need to take into account data from social media.

This will be unstructured data which will require a dynamic data model and can also produce exponentially large volumes. Since the mobile apps are just being developed, the data that is collected from mobile apps can change easily. This means that you will need a flexible data model. In this case, you would want to consider HBase.

## How do you pick?

1. What does your data look like?
2. Is your data model likely to change?
3. Is your data growing exponentially?
4. Will you be doing real-time analytics on operational data?



If you decide to go with NoSQL, you then have to decide which NoSQL database will suit your needs! HBase is an example of a NoSQL datastore

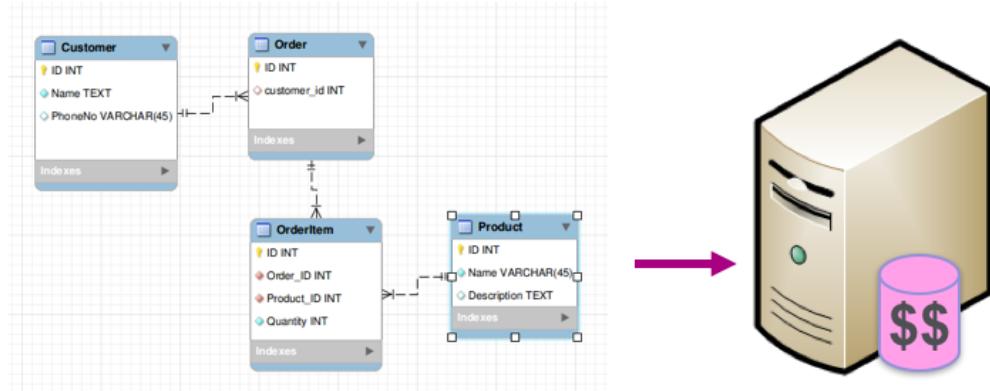
Here are some things to consider in making your decision to pick a NoSQL database over a relational database.

1. What does your data look like? Are there multiple levels of hierarchies? If your data does not have a simple tabular structure, then you may want to consider NoSQL.
2. Is your data model likely to change? Do you have all the information you need at the time of design? If not, then you need the flexibility to change the schema - NoSQL
3. Is your data growing exponentially? Are the number of users going to multiply? This has to do with being able to scale easily and cheaply – then NoSQL.
4. Will you be doing real-time analytics on operational data? If you are looking at data that is brought together from many upstream systems to build an application, use NoSQL.

Now let us take a step back and take a look at the pioneers and history of HBase. The next few slides will go over some of the limitations of relational databases and how HBase compares.

## Relational Databases vs HBase - Scaling

### RDBMS - Scale UP approach



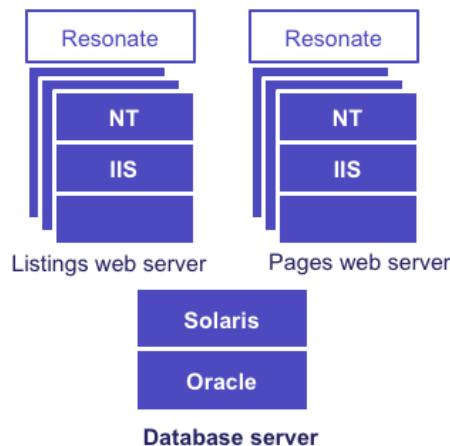
Vertical scale = **big box**

Let us take a look at some of the challenges that people face nowadays with relational databases.

Relational databases were the standard for years, so what changed? With more and more data came the need to scale. One way to scale is vertically with a bigger server. This can get expensive, however, and there are limits as your size increases.

## RDBMS Scaling Up Example - eBay

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



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

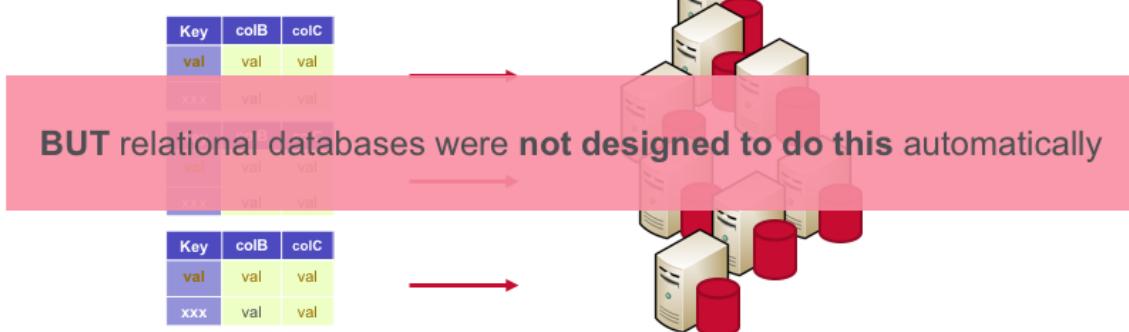
© 2016 MapR Technologies

L1-16

Here is an example of how a relational database was scaled vertically. In 1999, eBay scaled vertically by buying bigger servers, but this did not scale for very long.

## Relational Databases vs. HBase - Scaling (2)

RDBMS - Scale OUT Approach



**Cheaper than vertical  
Horizontal scale = parallel execution  
high reliability**

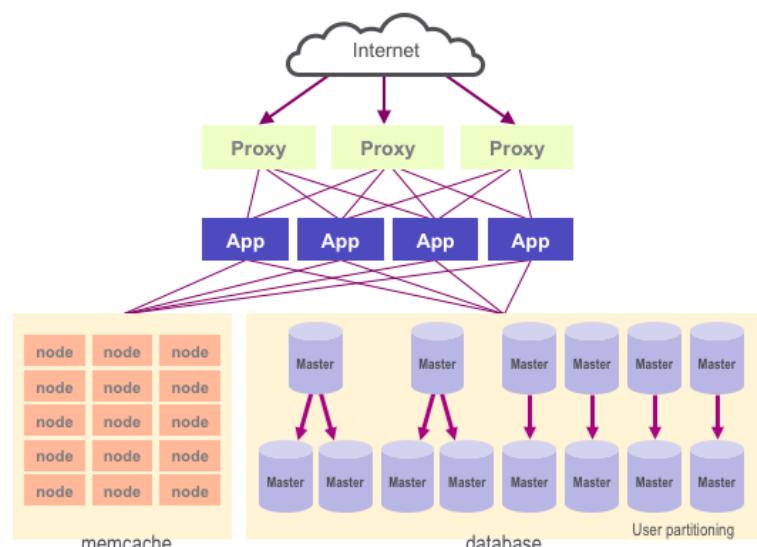
An alternative to vertical scaling is to scale horizontally with a cluster of machines, which can use commodity hardware. This can be cheaper and more reliable.

To horizontally partition, or shard, an RDBMS, data is distributed on the basis of rows. Some rows reside on a single machine and the other rows reside on other machines. It is complicated to partition or shard a relational database, as it was not designed to do this automatically. You also lose the querying, transactions, and consistency controls across shards. Relational databases are not designed to run efficiently on clusters.

## RDBMS Scaling Out Example –Facebook MySQL

### FACEBOOK

- 9000 memcached instances
- **4000 Shards mysql**



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

© 2016 MapR Technologies

L1-18

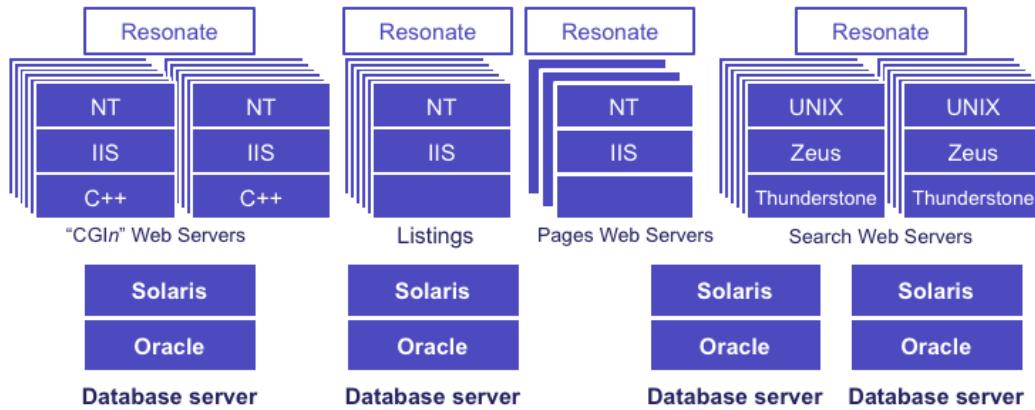
Facebook paired complex sharding and caching to MySQL.

Facebook split its MySQL database into approximately 4,000 shards, and 9,000 instances of memcached, in-memory key-value store for small chunks of data, in order to handle the site's massive data volume. This became very difficult to maintain and scale, and now Facebook has moved their messaging to HBase. There are several videos on the internet about this.

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

## RDBMS Scaling Out Example – eBay v2.4

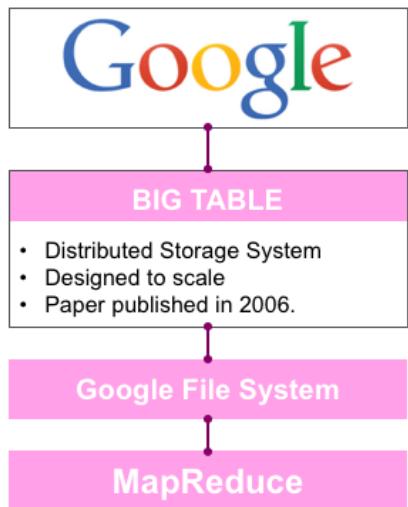
- Split databases **horizontally**
- Logically partition databases for user data, item, data, purchase data



<http://www.slideshare.net/RandyShoup/best-practices-for-largescale-websites-lessons-from-ebay>

In late 1999, EBAY scaled out across a cluster by logically partitioning their databases for user data, item data, and purchase data. However this SQL option did not scale enough for EBAY, and they have now moved their items catalog to HBase.

## Google NoSQL BigTable



Designed to run on a cluster

- sparse
- distributed
- persistent
- multi-dimensional sorted map

HBase based on Bigtable

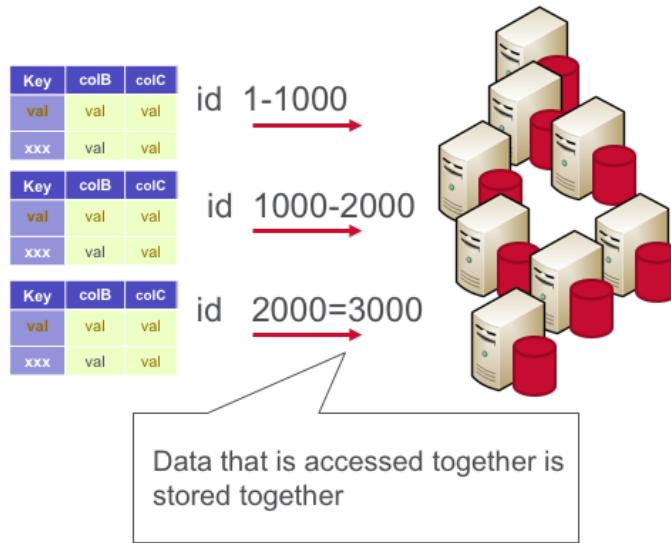
In 2006 Google came up with BigTable, which was designed to run on a cluster. Google published a paper titled “Bigtable: A Distributed Storage System for Structured Data”.

Bigtable is a distributed storage system for managing structured data that is designed to scale to a very large size: petabytes of data across thousands of commodity servers. It is a sparse, distributed, persistent, multi-dimensional sorted map.

HBase is the open source implementation of the the Bigtable storage architecture.

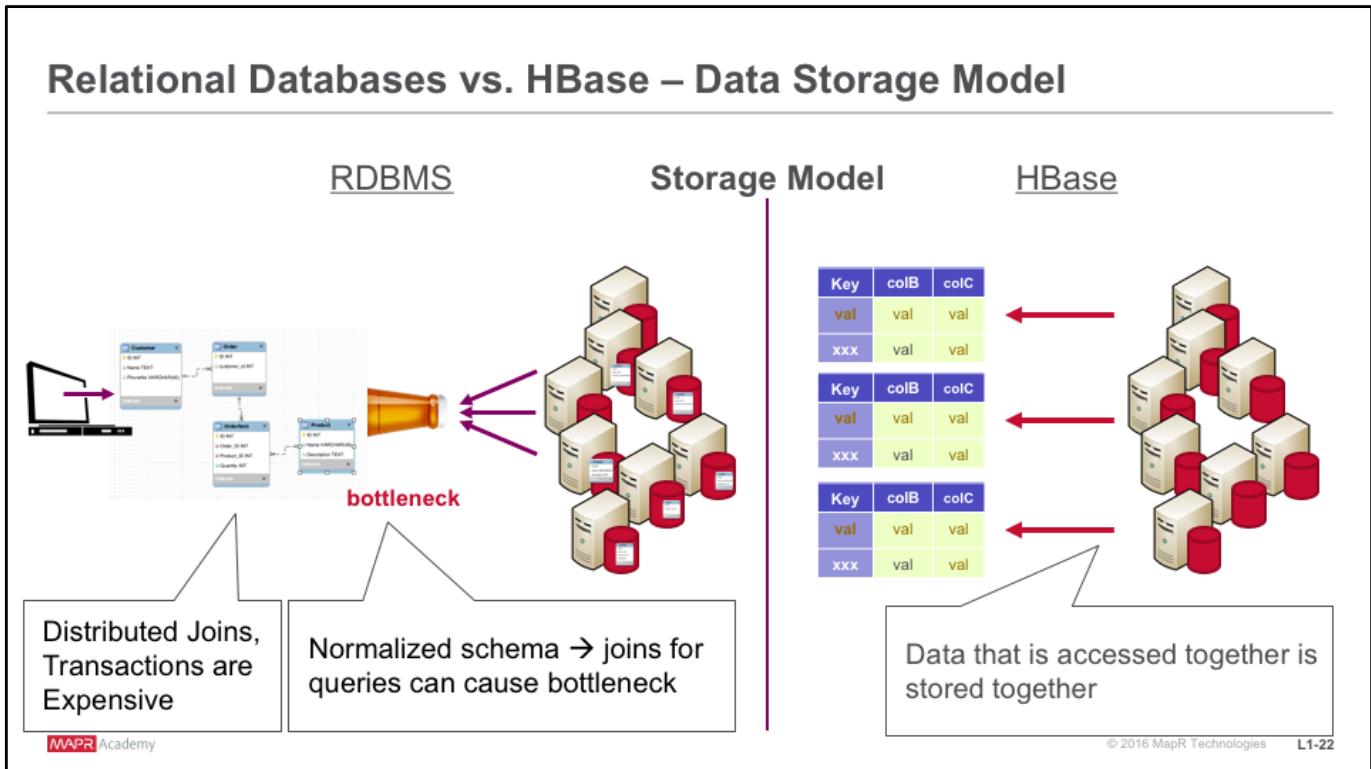
## Relational Databases vs. HBase - Scaling (3)

### HBase – Designed for Cluster



The vital factor for HBase was the ability to support large volumes of data by running on clusters.

HBase was designed to scale due to the fact that data that is accessed together is stored together. Grouping the data by key is central to running on a cluster. In HBase, the data is automatically distributed across a cluster. Sharding distributes different data across multiple servers, and each server is the source for a subset of data. Distributed data is accessed together which makes it faster for scaling.

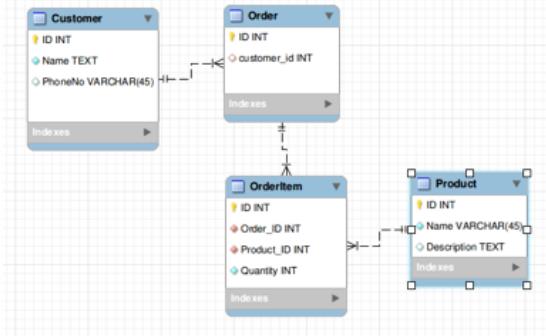


Database normalization eliminates redundant data, which makes storage efficient. However a normalized schema causes joins for queries, to bring the data back together again. Complex JOIN queries can be difficult to implement and maintain, and also use up a lot of database resources. As shown in the figure, you can end up with a bottleneck.

HBase does not support relationships and joins, so it can avoid the limitations associated with a relational model.

## Relational Databases vs. HBase – Schema Flexibility

### RDBMS



### Schema Flexibility

### HBase

First Name	Albert
Last Name	Einstein
Address	2 Physics Place
City	New York
State	NY
Zip Code	10003
Order Number 1	1003
First Name	John
Last Name	Doe
Address	1 Junction Avenue
City	San Jose
State	CA
Zip Code	95134
Order Number 1	1001
Item 1 - 1	Hamilton Beach Coffee Maker   1   54.99   54.99
Item 1 - 2	Kindle Fire 7"   1   139.00   139.00
Order Number 2	1002
Item 2 - 1	Divergent (paperback)   1   5.49   5.49
Item 2 - 2	Seagate Desktop HDD 4TB   1   156.95   156.95

Relational DBs do not handle variable data. An RDBMS uses defined data types, and any changes to the schema means altering the database.

In HBase however, two records do not have to look alike. All data is stored as bytes, and new data types or structures can be added dynamically, as can new fields.

## Summary

CAPABILITIES	RDBMS	HBase
Scaling	Afterthought	Designed for it
Transactions	Distributed transactions	Row-based only
Data Model	Fixed	Flexible
Joins	Yes	No

Here is a summary of capabilities of relational databases versus HBase.

## Knowledge Check



## Knowledge Check



1. Complex query requirements
2. Row-based transactions
3. Rapid data volume growth
4. On-Line transaction processing (OLTP) required
5. ACID (Atomicity, Consistency, Isolation, Durability) required
6. Session data
7. Data changes over time

RDBMS

HBase

Match the features listed on the left with the appropriate database, an RDBMS or HBase.

## Knowledge Check



- |  |         |
|--|---------|
| 1. Complex query requirements                                    | RDBMS   |
| 2. Row-based transactions  | 1,4,5   |
| 3. Rapid data volume growth                                      |         |
| 4. On-Line transaction processing (OLTP) required                |         |
| 5. ACID (Atomicity, Consistency, Isolation, Durability) required |         |
| 6. Session data  | HBase   |
| 7. Data changes over time  | 2,3,6,7 |

## Learning Goals



## Learning Goals



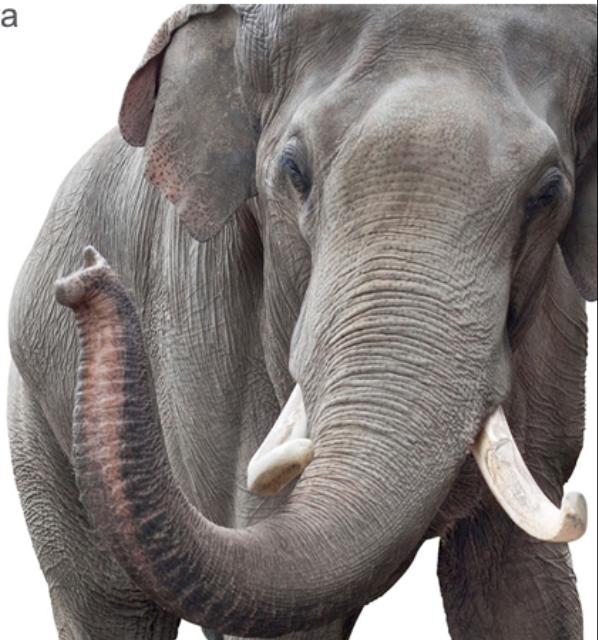
1.1 Differentiate between RDBMS and HBase

**1.2 Identify typical HBase use cases**

Now that you have seen the difference between RDBMS and HBase, let's take a look at some typical HBase use cases.

## Main Use Case Categories

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



MAPR Academy

HBase has been used in three major types of use cases.

### Capturing Incremental Data

This is the case with high volume and velocity writes. Some examples:

- Capturing Metrics: OpenTSDB
- Capturing User-Interaction Data: Facebook and Stumbleupon
- Advertisement Impressions and Clickstream with detailed capture and analysis of user-interaction data

### Information Exchange

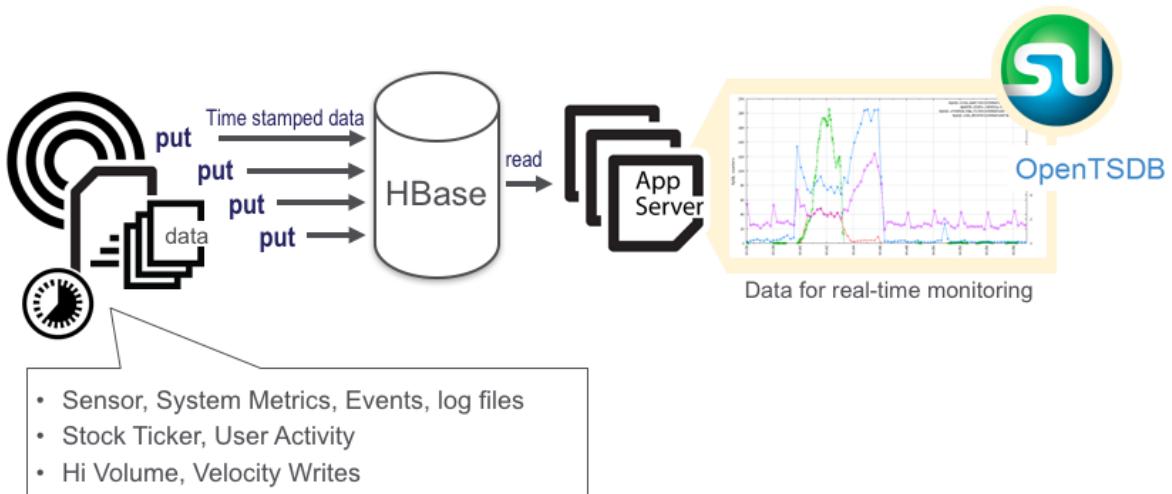
These cases represent high volume and velocity writes and reads. An example is Facebook messages.

### Content Serving

These cases use high volume and velocity reads.

For example, consider a web store that refers to web applications that serve data for millions of Internet users. Web applications require a data store that can manage a huge amount of data.

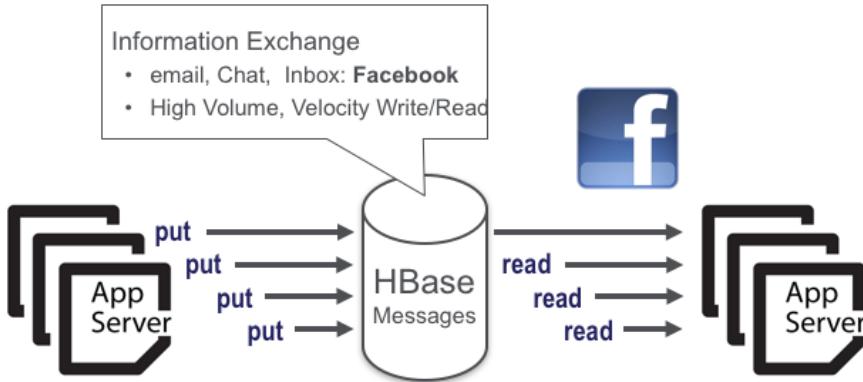
## Time Series Data



Many real-world applications today can fall under the category of time series applications that manage and store data generated over time, such as server logs, performance metrics, sensor data, a stock ticker, and transport tracking.

Data often trickles in and is added to an existing data store for further usage, such as analytics, processing, and serving. Many HBase use cases fall in this category, using HBase as the data store that captures incremental data coming in from various data sources.

## Information Exchange



<https://www.facebook.com/UsingHBase>

MAPR Academy

© 2016 MapR Technologies

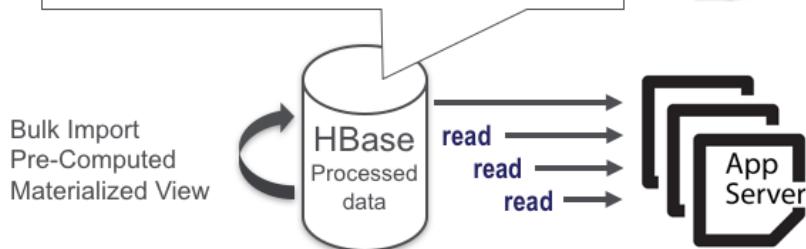
L1-32

Information Exchange use cases are characterized by high volume reads and writes. An example is the enormous volume of reads and writes used storing Facebook message histories.

## Content Serving Web Application Backend

Content Serving, Web Application Backend

- Online Catalog: Gap, World Library Catalog.
- Search Index: ebay
- Online Pre-Computed View: Groupon, Pinterest
- High Volume, Velocity Reads



A content serving web application is read heavy, and is characterized by high volume, high velocity reads.

An example is a web store for web applications, that requires a data store that can manage a huge amount of data for millions of Internet users.

Writes to the database are random with a few updates. Writes have less throughput, and low latency.

Reads to the data base are low latency. EBay uses HBase to perform low latency reads for their items catalog.

## Knowledge Check



## Knowledge Check



Which example best fits the use case category?

1. Time series data

Email, Chat

2. Information exchange

Online catalog

3. Content serving web application

System metrics



## Knowledge Check

Which example best fits the use case category?

1. Time series data

2 Email, Chat

2. Information exchange

3 Online catalog

3. Content serving web application

1 System metrics

## References

- <http://hadoop.apache.org>
- <http://hbase.apache.org>
- <http://www.mapr.com>
- NoSQL Distilled, Pramod J. Sadalage, Martin Fowler
- HBase in Action, Nick Dimiduk, Amandeep Khurana
- HBase: The Definitive Guide, Lars George

Here are some references to get more information.



## Next Steps

### DEV 320 – Apache HBase Model and Architecture

Lesson 2: Apache HBase Data Model

In the next lesson, we will take an in-depth look at the HBase data model.



## **DEV 320 – Apache HBase Data Model and Architecture**

Lesson 2: Apache HBase Data Model

Winter 2017, v5.1

Welcome to DEV 320 – Apache HBase Data Model and Architecture, Lesson 2: Apache HBase Data Model.

## Learning Goals



## Learning Goals



- 2.1 List data model components
- 2.2 Describe how an HBase table maps to physical storage
- 2.3 Use data model operations
- 2.4 Create an HBase table

This lesson describes the HBase data model. You will look at how data is stored in HBase table cells and how an HBase table is physically stored on disk. You will also use basic operations to create an HBase table using the HBase shell.

## Learning Goals



### 2.1 List data model components

- 2.2 Describe how an HBase table maps to physical storage
- 2.3 Use data model operations
- 2.4 Create an HBase table

In this section we will take a look at the HBase data model components.

## HBase Data Model Components

Component	Description
Table	Data organized into tables; comprised rows
Row key	Data stored in rows; rows identified by row keys
Column family	Columns grouped into families
Column Qualifier	Identifies the column
Cell	Combination of the row key, column family, column, version; contains the value
Version	Values within cell versioned by version number → timestamp

Let's list the components of the HBase data model.

Data is organized into HBase tables. Tables are made up of a number of rows. Data is stored in rows.

Each row is uniquely identified by its row key.

Each row is composed of columns that are grouped into column families.

Columns are identified by column qualifiers, or column names.

Every column value or cell is versioned by a timestamp. The entire cell, with the added structural information, is called KeyValue in HBase terms. The entire cell, the row key, column family name, column name, timestamp, and value are stored for every cell for which you have set a value. The key consists of the row key, column family name, column name, and the timestamp.

The value in a cell is versioned by version number, which is the timestamp of when the cell was written.

We will go into how to create an HBase table later. Now let us take a look at some of the other components in a little more detail.

## HBase Data Model – Rowkeys

Row key	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							
twistr	val		val	val			val	
...								
zaxx	val	val	val	val	val	val		
zxxx	val						val	

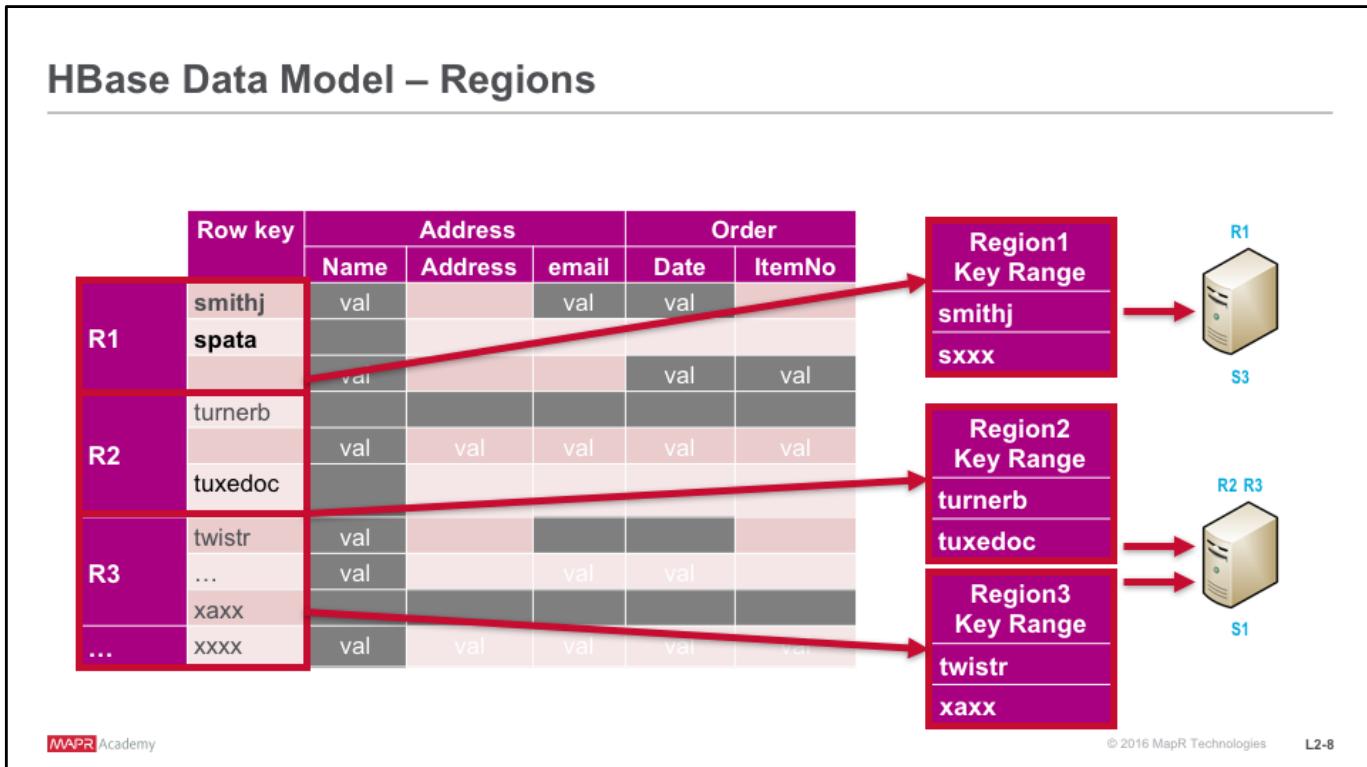
Row keys: identify the rows in an HBase table.

Let us look at a row key in more detail. Data stored in HBase is located by a unique identifier, its **row key**. This is like a primary key from a relational database. Once you have set up and added data to the HBase table, you cannot change the row key to be another column. In the above example, once data has been put in the table, you cannot pick the column, **street** in **Address**, to be the row key.

## HBase Data Model – Row keys

- Sorting of row key is based upon binary values
  - Sort is lexicographic at byte level
  - Comparison is “left to right”
- Example:
  - Sort order for String 1, 2, 3, ..., 99, 100:
    - 1, 10, 100, 11, 12, ..., 2, 20, 21, ..., 9, 91, 92, ..., 98, 99
  - Sort order for String 001, 002, 003, ..., 099, 100:
    - 001, 002, 003, ..., 099, 100

Records in HBase are stored in sorted order according to the row key. This is a fundamental tenet of HBase, and is also a critical semantic used in HBase schema design. Row keys are stored in lexicographical order, which is why **Row 100** comes before **Row 2** in the example shown here.



Tables are divided into sequences of rows, by row key range, called regions. These regions are then assigned to the data nodes in the cluster called RegionServers. This scales read and write capacity by spreading it across the cluster. This is done automatically and this is how HBase was designed for horizontal sharding.

## HBase Data Model – Column Family

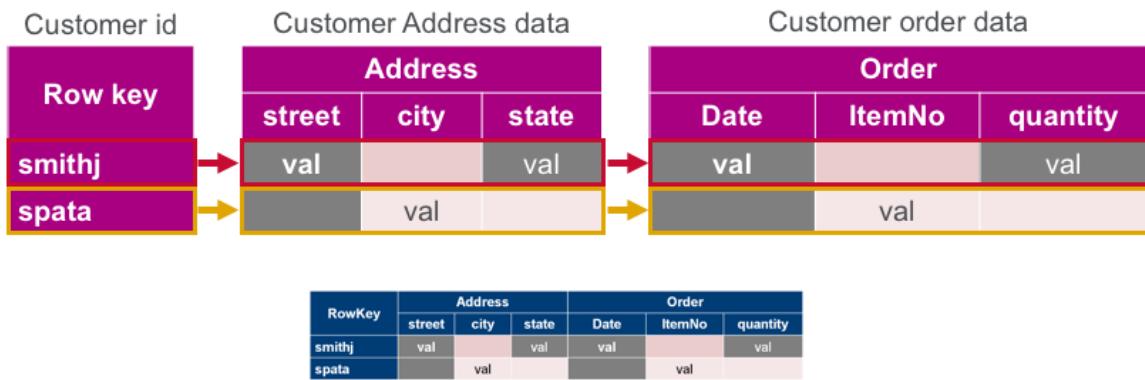
Row key	Address			Order			
	street	city	state	Date	ItemNo	Quantity	Price
smithj	Central Dr	Nashville	TN	2/2/15	10213A	1	109.99
spata	High Ave	Columbus	OH	1/31/14	23401V	24	21.21
turnerb	Cedar St	Seattle	WA	7/8/14	10938A	15	1.54

Each row is comprised of columns grouped into column families. Columns are added dynamically. Column families can contain an arbitrary number of columns.  
Every row in a table has the same number of column families.

## HBase Data Model – Column Family

Column Families are part of the table **schema**.

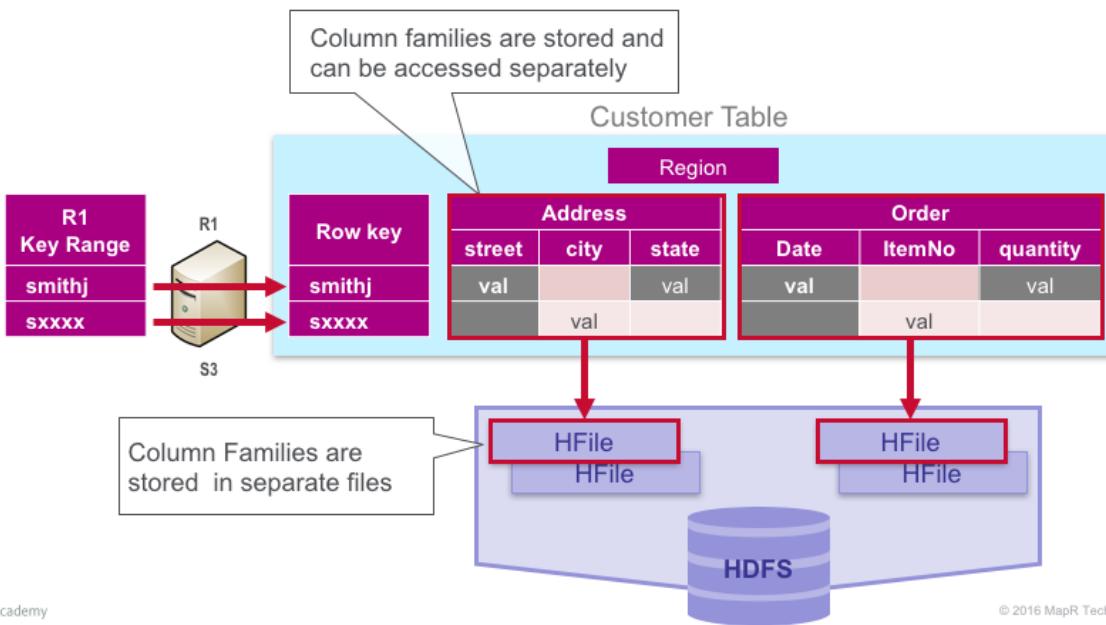
Columns are **grouped** into column families:



Column family grouping should be such that it facilitates common access patterns of data. Columns that read or write together make good column families. In the above example, customer address data is grouped together and customer order data is grouped together.

Also note that in-memory column families provide fast access to small data. We will talk later about access patterns and also designing schemas for better performance.

## HBase Data Model – Column Family



HBase data is stored in Hfiles on HDFS, the Hadoop Distributed File System. Column families are stored in separate files and can also be accessed separately.

## HBase Data Model – Columns

Row key	Address			Order
	street	city	state	
smithj	Central Dr	Nashville	TN	
...				

CF:col ↔ Address:street

Column families are made of columns. Column names or column qualifiers identify the column.

Here we are looking at a table that contains customer information. The **Address** column family contains the following columns; **street**, **city**, and **state**.

## HBase Data Model – Cells

Data is stored in **KeyValue** format

Value for each **cell** is specified by complete coordinates:

(Row key, ColumnFamily, Column Qualifier, timestamp ) => Value

- Row key:CF:Col:Version:Value
- smithj:data:city:1391813876369:nashville

Cell Coordinates= Key				Value
Row key	Column Family	Column Qualifier	Timestamp	Value
Smithj	Address	city	1391813876369	Nashville

The data is stored in HBase table cells. The entire cell, with the added structural information, is called **KeyValue** in HBase terms. The entire cell, the row key, column family name, column name, timestamp, and value are stored for every cell for which you have set a value.

The Key consists of the row key, column family name, column name, and timestamp.

In the next slide, we will see how the **KeyValue** works when retrieving data.

## HBase Data Model – Cells

```
smithj → {Address:{street:{timestamp2: Main St.}, city:{timestamp1:Nashville},state:{timestamp1:TN}}
          {Order{Date:{timestamp1:10213A},quantity:{timestamp1:1}}}
```

		Address			Order	
Row	key	street	city	state	Date	ItemNo
timestamp1	smithj	Central Dr	Nashville	TN	2/2/15	10213A
timestamp2	spata	Main St	High Ave	OH	1/31/14	23401V
	turnerb	Cedar St	Seattle	WA	7/8/14	10938A

- 1.If you were to retrieve the item that a row key maps to, for example smithj, you get data from all the columns.
- 2.If you retrieved the item from the row key, and column family, you would get the data for the columns in that column family.
- 3.In addition, if you specify the column qualifier, you would get all the timestamps and associated values.
4. You can then specify which timestamp and associated value you would like to return. By default, only the latest version is returned.

## Knowledge Check



## Knowledge Check



The cell in an HBase table is called KeyValue in HBase terms. The Key consists of:

- A. Row key, column family name, cell
- B. Column family name, column name, cell, timestamp
- C. Row key, column family name, column name, timestamp
- D. Row key, column name, version, value

## Knowledge Check



The cell in an HBase table is called KeyValue in HBase terms. The Key consists of:

- A. Row key, column family name, cell
- B. Column family name, column name, cell, timestamp
- C. **Row key, column family name, column name, timestamp**
- D. Row key, column name, version, value

## Knowledge Check



Once you set up the table and have added data, you

- A. Can change the column families
- B. Cannot change the row key to be another column
- C. Cannot add columns

## Knowledge Check



Once you set up the table and have added data, you

- A. Can change the column families
- B. **Cannot change the row key to be another column**
- C. Cannot add columns

## Learning Goals



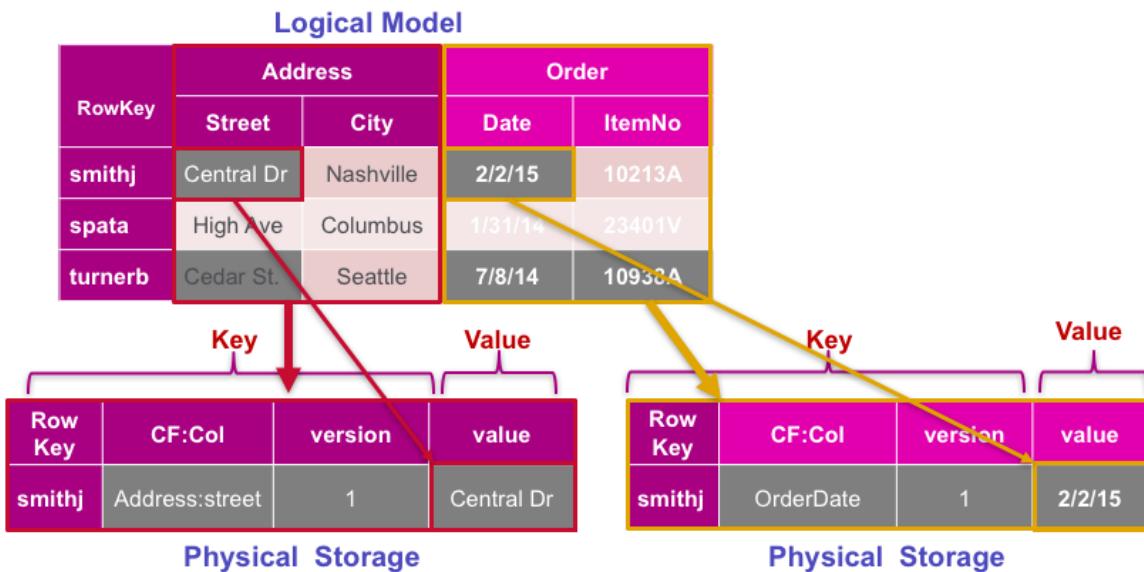
## Learning Goals



- 2.1 List data model components
- 2.2 Describe how an HBase table maps to physical storage**
- 2.3 Use data model operations
- 2.4 Create an HBase table

In this section, you will see how the logical HBase table model maps to physical storage on disk.

## Logical Data Model vs Physical Data Storage



Logically cells are stored in a table format, but physically rows are stored as linear sets of cells containing all the key value information inside them.

The table on top shows the logical layout of the data.

The lower tables shows the physical storage in files. Column families are stored in separate files. The entire cell, the row key, column family name, column name, timestamp, and value are stored for every cell for which you have set a value.

## Sparse Data with Cell Versions

	CF1:colA	CF1:colB	CF1:colC
Row1		@time7: value3 @time6: value3 @time5: value3	
Row10		@time2: value1	@time2: value1
Row11		@time6: value2 @time5: value2	
Row2		@time4: value1	
			@time4: value1

As mentioned before, complete coordinates to a cell's value are: Table:Row:Family:Column:Timestamp  
→ Value.

HBase tables are sparse, if data doesn't exist at a column, it's not stored.

Table cells are versioned, uninterpreted arrays of bytes. The version is by default a timestamp that is a long, and you can also set up your own versioning system. So for every coordinate row:family:column, there can be multiple versions of the value.

## Versioned Data

Key	CF:Col	version	value
smithj	Address:street	v3	19 <sup>th</sup> Ave
smithj	Address:street	v2	Main St
smithj	Address:street	v1	 Central Dr

Versioning is built-in. A **put** is both an **insert** (create) and an **update** and each one gets its own version. **Delete** gets a tombstone marker. The tombstone marker prevents the data being returned in queries.

**Get** requests return specific version(s) based on parameters. If you do not specify any parameters, the most recent version is returned. You can configure how many versions you want to keep and this is done per column family. The default is to keep only one version. When the max number of versions is exceeded extra records will be eventually removed.

The version or timestamp is part of the key and you can access it from a Result object. You can specify what versions you want to delete.

## 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

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	

Review of physical storage properties:

- The physical layout in storage is different than the logical representation of a table. Physically, data is stored by column family as a sorted map of nested maps.
- Rowkey:ColFamily maps to a sub-map of Columns
- Rowkey:ColFamily:Column maps to a sub-sub-map of versions
- All values are stored with the full coordinates: Table:Row:ColFamily:Column:Timestamp → Value
- Every cell version is a separate entry in the table, represented as a row in this slide
- Physically the data is stored by column family as a sorted map

## Logical Data Model vs Physical Data Storage

**Logical Model**

RowKey	Address		Order	
	Street	City	Date	ItemNo
smithj	Central Dr.	Nashville	2/2/15	10213A
spata	Main St	High Ave	Columbus	1/31/14
turnerb	Cedar St.	Seattle	7/8/14	10938A

Key

Value

Row Key	CF:Col	version	value
smithj	Address:street	2	Main St
smithj	Address:street	1	Central Dr.

**Physical Storage**

Key

Value

Row Key	CF:Col	version	value
smithj	Order:Date	1	2/2/15 51

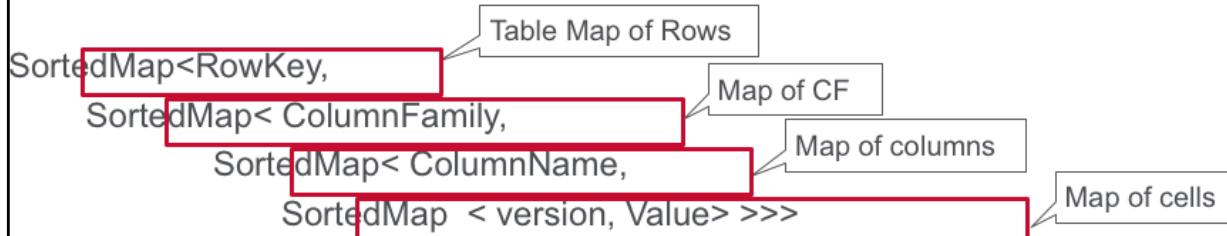
**Physical Storage**

The lower-right part shows the physical storage in files. Column families are stored in separate files. The entire cell, the row key, column family name, column name, timestamp, and value are stored for every cell for which you have set a value.

The rows are sorted by row key first, and then by column qualifier/name. If there is more than one value per cell, then it is sorted by timestamp descending, with the most recent first.

## HBase Table is a Sorted Map of Maps

In Java this is the table:



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

It is best to think of an HBase table as a map of a map: an outer-sorted map keyed by a row key, and an inner-sorted map keyed by column name.

In Java the table is:

```
SortedMap<RowKey,
  SortedMap<familyName,
    SortedMap<ColumnName,
      SortedMap<TimeStamp, ColumnValue>>>
```

or

```
SortedMap<RowKey, List<SortedMap<ColumnKey, ColumnValue>>>
```

## 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

This is how rows are stored in files :

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

## Knowledge Check



## Knowledge Check



Which are true of an HBase table?

- A. They are sparse tables
- B. Table cells are versioned
- C. Empty cells are populated with NULL values
- D. Data types that can be used in tables are arrays of bytes

## Knowledge Check



Which are true of an HBase table?

- A. They are sparse tables
- B. Table cells are versioned
- C. Empty cells are populated with NULL values
- D. Data types that can be used in tables are arrays of bytes

## Knowledge Check



**How does physical layout in storage differ from the logical representation of a table?**

- A. Data is stored by column family as a sorted map of nested maps
- B. All values are sorted with the full coordinates i.e. Key and Value
- C. Every cell version is a separate column

## Knowledge Check



How does physical layout in storage differ from the logical representation of a table?

- A. Data is stored by column family as a sorted map of nested maps
- B. All values are sorted with the full coordinates i.e. Key and Value
- C. Every cell version is a separate column

## Knowledge Check



Which are true of HBase versioning?

- A. Number of versions can be configured per column family
- B. If you don't specify a version, all versions are returned.
- C. Version by default is the timestamp.

## Knowledge Check



### Which are true of HBase versioning?

- A. Number of versions can be configured per column family
- B. If you don't specify a version, all versions are returned.
- C. Version by default is the timestamp.

## Learning Goals



## Learning Goals

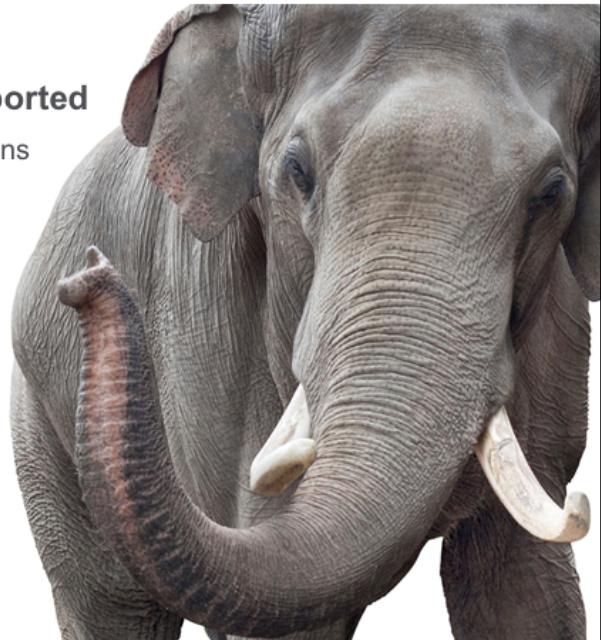


- 2.1 List data model components
- 2.2 Describe how an HBase table maps to physical storage
- 2.3 Use data model operations**
- 2.4 Create an HBase table

In this section, you will take a look at some basic data model operations.

## Basic Table Operations

- **Create Table**
- **Define Column Families before data is imported**
  - but not the rows keys or number/names of columns
- Columns defined on the fly



MAPR Academy

Once you have created a table you define column families. Columns may be defined on the fly. You can define them ahead of the time but that is not necessary. Normally, you will define rows when you add the data.

## Basic Table Operations

### Basic data access operations (CRUD)

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

Table operations are fairly simple.

**put** - Inserts data into rows, both add and update. **Put** will add new rows to the table if the key is new or update an existing row if the key already exists.

**get** - Returns the data from a specified row.

**scan** - Accesses data from a range of rows. It allows iteration over multiple rows for specified attributes.

**delete** – Removes a row from a table. HBase does not modify data in place. Deletes are handled by creating new markers called tombstones. These are cleaned up along with the dead values on major compactions.

## Learning Goals



## Learning Goals



- 2.1 List data model components
- 2.2 Describe how an HBase table maps to physical storage
- 2.3 Use data model operations
- 2.4 Create an HBase table**

In this section, we will see how to create an HBase table using the HBase shell.

## Create HBase Table

An HBase table can be created using

- HBase shell
- Java API (HBaseAdmin)
- MCS

You can create an HBase table using the HBase shell or the Java API.

## Look at Sample Data – BOSC Customer Information

BOSC – Customer Info (partial data)

Customer username	Bldng number	street	city	state	item no	quantity	color	unit price
drewt101	100	Maple	Kansas City	MO	10923A	2	blue	21.29
elmg	20134	11 <sup>th</sup> Ave	Chicago	IL	21098A	5	white	99.99
josep	344	B Dr	Seattle	WA	10924A	100	red	10.99



Here is a partial data for the customers of the Big Office Supply Company.

Before we start to create a table

1. What do we want to call our table?
2. Given what we know of row keys, so far, what do we want to use as row key?
3. Looking at this sample data, how might we want to group the data into column families?
4. What columns are we going to have?

## Sample Data – The Big Office Supply Company Customer Information

Customer username	Bldng number	street	city	state	item no	quantity	color	unit price
-------------------	--------------	--------	------	-------	---------	----------	-------	------------

← Customer address information →      ← Customer order information →

CF1 | CF2

Table name – Customer Info or Customer  
Rowkey – customer username

Grouping data – Customer Address and  
Customer order  
Column Family 1 = Address  
Column Family 2 = Order

  
**BIG OFFICE**  
SUPPLY CO.

© 2016 MapR Technologies L2-44

1. To simplify things, we may call our table **Customer Info**
2. Choosing which information should be the row key is very important when designing the HBase schema. In this example, we will pick the customer username.
3. Looking at the sample data, there seems to be personal information relating to the customer such as address and information pertaining to their order. We will likely want to retrieve just the order information to fulfill the order, and retrieve the address information separately when it is ready to ship. Therefore, it makes sense to create 2 column families, one for the address and one for the order.
4. You can add the appropriate columns to the column families.

## 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

Using the create statement creates the table. Here we are specifying where we want to create the table and that it should be called **Customer**. We are also defining two column families, **Address** and **Order**.

In the first **put** statement, we are specifying the row key **smithj**, adding the value **Central Dr** for the **street** column in the **Address** column family.

In the next **put** statement, we are specifying the row key **smithj** adding the value **2/2/2015** for the **OrderDate** column in the **Order** column family.

The third put statement here is adding the value Columbus to the **city** column in the **Address** column family.

## Create HBase Table – Using HBase Shell

Minimize disk seek

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

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

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

The **get** statement will retrieve the latest version in all the columns in all the column families for **smithj**.

The **scan** statement retrieves the entire table in this case.

Use **describe** to return the description of the table.

## Knowledge Check



## Knowledge Check



What are some of the HBase shell operations to add/delete data?

- A. Put
- B. Add
- C. Delete
- D. Get
- E. Scan

## Knowledge Check



What are some of the HBase shell operations to add/delete data?

- A. Put
- B. Add
- C. Delete
- D. Get
- E. Scan

## Knowledge Check



**“Get” accesses data from one row. “Scan”**

- A. Accesses data from a specific row
- B. Accesses data from a range of rows
- C. Accesses data from one column family

## Knowledge Check



**“Get” accesses data from one row. “Scan”**

- A. Accesses data from a specific row
- B. Accesses data from a range of rows**
- C. Accesses data from one column family

## Refer to Lab Guide Exercise 2.4a and 2.4b



Open your lab guide, and complete exercises 2.4a and 2.4b.



## Next Steps

**DEV 320 – Apache HBase Model and Architecture**

Lesson 3: Apache HBase Architecture



## **DEV 320 – Apache HBase Data Model and Architecture**

Lesson 3: Apache HBase Architecture

Winter 2017, v5.1

Welcome to DEV 320 – Apache HBase Data Model and Architecture, Lesson 3: Apache HBase Architecture.

## Learning Goals



## Learning Goals



- 3.1 Identify components of an HBase cluster
- 3.2 Describe how HBase components work together
- 3.3 Describe how regions work and their benefits
- 3.4 Define the function of minor and major compactions
- 3.5 Describe RegionServer splits
- 3.6 Describe how HBase handles fault tolerance
- 3.7 Differentiate MapR-DB from HBase

Lesson 3 introduces HBase architectural components. We will see how these components work together and how data is stored in HBase. We will also differentiate between HBase and MapR-DB.

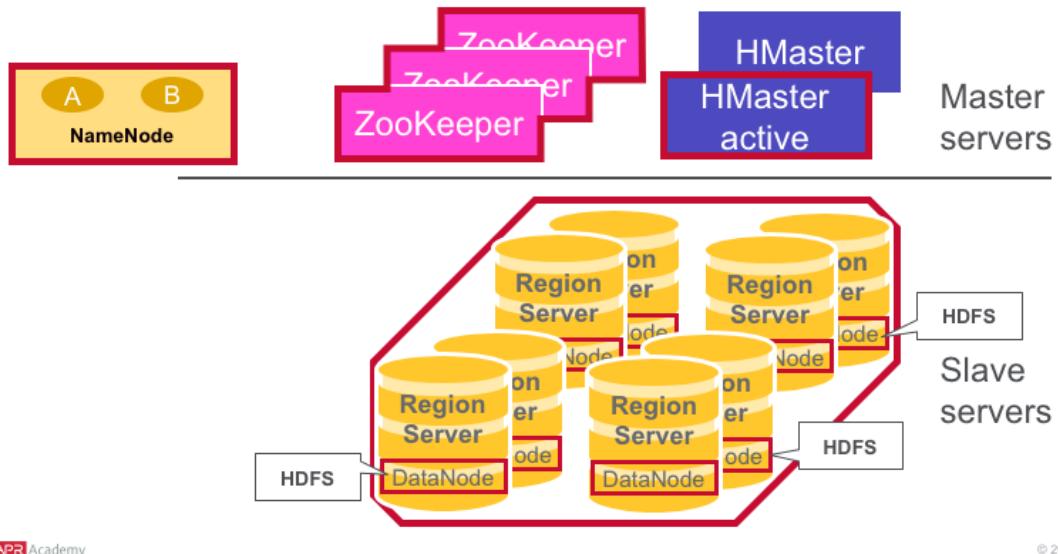
## Learning Goals



- 3.1 Identify components of an HBase cluster**
- 3.2 Describe how HBase components work together
- 3.3 Describe how regions work and their benefits
- 3.4 Define the function of minor and major compactions
- 3.5 Describe RegionServer splits
- 3.6 Describe how HBase handles fault tolerance
- 3.7 Differentiate MapR-DB from HBase

In this section, we will look at the architectural components of an HBase cluster.

## HBase Architectural Components



Physically, HBase is composed of 3 types of servers in a master slave type of architecture.

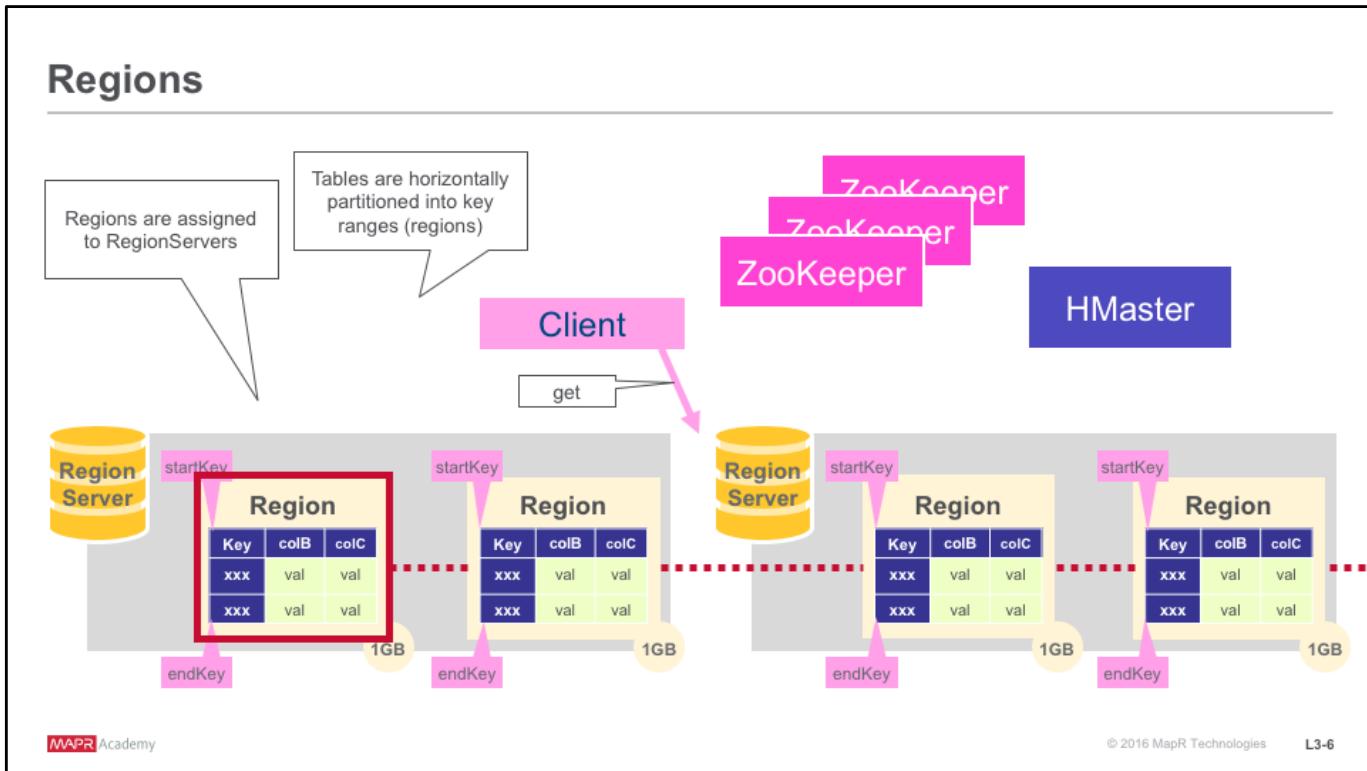
RegionServers serve data for reads and writes. When accessing data, clients communicate with HBase RegionServers directly.

Region assignment and DDL operations, create and delete tables, are handled by the HBase Master process.

ZooKeeper, which is a Hadoop service, maintains the live cluster state.

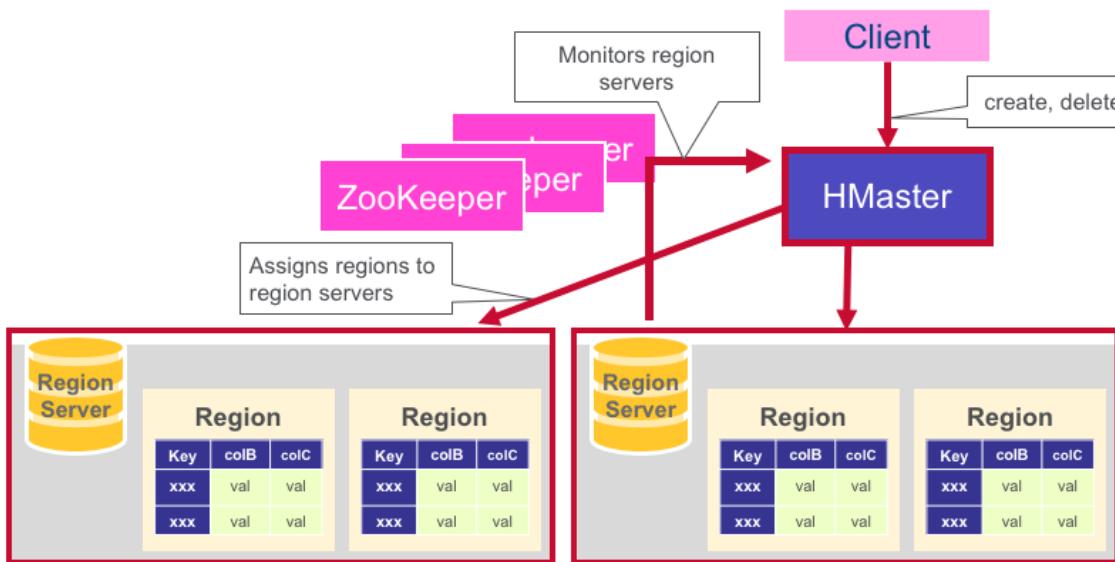
Hadoop data nodes store the data that the RegionServer is managing. All HBase data is stored in HDFS files. RegionServers are collocated with the HDFS data nodes, which enables data locality for the data served by the RegionServers. Data locality means putting the data close to where it is needed, to reduce the amount of time spent in data transfer.

The NameNode is for HDFS, The NameNode maintains metadata information for all the physical data blocks that comprise the files.



HBase tables are divided horizontally by row key range into regions. A region contains all rows in the table between the start key and end key. Regions are assigned to specific nodes in the cluster called RegionServers. RegionServers serve data for reads and writes. A RegionServer can serve about 1,000 regions.

## HBase HMaster

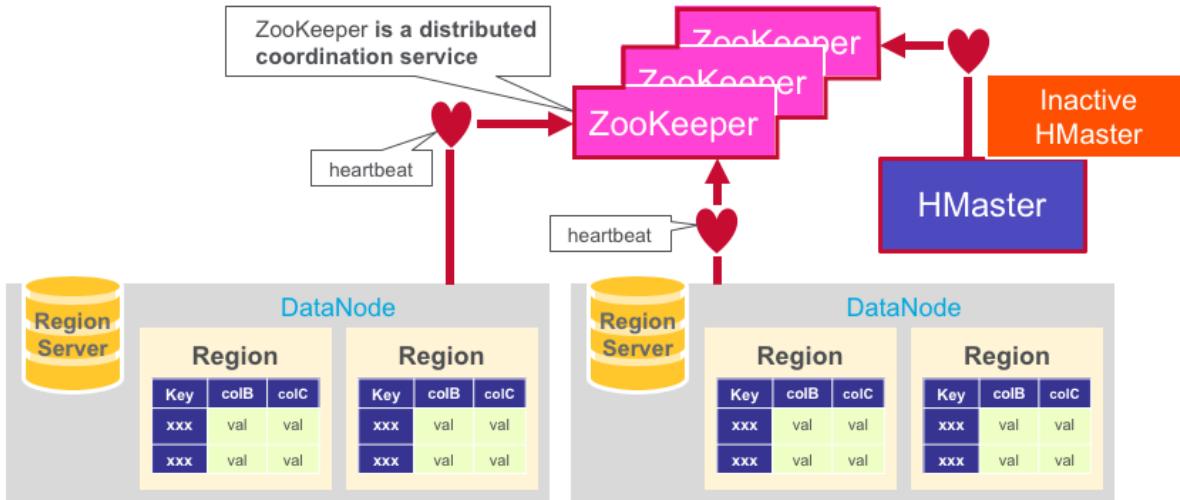


Region assignment and DDL operations are handled by the HBase HMaster.

An HMaster is responsible for:

- Coordinating the RegionServers:
  - assigning regions on startup
  - re-assigning regions for recovery or load balancing
  - monitoring all RegionServer instances in the cluster by listening for notifications from ZooKeeper
- Admin functions:
  - interface for creating, deleting, and updating tables

## Zookeeper The Coordinator



HBase uses ZooKeeper as a distributed coordination service to maintain server state in the cluster. ZooKeeper maintains which servers are alive and available, and provides server failure notification. Zookeeper uses consensus to guarantee common shared state. Note that there should be three or five machines for consensus.

## Knowledge Check





## Knowledge Check

Drag and drop the terms from the top of the page to the correct definition answer box (it will have not empty box in the end)

RegionServer

HBase HMaster

ZooKeeper

handles region assignment

ANSWER

maintains cluster state

ANSWER

serves data for read and writes

ANSWER



## Knowledge Check

Drag and drop the terms from the top of the page to the correct definition answer box (it will have not empty box in the end)

RegionServer

HBase HMaster

ZooKeeper

handles region assignment

ANSWER

maintains cluster state

ANSWER

serves data for read and writes

ANSWER

## Learning Goals



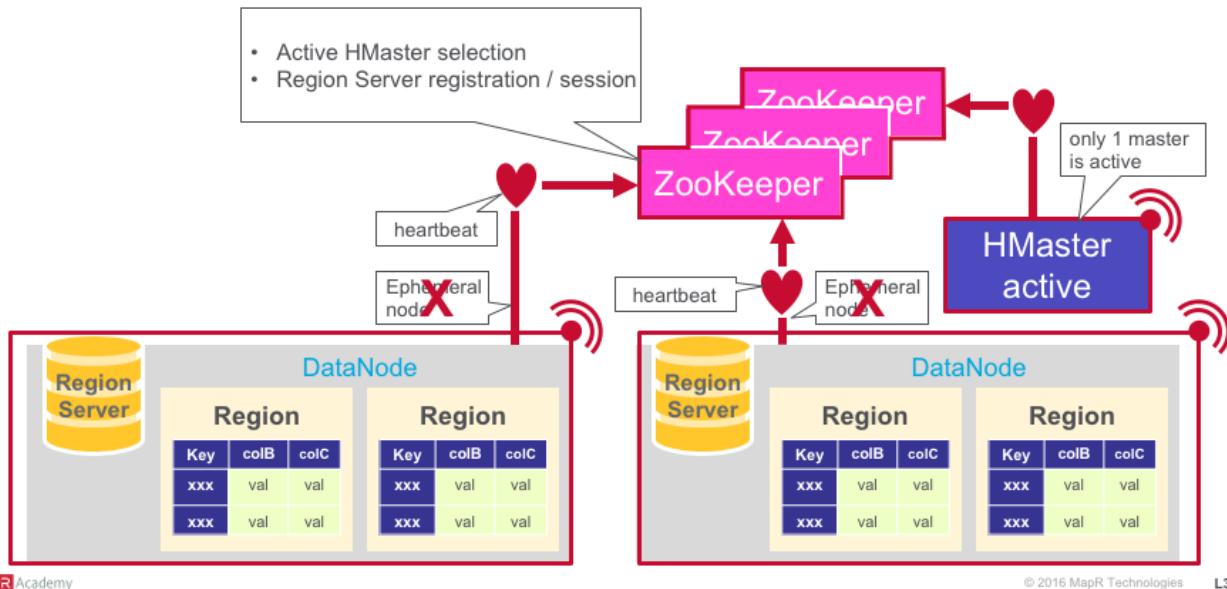
## Learning Goals



- 3.1 Identify components of an HBase cluster
- 3.2 Describe how HBase components work together**
- 3.3 Describe how regions work and their benefits
- 3.4 Define the function of minor and major compactions
- 3.5 Describe RegionServer splits
- 3.6 Describe how HBase handles fault tolerance
- 3.7 Differentiate MapR-DB from HBase

Now that we have taken a look at the main architectural components, let us see how they work together.

## How the Components Work Together



ZooKeeper coordinates shared state information for members of distributed systems.

RegionServers and the active HMaster connect to ZooKeeper. The ZooKeeper maintains ephemeral nodes for active sessions via heartbeats. Each RegionServer creates an ephemeral node.

The HMaster monitors these nodes to discover available RegionServers, and for server failures.

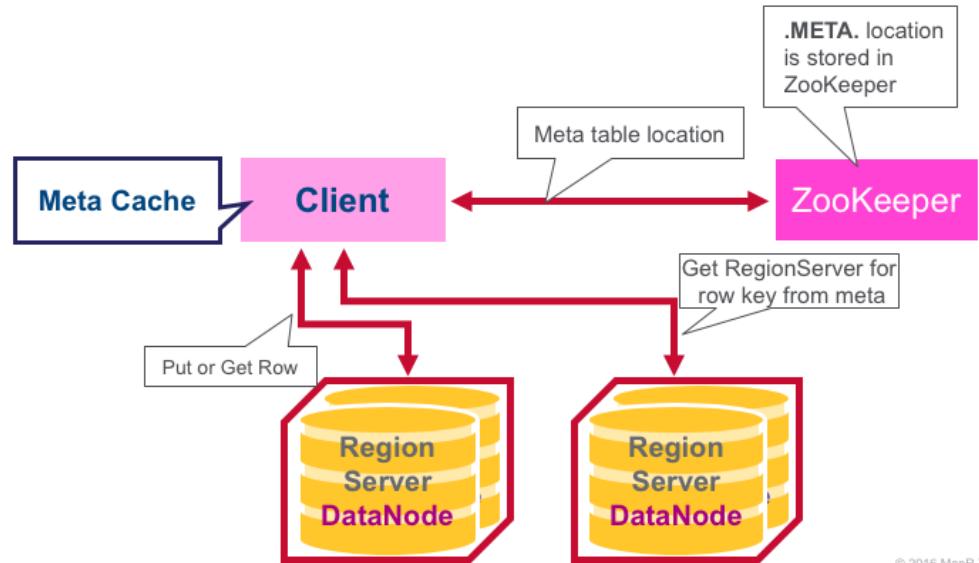
HMasters vie to create an ephemeral node. ZooKeeper determines the first one and uses it to make sure that only one master is active.

The active HMaster sends heartbeats to ZooKeeper. The inactive HMaster listens for notifications of the active HMaster failure.

If a region server or the active HMaster fails to send a heartbeat, the session is expired and the corresponding ephemeral node is deleted.

Listeners for updates will be notified of the deleted nodes. The active HMaster listens for RegionServers, and will recover RegionServers on failure. The inactive HMaster listens for active HMaster failure and if an active HMaster fails, the inactive HMaster becomes active.

## HBase First Read or Write



MAPR Academy

© 2016 MapR Technologies

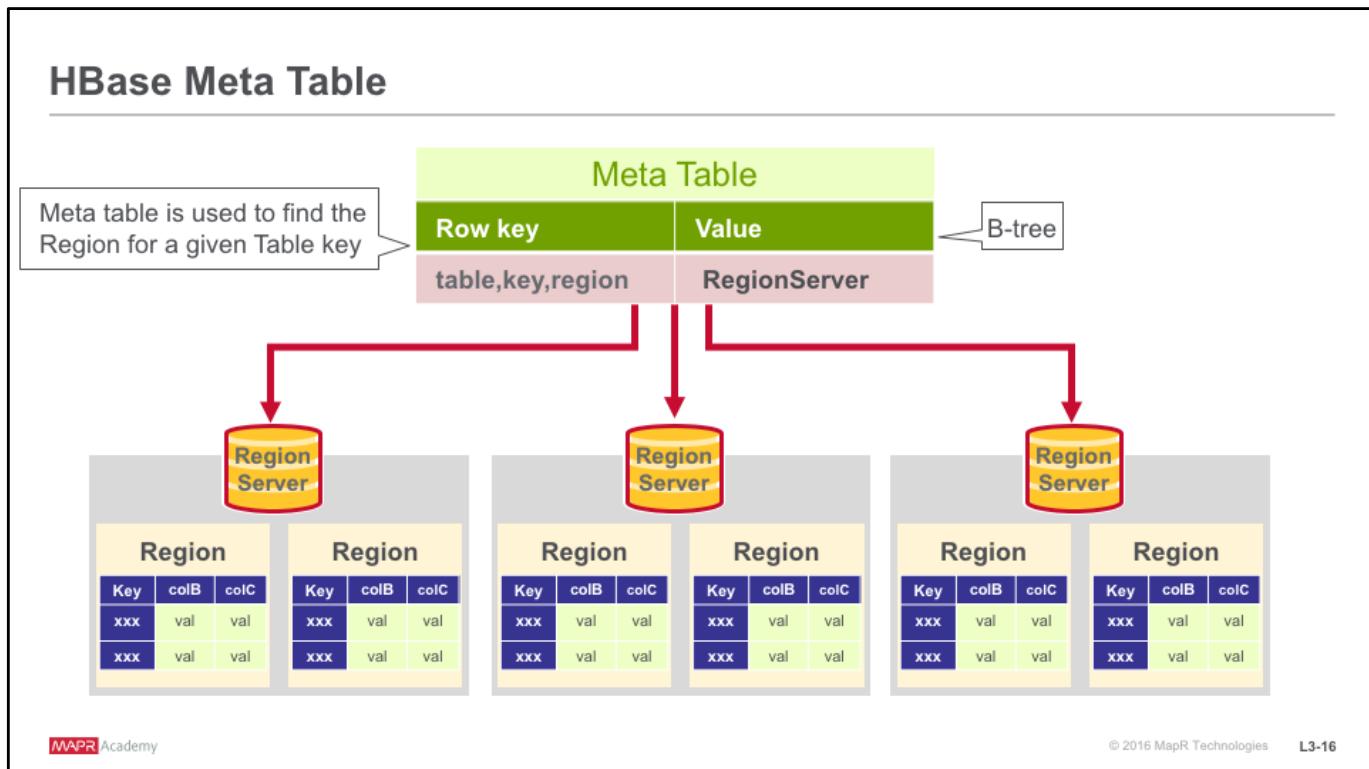
L3-15

There is a special HBase Catalog table called the Meta table which holds the location of the regions in the cluster. ZooKeeper stores the location of the Meta table.

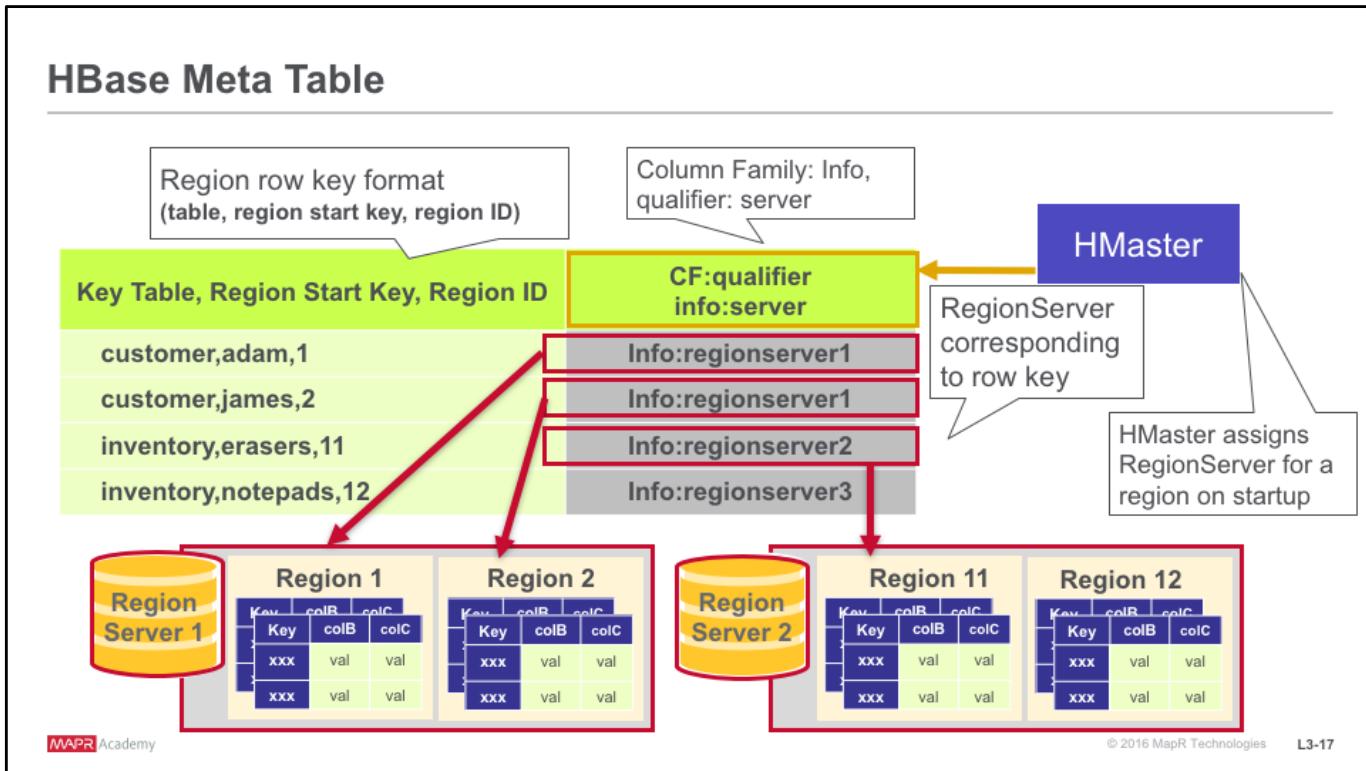
The first time a client reads or writes to HBase:

1. The client gets the RegionServer that hosts the Meta table from ZooKeeper
2. The client will query the .META. server to get the RegionServer corresponding to the row key it wants to access. The client caches this information along with the Meta table location.
3. And finally, the client will get the row from the corresponding RegionServer

For future reads, the client uses the cache to retrieve the Meta location and previously read row keys. Over time it does not need to query the Meta table, unless there is a miss because a region has moved, then it will re-query ZooKeeper, and update the cache.



This Meta table is an HBase table that keeps a list of all regions in the system. The Meta Table is like a B-tree. The Key is the table name, the region start key, and the region ID. The Values are the RegionServers. Given a key, the Meta table will define which RegionServer to go to.



The HBase Meta table structure is as follows:

The row key contains the table name, region start key, and region ID

There is one column family named Info, and one column named server

The cell value contains the RegionServer corresponding to the row key

### Startup Sequencing

When HBase starts up, regions are assigned. The HMaster looks at the existing region assignments in the Meta table, and if needed the Meta table is updated with RegionServer assignments.

1. the location of hbase:meta is looked up in ZooKeeper
2. hbase:meta is updated with server values

## Knowledge Check



## Knowledge Check



Arrange the following statements in the correct sequence.

The first time a client reads/writes to HBase, how is the Region Server to which to write or read determined?

1. Gets Meta table from a region server and caches the meta table
2. Gets Row from Region
3. Gets the region server for row from meta
4. Gets Meta table location from zookeeper



## Knowledge Check

Arrange the following statements in the correct sequence.

The first time a client reads/writes to HBase, how is the Region Server to which to write or read determined?

1. Gets Meta table from a region server and caches the meta table
2. Gets Row from Region
3. Gets the region server for row from meta
4. Gets Meta table location from zookeeper

**Sequence = 4, 1, 3, 2**

## Knowledge Check



Which of the following statements is true of the HBase Meta table?

- A. It is an HBase table
- B. It keeps a list of all the regions in the system
- C. Given a key, it returns the region to which to go
- D. All of the above

## Knowledge Check



Which of the following statements is true of the HBase Meta table?

- A. It is an HBase table
- B. It keeps a list of all the regions in the system
- C. Given a key, it returns the region to which to go
- D. **All of the above**

## Learning Goals



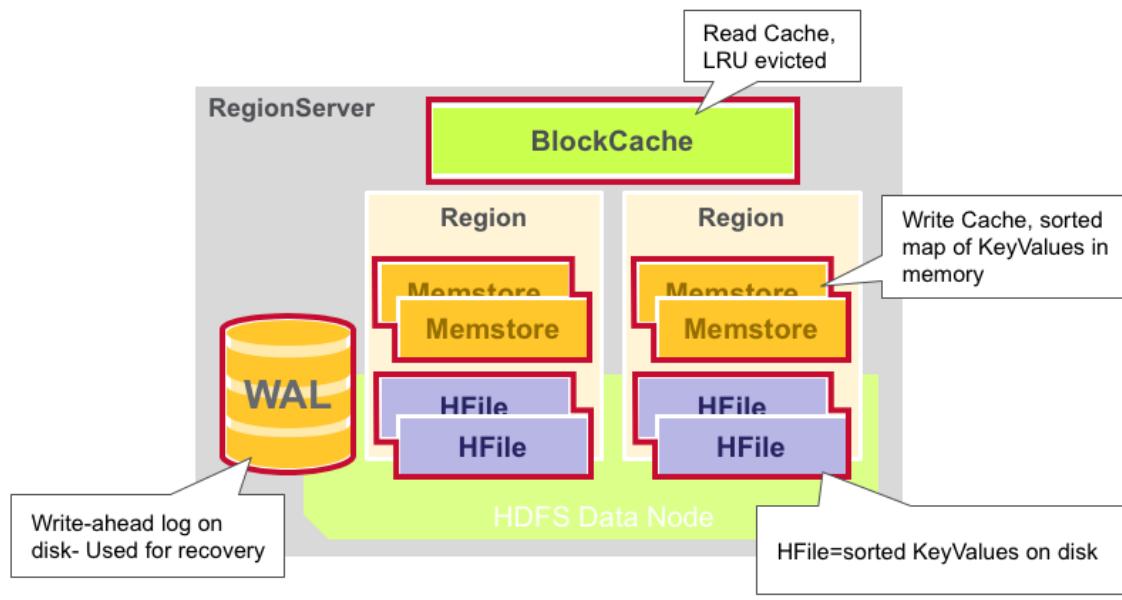
## Learning Goals



- 3.1 Identify components of an HBase cluster
- 3.2 Describe how HBase components work together
- 3.3 Describe how regions work and their benefits**
- 3.4 Define the function of minor and major compactions
- 3.5 Describe RegionServer splits
- 3.6 Describe how HBase handles fault tolerance
- 3.7 Differentiate MapR-DB from HBase

In this section, we will take a look at regions, how they work and their benefits.

## RegionServer Components

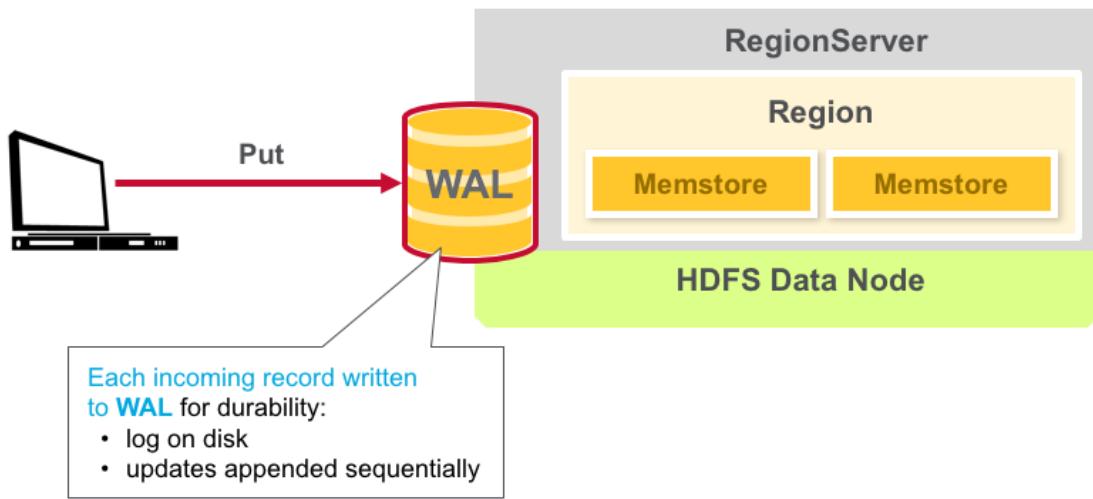


MAPR Academy © 2016 MapR Technologies L3-25

A RegionServer runs on an HDFS DataNode, and has the following components:

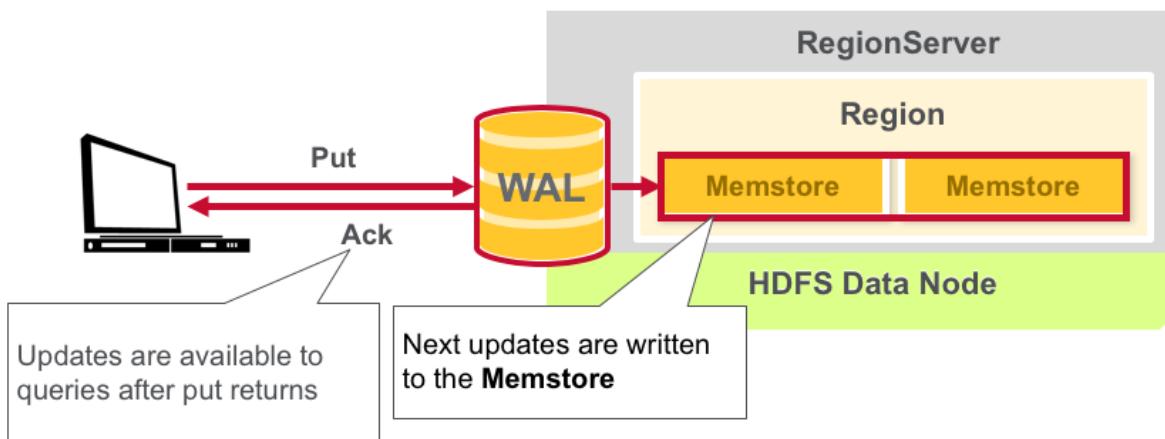
- The Write-Ahead Log, or WAL, is a file on the distributed file system. The WAL is used to store new data that hasn't yet been persisted to permanent storage. It is used for recovery in the case of failure.
- BlockCache is the read cache, it stores frequently read data in memory. Least Recently Used data is evicted when the BlockCache is full.
- MemStore is the write cache, it stores new data which has not yet been written to disk. It is sorted before writing to disk. There is one memstore per column family, per region.
- HFiles store the rows as sorted key-values on disk.

## HBase Write Steps

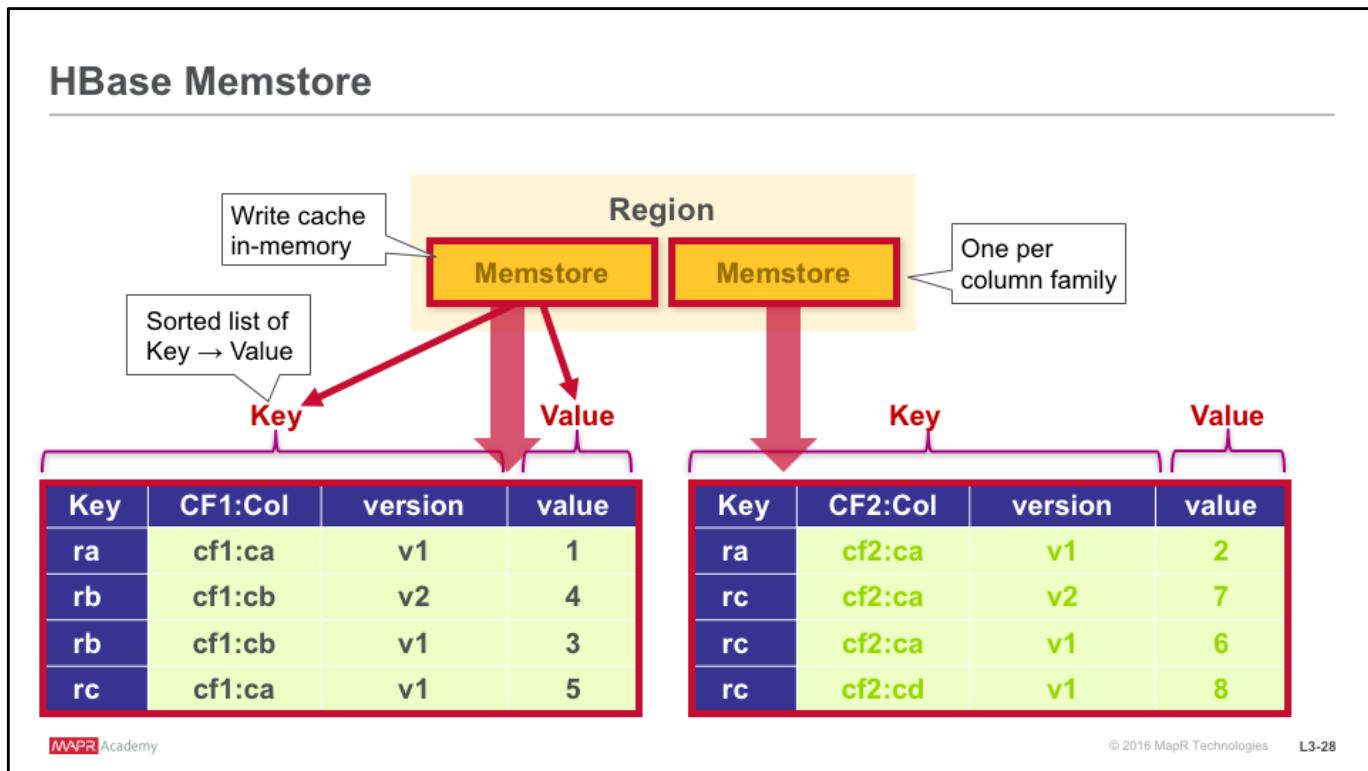


When the client issues an Put request the first step is to write the data to the write-ahead log. Edits are appended to the end of the WAL file that is stored on disk. The WAL is used to recover not-yet-persisted data in case a server crashes. WAL a file in HDFS, so is replicated 3 times across the cluster.

## HBase Write Steps



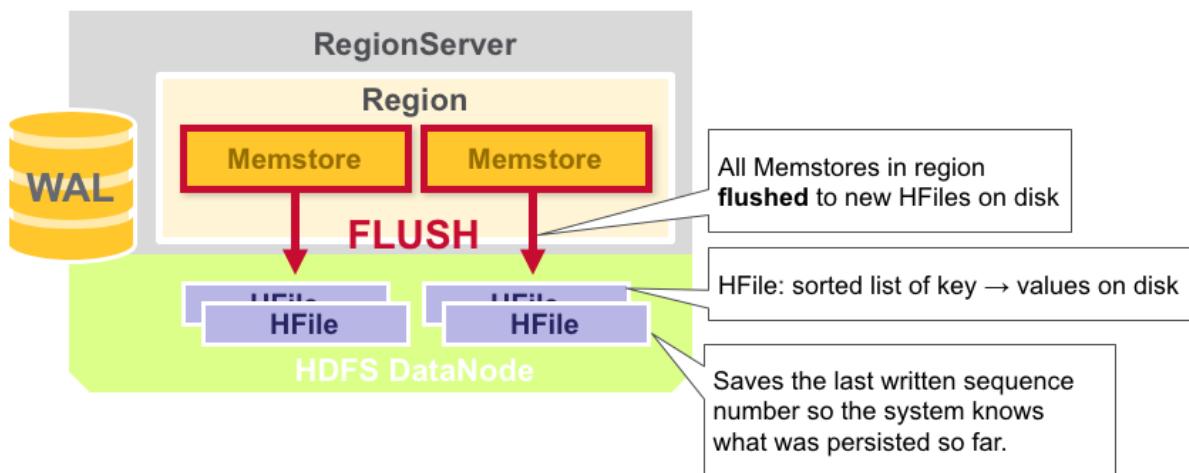
Once the data is written to the WAL, it is placed in the MemStore. Then the put request acknowledgement returns to the client.



The Memstore stores updates in memory as sorted key-values, the same as it will be stored in a HFile.

There is one Memstore per column family, and the updates are sorted per column family.

## HBase Region Flush



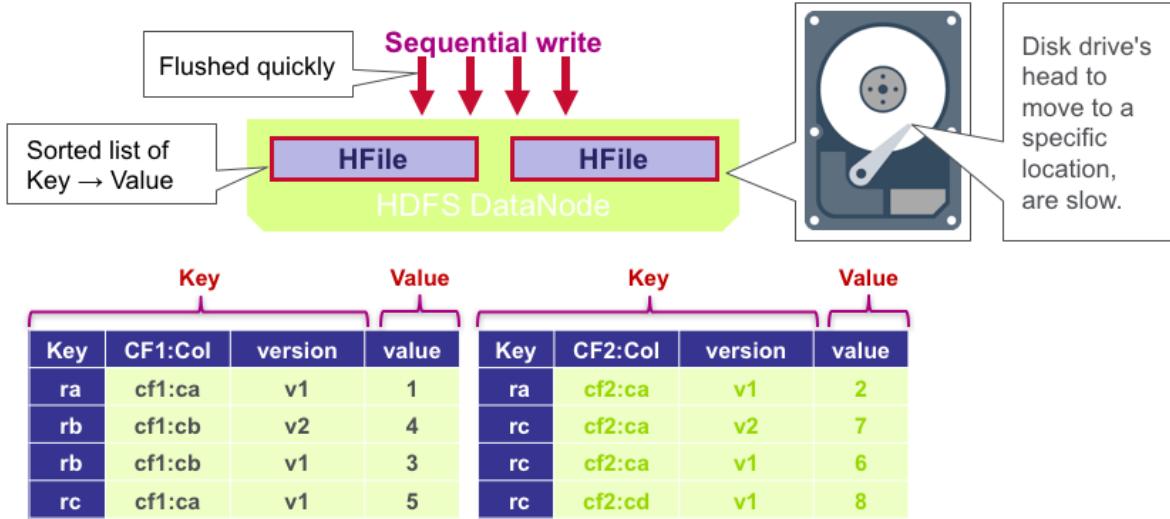
When the Memstore accumulates enough data, the entire sorted set is written to a new HFile in HDFS.

HBase uses multiple HFiles per column family, which contain the actual cells, or key-value instances. These files are created over time, as key-value edits sorted in the Memstores are flushed as files to disk.

Note this is one reason why there is a limit to the number of column families in HBase. There is one Memstore per column family, when one is full they all flush. It also saves the last written sequence number so the system knows what has persisted so far.

The highest sequence number is stored as a meta field in each HFile, to reflect where persisting has ended and where to continue. On region startup, the sequence number is read and the highest is used as the sequence number for new edits.

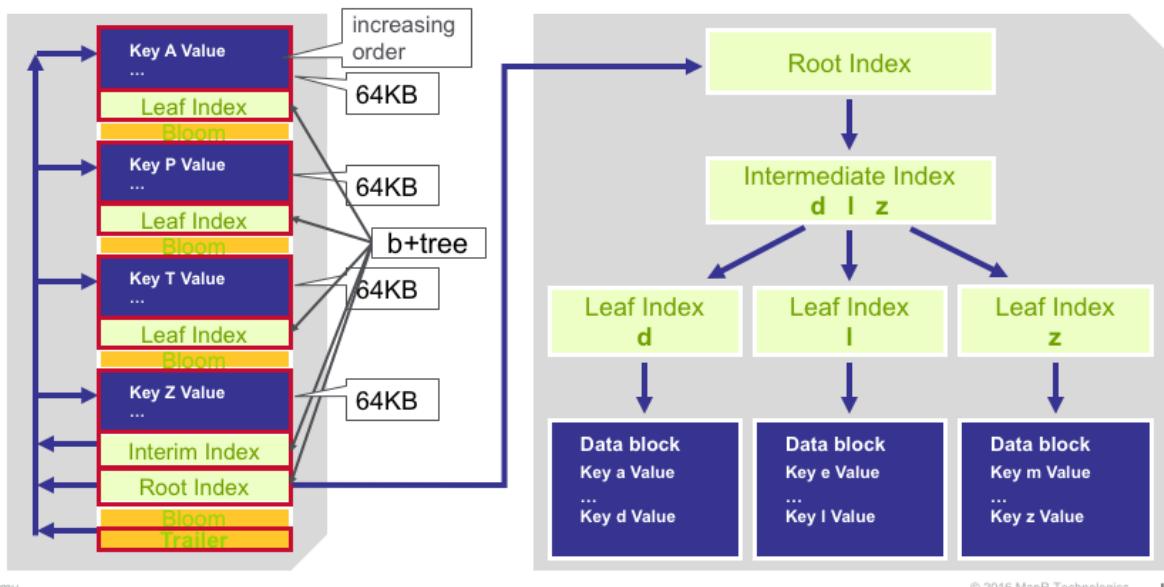
## HBase HFile



Data is stored in an HFile, which contains sorted Key-values.

When the Memstore accumulates enough data, the entire sorted Key-value set is written to a new HFile in HDFS. This is a sequential write. It is very fast as it avoids moving the disk drive head.

## HBase HFile Structure



MAPR Academy

© 2016 MapR Technologies

L3-31

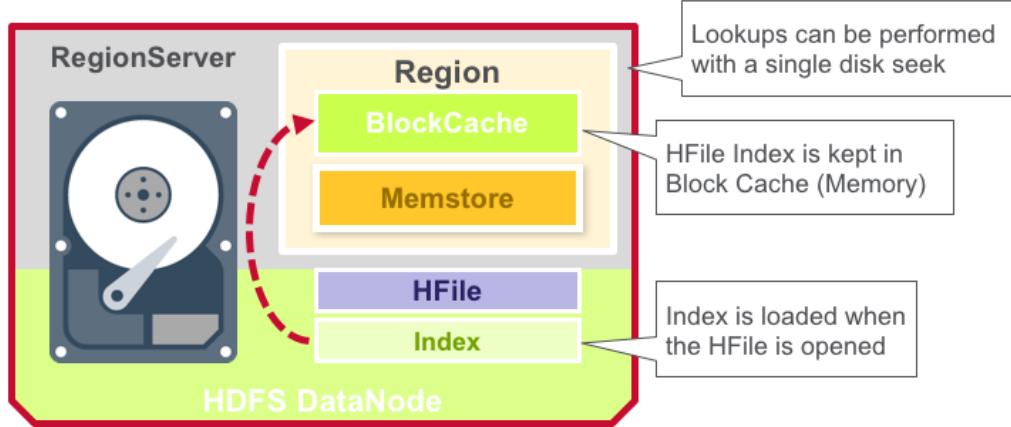
An HFile contains a multi-layered index which allows HBase to seek to the data, without having to read the whole file.

The multi-level index is like a B+tree:

- Key-value pairs are stored in increasing order
- Indexes point by row key to the key value data in 64KB blocks
- Each block has its own leaf-index
- The last key of each block is put in the intermediate index
- The root index points to the intermediate index

The Trailer points to the meta blocks, and is written at the end of persisting the data to the file. The trailer also has information like bloom filters and time range info. Bloom filters help to skip files that do not contain a certain row key. The time range info is useful for skipping the file if it is not in the time range the read is looking for.

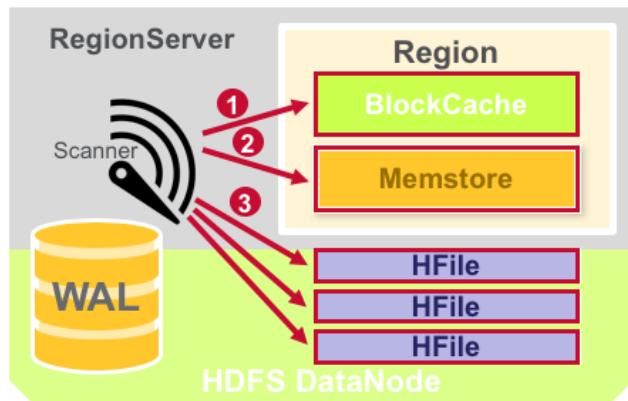
## HFile Index



The Index is loaded when the HFile is opened and kept in memory. This allows lookups to be performed with a single disk seek.

## HBase Read Merge

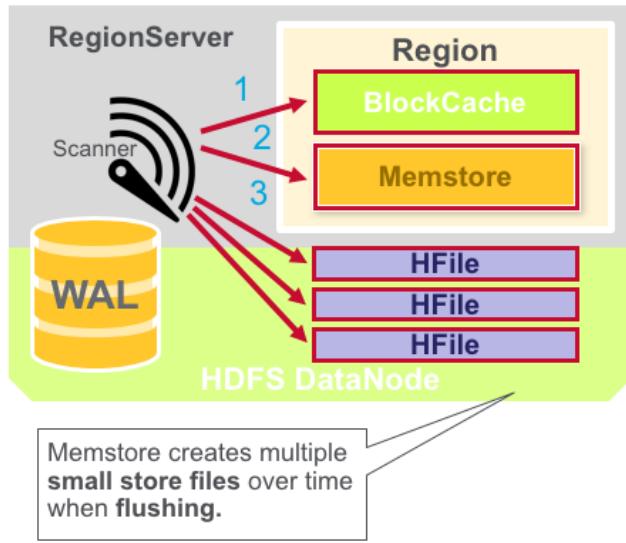
- 1 First the scanner looks for the row key-values in the BlockCache
- 2 Next the scanner looks in the Memstore
- 3 If all row cells not in Memstore or BlockCache, look in HFiles



We have seen that the key-value cells corresponding to one row can be in multiple places. Row cells already persisted are in HFiles, recently updated cells are in the Memstore, and recently read cells are in the BlockCache. When you read a row, key-values are merged from the BlockCache, Memstore, and HFiles in the following steps:

1. First, the scanner looks for the row cells in the BlockCache. Recently read key-values are cached here, and the least recently used are evicted when memory is needed.
2. Next, the scanner looks in the Memstore, the write cache in memory containing the most recent writes.
3. If the scanner does not find all of the row cells in the Memstore and Block Cache, then HBase will use the BlockCache indexes and Bloom filters to load HFiles into memory, which may contain the target row cells.

## HBase Read Merge

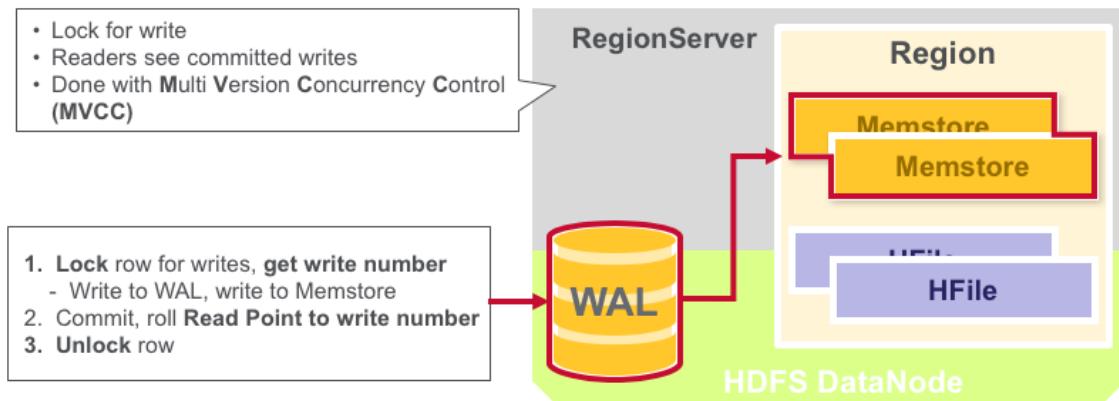


### Read Amplification

- If row cells not in block cache or Memstore → read from HFile on disk
- Many HFiles/Memstore
- Implies multiple files to be read
- Affects performance

Read amplification occurs when there are many HFiles written by Memstore flushes. This causes multiple files to be examined for a read, which can affect the performance.

## Consistency Concurrency Control



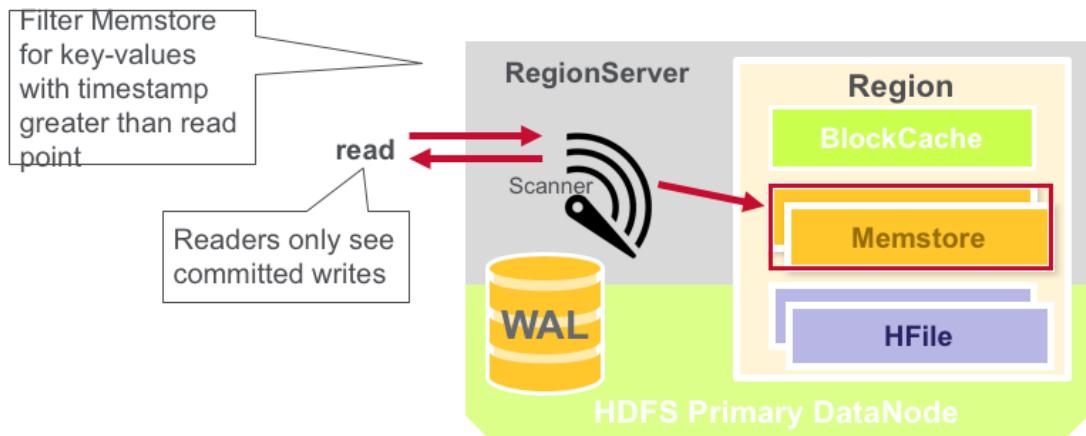
HBase ensures consistency by having a single RegionServer responsible for all reads and writes for a region of data. The region server's WAL and files are stored on an HDFS node, which replicates to secondary and tertiary HDFS nodes for availability in case of node failure. HBase provides strong consistency, meaning that when a write returns, all readers will see the same value.

HBase also has row atomicity using Multi Version Concurrency Control or (MVCC).

The high level flow of a MVCC write transaction in HBase is described here:

1. Lock the row before a write, to guard against concurrent writes
2. Retrieve the current write number
3. Update the WAL and Memstore, using the acquired write number to tag the key-values
4. Commit the transaction, then roll the read point forward to the write number
5. Unlock the row

## Consistency Concurrency Control



Reads do not lock a row, like a write does. Only those rows that are already committed are read.

Here is the description of the high level flow of MVCC reads:

- 1 Open the scanner
- 2 Get the current read point
- 3 Filter all scanned key-values with Memstore timestamp > the read point
- 4 Close the scanner (this is initiated by the client)

## Knowledge Check





## Knowledge Check

Match the terms from the top of the page to the correct definition answer box

WAL

BlockCache

Memstore

HFile

stores rows as sorted key-values on disk

HFile

stores new data not yet written to disk

Memstore

stores frequently read data in memory

BlockCache

file to store new data not yet persisted to permanent storage

WAL



## Knowledge Check

Match the terms from the top of the page to the correct definition answer box

WAL

BlockCache

Memstore

HFile

stores rows as sorted key-values on disk

HFile

stores new data not yet written to disk

Memstore

stores frequently read data in memory

BlockCache

file to store new data not yet persisted to permanent storage

WAL

- 1-d
- 2-c
- 3-b
- 4-a

## Knowledge Check



Arrange the following statements in the correct sequence.

Describe the write path. When a client issues a put request:

- A. Memstore is full, flushed to HFile
- B. The data is placed and sorted in the Memstore
- C. The data is written to the Write Ahead Log.
- D. Acknowledgement sent to client



## Knowledge Check

Arrange the following statements in the correct sequence.

Describe the write path. When a client issues a put request:

- A. Memstore is full, flushed to HFile
- B. The data is placed and sorted in the Memstore
- C. The data is written to the Write Ahead Log.
- D. Acknowledgement sent to client

Sequence = C->B->D->A

## Knowledge Check



### What happens when one Memstore is full?

1. All Memstores in the region are flushed to new HFiles on disk
2. The last written sequence number is stored as a meta field

## Knowledge Check



What happens when one Memstore is full?

1. All Memstores in the region are flushed to new HFiles on disk
2. The last written sequence number is stored as a meta field



## Knowledge Check

Arrange the following statements in the correct sequence.

Describe the read steps in HBase:

- A. Look in HFiles
- B. Look for the row key-values in the BlockCache
- C. Filter Memstore key-values with timestamp greater than read point



## Knowledge Check

Arrange the following statements in the correct sequence.

Describe the read steps in HBase:

- A. Look in HFiles
- B. Look for the row key-values in the BlockCache
- C. Filter Memstore key-values with timestamp greater than read point

Sequence = B->C->A

## Learning Goals





## Learning Goals

- 3.1 Identify components of an HBase cluster
- 3.2 Describe how HBase components work together
- 3.3 Describe how regions work and their benefits
- 3.4 Define the function of minor and major compactions**
- 3.5 Describe RegionServer splits
- 3.6 Describe how HBase handles fault tolerance
- 3.7 Differentiate MapR-DB from HBase

We have seen how the architectural components work together and also how read and writes are done in HBase. In this next section, we will look at major and minor compactions.

## HBase Compaction

- What is Compaction?
  - HBase mechanism to merge HFiles.
- 2 types of HBase Compaction
  - Minor Compaction
  - Major Compaction



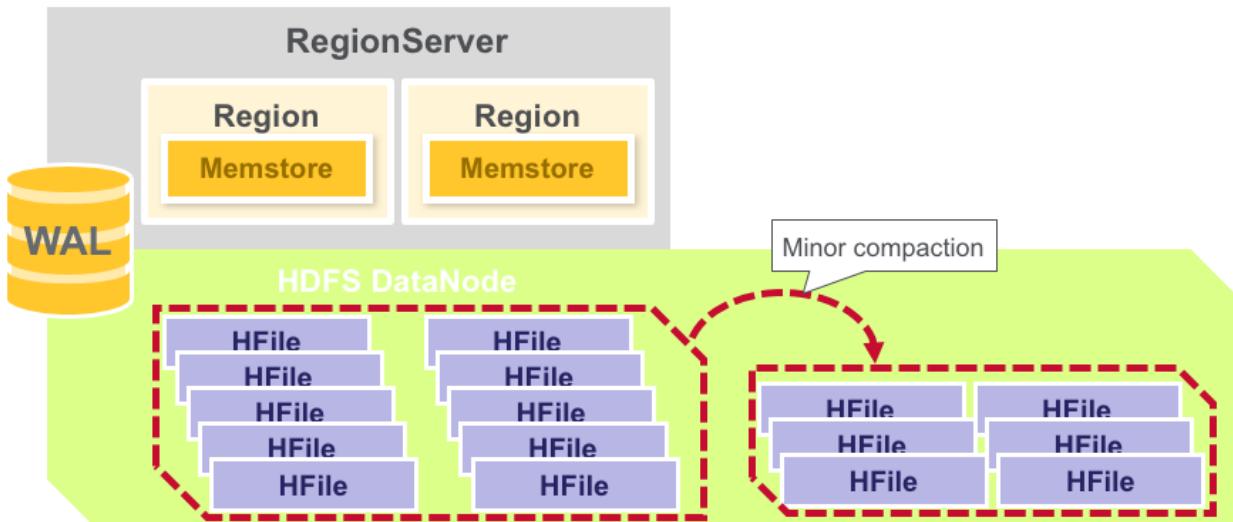
MAPR Academy

All files in HDFS are write once, and once written they are immutable. Therefore, each Memstore flush to disk is written to a new HFile.

Flushing Memstores to disk causes more and more HFiles to be created. As data increases, there may be many HFiles on HDFS. Multiple files may have to be examined for each new read, which is not good for read performance. To improve read performance, HBase merges and compacts HFiles into larger new consolidated files.

The goal of compactions is to reduce read amplification.

## HBase Minor Compaction



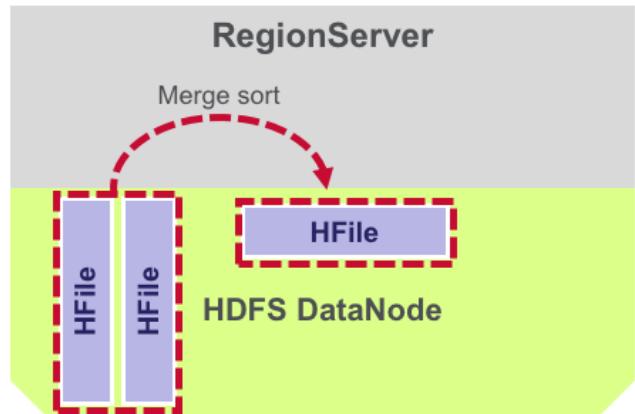
HBase will automatically pick some smaller HFiles and rewrite them into fewer bigger HFiles. This process is called minor compaction.

Minor compaction reduces the number of storage files by rewriting smaller files into fewer but larger ones, performing a merge sort.

## HBase Minor Compaction

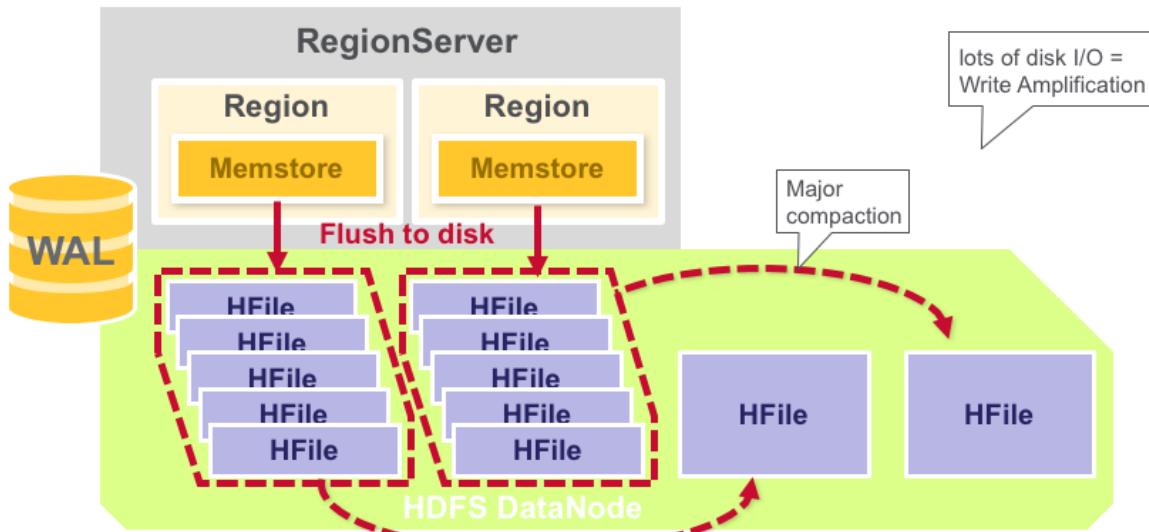
Minor compaction:

- Background process
- Sequential read
  - Read couple small files
  - Merge sort
- Sequential write
  - One bigger file
  - Delete smaller



To avoid reading too many files, there is a background thread that will detect when there are too many files. The system performs compaction by reading some small files, merge sorting them in memory, and writing the sorted key-values into a new, larger file. This involves sequential reads and writes, with sorting done in memory to avoid disk seeks.

## HBase Major Compaction



MAPR Academy

© 2016 MapR Technologies

L3-51

Major compaction merges and rewrites all the HFiles in a region to one HFile per Column family. During the process, deleted or expired cells are dropped. Major compaction improves read performance.

Since major compaction rewrites all of the files, lots of disk I/O and network traffic might occur during the process. This is called write amplification.

Major compactions can be scheduled to run automatically. Due to write amplification, major compactions are usually scheduled for weekends or evenings.

A major compaction also makes any remote data files local to the region server, such as those from a server failure or load balancing.

## Compaction Overview

Minor Compaction	Major Compaction
Take a few store files and combine them	Take all store files and merge them into one
	Also remove ttl, max versions, or records with tombstone markers

The above table outlines the difference between minor and major compaction.

## Knowledge Check



## Knowledge Check



1. Rewrites and merges files into fewer larger ones
2. Causes write amplification
3. Compacts all files into a single file
4. Removes records marked with tombstone markers

Major Compactions

Minor Compactions



## Knowledge Check

1. Rewrites and merges files into fewer larger ones
2. Causes write amplification
3. Compacts all files into a single file
4. Removes records marked with tombstone markers

Major Compactions

2,3,4

Minor Compactions

1

## Learning Goals



## Learning Goals



- 3.1 Identify components of an HBase cluster
- 3.2 Describe how HBase components work together
- 3.3 Describe how regions work and their benefits
- 3.4 Define the function of minor and major compactions
- 3.5 Describe RegionServer splits**
- 3.6 Describe how HBase handles fault tolerance
- 3.7 Differentiate MapR-DB from HBase

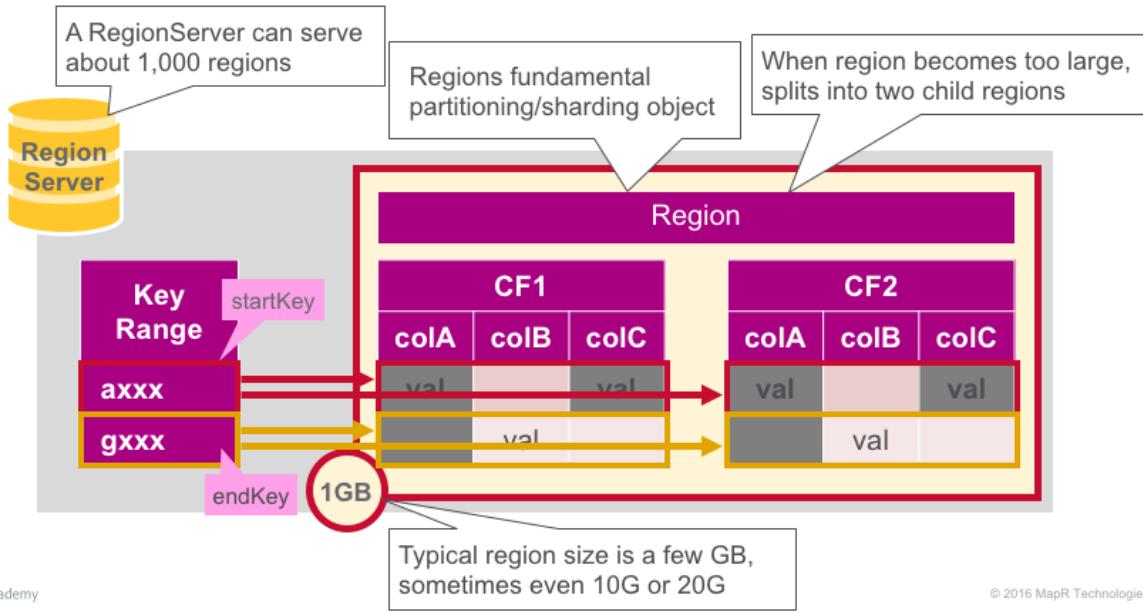
In this section, we will take a look at how a RegionServer splits regions as a table grows.

## Review





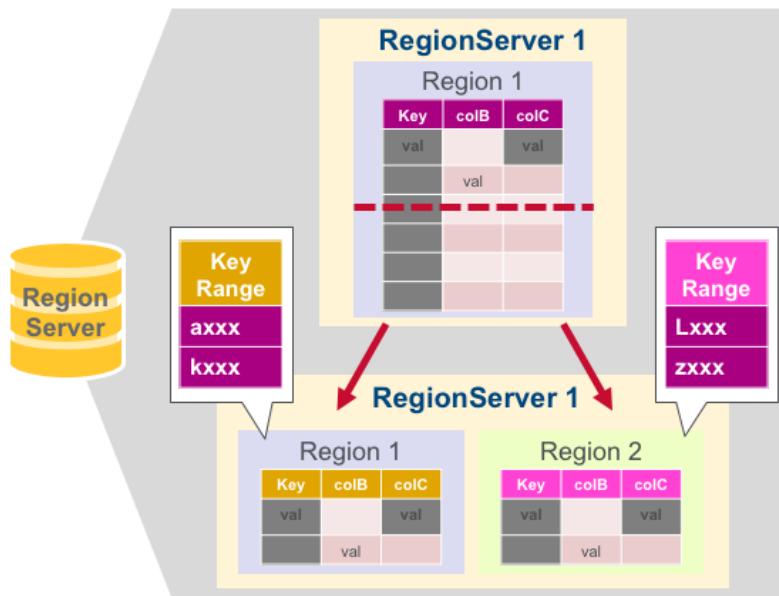
## Review



Let us do a quick review of regions.

- A table can be divided horizontally into one or more regions.
- A region contains a contiguous, sorted range of rows between a start key and an end Key.
- Each region is 1GB in size by default.
- A region of a table is served to the client by a RegionServer.
- A RegionServer can serve about 1,000 regions, which may belong to the same table or different tables.

## Region Split

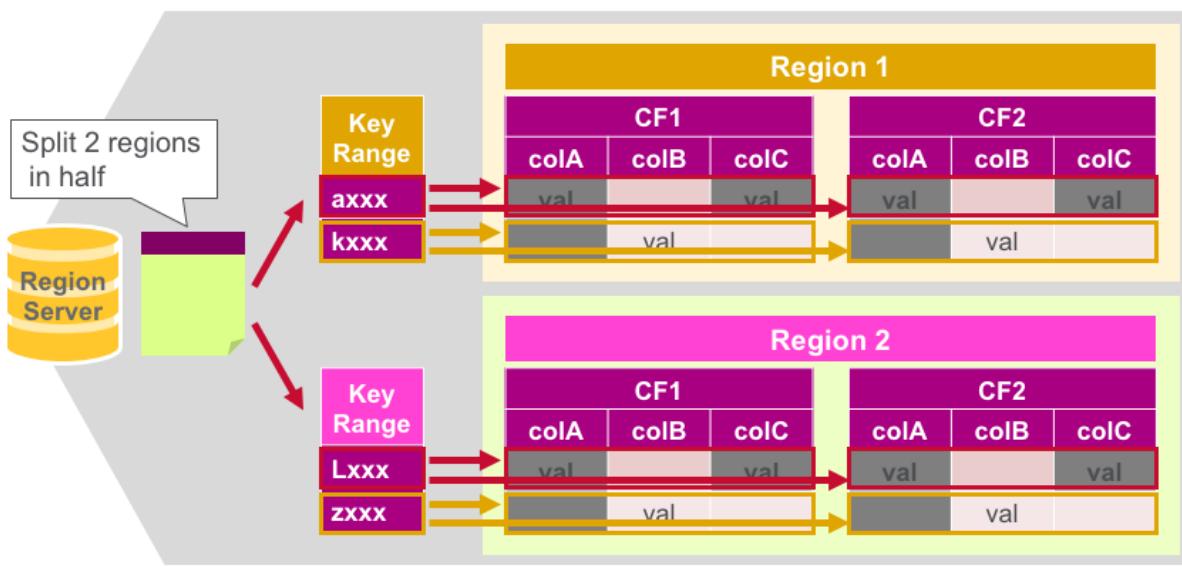


when region size >  
hbase.hregion.max.  
filesize → split

Initially there is one region per table. When a region grows too large, it splits into two child regions. Both child regions, representing one-half of the original region, are opened in parallel on the same RegionServer, then the split is reported to the HMaster.

For load balancing reasons the HMaster may schedule for new regions to be moved off to other servers. The Meta table is updated, and the RegionServer updates ZooKeeper.

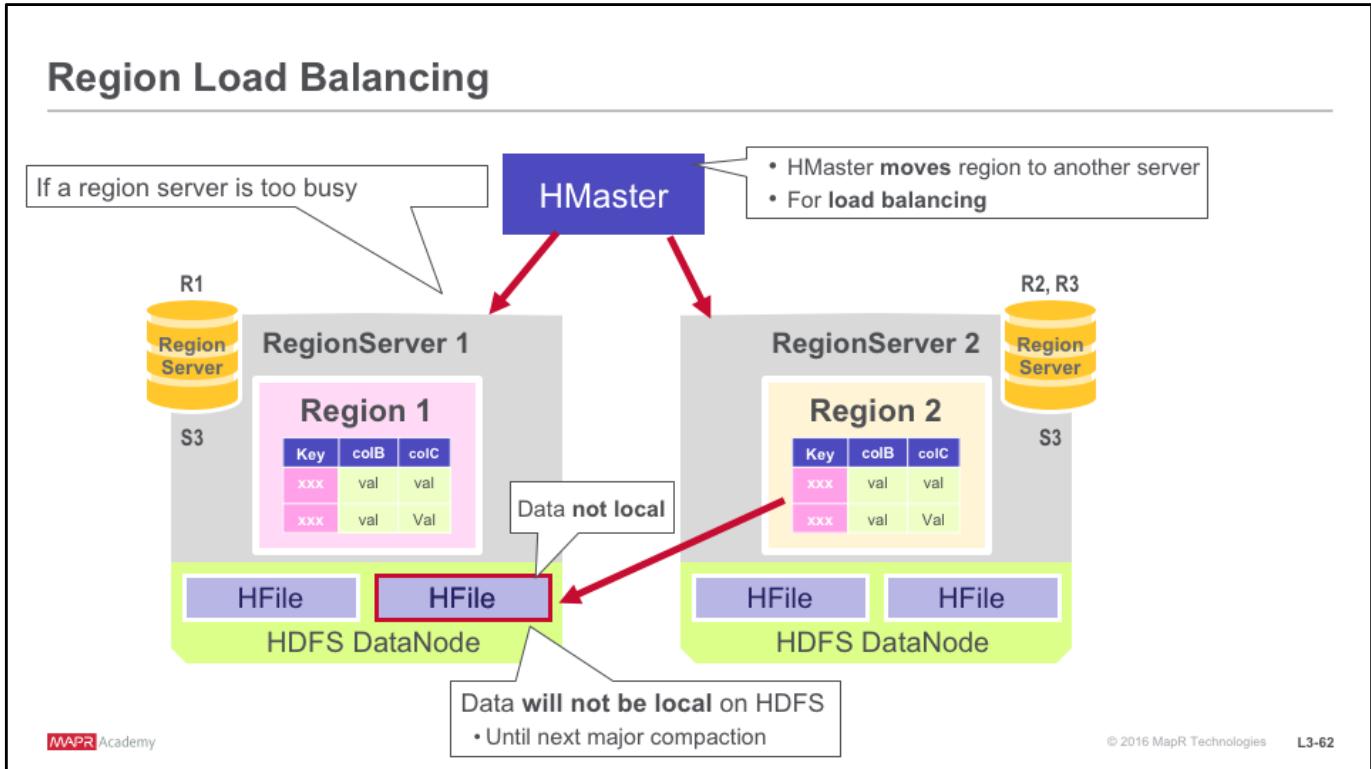
## Region Splits



Note that when a region splits, all the ColumnFamilies are split based on the row key range.

This is one reason for the recommended 3 ColumnFamily limit for HBase.

Also it is better to have a similar amount of data in different column families in a table, so that the columns will be balanced for scanning across regions.



Splitting happens initially on the same region server, but for load balancing reasons the HMaster may schedule for new regions to be moved off to other servers. This results in the new RegionServer serving data from a remote HDFS node until a major compaction moves the data files to the RegionServer's local node.

HBase data is local when it is written, but when a region is moved, for load balancing or recovery, it is not local until the next major compaction.

## Pre-Split Regions

### Pre-Splits

- Can manually split regions
- Advantages
  - Can help with load balancing at the start of a new cluster.
- Lots of planning required to pre-split regions.



MAPR Academy

Based on the target number of regions per RegionServer, one can pre-split the table at creation time. This is one way to ensure that the table starts out already distributed across many servers, which helps with load balancing at the start of a new cluster. While this has its advantages, pre-splitting requires careful planning. You need to consider how your table will grow based on the row key.

## Learning Goals



## Learning Goals



- 3.1 Identify components of an HBase cluster
- 3.2 Describe how HBase components work together
- 3.3 Describe how regions work and their benefits
- 3.4 Define the function of minor and major compactions
- 3.5 Describe RegionServer splits
- 3.6 Describe how HBase handles fault tolerance**
- 3.7 Differentiate MapR-DB from HBase

In this section, we will see how HBase handles fault tolerance.

## HDFS and Fault Tolerance

1. Data writes recorded in WAL
  2. Data written to Memstore
  3. When Memstore full, data written to disk in HFile
- Q.** What happens if there is a failure when data still in Memstore and not persisted?
- A.** HDFS data replication – data replicated 3 times  
WAL also replicated 3 times – But WAL not readable

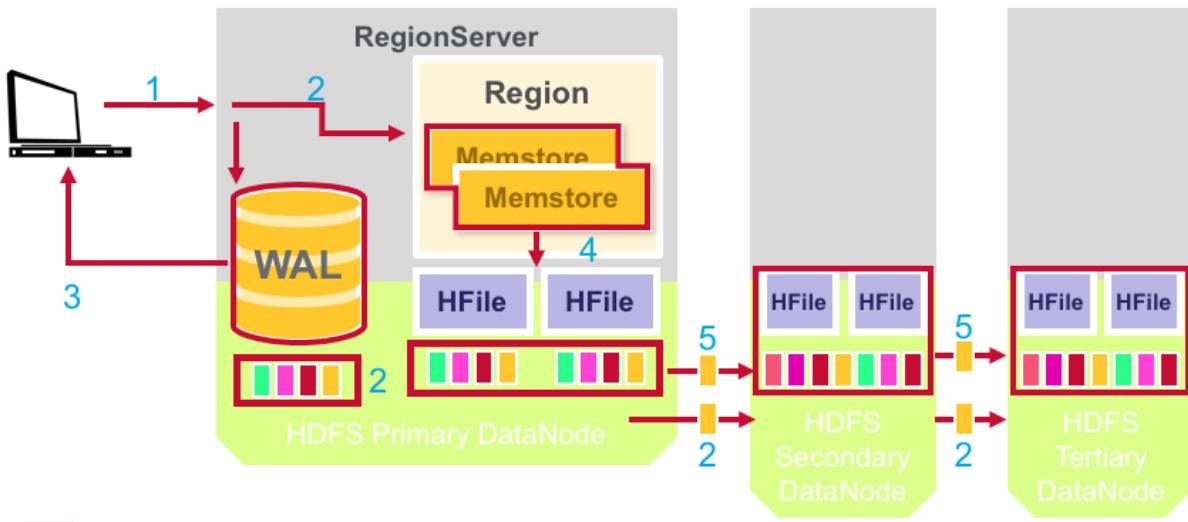
All writes and reads are to and from the primary node.

1. Data writes are first recorded in the WAL
2. The data itself is written to a Memstore.
3. When the Memstore is full, the data is persisted in the background to a disk in an HFile.

What happens if there is a failure when the data is still in memory and not persisted to an Hfile?

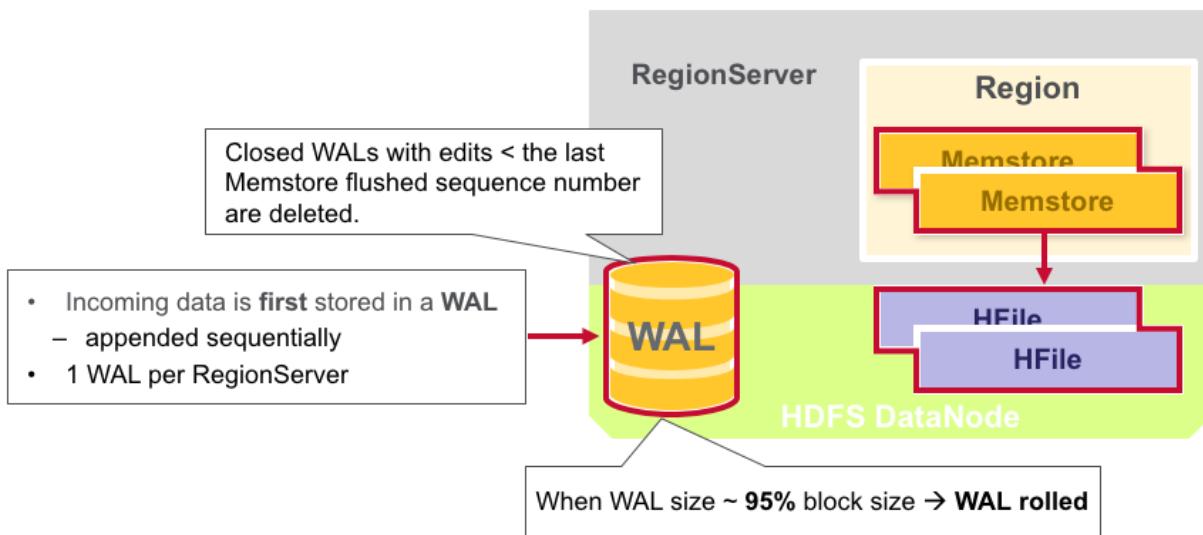
The resiliency to failures comes from HDFS. The HFile in HDFS is replicated by default 3 times. The WAL is also replicated 3 times.

## HDFS Data Replication



HFile block replication happens automatically. HBase relies on HDFS to provide the data safety. When data is written in HDFS, one copy is written locally, then it is replicated to a secondary node, and a third copy is written to tertiary node.

## HBase Write Ahead Log – WAL



MAPR Academy © 2016 MapR Technologies L3-68

Incoming data is first stored in a WAL, which is shared across all regions in a RegionServer. There is 1 WAL per RegionServer.

As WALs grow, they are eventually closed and a new, active WAL file is created to accept additional edits. This is called rolling the WAL file. By default, the WAL file is rolled when its size is about 95% of the HDFS block size.

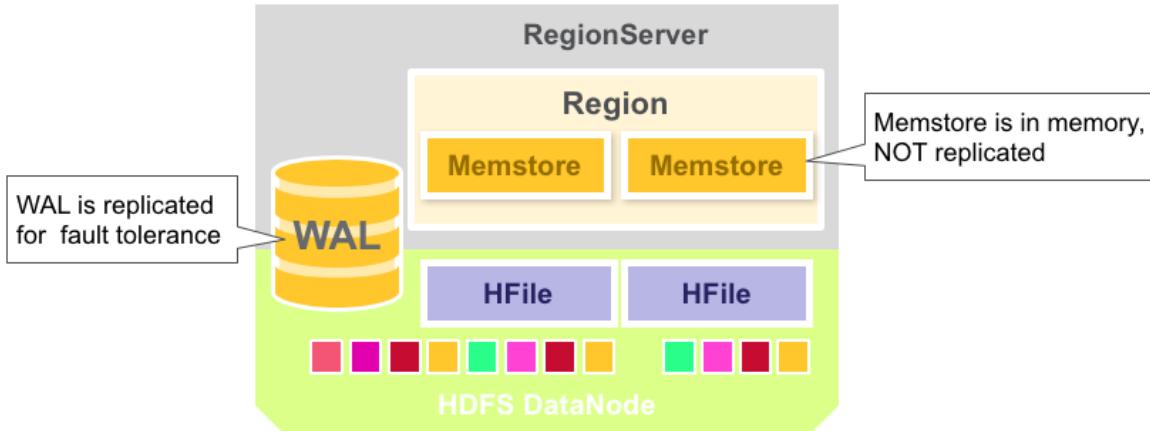
Edits in the WAL have a sequence number. When a Memstore flushes, the highest sequence number is stored as a meta field in the HFile.

When a region is opened, the sequence number is read and used in the WAL to reflect where persisting has ended, and where to continue.

Closed WALs with edits that have a sequence number less than the last Memstore flushed sequence number are deleted.

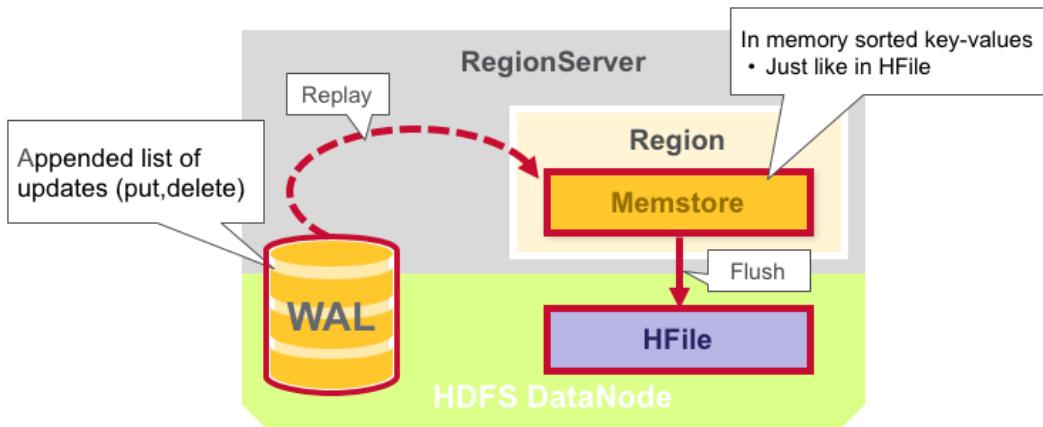
HBase client never reads from WAL. The WAL is shared by all regions hosted by the same RegionServer, which acts as a central logging backbone for every modification.

## HDFS Data Replication



The WAL file and the HFiles are persisted on disk and replicated. How does HBase recover the Memstore updates not persisted to HFiles?

## Data Recovery

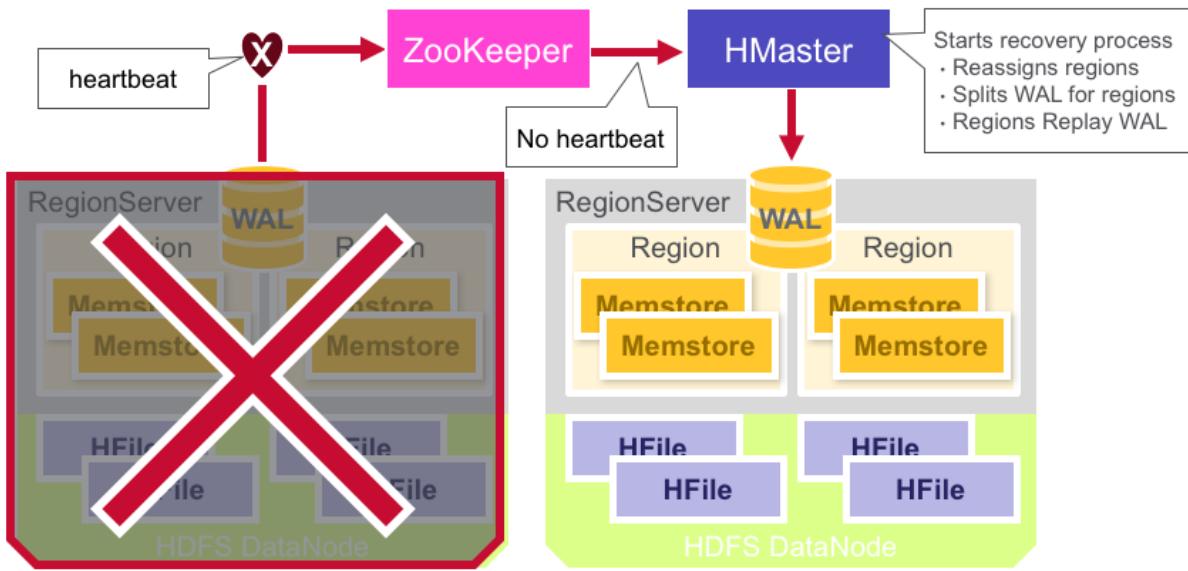


WAL files contain a list of edits, with one edit representing a single put or delete. Edits are written chronologically, so, for persistence, additions are appended to the end of the WAL file that is stored on disk.

What happens if there is a failure when the data is still in memory and not persisted to an HFile?

The WAL is replayed. Replaying a WAL is done by reading the WAL, then adding and sorting the contained edits to the current Memstore. At the end of the replay, the Memstore is flushed to write changes to an HFile.

## HBase Crash Recovery



When a RegionServer fails:

- Crashed regions are unavailable until detection and recovery steps have happened.
- ZooKeeper will determine node failure when it loses RegionServer heart beats.
- The HMaster will then be notified that the RegionServer has failed.
- When the HMaster detects that a RegionServer has crashed:
- The HMaster reassigns the regions from the crashed server to active RegionServers.
- To recover the crashed RegionServer's Memstore, edits that were not flushed to disk:
  - The HMaster splits the WAL belonging to the crashed RegionServer into separate files and stores these files in the new RegionServer's data nodes.
  - Each RegionServer then replays the WAL from the respective split WAL, to rebuild the Memstore for that region.

## RDBMS vs HBase

RDBMS Tables	HBase
Transactions across tables	Row based atomicity
Partitioning requires DBA	Automatic Distributed Regions
SQL queries	Key lookup/key range scans
Indexes for queries	Row key, no built in secondary index
Schema defines primitive data types	Byte arrays
In place update	Cell versioning

HBase is not a fully ACID compliant database, but does guarantee certain ACID characteristics.

Record-level puts are atomic for a single row, multi record/multi table puts are not.

To learn more about the specifics of HBase ACID compliance, refer to the HBase documentation.

## 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

HBase provides the following benefits

Strong consistency model

when a write returns, all readers will see same value

Scales automatically

Splits when regions become too large

Uses HDFS to spread data and manage space

Built-in recovery

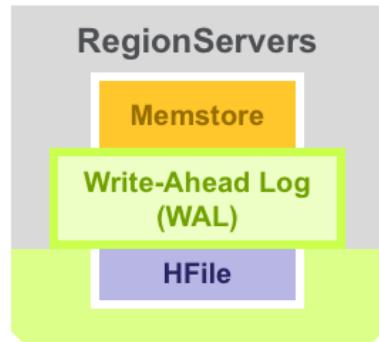
Using a Write Ahead Log, similar to journaling on file system

Integrated with Hadoop

MapReduce on HBase is straightforward

## Apache HBase Has Problems Too...

- **Reliability**
  - WAL replay slow
  - Slow complex crash recovery
- **Business Continuity**
  - Major Compaction I/O storms
- **Manageability**
  - complex moving parts: ZooKeeper, HBase Master, RegionServer, HDFS
  - Compactions, splits often must be done manually
- **MapReduce on HBase is straightforward**



While Apache HBase has many advantages, it also has some limitations.

Recovering a RegionServer after a crash can take 30 minutes or more. RegionServer crash can cause data to be unavailable during this time, while WALs are replayed for impacted regions.

Compactions use a lot of resources, and disrupt HBase operations. The I/O bursts during compaction can overwhelm nodes.

HBase does not have client throttling, and clients can easily overwhelm RegionServers causing downtime.

Basic administration is complex. Major compactions and splitting often have to be done manually, and coordinating separate distributed systems is very hard.

## Knowledge Check



## Knowledge Check



### What happens when a RegionServer fails?

1. ZooKeeper determines the node failure.
2. ZooKeeper notifies HMaster
3. HMaster reassigned regions from crashed server

## Knowledge Check



What happens when a RegionServer fails?

1. ZooKeeper determines the node failure
2. ZooKeeper notifies HMaster
3. HMaster reassigned regions from crashed server

## Knowledge Check



**How does HBase recover updates not persisted to HFiles when a RegionServer fails?**

1. HMaster splits the WAL belonging to crashed region server into separate files and stores in region directories to which they belong
2. Each RegionServer then replays WAL from respective split WAL to rebuild Memstore for that region

## Knowledge Check



**How does HBase recover updates not persisted to HFiles when a RegionServer fails?**

1. HMaster splits the WAL belonging to crashed region server into separate files and stores in region directories to which they belong
2. **Each RegionServer then replays WAL from respective split WAL to rebuild Memstore for that region**

## Learning Goals



## Learning Goals



- 3.1 Identify components of an HBase cluster
- 3.2 Describe how HBase components work together
- 3.3 Describe how regions work and their benefits
- 3.4 Define the function of minor and major compactions
- 3.5 Describe RegionServer splits
- 3.6 Describe how HBase handles fault tolerance
- 3.7 Differentiate MapR-DB from HBase**

We have looked at HBase architecture. We have seen the advantages and limitations of HBase. This section describes the differences between HBase and MapR-DB.

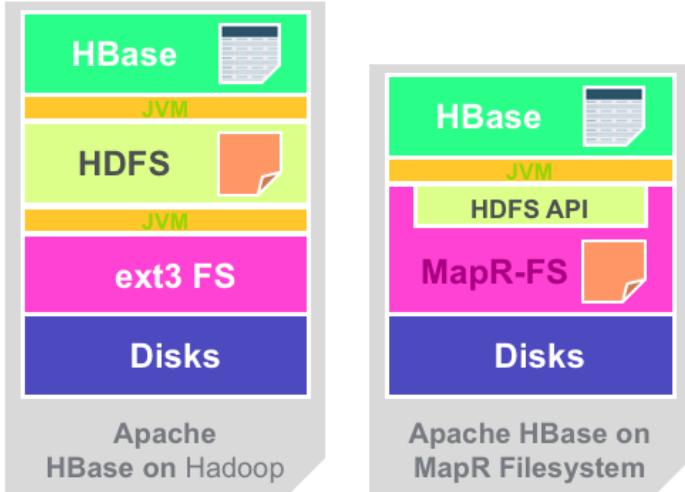
## What is MapR-DB?

- NoSQL database on Hadoop
- Supported by MapR distribution of Apache Hadoop
- HBase API compatibility
- NoSQL key value store

### What is MapR-DB?

It is a NoSQL database on Hadoop, and is supported by the MapR distribution of Hadoop. MapR-DB exposes the same HBase API, and stores data structured as a nested sequence of key-value pairs.

## HBase on HDFS vs. MapR-FS

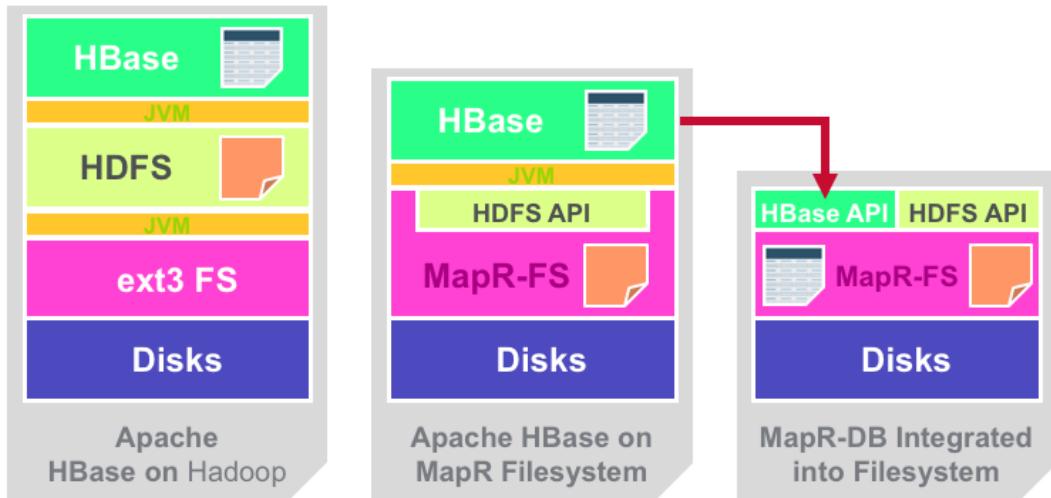


HBase on top of HDFS is shown on the left. You see in the diagram there are many layers. HDFS is separate from the underlying file system, and HBase is separate from HDFS. These layers and separation, plus the limitations of a write once HDFS file system lead to the HBase problems we discussed earlier.

MapR-FS maintains compatibility with the Hadoop APIs, as shown here, and HBase can run on top of MapR-FS, shown here on the right. Compared to HDFS, the MapR Filesystem implements the storage layer natively in C++, and accesses disks directly. This eliminates a JVM and the ext3 filesystem layer. The benefits of accelerated disk performance and system stability are multiplied across all nodes in the cluster.

MapR took the append-only architecture of HDFS, made it a fully read-write system, and gave it a Network File System mount. MapR-FS also provides high availability for NameNode functionality, the part of HDFS that acts like a file allocation table in your disk drive, remembering where all the data is, and that was a single point of failure in Hadoop.

## MapR-DB and Files in a Unified Storage Layer



MAPR Academy © 2016 MapR Technologies L3-84

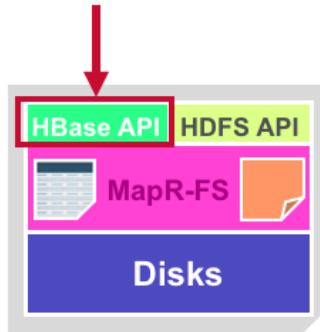
The diagram shown here compares the application stacks for different HBase implementations.

MapR-DB exposes the HBase API and the Data model, and stores data structured as a nested sequence of key/value pairs. The value in one pair serves as the key for another pair. The MapR-DB implementation integrates table storage into the MapR file system, eliminating all JVM layers and interacting directly with disks for both file and table storage.

The MapR file system is written in C and optimized for performance. As a result, MapR-FS runs significantly faster than JVM-based HBase. With MapR-FS, HDFS functionality is pushed down into a distributed NFS file system, supporting all of the APIs of HDFS. With MapR-DB, the file system can handle small chunks of data, and also small pieces of HBase tables.

## Portability

- MapR-DB uses the HBase data model and API
- Apache HBase applications work as-is on MapR
  - No need to recompile
  - No vendor lock-in



MapR-DB uses the same data model as HBase, and supports the core HBase API. This means that you can port your application from HBase to Map-DB without re-compiling and you are not locked in to a specific vendor.

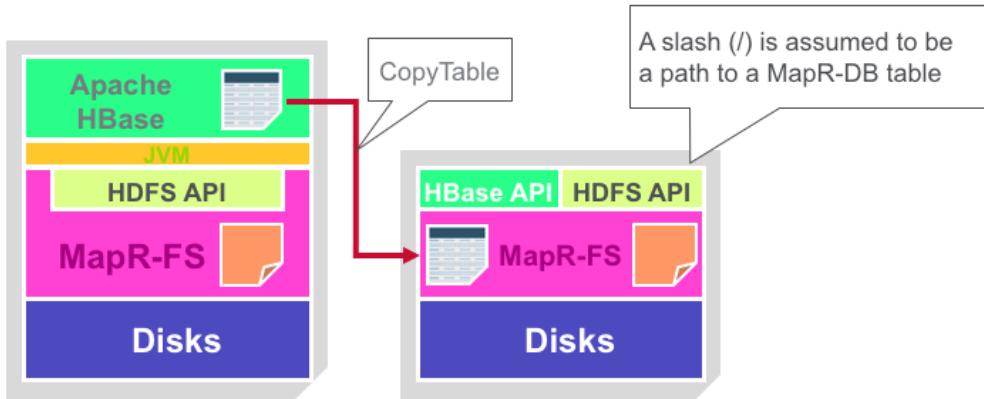
Also you can use MapR-DB tables as the data store for Hadoop, Hive, and other Hadoop ecosystem applications, just like you would use HBase. MapR preserves the standard Hadoop and HBase APIs, so all ecosystem components continue to operate without modification.

Because MapR uses the open-standard HBase API, many legacy HBase applications can continue to run on MapR without modification.

- The MapR table API works with the core HBase API
- MapR tables implement the HBase feature set
- You can use MapR tables as the data store for Hive applications

## Portability

You can run HBase and/or MapR-DB on a MapR cluster



You can run HBase and MapR-DB on a MapR cluster, you can use MapR-DB exclusively, or in a mixed environment with HBase tables. A mixed environment is recommended when porting applications from HBase to MapR-DB, and you can use the standard CopyTable tool to copy a table from HBase to MapR-DB.

### Using MapR and Apache HBase Tables Together

MapR-DB table storage is independent from HBase table storage, enabling a single MapR cluster to run both systems. Users typically run both systems concurrently, particularly during the migration phase. Alternately, you can leave HBase running for existing applications, and use MapR-DB tables for new applications. You can set up [namespace mappings](#) for your cluster to run both MapR tables and Apache HBase tables concurrently, during [migration](#) or on an ongoing basis.

MapR's implementation of the HBase API differentiates between Apache HBase tables and MapR-DB tables, based on the table name.

- By default, if a table name includes a forward slash, the name is assumed to be a path to a MapR-DB table. Forward slash is not a valid character for HBase table names.

You can treat a MapR-DB table just as you would a file, specifying a path to a location in a directory. The table appears in the same namespace as your regular files, without the need to coordinate with a database administrator.

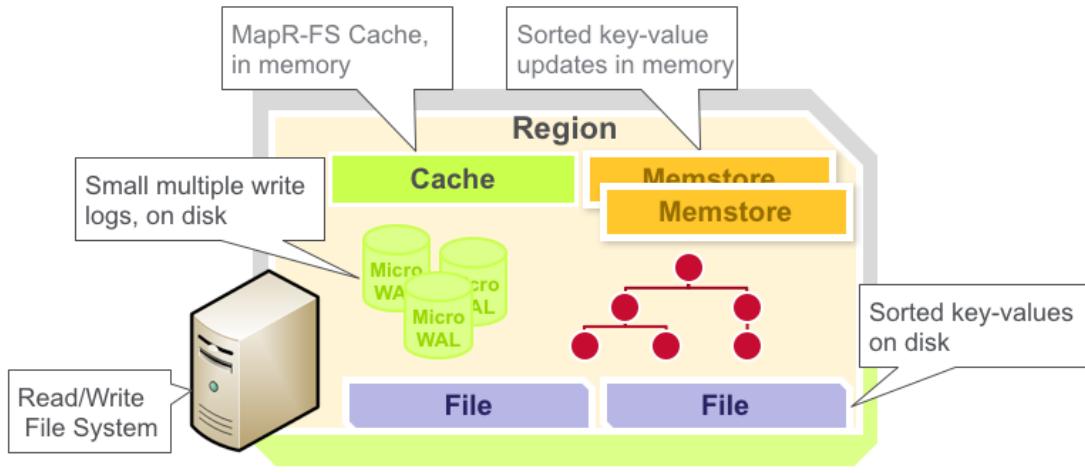
- MapR-DB has table mapping support, so to the user, the table names can just be names without forward slash
- Namespace mappings can be setup to distinguish between HBase tables and MapR tables based on path/table name
- During [data migration](#) or other specific scenarios where you need to refer to a MapR table of the same name as an HBase table in the same cluster, you can [map the table namespace](#) to enable that operation

The `hbase.table.namespace.mappings` property allows you to map HBase table names to MapR tables. This property is set in the configuration file `/opt/mapr/hadoop/hadoop-<version>/conf/core-site.xml`.

In this example, any flat table name `foo` is treated as a MapR-DB table in the directory `/tables_dir/foo`.

```
<property>
<name>hbase.table.namespace.mappings</name>
<value>*:./tables_dir</value>
</property>
```

## MapR-DB Region Components



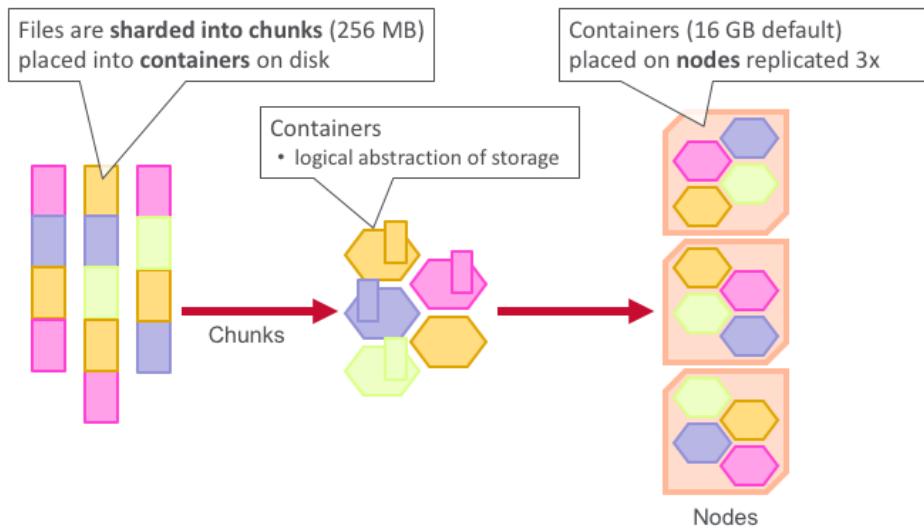
Let us take a look at the key differences in the MapR-DB architecture and what MapR-DB brings.

Here you see the MapR-DB region components:

- Micro WALS: instead of one big WAL per region for recovery, MapR-DB has many small micro-WALs
- Integrated caching: there is no special separate MapR-DB read-cache like for HBase. The filesystem cache is used by both tables and regular files.
- Memstore: stores sorted key values in memory. Like HBase there is one Memstore per column family per region, however MapR-DB can have more column families than HBase

Table Cells are stored in Files on MapR-FS. HBase has to work with HDFS. That means that only large I/O operations are efficient. These limits are imposed by the HDFS architecture. The result of these architectural limits is that HDFS has to share a commit log (WAL) across all regions in a region server. MapR's read/write file system supports efficient small I/O operations which allows MapR-DB to have multiple micro-WALS per region.

## Data Storage Overview:

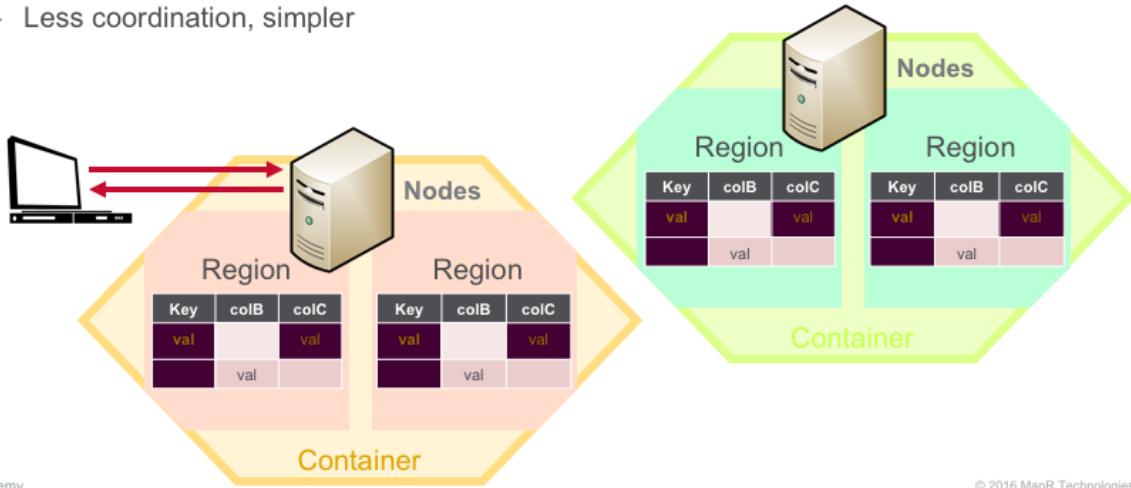


A container is the fundamental unit of storage within a MapR cluster, and is also the unit of replication. A container is big, 16GB by default, and can range from 10-30 GB. MapR-FS files are sharded into chunks that are written into a number of containers. The container contents are then replicated across the cluster.

## MapR-DB Table Storage

### MapR-DB regions live inside a container

- served by MapR fileserver service running on nodes
- No RegionServer, No HBase Master
- Less coordination, simpler

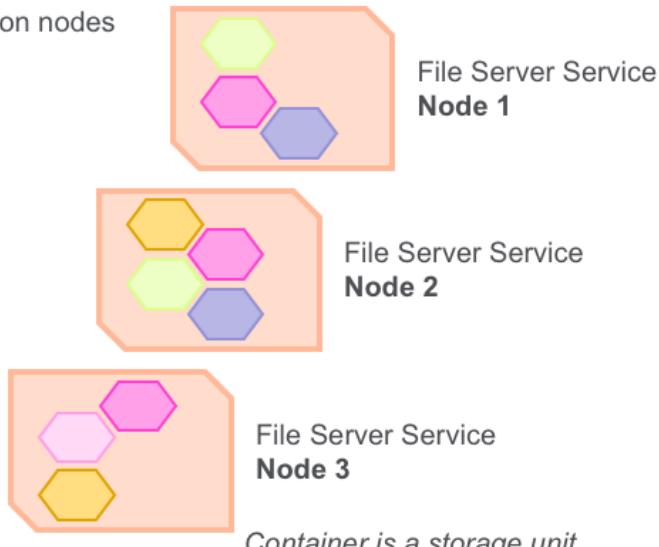


With MapR-DB tables, like with HBase, continuous sequences of rows are divided into regions. In MapR-DB however, the regions live inside a container. Because the tables are integrated into the file system, MapR-DB can guarantee data locality. There are multiple regions per container. Regions are 4GB~6 GB in size and containers are 10-30GB in size. Therefore, there are about 4-5 regions per container.

## MapR-FS: Fileservers Serve Regions

### Region lives entirely inside a container:

- Served by fileserver service running on nodes
- Guaranteed data locality
- Smarter load balancing
  - o Uses container Replicas
- Smarter fail over
  - o Uses container replicas



This summarizes the features of regions living inside a container.

**Guaranteed data locality:** Because the tables are integrated into the file system, MapR-DB can guarantee data locality. The table data is stored in files which are guaranteed to be on the same node because they are in the same container. With HBase, whenever a region is moved for failover or load balancing the data is not local. The RegionServer is reading from files on another node.

**Smarter load balancing**

**Uses container Replicas:** If a region needs to be moved for load-balancing, it is moved to a container replica. The data is still local.

**Smarter, simpler fail over:** If a region needs to recover from fail over, it is recovered from a container replica. The data, the WAL and table data in files, is still local. No complex co-ordination required between ZooKeeper, the HBase-master, and the RegionServer on failover.

**Uses container replicas:** container replica regions replay their micro-WALs independently

**Less coordination layers:** Simpler, more reliable.

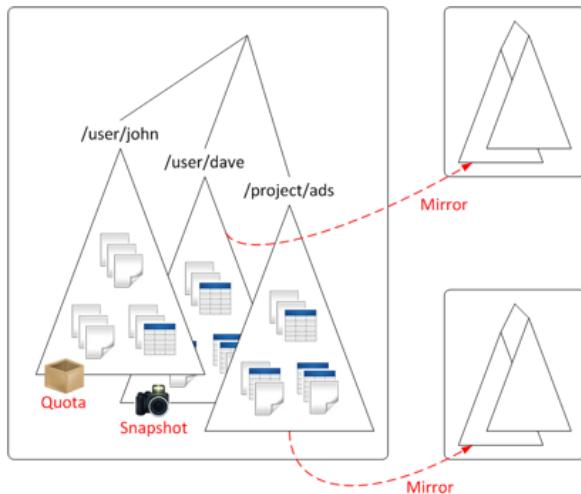
## MapR-DB Volumes Hold Files and MapR-DB Tables

### MapR-DB tables are a type of file in MapR-FS

- Store files and tables in the same volumes through out the cluster

### Benefit from same Volume capabilities

- Snapshot, mirror support
- Dump, restore support



© 2016 MapR Technologies

L3-91

One powerful feature of MapR is volume management, which allows you to logically segment portions of your cluster into separate volumes. This helps with multi-tenant applications that require a separation of data and users in a single cluster.

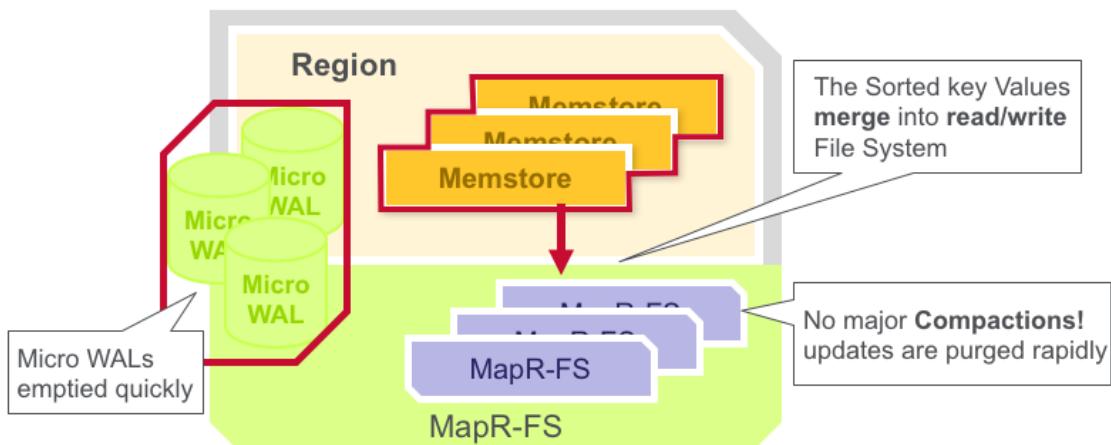
MapR-DB tables are a file in MapR-FS, and benefit from all built-in MapR-FS volume capabilities, such as snapshots and mirroring.

HBase and HDFS are separate storage systems. You have limited data management options on the HBase tables, and you also can't do snapshots, backups, or mirrors of files and tables the way that you can with the MapR file system.

With MapR-DB, the unstructured data in the file system and the MapR-DB HBase tables of the file system can be interwoven. The tables and their underlying data files can be grouped together, snapshotted together, and backed up to mirrors together.

MapR-DB creates a unified data layer that makes the same snapshots, mirroring, namespace, files, and tables accessible from the same management.

## Map-DB - No Compaction



When the Memstore accumulates enough data:

1. The sorted key-values are merged into read/write files on MapR-FS
2. During this merge, deleted key-values are purged from files
3. The Memstore cache is emptied
4. The micro-Wal is deleted

## Purging Region Data

### MapR-FS purges regions automatically

- Tombstoned rows. Over-aged cell versions (ttl). Etc...
- Seamlessly behind the scenes.

### No Apache HBase-style compactions

- No manual intervention
- No performance degradation
- No downtime

HBase gets rid of deleted data or expired data during major compactions.

With MapR-DB, deleted or expired data is purged automatically. This happens whenever the data is updated.

The big difference is that MapR-DB does not need to do compaction. The merges are smaller than HBase. The sorted read/write files on disk are 2-4MB, to allow for small purges.

## MapR Region Splits

- **Split operation is automatic and very fast in MapR-FS.**
  - Zero-Copy splits, new data goes into new region
  - Smart splits, based on size not just key range
  - Transparent to users
- **Pre-splitting table regions is supported:**
  - Generally not required
  - Can help with load balancing at the start of a new cluster
  - Access patterns must be known to pre-split regions

In MapR-FS, the split operation is automatic and fast, and supports pre-splitting a region. While pre-splitting is not required, it can help with load-balancing.

## HBase – MapR-DB

	HBase	MapR-DB
Reliability	WAL large → replaying can take long → long recovery	Many micro-WALs
Business Continuity	Major compaction → I/O storms	No compaction
Manageability	Compactions and splits quite often done manually	No need for manual intervention

## Knowledge Check



## Knowledge Check



1. Tables integrated into filesystem guaranteeing data locality
2. Efficient large I/O operations only
3. Write amplifications during major compaction
4. Efficient large and small I/O operations
5. Fast recovery since region recovered from container replica
6. Deleted data, expired data purged automatically, when data is touched
7. Not easy to do snapshots and mirroring
8. Unified data layer results in easy snapshotting and mirroring

HBase

MapR-DB

## Knowledge Check



1. Tables integrated into filesystem guaranteeing data locality
2. Efficient large I/O operations only
3. Write amplifications during major compaction
4. Efficient large and small I/O operations
5. Fast recovery since region recovered from container replica
6. Deleted data, expired data purged automatically, when data is touched
7. Not easy to do snapshots and mirroring
8. Unified data layer results in easy snapshotting and mirroring

HBase

**2, 3, 7**

MapR-DB

**1, 4, 5, 6, 8**

## References

- <http://hadoop.apache.org/>
- <http://hbase.apache.org/>
- <http://www.mapr.com/>
- HBase in Action, Nick Dimiduk, Amandeep Khurana
- HBase: The Definitive Guide, Lars George



## Next Steps

### DEV 325 – Apache HBase Schema Design

Lesson 4: Basic Schema Design

Congratulations! You have now completed DEV 320 Lesson 3. You should now have an understanding of the HBase data model and architecture.



## DEV 325 – Apache HBase Schema Design

Lesson 4: Basic Schema Design

Winter 2017, v5.1

Welcome to DEV 325 – Apache HBase Schema Design, Lesson 4: Basic Schema Design.

## Learning Goals



## Learning Goals



- 4.1 List the Elements of Schema Design
- 4.2 Design Row Keys for Data Access Patterns
- 4.3 Design Table Shape and Column Families for Data Access Patterns
- 4.4 Define Column Family Properties
- 4.5 Design Schema for Given Scenario

In this lesson, we will look at schema design principles for HBase tables. These principles apply equally to MapR-DB tables.

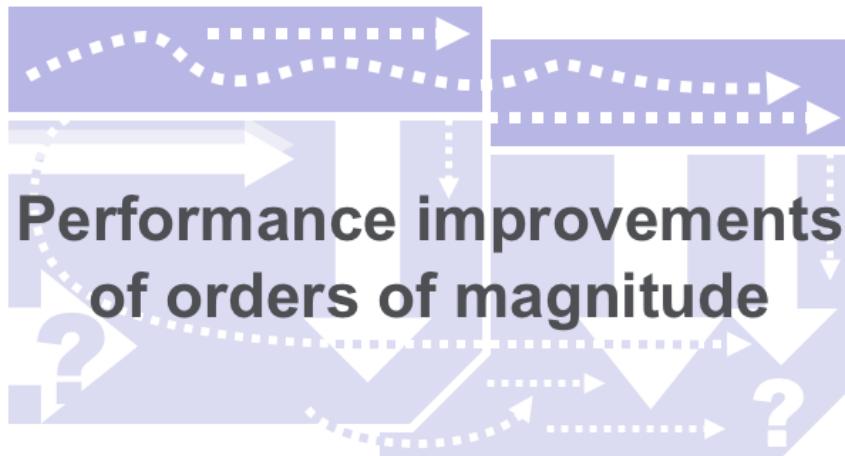
## Learning Goals



- 4.1 List the Elements of Schema Design**
- 4.2 Design Row Keys for Data Access Patterns
- 4.3 Design Table Shape and Column Families for Data Access Patterns
- 4.4 Define Column Family Properties
- 4.5 Design Schema for Given Scenario

In this first section, we will take a look at the elements of schema design.

## Why Design Matters



Why does design matter?

Schema design matters for the same reasons that design always matters for everything: If you use the most advanced materials in the world to construct an inefficient car design, you still get an inefficient car. A properly designed schema can make all the difference in how your application performs, and either improves or hinders all aspects of the application that rely on the performance of the datastore.

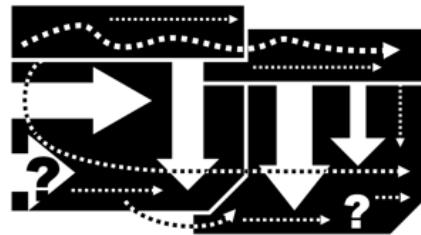
The importance of design isn't new. In relational databases, for example, things work better when you understand your schemas and optimize queries. Similarly with HBase tables, knowing how you will access data and what kinds of queries you plan to execute, will help get the best performance.

We have a favorite anecdote at MapR where one of our data scientists worked with a customer, and in a one-hour conversation about schema design, was able to improve access performance by a factor of 1,000x. These concepts matter.

## A Schema-less Database?

**HBase is “schema-less?”**

**No - it is a misnomer**



HBase is sometimes called a schema-less database but this is a misnomer. HBase doesn't require the same predefined structure as a relational database, but you do have to define the facets of how you plan to organize your data.

Before we delve into designing schemas in HBase, let us do a quick review of the HBase Data Model.

## Review



## Review



Component	Description
Table	Data organized into tables; comprised rows
Row Key	Data stored in rows; rows identified by row keys
Column Family	Columns grouped into families
Column Qualifier	Identifies the column
Cell	Combination of the row key, column family, column, timestamp; contains the value
Version	Values within cell versioned by version number → timestamp

Here is a table that lists the data model components:

- HBase tables are comprised of rows and columns.
- The row key is the primary index into the table and is sometimes called the row ID.
- Row keys are stored in lexicographical order.
- Values are stored associated with row key and column coordinates. Columns are sometimes called qualifiers, which you'll often see in the HBase Javadoc.

All columns belong to a particular column family.

## Review: HBase Data Model Column Families

Row Key	Address			Order			
	street	city	state	Date	ItemNo	Quantity	Price
smithj	Central Dr	Nashville	TN	2/2/15	10213A	1	109.99
spata	High Ave	Columbus	OH	1/31/14	23401V	24	21.21
turnerb	Cedar St	Seattle	WA	7/8/14	10938A	15	1.54

Row Keys: Identify the rows in an HBase table

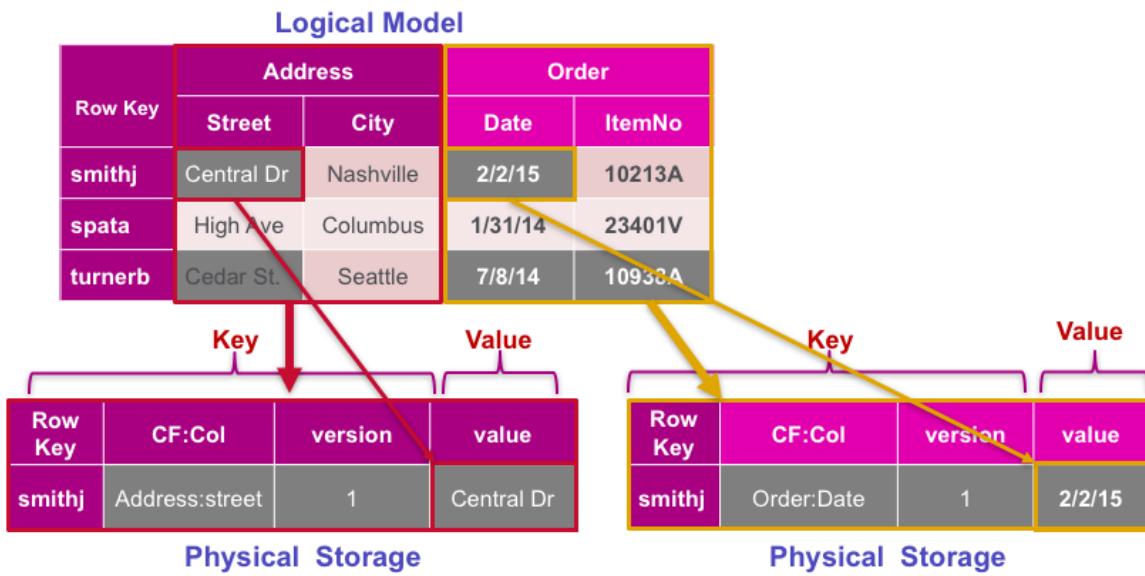
© 2016 MapR Technologies

L4-9

Going a little more into the data model components:

- A table has one or more column families. This diagram shows a table with two column families, namely Address and Order.
- Column Family is often referred to as "family" in the HBase API Javadoc.
- Data in column families are stored separately and can be accessed independently. You can think of column families as separate tables that have records that use the same set of row keys.
- In Apache HBase, you don't want to go over three or so column families or performance suffers dramatically. With MapR-DB, you can use up to 64 column families with no performance penalty.
- A column family can have any number of associated columns – it could be two or 1000.
- Column families are typically defined when you create a table, whereas columns may be added and named dynamically.

## Review: Logical Data Model vs. Physical Data Storage



The top graphic shows the logical layout of the data in table format, the lower graphic shows how the data is actually physically stored in files.

Physically column families are stored in separate files, rows are stored as sorted sets of key-value cells. The entire cell coordinates, the row key, column family name, column name, and timestamp are stored for every cell which has a value.

## Review: Sparse Data with Cell Versions

	CF1:colA	CF1:colB	CF1:colC
Row1	@time7: value3 @time6: value1 @time5: value1		
Row10	@time2: value1	@time2: value1	
Row11	@time6: value2 @time5: value3		
Row2	@time4: value1		@time4: value1

RowKey:Family:Column:Version → Value

Complete coordinates to a cell's value are: Table:Row:Family:Column:Timestamp → Value.

HBase tables are sparsely populated. If data doesn't exist at a column, it's not stored.

Table cells are versioned uninterpreted arrays of bytes. The version is by default a timestamp that is a long. You can use the timestamp or set up your own versioning system. So for every coordinate row:family:column, there can be multiple versions of the value.

Finally, recall that cells can have properties like time-to-live or max-versions, which can cause the values in certain cells to be purged automatically at some future point.

## Review: 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

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	

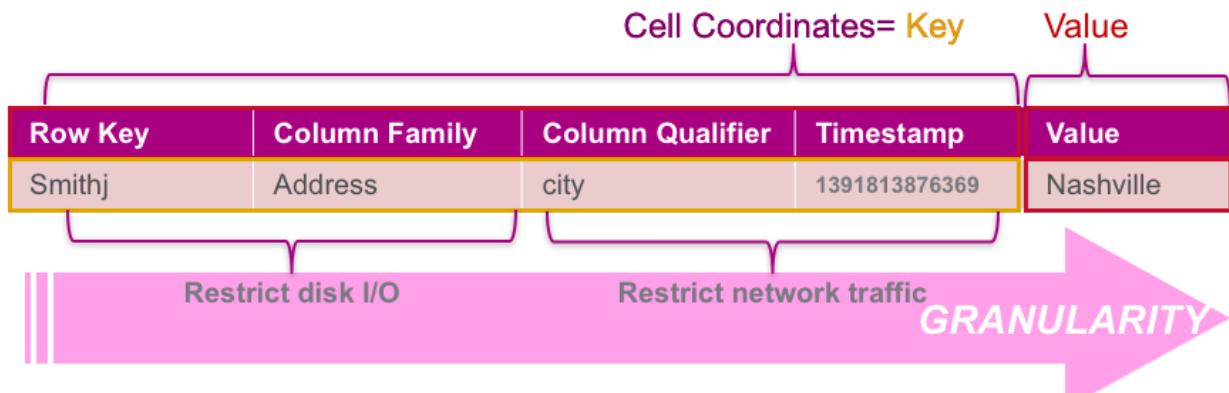
Sorted by Row Key and Column

Sorted in descending order

Review of physical storage properties:

- The physical layout in storage is different than the logical representation of a table.
- Rowkey:ColFamily maps to a sub-map of Columns.  
Rowkey:ColFamily:Column maps to a sub-sub-map of versions.
- All values are stored with the full coordinates:  
Table:Row:ColFamily:Column:Timestamp → Value
- Every cell version is a separate entry in the table, represented as a "row" in this slide.
- Physically data is stored by column family as a sorted map of nested maps.

## Query Granularity



The Cell coordinates or Key consists of the row key, column family name, column name, and timestamp.

When retrieving data, we can specify more or less information about the coordinates for the values we want to get back. In broad terms, we can think of the granularity of the query getting finer as we move to the right, in other words, as we provide more criteria for the cell coordinates. We could require an exact match for certain coordinates, or we can specify columns, timestamps or use filters to retrieve all data that matches certain patterns.

Remember only the row key is indexed in HBase tables.

Thinking about system performance:

- Specifying row key and column family limits disk I/O, since these elements allow the server to skip over entire chunks of data in files on disk that don't apply.
- Specifying column name and timestamp limits network traffic. The server still has to spin the disks to find the data, but will strip off the parts you're not interested in before sending it over the network.

In other words, smart row key design is going to improve efficiency in scanning just the right rows, and specifying to get a subset of columns cells will reduce network traffic to return your results.

In the next slide, we will see how the key-value works when retrieving data.

## Query Granularity

Minimize disk seek

```
hbase> scan '/user/user01/customer'  
hbase> get 'jsmith', '/user/user01/customer'  
hbase> get 'jsmith', '/user/user01/Customer', {COLUMNS=>['addr']}  
hbase> get 'jsmith', '/user/user01/Customer', {COLUMNS=>['order:numb']}
```

Gets all the data for the row  
Gets every column in a column family  
Gets a specific column in a column family  
Gets a specific column in a column family

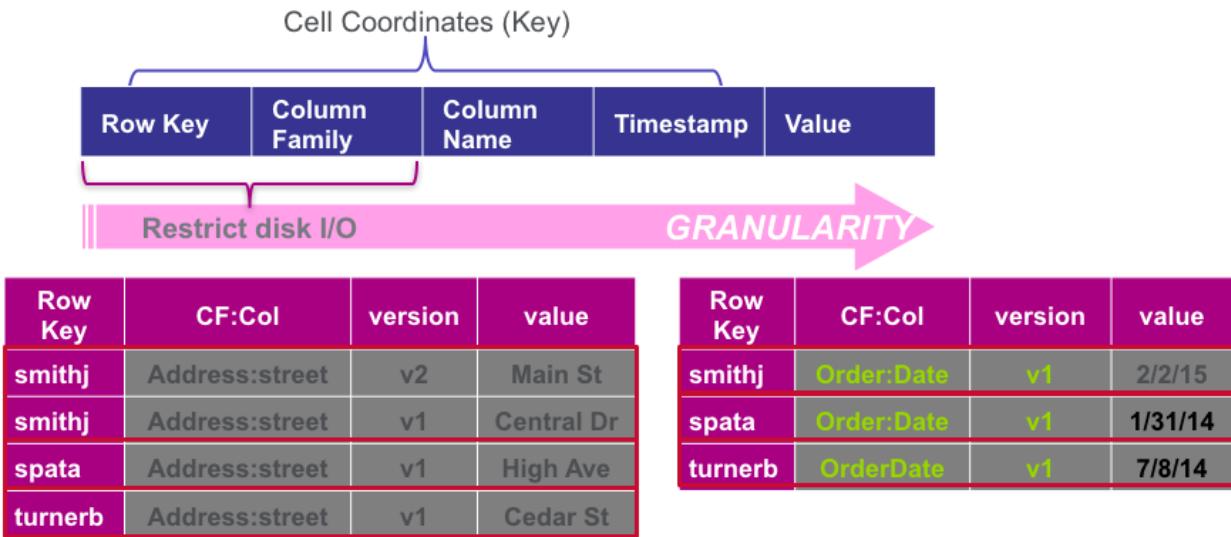
Restrict network traffic      GRANULARITY

MAPR Academy      © 2016 MapR Technologies      L4-14

As you provide more criteria for the coordinates, you get less data returned.

You can limit the amount of data you read off the disk or transfer over the network. Specifying the row key lets you get just the exact row you need. Specifying the column family lets you specify what part of the row to read, sparing reading multiple HFiles if the row spans multiple families. Specifying the column name lets you save on the number of columns returned to the client, thereby saving on network I/O.

## Query Granularity



As a reminder, the image in the bottom shows how the data is stored. Since only the row key is indexed in HBase tables, smart row key design is going to improve efficiency in finding the right rows for scans.

Specifying the column family limits disk I/O by helping to find the file where the requested data is stored. If you do a get just providing the row key smithj it will read from two files to get all the columns in CF. As you provide more criteria for the coordinates, you get less data returned.

## Query Granularity



	Row Key	Column Family	Column Qualifier	Time Stamp
Limit Rows Scanned	✓			
Limit HFile access (Apache HBase only)	✓	✓		✓
Limit Disk I/O	✓	✓		✓
Limit Network I/O	✓	✓	✓	✓

Source: Lars George's HBase: The Definitive Guide

© 2016 MapR Technologies

L4-16

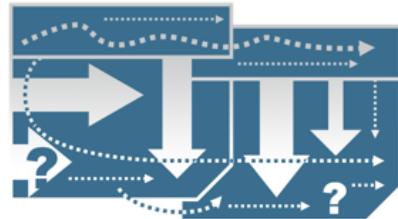
This table is another way to visualize how query parameters impact system behavior. A yellow check indicates a relationship.

You see that specifying a column qualifier will limit the network I/O only, but it doesn't do anything to limit the rows scanned or limit disk I/O. We also see that specifying a column qualifier has no impact on limiting HFile access since the HFile has to be read in order to examine the column qualifier.

The timestamp can help you find the file and add more granularity, thereby limiting the disk I/O.

## Elements of Schema Design

- Column families
- Row key design
- Use of columns
- Use of cell versions
- Column family attributes
  - Compression
  - Time-to-live
  - Max versions



Now that we have reviewed HBase data model concepts, let us look at the elements of schema design.

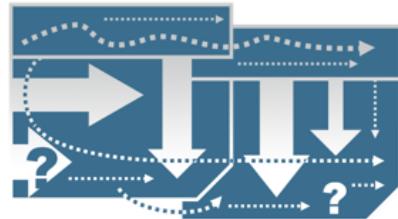
When we say "schema design," these are the design choices we're referring to:

- Grouping of data into column families
- Composite row key design. We'd like to optimize for:
  - Write patterns, and
  - Read patterns
- Usage of columns and sparse data in columns
- Usage of cell versions
- Column family attributes that impact cell values, such as compression, time-to-live, and max-versions

## Data Access Patterns

Model for the questions:

- How will your application access data?
- What information is accessed together?
  - Get (returns data in a row)
  - Put data that needs to be read together in row
  - Scan (returns data by row key range)



Schema design matters because proper design directly affects the bottom line. In HBase, we model for the questions. The schema should be designed based on:

- How is the application going to access data?
- What information is accessed together?
- When you use the get operation to return data, it returns data in a row. Thus put the data that should be read or updated together in a row.
- Scans return data by row range key – so design the row key appropriately.

## Main Use Case Categories

- Time Series Data
  - High Volume, Velocity Writes
- Information Exchange, Messaging
  - High Volume, Velocity Write/Read
- Web Application Backend
  - High Volume, Velocity Reads

**What is the best schema to use?  
Depends on the data access patterns!**

Remember that HBase has been used in three major types of use cases:

1. Capturing incremental data – time series data  
This is the case with high volume and velocity writes.
2. Information exchange  
These cases represent high volume and velocity writes and reads.
3. Content serving  
These cases use high volume and velocity reads.

The best schema design for your particular application will depend on your data access patterns. As with any database design, you have to know how data will arrive and be accessed. Let us now take a look at an example use case.

## 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



We are going to look at a concrete example to explore the impact of schema design choices. Let's contemplate a system that records all trade events on a stock exchange.

In a day, there could be millions of trade events, which we'll represent with a trade object. A minimal trade object would comprise the details shown here.

## Consider Access Patterns for Application

- Data retrieval – by date/company: row key design
- What data accessed together: column family design
- What needs fastest access (or most frequent): row key ordering (or timestamp design)
- Do all trades need to be saved forever: column family attributes

We will go into more detail in later slides, but to get the discussion going, let's pose the following questions about our stock trades example:

- How will data be retrieved? By date? By company? → (\*) These factors drive at row key design. If we want to scan based on company name, then we will want to build the company name into the row key somehow.
- Are Price and Volume data typically accessed together, or are they unrelated? → (\*) These factors drive at column family design. You want to keep together in the same column family those data that are accessed together.
- Which trade data needs fastest access (or most frequent)? → (\*) These factors drive at row key ordering and possibly design of the timestamp. Scan order of row keys is from lowest-to-highest, so you might choose to insert the MOST RECENT items with the LOWEST row key.
- Do all trades need to be saved forever? → (\*) This factor drives at purging past cell version by setting Time-To-Live (TTL) or MaxVersions on a column family.
- What are the needs for atomicity of transactions? → (\*) This factor drives at column design. Atomicity is only guaranteed when putting to one row key, so if you need atomic updates to some elements in a table, you need to keep all elements on the same row.

## Knowledge Check



## Knowledge Check



What is the importance of a well-designed schema?

- 1) Affects disk storage
- 2) Improves performance by orders of magnitude

Answer: 2

## Knowledge Check



Match the design choices you can make to what they affect:

- a) Column families
  - b) Composite row keys
  - c) Column family attributes
- 
- 1) How it impacts cell values
  - 2) How data is grouped
  - 3) Optimize for read patterns and write patterns

Answers: A - 3, B - 1, C - 2

## Learning Goals



## Learning Goals



- 4.1 List the Elements of Schema Design
- 4.2 Design Row Keys for Data Access Patterns**
- 4.3 Design Table Shape and Column Families for Data Access Patterns
- 4.4 Define Column Family Properties
- 4.5 Design Schema for Given Scenario

In this section, we are going to look at designing row keys.

## Row Key Basics

- Primary index for the table
- A row key cannot be changed after writing
- Sorted in lexical order
- Sorted on insert, not scan
- Row key design directly impacts performance



Row keys are the primary index for a table. There are no secondary indexes out of the box with HBase. That means you HAVE to use the row key or a column name to get to data, thus what goes into a row key is important.

Also note that the only way to change row keys once data is written to the table is to delete and then re-insert data. This could be expensive!

Remember that row keys are sorted when inserted in lexical order, so take this into consideration when designing your row keys. If you need to keep a more natural order and if your keys contain digits, then consider padding the keys with zeroes to the left.

Row keys are sorted on inserts and not on scans.

## Composite Keys

- Use composite row key to bound scan ranges and provide sub-indexing to cell data
- Include multiple elements in the row key
- Use a separator or fixed length
- Example row key format:

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

- Ex: GOOG\_20131012
  - Get operations require complete row key
- Scans can use partial keys
  - Ex: "GOOG" or "GOOG\_2014"

HBase tables typically use a composite key design, which means that multiple data elements are included in the row key. We use that composite row key to bound scan ranges and provide sub-indexing to cell data.

You can include multiple data elements in the row key. For example, date or timestamp, data source, user, or other metadata can be part of the row key. In extreme cases, the row key might contain a lot of metadata about a value.

Note that Get operations require you to provide a precise row key. Thus for Gets, the application logic has to be able to construct a complete row key. Scans, on the other hand, can use partial row keys to capture all row keys starting with a certain value.

## Consider Access Patterns for Application

**How will data be retrieved?**

By date? By hour? By companyId?

- *Row key design*

**What if the Date/Timestamp is leftmost?**

Row Key				
1391813876369_AMZN				
1391813876370_AMZN				
1391813876371_GOOG				

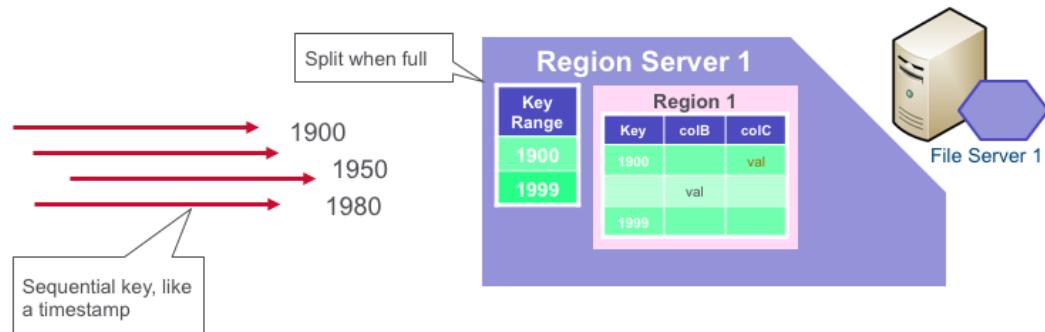
Thinking about access patterns ahead of time is probably the single most important consideration. Designing your schema around how the data will be accessed, both reads and writes, will directly impact performance.

An important question to ask is how data will be retrieved. Referring to our example with trades, how we want to retrieve data will impact the row key design. Do we want to retrieve by date, by hour, or companyId? If yes then we should include these in the row key.

What happens if we use the time stamp in the row key and it is leftmost in the row key? We know that row keys are sorted in lexical order – so in this case it means our row keys are sequential by time stamp.

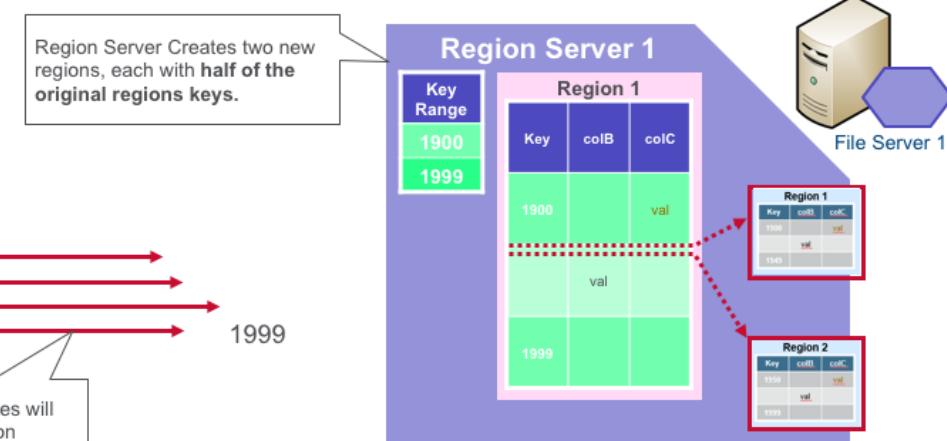
Let us see next what happens when we have sequential row keys written in order.

## Hot Spotting and Region Splits



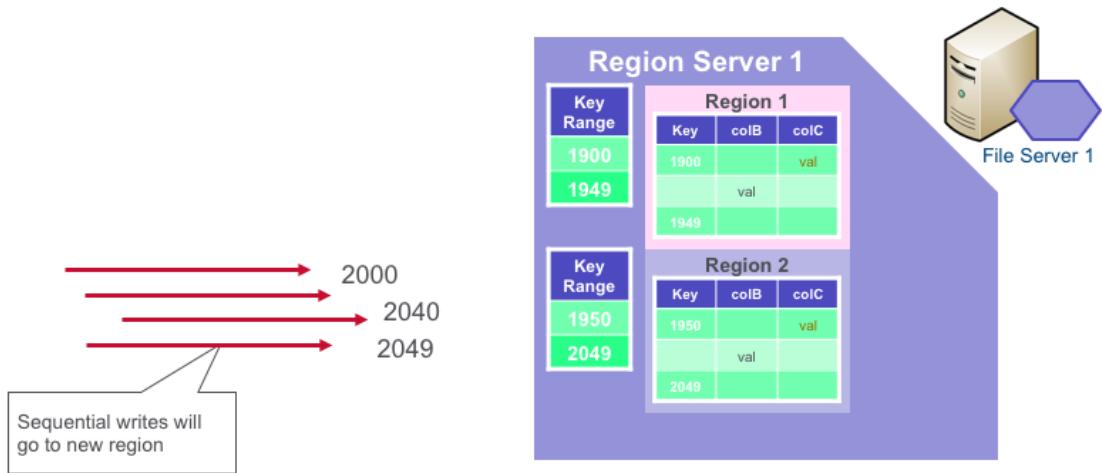
When we have sequential row keys, data is written to the table sequentially. Here we see data being written to the table in region 1 in region server 1 which is on file server 1. We know that when a region is full, it splits.

## Hot Spotting and Region Splits



Since data is written in sorted order, if the keys only increase with time, then every key is written to the end of a table. The end of the table will be contained in a single region, served by a single file server. As you see here, the data is being added to region 1. When it is full, it splits into two with data from region 1 being divided equally between region 1 and region 2.

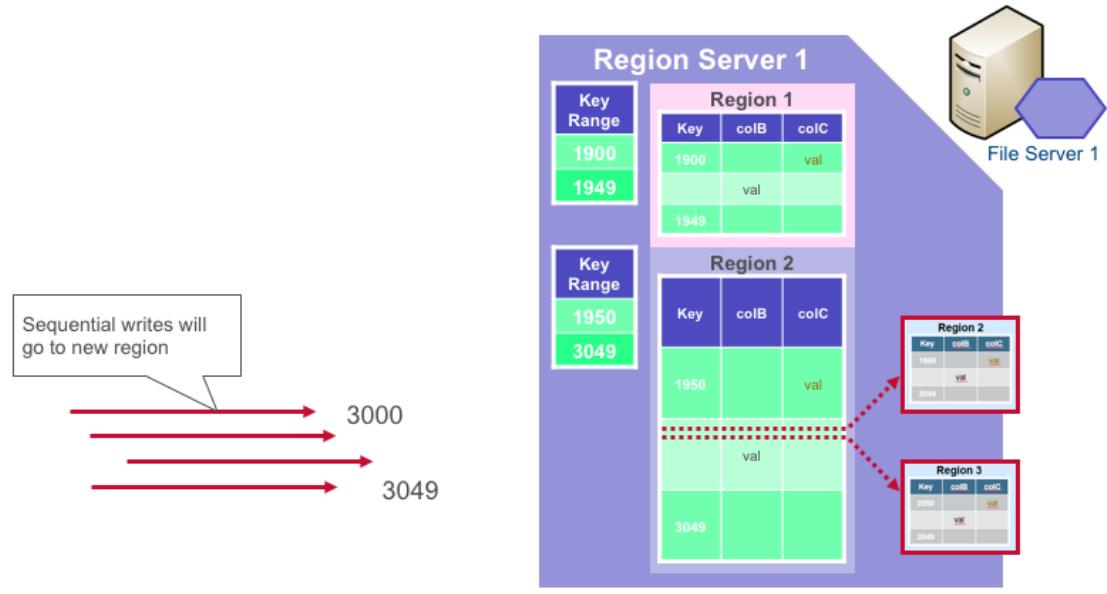
## Hot Spotting and Region Splits



Since data is written in sorted order, if the keys only increase with time, every key is written to the end of a table. The end of the table will be contained in a single region, served by a single file server.

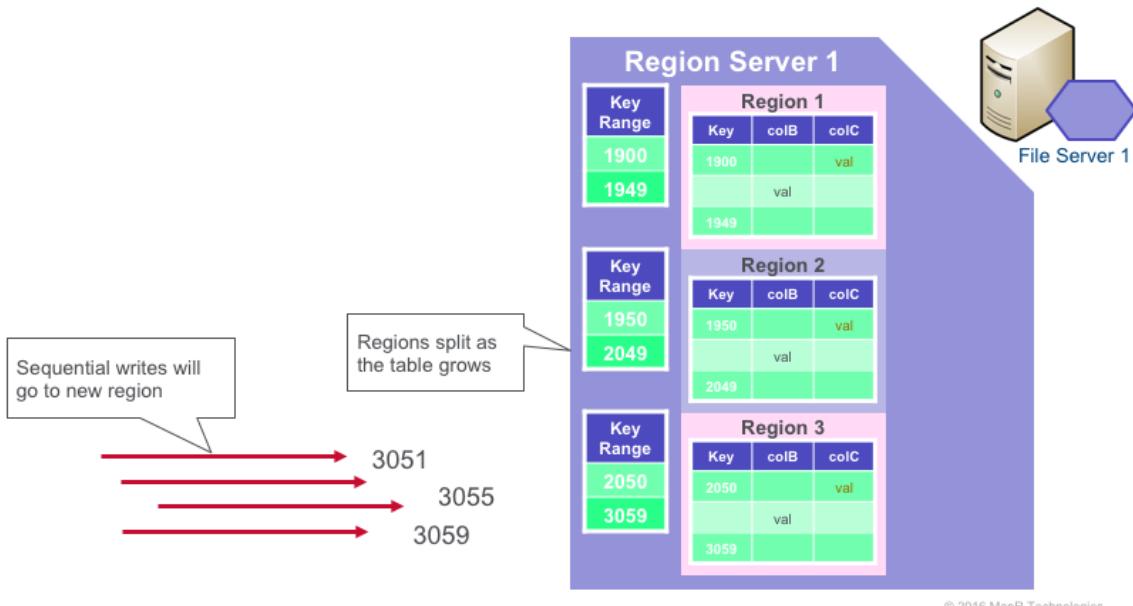
All sequential writes now go to region 2.

## Hot Spotting and Region Splits



The table in region 2 gets larger and eventually region 2 will split.

## Hot Spotting and Region Splits



Region 2 then splits into two with all the data from region 2 being divided equally between region 2 and region 3.

Since records in HBase tables are stored in lexicographical order, using a sequential generation method for row keys can lead to a hot spot problem, where all Puts will "hot spot" to the one "hot" server, and no other servers will ever take Puts. In addition to concentrating activity on a single region, all the other splits remain at half their maximum size. This destroys the advantages of using a distributed architecture.

Note that with MapR-DB, the cluster handles sequential keys and table splits to keep potential hot spots moving across nodes, decreasing the intensity and performance impact of the hot spot.

## Hot Spotting Summary

- Caused by **row keys** that are *written* in **sequential** order
  - Example: row keys written in order, 0000, 0001, 0002, 0003...
- All writes go to only one server at a time
- **Bottlenecks write** performance
- Results in **inefficient splitting**
- Regions fill to half the maximum, but never more

Hot spotting is an issue that occurs when row keys are written in monotonically increasing (or decreasing) order. Since data is written in sorted order, if the keys only increase with time, it means that every key is written to the end of a table. The end of the table will be contained in a single region, served by a single fileserver. Therefore, all Puts will "hot spot" to the one "hot" server, and no other servers will ever take Puts.

Hot spotting also results in inefficient region splitting, because after a table splits when the table grows too large for a single region, the first half of the table never gets more data; it remains forever at half the maximum potential size for a container.

Hot spotting isn't an issue for all applications. For example, if row key leads with a last name and names arrive in random order, then row keys will naturally spread over A-Z. On the other hand, writing time-series data has a high potential to hot spot, if the timestamp is used at the head of the row key.

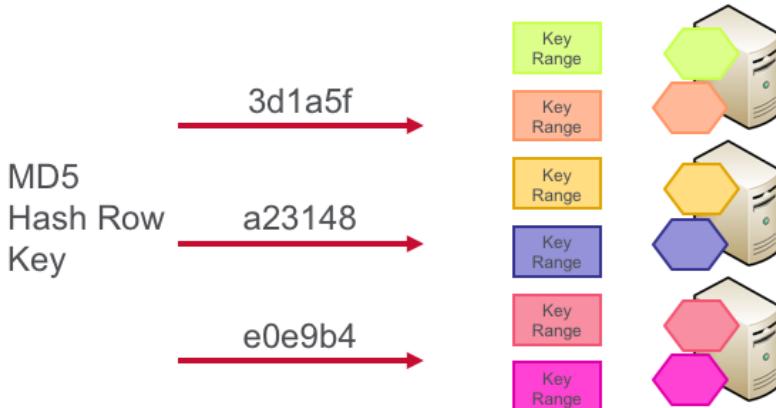
There are ways to prevent hot spotting, which we will look at next.

## 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));
```



Random writes will go to different regions, if the table was pre-split or big enough to have split. Hashing a sequential row key will create a random key which will distribute the writes, however random row keys are not good for scanning.

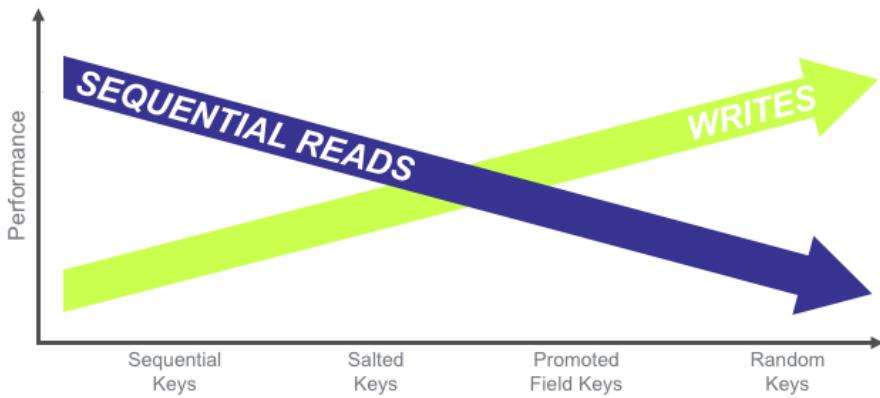
The java code here gets an instance of the `MessageDigest` class with the MD5 algorithm. This class provides hash algorithm function which takes a byte array and outputs a fixed-length hash value.

```
d = MessageDigest.getInstance("MD5");
```

Here the `digest` method is called and it returns a fixed length random hash of the input:

```
byte[] prefix = d.digest(Bytes.toBytes(s));
```

## Sequential vs. Random Keys



Source: Lars George's HBase: The Definitive Guide.

This diagram shows that random row keys are better for writing, but sequential row keys make it easier and faster to scan ranges of data.

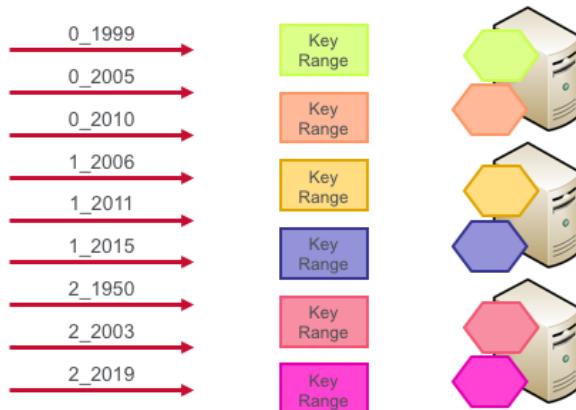
A way to prevent hot spotting is to "spread" or "spray" the row keys over multiple regions for consecutive writes. Hashing the row key distributes the row keys over regions which improves write performance but this makes scanning continuous ranges of data impossible based solely on the row key.

Solutions in between exist, shown in the middle of this diagram, which we will look at next.

## Salt the Row Key

Prefix the row key with a Salt:

```
byte salt=(byte) (Long.hashCode(timestamp)% numb_Region_serv
byte[] rowkey = Bytes.add(salt, timestamp);
```



MAPR Academy

Source: Lars George's HBase: The Definitive Guide.

© 2016 MapR Technologies

L4-38

To guarantee a spread or region row ranges across all region servers, you can prefix the row key with a salt. You can generate a random salt number by taking the hash code of the timestamp and taking its modulus with some multiple of the number of region servers:

```
int salt = new Integer(new Long(timestamp).hashCode()).shortValue() %
<number of region servers>
```

This involves taking the salt number and putting it in front of the timestamp to generate your timestamp:

```
byte[] rowkey = Bytes.add(Bytes.toBytes(salt) \
+ Bytes.toBytes("_") + Bytes.toBytes(timestamp));
```

Now your row keys are something like the following:

0\_timestamp1

This will distribute regions based on the first part of the key, which is the random salt number. Data for consecutive timestamps is distributed across multiple regions. The disadvantage to this is reads now involve distributing the scans to all the regions and finding the relevant rows. A better solution is to prefix with a field key or a hashed field key which we will look at next.

This is example code for salting from HBase the definitive guide:

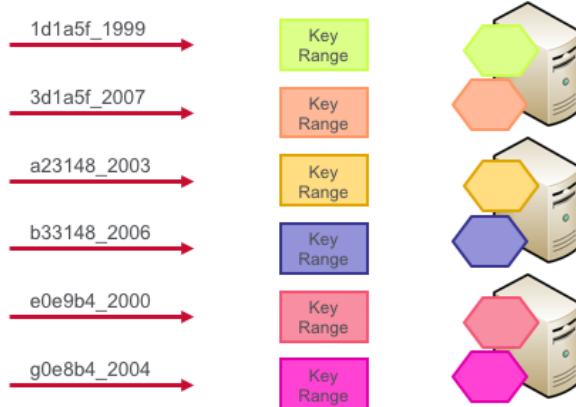
```
byte salt = (byte) (Long.hashCode(timestamp) % <number of region
servers>);
byte[] rowkey = Bytes.add(salt), timestamp);
```

## Prefix with a Hashed Field Key

- Prefix the row key with a (shortened) hash:

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

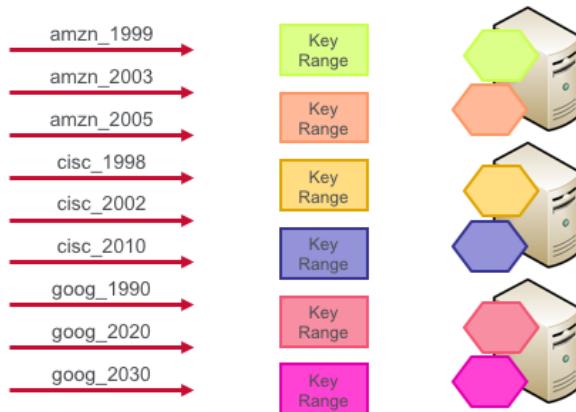
MD5 Hash  
Prefix Row  
Key



A better solution than hashing the whole key or a random salt is to prefix the row key with a shortened hashed field key. You can create a prefix hash based on the original key value so that when you want to get or scan you can calculate the hash, as opposed to searching each salted region.

## Prefix, Promote a Field Key

- Prefix or promote identifying/searchable value to front of key



You can “prefix” or promote an identifying or searchable value to the front of the row key in order to spread rows across region servers. This makes it easy to scan by that value. For example in the stock application, if we want to scan by company name then we can put the company name in front of the timestamp, this will distribute the key ranges by company name and you can scan by date after the company name.

## Solutions to Prevent Hot Spotting

### Spread out consecutive row keys

Hash row key with SHA-1 or MD5

Prefix with a Hash

Prefix, promote an identifying attribute

Raw Data	Hash	Hash-prefix	Promote prefix
1234	7427567902	7427_1234	cSCO_1234
1235	0467257887	0467_1235	amzn_1235
1236	6229078211	6229_1236	goog_1236

This table shows the three solutions to hot spotting that we discussed.

The solution for hot spotting is to "spread" or "spray" the row keys over multiple regions for consecutive writes. As with any design decision, there are trade-offs involved, and choosing any one of these methods adds complexity to your application.

For example, scanning continuous ranges of data becomes impossible if you hash the complete row key. Partial solutions exist, such as prefixing, which sprays the data over regions, but within each region data is sorted and can be scanned in order.

## Consider Access Patterns for Application

Which trade data needs fastest access (or most frequent)?

- **Row key ordering**

What if you want to retrieve the stocks by symbol and date?

- **Scan by row key prefix Increasing time: STOCKSYMBOL\_TIMESTAMP**

SYMBOL	+	timestamp				
Row Key						
AMZN_1391813876369						
AMZN_1391813876370						
GOOG_1391813876371						



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

Let's go back to our stock trades example. Thinking again about access patterns, we ask the following questions:

- Which trade data needs fastest access (or most frequent)? → (\*) These factors drive at row key ordering and possibly design of the timestamp.
- What if you want to retrieve the stocks by symbol and date?
  - Then you would compose the key with the stock symbol on the left followed by the timestamp: symbol\_timestamp which is shown in the table.
  - Note that the timestamp is increasing over time so that the oldest (by symbol) are ordered at the top.
- What if you usually want to retrieve the most recent? Then you would want the youngest at the top instead of the oldest.

Let us take a look at how to do this next.

## Last In First Out Access: Use Reverse-Timestamp

Row keys are sorted in increasing order

For fast access to **most-recent** writes:

- Design composite row key with **reverse-timestamp** that **decreases** over time.
- Scan by row key prefix Decreasing: [MAXTIME-TIMESTAMP]
  - Ex: `Long.MAX_VALUE-date.getTime()`

SYMBOL	+	Reverse timestamp					
AMZN							
AMZN							
GOOG							



\*`Long.MAX_VALUE = 263 -1`

Recall that row keys are sorted in increasing order. So, if you want to retrieve the most recent, you would want the youngest at the top instead of the oldest. Since the scan order of row keys is from lowest-to-highest, you might choose to insert the **MOST RECENT** items with the **LOWEST** row key.

Here's a tip for improving access time to the most recently-written data, if last-in-first-out is your predominant access pattern.

Design a composite row key that appends the reverse timestamp to the end of the key. In this example, we append the reverse timestamp to the right of the stock symbol. When you scan by row key, you will get the most recent trade first for each stock symbol.

Just as an FYI: `Long.max_value` is a constant that holds the maximum value a long can have – i.e.  $2^{63} -1$ .

## Row Key Reversal

- Reverse to get most significant search parts first
- To group for scans
- Example reverse domain for key
  - com.mapr.doc
  - com.mapr.blog
  - com.mapr.www
  - gov.whitehouse.www
  - org.redcross.blog
- Scans can use partial keys for example
  - Scan “com.mapr”

Since rows are stored in sorted order, you can affect the results of the sort by changing the ordering of the fields that make up the composite row key. When designing a composite key, consider how the data will be queried during production use. Place the fields that will be queried the most often towards the front of the composite key.

## Knowledge Check



## Knowledge Check



Select all the statements that are true of row keys:

- a) Primary index for the table
- b) Row key design directly impacts performance
- c) A row key cannot be changed after writing
- d) All of the above

Answer: D

## Knowledge Check



When would you use a composite row key?

- 1) To bound scans
- 2) To provide sub-indexing

Answers: 1 and 2

## Knowledge Check



Which of the following statements are true of hot spotting?

- 1) It is caused by row keys written in sequential order all going to one region server
- 2) Regions only fill to half capacity
- 3) It is caused by row keys written in random order
- 4) It causes write bottlenecks

Answers: 1, 2, and 4.

## Knowledge Check



Some cures for hot spotting include:

- 1) Salting the row keys
- 2) Promote an identifying attribute
- 3) Using prefixes
- 4) All of the above

Answer: 4

## Lab 4.2: Import Data with Different Row Key Designs



Open your lab guide to perform Lab 4.2.

## Learning Goals



## Learning Goals

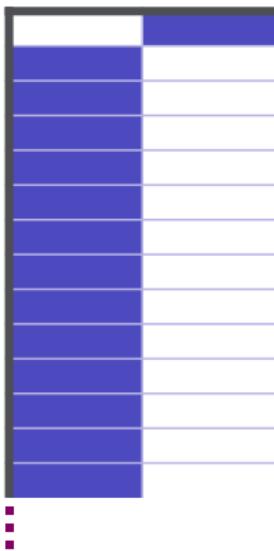


- 4.1 List the Elements of Schema Design
- 4.2 Design Row Keys for Data Access Patterns
- 4.3 Design Table Shape and Column Families for Data Access Patterns**
- 4.4 Define Column Family Properties
- 4.5 Design Schema for Given Scenario

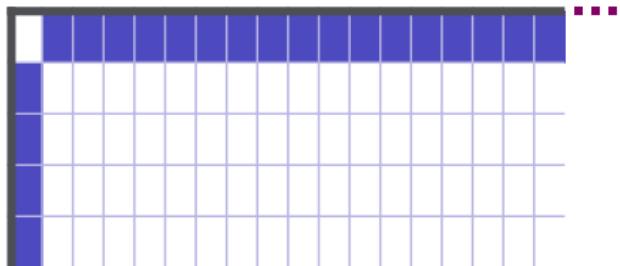
In this section, we will take a look at the HBase table design, including tall vs. wide tables. We will also see how row key design and column family definition play a part in the table shape.

## Tall or Flat Tables

Tall Narrow



Flat Wide



© 2016 MapR Technologies

L4-53

Design of the row key impacts the shape of HBase tables. A table may grow "tall-narrow," where most new entries get a unique row key. Tall tables typically use a composite row key, storing multiple pieces of data in the row key itself.

By contrast, HBase tables can also grow "flat-wide," where many data elements share the same row key and store data across many columns.

Note that growing horizontally is not possible in a typical relational database, because columns are predefined. A relational table with many columns and few rows might look flat, but it can only grow "taller" as data is added.

## Consider Access Patterns for Application



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

- Columns

- Column names are dynamic, not pre-defined
  - Every row does not need to have same columns

© 2016 MapR Technologies

L4-54

Consider again the access patterns for our stock trades example.

Are Price and Volume data typically accessed together, or are they unrelated? These factors drive a column family design. You want to keep together in the same column family those data that are accessed together.

## Tall Table for Stock Trades

Row key format: SYMBOL + Reverse timestamp

Row Key	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
...		

Ex: AMZN\_98618600888  
Group data that will be read & stored together

MAPR Academy © 2016 MapR Technologies L4-55

Here is a tall-narrow implementation as a possible solution for our stock trades example. To give us some direction, let's say that we want to be able to pick a company and read back all trades for a span of time.

In the design shown here:

- We use a composite row key combining the stock ticker symbol and a reverse timestamp.
- We have only one column family, called CF1, and we use exactly two columns: Price and Vol.
- We don't have to include columns for the trade time or the stock symbol, because this data is embedded in the row key.
- Every trade is stored with a new row key, so a single Get operation returns data for a single trade.
- It's clear that this table will grow tall as new trades come in. And because the timestamp in the row key is reversed new trades for a particular company get inserted higher in the list. Scans will retrieve the most recent data first.
- Does this design hot spot? No. For a given company, say Amazon AMZN, row keys arrive in decreasing order, so it looks like hot spotting might occur. However, trades for all companies A-to-Z are happening concurrently in random order. These timestamps are distributed by the company symbol. So, as shown here, two trades arrive at the same millisecond, but they're not necessarily hitting the same region server.

What are limitations of this implementation?

- One limitation is that row keys are only unique per company and timestamp. This presumes that a company cannot have two trades in the same millisecond.
- Since all row keys start with the ticker symbol, it would be inefficient to scan for a time range across multiple companies.

In our stock trades example, in a tall configuration, new stock trades will each get a new row key and grow the table taller. In this tall schema, every row represents one trade. There is only one column family, with two columns to store Price and Volume values. The composite row key is formed by combining the stock symbol and a reversed timestamp, (`Long.MAX_VALUE - timestamp`). For example: AMZN\_98618600888. Because the row key contains timestamp data, the trade time is not stored anywhere else in the table.

## Consider Access Patterns for Application

Are Price and Volume data typically accessed separately?

- *Column family structure*
- Column Families
  - Separate data that will be not be read together
- Columns
  - Column names are dynamic, not pre-defined
  - Every row does not need to have same columns

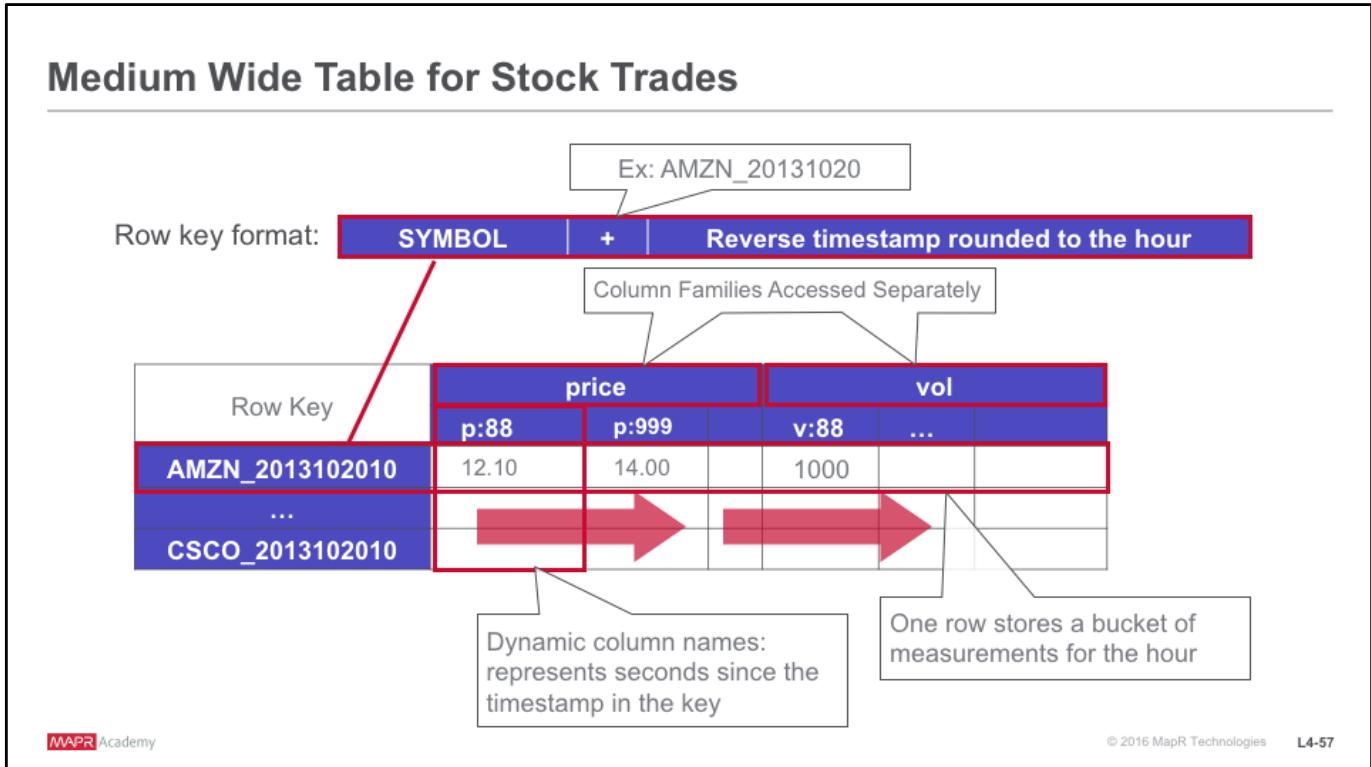


© 2016 MapR Technologies

L4-56

Consider the access patterns for our stock trades example. If Price and Volume are not typically accessed together then we should put them in separate column families.

We are now also going to consider dynamic column names. We will look at the resulting table shape if we use dynamic columns.



In this flat medium wide schema, all trades for an hour are stored in a single row. The row key is a composite of the company symbol and the date and hour, formatted YYYYMMDDHH. For example: AMZN\_20131020. The column name or qualifier is the seconds since the hour and is dynamic. Price and Volume values are stored in separate column families. Each column cell represents one trade.

We are essentially segregating time into buckets. Time is rounded to the hour in the row key. One row stores a bucket of measurements for an hour.

## Consider Access Patterns for Application



- Column Families
  - How many Versions?
    - *Max Versions*

© 2016 MapR Technologies

L4-58

Thinking about the access patterns for a third example schema for our stock trades example, how many versions do we wish to keep?

## Wide Table for Stock Trades

Row key format: **SYMBOL** + **date YYYYMMDD**

Ex: AMZN\_20131020

The diagram illustrates the structure of a wide table for stock trades. It shows a row key format: SYMBOL + date YYYYMMDD. An example is given as AMZN\_20131020. The table has a Row Key column and three Column Family (CF) columns: CF price, CF vol, and CF stats. The CF price column contains price data (e.g., price:00, ..., price:23). The CF vol column contains volume data (e.g., vol:00, ..., vol:23). The CF stats column contains daily high and low price information (Day Hi, Day Lo). Red arrows point from the row key components to the corresponding parts in the table. Callouts explain: 'Date in the row key' points to the date part; 'Hour in the column name' points to the hour part in the price column; 'Separate price and volume data into column families' points to the two distinct CFs; and 'Set Column Family to store Max Versions, timestamp in the version' points to the CF stats column.

Row Key	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	23.01			1000				

We'll use the same assumption that we want to be able to pick a company and read back all trades for a span of time. In this flat wide schema, all trades for each day are stored in a single row.

In this design:

- This time we use a composite row key formed from the stock symbol and the date of the trade.
- We use separate column families for price and volume. This means that we could easily scan for just price, without having to waste disk cycles reading volume data.
- The stock symbol is embedded in the row key, so we don't have a column for it.
- We want to be able to scan a company for a particular time range. This implementation segregates time into buckets:
  - The date of a trade is stored in the row key. This means that a single Get operation on a row reads back a whole day's worth of trades.
  - We store the hour of the trade as the column name, from 00 to 23. This allows us to return a specific hour of trades by specifying a column.
  - As the hours of the day go by, you can see that the row will grow wider.
  - In this example, the column families Price and Vol are set to store Max Version. Every version of a cell represents one trade, and the cell version (a long) stores the timestamp of the trade in milliseconds. Within a specific hour, as trades are written, we write more cell versions into the hour column.
- Does this design hot spot? No. For the same reasons as the tall-narrow example, prefixing the row key with the company name prevents hot spotting.

Since each row represents a single day's worth of trades, we might also want to store some useful information about the day right there on the same row. For example, we could create a column family for daily statistics and keep track of the daily high and low price for a company, or the total trade volume.

What are limitations of this implementation?

- Like our previous example, this model cannot represent two trades in the same millisecond for one company.
- And here too, it would be inefficient to scan for trades for all companies within a time range.
- In this implementation, because price and volume are stored in separate column families, the lists of versions become separate maps. The version timestamp acts as an index to correlate each price to a volume. This introduces challenges in the code to manage the values as a unit.

## 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

In a tall-narrow table we achieve better query granularity since the row key is rich and we can query by row key.

Put operations at a row level are atomic and you leverage that in a flat-wide design. So you give up atomicity with a tall table to gain performance benefits.

Accessing wider rows is more expensive than accessing smaller ones, because the row key is the dominating component of indexes. Knowing the row key is what gives you all the benefits of how HBase indexes under the hood, but you give up atomicity in order to gain performance benefits that come with a tall table.

MapR tables split at the row level, not the column level. For this reason, extremely wide tables with very large numbers of columns can sometimes reach the recommended size for a table split at a comparatively small number of rows. In general, design your schema to prioritize more rows and fewer columns.

## Knowledge Check



## Knowledge Check



A Tall-Narrow design (select all that apply):

- 1) Provides better query granularity
- 2) Supports built-in row atomicity
- 3) Works well with Scan operations
- 4) Works well with Get operations

Answers: 1 and 3

## Knowledge Check



A Flat-Wide design (select all that apply):

- 1) Provides better query granularity
- 2) Supports built-in row atomicity
- 3) Works well with Scan operations
- 4) Works well with Get operations

Answers: 2 and 4

## Lab 4.3: Populate and Examine Trades Tall and Flat Tables



Proceed to your lab guide to perform Lab 4.3.

## Learning Goals



## Learning Goals



- 4.1 List the Elements of Schema Design
- 4.2 Design Row Keys for Data Access Patterns
- 4.3 Design Table Shape and Column Families for Data Access Patterns
- 4.4 Define Column Family Properties**
- 4.5 Design Schema for Given Scenario

In this section, we will take a look at the properties that we can define at the column family level.

## Consider Access Patterns for Application



- Column Families

- Group data that will be read and stored together
- Can set attributes:
  - # Min/Max versions, compression, in-memory, Time-To-Live

© 2016 MapR Technologies

L4-67

Looking again at the questions about our stock trades example:

- Are Price and Volume data typically accessed together, or are they unrelated? ➔ (\*) These factors drive a column family design. You want to keep together in the same column family those data that are accessed together.
- Do all trades need to be saved forever? ➔ (\*) This factor drives at purging past cell versions by setting Time-To-Live (TTL) on a column family.

## Grouping Data in Column Families

Keep data that is **accessed** together in one column family

- Example: Keep together LASTNAME, FIRSTNAME
- Example: Consider separating INVENTORY QUANTITY, VENDOR ADDRESS

Keep data that **compress well** together in same column family

- Text-only cell data compress well together
- Mixed binary (example: PNG data) and text don't compress well together

As stated before, data that is accessed together should be kept together in one column family.

Here is an additional guideline to follow: data that compresses well together should be in the same column family. Since text-only cell data compresses well together, place those columns in the same column family. On the other hand, mixed binary and text do not compress well together, so don't place these columns in the same column family.

## Consider Access Patterns for Application



Do all trades need to be saved forever?

- *TTL Time-to-Live, column family can be set to expire cells*

How many Versions?

- MaxVersions

© 2016 MapR Technologies

L4-69

Continuing with our analysis of the stock trades example, here are some questions regarding our data access patterns.

- Do all trades need to be saved forever? This factor drives at purging past cell version by setting Time-To-Live (TTL) or MaxVersions on a column family.
- How many version do we want to keep? The default value for max versions used to be three, but is now one.

## Column Family Properties

### Compression

- MapR supports LZ4, LZF, and ZLIB
- LZO and SNAPPY (Apache HBase only. Uses LZ4 in MapR.)

### Time To Live (TTL)

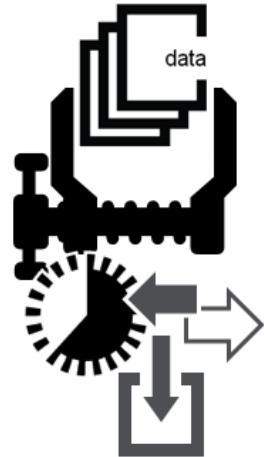
- Keep data for some time and then **delete** when TTL is passed

### Versioning

- Keeping fewer versions means less data in scans. **Default is now one.**
- Combine **MIN VERSIONS** with **TTL** to **keep data older than TTL**.

### In-Memory setting

- A setting to *suggest* that server **keep data in cache**. Not guaranteed.
- Use for **smaller, high-access** column families.



Here are properties you can enable at the column family level that affect behavior of a table. These properties provide tuning mechanisms you should consider when you design your system.

- Compression - Compression is usually turned on when creating tables and is a good choice most of the time. You can specify the compression algorithm you want for each column family at table creation time. MapR supports LZ4, LZF, and ZLIB. If you specify LZO or SNAPPY (which are valid choices in the Apache HBase API), MapR will use LZ4 under the hood.
- TTL – Not every stored value needs to live forever. You can specify a Time-To-Live attribute so that data older than the TTL will be deleted.
- Versioning - How many versions for each cell you keep will affect queries and how much data is read from disk. The default used to be three versions for each cell, now the default is one. If you don't need to keep old values, leave the default to one version so that multiple updates will not store multiple versions.
  - You can also specify the minimum versions that are stored for a column family using the **MIN VERSIONS** parameter. This can be used together with the TTL attribute. When all versions currently stored are older than the TTL, at least the **MIN VERSIONS** number of values will be retained.
- In-Memory – This setting is a suggestion that the server keep a column family's data in memory for fast retrieval. This works well for column families with few columns, like a map between username and password, or column families with sparse data. The in-memory setting is not a guarantee that data will be kept in memory; it is merely elevated priority for data to be kept in memory.

## Summary

- Why design matters
- Data model review
- Designing schema for data access patterns
- Row key design
- Avoiding hot-spotting
- Designing column families
- Column family properties

In this lesson, we looked at the basics of schema design, its importance, design guidelines, and in particular, row key design.

We are now going to look at a real world example.

## Knowledge Check



## Knowledge Check



**Which column family attributes should be considered at design time in order to optimize performance (select all that apply):**

1. Triggers
2. Compression
3. Time-To-Live
4. Versions
5. Secondary Indexes

Answers: 2, 3 & 4

Column family name is also important, as it takes up space in the file storage and network.

## Learning Goals



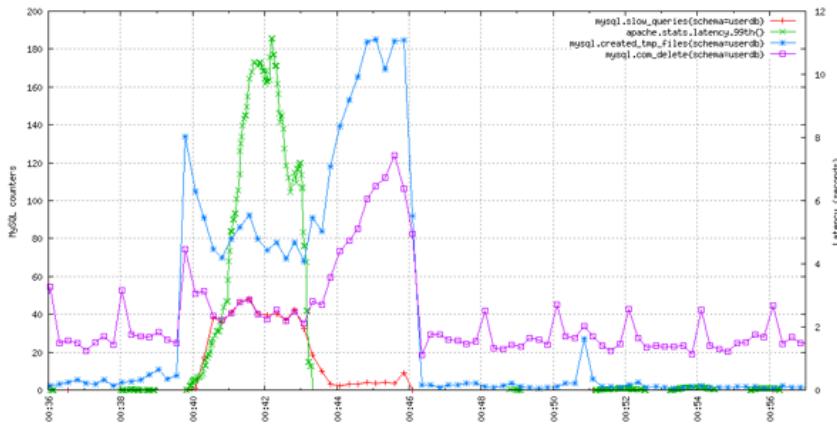
## Learning Goals



- 4.1 List the Elements of Schema Design
- 4.2 Design Row Keys for Data Access Patterns
- 4.3 Design Table Shape and Column Families for Data Access Patterns
- 4.4 Define Column Family Properties
- 4.5 Design Schema for Given Scenario**

We are now going to take a look at how OpenTSDB has designed their schema.

## OpenTSDB



Time series infrastructure metrics data for monitoring  
Full schema at <http://opentsdb.net/schema>

OpenTSDB is an open source project. It is a Time Series Database (or TSDB) created to store, index and serve metrics from computer systems, like network gear, operating systems, and applications, at a large scale. It's useful for fine-grained, real-time monitoring. Existing systems today handle tens of billions of data points per day. The data can be used for charts like this one to show correlations in behavior between complex systems.

OpenTSDB provides a good example of row key design for time series data. It is also a good study of row key design in general, because of clever composition of data elements into the row key. You can read more online at [opentsdb.net/schema](http://opentsdb.net/schema).

A time series is a series of numeric data points of some particular metric over time. A metric is any particular piece of data, like hits to an Apache hosted file, that you wish to track over time. Each time series consists of a metric plus zero or more tags associated with this metric. Tags are name value pairs which further identify a metric, for example, host=hostname

Here is the use case situation: Need to record values of hundreds of metrics from thousands of machines sampled every few seconds.

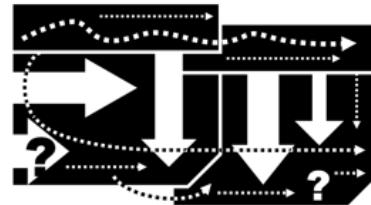
Here is some sample data:

```
[metric name] [timestamp] [value] [tag name=value]
http.hits      1234567890  34877  node=n1
```

## OpenTSDB Modeling Questions

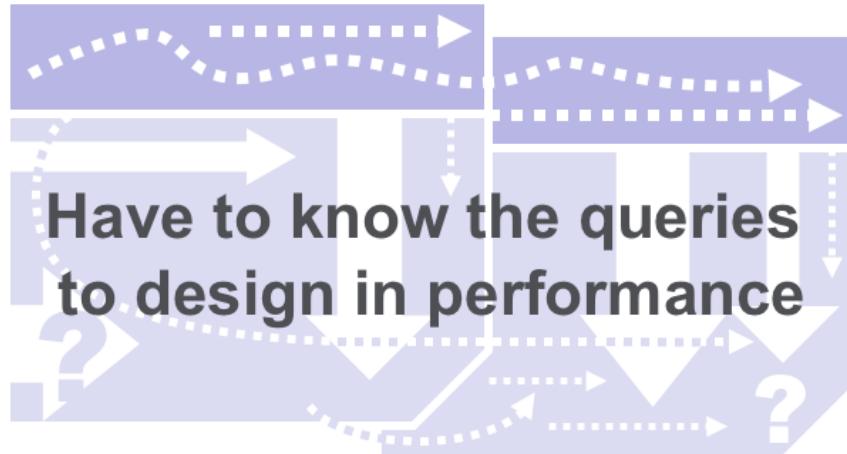
### Schema modeling questions:

- What was response time during peak period?
- Which hosts had problematic response time?
- How many http hits/hour? On which hosts?
- When was peak traffic (hits)?
- What happened with response time during peak traffic?
- When was the system down?
- How many requests/hour?



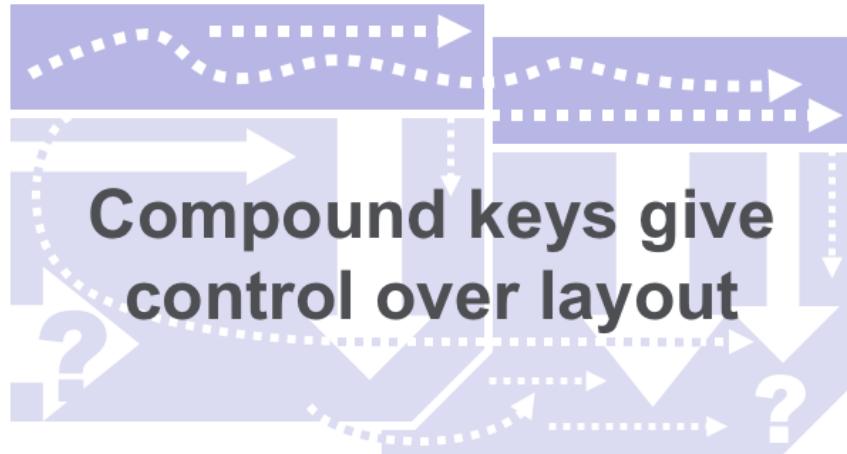
This shows some example data and the queries or questions we would like to retrieve from the data. For example: What was the response time during peak periods? How many http hits/hour? When was peak traffic or hits? And so on.

## Design Guidelines



To reiterate, you have to know the questions in order to design for performance, such as what information are you going to retrieve?

## Designing Row Keys



One of the first things we need to do is design the row key. Recall that using compound keys will give us control over layout. We will take a look at a few row key design choices next.

## Key Composition #1

Row key format: Time | + | Metric | + | Tags

Time	Metric	Tags	Value
10667	http.hits	host=n1	3400
10667	http.resptime	host=n1	10
10668	http.hits	host=n2	2400
10668	http.resptime	host=n2	5
10727	http.hits	host=n1	4000
10727	http.resptime	host=n1	20

Samples written in sequential order

Could this lead to any issues?

YES → Hot-spotting

The row key format here is a composite key consisting of Time, Metric, and Tags in this order.

This means that the sample data will be written in sequential order. Could this lead to any issues? Yes – remember that using sequential keys can cause hot spotting.

## Key Composition #2

Row key format: **Tags | + | Metric | + | Time**

Tags	Metric	Time	Value
host=n1	http.hits	10667	3400
host=n1	http.hits	10727	1000
host=n1	http.resptime	10667	10
host=n1	http.resptime	10727	9
host=n2	http.hits	10668	2000
host=n2	http.hits	10727	1000

Samples for same host  
ordered/grouped together

Queries require data for same  
metric, but different hosts

In this design, the row key is again a composite key consisting of Tags, Metric, and Time in that order. Tags are name value pairs which further identify a Metric, for example, host = hostname. Samples for the same host will be ordered and grouped together.

This row key design would not have a hot spotting problem. However, queries commonly require data for the same metric across different hosts, so having the data grouped by host is not ideal. Let us take a look at a third option.

## Key Composition #3

Row key format: Metric | + | Time | + | Tags

Metric	Time	Tags	Value
http.hits	10667	host=n1	3400
http.hits	10668	host=n2	3000
http.hits	10727	host=n1	2000
http.resptime	10667	host=n1	10
http.resptime	10668	host=n2	5
http.resptime	10727	host=n1	15

All samples for same metric grouped together

Queries focus on one or few metrics

This row key is composed of the Metric, Time, and Tags in that order. This means that all samples for the same metric are grouped together. Queries commonly focus on data for the same metric or a few metrics at a time, so having the data grouped by metric type would make scans by metric type easier.

## Best Choice

Samples written in sequential order → hot spotting

Time	Metric	Tags	Value
10667	http.hits	host=n1	3400
10667	http.resptime	host=n1	10
10668	http.hits	host=n2	2400
10668	http.resptime	host=n2	5
10727	http.hits	host=n1	4000
10727	http.resptime	host=n1	20

Samples for same host ordered/grouped together

Tags	Metric	Time	Value
host=n1	http.hits	10667	3400
host=n1	http.hits	10727	1000
host=n1	http.resptime	10667	10
host=n1	http.resptime	10727	9
host=n2	http.hits	10668	2000
host=n2	http.hits	10727	1000

All samples for same Metric grouped together

Metric	Time	Tags	Value
http.hits	10667	host=n1	3400
http.hits	10668	host=n2	3000
http.hits	10727	host=n1	2000
http.resptime	10667	host=n1	10
http.resptime	10668	host=n2	5
http.resptime	10727	host=n1	15

Queries focus on one or few Metrics

Group desired data together – put most queried information on left most part of key

© 2016 MapR Technologies L4-84

The best choice in this case is number three, where the row key design groups the desired data together by putting most queried information on the left most part of the key.

Let us take a look at how OpenTSDB designed their row key.

This is how OpenTSDB has designed their row key. It is very similar to the third option that we saw – MetricID (instead of Metric + Time + Tags). The difference is that OpenTSDB shortened the Metric name and Tags to IDs in order to put a lot information in the row key without taking up too much space.

OpenTSDB's row key consists of:

- An ID for the event or metric, for example Mysql.aborted.connects
- Followed by a timestamp (rounded off to the hour)
- Followed by IDs for all of the tag names and values, for example host = mydb.example.com

This allows to scan by Metric, scan by Metric and Time, scan by Metric, Time, and Tags. Also regular expressions can be used with a row key filter.

Here is a tip for row key design: Add Tags, key-value pairs, to end of key for additional information for Scans.

## Tall or Wide?

Metric	Time hour	Node	+17	+77	+137		
http.hits	13:00	host=n1	100	10	50		
						-----	→

Filtering overhead is non-trivial ...  
wide has to filter fewer rows

Did OpenTSDB pick tall and skinny OR wide and flat? This is determined by the row key design as well as how column families and columns are defined.

In the tall table example on the left the timestamp in the row key is to the millisecond, there is one row per cell value. The tall table will grow taller because each value inserts a new row.

In the medium wide table example on the right the timestamp in the row key is rounded off to the hour, which means one row will bucket an hour of metric values. The dynamic column name is the offset in seconds from the hour in the row key.

In the wide table, the scan has to filter fewer rows, the tall table would have too much filtering overhead. OpenTSDB chose the wide solution on the right.

Note that schemas can be very flexible and can even change on the fly.

## OpenTSDB Unique ID Table

Unique ID table stores unique IDs for metrics

These IDs are used in the main tsdb table

Each new ID creates two rows: name-to-ID, ID-to-name

Metric IDs, tag name IDs, tag value IDs.

metric name eg. myservice.latency.avg

Tag name eg. host

Tag value eg. mydb.example.com

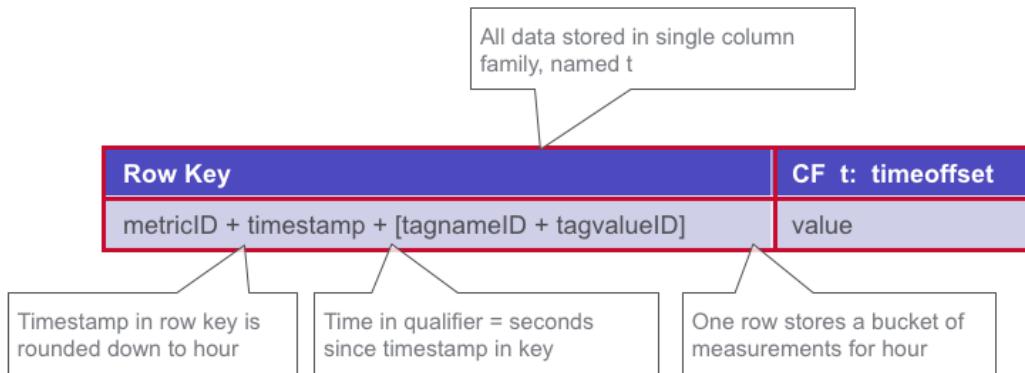
Metric name	metricID
MetricID	Metric name
Key	ID

In order to compress the amount of data stored for each data point, OpenTSDB creates a lookup table which maps metric and tag names to Unique IDs.

The Unique IDs are of a fixed 3-byte width and are used to lookup the name associated with the IDs used in the row key.

Next, we will show the OpenTSDB key and columns in a little more detail.

## OpenTSDB Table Schema



As mentioned earlier, a lot of data is stored in the row key. The timestamp in row key is rounded down to the hour. There is one column family t. The column name is the seconds since the timestamp in the key. One row stores a bucket of measurements for the hour.

OpenTSDB provides maps for Metric ID and Tag IDs, to compress the amount of data stored in the row key. This makes row keys less readable, but also makes them much smaller for data that repeats often.

## OpenTSDB Between Tall and Wide

Example row stores a bucket:

Row Key	t:10	t:100	t:125
idts1	38	40	50
idts2			

Timestamp in row key is rounded down to hour

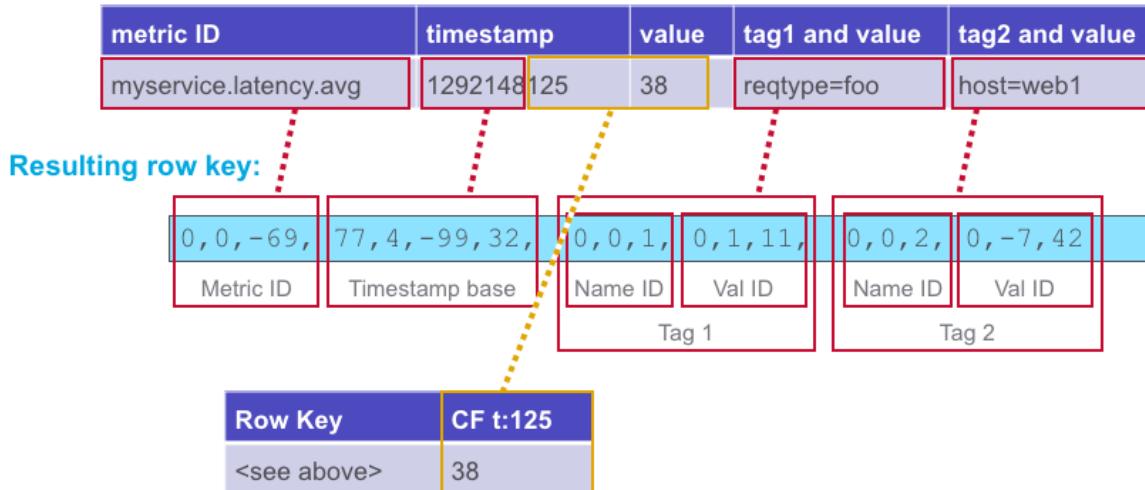
Time in column represents seconds since timestamp in key

One row stores bucket of measurements for hour

The middle path between tall vs. wide is packing data that would be a separate in rows into columns, for certain rows. OpenTSDB is the best example of this case where a single row represents an hour time-range and discrete events in that time range are stored as columns. This has the advantage of being I/O efficient.

## Example Data Point

### Example data point:



Let's look at an example data point to get a clearer picture of how this works.

The diagram shows which elements of the data point get embedded into the row key, the column name, and the cell value itself. Notice that for the tag data, this schema effectively nests a data structure inside the row key. This nesting of structures is a common design pattern for HBase. The row key, column names, and values are all variable-length byte arrays, and can contain whatever data you want, as long as your application can interpret the values.

In the example shown, the full data point timestamp is 1292148125, and the row key will store the base value 1292148000. The column name will provide the offset 125.

The Metric ID for myservice.latency.avg is [0, 0, -69].

The Tag name ID for reqtype is [0, 0, 1], and the tag value ID for foo is [0, 1, 11].

The Tag name ID for host is [0, 0, 2], and the tag value ID for web1 is [0, -7, 42].

The bytes for the timestamp are [77, 4, -99, 32] = 1292148000, rounded down to a 60 minute boundary.

Storing multiple observations per row lets filtered scans disqualify more data in a single exclusion. It also drastically reduces the overall number of rows that must be tracked by the Bloom Filter on row key.

As an exercise, can you think of a way to nest the tag metadata in the column name, instead of the row key?

## Summary

- Why design matters
- Data model review
- Design guidelines
- Row key design
- Avoiding hot-spotting
- OpenTSDB example

In this lesson, we looked at the basics of schema design, its importance, design guidelines and in particular, row key design. We concluded this lesson by taking a look at a real-world example. OpenTSDB provides a good example of composite row keys using IDs.

## Knowledge Check



## Knowledge Check



Which of the following are true about OpenTSDB (select all that apply):

1. HBase was started by the OpenTSDB project
2. It is not a very active open source project anymore
3. It is a good example of composite row key design
4. It is a good example of a medium wide design where a range of new data entries share the same row key

Answers: 3 and 4



## Next Steps

### DEV 325- Apache HBase Schema Design

Lesson 5: Design Schemas for Complex Data Structures

Congratulations. You have completed Lesson 4. In the next lesson, we will design schemas for complex data structures.



## DEV 325 – Apache HBase Schema Design

Lesson 5: Design Schemas for Complex Data Structures

Winter 2017, v5.1

Welcome to DEV 325, Apache HBase Schema Design, Lesson 5: Design Schemas for Complex Data Structures.

## Learning Goals



## Learning Goals



- 5.1 Transition From Relational Model to HBase
- 5.2 Use Intelligent Keys
- 5.3 Use Lookup Tables for Secondary Indexes
- 5.4 Design for Other Complex Data Structures
- 5.5 Evolve Schemas Over Time

In the previous lesson, you have seen how to define a schema to fit the HBase data model. In this lesson, we will look at transitioning from a relational model to HBase. We are also going to look at designing for nested entities, using secondary indexes, and designing for other complex data structures.

We are going to go into more detail about schema design, and using some examples, we will compare schema design for relational databases to schema design for HBase.

## Learning Goals



### 5.1 Transition From Relational Model to HBase

- 5.2 Use Intelligent Keys
- 5.3 Use Lookup Tables for Secondary Indexes
- 5.4 Design for Other Complex Data Structures
- 5.5 Evolve Schemas Over Time

In this section, we will compare the relational model to the HBase model and describe ways to transition from the relational model to HBase.

## Transition from Relational Model to HBase

### Denormalization

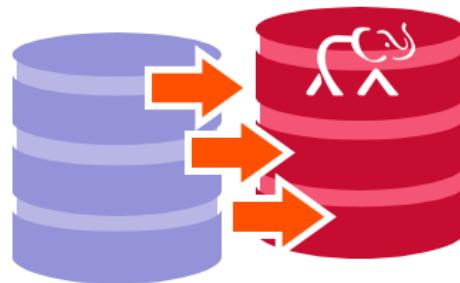
- Replace JOINS

### Duplication

- Design for reads

### Intelligent Keys

- Used for indexing, sorting
- Optimize reads



There can be situations where it would make sense to move a database defined for RDBMS to HBase. The following principles of denormalization, duplication, and using intelligence keys are commonly used when designing schemas for moving a relational model to HBase. These principles are discussed here.

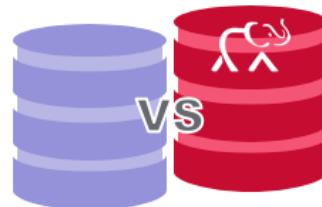
## Relational vs. HBase Schemas

### 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



Designing an HBase schema is different than designing a relational schema. There is no one-to-one mapping from relational databases to HBase.

In relational design the focus and effort is around describing the entity and its interaction with other entities. The queries and indexes are designed later. Recall that HBase is designed such that distributed data is accessed together, for clustering.

With HBase you have a query-first-driven schema design:

Unlike traditional relational schema modeling, all the possible queries should be identified first, and the schema model designed accordingly.

You should design your HBase schema to take advantage of the strengths of HBase. Think about your access patterns and design so that the data that is read together is stored together.

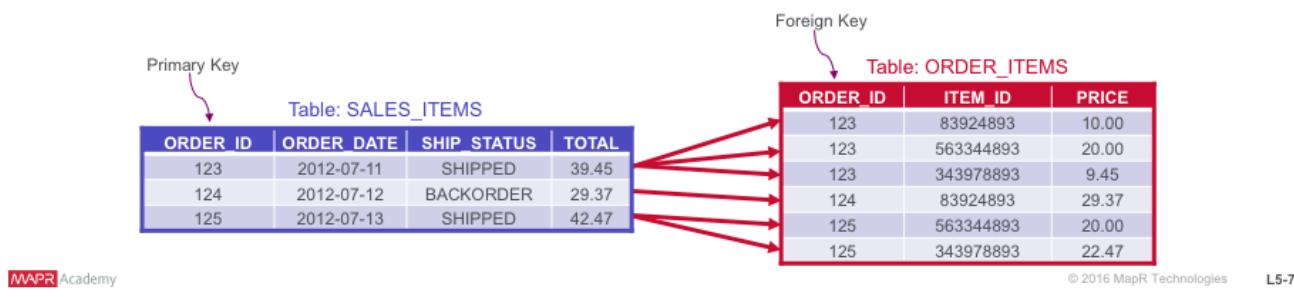
## Normalization

### Goal of 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



In a relational database, you normalize the schema to eliminate redundancy by putting repeating information into a table of its own. This has the following benefits:

- You don't have to update multiple copies when an update happens, which makes writes faster.
- You reduce the storage size by having a single copy instead of multiple copies.

However this causes joins. Since data has to be retrieved from more tables, queries can take more time to complete. In this example we have an order table which has a one-to-many relationship with an order items table. The order items table has a foreign key with the ID of the corresponding order.

## What Problem Did Normalization Solve?

Think back to the **70'S** for a moment...

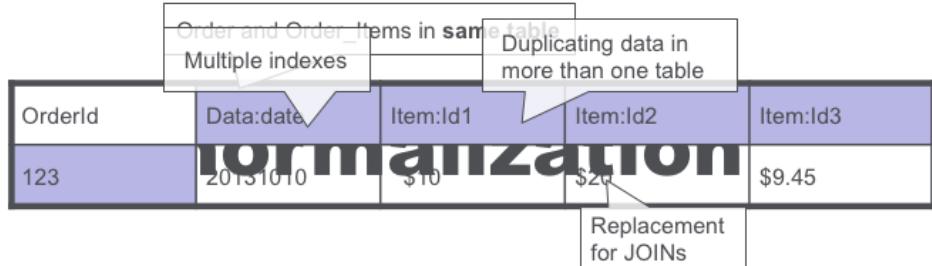


Data density has gone up by a factor of one million since the start of relational database management systems.

In support of denormalizing data, think back to the 1970's when the push toward normalization began. Storage was expensive and access was slow.

These economics don't apply anymore. Storage is cheap and HBase provides sparse tables that consume storage for cells only when you write to them.

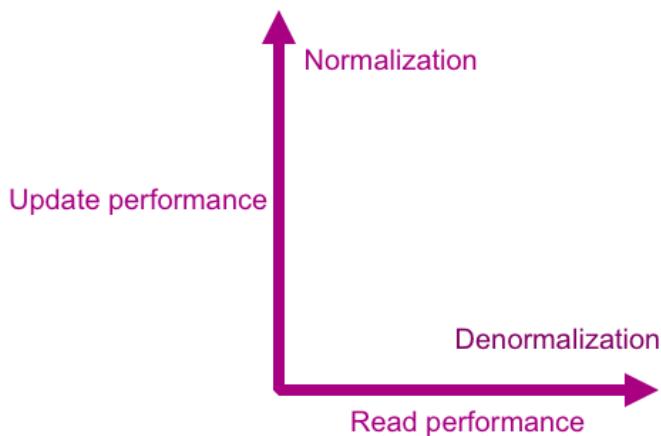
## Denormalization



In a denormalized datastore, you store in one table what would be multiple indexes in a relational world. Denormalization can be thought of as a replacement for JOINs.

Often with HBase you denormalize or duplicate data so that data is accessed and stored together. In this example, the order and related line items are stored together and can be read together with a get on the row key. This makes the reads a lot faster than joining tables together.

## Duplication



"Cloud Data Structure Diagramming Techniques and Design Patterns"  
by D. Salmen et al. (2009).

For big data to scale, you design schemas differently than for a relation database. A general principle is to denormalize schemas by duplicating data.

Why do we do this? We want to achieve one read per request if possible. To achieve this, group all data that is needed to process a query in one place. This often means that for different query flows the same data will be accessed in different combinations. Hence, we need to duplicate data which provides the ability to easily retrieve data for multiple querying patterns. This improves read performance. However, when we duplicate the data, it increases total data volume and update performance decreases.

You may want to duplicate data for:

- Materialization of views.
- To store data about an entity/entities related to it in the same table allowing for retrieval of data in one read operation, with no joins.

Reference "Cloud Data Structure Diagramming Techniques and Design Patterns" by D. Salmen et al. (2009).

## Denormalization Design Considerations

- Group related data to be read together
- Eliminate JOINs
- Duplicate data where necessary
  - Wide tables and column-oriented design
- Must have good key design
  - Compound keys are essential
  - Data partitioning based on keys

With HBase, the support for sparse, wide tables and column-oriented design often eliminates the need to normalize data to save space. Therefore JOIN operations are not needed to aggregate the data at query time. Furthermore, HBase was designed for horizontal scaling and provides no automatic JOIN capability. Thus designing your schema properly enables you to scale your application.

Use denormalization and duplicate data in tables that you would typically have to join in a relational database. Group related data so that it can be read together.

However, denormalization requires design consideration. As with all design decisions, there are trade-offs. If you have the same data replicated in multiple tables, it now becomes the responsibility of your application to maintain consistency or handle inconsistency gracefully.

## Schema Example – Social Application

The screenshot shows a web browser window displaying a list of posts from the 'TECHNOLOGY' subreddit on Reddit. The page title is 'www.reddit.com/r/technology/'. The interface includes a sidebar with 'MY SUBREDDITS' and various navigation links like 'FRONT - ALL - RANDOM | PICS - FUNNY - GAMING - ASKREDDIT - WORLDNEWS - NEWS - VIDEOS - IAMA - TO'. Below the sidebar, there's a 'TECHNOLOGY' icon and a 'hot' button. The main content area lists six posts:

- 1 1508 Solar switch forces utilities to shift priorities - "A gentleman came up and said, solar, and now the utility wants to hit me with a fixed charge," Guess I'll just put solar and get off the grid altogether.' That's the death spiral. That's what we do (m.sfgate.com)  
submitted 8 hours ago by pnawall  
363 comments share save hide report
- 2 2954 Cheapest 150Mbps broadband in big US cities costs 100% more than overseas  
submitted 15 hours ago by badcookies  
1304 comments share save hide report
- 3 531 Almost a third of Samsung Galaxy Gear smartwatches are being returned- The I (theinquirer.net)  
submitted 7 hours ago by millerdmb  
160 comments share save hide report
- 4 231 Thorium Energy Conference at CERN is webcasted, here is a life feed. (webcast.web...  
submitted 3 hours ago by t233  
8 comments share save hide report
- 5 160 A New form of Energy Production - Massive Energy Skyscrapers On U.S Mexico B...  
Out 500 MWs to Electric Grid (industrytap.com)  
submitted 3 hours ago by drfr  
27 comments share save hide report
- 6 171 Million lines of code. (informationisbeautiful.net)  
submitted 5 hours ago by haatee  
28 comments share save hide report

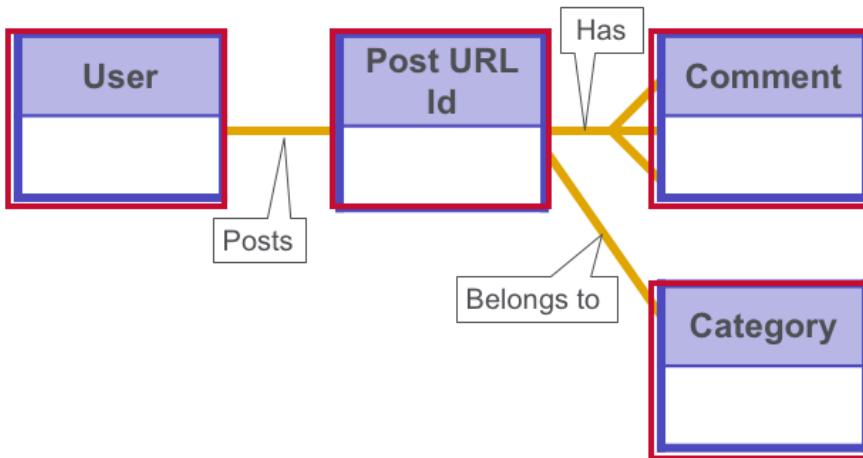
As a modeling example we will use a social app. Here are the use cases:

- Users can post URLs to articles by category (like news, sports...)
- Users can then make comments on the article

Some of the query requirements are:

- Display the posts by category and date (most recent first)
- Display the comments by post
- Display the posts by userID

## Entity Relationship Diagram



Let us take a look at an Entity Relationship Diagram.

The entities are:

- User, Post, Comment, Category

The relations are:

- A User makes a Post
- A Post has Comments
- A Post belongs to a Category

## Relational Model

Relational database modeling concepts

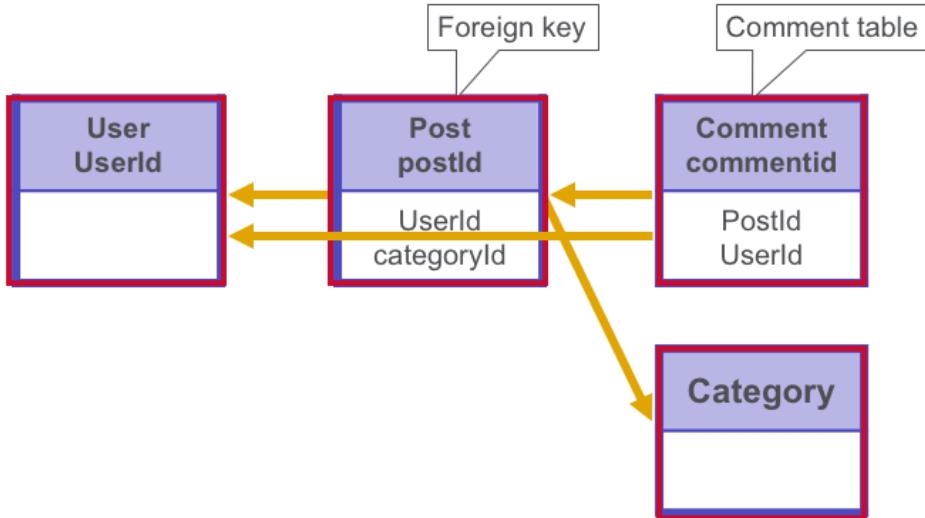
E-R	RDMS
Entity	Table
Attribute	Column
Relationship	Foreign key relationship

Relational database modeling consists of three primary concepts:

1. *Entities*—map to *tables*.
2. *Attributes*—map to *columns*.
3. *Relationships*—map to *foreign-key relationships*.

Later, you think about the queries and indexes.

## Relational Tables



This shows the Relational Model for our social application.

- Users are stored in the user table.
- The posted URL is stored in the Post table with a foreign key to the user that posted it, and a foreign key to the category for the post (Users can subscribe to another user's posts and/or subscribe to categories, so you need to be able to get the list of URLs for a category and for a User. This links post tables to the User and Category tables with a foreign key relationship).
- Comments about a post are stored in the comments table with a foreign key to the post and a foreign key to the user that commented.

## Social App Modeling Questions

Think about how the data is read!

Query requirements are:

- Display the posts by Category and Date (most recent first)
- Display the comments by Post
- Display the posts by UserId



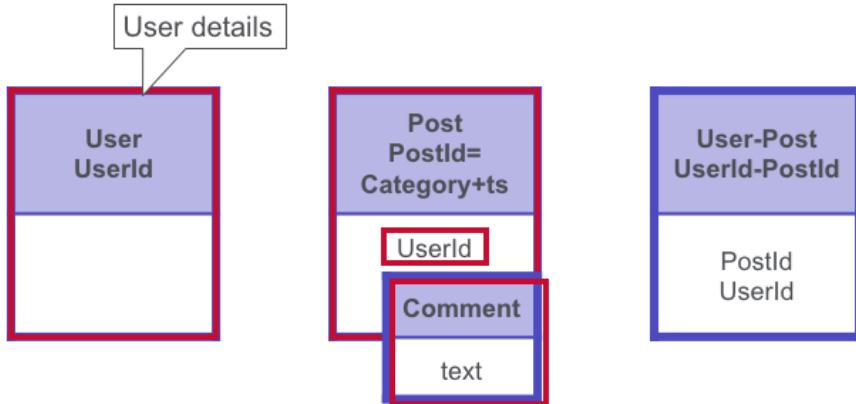
As we learned in the previous lesson, design your schema for the questions. In HBase design, the focus is around how the application will retrieve the data. Also think about how the data is read!

The query requirements for our social app are:

- Display the posts by category and date (most recent first)
- Display the comments by Post
- Display the posts by UserId

Next we will look at the HBase tables for our sample app.

## HBase Tables



There are various approaches to converting entity relationships to fit the underlying architecture of HBase. You could implement this example in different ways. This is how the schema could be represented in HBase:

- The user table stores the user details.
- Posted URLs are stored in the Post table.
- The user-post table acts as a lookup so that you can quickly find all of the PostIds for a given user.
- The user-post table is replacing the foreign key index, making user-related lookups faster.
- The comments table has been absorbed by the post table, the comments table is nested (embedded) in the post table.

Before we look at nested entities, we are going to compare normalized and denormalized schemas.

## Entity Attributes

### Identifying attributes

- Map to a row key

### Non-identifying attributes

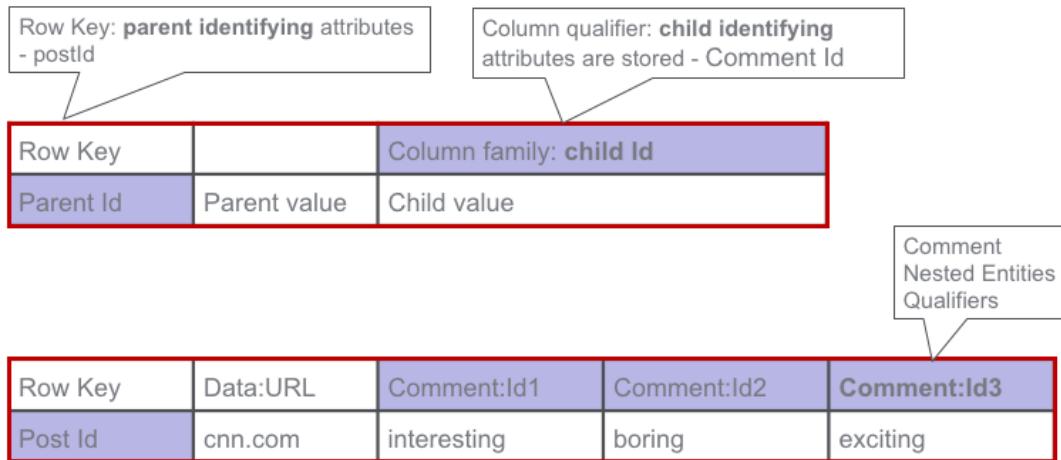
- Usually map to columns



What should you do with Entity attributes? Identify attributes mapping to the row key.

Other non-identifying attributes include password, user name, URL title, for example, anything that's not the unique or primary key. These non-unique attributes usually map to columns which may be defined dynamically in HBase. You may also consider saving them as part of the value (analog to a SQL blob) if there are a lot of them.

## Parent-Child Relationship – Nested Entity



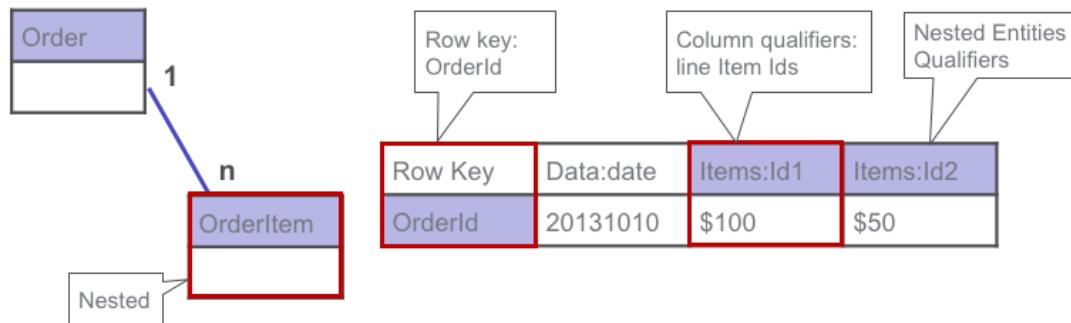
Here is an example of denormalization in our social application's HBase schema. Instead of storing the comments in a separate table like in the relational model, we can store them in the row for the post. This allows us to get all comments for a post in one read with no table joins.

If your tables exist in a parent-child, master-detail, or other strict one-to-many relationship, it's possible to model it in HBase as a single row.

A unique feature in HBase is the ability to have dynamic column names. This allows us to nest, or embed, an entity inside the row of a parent or primary entity. The row key will correspond to the parent entity ID. The nested values will contain the children, where each child entity gets one column name into which the ID is stored. The remainder of the non-identifying attributes are put in the value. This kind of schema design is appropriate if the only way you get at the child entities is via the parent entity.

The embedding of the comment entity in the post table is an example of this.

## Parent-Child Relationship –Nested Entity



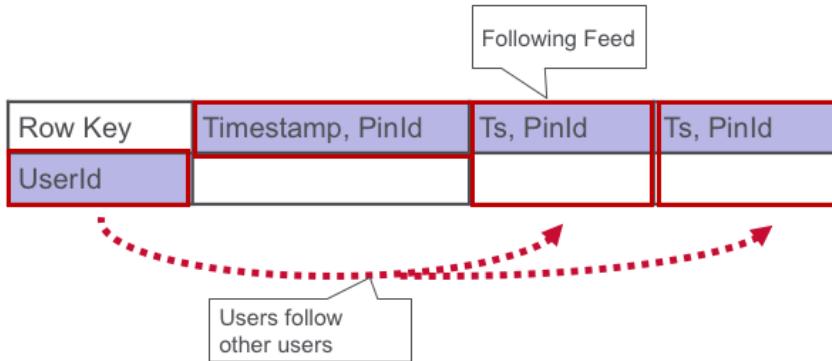
Here is another example of a one-to-many relationship, modeled in HBase as a single row.

- The row key corresponds to the parent entity ID, the OrderId.
- There is one column family for the order data, and one column family for the order items.
- The Order Items are nested, the Order Item IDs are put into the column qualifier and any non-identifying attributes, into the value.

Once again, this kind of schema design is appropriate if the only way you get at the child entities is via the parent entity.

## Pinterest Example HBase

Following Feed table



Here is a real world example from Pinterest (hbasecon).

Pinterest uses a follow model, where users follow other users. There is a following feed for every user that gets updated every time a followee creates or updates a pin. This results in 100's of millions of possible pins per month, and billions of writes per day. The 'Following Feed' is implemented as a fat wide HBase table, where each user's following feed is a single row. In the "Following Feed" table, a user's following pins are put in one row as nested entities, the timestamp and PinId are put into the column qualifier. A background queue writes the pins.

## Knowledge Check



## Knowledge Check



What are the basic principles used when transitioning from a relational model to HBase?

1. Denormalization
  2. Duplication
  3. Use intelligent keys
- 
- A. Used for indexing
  - B. Design for reads
  - C. Replaces JOINS

Answers: 1 - C; 2 - B, 3 - A

## Learning Goals



## Learning Goals



5.1 Transition From Relational Model to HBase

### **5.2 Use Intelligent Keys**

5.3 Use Lookup Tables for Secondary Indexes

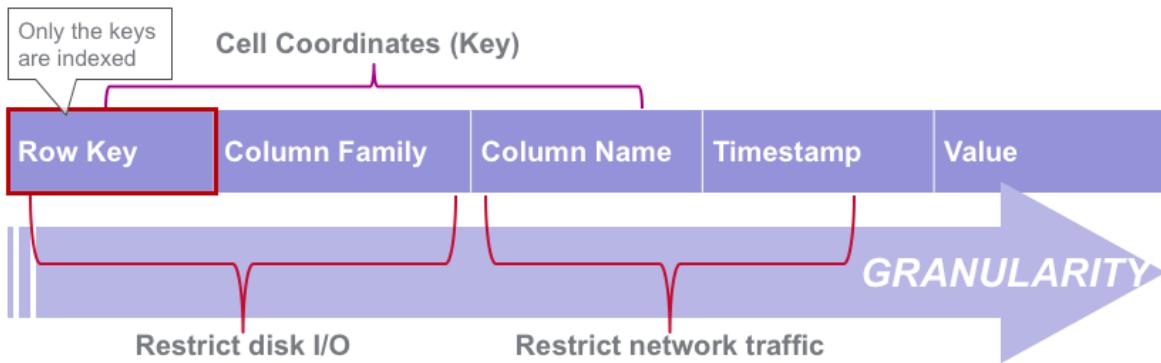
5.4 Design for Other Complex Data Structures

5.5 Evolve Schemas Over Time

We have taken a look at denormalization and duplication. In this section, we will describe how to use intelligent keys.

## Intelligent Keys

- Compose the key with attributes used for searching
  - Composite key: two or more identifying attributes
  - Like multi-column index design in RDBMS
  - Use fixed length, or separators



You can use intelligent keys to implement indexing, sorting and optimizing reads. The use of intelligent keys gives you control over how data is retrieved. Remember, rows are ordered and partitioned by row keys. Only the keys are indexed in HBase tables. To take advantage of this storage organization, the key should be designed as a composition of the attributes that are most often used as search criteria - similar to a multi-column index design in relational databases.

You can also use composite keys combined with partial key scans as a leading, left-hand index, with each key field adding to its precision.

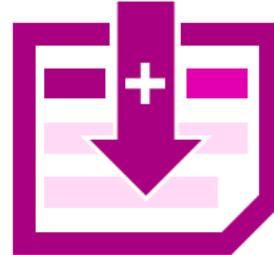
## Composite vs. Compound Keys

### Relational database

- Compound key: consists of two or more identifying **key** attributes

### HBase

- Composite key: two or more identifying attributes
- Save multiple elements into a row key
  - User ID + time stamp in tall narrow design (example - OpenTSDB key)
  - Use fixed length, or separators



In relational databases, a compound key is a key whereby any part of the key is a foreign key.

Example: In an hotel reservation system, a reservation has the compound key, (GuestId, HotelId). GuestId identifies a Guest, and references the Guests table. HotelId identifies a Hotel, and references the Hotels table.

In HBase, a composite key is made up of elements that may or may not be foreign keys.

Example: In a table of transaction details, the key is (TransactionId, ItemNumber). A transaction detail is a sub-entity of a transaction. TransactionId is a foreign key, referencing the Transactions table. ItemNumber is not a key in and of itself. It only uniquely identifies an item within the context of a single transaction.

Map multiple elements into a row key. It's common practice to take multiple attributes and make them a part of the row key. The key should be designed as a composition of the attributes that are most often used as search criteria.

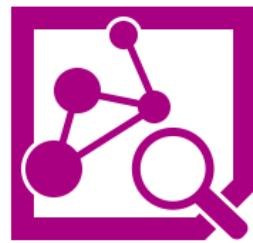
Using values of fixed length makes life much easier. Variable lengths mean you need delimiters and escaping logic in your client code to figure out the composite attributes that form the key. Fixed length also makes it easier to reason about start and stop keys.

## Intelligent Keys in Example

Composite key: Two or more identifying attributes

**Post** table key:

- Category code + separator + reverse time stamp
- Allows to scan for the latest post URLs by **category, date**
- Scan Posts WHERE category="science"



In the social app we have the following row keys:

User table key = UserId

Post table key = Category code + separator + reverse time stamp

This allows to get the latest URLs by category

User-Post lookup index table key = UserId + separator + post key

This allows to scan for post keys by user

Recall that you can use composite keys combined with partial key scans as a leading, left-hand index, with each key field adding to its precision. So this also allows us to scan for post keys by user + category + date.

Setting the nested Comment Column qualifier to be UserId+timestamp will identify nested comments by user and time.

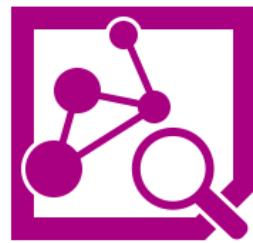
## Intelligent Keys in Example

### User-Post table key:

- UserId + separator + post key
- Allows to scan for post keys by user, category, and date
- Scan for post keys WHERE user and category = “**user+science\***”

### Nested Comment Column qualifier:

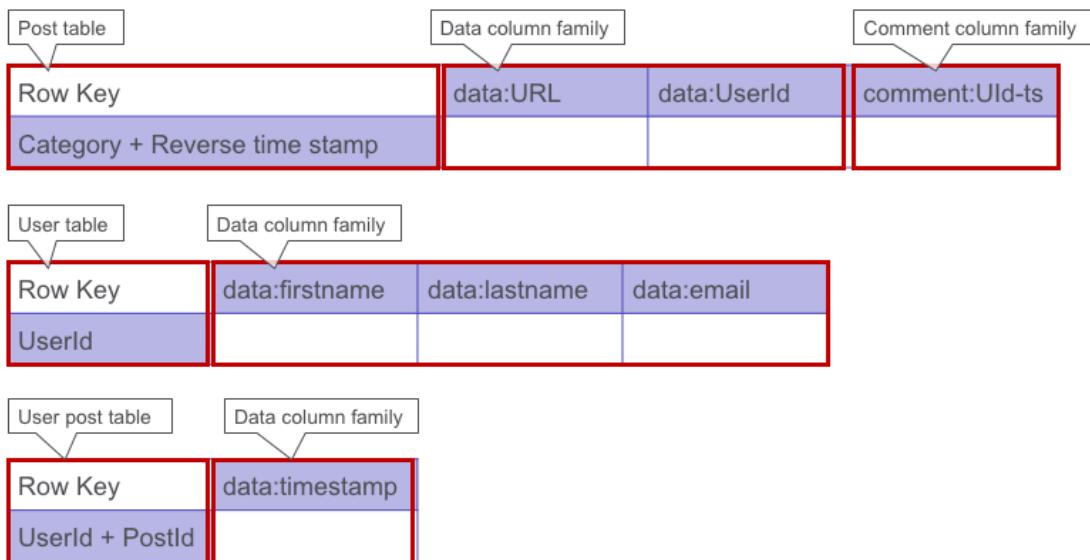
- UserId + timestamp



User-Post lookup (index) table key = UserId + separator + post key. This allows us to scan for post keys by user. Recall that you can use composite keys combined with partial key scans as a leading, left-hand index, with each key field adding to its precision. This also allows us to scan for post keys by user + category + date.

Setting the nested Comment Column qualifier to be UserId + timestamp will identify nested comments by user and time.

## HBase Tables Keys and Columns



This is the HBase schema again, now with the row keys and column qualifiers. Posted URLs are stored in the post table.

The user-post table acts as a lookup so that you can quickly find all of the PostIds for a given user.

The user-post table is replacing the foreign key relationship, making user-related lookups faster.

The user table stores the user details.

The comments table has been absorbed by the post table, the comments table is nested in the post table, the comments columns qualifier is the UserId-date.

There are various approaches to converting one-to-one, one-to-many, and many-to-many relationships to fit the underlying architecture of HBase. You could implement this example in different ways.

## Knowledge Check



## Knowledge Check



Why would you use intelligent keys?

1. To implement indexing and sorting
2. To optimize reads
3. To save disk space

Answers: 1 and 2

## Knowledge Check



To take advantage of the fact that only row keys are indexed in HBase tables,

1. Design a simple key that uses only the primary key
2. Design a composite key that uses attributes most often used as search criteria

Answer: 2

## Learning Goals



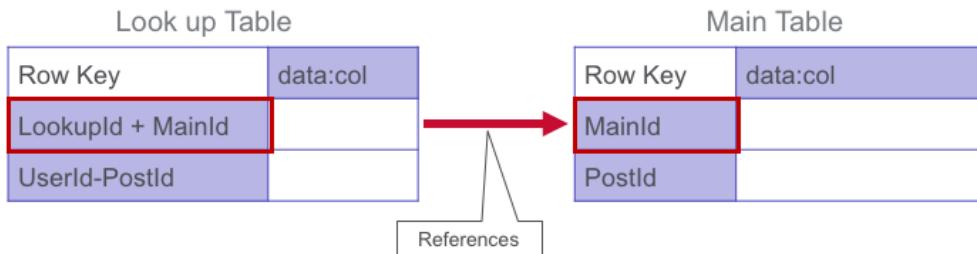
## Learning Goals



- 5.1 Transition From Relational Model to HBase
- 5.2 Use Intelligent Keys
- 5.3 Use Lookup Tables for Secondary Indexes**
- 5.4 Design for Other Complex Data Structures
- 5.5 Evolve Schemas Over Time

This section describes using lookup tables to implement secondary indexes.

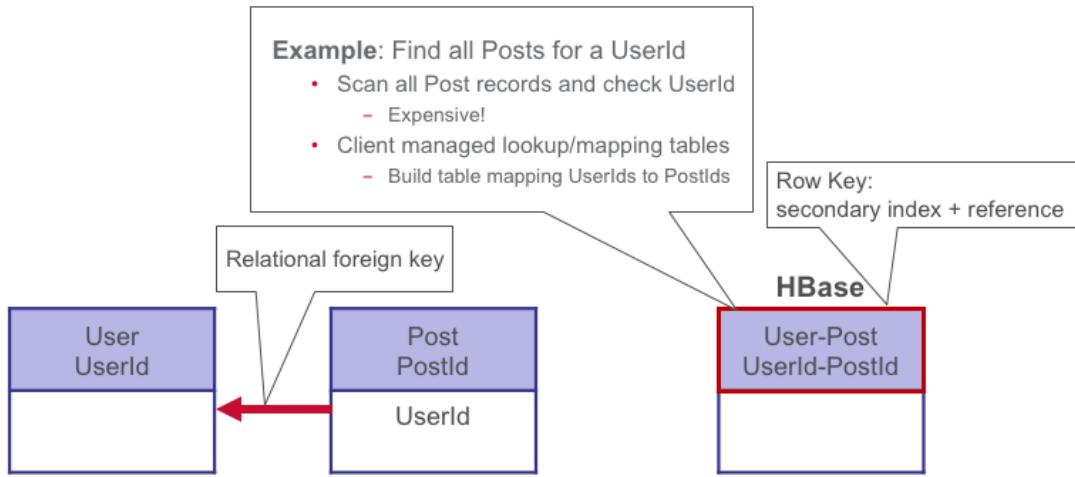
## Secondary Index – Lookup Table



A secondary index is a way to efficiently access records in a database, by means of some piece of information other than the usual primary key. Secondary indexing is a feature that evolved and matured in relational databases, but there is no built-in secondary indexing method in HBase. You can, however, approximate secondary indexing by using clever row key design and/or by maintaining a secondary lookup table.

One solution to implementing secondary indexes is to have a secondary index table, where the application maintains a lookup/mapping table. Looking up the data referenced from a secondary index requires a lookup of the main row key and then retrieval of the data in a second operation.

## Secondary Index – Lookup Table Example



In the social app example we want to find all URLs for a specific user. A simple approach is to scan all post records and check the UserId. This could be very expensive obviously, so you need an index that keeps track of posts by user.

A solution is a user-post lookup/mapping table. The user-post table acts as a lookup so that you can quickly find all of the PostIds for a given user.

The user-post table is replacing the foreign key index, making user-related lookups faster. This is a lookup table (basically an index) to find all URLs for a given user. Getting a post by UserId requires a lookup of the post row key, and then retrieval of the post in a second operation.

Note that this table is not automatically generated, nor maintained by HBase.

## Secondary Index – Column Family Example

Secondary Index

**Example:** Find all URLs posted by UserId cmcdonad

- Put PostIds in a user table column family

Row Key	data:firstname	...	post:PostId	post:PostId
UserId				

Here is another solution for the same example. In the social app example we want to find all URLs for a specific user.

Another solution is to add a column family to the user table for the posts that this user has made. Now, to find all the posts by UserId, do a Get on the user table row key, which is the UserId, to Get all of the posts for this user, then to retrieve a post, you have the PostId to use to do a Get from the post table.

## Index Table – Client Managed

HBase

No native support for secondary indexes

No cross row atomicity, no transactions across tables

For consistency:

- Write to the index table first
- Write to main table next
- MapReduce job remove orphan mappings

There is no built-in secondary indexing maintenance in HBase. For an application managed index table, you have to maintain consistency on your own.

If an index table is updated as the main table data is updated, then both tables need to be updated at the same time.

Since there are no cross table transactions for consistency, this could result in data being stored in the main table, but with no mapping in the secondary index table, because the operation failed after the main table was updated, but before the index table was written.

A solution can be to write to the secondary index tables first and to the main table second. Should anything fail in the process, you are left with orphaned index mappings, but those could be removed by regular background cleanup jobs.

For an existing table, a lookup table could be created by MapReduce jobs.

If using TTL, setup the main table to expire slightly before the index table(s).

## Knowledge Check



## Knowledge Check



HBase does not have built in secondary indexes. One way to implement secondary indexes in HBase is by:

1. Setting row key to be the primary key
2. Using a lookup table
3. Using a clever row key design
4. Designing column families appropriately

Answers: 2, 3, and 4

## Knowledge Check



Which of the following is true of index tables in HBase?

1. Index tables are used to support reverse lookup
2. The key is typically based on search criteria identified for your queries
3. All your queries will use the same index table
4. It is not uncommon for each query to have its own index table

Answers: 1, 2, and 4

## Learning Goals



## Learning Goals



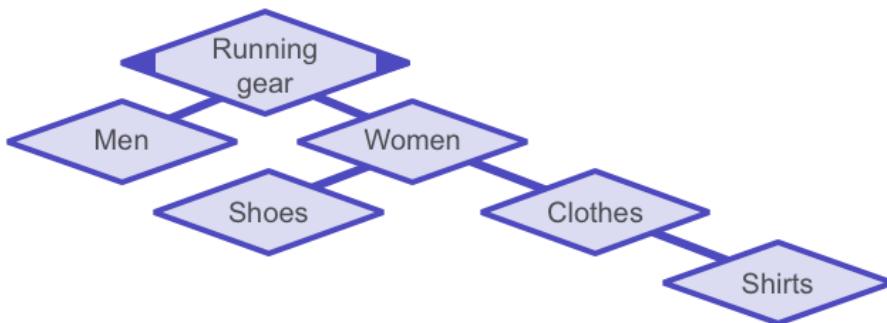
- 5.1 Transition From Relational Model to HBase
- 5.2 Use Intelligent Keys
- 5.3 Use Lookup Tables for Secondary Indexes
- 5.4 Design for Other Complex Data Structures**
- 5.5 Evolve Schemas Over Time

Now we will take a look at designing schemas for other complex data structures such as hierarchical data.

## Hierarchical Data - Example

Tree-like structure

- Online running store example
- Query: Women's running gear?



Data may be organized in a tree-like or hierarchical way. Here are some examples.

- Organization chart: The relationship of employees to managers is the textbook example of tree-structured data. It appears in countless books and articles on SQL. In an organizational chart, each employee has a manager, who represents the employee's parent in a tree structure. The manager is also an employee.
- Threaded discussion: A tree-structure may be used for the chain of comments in reply to other comments. In the tree, the children of a comment node are its replies.

The most common solution in SQL is to store the parent ID with a child. For example each comment is linked to its parent that way. This is known as an adjacency list. You can't query all descendants with such a structure.

Another way to query a tree-structure from adjacency list is to retrieve all the rows in the collection and reconstruct the hierarchy in the application before you can use it like a tree. Copying a large volume of data from the database to the application before you can analyze it is grossly inefficient.

Solutions with HBase:

Consider the flat-wide or tall-narrow design as seen before. You can nest entity descendants for each entry as necessary. Keep the level of nesting to a minimum: it's not as efficient to access an individual value stored as a nested column inside a row, as compared to accessing a row in another table.

Next we will look at some examples for this online running store.

## Hierarchical Data – Child References

### Child references

- Query to retrieve immediate children of a node is fast



Row Key	Children
rungear	men, women
women	clothes, shoes
clothes	shirts
shirts	0

We are now going to look at some ways of storing the tree in HBase.

The first is the child references pattern. Here we store a string in a column that contains the children of the current row.

Path enumeration: store a string in a column with the children of the current row.

## Hierarchical Data – Parent, Children

Parent, children in columns



Row Key	Parent	Child
rungear		men, women
women	rungear	clothes, shoes
clothes	women	shirts
shirts	clothes	

In this example, we store a string in a column with the sequence of the parent and a string with the sequence with the children in another column for each row. One column shows parents and the other column shows children.

## Hierarchical Data – Materialized Path

Materialized path



Row Key	Path
rungear	0
women	rungear
clothes	rungear, women
shirts	rungear, women, clothes

Here we are storing a string in a column with the sequence of ancestors of the current row in order from the top of the tree down, just like a UNIX path. This example uses a comma as a separator. You can find ancestors or descendants as substrings of the path.

## Hierarchical Data

Parent and ancestors



Row Key	Parent	Ancestors
rungear	0	0
women	rungear	rungear
clothes	women	rungear, women
shirts	clothes	rungear, women, clothes

In this example, one column shows the parent and the other column shows ancestors.

## Hierarchical Data

Use a tall-narrow

- Materialized path in row key
- Row key filter /^,women,/



Row Key	
runstuff,women,clothes,shirts	

Here is an example using path enumeration with the path in the row key. You can use the materialized path in the row key.

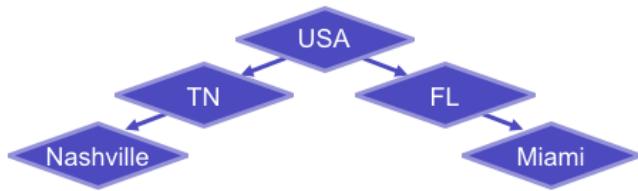
## Tree, Graph Data

Row = node

Row key = node ID

Parents, children in columns

- Col Qualifier = Edge ID



Row 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				

Here is an example for an adjacency list or graph, using a separate column for each parent and child. Each row shows a node. The row key is equal to the node ID. There is a column family for parent p, and a column family children c.

The column qualifiers are equal to the parent or child node IDs and the value is equal to the type to node.

You can see there are multiple ways to represent trees, the best way depends on your queries. HBase is great for sparse data, so design with this sort of data in mind.

## Generic Data, Event Data, Entity-Attribute-Value

### Generic table: Entity, Attributes, Values

- Event Id, Event attributes, Values
- Object-property-value, name-value pairs, **schema-less**

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

### Advantage of HBase

- Define columns on the fly, put attribute name in column qualifier
- Group data by column families

Event Id = row key	Event type name = qualifier	Event measurement = value
Row Key	event:heartrate	event:coughing
Patientxyz-ts1	98	true

Generic data that is schema-less, is often expressed as name value or entity attribute value. In a relational database this is complicated to represent. The advantage of HBase is that you can define columns on the fly, put attribute names in column qualifiers, and group data by column families.

Here is an example of clinical patient event data. The row key is the patient ID plus a time stamp. The variable event type is put in the column qualifier. The event measurement is put in the column value. OpenTSDB is an example of variable system monitoring data.

A conventional relational table consists of attribute columns that are relevant for every row in the table, because every row represents an instance of a similar object. A different set of attributes represents a different type of object and thus belongs in a different table. When having to deal with variable attributes, a common SQL solution is to create a second table, storing attributes as rows:

The Entity: Typically this is a foreign key to a parent table that has one row per entity.

The Attribute: This is simply the name of a column in a conventional table, but in this new design, we have to identify the attribute on each given row.

The Value: Each entity has a value for each of its attributes.

## Inheritance Mapping

### Subtypes of Entities



In modern object-oriented programming models, different object types can be related, for instance, by extending the same base type. In object-oriented design, these objects are considered instances of the same base type, as well as instances of their respective subtypes. We would like to store objects as rows in a single database table to simplify comparisons and calculations over multiple objects. But we also need to allow objects of each subtype to store their respective attribute columns, which may not apply to the base type or to other subtypes.

You can't join to a different table per row in SQL. SQL syntax requires that you name all the tables literally at the time you submit the query.

## Inheritance Mapping - HBase

Similar to Entity, Attribute, Value

Put subclass type as prefix in row key OR

Column qualifier

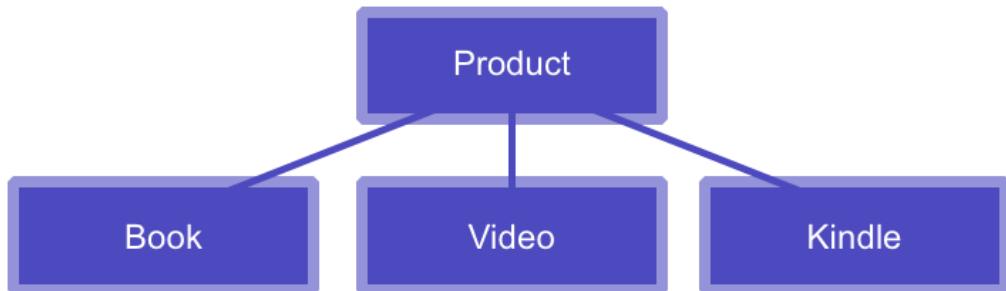
Payment Key	
check+Id	
credit+Id	

Put subclass type as prefix

In HBase, you can denormalize and/or use column families. For example, put the object subclass type i.e. check or credit, in the row key or column qualifier. This is one way of handling variable attributes.

## Inheritance Mapping - Example

Online Store Example



Here is another inheritance example, this time for an online store with different kinds of products.

## Inheritance Mapping – Example

Online Store Example Product Table

- Put **subclass type** in **column qualifier**
- Columns do not all have to be the same for different types

Row Key	type	price	title	details	model
Id	book	10	HBase		
Id2	dvd	15	stones		
Id3	kindle	100			fire

In this example, the type of product is a column name and some of the columns are different and maybe empty depending on the type of product.

## Inheritance Mapping – Example

Online Store Example Product Table

- Put **subclass type abbreviation** in **key prefix** for searching
- Columns do not all have to be the same for different types

Row Key	price	title	details	model
Bok+Id1	10	HBase		
Dvd+Id2	15	stones		
Kin+Id3	100			fire

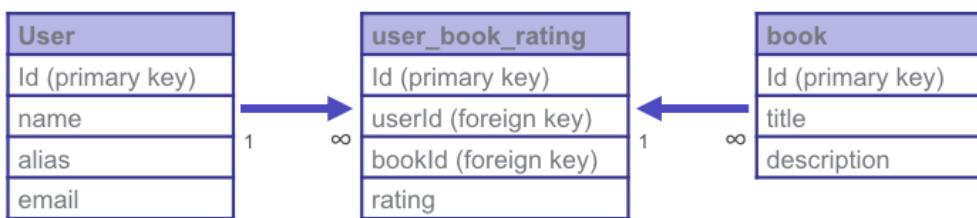
In this example, the type of product is a prefix in the row key and some of the columns are different and maybe empty depending on the type of product. HBase is designed for sparse data.

## Many to Many Relationship - RDBMS

### Queries

- Get name for user x
- Get title for book x
- Get books and corresponding ratings for UserId x
- Get all UserIds and corresponding ratings for book y

Online book store



Here is an example of a many-to-many relationship in a relational database. These are the query requirements:

- Get name for user x
- Get title for book x
- Get books and corresponding ratings for UserId x
- Get all UserIds and corresponding ratings for book y

Next we will look at a way to do this with HBase.

## Many to Many Relationship - HBase

### Queries

Get books and corresponding ratings for UserId x

Get all UserIds and corresponding ratings for book y

User table Column family for book ratings by UserId for BookIds

Row Key	data:fname	...	rating:BookId1	rating:BookId2
UserId1			5	4

Book table Column family for ratings for BookId by UserId

Row Key	data:title	...	rating:UserId1	rating:UserId2
BookId1			5	4

The queries that we are interested in are:

- Get books and corresponding ratings for UserId x
- Get all UserIds and corresponding ratings for book y

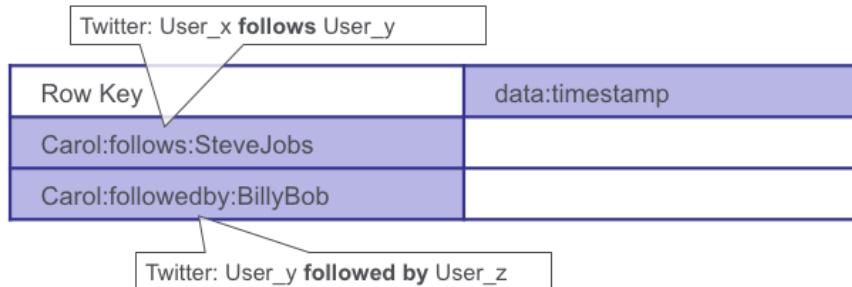
The entity tables are as shown here. For an entity table, it is pretty common to have one column family storing all the entity attributes, and column families to store the links to other entities.

## Self Join Relationship - HBase

### Queries

Get all users who Carol follows

Get all users following Carol



A self-join is a relationship in which both match fields are defined in the same table. Here we have a twitter table for Queries:

Which users does userX follow?

Which users follow userX?

A possible solution: The UserIds are put in the row key with the relationship type. For example, Carol follows Steve Jobs and Carol followed by BillyBob.

## Facebook Inbox

	Row Key = UserId	ColumnFamily = msg	Column = message Id
UserId1		msg:16	msg:17
UserId2	Hello carol ...		Hi here is a message

This is a real world example from Facebook. Facebook stores messages in HBase. The UserId is the row key, there is one column family for messages, the messages Id is the column name, and the message is the value. Next we will look at how they index searches.

## Facebook Inbox Search

Data is stored sorted by

<key=UserId, col=word, version=messageID>: value offset in document

User1:s:hi:17->offset1

User1:s:hi:16->offset2

User1:s:hello:16->offset3

User1:s:hello:2->offset4

Version = messageID

Col = word

	msg:16	msg:17	s:hi	s:carol
UserId1		Hi here is a message	17: 0 version: value	
UserId2	Hello carol ...			16: 7

For indexing searches on words in messages, this is what Facebook does:

There is a column family "s" for storing work searches, the column name is the indexed word, the version is the message Id of where to find the word, the value is the offset of the word in the message. For example, the word "hi" is in message 17 at offset 0.

## HBase Modeling Concepts

Identify entities

Identify relationships

Identify queries

- What information accessed together in one Get
- What information needs scanning

Identify identifying attributes used  
in queries-> composite row key



Start with listing all of the use cases your application needs to support. Think about the data you want to capture and the lookups your application needs to do. Since there is no index concept, you have to plan out carefully how your data is physically sorted. Therefore it is important to find all your query use cases first.

With HBase, you need to design the row keys and table structure in terms of rows and column families to match the data access patterns of your application.

1. Start with identifying Entities. For an entity table, it is pretty common to have one column family storing all the entity attributes, and multiple column families to store the links to other entities.
2. Identify relationships. Identify the type of relationship that currently exists.
3. Identify Queries. What information is accessed together in one Get. What information needs scanning.
4. Find identifying Attributes used in queries. Which attributes uniquely identify a particular instance of the entity? These then could become part of the composite row key.

## HBase Modeling Concepts

Non-identifying attributes -> columns

### Relationships

- One to many -> nested or embedded entities in column family
- Many to many -> Two tables: entity1\_by\_entity2, entity2\_by\_entity1

### Secondary Index

- Lookup table, or put in Column Family



Non identifying attributes -> columns

Relationships: The type of relationship will affect your design.

One to many -> nested or embedded entities in column family

Many to many -> Two tables: entity1\_by\_entity2, entity2\_by\_entity1

Secondary Index:

Lookup table, or put in Column Family

Indexing is not an afterthought, anymore

Think about query patterns and indexes upfront.

## Lab 5.4a-c



Open your lab guide to perform the following labs:

1. Lab 5.4a: Model Person-Relatives Schema
2. Lab 5.4b: Model Movie Rental Online Store Schema
3. Lab 5.4c: Model Customer Click Event or Action

## Learning Goals



## Learning Goals



- 5.1 Transition From Relational Model to HBase
- 5.2 Use Intelligent Keys
- 5.3 Use Lookup Tables for Secondary Indexes
- 5.4 Design for Other Complex Data Structures
- 5.5 Evolve Schemas Over Time**

In this section, we will describe how to manage schemas that need to evolve over time.

## Common Scenarios

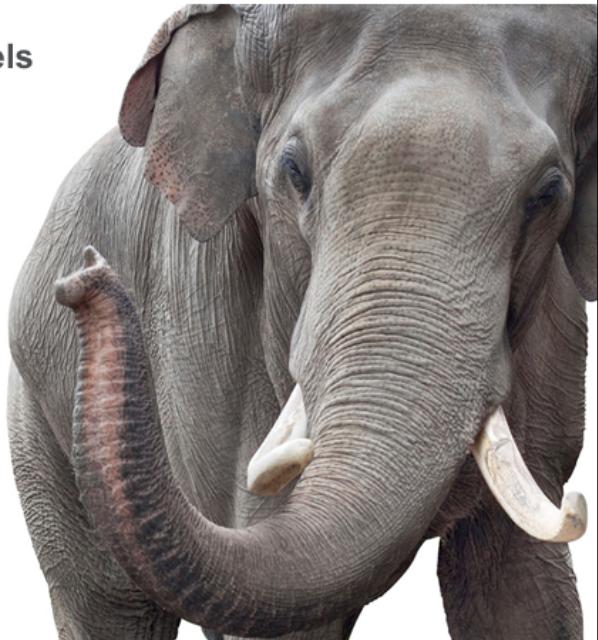
### Evolving schema → changes at two levels

#### 1 Change or update the schema

- Schema itself needs to change OR
- Table structure needs to be modified

#### 2 Change or update data

- Content needs to change
- Quality of data needs to change



MAPR Academy

Evolving a schema usually involves changes at at least two levels:

1. The schema itself or structure of the database needs to change
2. The content or quality of the data needs to change or be transformed

## Migrating a Database

Leverage infrastructure (MapR-DB)

- Use snapshots or mirrors

Avoid having to take system offline

Helps migrating current system to new one

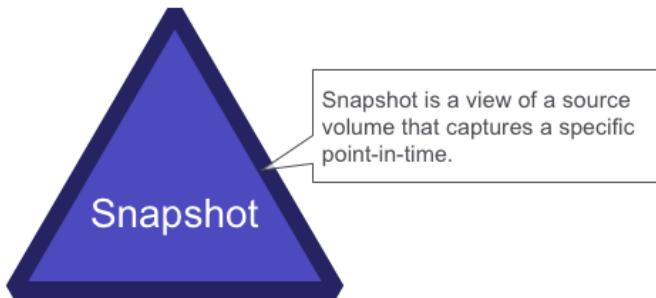
There are some tools you can use to migrate a database whether you are changing the structure or you are changing the data. You can use snapshots or mirrors if you are using MapR-DB.

## Snapshot

Provides point-in-time view of data

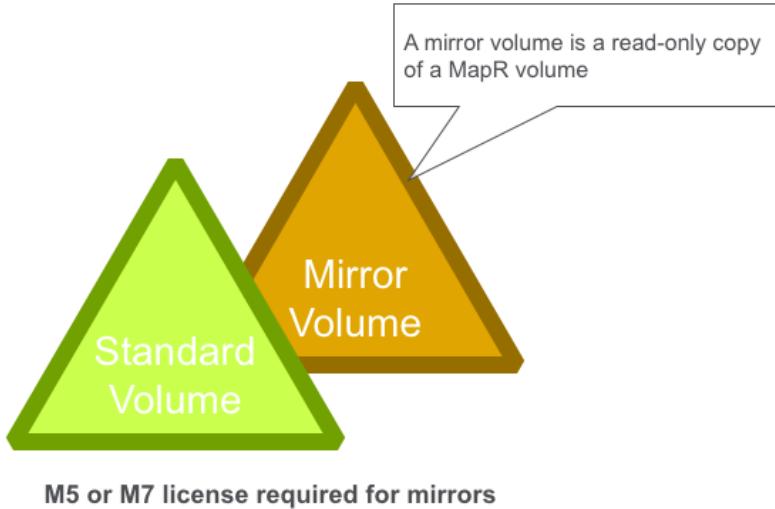
Read-only

Like a backup but without the space and effort normally required.



Instead of taking the system offline, take advantage of snapshots to migrate data from read-only source. Snapshots provide a view of a volume at a specific point-in-time.

## Mirror Volumes



Instead of taking the system offline take advantage of mirrors to migrate data from read-only source. Mirrors are read-only copies of a MapR volume.

## Schema Changes

Use HBase shell, java API or MapR Control System

Change the following properties on a column family:

- Name
- Min, max versions
- TTL
- Compression
- Memory residence (caching)

For schema changes:

Using the MapR Control System you can change the following properties on a column family:

A column family's name, minimum and maximum versions, time-to-live, compression, memory residence status. You can also change column family properties using the HBase shell or the java API.

## Schema Changes

Note:

- Tables can be **altered**
- Column families can be added
- Columns may be **added dynamically**

Recommendations:

- Keep table and **column families simple** and generic
- Add **column families** for migration

Note:

Tables can be altered

Column families can be added

Columns may be added dynamically

Columns may be altered (modifyColumn())

Recommendations:

Keep table and column families simple and generic

Add column families for migration

## Data Changes

There is no difference between an insert and an update

- Only the version differs

Use versions to keep only what is needed

Expire data with TTL

Adding new values to column - a way to migrate values

Row keys cannot be changed

There is no difference between an insert and an update, only the version differs. Use versions to keep only what is needed and expire data with TTL. Adding new values to a column may be a way to migrate values. Recall that row keys cannot be changed once data has been written to the table.

## Summary

- General guidelines for moving from a relational model
- Nested or embedded attributes
- Intelligent keys
- Other data structures
- Evolving schema over time

In this lesson, we have looked at some general guidelines for moving from a relational model to HBase. We looked at denormalization, duplication, and intelligent keys. This lesson also went over designing schemas for hierarchical data and evolving schemas over time.

We are now going to look at some use cases.



## Next Steps

### DEV 325- Apache HBase Schema Design

Lesson 6: Query HBase with Hive

Congratulations. You have completed Lesson 5. Now that you have seen how design schemas, we are going to see how to query HBase.



## DEV 325 – Apache HBase Schema Design

Lesson 6: Query HBase with Hive

Winter 2017, v5.1

Welcome to DEV 325 – Apache HBase Schema Design, Lesson 6: Query HBase with Hive.

## Learning Goals



## Learning Goals



- 6.1 Use Hive to Query HBase/MapR Tables

So far you have seen how to design schemas in HBase.

## Learning Goals



### 6.1 Use Hive to Query HBase/MapR Tables

We are now going to take a brief look at Hive, which we will use to query HBase tables.

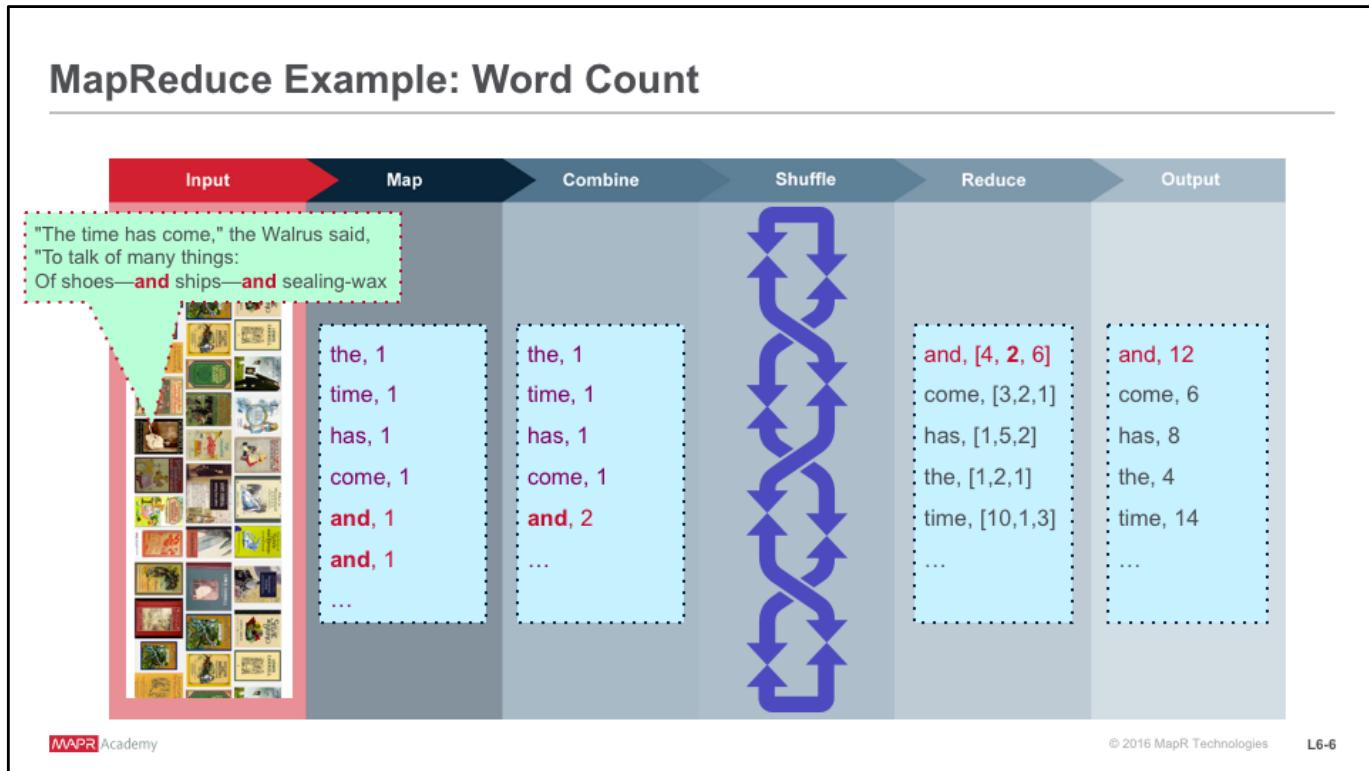
## What is Hive?

- Data Warehouse on top of Hadoop
  - Open Source Apache project
- SQL like querying for Hadoop
  - Uses a SQL variant called HiveQL
- HiveQL executes using MapReduce
  - Submits jobs to your cluster



Hive is a data warehouse infrastructure built on top of Hadoop that provides a SQL-like syntax for performing queries using MapReduce. The HiveQL language provides a set of operations that are structured similar to SQL, and are transformed into MapReduce programs to produce the desired query results.

Hive should be used for data warehouse analytic processing where fast response times are not required.



To best understand Hive, we need to peek under the hood and see the engine it is running on. SQL usually exists in a land of strict organization, indexes, and reverse indexes that facilitate very efficient data access, that we've discussed as RDBMS's. In contrast, Hive is more an interface to Hadoop than being a database.

Let's step through an example word-count program using MapReduce to count the occurrences of each word in a set of input text files.

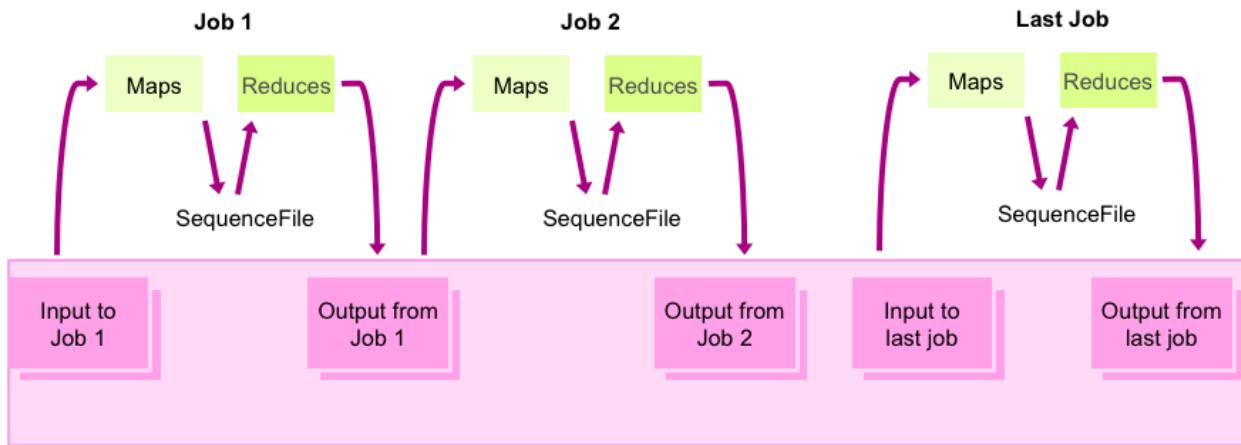
As input, let's say we have all Lewis Carroll's books and we want to count the occurrence of every word.

1. Hadoop divides this into "input splits."
2. One of the nodes contains Tweedledee's poem, "The time has come to talk of many things" and so forth. In the case of text input, every line of text is a record. Each record gets fed individually to the Map function. The key is the byte-offset into the file, the value is the text. In this case, the key isn't useful for this program, so we ignore it.
3. The map function tokenizes the input string, and outputs a key-value pair for every word. The key is the word, and the value is simply 1. As you can see, the word "and" shows up twice, and it produces two distinct key-value pairs.
4. Then begins the reduce phase, which just has to sum up values from the Mapper.

In this case, some combining occurs before shuffle-and-sort. The combiner aggregates multiple instances of the same key coming out of the map into a subtotal. So in our case, the two instances of "and" get combined, and only one record of value 2 goes through shuffle-and-sort.

5. The framework sorts records by key and, for each key, sends all records to a particular reducer. This particular reducer gets the keys: and, come, has, the and time. You can see that the combined value 2 for "and" shows up here. The other values come from other map tasks.
6. Finally, the framework gathers the output and deposits it in the file system where we can read it. (Of course, if we really counted all words in a book, there would be a lot more occurrences of these words than shown here.)

## Typical MapReduce Workflows



You can use MapReduce workflows to analyze data. The MapReduce workflow shown here involves many MapReduce jobs. Running multiple jobs in series is very typical when using MapReduce.

## MapReduce Can be Complex

```
public void reduce(Text key, Iterable<Text> values, Context context) throws
IOException, InterruptedException {
for (Text value: values) {
    compositeString = value.toString();
    compositeStringArray = compositeString.split("_");
    tempYear = new Text(compositeStringArray[0]);
    tempValue = new Long(compositeStringArray[1]).longValue();
    if(tempValue < min) {
        min=tempValue;
        minYear=tempYear;
    }
}
Text keyText = new Text("min" + "(" + minYear.toString() + "): ");
context.write(keyText, new FloatWritable(min));
}
```

Here is a small sample of code from a reducer class. As you can see, writing MapReduce code involves java programming which can be complex for non programmers. We will see that Hive provides a way for those familiar with SQL syntax to easily create MapReduce programs, and to execute queries on data stored in Hadoop.

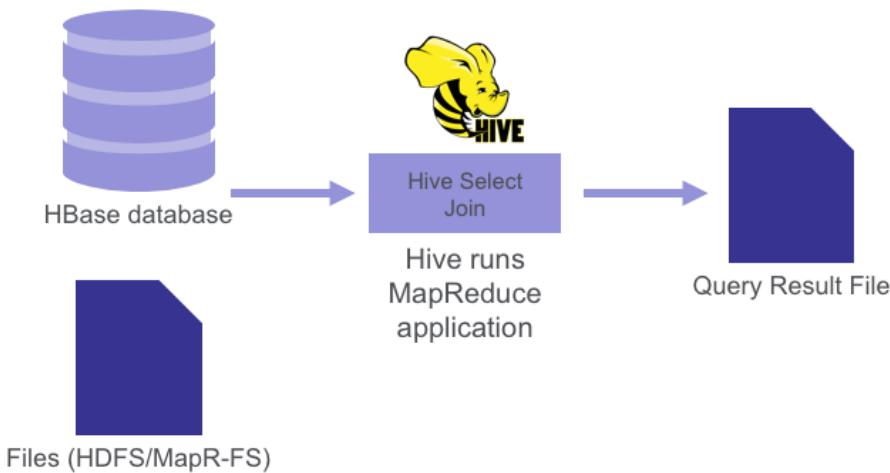
## Hive and HBase

- Hive is data warehouse infrastructure on Hadoop
  - Gives ability to query without programming
  - Used for analytical querying of data over time
- HBase is NoSQL datastore on HDFS
  - Queries run real-time in database
- Can use Hive as query layer to HBase

Hive provides a language that is very similar in both structure and syntax to SQL, making it easy to learn given the very wide use of SQL in modern databases. It is an analytics tool that was designed for ad hoc batch processing of potentially enormous amounts of data by leveraging MapReduce. Hive allows data analysts an easy way to use Hadoop, without programming, to facilitate querying and managing large datasets residing in distributed storage in much the same manner as traditional data warehousing software.

We can use HiveQL statements to access HBase tables for both read (SELECT) and write (INSERT). It is even possible to combine access to HBase tables with native Hive tables via joins and unions.

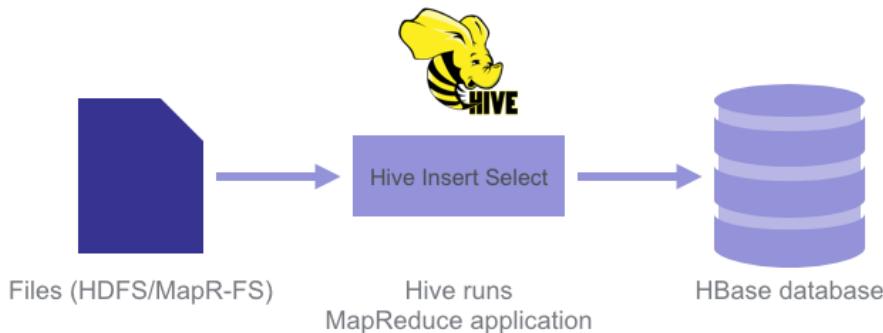
## Using HBase as a Hive Source



*EXAMPLE: Data Warehouse for Analytical Processing Queries*

One way to leverage Hive with HBase is when you use the data in the HBase database as source of your data flow. An example of doing so is a set of well-defined Batch Analytical Processing queries that you've implemented using HiveQL.

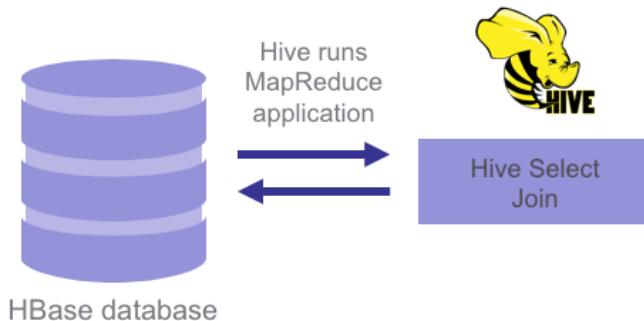
## Using HBase as a Hive Sink



### *EXAMPLE: Bulk Load Data into a Table*

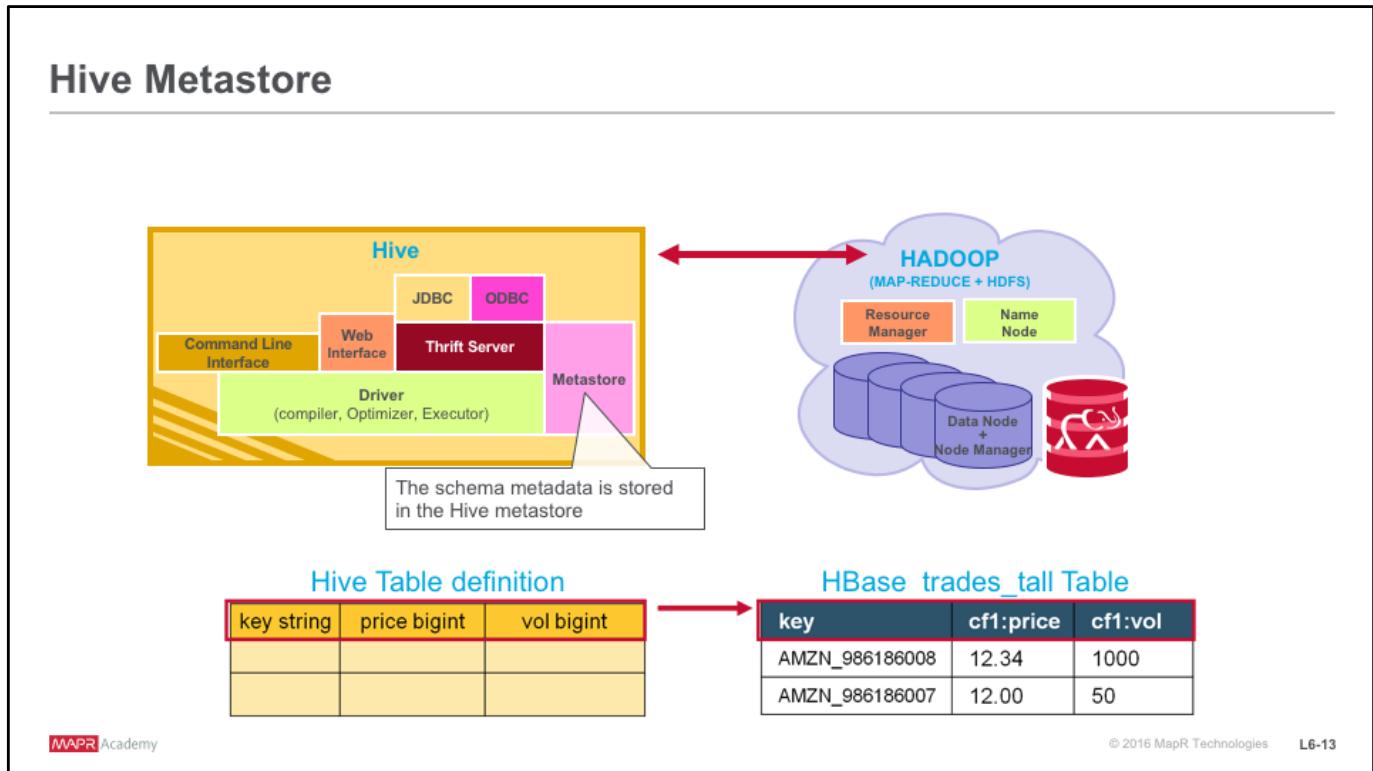
Another way to leverage Hive with HBase is when you use the HBase database as a sink for your data flow. An example of using HBase as a Hive sink application is when you want to use a batch job to bulk load data from files into your HBase table.

## Using HBase as a Source and Sink



*EXAMPLE: Calculate and Store Summaries,  
Pre-Computed, Materialized View*

The last way to leverage Hive with HBase is when you use the HBase database as both a source and sink in your data flow. One example of this use case is when you calculate summaries across your HBase data and then store those summaries back in the HBase database. Another example of this use case is to reorganize HBase data into materialized views for faster reads.



This diagram shows the components that make up Hive and how they interact with the Hadoop framework.

Hive provides a number of ways to submit queries. Hive provides an interactive command line shell that resembles a SQL shell. There are also Java DataBase Connectivity (JDBC) and Open DataBase Connectivity (ODBC) drivers for Hive that allow applications to access Hive much like they would access any traditional database. Hive also provides an Apache Thrift client, which enables access to Hive from many different client-side languages, including java, C++, Python, Ruby, or PHP.

The Driver moves a HiveQL statement through all the phases needed to become a MapReduce job and returns the results.

The Metastore contains metadata about Hive tables. In order to query data in files or HBase with Hive you have to define a schema for this data.

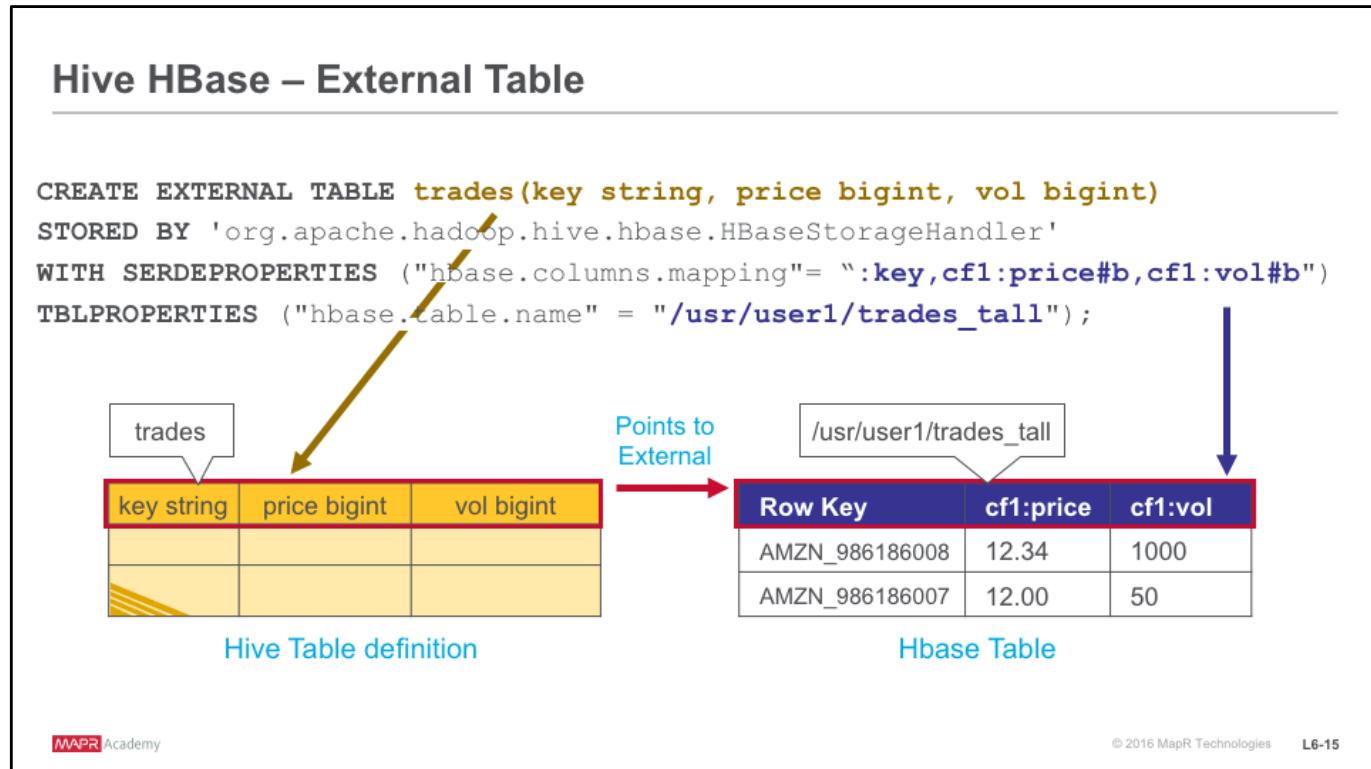
The Hive Metastore contains the definitions of tables (table name, columns and data types), the location of data files and the routines required to parse data (i.e. StorageHandlers, InputFormats and SerDes). In this example we see the Hive schema for the `HBase trades_tall` table.

## Hive HBase



There are two ways that you can run Hive queries on HBase tables:

- You can create and manage HBase tables from Hive that can be accessed by both Hive and HBase. These are called Hive Managed tables  
Or
- Hive can map to existing HBase tables. These are called external tables. When using an already existing table you can create multiple Hive tables that point an HBase table. This can be useful for example when you only want to query one out of multiple column families.



Here is an example of using Hive's external table functionality to create a table which sources its data from an existing HBase table. The Hive external table is metadata that is defined over the HBase table. The metadata maps the table name, column names and types to the HBase table. Once the Hive external table has been defined, it can be queried using HiveQL.

The above statement registers the HBase table named /usr/user1/trades\_tall in the Hive metastore, accessible from Hive by the name trades.

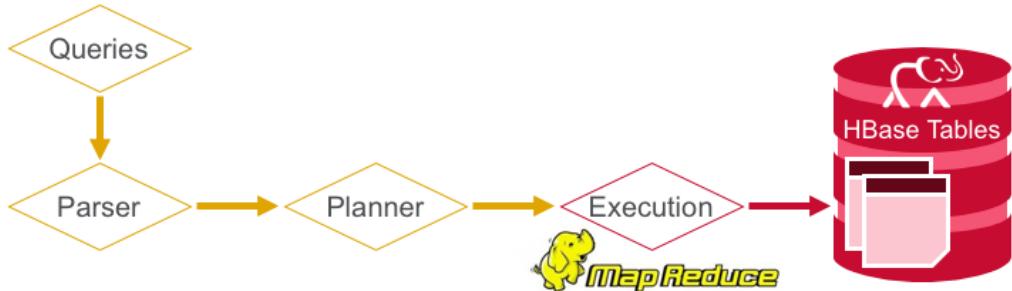
The `hbase.columns.mapping` SerDe property specifies a column mapping linking the Hive trades table key and column names price, vol, to the HBase table's row key and columns price and vol.

#b means binary data type.

## Hive HBase – Hive Query

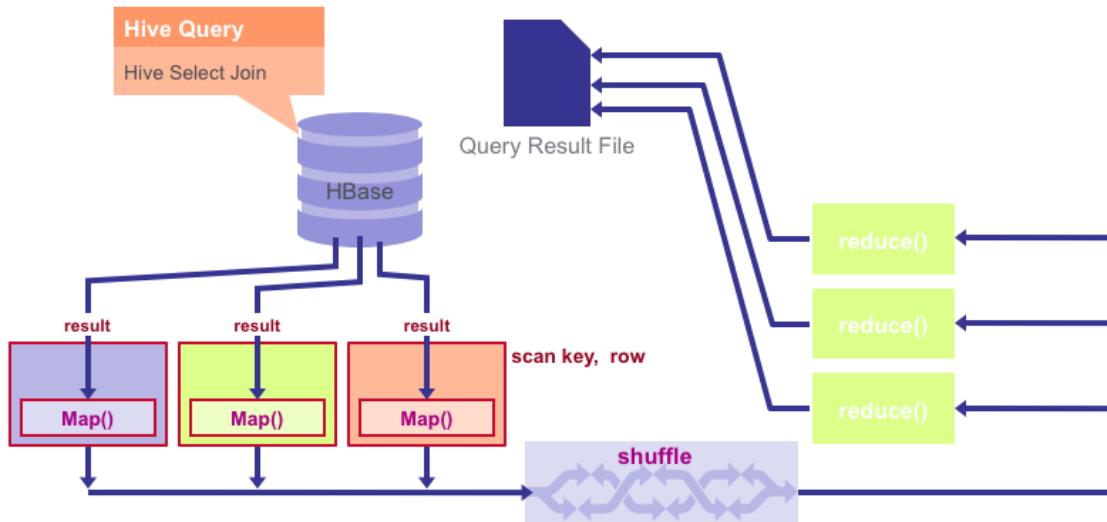
SQL evaluates to MapReduce code

```
SELECT AVG(price) FROM trades WHERE key LIKE "GOOG"  
;
```



Here we see the flow of a Hive query. The Parser translates the HiveQL statement into a plan which consists of series of MapReduce jobs. The driver submits the MapReduce jobs from the plan to the Execution Engine. Hive uses MapReduce as its execution engine.

## Hive MapReduce



Hive maps queries into MapReduce jobs, simplifying the process of querying large datasets in HDFS. HiveQL statements can be mapped to phases of the MapReduce framework. Selection and transformation operations occur in map tasks, while aggregation is handled by reducers. Join operations are flexible: they can be performed in the reducer or mappers depending on the size of the leftmost table.

## Hive Query Plan

```
EXPLAIN SELECT AVG(price) FROM trades WHERE key LIKE "GOOG%";
```

**STAGE PLANS:****Stage: Stage-1**

Map Reduce

Map Operator Tree:

TableScan

Filter Operator

predicate: (key like 'GOOG%') (type: boolean)

Select Operator

Group By Operator

**Reduce Operator Tree:**

Group By Operator

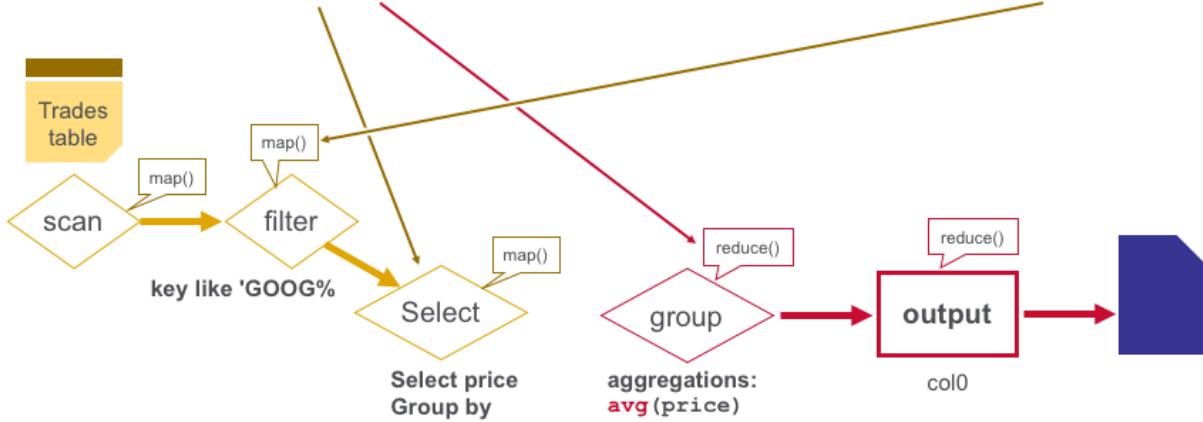
Select Operator

File Output Operator

Next, we will look at how a Hive query gets transformed into MapReduce jobs. If you have EXPLAIN before your query, Hive will output the query plan for the MapReduce jobs to run the query.

## Hive Query Plan

Hive> SELECT AVG(price) FROM trades WHERE key LIKE "GOOG%";



This diagram shows an example of SQL transformed into MapReduce jobs, in this case just one MapReduce job. The scanning, selecting columns, filtering values, happens first during the map job, results are sent to the reducer where they are aggregated and then output.

## Simple Query Examples

### Implicit Join (0.13 and above)

```
SELECT tableA.field1, tableB.field2, [any list of fields here ...]
FROM tableA, tableB
WHERE tableA.field1 = tableB.field2;
```

### Explicit Join

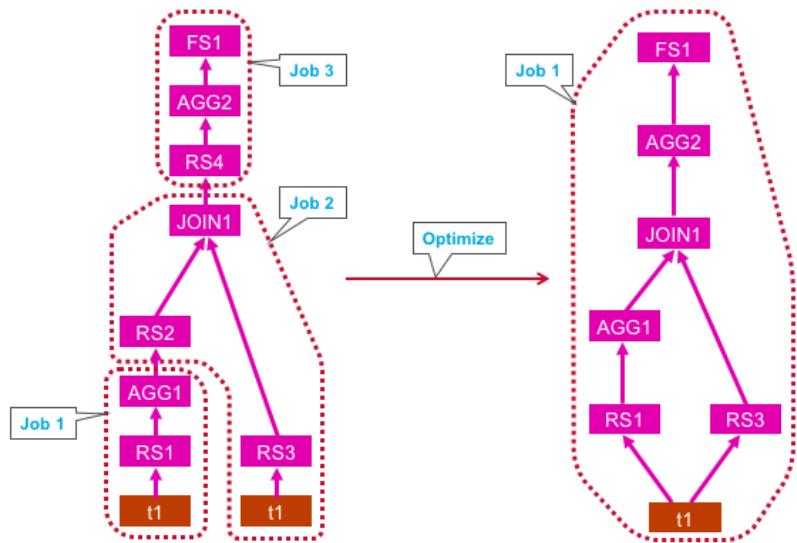
```
SELECT tableA.field1, tableB.field2, [any list of fields here ...]
FROM tableA JOIN tableB
ON tableA.field1 = tableB.field2;
```

Joins are a powerful feature of SQL-based databases and are also implemented in Hive, with some restrictions.

Joins give the Cartesian product of two or more tables based on the value of a specified pair of columns being equal.

Here we see the difference between an implicit and explicit join. These queries are functionally equivalent, but the implicit form is only available in Hive 0.13 and above.

## Hive Query Plan MapReduce Execution



MAPR Academy

© 2016 MapR Technologies

L6-21

This example shows the query plan or a more complex Hive query with joins. On the left the query plan will run as three MapReduce jobs, but on the right, the optimization engine has optimized the plan to one MapReduce job.

<https://cwiki.apache.org/confluence/display/Hive/Correlation+Optimizer>

```
SELECT tmp1.key, count(*)
FROM (SELECT key, avg(value) AS avg
      FROM t1
      GROUP BY /*AGG1*/ key) tmp1
JOIN /*JOIN1*/ t1 ON (tmp1.key = t2.key)
WHERE t1.value > tmp1.avg
GROUP BY /*AGG2*/ tmp1.key;
```

## Knowledge Check



## Knowledge Check



The different ways to leverage HBase with Hive include:

1. Using HBase as a Hive source
2. Using HBase as a Hive sink
3. Using HBase as a Hive source and sink
4. All of the above

Answer: 4

## Knowledge Check



Which of the following is true of the Hive Metastore?

1. It contains metadata about Hive tables
2. Metastore contains the definition of schema for HBase tables
3. It moves a HiveQL statement through phases to become a MapReduce job
4. All of the above

Answer: 1

## Knowledge Check



The different ways to run Hive queries on HBase tables include:

1. Using Hive managed tables
2. You cannot use HBase tables
3. Using external tables
4. Using only HBase tables serving as a sink

Answer: 1 and 3

## Lab 6.1a-b



In this lab you will use Hive to query HBase tables. Refer to your lab guide to run the following:

1. Lab 6.1a: Use Hive with the Airlines HBase table
2. Lab 6.1b: Use Hive to Query the Trades Table



## Next Steps

### DEV 330 – Developing HBase Applications: Basics

Lesson 7: Java Client API, Part 1

You have now completed Lesson 6. Congratulations! You have completed the course!



## DEV 330 – Developing HBase Applications: Basics

Lesson 7: Java Client API, Part 1

Winter 2017, v5.1

Welcome to DEV 330: Developing HBase Applications Basics, Lesson 1: Java Client API, Part 1.

## Learning Goals



## Learning Goals



- 7.1 Connect a Client to HBase Tables
- 7.2 Perform CRUD Operations
- 7.3 Describe Versioning

In this lesson you will learn how to access data with the Java HBase API. We will describe a sample application that we will use in this lecture and lab exercises.

When you have finished with this lesson, you will be able to:

- Connect a client to HBase tables
- Perform standard CRUD operations on an HBase table, which are Create, Read, Update and Delete, and define the various helper classes using the Java API
- And finally, you will be able to describe the process of cell versioning within the tables

## Learning Goals



### 7.1 Connect a Client to HBase Tables

- 7.2 Perform CRUD Operations
- 7.3 Describe Versioning

First, we will look at how the client connects to an existing HBase table.

## Use Case

### Shopping Cart Application

- Store Item Inventory
- Store shopping cart data
- Tables: Inventory and Shoppingcart



We want to build a shopping cart application for a client, the Big Office Supply Company, which sells their office supply products through different outlets. They have their own stores around the country, as well as an online shop.

The company sells office supplies such as pens, notepads, pencils, and erasers. When building this shopping cart, we need somewhere to store the inventory of items in the store, collect items in the customer shopping cart as they browse the site, and update the inventory when they make a purchase.

To build this cart, we will use two HBase Tables: Inventory and Shoppingcart.

## Inventory and Shoppingcart Tables

Inventory Table

RowKey	Stock	
	quantity	price
pens	24	95
notepads	54	125
erasers	15	100

Shoppingcart Table

RowKey	Items		
	pens	notepads	erasers
Mike	10	9	1
John	5	12	
Mary	15	10	
Adam	10	1	

The row key for the Inventory table is the item ID, such as pens, notepads or erasers. The Inventory table has one column family Stock with the dynamic columns for quantity and price values.

The Shopping Cart table row key is the user ID, such as Mike or John. This table also has one column family Items, which includes the dynamic columns for storing the amount of each item in the user's cart.

## Application Requirements

- Create Cart Tables
- Create, Read, Update, and Delete Items from tables
- Add Advanced Functionality

To build our shopping cart application, then we need to:

- Create the Inventory and Shoppingcart tables
- Perform Create, Read, Update and Delete operations on these tables
- and finally, make sure that we maintain data consistency. When we checkout the items for a customer, we need to update the inventory.

As we go through the lessons of this course, we will visit the different parts of this process. In this lesson, we will focus on the second one, performing operations on existing HBase tables.

## Java Client Uses

- HTableInterface for Data operations on table
  - Put, Get, Scan, Delete, CheckAndPut, Increment
- HBaseAdmin for DDL operations
  - Create Table, Alter Table, Enable/Disable
- org.apache.hadoop.hbase.client

The main interfaces included in the HBase client package are the HtableInterface and HBaseAdmin.

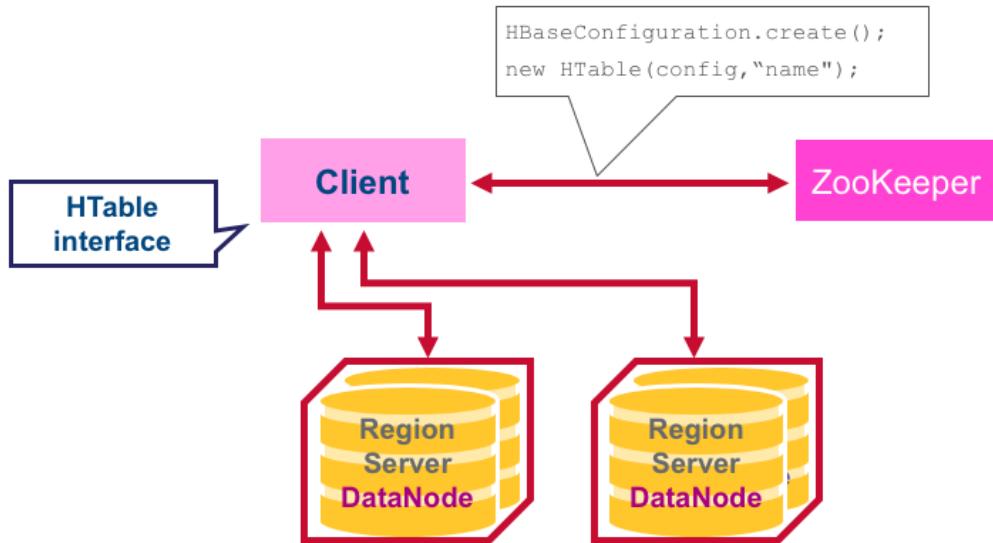
A Java client uses the HTableInterface to communicate with an existing HBase table, similar to a handle. The client uses the HTableInterface for Create, Read, Update, and Delete type of operations on table data.

The client uses the HBaseAdmin interface for creating, altering or deleting tables.

# Review



## Review



To review from DEV 320 HBase Architecture, when a client wants to read or write to an HBase table, they must first get the region information from ZooKeeper. The first time a client reads or writes to HBase, it connects to ZooKeeper to get the the location of the HBase Catalog table, called the Meta table. The Meta table provides the client with the RegionServer corresponding to row key ranges. Once the client receives this information, it is cached so the client can return to the RegionServer without the need for additional lookups.

## Setup Configuration Object

Get configuration object to tell program (client) where to connect:

- HBaseConfiguration reads hbase-site.xml on CLASSPATH
- /opt/mapr/hbase/hbase-0.98/conf/hbase-site.xml

```
Configuration config = HBaseConfiguration.create();
//new() HTable object to connect to foo table
HTableInterface table = new HTable(config, "/path/table");
```

In our Java code, we create a configuration object, HBaseConfiguration, which will read the location of ZooKeeper from the hbase-site.xml configuration file on class path.

For a MapR cluster, this file is located in the directory shown here. Note that the path includes the current version of HBase on your cluster.

We can get an instance of the configuration object by calling the create method from HBaseConfiguration. Once you have a configuration object instantiated, you pass it as a parameter in the HTable constructor, as shown here.

## Setup Configuration Object

```
<configuration>
  <property>
    <name>hbase.rootdir</name>
    <value>maprfs:///hbase</value>
  </property>
  <property>
    <name>hbase.zookeeper.quorum</name>
    <value>ip-10-170-227-11,ip-10-172-6-183,ip-10-170-97-198</value>
  </property>
  <property>
    <name>hbase.zookeeper.property.clientPort</name>
    <value>5181</value>
  </property>
</configuration>
```

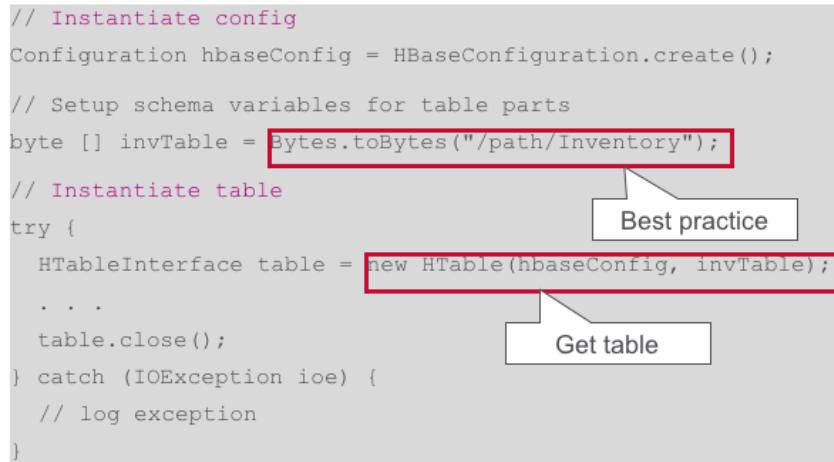
The hbase-site.xml file for client applications contains the location of ZooKeeper in the configuration information. It tells the client how to connect to ZooKeeper and the cluster, providing such details as the ZooKeeper quorum IP addresses and port number.

## Get HTableInterface For Existing Table

```
// Instantiate config
Configuration hbaseConfig = HBaseConfiguration.create();

// Setup schema variables for table parts
byte [] invTable = Bytes.toBytes("/path/Inventory");

// Instantiate table
try {
    HTableInterface table = new HTable(hbaseConfig, invTable);
    ...
    table.close();
} catch (IOException ioe) {
    // log exception
}
```



In a typical setup for working with an existing table, first we will instantiate an HBaseConfiguration object.

Next, we will setup components of the table that we are going to work with. These usually include the table name, and for MapR tables we will also specify the path to the table.

It is a best practice to set up our variables ahead of time rather than calling Bytes.toBytes every time we need to specify a table component. This will save resources and is more efficient. We will convert our Java strings to a byte array, to specify the various components of a table, including the table name, column families, column names, row key names, and a value.

Now we are ready to instantiate the table with its constructor.

Note that Apache HTable is NOT thread safe, for reads nor writes. Unlike HBase, however, an HTable instance in MapR-DB is thread safe and can be shared among multiple threads.

## Working With A Table Instance

```
// Retrieve values from rowA in columnFamily:columnName1
Get get = new Get(rowA);
get.addColumn(myCF, myCol); Use table
Result result = invTable.get(get);
...
// Insert value1 into rowA Use table
Put put = new Put(rowA);
put.addColumn(myCF, myCol, value1);
invTable.put(put);
...
// Release all resources associated with myTable
myTable.close();
```

Now we are ready to work with the table myTable which we just instantiated.

We can use the HTableInterface to perform standard CRUD operations, Create, Read, Update, and Delete.

First we retrieve some data with a get operation.

Next we insert some data with a put operation.

And finally we close the table to release all of the resources associated with it.

## Lab 7.1: Import, Build, and Run "lab-exercises-shopping" Project



### See Lab 7.1

Import the project "lab-exercises-shopping" into Eclipse  
Setup creates Inventory and Shoppingcart Tables and inserts data  
Use Get, Put, Scan, and Delete operations

## Learning Goals



## Learning Goals



7.1 Connect a Client to HBase Tables

### **7.2 Perform CRUD Operations**

7.3 Describe Versioning

Now we are going to look at CRUD operations from the Java API. As we go through, you will see some patterns in how you use the various methods.

## CRUD Operations Follow A Pattern (mostly)

### Common Pattern:

- Instantiate object for 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);
```

When we perform CRUD operations using the Java API, we often follow a similar pattern regardless of which operation we are performing.

For example,

- First we instantiate an object for the operation we are about to execute: put, get, scan or delete
- Then we add details to that object, and specify what we need from it. We do this by calling an add method and sometimes a set method.
- Once our object is specified with these attributes we are ready to execute the operation against a table. To do that we invoke the operation with the object we have prepared.

For example for a put operation we call `table.put()` and pass the put object we created as the parameter.

## Inventory Table

### Logical Model

RowKey	stock
	quantity
erasers	15
notepads	54
pens	24

### Physical Storage

Key	CF:COL	ts	value
erasers	stock:quantity	1391813876369	15
notepads	stock:quantity	1391813876369	54
pens	stock:quantity	1391813877777	24

We will look at how to create or update a row in the Inventory table using `HTable put()`. In this example we will use the Inventory table shown here, with the logical model on the left and the physical model, how it is stored on disk, on the right.

## Put Example

Key	CF:COL	ts	value
pens	stock:quantity	1391813877777	24

```
byte [] invTable = Bytes.toBytes("/path/Inventory");
byte [] stockCF = Bytes.toBytes("stock");
byte [] quantityCol = Bytes.toBytes ("quantity");
long amt = 241;
HTableInterface table = new HTable(hbaseConfig, invTable);

Put put = new Put(Bytes.toBytes ("pens"));

put.add(stockCF, quantityCol, Bytes.toBytes(amt));

table.put(put);
```

With this put operation, we insert the cell value shown in the diagram.

1. First, we convert the Java strings to byte arrays to specify the various components of the Inventory table:
  - the table name /path/Inventory
  - the stock column family
  - and the quantity column
2. Next, we create the HTable interface with the configuration object, and the table name
3. Then, we instantiate the put object, in this case with the pens row key
4. Then, call the put add method to add the stock column family, quantity column and the value of 24 long to the put object
5. Once the put object is specified with these attributes, we are ready to execute the put operation against the inventory table. We invoke the put operation with the put object by calling table.put(), and pass the put object we created as the parameter.

## Put Operation: Add Method

```
Put put = new Put(rowKey);
    ...
    put.add(columnFamilyName, columnName1, value1);
...
table.put(put);
```

Add Method

After we have instantiated a **put** object, we need to add the values that we want to insert, before we call the `HTable.put()` method.

## Put Operation: Add Method

Once **put** instance is created you call **add** method on it

Add **value** for specific **column** in **column family**

- ("column name" and "qualifier" mean the same thing)

```
Put add(byte[] family, byte[] qualifier, byte[] value)
```

Option: Set **timestamp** for cell

```
Put add(byte[] family, byte[] qualifier, long ts, byte[] value)
```

Once we have an instance of a **put** object for a specified row key, we will need provide some details, such as what value we want to insert or update. In general we will add a value for a column that belongs to a column family.

In the constructor for the **put** object, or in the **put add** method we can optionally set the timestamp to any long value, but usually we will let HBase set this.

## Put into the Shoppingcart Table

Shoppingcart Table

Row Key	Items		
	erasers	notepads	pens
Mike	2	5	1

Physical Storage

Key	CF:COL	ts	value
Mike	items:erasers	1391813876369	2
Mike	items:notepads	1391813876369	5
Mike	items:pens	1391813876369	1

Next we will look at the code to insert a row into the Shoppingcart table, which has column values for pens, notepads, and erasers.

## Put Operation

Key	CF:COL	ts	value
Mike	items:erasers	1391813876369	2
Mike	items:notepads	1391813876369	5
Mike	items:pens	1391813876369	1

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

table.put(new Put(Bytes.toBytes("Mike")).add(itemsCF, penCol,
    Bytes.toBytes(11)).add(itemsCF, noteCol,
    Bytes.toBytes(51)).add(itemsCF, eraserCol, Bytes.toBytes(21)));
```

We will use a similar process to when we updated the inventory table earlier, except that in this case we are adding three different column values; 2 erasers, 5 notepads, and 1 pen, to the row with row key, Mike, using a put object.

Each call to add() specifies the column family name, column name, and column value. For a put operation we will typically call add for each column in the row which we are updating.

## Bytes Class

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

Provides methods to convert Java types **to** and **from byte[ ] arrays**

Support for

- String, boolean, short, int, long, double, & float

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

byte[] amountBytes = Bytes.toBytes(10001);
long amount = Bytes.toLong(amount);
```

Everything in HBase is stored as bytes. The Bytes class is a utility class that provides methods to convert Java types to and from byte arrays.

The native Java types supported are String, short, int, long, double, and float.

We can use the Bytes class Bytes.toBytes method to boolean, to convert Java types for putting data into HBase. When reading data from HBase, we use the Bytes class Bytes.toLong(), Bytes.toString (), etc, for converting HBase bytes into Java types.

The HBase Bytes class is similar to the Java ByteBuffer class, but the HBase class performs all of its operations without instantiating new classes. Thus, it avoids garbage collection.

Visit the HBase documentation for more details about the Bytes class methods.

## Get Operation Pattern

Instantiate get object: `Get get = new Get(key)`

Specify what to get: `get.addColumn(...)`

Invoke get operation with HTable: `myTable.get(get)`

```
// Retrieve values from rowA in columnFamily:columnName1
Get get = new Get(rowKey);
get.addColumn(columnFamily, columnName1);
Result result = myTable.get(get);
```

When we look at reading data using the get operation, we will see a similar pattern to the put operation.

- First, we instantiate a get object with the row key for the row we want to read
- Then, we specify what to get by calling add methods on the get object
- Finally, we execute the get operation against a table by calling table.get(), and passing the get object we created as the parameter.

## Get Data from the Shoppingcart Table

Shoppingcart Table

RowKey	Items		
	erasers	notepads	pens
Mike	2	5	1

Physical Storage

Key	CF:COL	ts	value
Mike	items:erasers	1391813876369	2
Mike	items:notepads	1391813876369	5
Mike	items:pens	1391813876369	1

We will look at the code to read a row from the Shoppingcart table shown here, which has column values for pens, notepads, and erasers.

## Get Example

Key	CF:COL	ts	value
Mike	items:pens	1391813876369	5

```
byte [] tableName = Bytes.toBytes("/path/Shopping");
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 5
```

The **get** operation reads in the row with row key Mike, the column family items, and the column pens, and then prints out the value from that cell, which in this case is 5.

## Get Operation: Add and Set Methods

```
Get get = new Get(rowA);  
get.addColumn(columnFamilyName, columnName1);  
get.addColumn(columnFamilyName, columnName2);  
  
Add & Set Methods  
  
Result result = myTable.get(get);  
  
byte[] val =  
result.getValue(columnFamilyName, columnName1);  
  
System.out.println("Value: " + Bytes.toString(val));
```

Next we will talk about the **add** and **set** methods on the **get** object.

## Get Operation: Add and Set Methods

```
get.addColumn(columnFamilyName, columnName1);
```

Using just get object will **return everything** for row

To **narrow** down results call one or more:

- **addFamily**: get **all columns** for specific **family**
- **addColumn**: get **specific column**
- **setTimeRange**: retrieve columns in **range of** timestamps
- **setTimestamp**: retrieve columns with **specific timestamp**
- **setMaxVersions**: **set number of versions** to be returned
- **setFilter**: add filter

If we use the **get** object without specifying what we want to get, then everything in the row will be returned. Normally we will want to narrow down what columns or cells that are returned.

To do this,

- We call **addFamily** to get all the columns for a column family
- And call **addColumn** to get the value in a specific column of the row

If we want to be more precise, then we should call one of the set methods to be more specific. We can also control what timestamp or time range we are interested in, and how many versions of the data we want to see. We can even add a filter to the **get** operation.

## Get Object: Add and Set Methods Signatures

Method Signature	Description
Get addColumn(byte[] family, byte[] qualifier)	Get specified <b>column</b> from specified <b>family</b>
Get addFamily(byte[] family)	Get <b>all columns</b> from specified <b>family</b>
Get setMaxVersions()	Return <b>all versions</b>
Get setMaxVersions(int maxVersions)	Return <b>specified number of versions</b> of each column
Get setTimeRange(long minStamp, long maxStamp)	<b>Return versions in specified timestamp range</b>
Get timeStamp(long timestamp)	Return versions with <b>specified timestamp</b>

<http://hbase.apache.org/apidocs/org/apache/hadoop/hbase/client/Get.html>

This table shows the get add and set method signatures.

The addColumn method will get the column from the specified family,  
While the addFamily method will get all columns from the specified family.

setMaxVersions without a parameter means get will return all versions  
setMaxVersions with an int parameter means get will return up to the specified number of  
versions of each column

setTimeRange means get will return versions of columns only within the specified timestamp  
range

setTimeStamp means get will return versions of columns with the specified timestamp

For more information on the get operation, refer to the HBase documentation.

## Result: Retrieve Value From Result

Key	CF:COL	ts	value
adam	items:pens	1391813876369	18

```
public static final byte[] ITEMS_CF= Bytes.toBytes("items");
public static final byte[] PENS_COL = Bytes.toBytes("pens");

Get g = new Get(Bytes.toBytes("Adam"));
g.addColumn(ITEMS_CF , PENS_COL);
Result result = table.get(g);
byte[] b = result.getValue(ITEMS_CF, PENS_COL);

long valueInColumn = Bytes.toLong(b); // 18
```

In this `get` operation, we have narrowed down the request to the pens column. Once we get the result back from `table.get()`, we invoke one of the convenience methods on the result object. In this case we call `getValue`, to retrieve the value.

## Result Class

	Items:erasers	Items:notepads	Items:pens
Mike	10	7	18

Result instance: **Wraps** data from **row** returned from **get** or **scan** operation and wraps KeyValues

**Result toString():**

```
keyvalues={Mike/items:erasers/1422371668327/Put/vlen=8/mvcc=0,  
          Mike/items:notepads/1422371668327/Put/vlen=8/mvcc=0,  
          Mike/items:pens/1422371668327/Put/vlen=8/mvcc=0}
```

**Result** object provides methods to return values:

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

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

When we call **get()** on a table a result instance containing the requested data is returned. The result instance wraps cell data from a row returned from get or scan operations as key values.

Calling **Result.toString()** will print out the cell key values in the result and can be useful for debugging.

In this example, we see the **Result.toString()** output from performing a get on the Shoppingcart row, with row key Mike.

**result.getValue()** will get a specific column value from the result instance.

The result object provides more methods to manipulate the returned data. You can learn more about result in the HBase documentation.

## Key value: The Fundamental HBase Type

Key value implements Cell Interface

- Contains Key (cell coordinates) and Value (data)

Cell coordinates: Row key, Column family, Column qualifier, Timestamp

Key value `toString()`:

Mike/items:erasers/1422371668327/Put/vlen=8/mvcc=0

Key =Cell Coordinates					Value
Row key	Column Family	Column Qualifier	Timestamp	Value	
Mike	items	erasers	1422371668327	10	

Key value is the fundamental HBase type. It represents a value and its key, which is referred to as the cell coordinates.

When we successfully insert a record using a `put` operation, we save data that can be represented in a key value object.

When we perform a `get` operation, it returns a `result` object that is backed by key value instances.

## Delete Operation: Example

Constructing a  
delete object

```
Delete delete = new Delete(rowA);  
  
delete.deleteColumns(columnFamily, columnName1);  
  
myTable.delete(delete);
```

The typical pattern for deleting data from a row with the delete operation includes:

- First, instantiate a delete object with the row key
- Next, specify which columns or cells to delete from a row. If we do not specify which cells to delete, then the delete operation will delete the entire row.
- Finally, perform the delete operation on the HBase table.

## Delete Operation: Specify What To Delete

```
Delete delete = new Delete(rowA);  
  
delete.deleteColumns(columnFamily, columnName1);  
  
myTable.delete(delete);
```

As with the **get** and **put** operations, we can refine the cells that are affected by a **delete** operation.

## Delete Operation: Specify What To Delete

```
delete.deleteColumns(columnFamily, columnName1);
```

Just calling delete() with a Delete instance will delete **all columns and versions in row**

deleteFamily

- Removes entire **column family**
- Removes versions <= timestamp if supplied

deleteColumns

- Removes **multiple versions of specified column**
  - All versions if no timestamp given
  - Removes versions <= timestamp if supplied

deleteColumn

- Most **current version** if no timestamp given
- Matching the timestamp if supplied

Calling delete() with a delete instance will delete an entire row, and all of its columns and versions.

We can delete a column family, and all of its columns, by using one of the delete family methods.

We can also delete individual columns by using either the deleteColumn or deleteColumns methods.

Each of these methods can be used with or without a timestamp parameter.

## Delete Operation: DeleteXYZ Methods

Method Signature	Description
Delete deleteFamily(byte[] family)	Delete all versions of <b>all columns of the specified family</b> . A timestamp of 'now' is added.
Delete deleteFamily(byte[] family, long timestamp)	Same but delete all versions with a <b>timestamp less than or equal</b> to the one passed.
Delete deleteColumns(byte[] family, byte[] qualifier)	Delete <b>all versions</b> of the <b>specified column</b> . A timestamp of 'now' is added.
Delete deleteColumns(byte[] family, byte[] qualifier, long timestamp)	Same but delete all versions with a <b>timestamp less than or equal</b> to the one passed.
Delete deleteColumn(byte[] family, byte[] qualifier)	Delete the <b>latest version</b> of the <b>specified column</b> .
Delete deleteColumn(byte[] family, byte[] qualifier, long timestamp)	Delete the <b>specified version</b> of the <b>specified column</b> .

The table shown here lists the delete method signatures.

When specifying a timestamp, deleteFamily and deleteColumns will delete all versions with a timestamp less than or equal to that passed.

If no timestamp is specified, an entry is added with a timestamp of now, where now is the server's System.currentTimeMillis().

When specifying a timestamp, deleteColumn will delete versions only with a timestamp equal to the one passed.

If no timestamp is specified, deleteColumn only deletes the latest version of the column provided. This may be expensive as the server needs first to find the latest version by doing a get.

## Knowledge Check



## Knowledge Check



What are the operations that can be used to add/change/delete data in HTable

- 1) put
- 2) get
- 3) delete

What are the operations that can be used to add/change/delete data in HTable?

## Knowledge Check



What are the operations that can be used to add/change/delete data in HTable

- 1) **put**
- 2) **get**
- 3) **delete**

What are the operations that can be used to add/change/delete data in HTable?

put – correct  
get – incorrect  
delete - correct

## Scan Operation: Example

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

Scan scan = new Scan(startRow, stopRow);

scan.addFamily(columnFamily);

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

Scan is a very powerful operation that allows us to retrieve data from multiple rows.

The standard pattern for reading data with the HTable Scan operation is as follows:

- First, we construct a scan object
- Next, we specify which cells from the rows to read
- And then call htable.getScanner passing the scan object.

The result ResultScanner is returned, which lets us iterate over result objects.

## Scan Operation: Constructor Details

startRow - row to **start** scanner **at** (inclusive)

stopRow - row to **stop** scanner **before** (exclusive)

Constructor	Description
<code>Scan()</code>	Iterates over all rows, starting at beginning of table
<code>Scan(byte[] startRow)</code>	Start row is inclusive
<code>Scan(byte[] startRow, byte[] stopRow)</code>	Start row is inclusive, end row is exclusive

We can specify an optional startRow and stopRow for a scan operation. If no rows are specified, using the empty constructor, scan iterates over all rows, starting at the beginning of the table.

The start row is always inclusive, meaning the scan will match the first row key that is equal to or larger than the given start row.

The stop row is exclusive meaning the scan stops when the current row key is equal to or less than the optional stop row.

## Scan Operation: Add and Set Methods

```
Scan scan = new Scan();  
  
scan.addFamily(columnFamily);  
  
ResultScanner scanner = myTable.getScanner(scan);
```

Similar to our other operations, **scans** can also be limited to certain column families or columns.

## Scan Operation: Add and Set Methods

Method Signature	Description
Scan addFamily(byte [] family)	Get all columns for <b>specified column family</b>
Scan addColumn(byte[] family, byte[] qualifier)	Return data only for <b>specified column</b> in specified column family
Scan setTimeRange(long minStamp, long maxStamp) throws IOException	Get versions of columns only in <b>specified timestamp range</b> (minStamp, maxStamp)
Scan setTimeStamp(long timestamp)	Get versions of columns with <b>specified timestamp</b>
Scan setMaxVersions()	Returns <b>all versions</b>
Scan setMaxVersions(int maxVersions)	Get up to <b>specified number of versions</b> of each column

The default, empty constructor, will read the entire table, including all column families and their columns.

We can narrow down the scan by column or column family.

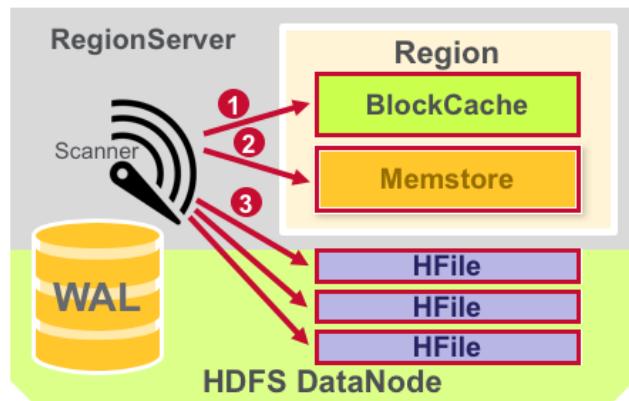
- Scan addFamily will return all columns for a specified column family
- Scan addColumn will return data only for a specified column in a specified column family

Table data is stored in column families, and column families are physically stored in separate files. When we omit a column family, scan will not read the file for that family. By specifying which column families to scan, we take advantage of the strength of column-family oriented architecture, saving time and resources.

## Scanner Merge from Memory, Files

Get or Scan must search for row cell key values in BlockCache Memstore and HFiles, and excludes the deleted cells

Specify certain versions or columns to save I/O



A read merges key values from the BlockCache, which is recently read cells in memory, the Memstore, which is recently inserted/updated cells in memory, and HFiles, which are cells on disk.

We can reduce the amount of data that we need to read off the disk or transfer over the network.

- Specify the row key to find the cells, since the row key is indexed
- Specify the column family for what part of the row to read, thereby reducing the number of HFiles read, if the row spans multiple families
- Specify the column name to reduce the number of columns returned to the client, thereby saving on network I/O

As we provide more criteria for the coordinates we do less work and return less data.

## ResultScanner: Example

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

Let's look at the ResultScanner object that is returned from a HTable getScanner operation.

A ResultScanner is like a Java iterator that will iterate over result instances.  
We can process the results by calling the ResultScanner next() method.

If we forget to close [ResultScanners](#) we can cause an error on the RegionServers.

A best practice to avoid performance problems is to always have ResultScanner process enclosed in the finally of a try/catch block. this way it will always complete, even if an exception is thrown.

## ResultScanner: Interface Overview

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

Resultscanner provides **iterator-like** functionality:

- Matching rows are returned on row basis,  
wrapped in Result object

ResultScanner methods:

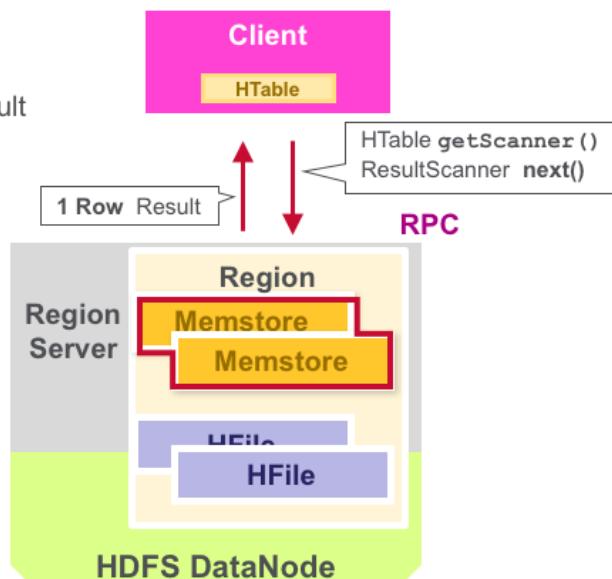
Method Signature	Description
Result <b>next()</b> throws IOException	Returns the <b>next row</b> .
Result[] <b>next(int nbRows)</b> throws IOException	Returns 0 < <b>number of rows</b> specified, depending on how many found
<b>void close()</b>	Closes scanner (put in <b>finally</b> clause of a try/catch)

The methods available in the ResultScanner interface include

- next() which will return the next row
- next(int nbRows) returns up to the number of rows specified, or zero if no rows are found or the number of rows found is less than the number specified. This can happen if the scan reaches the end of the table or the stop row.
- the close() method must be called to release all resources held by a scan.

## HBase Scan Remote Procedure Calls

ResultScanner  
`next()` calls return a single instance of Result



Scans do not ship all the matching rows in one RPC to the client, but instead do this on a row basis.

The ResultScanner wraps the result instance for each row into an iterator functionality.  
The `next()` calls return a single instance of result, representing the next available row.

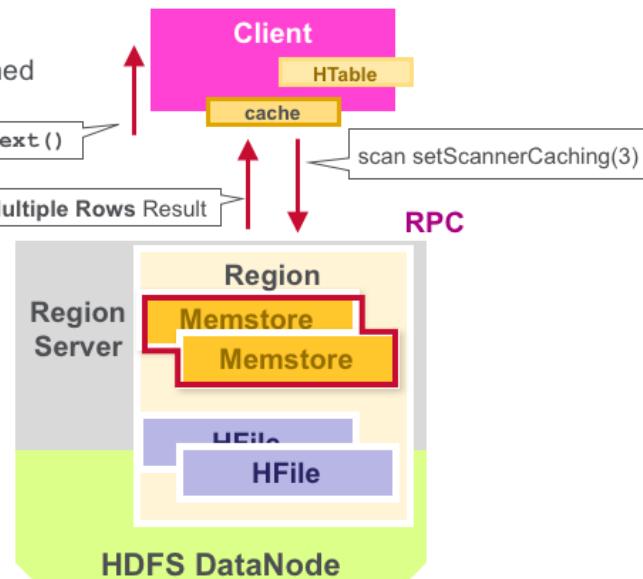
We can fetch a larger number of rows using the `next(int nbRows)` call, which returns an array of up to `nbRows` items, each an instance of result, representing a unique row.

Each call to `next()` will be a separate RPC for each row, even when using the `next(int nbRows)`.

## HBase Scanner Caching

With caching:

- Multiple rows returned
- Fewer RPC
- More memory



We can enable scan caching to control the number of rows returned per RPC call. By calling `setScannerCaching(number)` on the scan object, the specified number of rows will be returned in one RPC call and cached on the client side.

The `next()` calls will then return a single instance of result. This represents the next available row from the cache, until the cache is empty, then another RPC call will be made.

Scanner caching uses memory on the client. Setting the scanner caching higher will improve scanning performance most of the time, but setting it too high can cause an `OutOfMemoryException`.

## ResultScanner Caching: Example

```
Scan scan = new Scan();           Set to Return 3 rows
scan.addFamily(columnFamily);
scan.setCaching(3);
ResultScanner scanner = myTable.getScanner(scan);
for (Result res : scanner) {
    System.out.println(res);
}
scanner.close();
```

Calls scanner.next() after 3 rows, makes RPC

In this example of scan with scanner caching, `scan.setCaching` with a parameter of three, specifies for three rows to be returned in one RPC call and cached on the client side.

The `next()` calls will then return a single instance of result. As before, this represents the next available row from the cache until the cache is empty, then another RPC call will be made.

## Scanner Caching

Scanner caching controls **number of rows returned**

Set **scan level** caching using these **Scan** calls:

```
void setCaching(int caching)
```

With MapR-DB:

- long scans **auto-detected**
- client increases RPC size to optimize
- happens automatically
- when it will improve client performance

With MapR-DB, long scans are auto-detected, and the client increases the RPC size to optimize for them. While this happens automatically with MapR-DB, while we need to set this value with HBase.

## Lab 7.2: Insert and Get Data in the ShoppingCartDAO Class



This shows the Lab Exercise Program Structure which is explained in detail in the Lab Guide.

## Learning Goals



## Learning Goals



- 7.1 Connect a Client to HBase Tables
- 7.2 Perform CRUD Operations
- 7.3 Describe Versioning**

## Versioned Data

Key	CF:Col	version	value
smithj	Address:street	v3	19 <sup>th</sup> Ave
smithj	Address:street	v2	Main St
smithj	Address:street	v1	 Central Dr

Versioning is built-in with HBase. A put is both an insert, or create, and an update and each one of these actions gets stored as its own version.

Deleting a row places a tombstone marker on blocks of data. The tombstone marker prevents the data being returned in queries. The Physical data is deleted later during merge compactions.

Get requests return a specific version or versions based on parameters. If no parameters are specified, the most recent version is returned.

We can configure how many versions we want to keep, per column family, with a default of 1 version. When the max number of versions is exceeded, extra records will be eventually removed. We can specify which extra versions will be deleted.

## Sparse Data With Cell Versions

	CF1:colA	CF1:colB	CF1:colC
Row1	@time7: value3 @time5: value1 @time3: value1		
Row10	@time2: value1	@time2: value1	
Row11	@time6: value2 @time3: value1		
Row2	@time4: value1		@time4: value1

Table cells are versioned, uninterpreted arrays of bytes. The version is by default a timestamp that is a long. For every row:family:column coordinate, there can be multiple versions of the value.

Each **update**, **put** or **delete**, adds a new cell with a new version, which is by default the current time in milliseconds.

## Knowledge Check



## Knowledge Check



Indicate which of the following are true about Versioned Data in Hbase:

1. Each put action, insert, create and update, stores a new version of the cell value
2. Deleted cells are removed immediately
3. New versions are stored as the current time in milliseconds
4. When the Max number of versions is exceeded, old records are marked for deletion

## Knowledge Check



Indicate which of the following are true about Versioned Data in Hbase:

1. Each put action, insert, create and update, stores a new version of the cell value
2. Deleted cells are removed immediately
3. New versions are stored as the current time in milliseconds
4. When the Max number of versions is exceeded, old records are marked for deletion



## Next Steps

### DEV 330 – Developing HBase Applications: Basics

Lesson 8: Java Client API, Part 2

Congratulations! You have finished DEV 330, Lesson 1: Developing HBase Applications Basics – Java Fundamentals Part 1.

Continue on the Lesson 2 of this course to learn more about working with the Java API in HBase.



## DEV 330 – Developing HBase Applications: Basics

Lesson 8: Java Client API, Part 2

Winter 2017, v5.1

Welcome to DEV 330, Lesson 2: Developing HBase Applications Basics – Java Fundamentals Part 2.

## Learning Goals



## Learning Goals



- 8.1 Use Client-side Write Buffer
- 8.2 Perform HTable Batch and List Operations
- 8.3 Design CheckAndPut Operations
- 8.4 Define the KeyValue Object
- 8.5 Use the Result Object

When you have finished with this lesson, you will be able to:

- Use the client-side write buffer by turning off autoFlush,
- Perform HTable batch and list operations,
- Design checkAndPut operations for our shopping cart,
- Define how the key value object wraps HBase byte arrays, and what information you can obtain from this object,
- And use the result object to view and manipulate data stored in HBase.

## Learning Goals



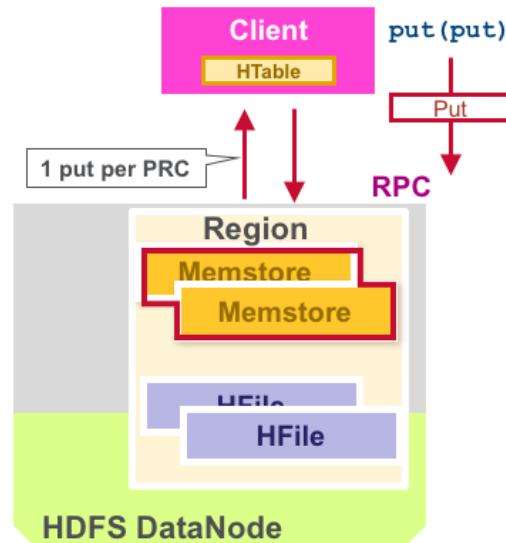
- 8.1 Use Client-side Write Buffer**
- 8.2 Perform HTable Batch and List Operations
- 8.3 Design CheckAndPut Operations
- 8.4 Define the KeyValue Object
- 8.5 Use the Result Object

## HBase Client Side Write No Buffer

Each put operation makes a RPC call

Round-trips are expensive

```
HTable.setAutoFlush(false)  
• Activates client buffer
```



By default each call to put() is wrapped and sent to the server using a Remote Procedure Call, or RPC.

As part of the HBase API there is a client side write buffer, which when turned on, will collect put operations so that they will be sent in one RPC call to the servers.

Using the client-side buffer is controlled by an autoFlush property on the HTable, which is set to true by default. Therefore, by default each put will be flushed and sent in its own RPC call.

This can be changed by calling HTable.setAutoFlush(), and setting the parameter to false, which activates the client-side write buffer to collect put operations.

## API Performance Tips: RPC Data



Network round-trips are expensive

Each single operation means a trip from the client to the server and back



For data throughput, it is better to have **fewer RPC** calls with **more data** compared to **lots of RPC** calls with **small amounts of data**

When updating small records, we will get better performance by grouping puts into fewer RPCs.

When autoFlush is on, each RPC will be sent one at a time, whether we will use a single put or a list of put operations.

Disabling autoFlush will prevent having each call being sent separately to the server, and the put operations will be sent when the write buffer is full instead.

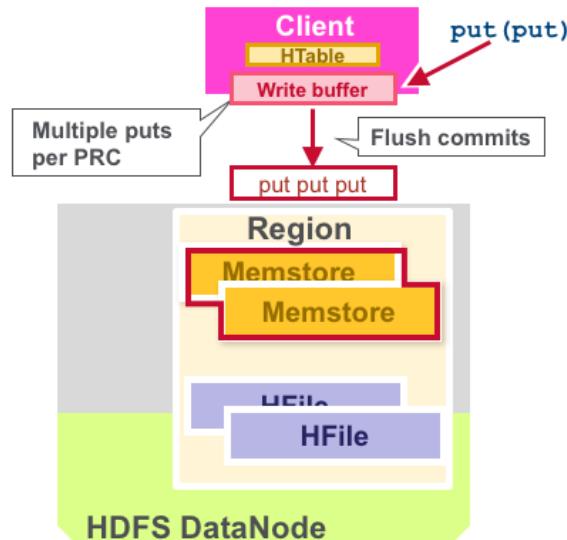
## HBase Client Side Write Buffer

### Enable client-side write buffer:

- `myTable.setAutoFlush(false)`
- Collects put operations into write buffer

### Flush data from put operations :

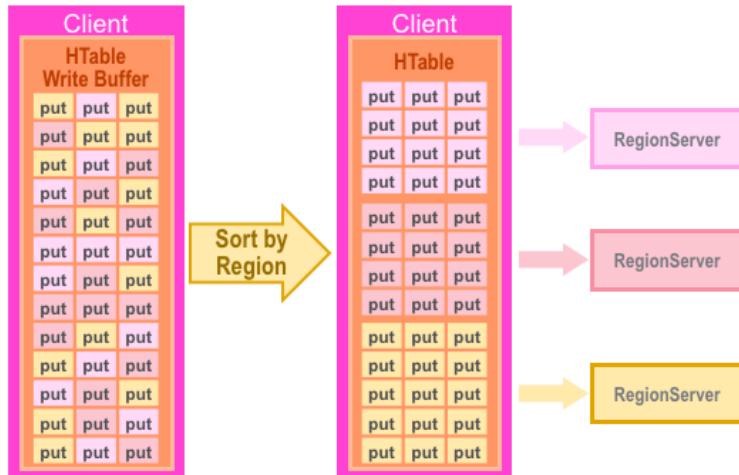
- `void flushCommits()`
- Sends buffered updates to remote server



We can activate the client side write buffer by invoking `table.setAutoFlush` and setting it to false. The client-side write buffer will then collect put operations in the write buffer.

When `flushCommits()` is called, the buffered updates will be sent in one RPC call to the server or servers. In addition, the client buffer will be flushed whenever the maximum write buffer size is reached, or when `table.close()` is called.

## HBase Client-side Write Buffer



Source: Diagram from Lars George's HBase: The Definitive Guide.

The client-side, write-buffered put instances may span different regions. Under the covers the put instances are batched to the RegionServer corresponding to the region.

Note that the client-side buffer is held on the client side, so if the client is terminated, for example it crashes, everything that was in the write buffer is lost.

## Put Client-side Write Buffer: Example

```
myTable.setAutoFlush(false);
Put put1 = new Put(rowA);
put1.add(columnFamily, columnName1, value1);
myTable.put(put1);
Put put2 = new Put(rowB);
put2.add(columnFamily, columnName1, value2);
myTable.put(put2);
Get get = new Get(rowA);
Result res1 = myTable.get(get);
System.out.println("Result: " + res1);
// prints Result: keyvalues=NONE !
myTable.flushCommits();
Result res2 = myTable.get(get);
System.out.println("Result: " + res2);
```

Here is an example of using flushCommits.

Invoking table.setAutoFlush(false) activates the client-side write buffer b.

Two rows are put in the table with myTable.put.

res1 = myTable.get(get); gets the row with rowkey rowA. However the result returns keyvalues=NONE, which is an empty row because flushCommits has not been invoked yet.

myTable.flushCommits(); is called, which will cause the buffered updates to be sent in one RPC call to the servers.

Result res2 = myTable.get(get); gets the row with rowkey rowA again. Res2 will display the contents of rowA, now that flushCommits has been called.

## Knowledge Check



## Knowledge Check



Indicate which of the following statements are true:

1. When autoFlush is ON, each **put** will be wrapped in its own RPC.
2. Grouping **puts** by having autoFlush OFF will yield better performance.
3. If the client side terminates for some reason, everything in the write buffer will be lost.
4. The autoFlush property is turned OFF by default.

## Knowledge Check



Indicate which of the following statements are true:

1. When autoFlush is ON, each Put will be wrapped in its own RPC.
2. Grouping Puts by having autoFlush OFF will yield better performance.
3. If the client side terminates for some reason, everything in the write buffer will be lost.
4. The autoFlush property is turned OFF by default.

## Learning Goals



## Learning Goals

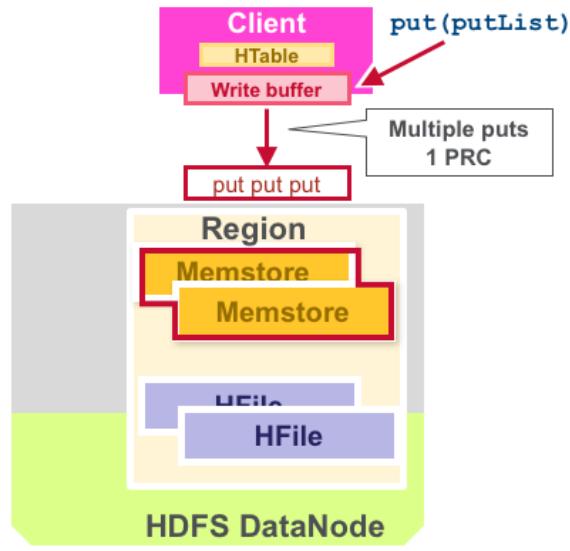


- 8.1 Use Client-side Write Buffer
- 8.2 Perform HTable Batch and List Operations**
- 8.3 Design CheckAndPut Operations
- 8.4 Define the KeyValue Object
- 8.5 Use the Result Object

## HTable Put List

HBase API provides 2 ways to **batch** operations

- `HTable put(), get(), delete()`  
`list methods:`
  - `void put(List<Put> puts)`
  - `Result[] get(List<Get> gets)`
- If `isAutoFlush` is false, the update is buffered until the write buffer is full.



The `putList` method is shown here.

Another way to get better performance with fewer RPCs is to use the `HTable putList` or `Batch` methods.

The `HTable put`, `get`, and `delete` List methods let us pass a list of objects for updating or reading.

If `isAutoFlush` is false, the update is buffered until the write buffer is full.

Puts added via `htable.add(Put)` and `htable.add( <List> Put)` will be in the same write buffer.

If `autoFlush = false`, these messages are not sent until the write-buffer is filled.

To explicitly flush the messages, call `flushCommits`.

Calling `close` on the `HTable` instance will invoke `flushCommits`.

## HTable List Of Puts Example

```
List<Put> puts = new ArrayList<Put>();  
Put put1 = new Put(rowA);  
put1.add(columnFamilyName, columnName1, value1);  
puts.add(put1);  
Put put2 = new Put(rowB);  
put2.add(columnFamilyName, columnName1, value2);  
puts.add(put2);  
table.put(puts);
```

Here is a List of puts example.

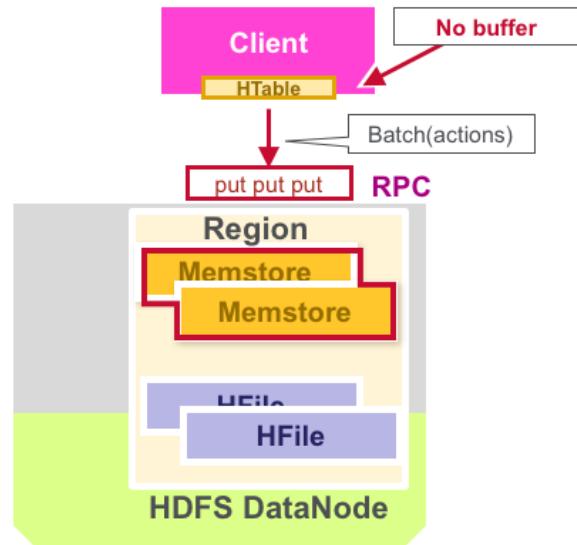
We create a list of put objects, in this case an ArrayList.

Then we can instantiate as many put objects as we need to, and add them to the list.

When we are done, we call HTable put, with the list as the parameter.

## HTable Batch Operations

- HBase API provides two ways to do **batch** operations
- Batch methods from HTable  
`void batch(List<Row> actions, Object[ ] results)`



MAPR Academy

© 2016 MapR Technologies

L8-17

When we use the htable batch() functionality, the included put instances will not be buffered using the client-side write buffer. The batch() calls are synchronous and send the operations directly to the servers. No delay nor other intermediate processing is used. This is different when compared to the put() calls, so choose which one you want to use carefully.

This is similar to the htable putList call. We pass in a List of get, put, or delete objects. Note that the Row class is the parent class for put, get and delete objects, and we can also pass a results array.

## Batch Methods From HTable

Method Signature	Description
<pre>void batch(List&lt;Row&gt; actions,           Object[] results) throws IOException, InterruptedException</pre>	Method that does batch call on Deletes, Gets, Puts, Increments
	<b>actions</b> - list of Get, Put, Delete, Increment
	<b>results</b> - Empty Object[ ], same size as actions. Provides access to results

The void batch method is shown here, with a list of get, put or delete actions, and a results array as its parameters.

When using the batch method

- gets, puts, and deletes are all supported
- The client-side write buffer is not used
- If there is a problem executing any action, a client-side exception is thrown, reporting the issues
- All batch operations are executed before the results are checked, even if one of the actions results in an error

## Result Values From HTable Batch Methods

Result	Description
null	Operation <b>failed</b>
EmptyResult	Returned for <b>successful Put</b> and <b>Delete</b> operations Displays as: keyvalues=NONE when printed
Result	Returned for <b>successful Get</b> operation. May also be empty if no matching row or column
Throwable	<b>Exception</b> from server returned as-is

This table shows what you can expect to be returned from HTable batch operations in the Object[] results parameter.

The results parameter gives access to the results of all succeeded operations, and the remote exceptions for those that failed. A null in the result array means the action failed even after retries.

## HTable Batch Method: Example

```
List<Row> list = new ArrayList<Row>();
Put put = new Put(rowA);
put.add(columnFamily, columnName1, value1);
list.add(put);

Put put = new Put(rowB);
put.add(columnFamily, columnName1, value2);
list.add(put);
...
Object[] results = new Object[list.size()];
try {
    table.batch(list, results);
} catch (Exception e) {
    e.printStackTrace();
}
```

The HTable batch method looks very similar to the example we looked at for a put with a list of put operations.

First, we create a list, in this case an ArrayList.

Then we instantiate as many put objects as we need to, and add them to the list.

When we are done, we call HTable batch with the list as the parameter.

## Knowledge Check



## Knowledge Check



Place the steps of using the `list` or `batch` method in the right order.

- A. Create a list of objects, such as an `ArrayList`
- B. Instantiate as many `put` objects as we need to
- C. Call `HTable batch` or call `HTable put`, the list as the parameter

## Lab 8.2:

### Work with Shoppingcart Application put list and batch



See Lab 8.2.

## Learning Goals



## Learning Goals



- 8.1 Use Client-side Write Buffer
- 8.2 Perform HTable Batch and List Operations
- 8.3 Design CheckAndPut Operations**
- 8.4 Define the KeyValue Object
- 8.5 Use the Result Object

## Inventory and Shoppingcart Tables before checkAndPut

Inventory Table

Row Key	Stock
	quantity
pens	24
notepads	54
erasers	15.

Shoppingcart Table

Row Key	Items		
	pens	notepads	erasers
Mike	5	5	
John	5	12	
Mary	15.	10	
Adam	10	1	

We want to implement checkout functionality for the Shoppingcart application tables shown here. There is one Shoppingcart per customer at any given time.

## Inventory and Shoppingcart Tables after checkAndPut for Mike

Inventory Table

Row Key	Stock	
	quantity	Mike
pens	24 19	5
notepads	54 48	6
erasers	15.	

Shoppingcart Table

Row Key	Items		
	pens	notepads	erasers
Mike	5	6	
John	5	12	
Mary	15.	10	
Adam	10	1	

Assumptions made for this applications are:

When the user places an order at checkout, the application will do the following:

- 1) Get the items for the user from the Shoppingcart table
- 2) For each item in the user's cart, get the inventory quantity to ensure that we have enough quantity for the user
- 3) Once we have seen that there is enough quantity in Inventory table, we use CheckAndPut to simulate a transaction in the Inventory table. We will:
  - add a new column for the userid, in this example Mike,
  - subtract the amount of items to purchase, 5 pens, from the Inventory quantity column
  - and reserve the amount of items to purchase under the Inventory userid column
- 4) We do this one row at a time because HBase only provides row atomicity
- 5) When the quantities have been reserved, we can remove the Shoppingcart entry for the user, and place the order

## The checkAndPut Method

```
boolean checkAndPut(byte[] row, byte[] family,  
byte[] qualifier, byte[] value, Put put)  
  
myTable.checkAndPut("pens", CF, COL, 24, put);
```

Check a Value

Inventory Table before

	CF "stock"
	quantity
pens	24

Put a row update

Inventory Table after

	"stock"
	quantity
pens	19

This example shows using a checkAndPut on the Inventory Table.

The checkAndPut method allows you to atomically check if a row/family/qualifier value matches an expected value. If it does, it updates the row with the put.

In this example, the check is made on the quantity column with the value 24. If the value is still 24, then the put updates the quantity column to 19 and adds the Mike column with 5.

## The checkAndPut Method: Atomic put with check

- Atomic operation guarded by a check
- **Check** is executed **first** then
  - if **success** **put** operation is executed
- Similar to java `compareAndSet(expectedValue, updateValue)`
- **Check and modify the same row**
- **Atomicity** guaranteed on single row
- **Method signature:**

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

- Use for concurrent transactions, multiple clients updating

The basic principle is that we read data from the HBase tables. When we want to update any value, we want to make sure that no other client has changed this value. We use the atomic check to compare that the value was not modified, and then apply our new value. This guarantees atomicity for a single row. This method returns a boolean true on success.

This method is similar to the atomic `compareAndSet()` method in Java. We will typically use this for concurrent transactions, such as checking if a ID exists before inserting it, or checking if an amount has changed before updating.

## The checkAndPut method

```
byte [] rowPens = Bytes.toBytes ("pens");
byte [] CF = Bytes.toBytes("stock");
byte [] COL = Bytes.toBytes ("quantity");
byte [] oldquantity= Bytes.toBytes(24);
byte [] amount= Bytes.toBytes(5);
byte [] newquantity= Bytes.toBytes(19);
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);
```

In this example, a check is made on the quantity column with the value 24. if the value is still 24, then the put updates the quantity column to 19 and adds the Mike column with 5.

## The checkAndPut Method

```
Put put1 = new Put(rowA);
put1.add(columnFamily, column1, value1);
// Passing null for value parameter means we expect
// there is no such column for this row.
boolean ret1 = myTable.checkAndPut(rowA, columnFamily,
    column1, null, put1);
System.out.println("Put succeeded: " + ret1);
// should be true if coll did not exist
Put put2 = new Put(rowA);
put2.add(columnFamily, column2, value2);
boolean ret2 = myTable.checkAndPut(columnFamily,
    column1, value1, put2);
System.out.println("Put succeeded: " + ret2);
// should be true if value1 has not been changed
```

In this example, the first checkAndPut checks if column1 is null. If it is, it puts value1 into column1.

The second checkAndPut checks if column1 is equal to value1. If it is equal, it puts value2 into column2.

ret2 will be equal to true if ret1 was successful, and if no one else changed the value since it was set.

Also for ret1 the checkAndPut call is an example of checking whether the column did not exist, or in other words there was no previous value there. We will only insert a value if there isn't one there.

## The checkAndDelete Method

- Special variation of Delete call at **table** level
- Method signature: `boolean checkAndDelete(byte[] row, byte[] family, byte[] qualifier, byte[] value, Delete delete) throws IOException`
- Atomic operation guarded by check
- Check executed first
- If success delete operation executed

In the HTable class, we also have the checkAndDelete method. This is similar to the put method we saw earlier. The two are often referred to as atomic compare-and-set operations abbreviated as CAS.

checkAndDelete gives us read-and-modify functionality on the server side. Similar to the put method, if the check fails, nothing is deleted and false is returned. Conversely if the check succeeds the delete is executed and the method returns true.

Passing a null value triggers the nonexistence test, meaning, the check is successful if the column does not exist.

Again this method has atomicity at the row level. Checking on one row key, while the supplied instance of delete points to another, will throw an exception.

## Knowledge Check





## Knowledge Check

Which of the following statements about the `checkAndPut` method are true?  
Check all that apply:

1. Guarantees atomicity for a single row.
2. Reads a value before performing a **put**.
3. Performs a **put**, and then compares the value to the input.
4. Compares that the value has not been changed before performing the **put**.

## Knowledge Check



Which of the following statements about the `checkAndPut` method are true?  
Check all that apply:

1. Guarantees atomicity for a single row.
2. Reads a value before performing a put.
3. Performs a `put`, and then compares the value to the input.
4. Compares that the value has not been changed before performing the `put`.

## Learning Goals

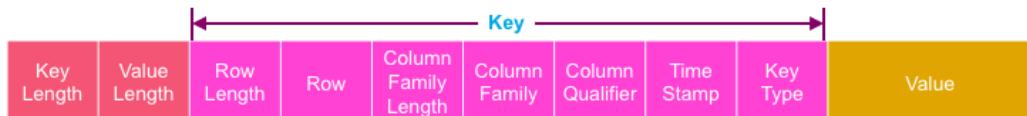


## Learning Goals



- 8.1 Use Client-side Write Buffer
- 8.2 Perform HTable Batch and List Operations
- 8.3 Design CheckAndPut Operations
- 8.4 Define the KeyValue Object**
- 8.5 Use the Result Object

## Cell Interface



KeyValue object: Implements Cell Interface

KeyValue: Contains data and cell coordinates

**KeyValue** `toString()`:

pens/stock:quantity/1422371668294/Put/vlen=8/mvcc=0

	Stock:quantity
pens	24

Cell coordinates

- Row key, column family, column qualifier, timestamp

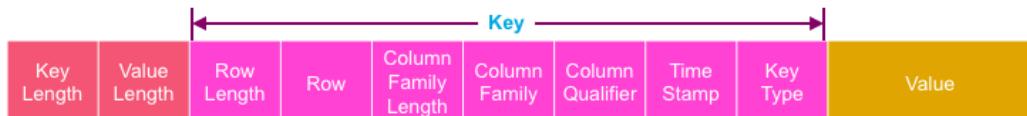
Data and coordinates are stored as byte[]

The KeyValue object implements the cell interface.

KeyValue objects contain the data as well as the coordinates of one specific cell.

Everything in HBase is stored as byte arrays, each KeyValue wraps the byte array of a particular cell.

## CellUtil Helper Methods



### CellUtil helper methods to get each part of Cell:

- They allocate new byte arrays and return copies
- `byte[] cloneFamily(cell)`
- `byte[] cloneQualifier(cell)`
- `byte[] cloneRow(cell)`
- `byte[] cloneValue(cell)`

### Cell interface:

- `byte[] getTimestamp()`

These helper methods allocate new byte arrays and return copies of the cell information.

`cloneRow`, with row as the parameter, refers to the row key. The key in the diagram refers to the coordinates of a cell, in their raw, byte array format.

## Knowledge Check



## Knowledge Check



Indicate which of the following statements about the `KeyValue` object are true.  
Check all that apply:

1. The `KeyValue` object stores the cell data
2. The `KeyValue` object stores the cell coordinates
3. The `KeyValue` is stored as a `long`

## Knowledge Check



Indicate which of the following statements about the **KeyValue** object are true.  
Check all that apply:

1. The **KeyValue** object stores the cell data
2. The **KeyValue** object stores the cell coordinates
3. The **KeyValue** is stored as a **long**

## Learning Goals



## Learning Goals



- 8.1 Use Client-side Write Buffer
- 8.2 Perform HTable Batch and List Operations
- 8.3 Design CheckAndPut Operations
- 8.4 Define the KeyValue Object
- 8.5 Use the Result Object**

## The Result Class

	Items:erasers	Items:notepads	Items:pens
Mike	10	7	18

Result object: **Wraps** data from **row** returned from **get** or **scan** operation and wraps key values

Result **toString()**:

```
keyvalues={Mike/items:erasers/1422371668327/Put/vlen=8/mvcc=0,  
Mike/items:notepads/1422371668327/Put/vlen=8/mvcc=0,  
Mike/items:pens/1422371668327/Put/vlen=8/mvcc=0}
```

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

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

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

A Result object wraps data from a row returned from a get or a scan operation.

The Result object wraps a row as an array of cell key values, and contains all of the cells for all of the columns from all of the column families that the get or scan request returned for that row.

Result.toString() can be useful for debugging, to see the key values that the Result contains.

The Result object provides several methods to manipulate the returned data.

## The Result Class: Retrieve Key Values

	Items:pens	Items:notepads	Items:erasers
Adam	18,7,3	7	10

```
public static final byte[] ITEMS_CF= Bytes.toBytes("items");
public static final byte[] PENS_COL = Bytes.toBytes("pens");

Get get = new Get(Bytes.toBytes ("Adam"));
get.setMaxVersions(3);

Result result = table.get(get);
//how many cells are returned below?
List<Cells> cells = result.getColumnCells(ITEMS_CF , PENS_COL);
byte[] v = result.getValue(ITEMS_CF , PENS_COL);
byte[] key = result.getRow();
```

In this example we use the Result method, getColumn, to get the key values for the pens column.

There are three versions stored in this pens column, and one for each of the notepads and erasers columns. When using the get object, five key values will be returned for Adam. get.setMaxVersions is set to 3, so values for all three versions in the pens column will be returned.

result.getValue(ITEMS\_CF , PENS\_COL) will return the last written version for the pens column, in this case 3.

result.getRow() will return the row ID, in this case Adam.

## The Result Class: Methods

Method Signature	Description
<code>List&lt;Cell&gt; getColumnCells(byte[] family, byte[] qualifier)</code>	Returns <b>Cells</b> for specified <b>column</b> . Returned list contains <b>0</b> (no value in row) or <b>1</b> the <b>latest</b> value. Or if requested more versions it will have more
<code>byte[] getValue(byte[] family, byte[] qualifier)</code>	Gets the <b>latest value</b> of specified <b>column</b>
<code>byte[] getRow()</code>	Returns <b>row key</b>

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

We will get a result instance from a get operation, or when we call next from a ResultScanner, from a scan operation.

`getColumnCells()` returns the key values for the specified column. The returned list contains zero, when the column has no value for the given row, or one entry, which is the newest version of the value. We can request more than one value, if we specified a value greater than one version to be returned.

`getValue()` gets the latest version of the specified column.

`getRow()` returns the row key.

## The Result Class: Retrieve Cells

	Items:pens	Items:notepads	Items:erasers
Adam	18,7,3	7	10

```
public static final byte[] ITEMS_CF= Bytes.toBytes("items");
public static final byte[] PENS_COL = Bytes.toBytes("pens");
Get get = new Get(Bytes.toBytes("Adam"));
get.setMaxVersions(3)
Result result = table.get(g);
System.out.println(result.size());
System.out.println(result.isEmpty());
List<Cell> kvs = result.listCells();
for (Cell kv : kvs) {
    System.out.println(kv);
}
```

Here is an example of using some more Result methods.

result.size() will return 5. There are five cells in this row, and get.setMaxVersions(3) sets the get to return a maximum of three versions per cell.

isEmpty() will return false.

listCells() will return a list of cells for the row.

## The Result Class: Methods

Method Signature	Description
Cell[] rawCells()	Returns the <b>sorted array of Cells</b> for the Result instance
public List<Cell> listCells()	Returns a <b>sorted list of the Cells</b> .
public int size()	Returns the size of the underlying Cell[]
public boolean isEmpty()	Checks if Cell array is empty: null or zero length

Methods for the Result class include:

rawCells() returns the sorted array of cells for the Result instance

listCells() returns a sorted list of the cells. This provides a convenient way to iterate over the results.

Size() returns the number of cells in the returned Cell[] array.

isEmpty() checks if cell array is empty. It will zero length if it is empty or null if the array is not empty.

## The Result Class: Retrieve Key Values

	Items:pens	Items:notepads	Items:erasers
Adam	18 ,7,3	7	

```
public static final byte[] ITEMS_CF= Bytes.toBytes("items");
public static final byte[] PENS_COL = Bytes.toBytes("pens");
Get get = new Get(Bytes.toBytes("Adam"));
get.setMaxVersions(3)
Result result = table.get(g);
Cell kv = result.getColumnLatestCell(ITEMS_CF , PENS_COL);
System.out.println(kv);
Boolean hasErasers = result.containsColumn(ITEMS_CF , Bytes.toBytes("erasers"));
System.out.println(hasErasers);
```

Here is an example of using some more Result methods

getColumnLatestCell returns the newest cell of the specified column. In contrast to getValue(), however, it does not return the raw byte array of the value, but the full key value instance instead. This may be useful when we need more than just the data. In this example getColumnLatestCell returns the key value for the cell Adam:Items:pens, which is 3

containsColumn is a convenience method to check if there was any cell returned in the specified column. In this case, it will return false.

## The Result Class: Column Oriented Methods

- Two column oriented methods besides `getColumn()`

```
Cell getColumnLatestCell(byte[] family, byte[] qualifier)  
boolean containsColumn(byte[] family, byte[] qualifier)
```

There are two other column oriented methods.

`getColumnLatest` returns the newest cell of the specified column. In contrast to `getValue()`, however, it does not return the raw byte array of the value, but the full key value instance instead. This may be useful when we need more than just the data.

`containsColumn` is a convenience method to check if there was any cell returned in the specified column.

## The Result Class: getMap()

Method Signature	Description
NavigableMap<byte[], NavigableMap<byte[], NavigableMap<Long, byte[]>>> <b>getMap()</b>	Returns entire result set in Java Map

Returns java **NavigableMap<Key,Value>**

- A SortedMap

```
Map <familyName,  
     Map<qualifierName,  
         Map<timestamp,value>>> map =result.getMap()
```

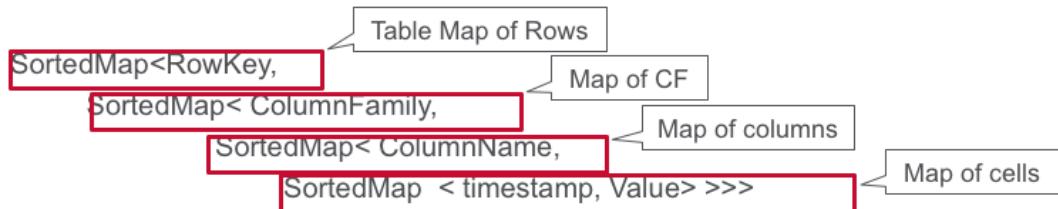
Result getMap() returns the entire result row data in a Java map, so we can iterate over it. The result may include multiple families and multiple versions.

The public interface, NavigableMap, with a key and value, extends [SortedMap](#), getMap

The Result row returned is a map, of a map, of a map. The outer map is keyed by the family name, and the middle-inner map is keyed by a column name, and the innermost map is keyed by timestamp.

## HBase Sorted Map Example

In Java, this is the table:

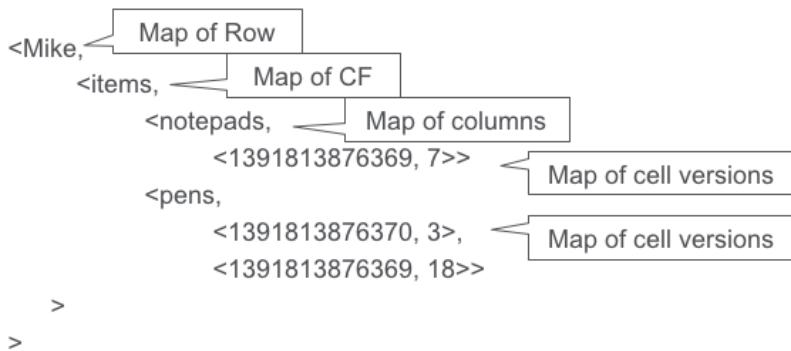


Key	CF:COL	ts	value
Mike	items:notepads	1391813876369	7
Mike	items:pens	1391813876369	3
Mike	items:pens	1391813876370	18

An HBase table is not a relational table. Instead, it is more like a map of maps, with an outer, sorted map keyed by a row key, and an inner, sorted map keyed by column name.

The Java representation of the table is seen here, showing how this table is a map, of a map, of a map.

## HBase NavigableMap Example



Key	CF:COL	ts	value
Mike	items:notepads	1391813876369	7
Mike	items:pens	1391813876370	3
Mike	items:pens	1391813876369	18

The Result object getMap() method will return a sorted NavigableMap of maps containing the row data.

The bottom table shows how the example row data is stored on disk. The code on the top shows how this row of data is returned in a Java sorted map of maps.

A map is keyed by the column family name, which contains a map keyed by column qualifier, which in turn contains a map of cells keyed by timestamp.

## Example: Result getMap()

	Items:pens	Items:notepads	Items:erasers
Mike	18,3	7	

```
// get column family map  
NavigableMap<byte[], NavigableMap<byte[], NavigableMap<Long, byte[]>>> cfMap =  
    result.getMap();  
// get qualifier map for items column family  
NavigableMap<byte[], NavigableMap<Long, byte[]>> qualMap = cfMap.get(CF);  
// Loop through column qualifiers  
for (byte[] qual : qualMap.keySet()) {  
    // get time stamp map  
    NavigableMap<Long, byte[]> tsMap = itemsQualMap.get(qual);  
    // loop through time stamps  
    for (Long tstamp : tsMap.keySet()) {  
        // get value  
        Long amt = Bytes.toLong(tsMap.get(tstamp));  
    }  
}
```

MA}

ologies

L8-55

In this example, the Result getMap() method returns the entire result set in a Java map so we can iterate over it.

Result getMap() returns a navigable map keyed by the column family name, which contains a map keyed by column qualifier, which contains a map of cells keyed by timestamp.

- Result getMap() returns the column family map
- In this example, the column family map gets a map of the columns, keyed by qualifiers, for the items column family
- For each column qualifier, it gets a map of the cells keyed by timestamps
- For each version timestamp, it gets the value

## The Result Class: Map Oriented Methods

### Return Nava NavigableMap

- SortedMap
- Can be Traversed in ascending or descending key order
  - descendingMap method
- <http://docs.oracle.com/javase/6/docs/api/java/util/NavigableMap.html>

Method Signature	Description
<code>NavigableMap&lt;byte[], byte[]&gt; getFamilyMap(byte[] family)</code>	Returns results for <b>specific family</b> only, including all versions
<code>NavigableMap&lt;byte[], NavigableMap&lt;byte[], NavigableMap&lt;Long, byte[]&gt;&gt;&gt; getMap()</code>	<b>Returns entire result set</b> in Java Map
<code>NavigableMap&lt;byte[], NavigableMap&lt;byte[], byte[]&gt;&gt; getNoVersionMap()</code>	Same but only the <b>latest version</b> of cell

getFamilyMap returns results for a specific family only, including all versions.

getMap returns the entire result set in a Java map, so we can iterate over it. The Result may include multiple families and multiple versions.

getNoVersionMap returns the same information as getMap, but includes only the latest version of a cell for each column.

Whether you use these map methods, or the other methods mentioned earlier, is a matter of style. Follow your established access patterns. There is no performance penalty in either technique, and the results have already been moved across the network from the server to your client process.

## Example: Result getFamilyMap()

```
Get get = new Get(Bytes.toBytes("Mike"));
result = shoppingcartTable.get(get);
//Returns a Map of the form: Map<qualifier,value>
NavigableMap<byte[], byte[]> nmap =
    result.getFamilyMap(itemsCF);
System.out.println("nmap.size="+ nmap.size());

pens = Bytes.toLong (nmap.get(penCol));
notepads = Bytes.toLong (nmap.get(notepadCol));
erasers = Bytes.toLong (nmap.get(eraserCol));
```

In this example of the getFamilyMap method:

- First we get the Result object, which wraps the row for the row key Mike
- Then we get all of the columns in the items column family, as a NavigableMap
- Finally, from the column NavigableMap, we get the values for the columns pens, notepads and erasers

## Knowledge Check





## Knowledge Check

Use the words and phrases below to complete this sentence:

The Result object wraps a row as an \_\_\_1\_\_\_, and contains \_\_\_2\_\_\_ for all of the columns from all of the column families that the get or scan request \_\_\_3\_\_\_ for that row. The result object provides several \_\_\_4\_\_\_ to manipulate the returned data.

- A. all of the cells
- B. methods
- C. array of Cell KeyValues
- D. returned

## Knowledge Check



Use the words and phrases below to complete this sentence:

The Result object wraps a row as an **array of Cell KeyValues**, and contains **all of the cells** for all of the columns from all of the column families that the get or scan request **returned** for that row. The result object provides several **methods** to manipulate the returned data.

- A. 1 :: array of Cell KeyValues
- B. 2 :: all of the cells
- C. 3 :: returned
- D. 4 :: methods

## Lab 8.5:

### Work with Shoppingcart Application checkout



Open your lab guide, and perform Lab 8.5.



## Next Steps

### DEV 330 – Developing HBase Applications: Basic

Lesson 9: Java Client API for Administrative Features

Congratulations! You have finished DEV 330, Lesson 8: Developing HBase Applications Basics – Java Fundamentals Part 2.

Continue on the Lesson 9 of this course to learn about the HBase Java API Admin interface.



## DEV 330 – Developing HBase Applications: Basics

Lesson 9: Java Client API for Administrative Features

Winter 2017, v5.1

Welcome to DEV 330, Developing HBase Applications Basics, Lesson 9, Java API Admin Interface and HBase compatibility.

## Learning Goals



## Learning Goals



- 9.1 Define Table and Column Family Properties
- 9.2 Create, Alter, and Delete Tables
- 9.3 Define HBase API Compatibility with MapR-DB

We talked about manipulating data in tables, similar to Data Manipulation Language, or DML in SQL. Now we are going to work with tables themselves, similar to Data Definition Language, or DDL in SQL.

When you have finished with this lesson, you will be able to:

- Define table and column family properties using the table and column family descriptor objects
- Create, alter, and delete tables using the HBaseAdmin class, which you use with the table and column family descriptors
- Define the HBase API compatibility with MapR-DB tables

## Learning Goals



### 9.1 Define Table and Column Family Properties

- 9.2 Create, Alter, and Delete Tables
- 9.3 Define HBase API Compatibility with MapR-DB

## Overview: Administrative API

Java client uses

- HBaseAdmin for DDL operations :
  - Create Table, Alter Table, Enable/Disable
- HTableDescriptor
  - Define table name
- HColumnDescriptor
  - Define Column Family properties

A Java client uses the HBaseAdmin interface to create, drop, list, enable, and disable tables, as well as to add and drop table column families.

We will use the HBaseAdmin interface with the HTableDescriptor, and with the HColumnDescriptor to define table properties.

## Administrative API Example

```
// Instantiate config
Configuration config = HBaseConfiguration.create();
// Setup schema variables for table parts
byte [] invTable = Bytes.toBytes("/path/Inventory");
byte [] stockCF = Bytes.toBytes("stock");
byte [] quantityCol = Bytes.toBytes ("quantity");
HBaseAdmin admin = new HBaseAdmin(config);
HTableDescriptor desc = new HTableDescriptor(TableName.valueOf(invTable));
HCColumnDescriptor c = new HCColumnDescriptor(stockCF);
desc.addFamily(c);
admin.createTable(desc);
```

```
Table Descriptor
Column Descriptor
```

```
HBaseAdmin
```

This example shows how the table descriptor, column descriptor, and HBaseAdmin can be used together to create the shopping application Inventory table.

- First, we instantiate HbaseAdmin with a configuration object
- We then create the table descriptor with the name of the table
- Create the column descriptor with the name of the column family
- Add the column descriptor to the table descriptor
- Then use HBaseAdmin to create a table

## Table Descriptor: Overview

`HTableDescriptor` holds **details** about **table** and **column families**

You create a descriptor with **table name**:

- Table name with forward slash denotes MapR-DB table
- For example: /path/to/myTable

Constructor	Description
<code>HTableDescriptor(TableName name)</code>	Construct table descriptor specifying <b>table name</b>
<code>HTableDescriptor(HTableDescriptor desc)</code>	Construct table descriptor by cloning descriptor passed as parameter

The `HTableDescriptor` class lets us control details about a table and its column families.

We can define column families by adding column descriptors to the `HTableDescriptor`, and create a table descriptor with a table name.

Table names follow the rules of file names.

MapR tables names are specified using a full path name, and we can also create a table with an existing descriptor.

## Table Descriptor: Column Family Methods

```
// Instantiate config
Configuration config = HBaseConfiguration.create();
// Setup schema variables for table parts
String invTable = "/path/Inventory";
byte [] stockCF = Bytes.toBytes("stock");
byte [] quantityCol = Bytes.toBytes ("quantity");

HBaseAdmin admin = new HBaseAdmin(config);
HTableDescriptor desc = new HTableDescriptor(TableName.valueOf(invTable));
HColumnDescriptor c = new HColumnDescriptor(stockCF);
desc.addFamily(c);
admin.createTable(desc);
```



Next we will look at the table descriptor column family methods.

## Table Descriptor: Column Family Methods

Method Signature	Description
<code>void addFamily(     HColumnDescriptor family)</code>	<b>Adds</b> column family
<code>boolean hasFamily(byte[] c)</code>	Checks if table has column family
<code>Collection&lt;HColumnDescriptor&gt;     getFamilies ()</code>	Returns list of <b>column</b> family descriptors for table
<code>HColumnDescriptor     getFamily(byte[] column)</code>	Returns descriptor for <b>column family</b>
<code>HColumnDescriptor     removeFamily(byte[] column)</code>	<b>Removes</b> specified family Descriptor

The table descriptor column family methods let us add, get, or remove column family properties by adding, getting or removing ColumnDescriptors.

We can:

- Add a family
- Check if the table contains a column family
- Get a list of column families for the table
- Get or remove a specific family

## Column Descriptor

```
// Instantiate config
Configuration config = HBaseConfiguration.create();
// Setup schema variables for table parts
String invTable = "/path/Inventory";
byte [] stockCF = Bytes.toBytes("stock");
byte [] quantityCol = Bytes.toBytes ("quantity");

HBaseAdmin admin = new HBaseAdmin(config);
HTableDescriptor desc = new HTableDescriptor(TableName.valueOf(invTable));
HColumnDescriptor c = new HColumnDescriptor(stockCF);
desc.addFamily(c);
admin.createTable(desc);
```

Next we will look at the HColumnDescriptor.

## Column Descriptor: Overview

`HColumnDescriptor` contains **info** about **column family** such as number of versions, compression settings, etc.

Add to Table descriptor before **creating or modifying table**

```
HColumnDescriptor c = newHColumnDescriptor(myCF);  
desc.addFamily(c);
```

Constructor	Description
<code>HColumnDescriptor (byte[] familyName)</code>	Construct a column descriptor specifying the <b>family name</b>
<code>HColumnDescriptor (HColumnDescriptor desc)</code>	Construct a column descriptor using an <b>existing descriptor</b> .

`HColumnDescriptor` wraps column family settings, and lets us control the details about a column family.

`HColumnDescriptor` contains information about a column family, such as the number of versions and compression settings.

We add a column descriptor to a table descriptor before creating or modifying a table, and we create a column descriptor with a family name or with an existing descriptor.

# Review



## Review



- Column families features apply to all columns
- Column families support arbitrary number of columns
- Columns may be defined on the fly
- Column family name must use printable characters
- Columns are addressed with family:qualifier
- A column family cannot be renamed in HBase
- With MapR-DB you can use MCS to rename a column family

Here is a quick refresher of column family features:

- Column families properties apply to all of the columns in the family
- Column family settings include min and max versions, time to live, compression, and in memory
- A client can create any number of columns dynamically by simply using new column qualifiers on the fly, when putting a value
- Columns are addressed with the column family name and the column qualifier divided by a colon, as in family:qualifier
- A column family name must be composed of printable characters
- A column family cannot be renamed in HBase, though with MapR-DB we can use the MapR Control System to rename a column family

## Column Descriptor: Getters and Setters Versions

Used to set **max versions** for column family

**Default** is 1 version (before HBase .98 it was 3)

When set to 1, will **only** store **latest** version

Values that exceed set maximum will be removed

Method Signature	Description
<code>int getMaxVersions()</code>	Returns maximum number of versions
<code>void setMaxVersions(int maxVersions)</code>	Sets maximum number of versions
<code>void setMinVersions(int versions)</code>	Sets minimum number of versions

We can set the minimum and maximum number of cell versions to keep for a column family. The default value is currently 1, before HBase version .98 the default was to store three versions.

Values that exceed the set maximum will be removed. For example, if we have the max versions set to 3, and already have three versions stored, then when a new version is added, the oldest one will be dropped.

If we only want the latest version, we set the max version to 1.

minVersions sets the minimum number of versions to keep, and is often used with timeToLive.

## Column Descriptor: Getters and Setters Compression

Use to set **compression type** for a column family:

- MapR supports 3 compression algorithms:
- LZ4 (default) - fast but less compression
- LZF
- ZLib – slow but higher compression

Method Signature	Description
<pre>void setCompressionType(     Compression.Algorithm     type)</pre>	Set compression type algorithm
<pre>Compression.Algorithm getCompressionType()</pre>	Returns compression type

HBase provides pluggable compression algorithms that let us choose the best compression for the data stored in a column family.

MapR supports the LZ4, LZF and ZLib compression algorithms.

LZ4 is the default algorithm. LZ4 is faster than the other options, but gives less compression. Zlib is slower but gives a higher level of compression. There is a tradeoff between compression ratio and speed.

To set the compression algorithm for a column family, call `setCompressionType`, passing in the enumeration type.

`Compression.Algorithm` is an enumeration.

## Column Descriptor: Getters and Setters Time To Live (TTL)

Use to set Time To Live for column family

TTL controls when content

- If value **exceeds its TTL** it's **dropped** based on timestamp
- Can be used with `setMinVersions` and `setMaxVersion`

Specified in seconds

By default set to `Integer.MAX_VALUE` ~2 billion seconds

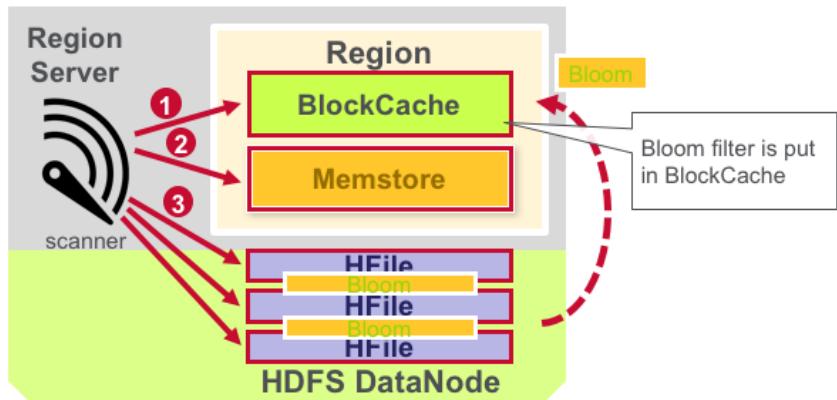
Method Signature	Description
<code>int getTimeToLive ()</code>	Returns Time-To-Live of cell contents, in seconds
<code>void setTimeToLive(int timeToLive)</code>	Set Time-to-live of cell contents, in seconds.

We use Time to Live, or TTL, to control when data should be expired. If a value exceeds its TTL, it is dropped, based on the timestamp.

We can set the minimum number of versions, to specify how many versions should be kept even if they are older than the TTL. If we set the minimum to 0, we will save no values.

The TTL is specified in seconds. By default it is set to `Integer.Max_Value`, which is more than 2 billion seconds.

## Overview: Bloom Filters



A read operation has to read cells corresponding to one row from multiple places. Row cells already persisted are in HFiles, recently updated cells are in the Memstore, and recently read cells are in the BlockCache.

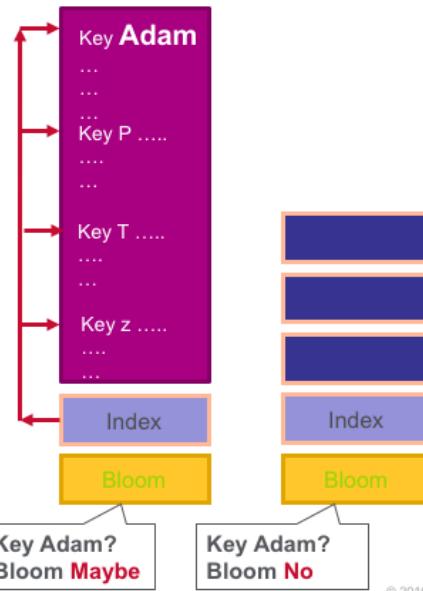
Bloom filters provide a lightweight, in-memory structure to reduce disk reads by determining which files do not contain row Cells.

Bloom filters are stored in the meta data of each HFile and are loaded into the BlockCache. A small amount of cache used for storing Bloom filters, which reduce the number of disk seeks required for read operations.

## Overview: Bloom Filters

Is row key Adam in this file?

- Returns Maybe or No
- **100% sure for No**
- Might report a *false positive*



<http://billmill.org/bloomfilter-tutorial/>

© 2016 MapR Technologies

L9-18

Bloom filters reduce disk access by determining whether an HFile might contain data for a specified row key.

A Bloom filter is a data structure designed to tell us, rapidly and memory-efficiently, whether an element is present in a set.

For example, Bloom filters allow us to ask a question like, is the row key "Adam" in this file? Bloom filters will occasionally return a false positive, but will never return a false negative.

In the example on the left, for the question, "Is row key Adam in this file?", the Bloom filter would return Maybe. In the example on the right, the bloom filter would return No. This eliminates reading files when the row cells will not be in this file.

## Column Descriptor: Getters and Setters Bloom Filters

**Bloom Filter index** determines:

- Whether **data exists in a file**
- **Before** incurring an **expensive disk read**
- **Saves I/O operations**

Apache HBase: Default value for BloomType is **NONE**.

**MapR-DB:** BloomType is always **ROW**.

- **Cannot be changed.**

Bloom filters reduce the number of HFiles that need to be read to find the cells for a given row.

Bloom filters are often referred to as a negative test. The algorithm tests for the presence of data in the index, and returns whether a file contains a particular row key or not.

For HBase the default BloomType value is none. However for MapR-DB, the BloomType is always row. The MapR-DB value cannot be changed.

HBase can have the following values for Bloom filters:

- **NONE**
- Bloomfilters disabled
- **ROW**
- Bloom enabled with table row as key
- **ROWCOL**
- Bloom enabled with table row and column (family+qualifier) as key

## Overview: HBaseAdmin

```
// Instantiate config
Configuration config = HBaseConfiguration.create();
// Setup schema variables for table parts
byte [] myTable = Bytes.toBytes("/path/to/myTable");
byte [] myCF = Bytes.toBytes("myColumnFamily");
byte [] myCol = Bytes.toBytes ("myColumnName");
byte [] rowA = Bytes.toBytes("rowA");
byte [] value1 = Bytes.toBytes("value1");

HTableDescriptor desc = new HTableDescriptor(myTable);
HColumnDescriptor c = new HColumnDescriptor(myCF);
desc.addFamily(c);
HBaseAdmin admin = new HBaseAdmin(config);
admin.createTable(desc);
```

Next we will talk about using the HBaseAdmin class.

## Overview: HBaseAdmin

- **Package**

org.apache.hadoop.hbase.client.HBaseAdmin

- Similar to **DDL** in SQL
- Use **HBaseAdmin** methods to
- Manage table **metadata**
- **Create** and **drop** tables
- Add and drop table **column families**

HBaseAdmin provides an API for administrative tasks, similar to the DDL (Data Definition Language) in relational databases.

It provides methods to create tables with specific column families, check for table existence, alter table and column family definitions, drop tables, and more.

## Knowledge Check





## Knowledge Check

Match the HBase table method with its function in defining and managing tables

Method	Function
HBaseAdmin	Name the table, add and remove column families
HTableDescriptor	Create or modify a table
HColumnDescriptor	Define column family properties such as TTL, compression and max



## Knowledge Check

Match the HBase table method with its function in defining and managing tables

Method	Function
HBaseAdmin	Create or modify a table
HTableDescriptor	Name the table, add and remove column families
HColumnDescriptor	Define column family properties such as TTL, compression and max

## Learning Goals



## Learning Goals



- 9.1 Define Table and Column Family Properties
- 9.2 Create, Alter, and Delete Tables**
- 9.3 Define HBase API Compatibility with MapR-DB

## Constructor : HBaseAdmin

```
// Instantiate config
Configuration config = HBaseConfiguration.create();

HBaseAdmin admin = new HBaseAdmin(config); constructor

admin.createTable(desc);

// Release resources
admin.close(); close
```

We can create an HBaseAdmin with a configuration object, the same configuration object we used with the HBaseTable interface.

We will need to call close, to release resources kept by the HBaseAdmin when we are done.

## Table Operations: HBaseAdmin

```
// Instantiate config
Configuration config = HBaseConfiguration.create();
// Setup schema variables for table parts
byte [] myTable = Bytes.toBytes("/path/to/myTable");
byte [] myCF = Bytes.toBytes("myColumnFamily");
byte [] myCol = Bytes.toBytes ("myColumnName");
byte [] rowA = Bytes.toBytes("rowA");
byte [] value1 = Bytes.toBytes("value1");

HTableDescriptor desc = new HTableDescriptor(myTable);
HColumnDescriptor c = new HColumnDescriptor(myCF);
desc.addFamily(c);
HBaseAdmin admin = new HBaseAdmin(config);    admin.createTable(desc);
```

HBaseAdmin  
createTable()

Next we will look at the HBaseAdmin table operations.

## Table Operations: HBaseAdmin

Method Signature	Description
<code>void createTable(     HTableDescriptor desc)</code>	Creates new table
<code>void createTable(     HTableDescriptor desc,     byte[][] splitKeys)</code>	Creates new table with initial set of <b>empty regions</b> defined by specified <b>split keys</b>
<code>void createTable(     HTableDescriptor desc,     byte[] startKey,     byte[] endKey,     int numRegions)</code>	Creates new table with specified number of regions start key = end key of first region end key = start key of last region

Table operations work with the tables, not the actual schemas inside them.

The first operation listed here, `createTable` with the `HTableDescriptor`, is the base method for creating a new table.

The next creates a new table, and adds an initial set of empty regions defined by the specified split keys. The total number of regions created will be the number of split keys plus one. The split keys define the start and end keys of the regions created.

The final operation creates a new table with a specific number of regions. The start key that is defined will become the end key of the first region of the table, and the end key defined will become the start key of the last region of the table. The first region has a null start key, and the last region has a null end key. BigInteger math will be used to divide the key range specified into enough segments to make the required number of total regions.

## Table Operations: HBaseAdmin

Method Signature	Description
<code>boolean tableExists(byte[] tableName)</code>	Returns true if table exists
<code>HTableDescriptor[] listTables()</code>	List all tables, by default in user home directory
<code>HTableDescriptor getTableDescriptor(byte[] tableName)</code>	Returns tableDescriptor

The `tableExists()` method will determine if a table already exists, or if a previous command to create succeeded.

The `listTables()` method returns a list of `HTableDescriptor` instances for every table that MapR knows about. By default, this will list instances from the user's home directory.

The `getTableDescriptor()` method returns the `HTableDescriptor` for a specific table.

## Table Operations: HBaseAdmin

Method Signature	Description
<code>void deleteTable(String tableName)</code>	Deletes a table
<code>void deleteTable(byte[] tableName)</code>	Deletes a table
<code>void modifyTable(byte[] tableName, HTableDescriptor htd)</code>	Modify an existing table, asynchronous operation.

The deleteTable() method will delete the table specified by its string or byte name.

To alter the structure of a table we can call modifyTable() passing a TableDescriptor. This is an asynchronous operation to modify table properties after creating a table.

## Schema Operations: HBaseAdmin

Method Signature	Description
<code>void modifyColumn(String tableName, HColumnDescriptor descriptor)</code>	Modify an existing <b>column family</b> on a table
<code>void modifyColumn(byte[] tableName, HColumnDescriptor descriptor)</code>	Modify an existing column family on a table

The methods shown here will delete a column or modify a column family. If there is data stored in the column, it will be deleted when the column is deleted.

## Administrative API: Putting It All Together

```
Configuration conf = HBaseConfiguration.create();
HBaseAdmin admin = new HBaseAdmin(conf);
...
HTableDescriptor tableDescriptor=new
HTableDescriptor(TableName.valueOf(invTable));
tableDescriptor.addFamily(new HColumnDescriptor(colFamilyOne));
tableDescriptor.addFamily(new HColumnDescriptor(colFamilyTwo));
admin.createTable(tableDescriptor);
...
HTable myTable = new HTable(conf, tableName);
```

In this example of using the admin API:

- First, instantiate an HBaseAdmin object with a configuration, to create a table
- A table descriptor is instantiated with two column descriptors for two column families
- Once the descriptors are instantiated, we can call create on our HBaseAdmin object
- Now that the table is created we can start using it for get, put, delete or scan operations

## Knowledge Check



## Knowledge Check



When we use the Admin API:

We first instantiate an `HBaseAdmin` object with a \*\*\*1\*\*\*, to create a table.

Before creating the table, a \*\*\*2\*\*\* is instantiated with two column descriptors for two column families. Once the descriptors are instantiated, we can \*\*\*3\*\*\* on our `HBaseAdmin` object.

1. call create
2. configuration
3. table descriptor

## Knowledge Check



When we use the Admin API:

We first instantiate an `HBaseAdmin` object with a **configuration**, to create a table. Before creating the table, a **table descriptor** is instantiated with two column descriptors for two column families. Once the descriptors are instantiated, we can **call create** on our `HBaseAdmin` object.

1. \*\*\*1\*\*\* :: configuration
2. \*\*\*2\*\*\* :: table descriptor
3. \*\*\*3\*\*\* :: call create

## Learning Goals



## Learning Goals



- 9.1 Define Table and Column Family Properties
- 9.2 Create, Alter, and Delete Tables
- 9.3 Define HBase API Compatibility with MapR-DB**

## MapR Tables Support for HBase Interfaces General Guidelines

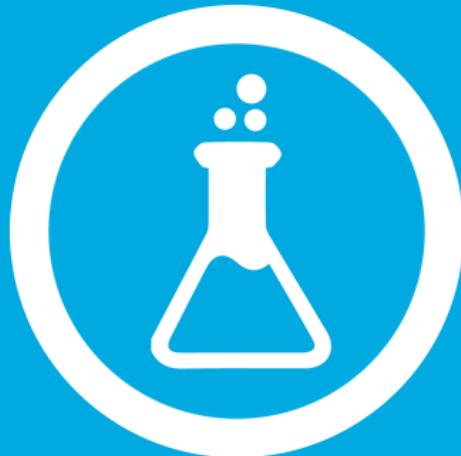
- **Coprocessors** are **not** supported
- **Row locks** are **not** supported
- Some table and HBaseAdmin operations **not** supported
  - **Master** related methods
  - Some **region** related methods
  - **Compaction** methods
  - Some **Connection** related methods
  - **After table creation, splits** commands are **ignored**.  
Use **pre-split** if you know size of data ingesting

The online MapR documentation will provide you with the details of what HBase API methods and shell commands are supported. For example,

Coprocessors are not supported  
Row locks are also not supported  
Some table and HBaseAdmin operations are not supported, such as  
    Master related methods are not supported  
    Some region related methods are not supported  
    Compaction methods are not supported  
    Some Connection related methods are not supported

## Lab 9.3:

### Working with LabAdminAPI in lab-exercises Project





## Next Steps

### DEV 335 – Developing HBase Applications: Advanced

Lesson 10: Advanced HBase Java API

Congratulations! You have finished DEV 330, Lesson 3: Developing HBase Applications Basics – Java Client API for Administrative Features.

Continue on to DEV 335 to learn more about the HBase Java API.



## DEV 335 – Developing Apache HBase Applications: Advanced

Lesson 10: Advanced HBase Java API

Winter 2017, v5.1

Welcome to DEV 335, Lesson 10: Advanced HBase Java API.

## Learning Goals



## Learning Goals



- 10.1 Define Different Filter Types and Apply Them to Applications
- 10.2 Use Counters on Incremental Occurrences

In this lesson, we will look at more advanced features of the HBase Java API such as using filters with scan operations to improve results that come back from a scan operation and using counters which are atomic update operations.

## Learning Goals



### 10.1 Define Different Filter Types and Apply Them to Applications

10.2 Use Counters on Incremental Occurrences

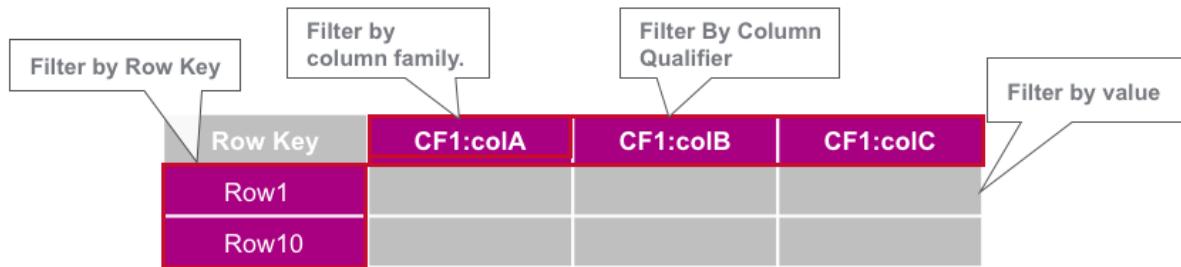
When you have finished with this lesson, you will be able to:

- Define the different types of filters available when using the Java API, and apply these filters to reduce the returned results based on column family, qualifier, value and row key.
- Use counters to store and retrieve incremental occurrences to the HBase tables.

## Filters Narrow Down Results

Filters help narrow down the results brought back from a Scan or a Get

You can filter by **row key**, column family, **column qualifier**, **value**



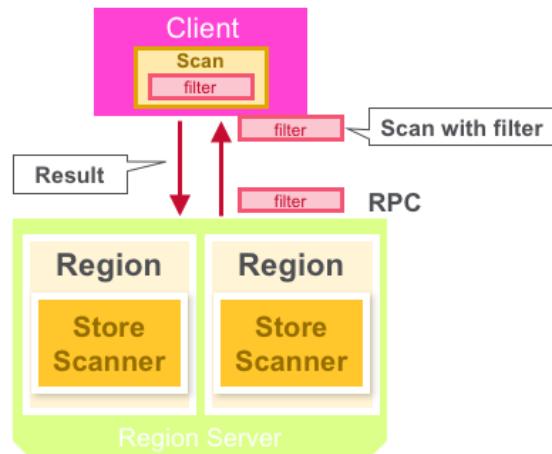
Filters let us narrow down the result set returned from a Scan or a Get, and they provide more fine-grained features than the add or set methods on a get or scan object. We can filter by row key, column family, column qualifier, and value. We can also compare using substrings and regular expressions.

## Filters Applied on the Server Side

Use:

- Define a new instance of the filter you want to apply
- Call `setFilter(filter)` method on Get or Scan instance

Filters are applied on the server side



We define a new instance of the filter that we want to apply and hand it to the Get or Scan instances, using `setFilter()`, and passing in the filter as a parameter. This sets the filter on the client side. Filters are then serialized so they can be sent over the network, deserialized, and then applied on the server side. On the server side, the filter is used to determine whether a record should be returned back to the client side.

## Types of Filters

### Comparison Filters

- Row Filter, Family Filter, Qualifier Filter, Value Filter

### Dedicated Filters

- KeyOnlyFilter, PrefixFilter, SingleColumnValueFilter ...

### Decorating Filters

- SkipFilter, WhileMatchFilter

### FilterList

- Used to combine filters

The different types of filters available are: dedicated filters, comparison filters, decorating filters, and finally the filter list.

With comparison filters:

The Row filter: gives us the ability to filter data based on the row key.

The Family filter: filters based on the column family.

The Qualifier filter: will filter based on the column qualifier.

And the Value filter: will filter based on column value.

## Reminder: Tall Table for Stock Trades

rowkey format:

**SYMBOL** + **Reverse timestamp**

Ex: AMZN\_98618600888

Column family data

Row Key	data: Price	data: Volume
AMZN_98618600888	12.34	1000
AMZN_98618600777	12.00	50

This slide shows the table for the stock trades tall schema where every row represents one trade. There is one column family, with two columns to store Price and Volume values. The composite row key is formed by combining the stock symbol and a reversed timestamp, which is calculated using `Long.MAX_VALUE` and subtracting the current timestamp.

## Filter Example: ValueFilter on Stock Trades

Want to Get the stocks with a Volume cell value  $\geq 8000$

Row Key	CF1: price	CF1: vol
AMZN_9223370655437496807	600.27	6007
CSCO_9223370655451096807	500.71	8326
GOOG_9223370655439000807	767.24	8327
GOOG_9223370655441159807	867.24	5327

Scan results for table without any filters:

```
RowKey AMZN_9223370655437496807, CF1:price 600.71, CF1:vol 6007
RowKey CSCO_9223370655451096807, CF1:price 500.71, CF1:vol 8326
RowKey GOOG_9223370655439000807, CF1:price 767.24, CF1:vol 8327
RowKey GOOG_9223370655441159807, CF1:price 867.24, CF1:vol 5327
```

This example shows a small sample output for putting and then scanning the table without any filtering.

## Filters – Comparison Filters Overview

- Comparison filters are used to filter by comparison
  - An Operator, **How** to compare (equal, greater, not equal, etc)
  - A Comparator, **What** to compare to

Filter cell value  $\geq 8000$

operator

```
new ValueFilter(CompareOp GREATER_OR_EQUAL, new  
BinaryComparator(Bytes.toBytes(80001)))
```

comparator

Comparison filters are used to filter by comparison. We can filter by comparing a value with the row key, column family, qualifier, or cell value.

We create a comparison filter with an operator such as equal, greater, not equal, and a comparator, which specifies what to value compare to, such as a String, SubString, Binary Value, or Regular Expression.

In this example a `valueFilter` is used to find values greater than 8000. The EQUAL operator is used together with a binary comparator.

## Comparison Filters - Example

```
// Retrieve cells value >= 8000 long
scan = new Scan();
scan.addColumn(Bytes.toBytes("CF1"), Bytes.toBytes("vol"))
scan.setFilter(new ValueFilter(
    CompareOp.GREATER_OR_EQUAL,
    new BinaryComparator(Bytes.toBytes(8000l))));
ResultScanner scanner = table.getScanner(scan);
for (Result result : scanner) {
    for (KeyValue kv : result.raw()) {
        System.out.println("KeyValue: " + kv +
            "Value: " + Bytes.toString(kv.getValue()));
    }
}
```

Here is the example code for a `valueFilter` used to find Values  $\geq 8000$  long on the volume column with a scan operation.

The code is the same as for a regular scan operation, we call `scan.setFilter` passing a filter as the argument.

We can then narrow down the data filtered by using the `addColumn` or `addColumnFamilies` methods with the `scan` object, and also by specifying a start and stop key when appropriate.

## Filter Example: Tall Table for Stock Trades

Want to Get the stocks with a cell value > 8000

Row Key	CF1: price	CF1: vol
AMZN_9223370655437496807	600.27	6007
CSCO_9223370655451096807	500.71	8326
GOOG_9223370655439000807	767.24	8327
GOOG_9223370655441159807	867.24	5327

(KeyValue= <row-key>/<family>:<qualifier>/<version>/<type>/<value-length>)

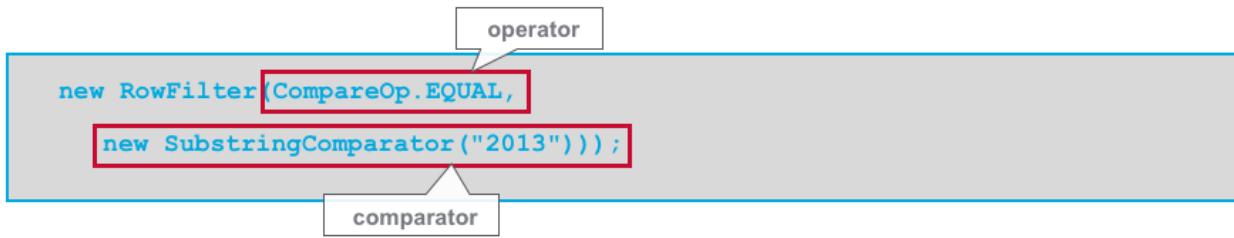
### Scan results for table with value filter:

```
KeyValue CSCO_9223370655451096807/CF1:vol/1391531237737/Put/vlen=8/ts=0, vol : 8326  
KeyValue GOOG_9223370655439000807/CF1:vol/1391531237737/Put/vlen=8/ts=0, vol : 8327
```

Here we see a small sample output for the previous code for scanning the trades table with the value filter  $\geq 8000$  on the volume column.

## Filters – Comparison Filters

- You can filter by comparisons on:
  - Row Key, Column Family, Column Qualifier, ValueWith a:
  - RowFilter, FamilyFilter, QualifierFilter, ValueFilter
- Comparison constructor takes:
  - An Operator, **How** to compare (equal, greater, not equal, etc)
  - A Comparator, **What** to compare to



A comparison filter is created with an operator and a comparator. The operator specifies how to compare, such as equal, greater or not equal and the comparator specifies what value to compare.

In this example a valueFilter is used to find a specific string. The EQUAL operator is used together with a sub-string comparator.

## Filters – Comparison Filters Operators

Operator	Description
EQUAL	Do an <b>exact match</b> on the value and the provided one
GREATER_OR_EQUAL	Match values that are equal to or greater than the provided one
GREATER	Only <b>include values greater</b> than the provided one
LESS	Match values less than the provided one
LESS_OR_EQUAL	Match values less than or equal to the provided one
NO_OP	<b>Exclude</b> everything
NOT_EQUAL	Include everything that <b>does not match</b> the provided value

<http://hbase.apache.org/apidocs/org/apache/hadoop/hbase/filter/CompareFilter.CompareOp.html>

The comparison operator specifies how to compare. They allow us to select the data that we want as either a range, subset, or exact match.

EQUAL will do an exact match on the value and the provided one.

GREATER\_OR\_EQUAL will match values that are equal to or greater than the one provided.

GREATER will Only include values greater than the one provided.

LESS will Match values less than the one provided.

LESS\_OR\_EQUAL will Match values less than or equal to the one provided.

NO\_OP will Exclude everything.

NOT\_EQUAL will Include everything that does not match the provided value.

For more information on comparison filter operators, refer to the java doc.

## Filters – Comparison Filters Comparators

Comparator	Description
BinaryComparator	Compares using the <code>Bytes.compareTo()</code> method
BinaryPrefixComparator	Does a prefix-based bytewise comparison using <code>Bytes.compareTo()</code> , starting from the <code>left</code>
NullComparator	Checks whether the given value is <code>null</code>
BitComparator	Does a <code>bitwise</code> comparison Only works with <code>EQUAL</code> and <code>NOT_EQUAL</code>
RegexStringComparator	Compares the passed value with <code>the regular expression</code> provided at the time of instantiating the comparator Only works with <code>EQUAL</code> and <code>NOT_EQUAL</code>
SubstringComparator	Does a <code>contains()</code> check in the passed value for the <code>substring</code> provided as a part of the comparator Only works with <code>EQUAL</code> and <code>NOT_EQUAL</code>

MAPR Academy <http://hbase.apache.org/apidocs/org/apache/hadoop/hbase/filter/CompareFilter.html>

© 2016 MapR Technologies L10-15

The comparator specifies what value to compare. We can create a comparator with the value that we want to compare against. Some of these constructors take a byte array to do a binary comparison and other comparators take a String parameter.

The string-based comparators, `RegexStringComparator` and `Substring Comparator` are more expensive in comparison to the purely byte-based version, as they need to convert a given value into a String first. The subsequent string or regular expression operation also adds to the overall cost.

The last three: `BitComparator`, `RegexStringComparator`, and `SubstringComparator`, only work with the `EQUAL` and `NOT_EQUAL` operators, as the `compareTo()` of these comparators returns zero for a match, or one when there is no match. Using them in a LESS or GREATER comparison will yield erroneous results.

## Reminder: Wide Table for Stock Trades

rowkey format: **SYMBOL** + **date**

Ex: AMZN\_20131020

Column Families Price, Volume: Column for every hour

Column Families Statistics: Columns Day Hi, Day Low

Cell Version: Stores the timestamp of the trade, keep all versions

Row Key	CF Price Column hours			CF Volume			CF Stats	
	price: 00	...	Price:23	Vol:00	...	Vol:23	Day Hi	Day Lo
AMZN_20131020	12.34		12.00	100				
GOOG_20130817	12.00		13.00	50				

In this flat schema, all trades for each day are stored in a single row. Trades are grouped by the hour of the day, using a column for every hour. Price and Volume values are stored in separate column families.

Every version of a cell represents one trade and the cell version, which is a long, stores the timestamp of the trade in milliseconds. Because every version of a cell is significant, the Price and Volume column families are set to keep all versions.

The row key is a composite of the company symbol and the day-of-year, formatted YYYYMMDD.

## Most Commonly Used Filters – RowFilter

- Filter data based on rowkeys
- Often used with substring matches, or regular-expression matches
- Example filter for 2013

Stock Flat Wide Table

Row Key	price: 00	...	Price:23	Vol:00	...	Vol:23
GOOG_20130810	12.00		13.00	50		

Filter data based on rowkeys

Often used with substring matches, or regular-expression matches

// Return row with a key equal to "2013"

```
Filter filter = new RowFilter(  
    CompareFilter.CompareOp.EQUAL,  
    new SubstringComparator ("2013"));
```

### Scan Result:

Row Key GOOG\_20130810 Hour: 23, Price: 13.00

© 2016 MapR Technologies L10-17

RowFilter gives us the ability to filter based on row keys. We can compare for exact matches, usually using a binary comparator, and also for substring matches or regular expression matches.

In the example here, rows with a key equal to or less than GOOG\_201308 will be returned.

## Most Commonly Used Filters – QualifierFilter

- Filters data based on column name
  - Often combined with other filters in a FilterList
- Example filter column names <= “10”

Row Key	price: 00	...	Price:23	Vol:00	...	Vol:23
GOOG_20130817	12.00		13.00	50		

Stock Flat Wide Table

```
//Return values where column name is less or equal to 10  
Filter filter = new QualifierFilter(CompareFilter.CompareOp.LESS_OR_EQUAL,  
        new BinaryComparator(Bytes.toBytes("10")));
```

The qualifier filter is similar to the row filter but instead of operating on row keys it works on column qualifiers.

In this example, we want to filter in the flat stock table for columns less than 10, which would be price and volume columns before 10 o'clock. If we know the exact column name that we are looking for, then we can add a column to a scan operation, using [scan.addColumn\(byte\[\], byte\[\]\)](#) directly rather than a filter. This filter can be wrapped with the [WhileMatchFilter](#) and [SkipFilter](#) to add more control.

Multiple filters can also be combined using [FilterList](#).

## Filter Example: QualifierFilter

Want to Get the stocks for qualifier **<= 10** o'clock

Stock Flat Wide Table

Row Key	price: 00	...	Price:23	Vol:00	...	Vol:23
GOOG_20130817	12.00		13.00	50		

Scan results for table without any filters:

AMZN: 8326 shares at \$600.71 at 2013.10.10 11:01:19

CSCO: 8326 shares at \$500.71 at 2013.10.09 07:14:39

GOOG: 7327 shares at \$867.24 at 2013.10.10 10:36:16

GOOG: 8327 shares at \$767.24 at 2013.10.10 10:36:15

Scan result with filter:

RowKey CSCO\_20131009 **Hour: 07**, Price: 500.71, Volume: 8326, TimeStamp : 1381317279000

RowKey GOOG\_20131010 **Hour: 10**, Price: 867.24, Volume: 7327, TimeStamp : 1381415776000

RowKey GOOG\_20131010 **Hour: 10**, Price: 767.24, Volume: 8327, TimeStamp : 1381415776000

This example shows the results of filtering in the flat stock table for columns less than 10, which would be prices and volume columns before 10 o'clock. The results are for hours less than 10.

## ValueFilter with RegexStringComparator

- `ValueFilter`:
  - Filters all columns that have a specific **cell value**
- `RegexStringComparator`:
  - Compares the passed value with **the regular expression**
  - Only works with **EQUAL** and **NOT\_EQUAL**

```
// filter all cell values between 000 and 999
Filter filter = new ValueFilter(CompareFilter.CompareOp.EQUAL,
    new RegexStringComparator ("^[0-9]{3}$"));
```

This is an example of a `ValueFilter` combined with a `RegexStringComparator`, which lets us compare using a regular expression. In this example, we are filtering all cell values between 000 and 999 using the regular expression shown.

## Knowledge Check





## Knowledge Check

### Match the comparison filter with its function:

- Row filter :: filters data based on the row key
- Family filter :: filters data based on the column family
- Qualifier filter :: filters data based on the column qualifier
- Value filter :: filters data based on column value

## Types of Filters

### Comparison Filters

- RowFilter, FamilyFilter, QualifierFilter ...

### Dedicated Filters

- KeyOnlyFilter, PrefixFilter, SingleColumnValueFilter ...

### Decorating Filters

- SkipFilter, WhileMatchFilter

Next, we will discuss Dedicated Filters.

## Filters – Dedicated Filters

Implement a specific filter for one of the elements of a row

### Examples

- SingleColumnValueFilter
- PrefixFilter
- PageFilter
- KeyOnlyFilter
- FirstKeyOnlyFilter
- InclusiveStopFilter
- TimestampsFilter
- ColumnCountGetFilter
- ColumnPrefixFilter
- RandomRowFilter

MAPR Academy



Dedicated filters implement specific use cases and generally apply to one of the elements in a row like a column or a timestamp.

The `SingleColumnValueFilter` is used to filter a specific cells value.

With the `PrefixFilter`, when given a prefix, which specified when the filter instance is instantiated, all rows that match this prefix are returned to the client.

`PageFilter` will paginate through rows.

`KeyOnlyFilter` will get just the keys of each `KeyValue`, while omitting the actual data.

And the `TimeStampFilter` gives fine-grained control over what versions are included in the scan result.

## Dedicated Filters - SingleColumnValueFilter

```
// filter trades with volume >= 8000
SingleColumnValueFilter filter = new SingleColumnValueFilter(
    Bytes.toBytes("CF1"),
    Bytes.toBytes("vol"),           Column
    CompareFilter.CompareOp.GREATER_OR_EQUAL,   operator
    Bytes.toBytes(8000L));          comparator
scan = new Scan();
scan.setFilter(filter);
```

key	CF1: price	CF1: vol
AMZN_9223370655437496807	600.27	1333
CSCO_9223370655451096807	500.71	6328
GOOG_9223370655439000807	767.24	9325

To test the value of a single column when scanning we can use the [SingleColumnValueFilter](#).

We must first specify the column that we want to compare to and then an operator and a comparator value to check against. In this example we want to get the stocks with the volume column value  $\geq 8000$  long

## SingleColumnValue on Stock Trades

Want to Get the stocks with a Single Column cell value  $\geq 8000$

Row Key	CF1: price	CF1: vol
AMZN_9223370655437496807	600.27	1333
CSCO_9223370655451096807	500.71	6328
GOOG_9223370655439000807	767.24	9325

### Scan No Filter

AMZN: 1333 shares at \$600.27 at 2013.10.10 05:12:43

CSCO: 6328 shares at \$500.71 at 2013.10.10 07:00:01

GOOG: 9325 shares at \$767.24 at 2013.10.10 11:26:22

### Scan With Filter

KeyValue GOOG\_9223370655435993807/CF1:vol/1391392036125/Put/vlen=8/ts=0, **vol : 9325**

This shows the output for filtering the stocks with  $vol \geq 8000$  in the Stock tall table using the SingleColumnValue filter.

## Most Commonly Used Dedicated Filters PrefixFilter

- Filters data based on a **prefix** value of the **rowkey**
- **Combine with scan start row** for better performance!
  - Scan is ended when the **rowkey > filter prefix**

```
// Return only rows with row keys that start with
String prefix = "GOOG 9223370655437";
Filter filter = new PrefixFilter(Bytes.toBytes(prefix));
```

KeyValue GOOG\_9223370655437496807/CF1:price/1391392036121/Put/vlen=8/ts=0  
KeyValue GOOG\_9223370655437496807/CF1:vol/1391392036121/Put/vlen=8/ts=0,  
KeyValue GOOG\_9223370655437997807/CF1:price/1391392036119/Put/vlen=8/ts=0  
KeyValue GOOG\_9223370655437997807/CF1:vol/1391392036119/Put/vlen=8/ts=0,

The `PrefixFilter` filters data based on the prefix value of the row key, all rows that match this prefix are returned to the client and the scan is ended when the filter encounters a row key that is larger than the prefix.

On a scan operation, combining this filter with a start row improves the performance of the scan. This example shows a prefix filter on row keys that start with the string highlighted here.

## InclusiveStopFilter

```
//userpost key = userId | postid
HTableInterface table = pool.getTable(USER_POST_TABLE_NAME);

byte[] startRow = Bytes.toBytes(userId);
Filter filter = new InclusiveStopFilter(Bytes.toBytes("userId"));

Scan scan = new Scan(startRow);
scan.setFilter(filter);
ResultScanner results = table.getScanner(scan);
```

Scan for Post keys WHERE user = "user"

The `InclusiveStopFilter` is a filter that stops scanning after the given row key bytes. When we specify the stop row in the scan constructor, it stops just before the stop row. Also, with scans, the start and stop row can specify just the leftmost part of a row key, in order to return all rows starting with this part of the key.

We can use this filter to include the stop row. In this example the start row is set to a userId, and it sets the filter to stop after the userId. This will only scan row keys starting with this userId and will be an inclusive stop. Setting the stop row key in the scan constructor would not work because it would cause it to stop before the userId.

## Most Commonly Used Filters – TimestampsFilter

- `TimestampsFilter` returns only cells whose timestamp (version) is in the specified list of timestamps
- Provides fine grain control over versions returned
- Example

```
List<Long> timestamps = new ArrayList<Long>();
timestamps.add(100L);
timestamps.add(200L);
timestamps.add(300L);
Filter filter = new TimestampsFilter(timestamps);
```

The `TimestampsFilter` returns only cells whose version timestamp is in the specified list of version timestamps. This filter allows fine-grained control over the versions that are returned to the client.

## Knowledge Check





## Knowledge Check

### Match the comparison filter with its function:

- SingleColumnValueFilter :: filter a specific cells value.
- PrefixFilter :: all rows that match a prefix are returned to the client.
- PageFilter :: paginate through rows.
- KeyOnlyFilter :: will get just the keys of each key-value, while omitting the actual data.
- TimeStampFilter :: gives fine-grained control over what versions are included in the scan result.

## Types of Filters

### Comparison Filters

- RowFilter, FamilyFilter, QualifierFilter ...

### Dedicated Filters

- KeyOnlyFilter, PrefixFilter, SingleColumnValueFilter ...

### Decorating Filters

- SkipFilter, WhileMatchFilter

Decorating filters extend the behavior of another filter to provide more control over what data is returned. The decorator pattern is a design pattern that allows behavior to be added to an object.

Decorating filters are applied to another filter, like the decorator pattern they wrap an existing filter, allowing behavior to be added to the filter.

## Decorating Filters - SkipFilter

- Decorating filters extend the behavior of another filter to provide more control over what data is returned.
- SkipFilter
  - **Exclude** an entire **row** based on wrapped filter
  - Often combined with a ValueFilter
- Example: **Exclude** any rows when **Value = 0**

```
// Filter out an entire row if any of its values = zero
scan.setFilter(new SkipFilter(
    new ValueFilter(CompareOp.EQUAL,
        new BinaryComparator(Bytes.toBytes(0))));
```

A SkipFilter wraps a given filter and extends it to exclude an entire row. As soon as the wrapped filter indicates a value is to be omitted then the entire row is omitted. Without this filter, the other non-zero valued columns in the row would still be emitted.

This is often used with a ValueFilter as shown in the example code here, to skip a row if any of its values are zero.

## Decorating Filters- WhileMatchFilter

- WhileMatchFilter
  - **Aborts** the entire scan once a piece of information is filtered
    - Based on wrapped filter
  - Example stop filtering when the row key! = GOOG\_20130817

```
// filter row != GOOG_20130817
Filter filter1 = new RowFilter(CompareFilter.CompareOp.NOT_EQUAL, new
    BinaryComparator(Bytes.toBytes("GOOG_20130817")));

// Aborts if row != GOOG_20130817
scan.setFilter(new WhileMatchFilter(filter1));
```

wrap

The WhileMatchFilter aborts the entire scan once a piece of information is filtered.

This works by checking the wrapped filter and seeing if it skips a row by its key, or a column of a row because of a key-value check.

## Filters – FilterList

- Lets you combine filters into an ordered list
- The list is evaluated with an operator:
  - `MUST_PASS_ALL` (default), `MUST_PASS_ONE`
- Constructor : `FilterList(Operator operator)`
- Method: `addFilter(Filter filter)`

```
Filter filter1 = new ValueFilter(CompareOp.EQUAL,
    new BinaryComparator(Bytes.toBytes(0)));
FilterList filters = new FilterList(Operator.MUST_PASS_ALL);
filters.addFilter(filter1);
scan.setFilter(filters);
```

The `FilterList` creates filtering logic by combining several filters in an ordered list. A `FilterList` is created with an operator, which will be used with the list of filters to combine their results.

The default operator is `MUST_PASS_ALL`, which means results in the final list are included if they pass all filters. `JUST_PASS_ONE` means results are included if they pass only one filter.

You can add a filter after you have created an instance of `FilterList` by calling `addFilter`.

## Filter List on Stock Trades

Want to Get Amazon stocks with a Volume cell value  $\geq 1000$

Row Key	CF1: price	CF1: vol
AMZN_9223370655437496807	304.82	1999

### Scan results for table without any filters:

AMZN: 1999 shares at \$304.82 at 2013.10.10 05:46:05  
AMZN: 1666 shares at \$303.91 at 2013.10.10 05:29:24  
AMZN: 1333 shares at \$304.66 at 2013.10.10 05:12:43  
CSCO: 6328 shares at \$22.99 at 2013.10.10 07:00:01  
CSCO: 5995 shares at \$22.98 at 2013.10.10 06:57:40  
CSCO: 5662 shares at \$22.96 at 2013.10.10 06:52:39  
CSCO: 4996 shares at \$22.92 at 2013.10.10 06:42:37  
GOOG: 9325 shares at \$864.18 at 2013.10.10 11:26:22  
GOOG: 8992 shares at \$864.69 at 2013.10.10 11:18:01  
GOOG: 8659 shares at \$865.20 at 2013.10.10 11:09:40

Here we see a small sample output for putting and then scanning this table without any filtering.

## FilterList - Example

```
FilterList filters = new FilterList(Operator.MUST_PASS_ALL);
Filter filter1 = new RowFilter(CompareFilter.CompareOp.EQUAL, new
    SubstringComparator ("AMZN"));
Filter filter2 = new QualifierFilter(
    CompareFilter.CompareOp.EQUAL,
    new BinaryComparator(Bytes.toBytes("vol")));
Filter filter3 = new ValueFilter(
    CompareOp.GREATER_OR_EQUAL,
    new BinaryComparator(Bytes.toBytes(1000l)));
filters.addFilter(filter1);
filters.addFilter(filter2);
filters.addFilter(filter3);
scan.setFilter(filters);
```

Here is an example using a `FilterList` to retrieve stocks for Amazon with a volume  $\geq 1000$ .

A `FilterList` is used to combine three filters that we have already looked at before. A `RowFilter` filters for the substring "AMZN" to get only amazon stocks. A `QualifierFilter` filters for the volume column name. A `ValueFilter` filters for a value greater than or equal to 1000 long.

## Filter List on Stock Trades

Want to Get Amazon stocks with a Volume value  $\geq 1000$

### Scan results for table without any filters:

```
AMZN: 1999 shares at $304.82 at 2013.10.10 05:46:05
AMZN: 1666 shares at $303.91 at 2013.10.10 05:29:24
AMZN: 1333 shares at $304.66 at 2013.10.10 05:12:43
CSCO: 6328 shares at $22.99 at 2013.10.10 07:00:01
CSCO: 5995 shares at $22.98 at 2013.10.10 06:57:40
CSCO: 5662 shares at $22.96 at 2013.10.10 06:52:39
CSCO: 4996 shares at $22.92 at 2013.10.10 06:42:37
GOOG: 9325 shares at $864.18 at 2013.10.10 11:26:22
GOOG: 8992 shares at $864.69 at 2013.10.10 11:18:01
GOOG: 8659 shares at $865.20 at 2013.10.10 11:09:40
```

### Scan results for table with list filters:

```
KeyValue AMZN_9223370655456410807/CF1:vol/1391392036093/Put/vlen=8/ts=0, vol : 1999
```

```
KeyValue AMZN_9223370655457411807/CF1:vol/1391392036090/Put/vlen=8/ts=0, vol : 1666
```

```
KeyValue AMZN_9223370655458412807/CF1:vol/1391392036087/Put/vlen=8/ts=0, vol : 1333
```

This shows output for the previous code example.

## Filter Performance Tips

Narrow down row key range, otherwise full table scan

- `Scan(byte[] startRow, byte[] stopRow)`
- Remember only row keys are indexed!

Narrow down the columns, values filtered

- `addColumn(byte[] family, byte[] qualifier)`
- `addFamily(byte[] family)`
- `setTimeRange(long minStamp, long maxStamp)`

Here are some tips to improve filter performance. In general, we will want to narrow down the data that we are filtering on. We can do this by limiting the rows scanned and by limiting the columns or cells scanned.

## Filter Performance Tips

Filter interface methods are called in sequence

- The method `filterRowKey()` is called first
- For each `KeyValue` `filterKeyValue (KeyValue v)`
- Therefore **order a filter list with row filters** first

If you want a **row count, or only row keys**:

- Use a `FilterList` with a `MUST_PASS_ALL` operator
- And a `KeyOnlyFilter` with a `FirstKeyOnlyFilter`
- Region server will only get the row key of the first `KeyValue` per row

The filters are called in sequence, if we are using a filter list we should put the row key filters first in the list.

When performing a table scan where only the row keys are needed, no families, qualifiers, values, nor timestamps, we will add a `FilterList` with a `MUST_PASS_ALL` operator to the scanner using `setFilter()`.

The filter list should include both a `FirstKeyOnlyFilter` and a `KeyOnlyFilter` instance. Using this filter combination will cause the region server to only load the row key of the first key-value found and return it to the client, resulting in minimized network traffic.

## Knowledge Check



## Knowledge Check



Indicate whether the following statement is true or false:

Decorating filters extend the behavior of another filter to provide more control over what data is returned.

Answer: True

# Lab 10.1: Java Applications Using Filters



Refer to your lab guide to complete Lab 10.1.

## Learning Goals



## Learning Goals



10.1 Define Different Filter Types and Apply Them to Applications

**10.2 Use Counters on Incremental Occurrences**

A counter is a special kind of column used to store a number that incrementally counts the occurrences of a particular event or process.

## Counters – Overview

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

Row Key	stats: clicks
AMcom.example/home	1000

Common use cases

- Clicks, page hits, views

A counter is a long column value

Two types of counters:

- Single column counter with **HTable** method
- Multiple columns counter with an Increment Object

A counter is a long value in a column. When a counter is created the column is added and set to whatever value we provide.

Counters are used to track activity in general like clicks, page hits, or ad display counts in web applications. Counters provide a fast atomic increment operation, instead of running a get and put which would be too expensive and time consuming.

There are two types of counters.

There is a single column counter with the **HTable** method `incrementColumnValue` and there is a multiple column counter with an Increment Object that is similar to the Put, Get objects.

## Single Column Counter

`HTable incrementColumnValue method`

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

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

	cf: qualifier
row	amount

The `HTable IncrementColumnValue` method increments a single column value. It provides an atomic read-and-modify operation and it is easy to use.

## HTable incrementColumnValue

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

Row Key	stats: clicks
rowA	1000

$$1000 + 42 = 1042$$

When using the HTable IncrementColumnValue method, we specify the exact column and the amount to add and the result is returned. In this example the counter is the column called clicks, the amount 42 long will be added to the clicks column. 1042 long will be returned.

## Counters – Possible Values

Effect based on provided values:

Value	Description
Greater than zero	Increase the counter by the given value
Zero	Retrieve the current value of the counter
Less than zero	Decreases the counter by the given value
None	Increase the counter by one

- You must use a long
- Don't need to initialize (zero by default)

The increment method effect is based on the provided value. The increment method will increment the counter when using a positive value, retrieve the current value of the counter when using zero, and decrease the counter when using a negative value.

We do not need to initialize counters. They are set to the zero, or the specified amount, when first using a new counter. The first increment call to a new counter will return one or whatever value is specified.

When using counters, we MUST use a long on input.

## Counters

Two types of counters:

- Single column counter with HTable method.
- **Multiple columns counter with an Increment Object.**

Next, we will look at the multiple columns counter.

## Multiple Column Counter - Increment Object

```
Increment increment1 = new Increment(rowKey);
increment1.addColumn(CF, qualifier1, amount1);
increment1.addColumn(CF, qualifier2, amount2);
Result result1 = table.increment(increment1);
```

The diagram illustrates the sequence of operations in the code. It starts with a call to `new Increment(rowKey)`, which is labeled "Create Increment object with Row Key4". This leads to the `addColumn` calls, which are grouped under the label "Add columns to Increment". Finally, the `table.increment(increment1)` call is shown, with its label "Call table increment" pointing to it.

With the Increment Object, an increment may be applied to multiple columns. The Increment Object is similar to the Put and Get objects.

First, we instantiate an Increment Object with the row key to which the increment will be applied. Next, we specify each column to increment with an `addColumn()` call. And finally call HTable increment, passing the Increment Object.

## Inventory Table Increment

cf: stock		
	quantity	Mike
pens	13	12

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

Here is an example of an increment operation on multiple columns for the shopping example inventory table. In this example, we want to subtract from the pens quantity column and add to the pens Mike column in one operation.

First, we instantiate the increment object with the pens row key. Then, we call `addColumn` for the quantity and Mike columns, subtracting from the quantity column the amount that we want to add to the Mike column.

And finally, we call `table.increment()` after which the pens quantity column will be 12, and the Mike column will be one.

## Inventory

### Inventory Table

Checkout for Mike with 1 pen, 3 notepads and 2 erasers

	cf: stock	
	quantity	
	13	12
pens	13	12
notepads	23	20
erasers	10	8

Next, we will look at how to do this for all three rows as shown in the diagram here.

## Counters – Increment Operation Example

```
Increment increment1 = new Increment(pens);
increment1.addColumn(stock, quantity, -1); //quantity counter=12
increment1.addColumn(stock, Mike, 1) // Mike counter =1
Result result1 = table.increment(increment1);

Increment increment2 = new Increment(notepads);
increment2.addColumn(stock, quantity, -3); //quantity counter=20
increment2.addColumn(stock, Mike, 3) // Mike counter =3
Result result2 = table.increment(increment2);

Increment increment3 = new Increment(erasers);
increment1.addColumn(stock, quantity, -2); //quantity counter=8
increment1.addColumn(stock, Mike, 2) // Mike counter =2
Result result3 = table.increment(increment3);
```

Here is same example for three rows, pens, notepads, and erasers. For each row we want to subtract from the quantity column and add to the Mike column. Remember that the atomicity is by row.

For each row we instantiate the increment object with the row key. Then we call `addColumn` for the quantity and Mike columns, subtracting from the quantity column the amount that we want to add to the Mike column.

Finally we call `table.increment()`. The result adds one pen, three notepads, and two erasers to the Mike column values, and subtracts these amounts from the quantity column values.

## Counters – Increment Operation Example

```
Increment increment1 = new Increment(rowA);
increment1.addColumn(hourly, clicks); // new clicks counter = 1
increment1.addColumn(hourly, hits, 25); // new hits counter = 25

Result result1 = table.increment(increment1);
for (KeyValue kv : result.raw()) {
    System.out.println("KV: " + kv +
        " Value: " + Bytes.toLong(kv.getValue()));
}

Increment increment2 = new Increment(rowA);
increment2.addColumn(hourly, clicks, 0); // clicks counter = 1
increment2.addColumn(hourly, hits, -5); // hits counter = 20

Result result2 = table.increment(increment2);
```

Here is another example of an increment operation. We have two counter columns, a hits column, and a clicks column. In this example we are working with a single column family called hourly and a single row, rowA.

Assume that rowA is new and remember that a new counter starts off with zero. Remember, passing no value to the new clicks counter will increment it to one. The result after `table.increment(increment1)` will be the click column value which is one, and the hits counter will be 25.

The result after `table.increment(increment2)` will be the clicks column value which is one, and the hits counter will be 20. Passing zero to the clicks counter just read the value one and passing a negative value to the hourly counter will decrement it.

## Knowledge Check



## Knowledge Check



Please check all of the statements below that are true:

1. Counters are used to track activity in general like clicks, page hits or ad display counts.
2. Counters are fast and atomic.
3. Each counter is designated to a specific column.
4. A counter must use a long as its value.

Answers: 1 – T, 2 – T, 3 – F, 4 – T

## Lab 10.2: Java Applications Using Increment



Open your lab guide to complete Lab 10.2.



## Next Steps

### DEV 335 – Developing Apache HBase Applications: Advanced

Lesson 11: Working with MapReduce on HBase

Congratulations! You have finished DEV 335, Lesson 10: Advanced HBase Java API.  
Continue on the Lesson 11 of this course to learn about working with MapReduce on HBase.



## **DEV 335 – Developing Apache HBase Applications: Advanced**

Lesson 11: Working with MapReduce on HBase

Winter 2017, v5.1

Welcome to DEV 335, Lesson 11: Working with MapReduce on HBase.

## Learning Goals



## Learning Goals



- 11.1 Describe MapReduce
- 11.2 Describe How MapReduce is Used on HBase
- 11.3 Develop MapReduce Applications for HBase

When you have finished this lesson, you will be able to:

- Define MapReduce
- Describe how MapReduce is used on HBase and
- Develop MapReduce programs for HBase

## Learning Goals



### 11.1 Describe MapReduce

11.2 Describe How MapReduce is Used on HBase

11.3 Develop MapReduce Applications for HBase

The primary learning objective of this section is to understand the fundamentals of the MapReduce programming paradigm.

## Lisp Map-Reduce



$(\text{map square } '(1 \ 2 \ 3 \ 4)) = (1 \ 4 \ 9 \ 16)$

- Applies same logic to each value, *one value at a time*
- Emits result for each value

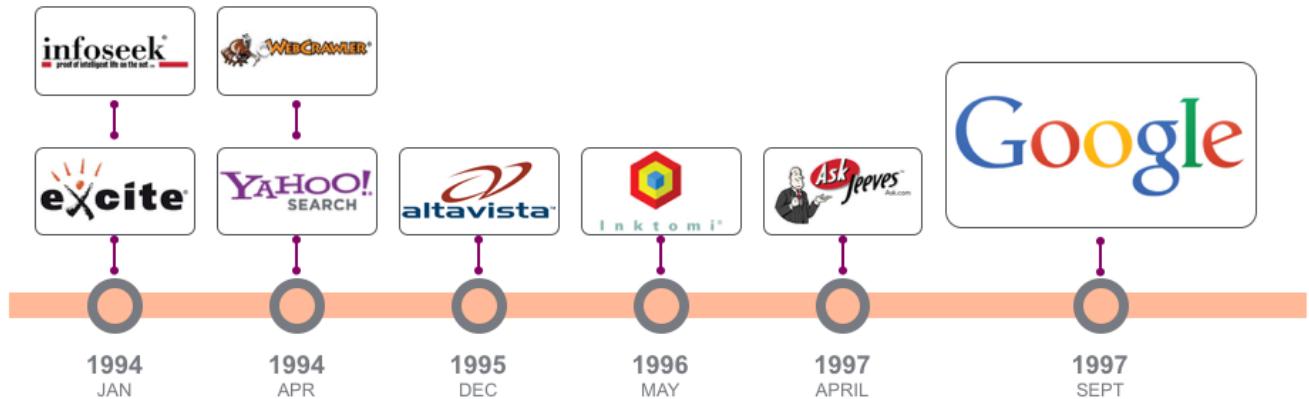
$(\text{reduce } + '(1 \ 4 \ 9 \ 16)) = 30$

- Applies same logic to *all the values* taken together
- Emits single result for all values

MapReduce was not invented at Google and did not start with Hadoop. For example, map and reduce functions have been used in Lisp since the 1970s.

This example shows a map of the square function on an input list from 1 to 4. The square function, since it is mapped, will apply to each of the inputs and produce a single output per input in this case 1, 4, 9, and 16. The addition function reduces the list and produces a single output which is the sum of the input.

## Web Search Engines



Google is the poster child for the power of the MapReduce paradigm, which is the engine behind Hadoop. Google was the 19th search engine to enter a crowded market. You might recall some of these old search brands, but you might not, because within a few short years, Google emerged and dominated the search market.

## Simplified Web Search Engine Approach

- 
- 1 Crawl Web
  - 2 Sort Pages by URL
  - 3 Remove Junk
  - 4 Rank Results
  - 5 Create Inverted Index

Let's take a high level look at how a search engine works.

First, we want to crawl the web. This involves using web "spiders" to crawl the web, following links within web pages to get to other web pages. Overall, this is the most time-consuming step.

Second, we will sort the pages by URL and third, we remove the junk. A good search engine removes "junk" from a known list of bad sites, or contextually based on the what is inside a given page.

Fourth, we rank the results. The URLs are sorted for a given word or set of words based on criteria like frequency, number of hits or freshness of page.

And finally, we will create an index. For each word, we create a list of URLs that contain that word.

## Word Count Algorithm from Google White Paper

```
map(String key, String value):
    // key: document name
    // value: document contents
    for each word w in value:
        EmitIntermediate(w, "1");

reduce(String key, Iterator values):
    // key: a word
    // values: a list of counts
    int result = 0;
    for each v in values:
        result +=ParseInt(v);
    Emit(AsString(result));
```

MapReduce: Simplified Data Processing on Large Clusters Jeffrey Dean (jeff@google.com) & Sanjay Ghemawat (sanjay@google.com) , Google,Inc.

MAPR Academy

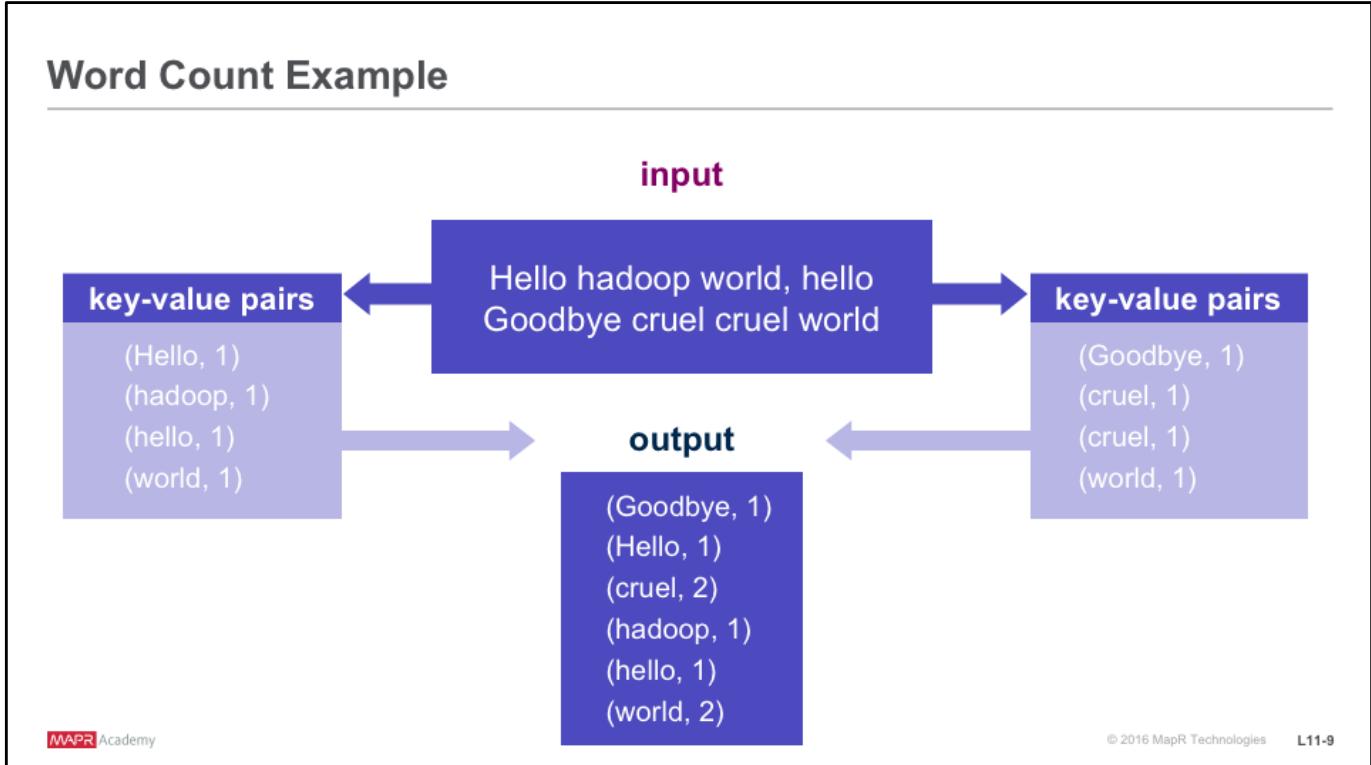
© 2016 MapR Technologies

L11-8

The word count algorithm shown here is taken directly from the seminal paper on MapReduce from Dean and Ghemawat. Algorithms for counting words existed before this paper was published, but the mechanism for doing it using the MapReduce framework was novel, and the algorithm is quite simple. This is true of most MapReduce programs.

The map method takes as input a key and a value, where the key represents the name of a document and the value is the contents of the document. The map method loops through each word in the document and emits a 2-tuple representing word, and 1 in this example.

The reduce method takes as input a key and a list of values, where the key represents a word. The list of values is the list of counts for that word. In this example, the value is a list of 1's. The reduce method loops through the counts and sums them. When the loop is done, the reduce method emits a 2-tuple representing word, and count.

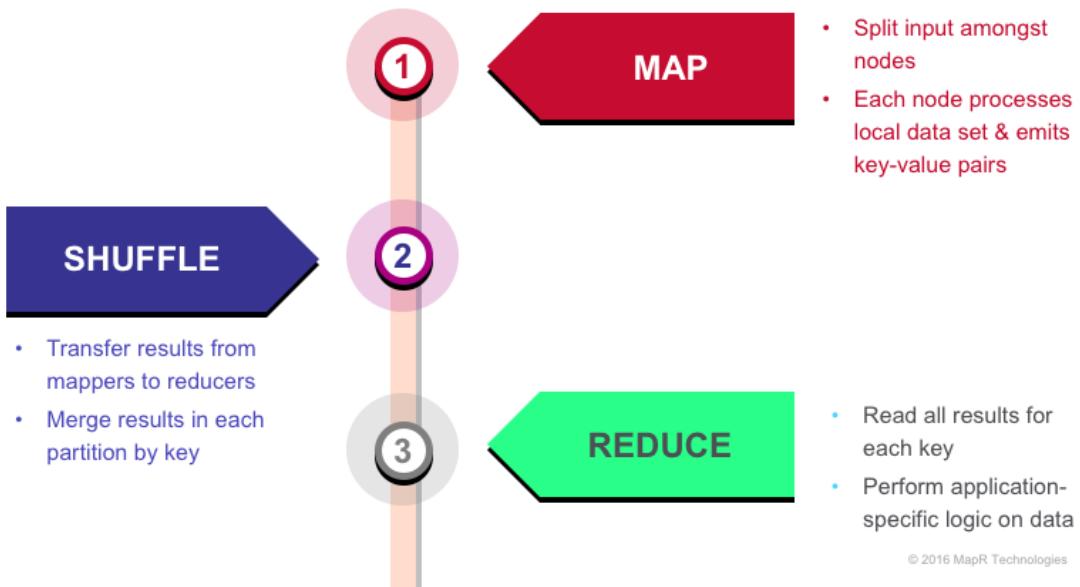


The input for our wordcount example is shown above.

First, each mapper in the map phase takes an input list, such as the first line “Hello hadoop world, hello” and maps it to a list of key-value pairs.

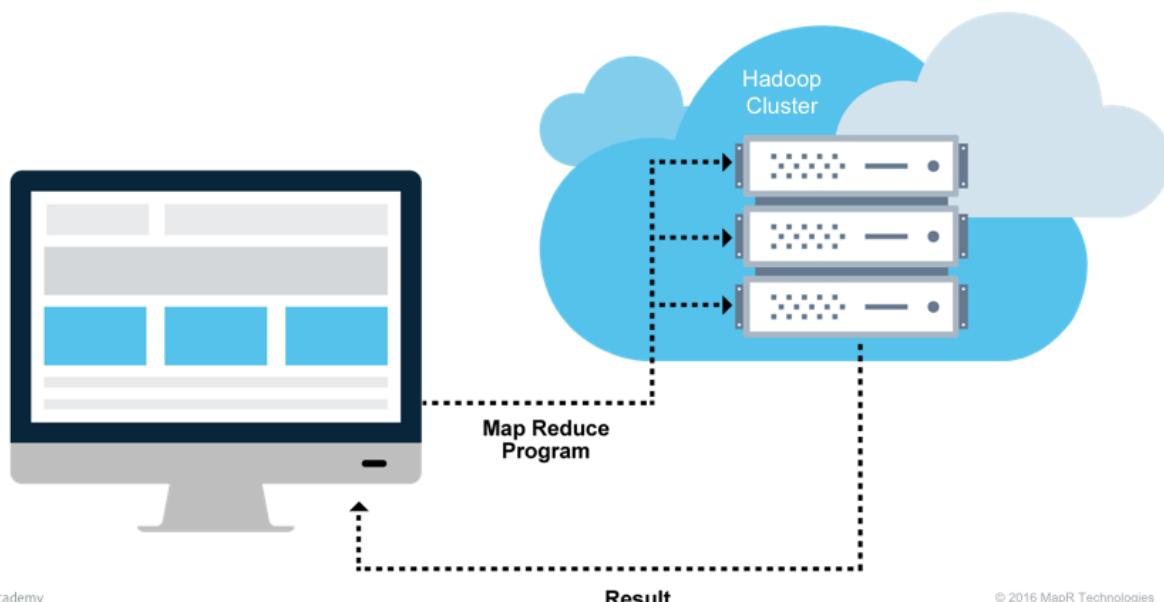
The reducer in the reduce phase takes an input of a key and the list of values associated with that key, such as the two world and 1 pairs from the two different mappers, and emits a single output for that word, world and the sum 2.

## Describe a Summary of Hadoop MapReduce



In this high-level summary of the Hadoop MapReduce computational model, we see that there are actually three phases in MapReduce: map, shuffle, and reduce. The data in the map phase is split amongst the task tracker nodes where the data is located. Each node in the map phase emits key-value pairs based on input one record at a time. The shuffle phase is handled by the Hadoop framework. Output from the mappers is sent to the reducers as partitions. The data in the reduce phase is divided into the partitions in which each reducer reads a key and iterable list of values associated with that key. The reducers emit zero or more key-value pairs based on the application logic.

## Describe the Hadoop Runtime Model

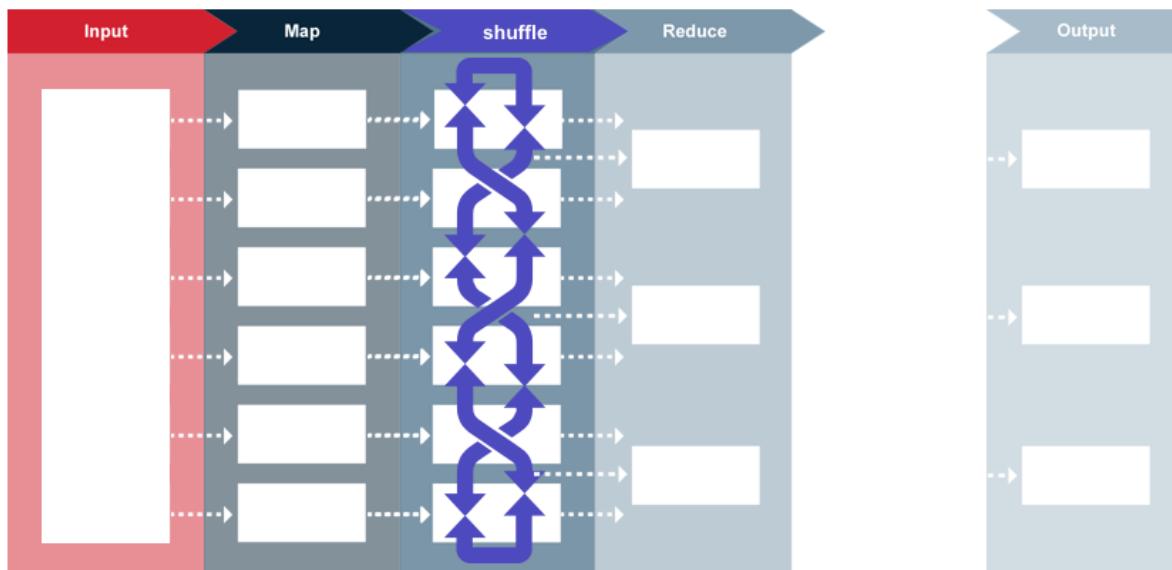


MAPR Academy © 2016 MapR Technologies L11-11

The MapReduce model is based on sending compute to where data resides.

We collect source data on the Hadoop cluster, either by bulk copying data in, or by simply accumulating data in the cluster over time. When we kick off a MapReduce job, Hadoop sends map and reduce tasks to appropriate servers in the cluster, and the framework manages all the details of data passing between nodes. Much of the compute happens on nodes with data on local disks, which minimizes network traffic. Finally, we can read back the result from the cluster.

## MapReduce Flow



MAPR Academy

© 2016 MapR Technologies

L11-12

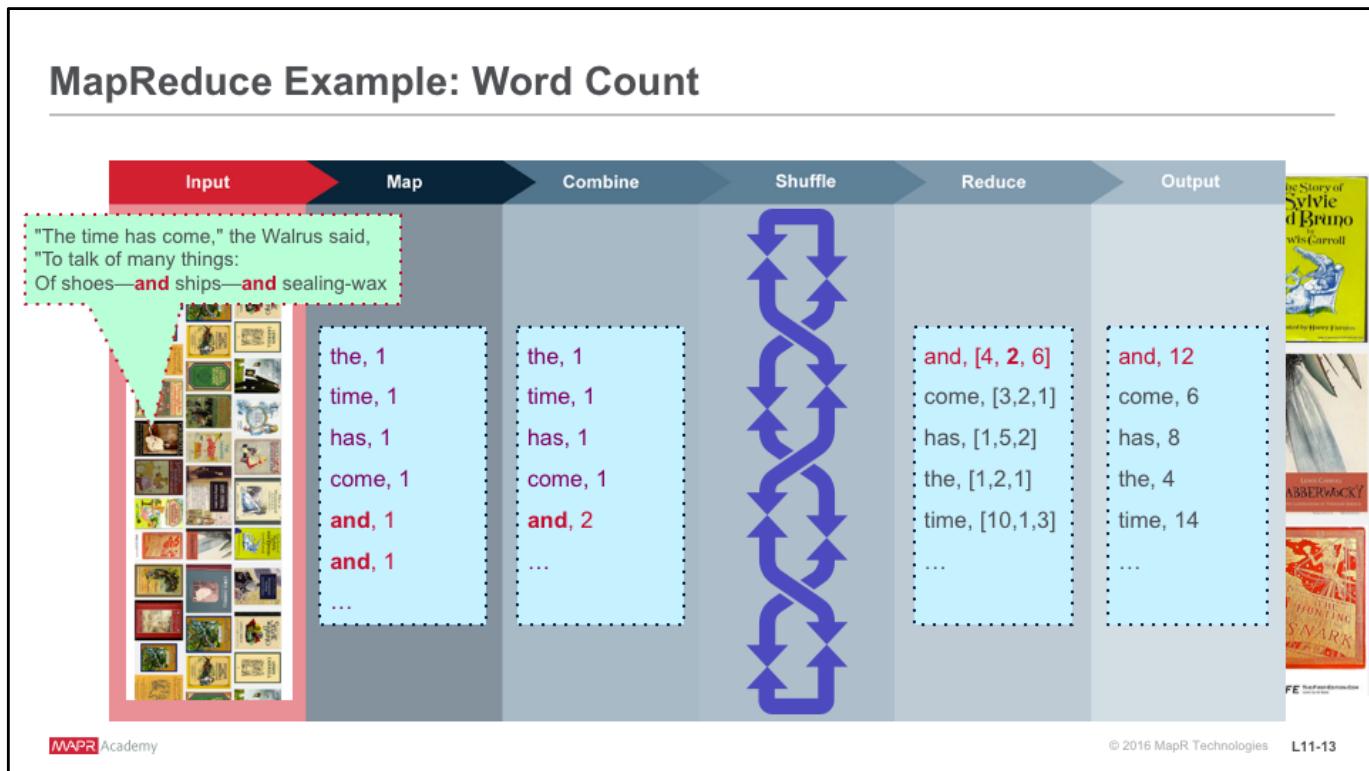
Stepping through the flow of a MapReduce job:

We start off with our input, maybe one file, maybe many files.

- The framework logically breaks up the input into “splits”. Each split contains many records. Records can be any type of information: text, audio data, structured records, or any other type of data we want to process. Each split typically corresponds to a block of data on a node where the data resides, but the programmer doesn’t need to be aware of this.
- Each split is processed by a map task. Each record in a split is passed, independently of other records, to the map() method. The map() method receives each record as a key-value pair, although in some cases it might only be interested in either the keys or the values. The mapper emits key-value pairs in response to the input record. It can emit zero records, or it can emit lot of records.
- The map output is partitioned such that all records of a particular key go to the same reduce task. This phase involves a lot of copying and coordination between nodes in the cluster, but the programmer does not have worry about these details.
- If the reduce method is like addition, where summing subtotals of terms is equivalent to summing all individual terms, it is more efficient to split the reduce step and do some of it before shuffle. In this case, a combiner method is used, which is often the same method as the reducer. This cuts down the number of records that need to be copied from node to node.
- Whether or not a combiner is invoked, the framework will send the intermediate results from the mappers to the reducers. The reduce() method receives a single key and a list of all values associated with that key. The reducer, based on your logic, will emit 0 or more key- value pairs which constitute the final results of this map-reduce job.
- Finally, the framework collects the reducer outputs so you can access the results.

Most of the parts here are handled by the framework. We only have to provide the code in the map and reduce columns.

Note that, at this level of detail, the boxes do not represent nodes in the cluster. We are talking only about logical flow of data. The framework issues map tasks and reduce tasks in parallel, and for example, several map tasks might run concurrently on a single node.



Let's step through an example word-count program using MapReduce to count the occurrences of each word in a set of input text files.

WordCount is the “Hello World” program for MapReduce, though counting strings may be of real use in your programs.

As input, let's say we have all Lewis Carroll's books, and we want to count the occurrence of every word.

Hadoop divides all of the input text into “input splits.”

One of the nodes contains Tweedledee's poem, “The time has come to talk of many things” and so forth. In the case of text input, every line of text is a record. Each record gets fed individually to the map method as key-value pairs. The key is the byte-offset into the file, the value is the text.

The map method tokenizes the input string, and outputs a key-value pair for every word. In this case, the key is the word, and the value is simply 1. As you can see, the word “and” shows up twice, and it produces two distinct key-value pairs

Some combining occurs before shuffle-and-sort. The combiner aggregates multiple instances of the same key coming out of the mapper into a subtotal. In our case, the two instances of the string “and” get combined, and only one record of value 2 goes through the shuffle.

The framework sorts records by key and, for each key, sends all records to a particular reducer. For example, one particular reducer may get the keys: and, come, has, the & time. You can see that the combined value 2 for “and” shows up here. The other values come from other map tasks.

Then begins the reduce phase, which just has to sum up values from the Mappers.

Finally, the framework gathers the output and deposits it in the file system where we can read it.

## Learning Goals



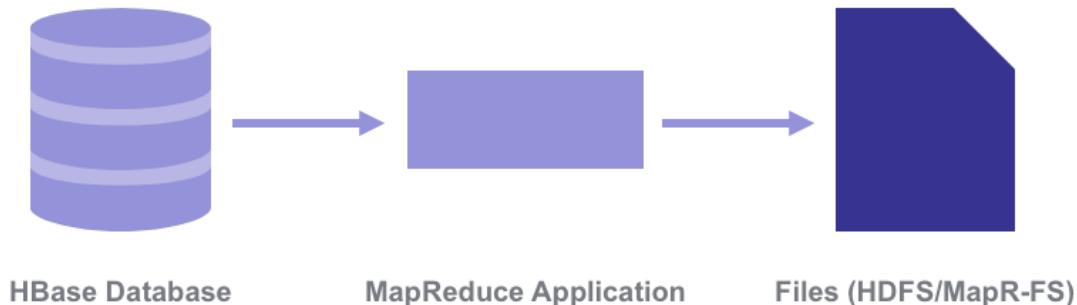
## Learning Goals



- 11.1 Describe MapReduce
- 11.2 Describe How MapReduce is Used on HBase**
- 11.3 Develop MapReduce Applications for HBase

The primary learning objective of this section is to describe how MapReduce may be used in HBase deployments.

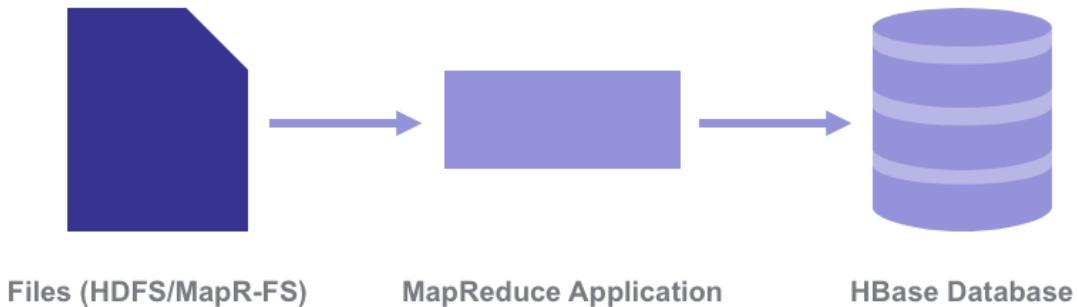
## Using HBase as a Source



### EXAMPLE: Well-defined Analytical Processing Queries

One way to leverage MapReduce with HBase is using the data in the HBase database as the source of your data flow. An example of doing so is a set of well-defined Batch Analytical Processing queries that we implement using MapReduce.

## Using HBase as a Sink

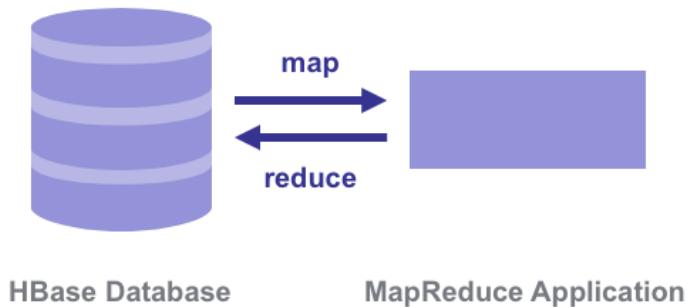


### EXAMPLE: Bulk Load Data Using ImportTsv

Another way to leverage MapReduce in HBase is to use the HBase database as a sink for our data flow. The input for the MapReduce application can come from any variety of sources, including files. For example, we can use HBase in a MapReduce application with the `ImportTsv` utility to bulk load data.

When writing a lot of data to an HBase table from a MapReduce job, such as with [TableOutputFormat](#), and specifically where Puts are being emitted from the mapper, we will skip the reducer step. When a reducer step is used, all of the output (Puts) from the mapper will get spooled to disk, then sorted/shuffled to other reducers that will most likely be off-node. It is far more efficient to just write directly to HBase.

## Using HBase as a Source and Sink



### EXAMPLE: Calculate and Store Summaries, Pre-Computed, Materialized View

The last way to leverage MapReduce in HBase is to use the HBase database as both a source and sink in our data flow. One example of this use case is to calculate summaries across the HBase data and then store those summaries back in the HBase database. For summary jobs where HBase is used as a source and a sink, then writes will be coming from the reducer step.

## Map and/or Reduce

		Output		
		Raw Data	Table 1	Table 2
Input	Raw Data	Map + Reduce (Hadoop)	Map only or Map + Reduce	Map only or Map + Reduce
	Table 1	Map only or Map + Reduce	Map + Reduce	Map
	Table 2	Map only or Map + Reduce	Map	Map + Reduce

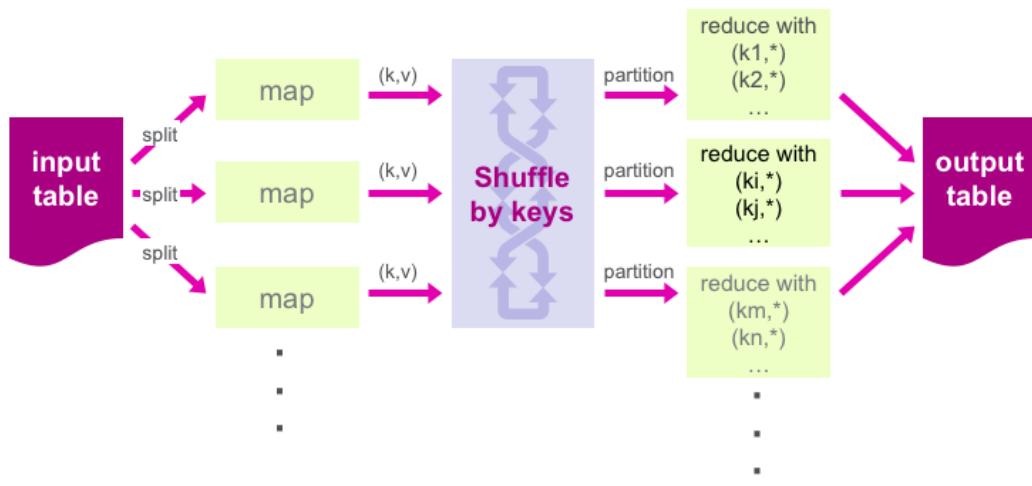
<http://www.larsgeorge.com/2009/05/hbase-mapreduce-101-part-i.html>

When we want to import a large set of data into an HBase table we can read the data using a mapper, and after aggregating it on a per key basis, use a reducer and finally write it into an HBase table. This involves all steps in MapReduce, including the shuffle and sort of intermediate files.

But what if we know that the data already has a unique key? Should we go through the extra step of copying and sorting when there is always just exactly one key-value pair? Wouldn't it be better if we could skip that whole reduce stage? We can do that, and when we do, we will harvest the pure computational power of all CPU's to crunch the data and writing it at top I/O speed to its final target.

As seen in the matrix shown here, there are quite a few scenarios where we can decide if we want Map only or both, map and reduce. When it comes to handling HBase tables as sources and targets, there are a few exceptions to this rule.

## Data Flow in MapReduce with HBase



In this example, HBase is used as both the source and sink for the MapReduce processing. The data is split based on regions and all map tasks that process data from the same region are sent to that file server. Intermediate results from the mappers are shuffled into partitions such that all the intermediate results with the same key belong to the same partition. There is one partition per reducer and there may be more than one key in the same partition.

Finally, the results are written to a new HBase table.

## Learning Goals



## Learning Goals



- 11.1 Describe MapReduce
- 11.2 Describe How MapReduce is Used on HBase
- 11.3 Develop MapReduce Applications for HBase**

The primary learning objective of this section is to describe how to write MapReduce applications, where HBase is a source, a sink, or both.

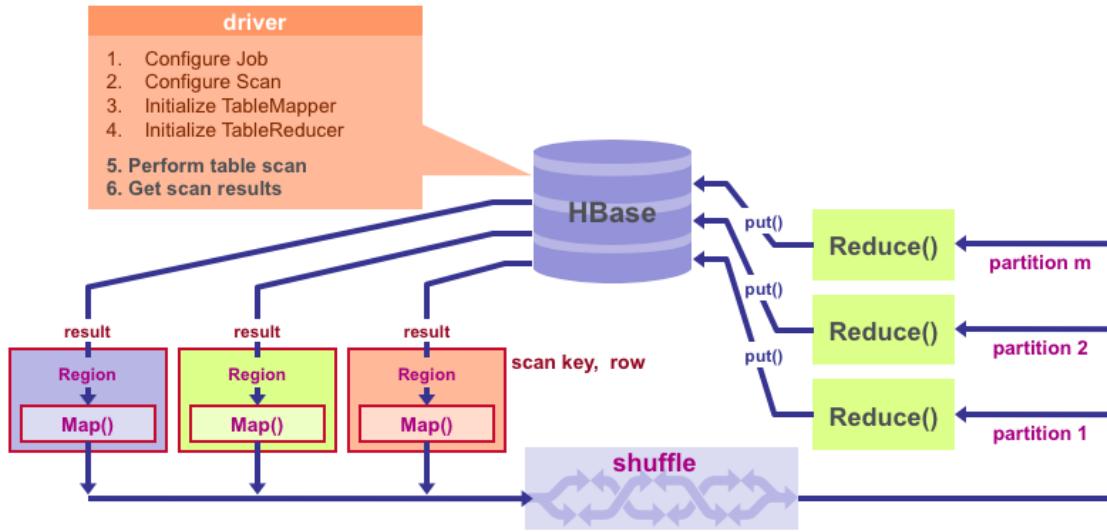
## Programming Choices

API	Pros	Cons
TableMapReduceUtil & HBase	<ul style="list-style-type: none"><li>Least programming</li></ul>	<ul style="list-style-type: none"><li>Least flexibility and control</li></ul>
MapReduce & HBase	<ul style="list-style-type: none"><li>More flexibility and control</li></ul>	<ul style="list-style-type: none"><li>More programming</li></ul>
TableMapReduceUtil & MapReduce & HBase	<ul style="list-style-type: none"><li>Most flexibility and control</li></ul>	<ul style="list-style-type: none"><li>Requires knowing all three</li></ul>

NOTE: You can use other systems (e.g. HDFS/MapR-FS, RDBMS) for source and sink

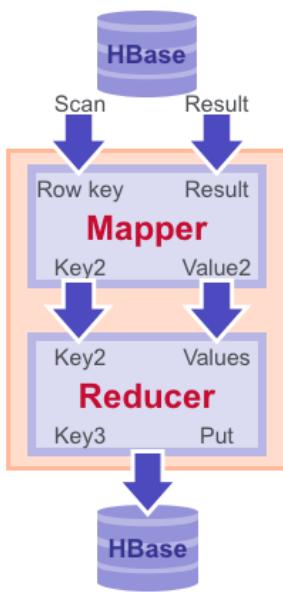
The table above identifies the three choices you have to develop MapReduce applications for HBase. Note that you'll have more choices if you leverage other sources or sinks in your data flow. It is not uncommon, for example, to read in data from HBase, perform MapReduce operations on that data, and then store the results in an external RDBMS database.

## TableMapReduceUtil Model



The graphic above illustrates the programming and execution model when using TableMapReduceUtil for MapReduce jobs in HBase. Note that the Map() and Reduce() steps identify the code we have to write. The rest of it is handled by the TableMapReduceUtil class and the underlying Hadoop framework. Specifically, the table scan is performed by TableMapReduceUtil, and the resulting rows from HBase are fed to Map() calls.

## TableMapper and TableReducer



▼ G Object - java.lang  
  ▼ G Mapper<KEYIN, VALUEIN, KEYOUT, VALUEOUT> - org.apache.hadoop.mapreduce  
    ► G<sup>A</sup> TableMapper<KEYOUT, VALUEOUT> - org.apache.hadoop.hbase.mapreduce

map(ImmutableBytesWritable key, Result row, Context context)

▼ G Object - java.lang  
  ▼ G Reducer<KEYIN, VALUEIN, KEYOUT, VALUEOUT> - org.apache.hadoop.mapreduce  
    ► G<sup>A</sup> TableReducer<KEYIN, VALUEIN, KEYOUT> - org.apache.hadoop.hbase.mapreduce

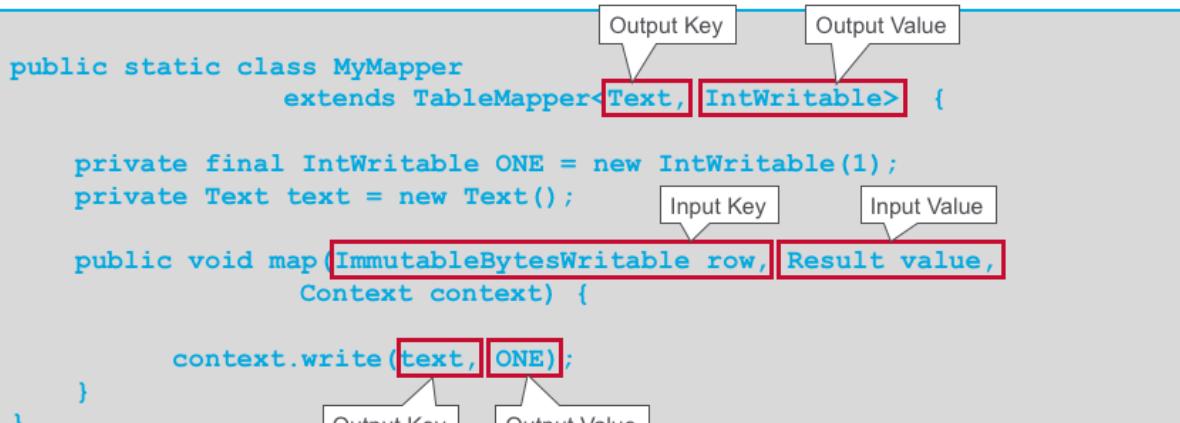
reduce(Text key, Iterable<LongWritable> values, Context context)

This shows the input and output for the TableMapper class map method and the TableReducer class reduce method for reading from and to HBase.

A scan result object and row key are sent to the mapper map method one row at a time, one or more key-value pairs are output by the map method. The reducer reduce method receives a key and an iterable list of corresponding values and outputs a Put object.

## TableMapper

```
public static class MyMapper  
    extends TableMapper<Text, IntWritable> {  
  
    private final IntWritable ONE = new IntWritable(1);  
    private Text text = new Text();  
  
    public void map(ImmutableBytesWritable row, Result value,  
        Context context) {  
  
        context.write(text, ONE);  
    }  
}
```



The diagram illustrates the mapping process. At the top, 'Input Key' and 'Input Value' are shown pointing to 'Output Key' and 'Output Value' respectively. Below this, the code shows the mapping from 'row' (Input Key) and 'value' (Input Value) to 'text' (Output Key) and 'ONE' (Output Value).

This shows the skeleton code for how to write your mapper class extending the TableMapper class which we looked at in the previous slide. The TableMapper class extends the base mapper class to add the required input key and value classes.

## TableReducer

```
public static class MyTableReducer extends TableReducer<Text, IntWritable,  
ImmutableBytesWritable> {  
    public void reduce(Text key, Iterable<IntWritable> values, Context  
    context) throws IOException, InterruptedException {  
        context.write(key, put);  
    }  
}
```



This shows the skeleton code for how to write your reducer class extending the TableReducer class.

The TableReducer class extends the basic Reducer class to add the required key and value input/output classes. The output value must be either a [Put](#) or a [Delete](#) instance when using this class.

## TableMapReduceUtil

```
TableMapReduceUtil.initTableMapperJob(  
    sourceTable,          // input table  
    scan,                // Scan instance  
    MyMapper.class,      // mapper class  
    Text.class,          // mapper output key  
    IntWritable.class,   // mapper output value  
    job);  
  
TableMapReduceUtil.initTableReducerJob(  
    targetTable,          // output table  
    MyReducer.class,     // reducer class  
    job);
```

This shows the `TableMapReduceUtil` init methods that we will use in the driver class for initializing the mapper and reducer jobs. With these methods we can set the input and output tables, the classes, input types, and job.

## Use a Flat-Wide Schema

```
create '/mapr/my.cluster.com/user/user01/trades_flat',
{NAME=>'price', VERSIONS=>1000000},
{NAME=>'vol', VERSIONS=>1000000},
{NAME=>'stats'}
```

Row Key	price:00	price:10	...	price:23	vol	stats:min	stats:max	stats:mean
AMZN_20131010								
GOOG_20131010								
CSCO_20131010								
...								

The slide above describes the Flat-Wide schema used in the code examples that follow. Note there are three column families: price, vol, and stats in the table. The prices are written as cell versions in each hour from “00,” midnight, to “23,” 11pm.

## Summary of Code Example

### Driver class:

- Instantiate Scan on HBase table
- Define TableMapper class and initialize map job
- Define TableReducer class and initialize reduce job
- Run MapReduce job

### TableMapper class:

### TableReducer class:

This slide above summarizes the code example we will walk through in the next few slides. First, we will talk about Driver class.

## Example Driver

```
public class StockDriver extends Configured implements Tool {  
    public int run(String[] args) throws Exception {  
        . . .  
        FileOutputFormat.setOutputPath(job, new Path(args[0]));  
        . . .  
    }  
    public static void main(String[] args) throws Exception {  
        int exitCode = ToolRunner.run(new StockDriver(), args);  
        System.exit(exitCode);  
    }  
}
```

The StockDriver class implements the Tool interface and implements the run method.

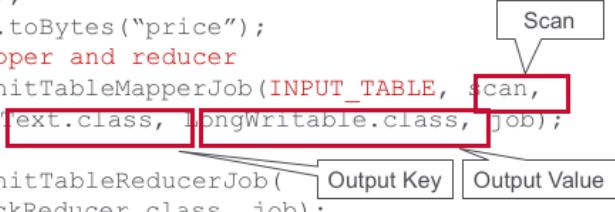
Tool is the standard for any MapReduce tool/application. The driver class delegates the handling of [standard command-line options](#) to the [ToolRunner.run\(Tool, String\[\]\)](#) method.

The driver class contains a main method which calls the run method with the arguments passed to the command line.

Next, we will look at the details for the run method shown here.

## Example Driver continued

```
public int run(String[] args) throws Exception {
    // set up the job
    Job job = Job.getInstance(getConf(), getClass().getSimpleName());
    job.setJarByClass(getClass());
    //Create and configure a Scan instance.
    Scan scan = new Scan();
    scan.setMaxVersions();
    scan.addFamily(Bytes.toBytes("price"));
    // initialize the mapper and reducer
    TableMapReduceUtil.initTableMapperJob(INPUT_TABLE, scan,
        StockMapper.class, Text.class, LongWritable.class, job);
    TableMapReduceUtil.initTableReducerJob(OUTPUT_TABLE, StockReducer.class, job);
    return job.waitForCompletion(true) ? 0 : 1;
}
```

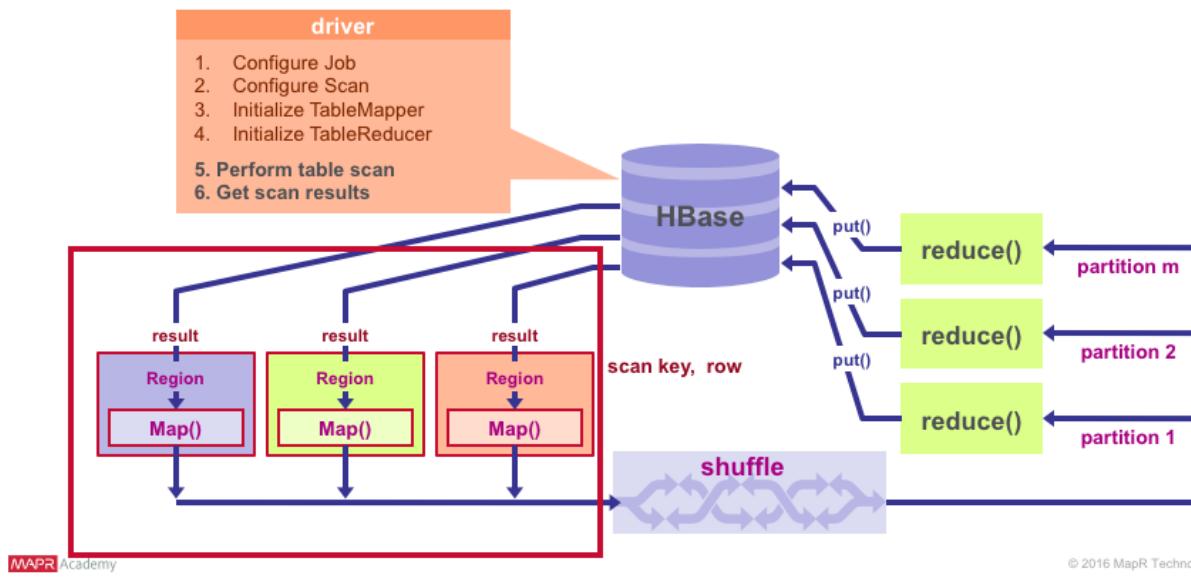


First, we will set up the job. To do this, we will set the jar file to the StockDriver class. We then instantiate a scan object, and request all cell versions and the price column family to be returned.

Next, we initialize the map and reduce classes, the input and output table, and the parameters using the TableMapReduce Util methods. The scan object is passed to the initTableMapperJob() method.

Finally, we launch the job. waitForCompletion will launch the job if it's not already running.

## TableMapReduceUtil Model



TableMapReduceUtil instantiates a ResultScanner before calling the map method and passes it the iterable results from the scan.

One map task is launched for every region in the HBase table. In other words, the map tasks are partitioned such that each map task reads from a region independently.

## Summary of Code Example

Driver class:

**TableMapper class:**

- Process records from Scan iterator
- Emit intermediate key-value pairs

TableReducer class:



MAPR Academy

Next, we will look at the code for the TableMapper class, which processes records from the scan and emits intermediate key-value pairs.

## Input Row from Schema Used in Example



Row Key	price:00	price:10	...	price:23
AMZN_20131010				
GOOG_20131010				
CSCO_20131010				
...				

This again is the HBase table schema for our example, showing the data that will be in the iterable results from the scan, which are passed to the map method. When we then instantiated the scan object, we requested all cell versions and the price column family to be returned.

## Example TableMapper

```
public class StockMapper extends TableMapper<Text, LongWritable> {  
  
    @Override  
    protected void map(ImmutableBytesWritable key, Result row, Context context)  
        throws IOException, InterruptedException {  
        ...  
        // emit key-value as symbol-price  
        context.write(tradeSymbolString, new LongWritable(price));  
    }  
}
```

The diagram illustrates the flow of data from the map method parameters to the final output. At the top, three boxes represent the input: 'Scan Key' (pointing to 'key'), 'Scan row' (pointing to 'row'), and 'Job Context' (pointing to 'context'). Arrows point from these inputs down to the corresponding lines in the code. Below the code, two boxes labeled 'Output Key' and 'Output Value' are shown, each with an arrow pointing to its respective part in the 'context.write' call: 'tradeSymbolString' for the key and 'new LongWritable(price)' for the value.

The StockMapper class extends the TableMapper class from the TableMapReduceUtil package.

Input to the map method includes the key and row from the scan result, along with the job context for this map job. The map method outputs a key-value by calling the job context write method with the output key and value corresponding to the input row key.

## Input Row from Schema Used

```
protected void map(ImmutableBytesWritable key, Result row,
                  Context context) {
    String tradeSymbolString = Bytes.toString(key.get());
    // for all columns in input row
    for (KeyValue col : row.list()) {
        byte[] cellValueBytes = col.getValue();
        long price = Bytes.toLong(cellValueBytes);
        // emit key-value as symbol-price
        context.write(tradeSymbolString, new LongWritable(price));
    }
}
```

rowkey	price:00	price:10	...	price:23
AMZN_20131010				
GOOG_20131010				
CSCO_20131010				
...				

This shows the details for creating the output key-value pairs in the map method. When we instantiated the scan, we requested all cell versions to be part of the scan result. This code iterates over all of the cell versions for every column in the row to emit a stock symbol and price for each cell in the input row.

## Example Mapper Test

```
public class StockMapperTest {  
    Scan Key  
    Scan row  
    Output Key  
    Output Value  
    MapDriver<ImmutableBytesWritable, Result, Text, LongWritable> mapDriver;  
  
    @Before  
    public void setUp() {  
        StockMapper mapper = new StockMapper();  
        // test harness  
        mapDriver = MapDriver.newMapDriver(mapper);  
    }  
}
```

Next we will go over the unit test for the StockMapper class. MRUnit makes it easy to test MapReduce jobs including the HBase ones.

This slide shows the setup for the unit test. The map driver is a harness that allows us to test a mapper instance. To set up the MapDriver with pass in our mapper class which we want to test.

## Example Mapper Test

```
@Test
public void testHBaseInsert() throws IOException {
    String inKey = "AMZN_20131021";
    // Setup Test input (key, scan Result row list)
    ImmutableBytesWritable key =
        new ImmutableBytesWritable(Bytes.toBytes(inKey));
    Result result = new Result(columnValuesList);
    // Set Input
    mapDriver.withInput(key, result);
    // Scan Key
    // Scan row
    // run the reducer and get its output
    List<Pair<Text, LongWritable>> mapResult = mapDriver.run();
    // output is key price
    assertEquals(outKey, mapResult.get(0).getFirst());
    // Output Key
    assertEquals(priceOut, mapResult.get(0).getSecond());
    // Output Value
}
```

-39

This shows the implementation of the Unit test for the StockMapper class.

The map driver is a harness that allows us to test a mapper instance. First we set up the input key and value that should be sent to the mapper, in this case a row key and result object. We then call the MapDriver method passing the input for the mapper.

The harness will deliver the input to the mapper and return the mapper output. Next ,we test the map result with the expected output.

## Summary of Code Example

Driver class:

TableMapper class:

### TableReducer class:

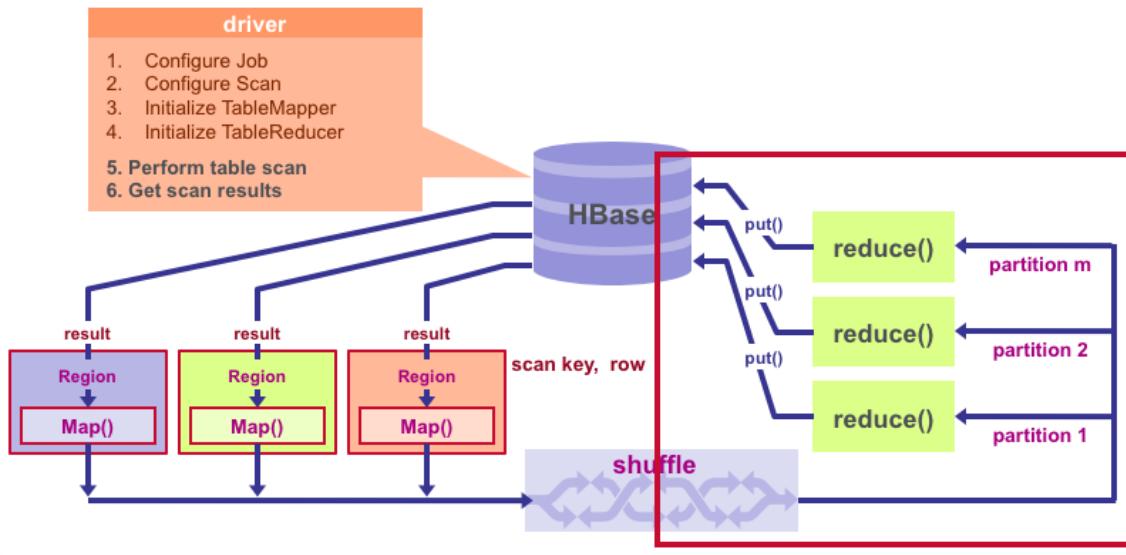
- Process intermediate key-value pairs
- Write back to HBase table



MAPR Academy

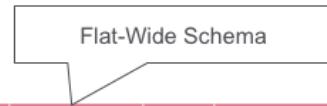
Next, we will go over the code for the reducer class.

## TableMapReduceUtil Model



TableMapReduceUtil and the MapReduce framework takes care of sorting the keys, sending a key and corresponding values to the reducers, and writing the output from the reducers to the HBase table.

## Put to Flat-Wide Schema Used in Example

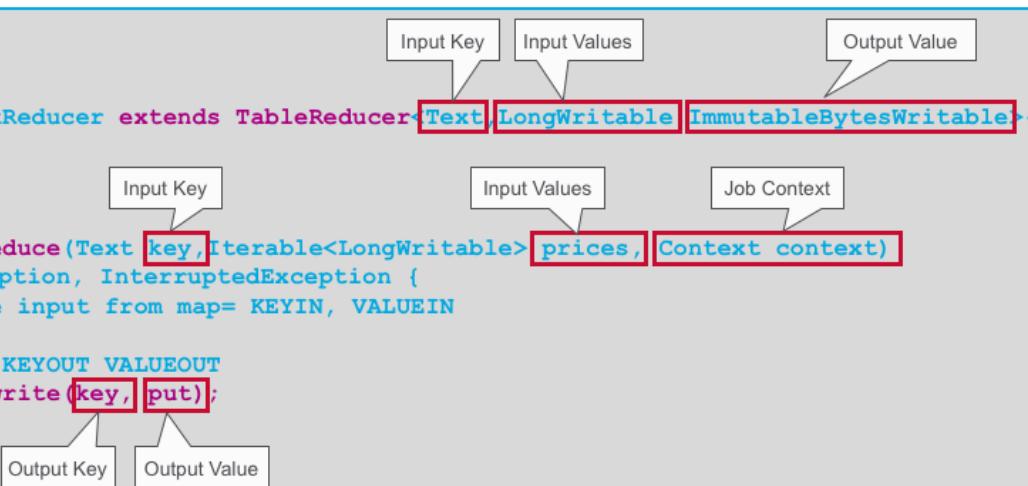


Row Key	price:00	price:10	...	price:23	vol	stats:min	stats:max	stats:mean
AMZN_20131010								
GOOG_20131010								
CSCO_20131010								
...								

Remember the reducer will output the Put row objects with the value for the stats column family min column.

## Example TableReducer

```
public class StockReducer extends TableReducer<Text, LongWritable, ImmutableBytesWritable> {  
  
    @Override  
    protected void reduce(Text key, Iterable<LongWritable> prices, Context context)  
        throws IOException, InterruptedException {  
        // reduce input from map= KEYIN, VALUEIN  
        . . .  
        // write KEYOUT VALUEOUT  
        context.write(key, put);  
    }  
}
```



The StockReducer class extends the TableReducer class.

The reduce method has three inputs: the input key, an iterable list of prices associated with that key, and the job context associated with the reduce job.

This reduce method outputs a row key and Put object with the context write() method. This Put object will be written to the table by the framework. Next, we will look at the details of calculating the min value and creating the Put object.

## Example TableReducer continued

```
public class StockReducer extends  
    TableReducer<Text, LongWritable, ImmutableBytesWritable> {  
  
    @Override  
    protected void reduce(Text key, Iterable<LongWritable> prices, Context context) {  
        long min=Long.MAX_VALUE;  
        long temp = 0L;  
        for (LongWritable price : prices) {  
            temp=price.get();  
            if(temp < min) min=temp;  
        }  
        Put put = new Put(Bytes.toBytes(key.toString()));  
        put.add(STATS_FAMILY, MIN_COLUMN),  
            Bytes.toBytes(Float.toString((float)min/100)));  
        context.write(key, put);  
    }  
}
```

This reduce method iterates over the input prices to find the minimum value. Other statistics could be calculated in a similar fashion. The reduce method then constructs a Put object with the row key and adds the minimum value to the stats column family min column. Then it outputs the put the context.write method.

## Put to Flat-Wide Schema Used in Example

```
Put put = new Put(Bytes.toBytes(key.toString()));
put.add(STATS_FAMILY, MIN_COLUMN),
       Bytes.toBytes(Float.toString((float)min/100));
context.write(key, put);
```

Row Key	price:00	price:10	...	price:23	vol	stats:min	stats:max	stats:mean
AMZN_20131010								
GOOG_20131010								
CSCO_20131010								
...								

This diagram shows that the constructed Put object value is for the min column in the stats column family.

## Example Reducer Test

```
public class StockReducerTest {  
      
    ReduceDriver<Text, LongWritable, ImmutableBytesWritable, Writable>  
    reduceDriver;  
  
    @Before  
    public void setUp() {  
        StockReducer reducer = new StockReducer();  
        reduceDriver = ReduceDriver.newReduceDriver(reducer);  
    }  
}
```

The ReduceDriver is a harness that allows us to test our reducer.

Here is the setup for testing the StockReducer: we instantiate the StockReducer and pass it to the ReduceDriver class's newreducedriver() method.

## Example Reducer Test

```
@Test
public void testHBaseReduce() throws IOException {
    // Setup Input values
    String strKey = "GOOG";
    List<LongWritable> list = new ArrayList<LongWritable>();
    // Set Input to what mapper would pass
    reduceDriver.withInput(new Text(strKey), list);
```

```
    List<Pair<ImmutableBytesWritable, Writable>> result = reduceDriver.run();

    // extract values for put CF/QUALIFIERS and verify
    Put put = (Put) result.get(0).getSecond();
```

```
    KeyValue minKeyValue = put.get(StockDriver.COLUMN_FAMILY1,
        StockDriver.MIN_QUALIFIER).get(0);
    String min = Bytes.toString(minKeyValue.getValue());
    assertEquals(Float.toString((float) inMin / 100f), min);
}
```

```
}
```

MAPR Academy

© 2016 MapR Technologies

L11-47

First, we provide a row key and a set of price values corresponding to inputs that should be sent to the reducer, as if they came from a mapper. By calling the ReduceDriver run method, the harness will deliver the input to the reducer and return its outputs.

Next, we check the output from the reducer against the expected results, in this case the output should be a put object with the min value corresponding to the input prices.

## Hints and Tips

### Turn off speculative execution for writing to HBase →

- Schedules map or reduce to multiple nodes
- Could **cause multiple writes** to database with reducer puts

### One split/map-task per region by default →

- For **custom splitting**
- **Override** `TableInputFormatBase.getInputSplits()`

### Use Tool/ToolRunner for **simplicity** of use →

- Don't need to use Hadoop command + enables **GenericOptionsParser** for args

### Be careful with your data types, object creation →

- Especially in the **Mapper** class which gets **called once per record**

The slide above defines some hints and tips for writing MapReduce applications in an HBase environment.

Limit child JVM memory

Number of concurrent tasks is limited by per machine RAM

Speculative execution is a feature of Hadoop wherein a map or reduce task is scheduled to multiple data nodes. This is a performance enhancement that will take the results from the first data node to produce the results. However, when there is an outside sink for your data (i.e. HBase), you may find that results have been written multiple times. Therefore, it is recommended to turn off speculative execution when running MapReduce applications with HBase.

By default, the Hadoop framework creates one split per region. If you wish to split the data in a custom way, then you need to override the `TableInputFormatBase.getInputSplits()` method of the `TableMapReduceUtil` class.

The `ToolRunner` utility in Hadoop allows users to run MapReduce jobs as java programs (rather than using the Hadoop command directly). It also allows you to leverage the Hadoop `GenericOptionsParser` if you have multiple command line arguments. The `GenericOptionsParser` enables any ordering of command-line arguments (rather than relying on the user to know the implicit ordering of `args[0], args[1], ... etc`).

Be judicious with your data types. Constructing String objects, for example, is expensive. You must use subclasses of `writable` for how data gets stored in your HBase database, but you are free to use whatever makes sense in your MapReduce applications as data types local to your map and reduce programs when you are transforming your data.

Scan Caching

If HBase is used as an input source for a MapReduce job, make sure that the input Scan instance to the MapReduce job has `setCaching` set to something greater than the default (which is one). Using the default value means that the map-task will make call back to the region-server for every record processed. Setting this value to 500, for example, will transfer 500 rows at a time to the client to be processed. There is a cost/benefit to have the cache value be large because it costs more in memory for both client and RegionServer, so bigger isn't always better.

Scan settings in MapReduce jobs deserve special attention. Timeouts can result (e.g. `UnknownScannerException`) in Map tasks if it takes longer to process a batch of records before the client goes back to the RegionServer for the next set of data. This problem can occur because there is non-trivial processing occurring per row. If you process rows quickly, set caching higher. If you process rows more slowly (e.g. lots of transformations per row, writes), then set caching lower.

Whenever a Scan is used to process large numbers of rows (and especially when used as a MapReduce source), be aware of which attributes are selected. If `scan.addFamily` is called then all of the attributes in the specified Column Family will be returned to the client. If only a small number of the available attributes are to be processed, then only those attributes should be specified in the input scan because attribute over-selection is a non-trivial performance penalty over large datasets.

When writing a lot of data to an HBase table from a MapReduce job (e.g., with [TableOutputFormat](#)), and specifically where Puts are being emitted from the Mapper, skip the Reducer step. When a Reducer step is used, all of the output (Puts) from the Mapper will get spooled to disk, then sorted/shuffled to other Reducers that will most likely be off-node. It's far more efficient to just write directly to HBase.

For summary jobs where HBase is used as a source and a sink, then writes will be coming from the Reducer step (e.g., summarize values then write out result). This is a different processing problem than from the above case.

Here is how to turn off speculative execution:  
`hadoop jar /opt/mapr/hadoop/hadoop-0.20.2/hadoop-0.20.2-dev-examples.jar terasort \`  
`-Dmapred.reduce.tasks=300 \`  
`-Dmapreduce.maprfs.use.compression=false \`  
`-Dmapred.map.tasks.speculative.execution=false \`  
`-Dmapred.reduce.tasks.speculative.execution=false \`

## Lab 11.3a-b



Open your lab guide to complete the following Labs:

1. Lab 11.3a: Developing MapReduce Applications for HBase (Flat-Wide)
2. Lab 11.3b: Developing MapReduce Applications for HBase (Tall-Narrow)



## Next Steps

### DEV 340 – HBase Applications: Bulk Loading, Security, and Performance

Lesson 12: Bulk Loading of Data

Congratulations, you have completed Lesson 11. Continue on to Lesson 12 to learn about the bulk loading of data.



## **DEV 340 – HBase Applications: Bulk Loading, Security, and Performance**

Lesson 12: Bulk Loading of Data

Winter 2017, v5.1

Welcome to DEV 340, Lesson 12, Bulk Loading Data into MapR Tables.

## Learning Goals



## Learning Goals



- 12.1 How Bulk Loading Data Works
- 12.2 Using the ImportTsv Bulk Load Tool
- 12.3 Use MapReduce Job to Import Data
- 12.4 Pre-splitting Table

In this lesson, we are going to discuss several techniques to import data in bulk with an emphasis on importing from a SQL database.

We will start by looking at how bulk loading data works.

Next, we will look at ImportTsv, which is a built-in HBase tool to import text data.

We will then revisit MapReduce to look at some other import techniques. We will go over other clients like Thrift, Hive, Pig, Rest and the command line interface (CLI).

Then we will conclude by looking at when you should consider pre-splitting data before importing it, which is an optimizing technique.

MapR extended the HBase `HFileOutputFormat` class to write to a MapR-DB table using bulk load. Any HBase MapReduce program which works with `HFileOutputFormat` will also work with MapR bulk load. Standard HBase MapReduce jobs for bulkload work with MapR-DB as well.

## Learning Goals



### 12.1 How Bulk Loading Data Works

- 12.2 Using the ImportTsv Bulk Load Tool
- 12.3 Use MapReduce Job to Import Data
- 12.4 Pre-splitting Table

First, we will discuss how bulk loading data works.

## Review

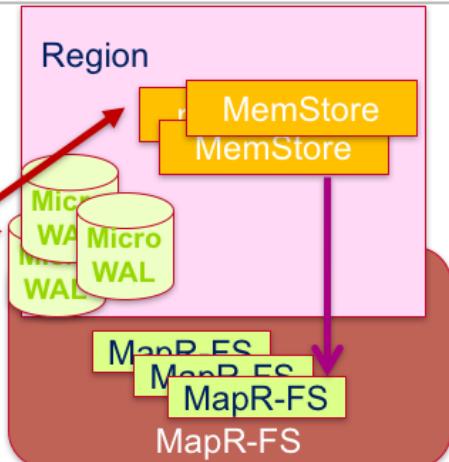


## Review HBase/MapR-DB Write Steps



Put →

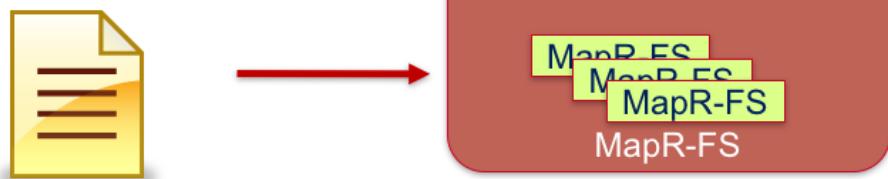
1. Each incoming record written to **WAL on disk**
2. Written to **MemStore** in memory
3. **When MemStore is full, written to disk**



Here is a review of the HBase write steps before we look at bulk loading. Every update writes to the write ahead log on disk, shown here as WAL, and to memory, referred to as the MemStore. The MemStore stores updates in memory as sorted key-values, the same way that it will be stored when it is flushed to disk into an HFile.

## MapR-DB Full Bulk load

1. Bulk import directly creates HBase/MapR-DB sorted key-value files in region
2. **Avoids write to WAL and MemStore**



Bulk loading uses a MapReduce job to perform the following steps:

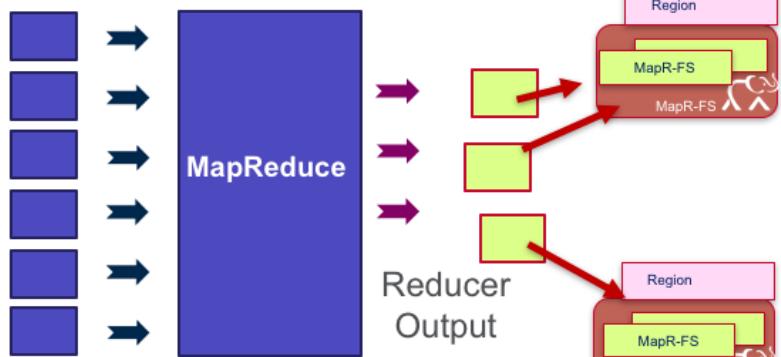
1. Transform the source data into the native file format used by MapR-DB tables.
2. Notify the database of the location of the resulting files.

A full bulk load operation can only be performed to an empty table and skips the write-ahead log typical of Apache HBase and MapR-DB table operations. This results in increased performance. Incremental bulk load operations DO use the write-ahead log.

## Bulk Load MapReduce Directly Loads Data

Directly loads data into MapR-DB table:

1. MapReduce creates files in MapR-DB table format.
2. Database notified of the location of the resulting files.



We can bulk load using a MapReduce job to load data directly into MapR-DB, with the following steps:

1. Transform the source data into the native file format used by MapR tables.
  - The map task emits key-value pairs.
  - The MapReduce framework collects and sorts the keys for partition and sends them to the reducers.
  - Each reducer create files in MapR-DB table format.
2. MapR-DB is notified of the location of the resulting files.

## MapR-DB Bulk Load

MapR-DB Bulk Loading can be performed as a *full bulk load* or as an *incremental bulk load*.

### Full Bulk Load

Offers the best performance advantage.

Only on **empty** table.

Bulkload flag should set to '**true**' Allowed only at **table creation** time.

**Reset** the bulkload=**false** to access the table **after** full bulk load.

No client operations like Put, Get, Scan allowed for full bulk load option.

### Incremental Bulk Load

Set bulkload flag = '**false**'

Client can access table

- Read existing data and data streamed from incremental bulk load.

Faster than Put API, but slower compared to full bulk load.

Bulk loading can be performed as a full bulk load or as an incremental bulk load. A full bulk load offers the best performance advantage for empty tables. Incremental bulk loads can add data to existing tables concurrently with other table operations. This will provide better performance than Put operations, but is allowed only at table creation time.

## MapR-DB Bulk Load Advantages

Apache HBase needs two stages:

1. **Generate HFiles**
2. **Load HFiles** into table

MapR-DB needs only one stage: Directly load data into table

MapR-DB bulkload tools/utilities:

- CopyTable - MapReduce
- CopyTableTest - Non-MapReduce
- ImportFiles
- ImportTsv
- Any custom MapReduce jobs

Apache HBase needs two stages for bulk loading: First, to generate HFiles and second, to load HFiles into the table.

MapR-DB needs only one stage, which is to directly load data into table.

Bulk loading is supported for the following tools, which can be used for both full or incremental bulk load operations:

The CopyTable tool uses a MapReduce job to copy a MapR table.

The CopyTableTest tool copies a MapR table without using MapReduce.

The ImportTsv tool imports a tab-separated value file into a MapR table.

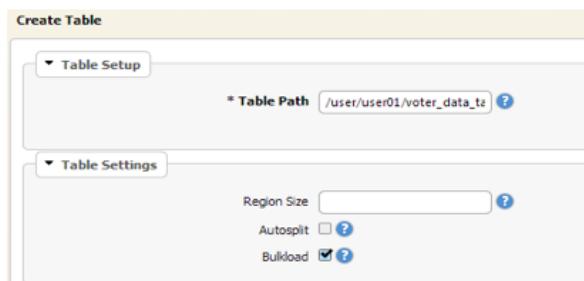
The ImportFiles tool imports HFile or result files into a MapR table.

Custom MapReduce jobs can use bulk loads with  
the `configureIncrementalLoad()` method from the `HFileOutputFormat` class.

## Create a Table with Bulk Load Support

HBase Shell: Create '/a0','f1', BULKLOAD => 'true'

MCS:



- When bulkload is finished set to false
- HBase Shell: alter '/a0','f1', BULKLOAD => 'false'

After completing a full bulk load operation, take the table out of bulk load mode to restore normal client operations. You can do this from the command line or the HBase shell with the commands shown here:

```
#maprcli table edit -path /user/juser/mytable -bulkload false (command line)
HBase shell> alter '/user/juser/mytable', 'f2', BULKLOAD =>
'false' (HBase shell)
```

## MapR-DB Bulk Load Performance : >2x Throughput

HW Specification : 20 data nodes, 1 admin node - 16x2 cores, 11 disks, 128GB RAM, 1x 10GBE			
FULL Bulk Load : 1TB load on empty table - 480 region pre-split (data generation in memory)			
Throughput in MB/s/node	CDH 4.7 + HBase 0.94.15	MapR 3.1.1.26103 + HBase 0.94.17	%Diff
Using full bulkload	84.84	123.65	45.74%
Using Put	21.02	70.28	234.35%
FULL Bulk Load: 3TB load on empty table - 160 region pre-split (ImportTSV - data source is file system)			
Throughput in MB/s/node	CDH 4.7 + HBase 0.94.15	MapR 3.1.1.26103 + HBase 0.94.17	%Diff
Using full bulkload	30.82	57.95	88.03%
Incremental Bulk Load: 500G load on an existing 3TB table (ImportTSV - data source is file system)			
Throughput in MB/s/node	CDH 4.7 + HBase 0.94.15	MapR 3.1.1.26103 + HBase 0.94.17	%Diff
Using Incr bulkload	25.96	40	54.08%

© 2016 MapR Technologies L12-12

This table shows the performance of MapR-DB bulk load compared to HBase.

## Learning Goals



## Learning Goals



12.1 How Bulk Loading Data Works

### **12.2 Using the ImportTsv Bulk Load Tool**

12.3 Use MapReduce Job to Import Data

12.4 Pre-splitting Table

Now, let's learn about how to use the ImportTsv bulk load tool.

## Overview of ImportTsv

- 
- The diagram illustrates the process of bulk loading data. It starts with a blue cylinder labeled "RDBMS". A green arrow labeled "export" points from the RDBMS to a yellow box labeled "TSV or CSV file". A pink arrow labeled "MapReduce" points from the TSV or CSV file to a red cylinder labeled "MapR-DB".
- Export the data from RDBMS to TSV file
    - **Faster** than executing SQL on RDBMS for large amounts of data
  - Use ImportTsv
    - **Efficient** tool to load TSV file into MapR-DB or HBase
    - Runs a MapReduce job to perform the import

ImportTsv is a very efficient tool to load text data into MapR-DB or HBase.

- We will first need to export the data from the RDBMS to a TSV file using our standard tools. This is faster than executing SQL on RDBMS for large amounts of data.
- We can then run ImportTsv on the TSV file. ImportTsv runs a MapReduce job to perform the import.

## Overview of ImportTsv

### ImportTsv

- Utility provided by the HBase package

### Import data from TSV (Tab Separated Values) files

- TSV file must contain a field representing the **row key** of the HBase table row
- Think about which information will be the new row key in the HBase table
- Usually the format is: **row key** \t valueA \t valueB... \t valueN
- Data must be in HDFS to run ImportTsv, not local file system

ImportTsv is a utility provided by the HBase package. The TSV file must contain a field representing the row key of the HBase table row. Usually the format is: row key \tab value1 \tab value2...up to valueN.

From a SQL store, dump data into a text file and then use ImportTsv. This is faster than executing SQL on RDBMS for large amounts of data.

The ImportTsv tool will only read data from MapR-FS or HDFS. The TSV file we want to import must first be on our cluster file system. This usage will load the data via Puts (non-bulk loading).

## Running ImportTsv

Create the target table with bulkload support

- Define column families.

ImportTsv usage:

```
importtsv -Dimporttsv.columns=a,b,c  
<tablename> <inputdir>
```

```
Input data: row key, value1, value2, ... valueN
```

```
$hbase org.apache.hadoop.hbase.mapreduce.ImportTsv \  
-Dimporttsv.columns=HBASE_ROW_KEY,family:name1,...,family:nameN \  
-Dimporttsv.bulk.output=/user/userXX/dummy \  
target_table_name \  
/user/joe/inputDirectory/data.tsv
```

Before we start the job, we need to create the MapR-DB table that we want to import into. We will also need to create the column families and set the column family properties for our schema. Once the job is started, we can monitor it via the web UI.

Usage:

- Imports the given input directory of TSV data into the specified table.
- The ImportTsv tool itself is a Java class included in the HBase JAR file. We will therefore run the tool by executing the Hadoop JAR command. This command will start the Java process and add all dependencies to it.
- The JAR to run is specified by the first parameter of the Hadoop JAR command, and includes the version of HBase we have installed in the name.
- Invoking ImportTsv without any arguments will show what the various options are.

-DimportTsv.columns

- Comma-separated column names of TSV data, where each column name is either a simple column qualifier, or a columnfamily:qualifier.
- HBASE\_ROW\_KEY is used to designate that this column should be used as the row key for each imported record.
- Specify exactly one column to be the row key and specify a column name for every column that exists in the input data.
- Including the -Dimporttsv.bulk.output will generate files for loading (i.e. bulk loading)

Map phase of the job, it reads and parses rows from TSV files under the specified input directory and Puts rows into the HBase table using the column mapping information. The Read and Put operations are executed in parallel on multiple servers, so it is much faster than loading data from a single client. By default, there is no reduce phase in the job.

## Monitoring Regions During Bulk Import

You can see the regions being created and split when importing a file

The screenshot shows the MapR Technologies interface with the cluster name 'BigBertha'. The navigation menu includes 'Cluster' (Dashboard, Nodes, Node Heatmap, Jobs), 'MapR-FS' (MapR Tables, Volumes, Mirror Volumes, User Disk Usage, Snapshots, Schedules), 'NFS HA' (NFS Setup, VIP Assignments, NFS Nodes), and 'Logs'. The main window displays the 'Regions' tab for the table 'table\_data2/table3'. The table has 9 regions:

Start Key	End Key	Physical Size	Logical Size	# Rows	Primary Node	Secondary Nodes	Last HB	Region Identifier
-∞	17084614	739.8MB	1.4GB	7,871,797	CentOS002	CentOS003 CentOS004	0 ago	2324.36.787384
17084614	241566	897.9MB	1.7GB	9,583,927	CentOS005	CentOS002 CentOS001	0 ago	2308.32.525090
241566	475149	3GB	5.9GB	33,141,360	CentOS004	CentOS006 CentOS005	0 ago	2361.32.656030
475149	∞	779.8MB	1.5GB	8,316,776	CentOS006	CentOS002 CentOS003	0 ago	2328.32.526362

MAPR Academy © 2016 MapR Technologies L12-18

As before, after we complete a full bulk load operation, we need to take the table out of bulk load mode to restore normal client operations.

## Pros and Cons

### Pros

- Works well with large amounts of data
- Faster than executing SQL from a relational database
  - You have to export data to text first (SQL dump)

### Cons

- Data needs to be converted to text format specifically TSV
- Often requires to massage the data once dumped from a source like SQL, for example can only have one TSV column as ROW\_KEY in new HBase table
- You don't have full control of how the MapReduce job runs

This process of using ImportTsv works well when you are bulk loading large amounts of data as it is using MapReduce framework to process (shard) the input file to divide the file/task into many mapping tasks. It is fast to use ImportTsv to input data into HBase once we have converted it into a TSV file.

## Lab 12.2: Use ImportTsv, CopyTable



Next, let's work on the lab exercises to use the ImportTsv tool and the CopyTable method for Lab 12.2.

## Learning Goals



## Learning Goals



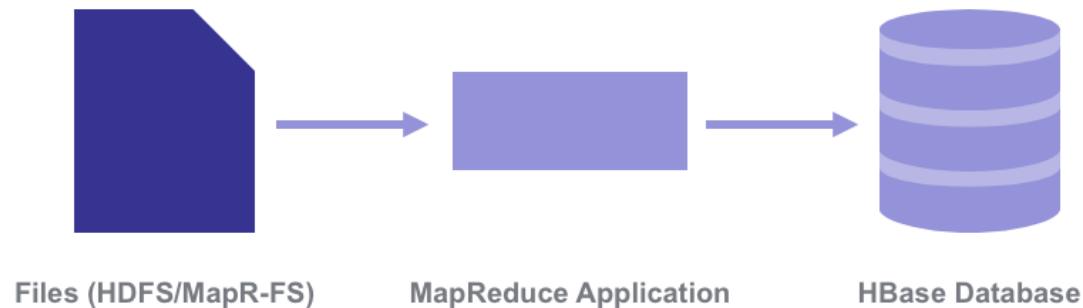
- 12.1 How Bulk Loading Data Works
- 12.2 Using the ImportTsv Bulk Load Tool
- 12.3 Use MapReduce Job to Import Data**
- 12.4 Pre-splitting Table

Now, let's use a MapReduce job to import data.

## MapReduce Output to Table or File

Writing your own MapReduce job helps when importing huge amounts of data

- Example: Ingesting log files to HBase or MapR tables



When we use the ImportTsv utility to bulk load data from our HDFS or MapR-FS file system, it runs a MapReduce application to load the data into the table. We can write a custom MapReduce job to perform bulk loading if ImportTsv does not meet all of your requirements.

## MapR-DB Bulk Load with MapReduce

Custom MapReduce jobs can use bulk loads with the `configureIncrementalLoad()` method from the `HFileOutputFormat` class.

```
HFileOutputFormat.configureIncrementalLoad(mrJob, htable);
```

Custom MapReduce jobs can use bulk loads with the `configureIncrementalLoad()` method from the `HFileOutputFormat` class.

The [HFileOutputFormat](#) class on MapR clusters distinguishes between Apache HBase tables and MapR tables, behaving appropriately for each type. Existing workflows that rely on the `HFileOutputFormat` class, such as the `ImportTsv` and `copytable` tools, support both types of tables without further configuration.

Any HBase MapReduce program which works with `HFileOutputFormat` will also work with MapR bulk load. Standard HBase MapReduce jobs for bulkload work with MapR-DB as well.

## MapReduce Setup

```
public static Job createJob(Configuration conf, String[] args) throws Exception {  
    . . .  
    Job job = new Job(conf, "mapreduce_import");  
    job.setMapperClass(ImportMapper.class);  
    . . .  
  
    // Insert into table directly using bulk import  
    HTable table = new HTable(jobConf, tableName);  
    HFileOutputFormat.configureIncrementalLoad(mrJob, table);  
    TableMapReduceUtil.initTableReducerJob(  
        tableName,  
        null, // reducer class is null  
        job);  
    return job;  
}
```

The code in this example is similar to what we saw in the MapReduce with HBase lesson.

Custom MapReduce jobs can use bulk loads with the `configureIncrementalLoad()` method from the `HFileOutputFormat` class as shown here highlighted in red.

In this example, the reducer class is null because there isn't actually a reducer step. Remember when Puts are being emitted from the mapper, you can skip the reducer step since the framework is already sorting the key-values from the mapper.

Next, we will look at the code for the mapper which will create a Put object and emit it, the `TableOutputFormat` class from the framework will take care of sending the Put to the target table.

## MapReduce Output Steps

```
@Override  
public void map(LongWritable offset, Text value, Context context) throws IOException {  
  
    String line = value.toString();  
    String stationID = line.substring(0, endIndex);  
    String something = line.substring(endIndex, yy);  
    String rowId = stationID + something;  
    byte[] bRowKey = Bytes.toBytes(rowId);  
    ImmutableBytesWritable rowKey =  
        new ImmutableBytesWritable(bRowKey);  
    Put put = new Put(rowKey);  
    for (int i = 1; i < someLimit ; i++) {  
        String column = "derive column value here";  
        String value = line.substring...;  
        put.add(cf, Bytes.toBytes(column), ts, Bytes.toBytes(value));  
    }  
    context.write(rowKey, put);  
}
```

Write put

This code is similar to what we covered earlier when we first looked at MapReduce, except the Put is being emitted from the mapper instead of the reducer.

Here we assume that the records are line oriented so extracting the data consists in calling substring on each line to retrieve elements that go into the row key, the column names, and values. The row key, column names, and values, constructed from the input, are used to create the Put object, then the Put object is emitted with the row key. The `TableOutputFormat` class from the framework will take care of sending the Put to the target table.

## Lab 12.3: Use a Custom MapReduce Program to Bulk Load Data



Open your lab guide to complete Lab 12.3: Use a custom MapReduce program that reads a text file, processes the data in map function and the uses Put to write to the HBase table.

## Learning Goals



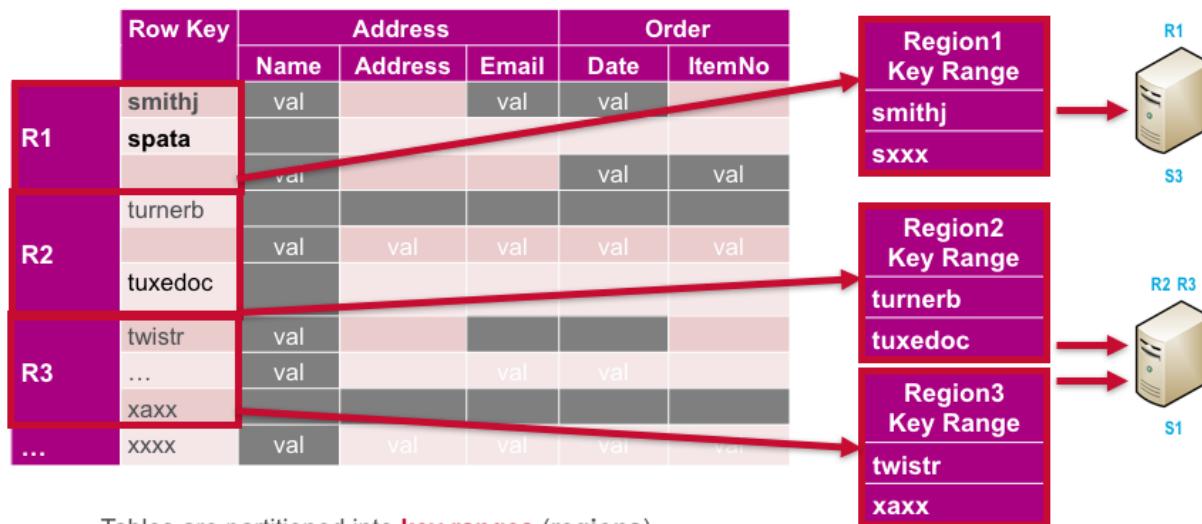
## Learning Goals



- 12.1 How Bulk Loading Data Works
- 12.2 Using the ImportTsv Bulk Load Tool
- 12.3 Use MapReduce Job to Import Data
- 12.4 Pre-splitting Table**

Let's discuss pre-splitting tables.

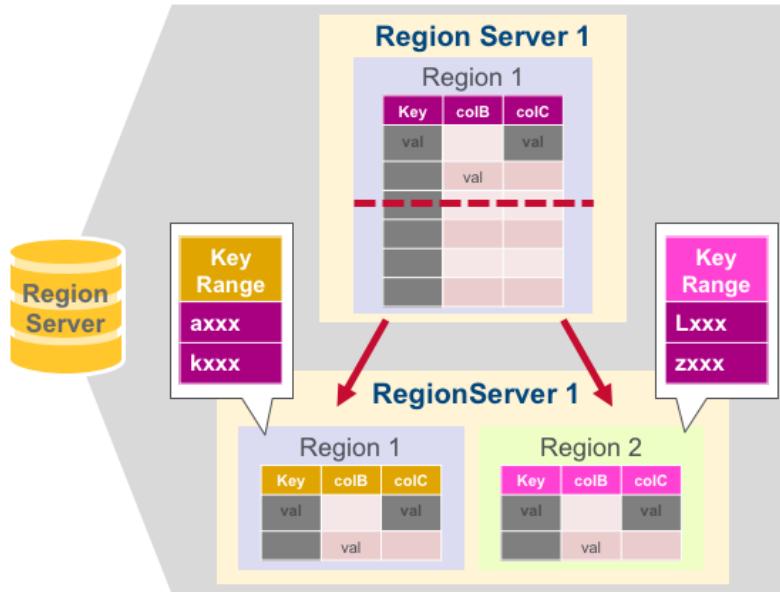
## Tables are Split into Regions = Contiguous Keys



Tables are partitioned into **key ranges** (regions)

As a reminder, HBase tables are split into sequences of rows, by key range, called regions.

## Region Split



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

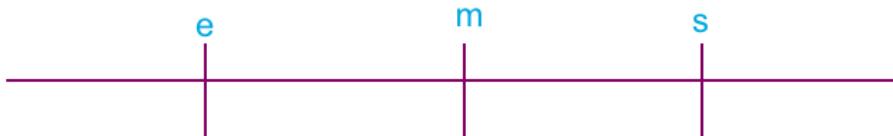
When a region grows too large, it splits into two child regions. It is also possible to pre-split a table by key range when you create it.

## Using the Shell to Split

```
> Create 'test_table', 'f1', SPLITS=>['e', 'm', 's']
```

OR

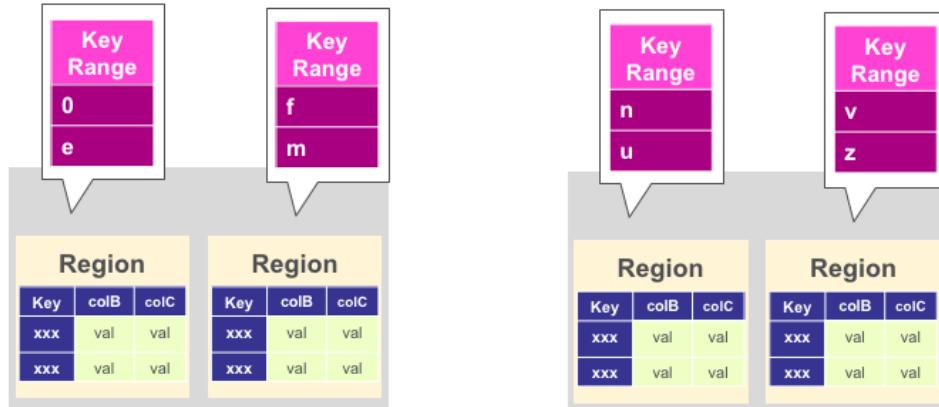
```
> Create 'test_table', 'f1', SPLITSFILE=>'tmp/splits'
```



When you create a table you can pre-split it. There are three ways you can do this, as discussed before you can use the Java HBase admin API, you also can use the MapR Control System, or the HBase shell. Here is an example of using the HBase shell to pre-split a table. You can specify the split keys in an array as shown or you can read the split keys from a file.

This creates and pre-splits a table with an initial set of empty regions defined by the specified split keys. The total number of regions created will be the number of split keys plus one. They form the start and end keys of the regions created. Here the split keys are e, m, and s, which creates four regions, with e as the end key for region one and u as the start key for region four.

## Example Splits



Here the split keys are e, m, and u, which creates four regions, with e as the end key for region one and u as the start key for region four.

## Summary

- HBase API and the java JDBC API to transfer data
- Used ImportTsv, a built-in HBase tool to import text data
- Saw how we can use a MapReduce task
- Overview of other options/clients like Thrift, Hive, Pig, Rest, and CLI
- Learned how to optimize by pre-splitting table before importing data

We have discussed several techniques to import data in bulk with an emphasis on importing from a SQL database.

- We looked at a simple client using the HBase API and the java JDBC API to transfer data from a SQL database to MapR-DB,
- We looked at using ImportTsv, a built-in HBase tool to import text data,
- We saw how we can use a MapReduce task and took a look at some other options/clients like Thrift, Hive, Pig, Rest, and CLI,
- Finally, we saw how to optimize by pre-splitting table before importing data into it.



## Next Steps

### DEV 340 – HBase Applications: Bulk Loading, Security, and Performance

Lesson 13: Performance

Congratulations, you have finished DEV 340 Lesson 12, Bulk Loading Data into MapR Tables. Continue to Lesson 13 to learn about Performance.



## **DEV 340 – HBase Applications: Bulk Loading, Security, and Performance**

Lesson 13: Performance

Winter 2017, v5.1

Welcome to DEV 340, HBase Applications: Bulk Loading, Security, and Performance, Lesson 13: Performance.

## Learning Goals



## Learning Goals



- 13.1 Data Access Patterns
- 13.2 Performance Considerations
- 13.3 Some Performance Tips
- 13.4 Sizes and Guidelines
- 13.5 Measure Performance
- 13.6 Performance Monitoring with MCS

When you have completed this lesson, you will be able to:

- Identify data access patterns
- Understand various performance considerations
- Learn some performance tips
- Recognize the importance of sizes and guidelines
- Measure performance
- And monitor performance with MCS

## Learning Goals



### 13.1 Data Access Patterns

- 13.2 Performance Considerations
- 13.3 Some Performance Tips
- 13.4 Sizes and Guidelines
- 13.5 Measure Performance
- 13.6 Performance Monitoring with MCS

First, let's look at data access patterns.

# Review



## Definitions

- Overall throughput (operations per second)
- Average latency (average time per operation)
- Low latency ~ fast response time
- High latency ~ slow response time
- Online systems
- Low-latency priority
- Batch
- Throughput priority

Before we talk about performance, here is a review of some terms we will be using:

Online systems have low-latency requirements, because there is a person waiting for a quick response.

The intent to be an online or an offline system influences many technology decisions when implementing an application. HBase is an online system. Its tight integration with Hadoop MapReduce makes it equally capable of offline access as well.

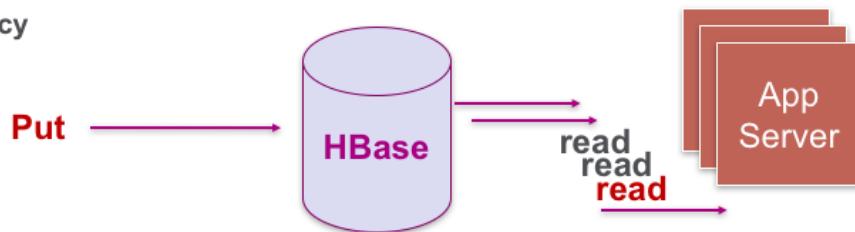
For online systems, low latency or a fast response time is the priority.

For batch systems, a high throughput is the priority.

## Data Access Patterns

Use Cases: Real Time Web Application Backend

- User profile, social media, web application storage
- Read heavy
- Writes
  - Random, few updates
  - Less throughput, low latency
- Reads
  - **Low latency**



When HBase is used as the backend store for a web application, the data access pattern is read heavy and the priority is fast, low latency reads.

## Data Access Patterns

For low latency reads:

- **Caching** is good for:
  - Slow changing, read-mostly data
  - Frequently read data
  - Gets or small range Scans
  - Set CF **in memory** for frequently read data
- Caching is not good for:
  - Table wide scans (LRU gets evicted)
  - **Row key** is critical for get and subset scans

Reading from memory is really fast. If the data is frequently read and not frequently updated then caching will give you faster reads. You can set a column family in memory property for frequently read data.

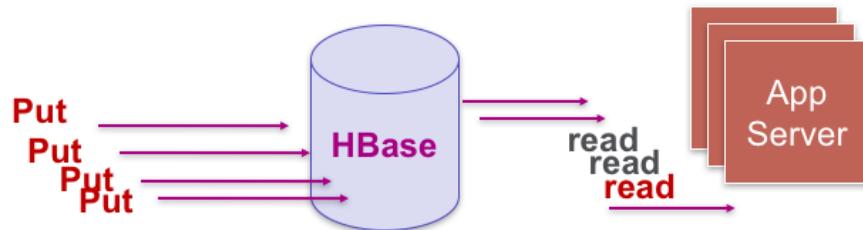
This is good for Gets and short Scans, however this will not be faster for tablewide Scans since the cache will become full. For fast Gets and short Scans the row key design is critical.

For nodes that have very large memory and with very high MapR-DB performance requirements, increasing memory cache allocation as high as 70% or more may be beneficial.

## Data Access Patterns

Use Cases: Real Time Web Application Backend

- Distributed Messaging
- Write/Read Heavy
- Writes
  - Low latency
- Reads
  - Low latency

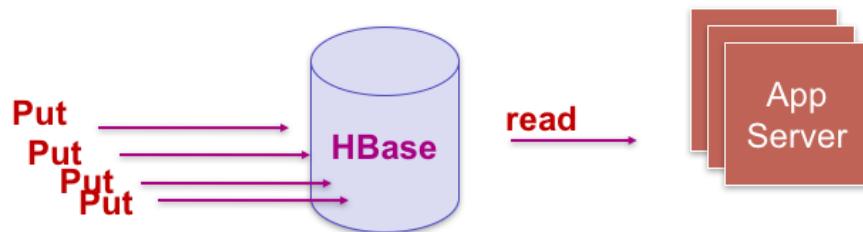


A distributed messaging application using HBase as the backend will be read and write heavy and the priority is low latency writes and reads.

## Data Access Patterns

Use Cases: Real Time Web Application Backend

- Machine Generated Data
- Write Heavy
- Writes
  - Low latency



For the use case of machine generated data or sensors being stored in HBase, the data access pattern will be write heavy low latency writes.

## Data Access Patterns

- For **random** writes
  - Low latency
- Make sure row keys **distribute writes** across the cluster
  - Balance work

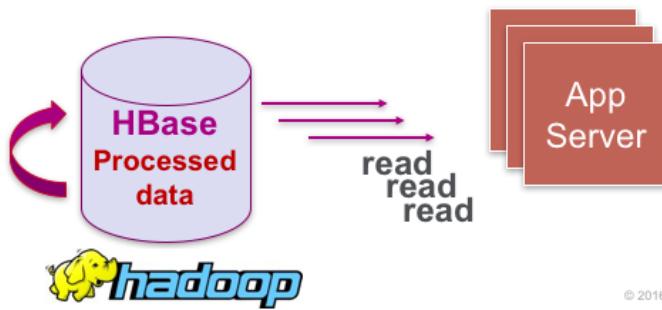
For fast writes it is important to distribute the writes across the cluster so that writes can happen in parallel to different region servers. The row key needs to be designed to not hot spot on one region server.

## Data Access Patterns

### Use Cases:

- **Materialized View, Pre-Calculated Summaries**
  - MapReduce to **update schema offline**
  - **Online Catalog, Online Dashboard**
- Many Writes (batch offline) Many Reads (**online**)
- Writes
  - High throughput (schedule offline)
- Reads
  - Low latency

**Bulk Import**



MAPR Academy

© 2016 MapR Technologies L13-12

To provide fast reads for online web sites or an online view of data for data analysis, MapReduce jobs can bulkload and/or reorganize the data into precomputed views or materialized views.

Remember, designing for reads means aggressively denormalizing data so that the data that is read together is stored together.

## Data Access Patterns

For Fast Reads:

- Data to be accessed together should be stored together
- Denormalize relationships, precompute, materialize view
  - Put all of the data for a Get in one row
  - Put data for Scans in contiguous rows
- Minimize number of disk seeks

For fast reads, the data that will be read together should be stored together. In order to group the data together you can:

- Denormalize or duplicate data
- Create precomputed materialized “views” in a table

For fast Gets put related data for a Get in a single row. For Scans put related data in contiguous rows. You need to think about the row key!

## Data Access Patterns

For Write Throughput:

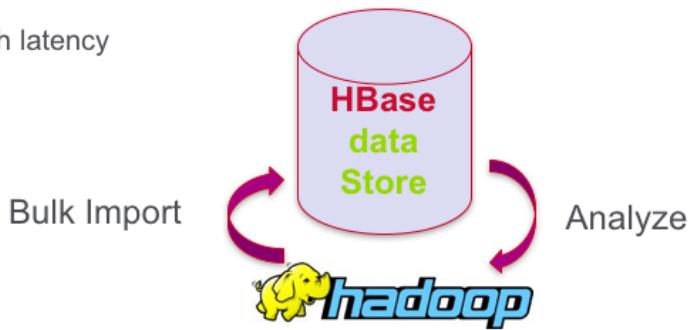
- Disable auto flush, or use batch Puts
- To write more rows/sec
  - **Tall skinny** = Consolidate data into fewer columns
  - Use compression
- Pre-split

For faster write throughput you should make sure to use batch Puts. Tall skinny tables will allow you to write more rows per second. Pre-splitting the table and using compression can also speed up throughput.

## Data Access Patterns

### Use Cases:

- Large scale **offline** ETL analytics, generating derived data
- Many writes
- Writes
  - Bulk import
  - High throughput, high latency
- Reads
  - High latency

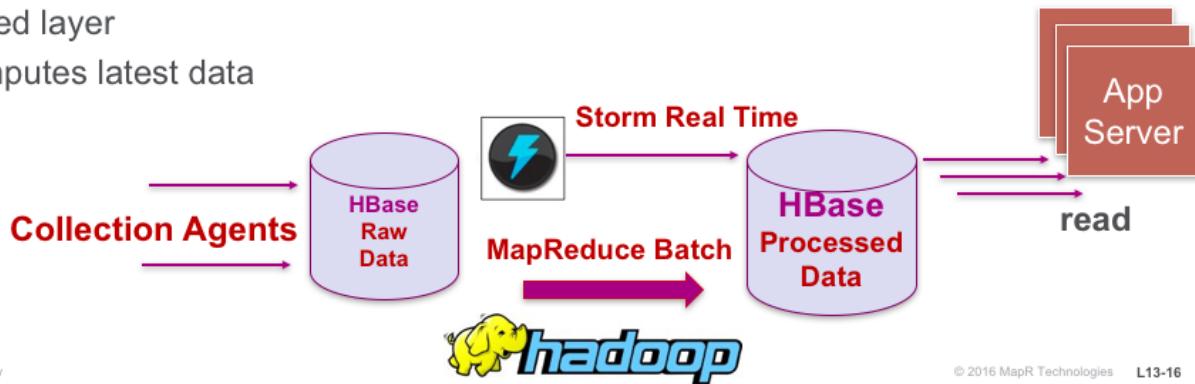


Offline HBase ETL data access patterns, such as MapReduce or Hive, are characterized by high latency reads and high throughput writes. Offline systems don't expect a response immediately.

## Data Access Patterns

### Lambda Architecture

- Serving layer
- Provides **precomputed view**, **low latency** reads
- Batch layer
- Precomputes data for serving layer
- Speed layer
- Computes latest data



Nathan Marz came up with the term Lambda Architecture (LA) for a generic, scalable, and fault-tolerant data processing architecture.

The Lambda Architecture has three layers: the batch layer, the serving layer, and the speed layer.

1. The batch layer precomputes data for the serving layer.
2. The serving layer provides a precomputed view for fast reads. The serving layer updates whenever the batch layer finishes precomputing a batch view.
3. The speed layer only produces views on recent data, the speed layer is for functions computed on data in the few hours not covered by the batch. The speed layer updates the real time view as it receives new data.

More information on this can be found online.  
<http://lambda-architecture.net/>

## Learning Goals



## Learning Goals



13.1 Data Access Patterns

### **13.2 Performance Considerations**

13.3 Some Performance Tips

13.4 Sizes and Guidelines

13.5 Measure Performance

13.6 Performance Monitoring with MCS

Now, let's discuss some performance considerations.

## Performance Strategies

- HBase and MapR tables are **lower-level** than relational tables
- This makes **design different** and more flexible
- Also allows **scaling** beyond relational limits
- **More planning for data access patterns is needed**
- Three **significant design** areas:
  - **Row key design**
    - **Find** and read data efficiently
  - **Column family design**
    - **Read selectively** from disk
  - **In-memory** column family
    - Super-**fast** reads

We have seen that with HBase you have to think about how your data will be read up front. You have to design your schema based on your access patterns, you can not rely on queries with secondary indexes and joins like with a relational database. This means more planning for data access patterns is needed, but on the other hand, HBase can scale across a cluster, unlike a relational database.

When you are designing your HBase application for performance there are three significant areas:

1. The row key design is critical in order to find and read data efficiently.
2. The column family design allows you to group the columns together that most typically will be read together and separate those which will not.
3. In-memory column families allow for super fast reads for data that is frequently read and not too big to cache.

## Main Performance Guidelines

Schema design

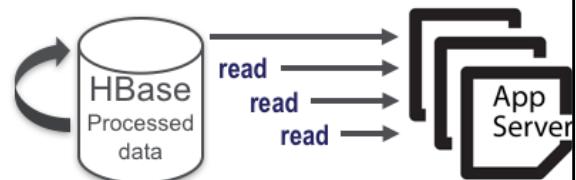
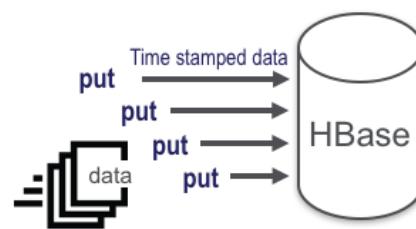
Think about your use cases

- Mostly writes, updates, get, scans?

- Capturing Incremental Data - Time Series Data  
High Volume, Velocity **Writes**

- Information Exchange, Messaging  
High Volume, Velocity **Write/Read**

- Content Serving, Web Application Backend  
High Volume, Velocity **Reads**

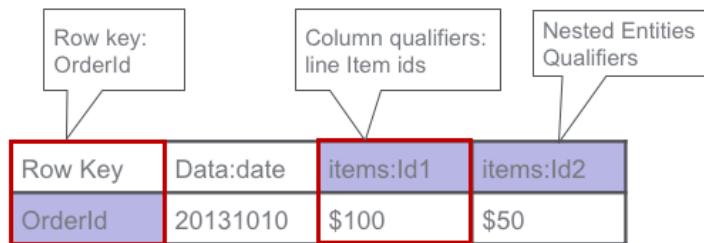


Your schema design will impact performance directly when you read and write data. You should think about your use cases and data access patterns compared to the ones we just discussed. Are your priorities low latency reads or writes or high throughput?

## Main Performance Guidelines

### Schema design

- Think about your questions
  - Get Nested Entities -> wide table



For fast Gets, it's possible to model one-to-many relationships in HBase as a single row. In this example, the order and related line items are stored together and can be read together with a single get on the row key.

## Main Performance Guidelines

### Schema design

- Think about your questions
  - Scan fine grained composite key -> tall table

Row key format: Metric | + | Time | + | Tags

All samples for same metric grouped together

Queries focus on one or few metrics

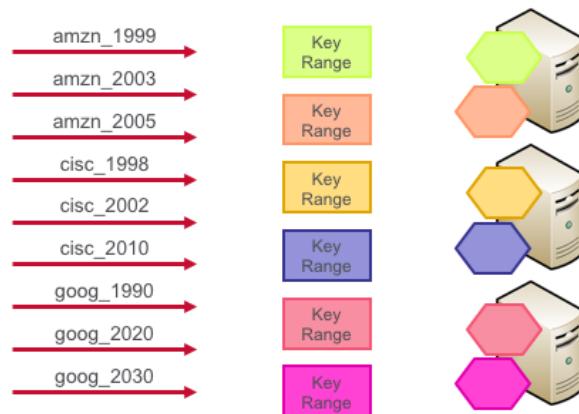
Metric	Time	Tags	Value
http.hits	10667	host=n1	3400
http.hits	10668	host=n2	3000
http.hits	10727	host=n1	2000
http.resptime	10667	host=n1	10
http.resptime	10668	host=n2	5
http.resptime	10727	host=n1	15

Group desired data together – put most queried information on leftmost part of key

For fine grained scanning you can put a lot of information in the row key in a tall table, like in the OpenTSDB example shown here, which we discussed in the schema design lecture.

## Main Performance Guidelines

- Schema design: Design row key for usage!
  - Distribution of writes



You should make sure your row key distributes writes, to avoid hot spotting.

## Main Performance Guidelines

### Schema design

- Design row key for usage!
  - Group data to minimize number of seeks
  - Make sure **key design** actually works!

SYMBOL	+	Reverse timestamp				
Row Key						
AMZN_98618600666						
AMZN_98618600777						
GOOG_98618608888						

Make sure your key groups related data together for scanning in order to minimize seeks. You may have to test to make sure that your key design actually works with usage.

## Performance Strategies: Row Keys

- Row keys determine **data locality**
- Row key significant for read performance
- Unique and useful meaning

Sorted by row key and Column

Row 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

Row keys determine data locality since HBase data is stored in key-value format in files sorted by the key. The row key is very significant for read performance, since reading data that is stored together is fast. Your row key must be unique and should have useful meaning for finding data.

## Performance Strategies: Row Keys

- Row keys determine **data locality**
- Examples where adjacency is **good**:
- **Related data** together
  - Get page Likes for partial key “com.mapr”

Row Key	stats:likes			
com.mapr.doc/hbase	100			
com.mapr.blog/carol	10000			
org.redcross.blog/donate	250			

Row keys determine data locality, grouping related data together is super fast for data that should be read together. Examples of this are when you want to scan by a partial key of related web URLs like com.mapr to read the counter of likes per page.

## Performance Strategies: Row Keys

- Row keys determine **data locality**
- Examples where adjacency is **good**:
- Sorting conveys **access pattern**, **sorted retrieval**
  - **Most recent stocks** sold by company, GOOG\_reversetimestamp

SYMBOL	+	Reverse timestamp				
Row Key						
AMZN_98618600666						
AMZN_98618600777						
GOOG_98618608888						

Grouping and ordering is also good when you want a sorted retrieval, for example by stock symbol and **most recent transaction time**.

## Performance Strategies: Row Keys

- Row keys determine **data locality**
- Examples where adjacency is **good**:
- Geographic **location related** questions:
  - Get restaurants, hotels by location, fast when **stored together**

Row Key				
NYC_HOTEL_PLAZA				
NYC_RESTAURANT_CAPITAL_GRILLE				

Grouping is also good for location related questions like what restaurants or hotels are near this city.

## Performance Strategies: Row Keys

- Adjacency is **good** for data commonly **read together**:
- Good for **caching** (**Least Recently Used** thrown out)
- Composite key
  - Good for **range scans**
- Adjacency is **bad** when **use of cluster is not spread out**:
- If *number of Gets/Puts* targets a **narrow range** of keyspace
  - Not using the full cluster
- More entries for certain keys, for example messaging by UserId

Grouping related data that is frequently read together is super fast for caching, reading from memory is always faster than from disk. In-memory column families will remain in memory with the Least Recently Used being taken out when the cache is full. Composite keys are good for fast range scans because full table scans will cause the memory to get full and will not benefit from caching.

Grouping data together is not good when your data is not well distributed across your cluster, for example when the number of Gets/Puts you are observing are not spread out across region servers. An example of this is for Facebook's messaging system, where Zuckerberg got way more messages than the average user.

## Learning Goals



## Learning Goals



- 13.1 Data Access Patterns
- 13.2 Performance Considerations
- 13.3 Some Performance Tips**
- 13.4 Sizes and Guidelines
- 13.5 Measure Performance
- 13.6 Performance Monitoring with MCS

Let's discuss some general performance tips.

## Performance Tips: RPC Data



Network round-trips are expensive

Each single operation means a trip from the client to the server and back.



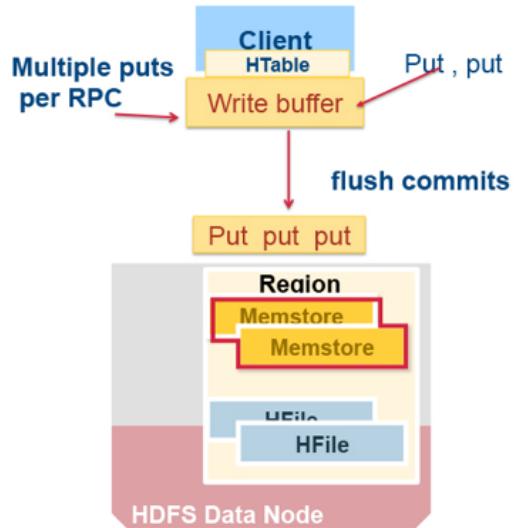
For data throughput it is better to have **fewer RPC** calls with **more data** compared to **lots of RPC** calls with **small amounts of data**

When updating small records, you will get better performance by grouping puts into fewer RPCs.

Whether you use a single Put or a list of Put operations, each call will be sent one at a time because autoflush is on by default. Disabling autoflush will prevent having each call be sent separately to the server. By turning it off, the Put operations will be sent when the write buffer is full instead.

## Performance Tips: RPC Buffer

- Enable client-side write buffer:
  - myTable.setAutoFlush(false)
  - Collects put operations into write buffer
- Flush data from put operations:
  - void flushCommits()
  - Sends buffered updates to remote server



If you are making a lot of Put calls then disabling autoflush will prevent having each call be sent separately to the server, because autoflush is on by default. By turning it off, with `HTable.setAutoFlush(false)` the Put operations will be sent when the write buffer is full instead. You can either explicitly call `flushCommits()` to flush messages or `flushCommits()` will be called implicitly when you close the table.

## Performance Tips: HTable Batch Operations

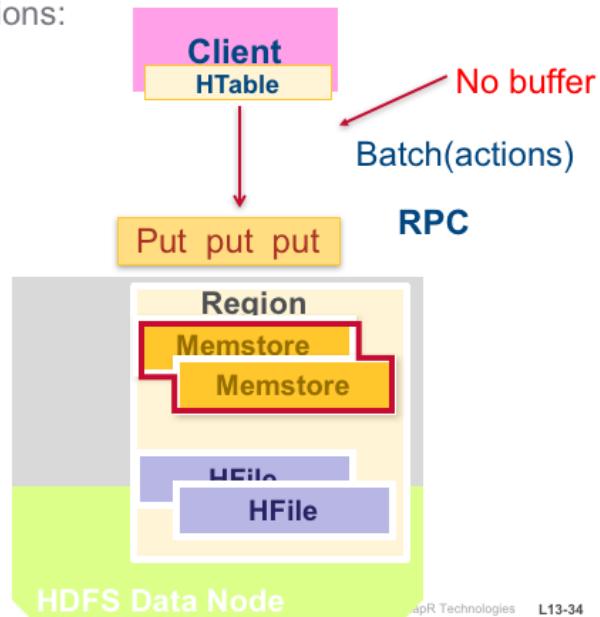
HBase API provides two ways to **batch** operations:

1. HTable put(), get(), delete() list methods:

– void put(List<Put> puts)

2. Batch methods from HTable

void batch(List<Row> actions,  
Object[] results)



As discussed before Network roundtrips are expensive. Another way to get better performance with fewer RPCs is to use the HTable Put List or Batch methods, shown here. These methods let you pass a list of objects for updating or reading, which will be sent in one RPC call.

## Performance Tips: Java API

- Limit what columns are selected in a Scan
- Scan.addFamily() will add all columns so consider using addColumn(family, qualifier) instead
- Use filters
- If you only want the row key
  - Use FilterList with KeyOnlyFilter and FirstKeyOnlyFilter

Here are some optimizations you should consider when using the API.

Remember that with a Get or Scan you should narrow down the data returned, otherwise all of the row will be returned.

If you are only interested in some of the columns in a column family, you will get better performance by specifying the columns with addColumn(), this will transfer less data to the client.

## Performance Tips: Filtering

Filters are applied on the server side

- Delegate the selection of rows to the servers
- **Reduce Network Traffic**
- Minimize data returned to client
- Combine filters
- With AND, OR into filter list.
- 20 filters, including:
  - “RowFilter”: filter on row key
  - “QualifierFilter”: filter on qualifier
  - “ValueFilter”: filter on values
  - + Other filters

As a review, you can use filters to reduce the amount of data transferred to the client. On the server side the filter is used to determine whether a record should be returned back to the client side.

## Performance Tips: Java Object Allocation

- Use constants
- Object allocation for strings, converting bytes for reading/writing can make a huge difference

```
public static final byte[] CF = cf.getBytes();
public static final byte[] COL = qual.getBytes();
byte[] b = r.getValue(CF, COL);
```

Reducing object allocation for converting bytes for reading/writing to HBase can make a huge difference in performance. You can reduce object allocation by setting up such variables once with “public static final” instead of calling Bytes.toBytes every time you need to specify a table component. This saves resources and is more efficient.

## Performance Tips: Close the ResultScanner

```
Scan scan = new Scan();
// set attributes...
ResultScanner rs = table.getScanner(scan);
try {
    for (Result r = rs.next(); r != null; r = rs.next()) {
        // process result...
    }
} catch (Exception e) {
...
} finally {
    rs.close(); // Always close the ResultScanner!
}
table.close();
```

A ResultScanner object is returned from a scan operation. Don't forget to call `ResultScanner.close()` on a Scan operation. The server can then dispose of objects no longer needed.

You should close the ResultScanner to release resources which are no longer needed. This affects things on the server more than on the client side and you should do this so that you don't create performance issues on the server side.

To avoid performance problems always have ResultScanner processing enclosed in the finally of a try/catch blocks, this means it will always be called, if an exception is thrown or not.

## Performance Tips: Column Families

You can set column family attributes using MCS  
Compression, MaxVersions, Time-to-live, In-memory

Column Family Name	Max Versions	Min Versions	Compression	Time-to-live	In Memory
cf1	3	1	zlib	1h 40m	✓
cf2	3	2	lzf	1w 6.9d	✓
cf3	4	3	lzf	2w 3.4d	
cf4	2	1	lz4	4mo 4.1w	✓

You can use the MapR control system, the HBase shell, or the Java API to set properties for column families.

You can set the Max Min versions, compression type, time-to-live, and the in-memory setting, whether the CF is kept in memory or not.

### Time-to-live TTL:

Specified in seconds and is by default set to Integer.MAX\_VALUE or 2,147,483,647 seconds. That's 2 billion 147 million etc.  $2^{31}$ .

If a value exceeds its TTL it is dropped and this is based on the timestamp. This is in addition to the number of versions that get kept.

### In-memory

Defaults to false.

Setting it to true is not a guarantee that values are loaded into memory nor that they stay there. It's more like an elevated priority. This setting is useful for small column families with few values, like passwords in a user table.

## Performance Tips: Compression Guidelines

<http://doc.mapr.com/display/MapR/Compression>

MapR supports three different compression algorithms:

- lz4 (default) Fast but less compression
- lzf
- Zlib – Slow but max compression

Compression Type	Compression Ratio	Compression Speed	Decompression Speed
lz4	2.084	330 MB/s	915 MB/s
lzf	2.076	197 MB/s	465 MB/s
zlib	3.095	14 MB/s	210 MB/s

You can set the compression type on the column family, lz4, the default, is faster but gives less compression, Zlib is slower but gives max compression. There is a tradeoff between compression ratio and speed. Compressed data uses disk space and less bandwidth on the network than uncompressed data, however for scanning files it is slower for reads.

## Learning Goals



## Learning Goals



- 13.1 Data Access Patterns
- 13.2 Performance Considerations
- 13.3 Some Performance Tips
- 13.4 Sizes and Guidelines**
- 13.5 Measure Performance
- 13.6 Performance Monitoring with MCS

## Sizes: Minimize Row Key, CF, Col Names

Cell value coordinates are repeated

- Remember a result, wraps a row and 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}
```

- Minimize row key, CF, column name sizes
- Keys can be up to **64K**, but keep short as possible, to still be useful
- Use one character for CF, two-three for column qualifiers
- Cost increase for adding columns:
- For write throughput:
  - Faster with **fewer columns** and big values vs. lots of columns small values

As a cell value passes through the system, it'll be accompanied by its row, column name, and timestamp. They take up space in the RPC calls, in-memory, indexes, and on disk.

ColumnFamilies, column qualifiers, and row keys could be repeated several billion times.

- You should keep row keys as short as is reasonable such that they can still be useful for required data access.
- Keep the ColumnFamily names as small as possible, preferably one character, for example, "d" for data.
- Keep the column names as small as possible, preferably one-three characters, abbreviate!
- You can use a lookup table to keep names short and meaningful, like in the openTSDB example.

For write throughput you can put more data with fewer columns verses lots of columns with small values. Note, MapR-DB does not store the full coordinates for each cell on disk like HBase, but size still matters. MapR-DB does not repeat the key or the CF details for anything, while HBase does. So, the number of bytes transferred from disk to find a subset of columns in a row is far less compared to HBase. Consequently, the in-memory footprint in MapR-DB is much tinier, leading to more efficient use of the memory. MapR-DB does store the column qualifier as part of data.

## Sizes: Regions

Row Key		CF1			CF2	
		colA	colB	colC	colA	colB
Region1	axxx	val			val	val
	gxxx	val			val	val
Region2	hxxx	val	val	val	val	val
	jxxx	val				
Region3	kxxx	val		val	val	
	rxxx	val	val	val	val	val
...	sxxx	val				



Nodes

A region ~1-7 GB

A container 16-32GB, Avg ~25GB

- More than one region may fit in container

*Container is a storage unit*

A container is 32 GB when full. MapR-DB regions are 4 to 7 GB, more than one region may fit in a container. The average size of a container is 25 GB, assuming it's only partially full.

## Sizes: CF, Rows

- Max limit for MapR-DB Column Families is 64 (HBase is three)
- You can have billions of rows
- MapR-DB tables support row size up to 2GB
  - Recommended size is <=100mb due to memory and other resource constraints
  - Larger the row, slower the overall performance
    - Optimal < 50-100KB, good < 1MB

The limit for the number of Column Families for a MapR-DB Table is 64 , with HBase 3 is the recommendation. You can have billions of rows.

The row size limit is enforced at the RPC level (per Put/Get).  
The Sweet Spot for row size is between 100 bytes to 50k

The Default max row size is 16MB. Marshalling/unmarshalling giant objects is not a good idea, otherwise Performance will drop dramatically .

MapR-DB comes with a read/write file-system built-in. For larger rows, you can put the data into a file and put the file-name into the row. Unlike HBASE with HDFS , MapR-DB has no limit on the number of files, so create as many as you like.

## MapR-DB Large Row Support

- MapR-DB supports row size up to 2 GB.
- Recommended size is <=100 MB.
- Larger the row, slower the overall performance
- Configure maximum row size: mfs.db.max.rowsize.kb set to 100MB row size:
- Maprcli config save -values {"mfs.db.max.rowsize.kb" :"102400"}
- Note: Cell versions count, two “versions” of 100MB = 200MB...
- To view the current setting:
- Maprcli config load -json | grep mfs.db.max.rowsize.kb
- If row size is x then it is recommended to have MFS started with 10x memory.

MapR tables support rows up to 2 GB in size. Rows in excess of 100MB may show decreased performance. You can configure this maximum by changing the value of the `mfs.db.max.rowsize.kb` value.

For more information, please reference the online documentation.

<http://doc.mapr.com/display/MapR/Setting+Up+MapR-FS+to+Use+Tables#SettingUpMapR-FS+to+UseTables-ConfiguringMaximumRowSizesforMapRTables>

## RAM

- Caching is good for data that is frequently read
- Can increase memory settings in warden.conf
- Controlled by three parameters in the warden.conf file:
  - Service.command.mfs.heapsize.percent –
    - The target percent of the **MFS Cache**
    - 35% -70%
  - Service.command.mfs.heapsize.maxpercent –
    - An **upper bound** on the target
  - Service.command.mfs.heapsize.min –
    - A **lower bound** on the target

You can increase memory settings in `warden.conf`. This is controlled by the three parameters in the `warden.conf` file shown here.

The target percent for MFS Cache to use is typically 35%, but for MapR-DB table specific usages, especially on nodes that have very large memory and with very high performance requirements, increasing memory cache allocation as high as 70% or more may be beneficial.

## LRU Cache

All caches in MFS use the Least-Recently-Used (LRU) caching algorithm.

LRU policies keep key data structures in-memory

Data accesses require only one disk seek

MapR uses very efficient caching techniques for all of its file system and MapR-DB data:

- There is no duplicate caching like in HBase - data is cached in the filesystem, and shared with MapR-DB.
- Paging of cached data happens using a Least Recently Used being evicted. Key data structures such as indexes are kept in-memory.
- Using these key data structure in-memory means out-of-memory data accesses require only one disk seek.
- Since all reads happen from the container master node, the replicas do minimal caching of MapR-DB data.

## Concurrency Monitor Data Spread

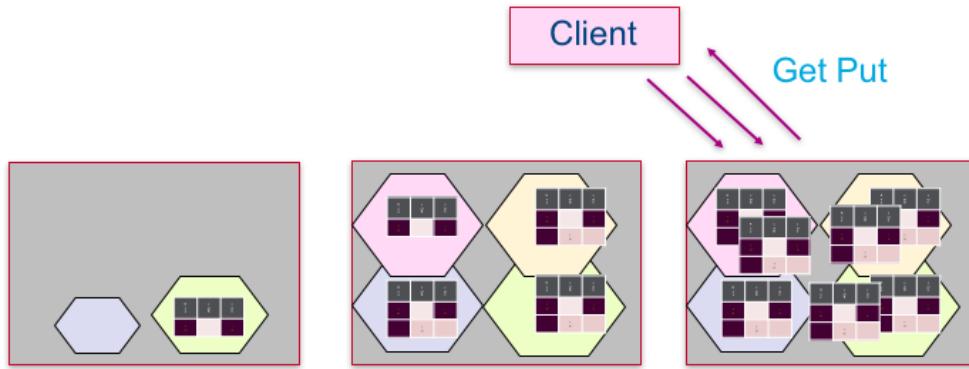
- For concurrency:
- Monitor data spread of table regions
- If too few regions then the data could be served by too few nodes
- Split to spread

You should monitor that your data is spread out over regions for good concurrency. If you don't have enough regions, you should split your table to spread out your data for better concurrency.

## Hot Spots

Subset of MapR-DB servers get disproportional numbers of Get/Put requests compared with the cluster size.

- Example 2 out of 10 get 90% of the Get/Put requests



As discussed in the schema design lecture, when a subset of MapR-DB servers get disproportional numbers of Get/Put requests compared with the cluster size, it is called “hot-spotting.” Next, we will look at causes of hot-spotting.

## Hot Spots

Possible causes:

- Table is too **small**, did **not** grow enough to **split** across all the servers
- Server was **added later** in the process
- Table is **not growing**, so **not splitting** to **new server**
- **Unbalanced** distribution of regions and Get/Put requests
- The row key was not well designed:
  - Sequential Id like a timestamp
  - **UserId** for social application, messaging, where one user is a lot **more active** than others

Some possible causes of hot-spotting are:

1. The HBase table is too small and did not grow enough to split across all the servers. This can happen when a server was added later to the cluster after the table has data in it. If the table is not growing, the table will not be splitting regions to the new server: The result will be an unbalanced distribution of regions and requests to regions.
2. Another cause of hot-spotting can be when the the row key was not well designed:
  - In the schema design lecture we discussed how a row key with a sequential Id like a timestamp will cause the rows to be inserted in sequential order, which will result in hot-spotting.
  - Also a row key with a UserId for a social application like messaging, where one user is a lot more active than others could cause an unequal distribution on the region server for very active users, which is what Facebook messaging experienced.

## Hot Spots

Possible causes:

- The client was not designed well:
  - Client processes large **batches** of **key ordered** requests:
  - Client **requests** **do not have a good spread** across the entire table, creates hot zones.

Another possible cause is when the client was not designed well, for example when a client processes large batches of key ordered requests and the client does not have a good spread across the entire table. This can create hot zones, since the batch is going to the same key range, and therefore region.

## How to Identify Hot Zones

- Check the **number** of puts/gets per second **per node**
- Should be similar across nodes on cluster
- What is the **pattern** of Gets and Puts?
- Should be distributed
- Check the **number of regions** per node
- It should be similar on all nodes
- If adding **new nodes**
- **Table data needs to grow** before it can spread out automatically
- Think about the **key design**
- Will it **spread keys** across regions?

Here is how you can identify hot zones:

1. Check the number of Puts and Gets per second per node. This should be similar across nodes on your MapR-DB cluster.
2. Look at the pattern of Gets and Puts, this should be distributed.
3. Check the number of regions per node, this should be similar on all nodes.
4. If you are adding new nodes, the existing table's data needs to grow before it can spread out automatically.
5. Think about the key design. Will it spread keys across regions? Review the schema design lecture if you are having problems.

## DB Gets, Puts, Scans on Node

The screenshot shows the MCS interface for a specific node. At the top, it displays the node's IP and physical topology information: Physical Topology: /newyork/rack00 and Physical IP: 1.2.5.1, 10.10.10.1. It also shows the last heartbeat and reboot times.

**Machine Performance:**

- Memory Used (M): 38% of 98.3KB
- Disk Used (GB): 54% of 8.2GB in use for 3 disk(s)
- CPU: # CPU 8, % CPU used 48
- Network I/O: In (per sec) 130.5KB, Out (per sec) 6MB
- RPC I/O: Count (per sec) 15, In (per sec) 9.4MB, Out (per sec) 309.3KB
- Disk I/O: Reads (per sec) 9.4MB, Writes (per sec) 763KB

**Map Reduce:**

TaskTracker Slots	Used	Total
Map Slots	6	12
Reduce Slots	1	15

**DB Gets, Puts, Scans:**

	Gets	Puts	Scans
10s	92305	63292	64747
1m	685552	465767	982700
5m	465902	430158	482473
15m	8942039	508467	460070

MAPR Academy © 2016 MapR Technologies L13-54

You can use the MCS Machine Performance pane to display the number of Get, Put, and Scan operations performed during various time intervals to check for hot zones.

## Region Size

- Small regions
- Greater **spread** across nodes
- Less work for MapReduce task
- Large regions
- Better **buffering** at client
- **Less metadata** to cache at client (CLDB key ranges)

There are tradeoffs with region size:

- Smaller regions will give you a better spread across nodes which can be better for parallel processing across nodes.
- Larger regions will give you less key range location meta data to cache on the client side.

Next, we will look at how you can manually tune your region size, which you should not have to do if you have a good row key design and application architecture from the beginning, but you may have to do otherwise.

## Manually Tuning Region Size

- To enable/disable autosplit
  - maprcli table edit -autosplit true/false -path tablePath
- To set the default region Size for a table
  - maprcli table edit -regionSizeMB regionSize -path tablePath
- To manually split a region
  - maprcli table region split -path tablePath -fid regionFid
- To manually merge regions
  - maprcli table region merge -path tablePath -fid regionFid

These are the commands which you can use to manually tune region size. See the MapR documentation online for more details.

<http://doc.mapr.com/display/MapR/table+region>

## Learning Goals



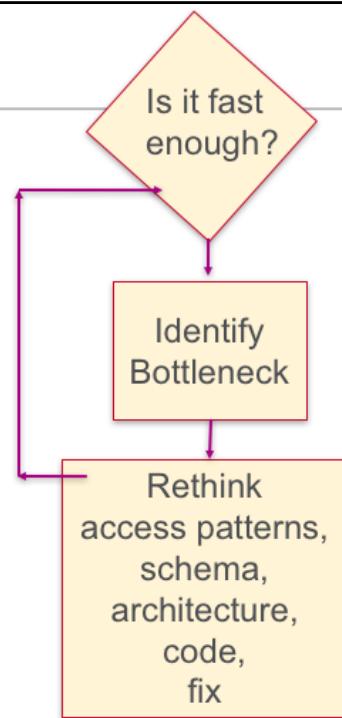
## Learning Goals



- 13.1 Data Access Patterns
- 13.2 Performance Considerations
- 13.3 Some Performance Tips
- 13.4 Sizes and Guidelines
- 13.5 Measure Performance**
- 13.6 Performance Monitoring with MCS

## Tuning Application Performance Guidelines

- Understand your **use cases, read, write heavy...**
- What are Service Level Agreements? online, offline, batch?
- Measure performance:
- Benchmark test:
  - Simulate **use cases – Access Patterns**
  - Random access, read/write
  - Bulk loading, bulk processing
  - Scan
- **Load** test:
  - **Normal** load, **peak** load
- **Stress** test:
  - Harder than normal conditions, what happens?
- Longevity:
  - **Sustained** load



We have been talking about tuning performance and we looked at some specifics. But just like any application there are some general guidelines you should keep in mind and that you should follow when you are looking into performance issues.

First, you should fully understand the architecture of the system and that includes all of its components. As mentioned earlier, hardware and network may have an impact on performance. If you are running into performance issues then first work on stabilizing the system, which means fix critical bugs before you tackle performance issues. Once you have a stable system, then you can measure the current performance. This will provide you with numbers that make up a baseline.

You should set performance targets so you know what your goals are and how far or close you are to reaching them. With the numbers from the baseline you will be able to determine how close to the targets you are.

Next, identify bottlenecks: use logging, profilers, monitoring tools like MCS. Fix the root cause of bottlenecks.

This is an iterative process so integrate performance at the beginning of the development effort if possible and keep an eye on it regularly. A way to do this is to build integration tests that record performance for known operations or transactions.

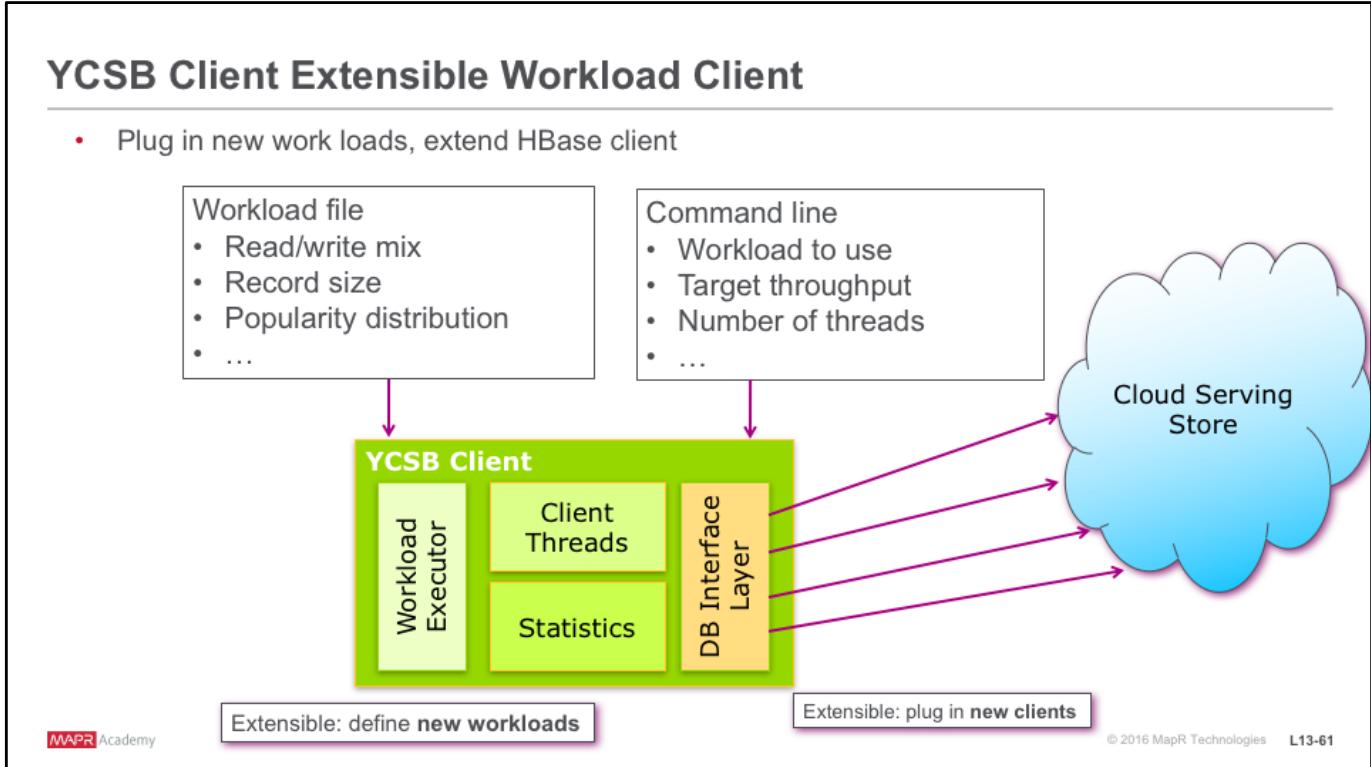
A lot of these guidelines may seem obvious, but often times development teams fall into some common traps. Let's look at these next.

## Benchmarking Tools YCSB

- YCSB (Yahoo Cloud Serving Benchmark)
- Framework to evaluate the performance of different data stores
- <https://github.com/brianfrankcooper/YCSB/>

A popular tool for measuring performance is YCSB. We are going to run it in the lab as well.

Yahoo! funded research to come up with a standard performance-testing tool that could be used to compare different databases. The company called it Yahoo! Cloud Serving Benchmark (YCSB). YCSB is the closest there is to a standard benchmarking tool that can be used to measure and compare the performance of different distributed databases. Although YCSB is built for comparing systems, you can use it to test the performance of any of the databases it supports.



The goal of the YCSB project is to develop a framework and common set of workloads for evaluating the performance of different “key-value” and “cloud” serving stores.

YCSB consists of the YCSB client, which is an extensible workload generator, and the core workloads, which are a set of workloads that comes prepackaged and can be generated by the YCSB client.

The Client is extensible so that you can define new and different workloads to examine system aspects, or application scenarios, not adequately covered by the core workload.

## Benchmarking Tools YCSB

- Two steps to run a test:
- Run the YCSB client **extensible workload generator**
  - **Loads** a workload
    - Comes with **prepackaged** set of workload scenarios
- Run the YCSB client to **execute** the workload generated

There are two steps to run a YCSB test:

You first run the client to generate some data, usually by using a prepackaged workload, and then you run the client to execute the workload you just generated.

## YCSB workloads

Read/write workload, read/update ratio: **50/50**

Example: **Session store** recording recent actions

Read mostly workload. Read/update ratio: **95/5**

Example: **photo tagging**; add a tag is an update, **but most operations are to read tags**

Read only: 100/0

Example: **user profile cache**, pre-materialized from Hadoop

Read latest Read/update/insert ratio: **95/0/5**

Example: **user status updates**; people want to **read the latest**

Read-modify-write workload: 50/50

User database, record **user activity**.

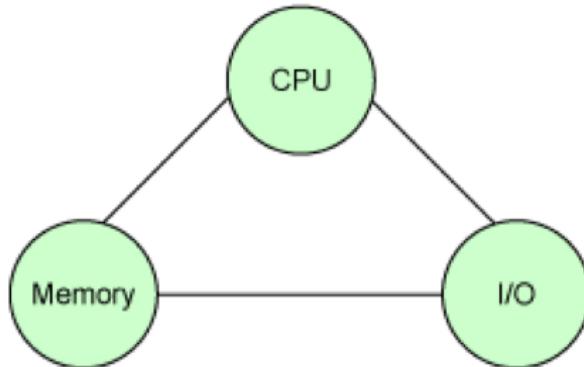
Scan/insert ratio: 95/5

Scan for the posts in **users conversation** thread

This shows the different prepackaged workloads that you can run with YCSB. As you see you can test for various workloads from read only, to read/write 50/50.

## What Performance Benchmarking Measures

While benchmarking you monitor:  
Usage for CPU, RAM, disk I/O, and network



While benchmarking you monitor usage for CPU, RAM, disk I/O, and network.

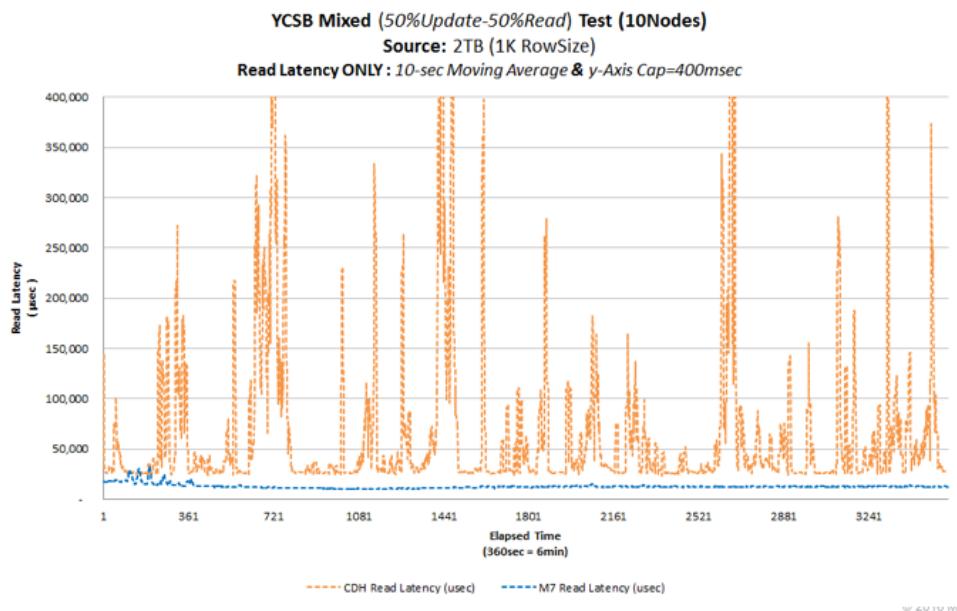
## High Throughput Across Varying Workloads

YCSB Benchmark (ops/sec/node)	MapR 3.0.1 (MapR-DB)	CDH 4.3.x (HBase)	MapR-DB Advantage
Put	30,478	10,275	3.0x
Read	5,934	865	6.9x
Scan (50 rows)	838	274	3.1x
95% read, 5% update	3,850	1,541	2.5x
50% read, 50% update	7,965	2,918	2.7x

Hardware Configuration	CPU : Intel® Xeon® CPU E5645 2.40GHz 12 cores x2 RAM : 48 GB Data Disk : 12x 3TB (7200 rpm) Size – record size = 1k, data size = 2TB OS : CentOS Release 6.2 (Final)	10-Node Cluster
------------------------	--	-----------------

This table shows a YCSB benchmark comparing MapR-DB to Cloudera HBase. The last column show the observed MapR-DB advantage in performance.

## Consistent, Low Read Latency



MAPR Academy

www.mapr.com Technologies

L13-66

This shows read latency over time, a smaller read latency means faster response time, so smaller is better. MapR-DB is shown in blue and Cloudera HBase in orange. The spikes for Cloudera HBase are because of the impact of garbage collection and compactions on performance. During the spikes the region server is experiencing garbage collection and compaction causing I/O storms.

## Learning Goals



## Learning Goals



- 13.1 Data Access Patterns
- 13.2 Performance Considerations
- 13.3 Some Performance Tips
- 13.4 Sizes and Guidelines
- 13.5 Measure Performance
- 13.6 Performance Monitoring with MCS**

## Monitoring and Management - Categorization

What is my...?

- Average job comp. time
- Average active users
- Reason for job slow/fail
  - Task details

Infrastructure

Alerts

Application

Data

What is my...?

- Node utilization (CPU, RAM, Disk)
- Busiest nodes
- MapReduce status
- History of --^

What is my...?

- Quota consumption
- Max user consumption
- Snapshot/mirror status

Here are some typical questions an admin would ask when monitoring and managing a MapR cluster. We will glance over monitoring with MCS next, for more information you should take the MapR Administrator training.

## DB Gets, Puts, Scans on Node

The screenshot shows the MapR Node Monitor interface for a node named **ip-10-160-119-224.us-west-1.compute.internal**. The top bar displays the node's name, physical topology, physical IP, and file server heartbeat status. Below the header, there are three main sections: **Machine Performance**, **Map Reduce**, and **DB Gets, Puts, Scans**.

- Machine Performance:** Displays memory usage (38% of 98.3KB), disk usage (54% of 8.2GB for 3 disk(s)), CPU usage (8 cores, 48% used), Network I/O (130.5KB in, 6MB out), RPC I/O (15 calls, 9.4MB in, 309.3KB out), and Disk I/O (9.4MB reads, 763KB writes).
- Map Reduce:** Shows TaskTracker Slots (Map Slots: 6/12, Reduce Slots: 1/15).
- DB Gets, Puts, Scans:** Displays the number of operations (Gets, Puts, Scans) over time intervals (10s, 1m, 5m, 15m). The data is as follows:

	Gets	Puts	Scans
10s	92305	63292	64747
1m	685552	465767	982700
5m	465902	430158	482473
15m	8942039	508467	460070

At the bottom left is the MAPR Academy logo, and at the bottom right are copyright and page number information: © 2016 MapR Technologies L13-70.

The DB Gets, Puts, Scans pane displays the number of Gets, Puts, Scan operations performed during various time intervals.

The Machine Performance pane displays the following information about the node's performance and resource usage since it last reported to the CLDB:

- Memory Used - the amount of memory in use on the node.
- Disk Used - the amount of disk space used on the node.
- CPU - the number of CPUs and the percentage of CPU used on the node.
- Network I/O - the input and output to the node per second.
- RPC I/O - the number of RPC calls on the node and the amount of RPC input and output.
- Disk I/O - the amount of data read to and written from the disk.
- Number of Operations - the number of disk reads and writes.

## Regions

Cluster Name: BigBertha

Navigation

- Cluster
  - Dashboard
  - Nodes
  - Node Heatmap
  - Jobs
- MapR-FS
  - MapR Tables
  - Volumes
  - Mirror Volumes
  - User Disk Usage
  - Snapshots
  - Schedules
- NFS HA
  - NFS Setup
  - VIP Assignments
  - NFS Nodes

Regions

Start Key	End Key	Physical Size	Logical Size	# Rows	Primary Node	Secondary Nodes	Last HB	Region Identifier
-∞	17084614	739.8MB	1.4GB	7,871,797	CentOS002	CentOS003 CentOS004	0 ago	2324.36.787384
17084614	241566	897.9MB	1.7GB	9,583,927	CentOS005	CentOS002 CentOS001	0 ago	2308.32.525090
241566	475149	3GB	5.9GB	33,141,360	CentOS004	CentOS006 CentOS005	0 ago	2361.32.656030
475149	∞	779.8MB	1.5GB	8,316,776	CentOS006	CentOS002 CentOS003	0 ago	2328.32.526362

MAPR Academy © 2016 MapR Technologies L13-71

For each region, you can see:

- Number of rows
- Start key and end key
- The physical size and logical size
- Which node the region is on

## Monitoring Table Region Information

Table splits happen on an ongoing basis, no need to manage region splits or data compaction

- No settings or operations to control region splits or data compaction

Region information: size and location of table data on the cluster

Column Families	Regions			
Start Key	Size	# Rows	Last HB	Primary Node
-∞	0	0	0s ago	qa-cnode101.lab

With MapR-DB there is no need for settings for region splits or data compaction. In a MapR-DB table regions panel you can see the size and location of table data on the cluster.

## Table Regions Tab

Select a table to open the display → Initial view is the table regions tab

- Includes start and end key
- Physical and logical size
- Number of rows in region
- Primary node and secondary nodes
- Last Heart Beat (HB)
- Region identifier

Delete Table		
Column Families	Regions	
Start Key	End Key	Physical Size
-∞	58919	42MB
58919	128125	48.6MB
128125	177839	35.3MB

Select a table to open the display → Initial view is the table regions tab

In a MapR-DB table regions tab, you can see the:

- Start and end key
- Physical and logical size
- Number of rows in region
- Primary node and secondary nodes
- Last Heart Beat (HB)
- Region identifier

## Primary and Secondary Nodes

Primary node is the region's original source for data and computation in an MapR-DB table

Primary Node	Secondary Nodes	Last HB
ip-10-172-f0-108.us...	ip-10-197-56-189.us...	0 ago
...	ip-10-174-31-113.us...	0 ago
ip-10-197-55-234.us...	ip-10-197-56-189.us...	0 ago
ip-10-197-56-189.us...	ip-10-172-10-108.us...	0 ago
ip-10-174-31-113.us...	ip-10-172-10-108.us...	0 ago
	ip-10-197-55-234.us...	0 ago

The secondary nodes provide the replicas for data

Primary node is the region's original source for data and computation in an MapR-DB table.  
The secondary nodes provide the replicas for data.

## Column Families

The screenshot shows the MapR Control System interface with the following details:

- Cluster Name:** BigBertha
- Navigation:** Cluster (Dashboard, Nodes, Node Heatmap, Jobs), MapR-FS (MapR Tables, Volumes, Mirror Volumes, User Disk Usage, Snapshots, Schedules).
- Current View:** MapR Tables > table/\_data2/table3 > Column Families.
- Actions:** Delete Table, New Column Family, Edit Column Family, Delete Column Family.
- Table Data:** A table listing column families with the following columns: Column Family Name, Max Versions, Min Versions, Compression, Time-to-live, and In Memory.

Column Family Name	Max Versions	Min Versions	Compression	Time-to-live	In Memory
cf1	3	1	zlib	1h 40m	✓
cf2	3	2	lzf	1w 6.9d	✓
cf3	4	3	lzf	2w 3.4d	
cf4	2	1	lz4	4mo 4.1w	

- Footer:** MAPR Academy, © 2016 MapR Technologies, L13-75.

You can use the MapR control system to set properties for column families. You can set the Max Min versions, Compression type, time-to-live, and the in-memory setting, whether the CF is kept in memory or not.

## Column Families Tab

### The Table Column Family Tab

- Column families are an important organizer of column data
- A column family is a group of columns within a table
- All regions in the table have the same column families
- MapR-DB easily supports up to 64 column families, more than HBase

Column Family Name	Max Versions	Min Versions	Compression	Time-to-live	In M...
cf1	119	1	lzf	1w 4.6d	
cf2	8	-	lzf	Forever	✓

- This table shows two column families

### The Table Column Family Tab

- Column families are an important organizer of column data
- A column family is a group of columns within a table
- All regions in the table have the same column families
- MapR-DB easily supports up to 64 column families, more than HBase

## Compression and In-Memory

	Compression	Time-to-live	In M...
Izf	1w 4.6d		<b>Compression:</b> The compression type applied to column family data. If left unspecified, the initial value inherits the table's compression type.

Compression is configurable at the column family level

	In M...	⋮
		<b>In Memory:</b> The column family can be pinned in memory. This value is editable.

To keep a column family in memory, select here. This can help performance.

Compression and in-memory is configurable at the column family level, which can help performance.

## Other Performance Monitoring Options

### Nagios

- <http://www.nagios.org/>

### Ganglia

- <http://ganglia.info/>

### JMX

- <http://www.oracle.com/technetwork/java/javase/tech/javamanagement-140525.html>

### OpenTSDB

- <http://opentsdb.net/>

You can use other tools to monitor performance besides MCS. We have discussed OpenTSDB and you can use it to monitor performance by collecting information from your infrastructure logs.

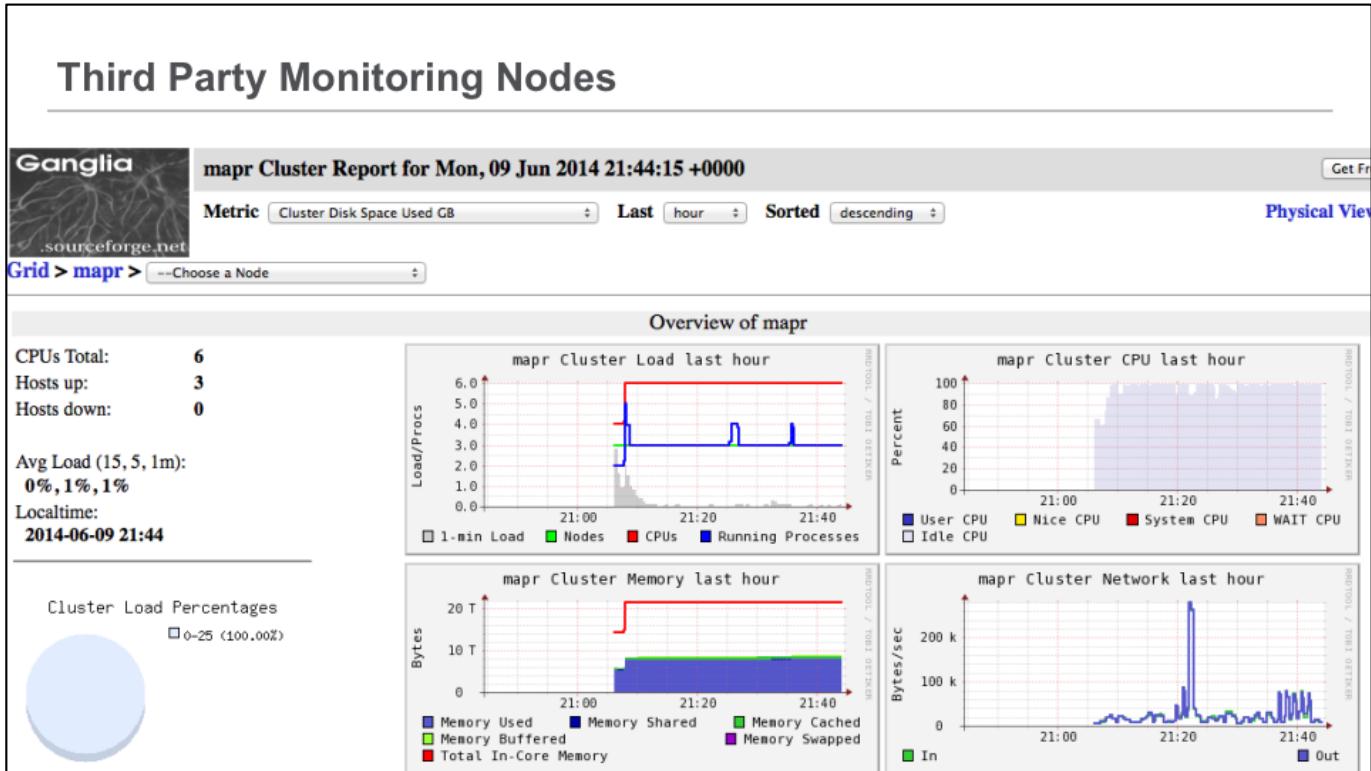
Nagios, Ganglia, and JMX are tools for monitoring actions in the cluster. They're not specifically geared for analyzing jobs, but they may help you identify slow performance in the cluster, which you can trace back to poor job structure.

More classical monitoring tools like Nagios can provide useful metrics.

Ganglia is an open source system for monitoring distributed systems.

JMX the Java Management Extensions framework can also be used for monitoring.

You can Mbeans to expose specific attributes and operations from your system to JMX. You can then monitor things from a JMX client like the java console. Most projects in the Hadoop ecosystem expose metrics via JMX and they can be enabled via configuration.



This is a screen shot of monitoring with Ganglia, which is a popular open source system for monitoring distributed systems.

## For More Information

- <http://doc.mapr.com/display/MapR/HBase#HBase-bestpractices>
- <http://hbase.apache.org/book.html#perf>

More information can be found in the online documentation.

<http://doc.mapr.com/display/MapR/HBase#HBase-bestpractices>

## Lab 13: Performance Benchmarking with YCSB





## Next Steps

### DEV 340 – HBase Applications: Bulk Loading, Security, and Performance

Lesson 14: Security

Congratulations, you have finished DEV 340 Lesson 13, Performance. Continue to Lesson 14 to learn about Security.



## **DEV 340 – HBase Applications: Bulk Loading, Security, and Performance**

Lesson 14: Security

Winter 2017, v5.1

Welcome to DEV 340, HBase Applications: Bulk Loading, Security, and Performance, Lesson 14: Security.

## Learning Goals



## Learning Goals



14.1 Fundamentals

14.2 Securing MapR-DB

After this lesson you will be able to:

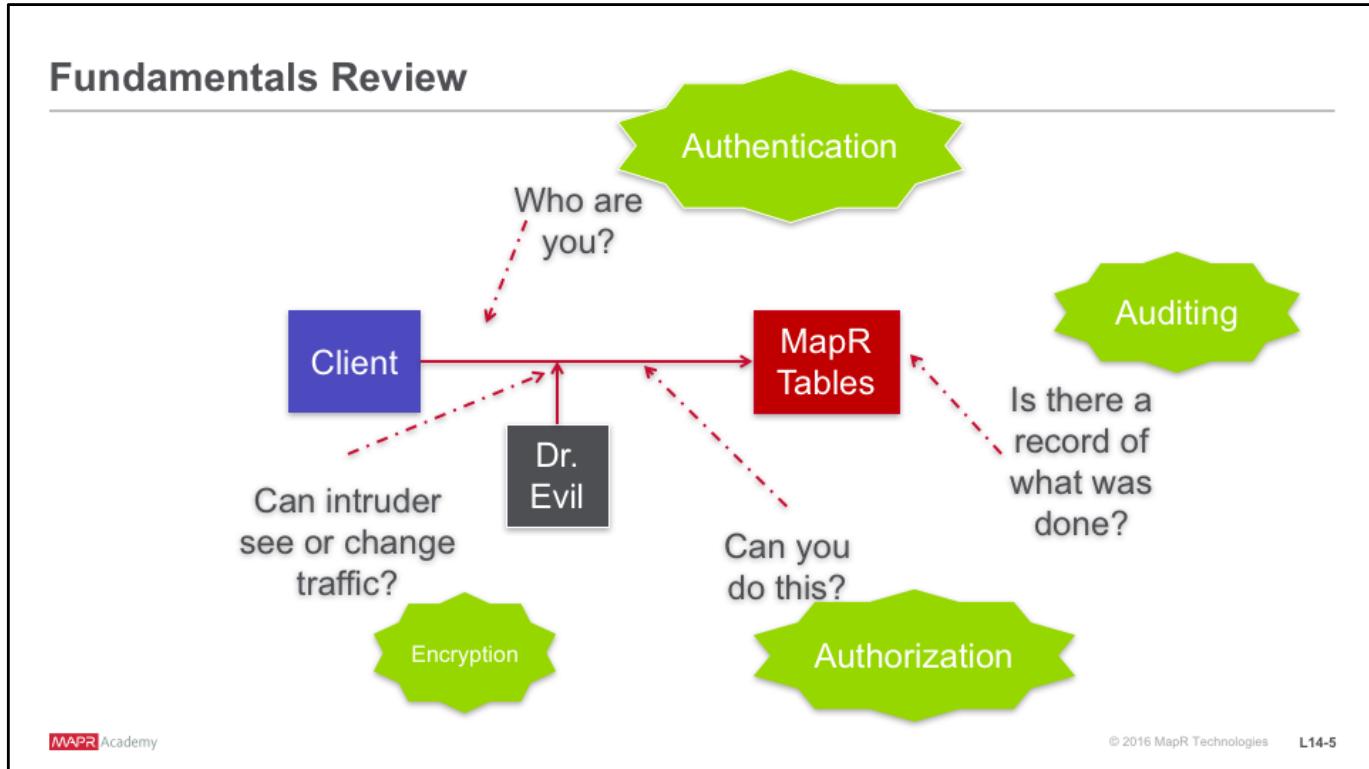
- Describe the fundamentals of security
- Understand how to secure MapR-DB

## Learning Goals



### 14.1 Fundamentals

14.2 Securing MapR-DB



A secure environment is predicated on the following capabilities:

**Authentication:** Restricting access to a specified set of users. Robust authentication prevents third parties from representing themselves as legitimate users.

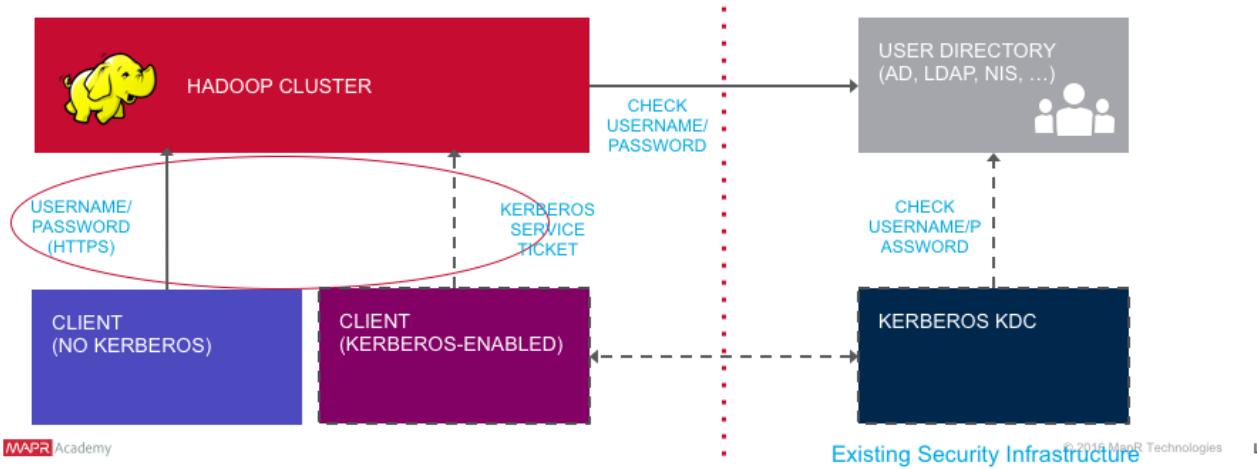
**Authorization:** Restricting an authenticated user's capabilities on the system. Flexible authorization systems enable a system to grant a user a set of capabilities that enable the user to perform desired tasks, but prevents the use of any capabilities outside of that scope.

**Encryption:** Restricting an external party's ability to read data. Data transmission between nodes in a secure MapR cluster is encrypted, preventing an attacker with access to that communication from gaining information about the transmission's contents.

## Authentication: Who Are You?

Identifies User

MapR: UserId and Password or Kerberos



The core component of user authentication in MapR is the ticket. A ticket is an object that contains specific information about a user, an expiration time, and a key. Tickets uniquely identify a user and are encrypted to protect their contents. Tickets are used to establish sessions between a user and the cluster.

MapR supports two methods of authenticating a user and generating a ticket: a username/password pair and Kerberos. Both of these methods are mediated by the [maprlogin](#) utility. When you authenticate with a username/password pair, the system verifies credentials using Pluggable Authentication Modules (PAM). You can configure the cluster to use any registry that has a PAM module.

## Learning Goals



## Learning Goals



14.1 Fundamentals

### **14.2 Securing MapR-DB**

## MapR-DB Tables Authorization

- Access Control Expressions (ACEs)
- Boolean expression of roles, users, groups to control access at table, column family, and column level

MAPR Academy

© 2016 MapR Technologies L14-9

Permissions for MapR tables, column families, and columns are defined by Access Control Expressions (ACEs).

An ACE (Access Control Expression) is a boolean expression of roles, users, groups and boolean operators "&" "|" "!", used to control MapR-DB access at table, column family, and column level.

When a user, group, or role requests to read data from, or write data to a column, MapR-DB checks whether that user, group, or role has read or write permission for the column family and read or write permission for the column.

This screen shot shows the Edit Table Permissions screen which allows you to set permissions for tables when you create or edit tables.

## MapR-DB Tables Authorization

- Access Control Expressions can be defined on
- MapR-DB Table **Data**: Table → Column Families → Columns (Read, Write, Append)
- MapR-DB Table **Operations** (add/delete cf, split/merge, pack operations)
- ACE Syntax
- Tokens
  - u(uid), g(gid), r(role)
  - p(public)
- Operators
  - | (OR), & (AND), !(NOT)
- Examples
  - u:1001 | r:engineering
  - g:admin | g:qa

Access Control Expressions can be defined on MapR-DB table Data from the table to the column level, and on MapR-DB Table Operations such as add/delete column family, split/merge, pack operations.

An ACE is defined by a combination of user, group, or role tokens with operators. !Negation operator, & AND operator, |OR operator.

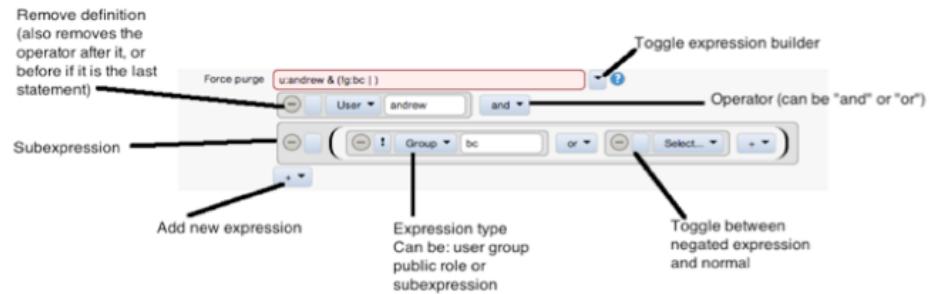
An example definition is (user 1001 or role engineering), which restricts access to the user with ID 1001 or to any user with the role engineering.

In the next example, (group admin OR group qa), members of the group admin are given access, and so are members of the group qa.

## Access Controls for MapR-DB

MCS GUI provides an expression builder that validates the correctness of the settings in real-time

Expression builder options and definitions (all these have tooltips which show what they do when the user mouse's over):



The MCS GUI provides an expression builder that validates the correctness of the settings in real-time.

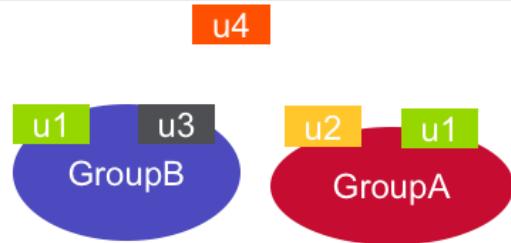
For more information refer to the documentation:

<http://doc.mapr.com/display/MapR/Setting+Permissions+by+Defining+Access+Control+Expressions#SettingPermissionsbyDefiningAccessControlExpressions-mcs>

## MapR-DB Tables Authorization

### ACE Examples

- g:groupB & (g:groupA)
- g:groupB | (g:groupA)
- !g:groupA **WARNING!**
- g:groupB & (!g:groupA)
- Subset of groupB
- (u:u1 & g:admin) | (u:u2 & g:deploy & (g:!test | g:!qa))
- User1 and in admin group OR
- User2 and in deploy group and is either not in test group or not in the qa group



Here we see two groups GroupB and GroupA with users u1, u3, in GroupB. u2, u1 in GroupA, and u4 in no group as shown.

Some ACE examples are:

groupb and groupa which would be only user u1  
groupb or groupa which would be users u1 , u3, u2

A warning about the NOT operator: NOT implies a very large universe. For example "groupA" is everyone that isn't in groupA which is a lot of people. NOT is best used to limit some other constraint. For example "groupB&!groupA" which would be user u3 in this example.

## MapR-DB Tables Authorization

- Access Control Expressions can be defined on
- MapR-DB **Tables Operations** (add/delete CF, split/merge, pack operations)
- Creator of table has all rights **by default**
- Others have none

Edit Table Permissions

Permissions

Force pack	u:mapr	?
Split Merge	u:mapr	?
Create/Rename Column Family	u:mapr	?
Delete Column Family	u:mapr	?
Admin access	u:mapr	?

Column Family Default Permissions

Set min/max versions	u:mapr	?
Set compression	u:mapr	?
Pin CF in memory	u:mapr	?
Read Data	u:mapr	?
Write Data	u:mapr	?

MAPR Academy

© 2016 MapR Technologies

L14-13

Table level ACEs mainly deal with restricting users from modifying table attributes, that is adding/renaming/removing column families.

A new table's permissions default to the UID of the user creating the table. The creator of the table has all rights by default, others have none.

## MapR-DB Tables Authorization

Access Control Expressions can be defined on

- Table → **Column families default** permissions
- **Defaults** for column families set at table level

The screenshot shows a list of five permissions under the heading "Column Family Default Permissions". Each permission has a "Set" button and a question mark icon.

Action	User	Set	?
Set min/max versions	u:mapr		
Set compression	u:mapr		
Pin CF in memory	u:mapr		
Read Data	u:mapr		
Write Data	u:mapr		

You can set default permissions for column families when you create or edit tables and you can override these defaults when you create column families.

This shows the column family default permissions screen, which allows you to specify default column family permissions when you are creating a table. These ACEs get inherited by column families when they are created. These permissions default to the creating user, as shown here user MapR, but this screen allows you to set the default to another ACE.

## MapR Tables Authorization

Access Control Expressions can be defined on

- Column Family permissions
- Inherited from default, if specified **overrides default**

The screenshot shows a list of permissions for a column family:

Action	Permission	Help
Set min/max versions	u:mapr	?
Set compression	u:mapr	?
Pin CF in memory	u:mapr	?
Read Data	u:mapr   r:role1	?
Write Data	u:mapr   w:role1	?

As we said before, you can set default permissions for column families when you create or edit tables, which will be inherited, you can also override these defaults when you create column families.

Column family permissions are inherited when they are created, from the default column family permissions, which you can set in the previous screen (column family default permissions). The column family permissions screen shown here allows you to explicitly set ACEs for a column family which will override the defaults.

## MapR Tables Authorization

Access Control Expressions can be defined on

- Columns (Read, Write, Append)
- Access = Column family and column permissions
  - Not set on creation which means must just pass CF

The screenshot shows the 'Column Permissions' configuration screen. On the left, there's a list of columns under 'Column Permissions'. A single column, 'c1', is selected and highlighted in blue. To the right of the list, there are two main sections: 'Title' and 'Values'. Under 'Title', the 'Name' field is set to 'c1'. Under 'Values', there are three fields: 'Read Data' containing 'u:mapr | r:role1', 'Write Data' (empty), and 'Append Data' (empty). Each field has a 'Help' button (a question mark icon) and a 'Copy' button (a blue square with a white arrow).

Unlike column families, columns are not predefined since users can dynamically add any column during a put operation.

Column access is an AND of the permissions on the column family AND column, that is if a user wants to Get or Put data from or to a particular column, he needs to pass access tests of both ColumnFamily and Column. For example when a user, group, or role requests to read data from, or write data to a column, MapR-DB checks whether that user, group, or role has read or write permission for the column family AND read or write permission for the column.

For example, suppose user carol tries to write data to columns col1 and col2 in column family cf1. MapR-DB checks whether carol has write permission on cf1 AND col1 AND col2. If carol does not have all three permissions, MapR-DB returns an error that says access for the write is denied.

If this user were to try to read from the same two columns, MapR-DB would simply not return the data. If the user tried to read from those two columns and additional columns on which she had read permissions, the results would contain the data for those additional columns but exclude the data for col1 and col2.

## Role Based Access Control

- Roles are a label that defines a common task or **set of behaviors** related to permissions for an application



A role is a name or label that defines a common task or set of behaviors related to permissions for an application, for example admin, staff.

1. You can define a logical role in an ACE, such as Admin, and give permissions for that role.
2. Then you can map this role to a set of user's userIds, giving those users the permissions for that role, in this case Admin role permissions.

Roles enable you to use functionality similar to Unix groups for your users without requiring you to alter your system's existing group hierarchy.

## Lab 14.2: Security





## Next Steps

### Congratulations!

Congratulations! You have finished DEV 340, HBase Applications: Bulk Loading, Security, and Performance.