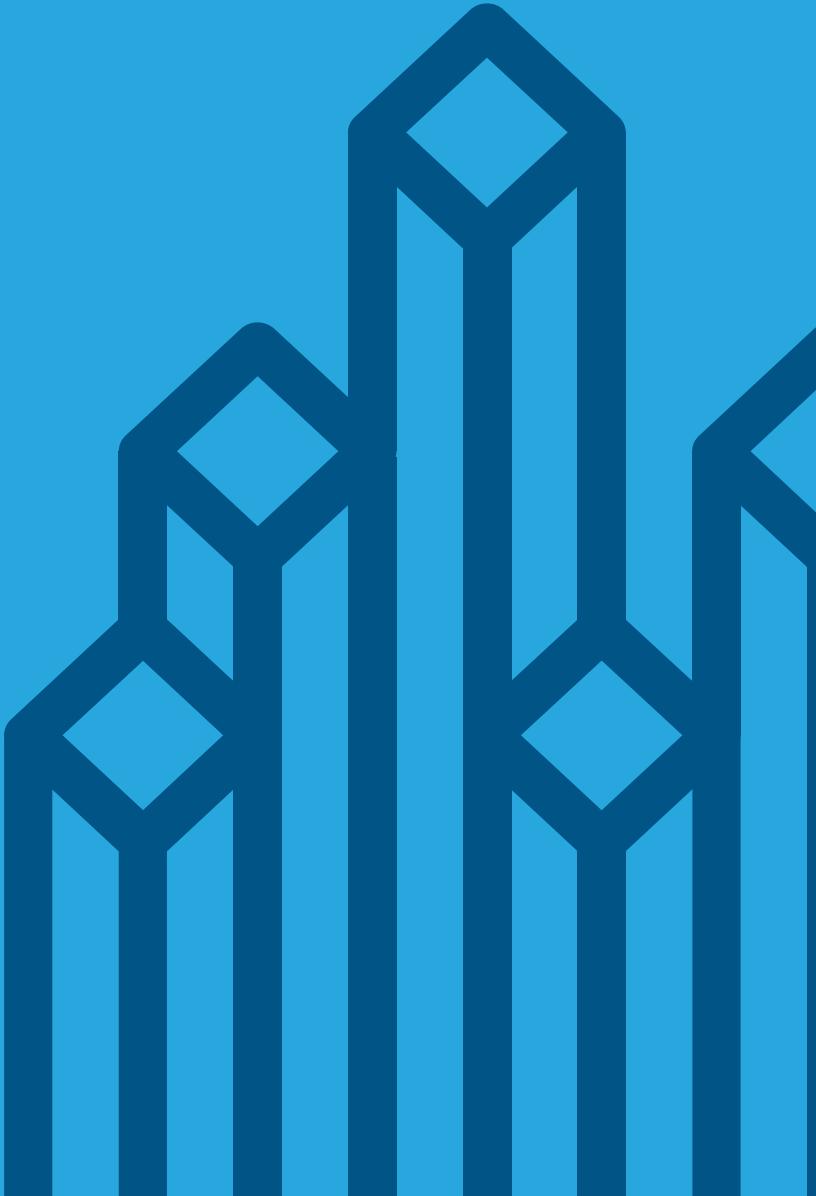




Cloudera Training for Apache HBase





Introduction

Chapter 1



Chapter Topics

Introduction

- **About this course**
- About Cloudera
- Course logistics
- Introductions

Course Objectives

During this course, you will learn:

- **The core technologies of Apache HBase**
- **How HBase and HDFS work together**
- **How to work with the HBase shell and the Java API**
- **The HBase storage and cluster architecture**
- **The fundamentals of HBase administration**
- **Advanced features of the HBase API**
- **The importance of schema design in HBase**
- **How to use Hive and Impala with HBase**

Course Chapters

- **Introduction**
- Introduction to Hadoop and HBase
- HBase Tables
- HBase Shell
- HBase Architecture Fundamentals
- HBase Schema Design
- Basic Data Access with the HBase API
- More Advanced HBase API Features
- HBase on the Cluster
- HBase Reads and Writes
- HBase Performance Tuning
- HBase Administration and Cluster Management
- HBase Replication and Backup
- Using Hive and Impala with HBase
- Conclusion
- Appendix: Using Python and Thrift to Access HBase Data
- Appendix: OpenTSDB

Chapter Topics

Introduction

- About this course
- **About Cloudera**
- Course logistics
- Introductions

About Cloudera (1)

- **The leader in Apache Hadoop-based software and services**
- **Founded by leading experts on Hadoop from Facebook, Yahoo, Google, and Oracle**
- **Provides support, consulting, training, and certification for Hadoop users**
- **Staff includes committers to virtually all Hadoop projects**
- **Many authors of industry standard books on Apache Hadoop projects**
 - Tom White, Lars George, Kathleen Ting, Amandeep Khurana, Ricky Saltzer, and others

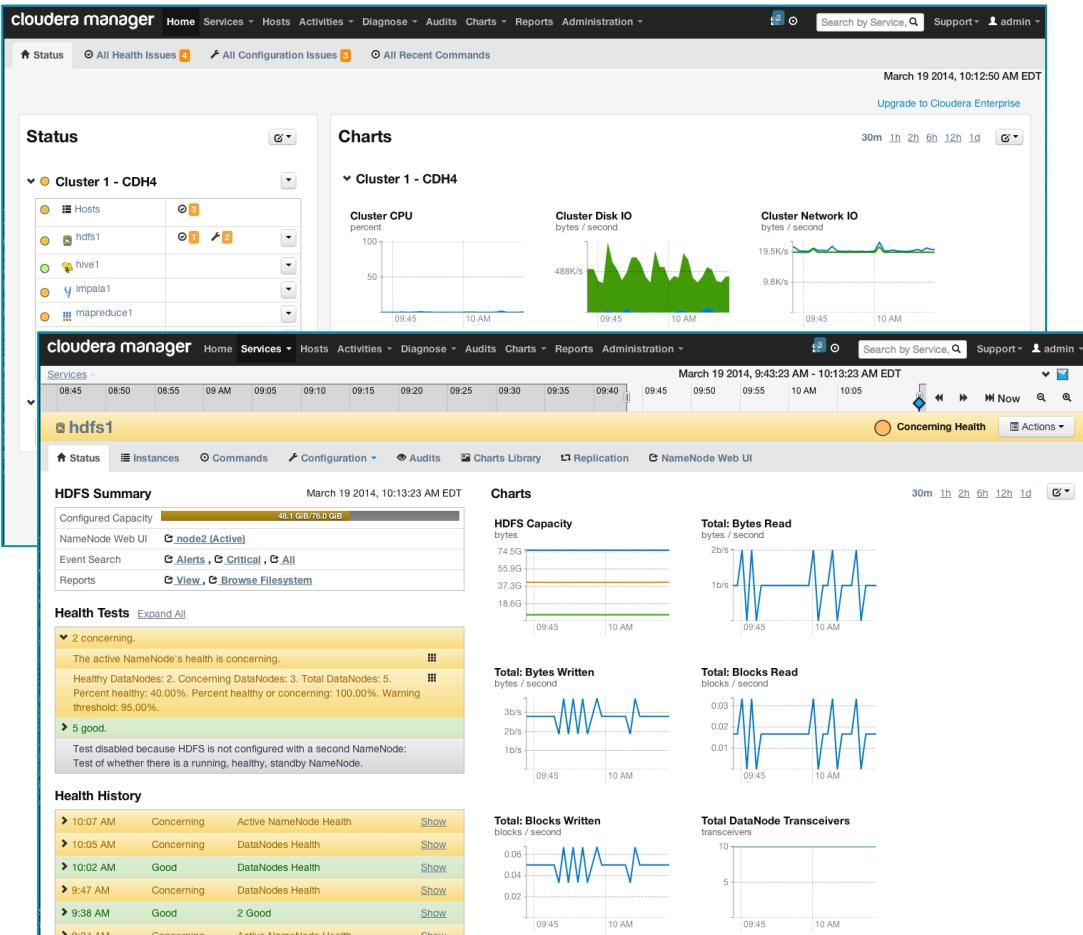
About Cloudera (2)

- **Customers include many key users of Hadoop**
 - Allstate, AOL Advertising, Box, CBS Interactive, eBay, Experian, Groupon, Macys.com, National Cancer Institute, Orbitz, Social Security Administration, Trend Micro, Trulia, US Army, ...
- **Cloudera public training:**
 - Cloudera Developer Training for Apache Spark
 - Cloudera Developer Training for Apache Hadoop
 - Designing and Building Big Data Applications
 - Cloudera Administrator Training for Apache Hadoop
 - Cloudera Data Analyst Training: Using Pig, Hive, and Impala with Hadoop
 - Cloudera Training for Apache HBase
 - Introduction to Data Science: Building Recommender Systems
 - Cloudera Essentials for Apache Hadoop
- **Onsite and custom training is also available**

- **100% open source, enterprise-ready distribution of Hadoop and related projects**
 - **The most complete, tested, and widely-deployed distribution of Hadoop**
 - **Integrates all key Hadoop ecosystem projects**
 - **Available as RPMs and Ubuntu/Debian/SuSE packages, or as a tarball**
-
- The diagram illustrates the CDH architecture with the following layers:
- METADATA**: A vertical blue bar on the left.
 - Engines**: A horizontal bar at the top spanning six orange boxes.
 - WORKLOAD MANAGEMENT (YARN)**: A blue horizontal bar below the engines.
 - STORAGE FOR ANY TYPE OF DATA UNIFIED, ELASTIC, RESILIENT, SECURE (Sentry)**: A large blue horizontal bar containing two white rounded rectangles.
 - DATA INTEGRATION (Sqoop, Flume, NFS)**: A blue horizontal bar at the bottom.
- Details for the Engines layer:
- | Category | Subsystems |
|-------------------|-----------------------------|
| BATCH PROCESSING | MapReduce, Spark, Hive, Pig |
| ANALYTIC SQL | Impala |
| SEARCH ENGINE | Cloudera Search |
| MACHINE LEARNING | Spark, MapReduce, Mahout |
| STREAM PROCESSING | Spark |
| 3rd PARTY APPS | Partners |

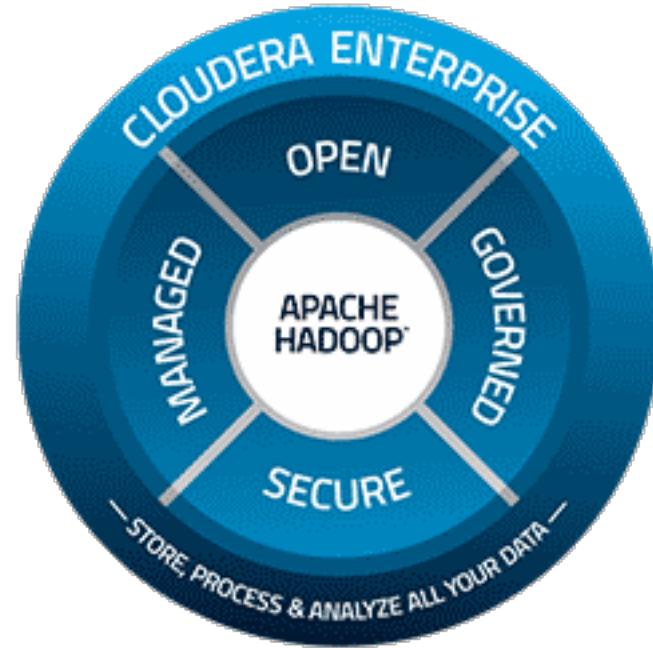
Cloudera Express

- Cloudera Express
 - Free download
- The best way to get started with Hadoop
- Includes CDH
- Includes Cloudera Manager
 - End-to-end administration for Hadoop
 - Deploy, manage, and monitor your cluster



Cloudera Enterprise

- **Cloudera Enterprise**
 - Subscription product including CDH and Cloudera Manager
- **Includes support**
- **Includes extra Cloudera Manager features**
 - Configuration history and rollbacks
 - Rolling updates
 - LDAP integration
 - SNMP support
 - Automated disaster recovery
- **Extend capabilities with Cloudera Navigator subscription**
 - Event auditing, metadata tagging capabilities, lineage exploration
 - Available in both the Cloudera Enterprise Flex and Data Hub editions



Chapter Topics

Introduction

- About this course
- About Cloudera
- **Course logistics**
- Introductions

Logistics

- Course start and end times
- Lunch
- Breaks
- Restrooms
- Can I come in early/stay late?
- Certification

Chapter Topics

Introduction

- About this course
- About Cloudera
- Course logistics
- **Introductions**

Introductions

- **About your instructor**
- **About you**
 - Experience with HBase?
 - Experience as a developer? As a system administrator?
 - Experience with relational or NoSQL Databases?
 - Expectations from the course?



Introduction to Hadoop and HBase

Chapter 2



Course Chapters

- Introduction
- **Introduction to Hadoop and HBase**
- HBase Tables
- HBase Shell
- HBase Architecture Fundamentals
- HBase Schema Design
- Basic Data Access with the HBase API
- More Advanced HBase API Features
- HBase on the Cluster
- HBase Reads and Writes
- HBase Performance Tuning
- HBase Administration and Cluster Management
- HBase Replication and Backup
- Using Hive and Impala with HBase
- Conclusion
- Appendix: Using Python and Thrift to Access HBase Data
- Appendix: OpenTSDB

Introduction to Hadoop and HBase

In this chapter you will learn

- The fundamentals of Hadoop
- Hadoop components
- About HBase
- When to use HBase

Chapter Topics

Introduction to Hadoop and HBase

- **Introducing Hadoop**
- Core Hadoop Components
- Hands-On Exercise: Using HDFS
- What is HBase?
- Strengths of HBase
- HBase in Production
- Weaknesses of HBase
- Conclusion

Hadoop's History

- **Hadoop is based on work done by Google in the late 1990s and early 2000s**
 - Specifically, Hadoop was developed from papers describing the Google File System (GFS), published in 2003, and MapReduce, published in 2004
- **This work takes a radical new approach to the problem of distributed computing**
- **Core concepts:**
 - Distribute the data across nodes as it is initially stored in the system
 - Individual nodes run programs that process data local to those nodes
 - Minimizes data transfer over the network

Hadoop and Big Data

- **Hadoop is a distributed system aimed at managing Big Data**
- **Elegantly handles the issues associated with Big Data**
- **Shared file system: the Hadoop Distributed File System (HDFS)**
 - Can grow to petabytes in size
 - Purpose-built to handle files that are gigabytes or even terabytes in size
 - Highly available with no single point of failure
- **Efficient and scalable processing system: MapReduce**
 - Can spread the processing load efficiently across many computers
 - Relatively easy to program, with no low-level coding needed

The Hadoop Ecosystem and Big Data

- **Other projects add functionality to ‘core Hadoop’**
- **HBase: a highly available and scalable database**
 - Can store massive amounts of data
 - Gigabytes, terabytes, or even petabytes of data in a table
 - Very high throughput to handle a massive user base
- **Spark: An alternative to MapReduce**
 - Faster, easier to use, can process streaming data
- **Hive and Impala: SQL-like languages**
 - Process big data using a familiar language (SQL)
 - Analyze data without needing to be a programmer
- **Sqoop and Flume: tools to ingest data from external systems**
 - Data held in RDBMSs or being generated on-the-fly can be easily imported into Hadoop

Chapter Topics

Introduction to Hadoop and HBase

- Introducing Hadoop
- **Core Hadoop Components**
- Hands-On Exercise: Using HDFS
- What is HBase?
- Strengths of HBase
- HBase in Production
- Weaknesses of HBase
- Conclusion

Hadoop Components

- A set of machines running HDFS and MapReduce is known as a **Hadoop cluster**
 - Individual machines are known as nodes
 - A cluster can have as few as one node, or as many as several thousands
 - More nodes = better performance!

Hadoop Components: MapReduce

- **MapReduce is the system used to process data in the Hadoop cluster**
- **Consists of two phases: Map, and then Reduce**
 - Between the two is a stage known as the *shuffle and sort*
- **Each Map task operates on a discrete portion of the overall dataset**
- **After all Maps are complete, the MapReduce system distributes the intermediate data to nodes which perform the Reduce phase**

Hadoop Components: HDFS

- **HDFS is a filesystem written in Java**
 - Based on Google's GFS
- **Sits on top of a native filesystem**
 - ext3, ext4, xfs, etc.
- **Provides redundant storage for massive amounts of data**
 - Using industry-standard hardware

How Files are Stored in HDFS

- **Files are split into blocks**
 - Each block is usually 64MB or 128MB
- **Data is distributed across many machines at load time**
 - Blocks are replicated across multiple machines
- **A master node keeps track of which blocks make up a file, and where those blocks are located**
 - Known as the *metadata*

hdfs dfs Examples (1)

- Access to HDFS from the command line is achieved with the `hdfs dfs` command
- Copy file `foo.txt` from local disk to the user's home directory in HDFS

```
hdfs dfs -put foo.txt foo.txt
```

- Copy that file to a file on local disk named as `baz.txt`

```
hdfs dfs -get /user/fred/bar.txt baz.txt
```

hdfs dfs Examples (2)

- Display the contents of the HDFS file /user/fred/bar.txt

```
hdfs dfs -cat /user/fred/bar.txt
```

- Get a directory listing of the user's home directory in HDFS

```
hdfs dfs -ls
```

- Get a directory listing of the HDFS root directory

```
hdfs dfs -ls /
```

hdfs dfs Examples (3)

- Create a directory called **input** under the user's home directory

```
hdfs dfs -mkdir input
```

- Delete the directory **input_old** and all of its contents

```
hdfs dfs -rm -r input_old
```

Chapter Topics

Introduction to Hadoop and HBase

- Introducing Hadoop
- Core Hadoop Components
- **Hands-On Exercise: Using HDFS**
- What is HBase?
- Strengths of HBase
- HBase in Production
- Weaknesses of HBase
- Conclusion

Aside: The Training Virtual Machine

- During this course, you will perform numerous Hands-On Exercises using the Cloudera Training Virtual Machine (VM)
- The VM has Hadoop installed in *pseudo-distributed mode*
 - This essentially means that it is a cluster comprised of a single node
 - Using a pseudo-distributed cluster is the typical way to test your code before you run it on your full cluster
 - It operates almost exactly like a ‘real’ cluster
 - A key difference is that the data replication factor is set to 1, not 3
- HBase is installed in *pseudo-distributed mode*
 - Runs each HBase daemon and a local ZooKeeper in individual JVMs
 - All HBase files are stored in HDFS
 - More on HBase’s installation modes later

Hands-On Exercise: Using HDFS

- In this Hands-On Exercise you will gain familiarity with manipulating files in HDFS
- Please refer to the Exercise Manual

Chapter Topics

Introduction to Hadoop and HBase

- Introducing Hadoop
- Core Hadoop Components
- Hands-On Exercise: Using HDFS
- **What is HBase?**
- Strengths of HBase
- HBase in Production
- Weaknesses of HBase
- Conclusion

What is HBase?

- **HBase is a NoSQL database that runs on top of HDFS**
- **HBase is...**
 - Highly available and fault tolerant
 - Very scalable, and can handle high throughput
 - Able to handle massive tables with ease
 - Well suited to sparse rows where the number of columns varies
 - An open-source, Apache project
- **HDFS provides:**
 - Fault tolerance
 - Scalability

Google BigTable

- **HBase is based on Google's BigTable**
- **The Google use case is to search the entire Internet**
 - BigTable is at the center of how the company accomplishes this
- **Engineering considerations for searching the Internet:**
 - How do you efficiently download the Web pages?
 - How do you store the entire Internet?
 - How do you index the Web pages?
 - How do you efficiently search the Web pages?

Chapter Topics

Introduction to Hadoop and HBase

- Introducing Hadoop
- Core Hadoop Components
- Hands-On Exercise: Using HDFS
- What is HBase?
- **Strengths of HBase**
- HBase in Production
- Weaknesses of HBase
- Conclusion

What Differentiates HBase?

- HBase helps solve data access issues where random access is required
- HBase scales easily, making it ideal for Big Data storage and processing needs
- Columns in an HBase table are defined dynamically, as required

HBase Usage Scenarios (1)

- **High capacity**
 - Massive amounts of data
 - Hundreds of gigabytes, growing to terabytes or even petabytes
 - Example: Storing the entire Internet
- **High read and write throughput**
 - 1000s/second per node
 - Example: Facebook Messages
 - 75 billion operations per day
 - Peak usage of 1.5 million operations per second
 - Runs entirely on HBase

HBase Usage Scenarios (2)

- **Scalable in-memory caching**
 - Adding nodes adds to available cache
- **Large amount of stored data, but queries often access a small subset**
 - Data is cached in memory to speed up queries by reducing disk I/O
- **Data layout**
 - HBase excels at key lookup
 - No penalty for sparse columns

When To Use HBase

- **Use HBase if...**
 - You need random write, random read, or both (but not neither)
 - Your application performs thousands of operations per second on multiple terabytes of data
 - Your access patterns are well-known and relatively simple

- **Don't use HBase if...**
 - Your application only appends to your dataset, and tends to read the whole thing when processing
 - Primary usage is for ad-hoc analytics (non-deterministic access patterns)
 - Your data easily fits on one large node

Chapter Topics

Introduction to Hadoop and HBase

- Introducing Hadoop
- Core Hadoop Components
- Hands-On Exercise: Using HDFS
- What is HBase?
- Strengths of HBase
- **HBase in Production**
- Weaknesses of HBase
- Conclusion

HBase In Production

- **Many enterprises are using HBase in production**
 - Many use cases are mission critical
- **Examples of companies using HBase:**
 - eBay
 - Facebook
 - FINRA
 - Pinterest
 - Salesforce
 - StumbleUpon
 - TrendMicro
 - Twitter
 - Yahoo!
 - ...and many others

HBase Use Cases

Messaging

- Facebook Messages
- Telco SMS/MMS services
- Feeds like Tumblr, Pinterest

Simple Entities

- Geolocation data
- Search index building

Graph Data

- Sessionization
 - Financial transactions
 - Click streams
 - Network traffic

Metrics

- Campaign impressions
- Click counts
- Sensor Data

HBase Use Case Characteristics

Messaging

- Facebook Messages
- Telco SMS/MMS services
- Feeds like Tumblr, Pinterest

Characteristics

- Realtime random writes
- Reading of top N entries

Graph Data

- Sessionization
 - Financial transactions
 - Click streams
 - Network traffic

Characteristics

- Batch/realtime, random reads/writes

Simple Entities

- Geolocation data
- Search index building

Characteristics

- Batch or realtime, random writes
- Realtime, random reads

Metrics

- Campaign impressions
- Click counts
- Sensor Data

Characteristics

- Frequently updated metrics

Chapter Topics

Introduction to Hadoop and HBase

- Introducing Hadoop
- Core Hadoop Components
- Hands-On Exercise: Using HDFS
- What is HBase?
- Strengths of HBase
- HBase in Production
- **Weaknesses of HBase**
- Conclusion

Features Missing from HBase

- **HBase does not provide certain popular RDBMS features**
 - Integrated support for SQL
 - External projects such as Hive, Impala, and Phoenix provide various ways of using SQL to access HBase tables
 - Support for transactions
 - Multiple indexes on a table
 - ACID compliance

Different Design Approach in HBase

- **Designing an HBase application requires developers to engineer the system using a data-centric approach**
 - This is a less familiar approach; relationship-centric is more common
 - We will cover this in detail later in the course

Chapter Topics

Introduction to Hadoop and HBase

- Introducing Hadoop
- Core Hadoop Components
- Hands-On Exercise: Using HDFS
- What is HBase?
- Strengths of HBase
- HBase in Production
- Weaknesses of HBase
- **Conclusion**

Key Points

- There are many technological challenges when dealing with Big Data
- Hadoop is a distributed system aimed at managing Big Data
- Hadoop's core components are HDFS and MapReduce
- HBase is a NoSQL database that runs on top of HDFS
- HBase is not a traditional RDBMS
- HBase requires a data-centric design approach
- HBase allows for large tables and high throughput



HBase Tables

Chapter 3



Course Chapters

- Introduction
- Introduction to Hadoop and HBase
- **HBase Tables**
- HBase Shell
- HBase Architecture Fundamentals
- HBase Schema Design
- Basic Data Access with the HBase API
- More Advanced HBase API Features
- HBase on the Cluster
- HBase Reads and Writes
- HBase Performance Tuning
- HBase Administration and Cluster Management
- HBase Replication and Backup
- Using Hive and Impala with HBase
- Conclusion
- Appendix: Using Python and Thrift to Access HBase Data
- Appendix: OpenTSDB

HBase Tables

In this chapter you will learn

- **The concepts behind HBase**
- **How HBase and RDBMSs relate to each other**

Chapter Topics

HBase Tables

- **HBase Concepts**
- HBase Table Fundamentals
- Thinking About Table Design
- Hands-On Exercise: HBase Data Import
- Conclusion

HBase Terms

■ Definitions

- Node
 - A single computer
- Cluster
 - A group of nodes connected and coordinated by certain nodes to perform tasks
- Master Node
 - A node performing coordination tasks
- Slave/Worker Node
 - A node performing tasks assigned to it by a master node
- Daemon
 - A process or program that runs in the background

Fundamental HBase Concepts

- **HBase stores data in tables**
 - Similar to RDBMS tables but with some important differences
- **Table data is stored on the Hadoop Distributed File System (HDFS)**
 - Data is split into HDFS blocks and stored on multiple nodes in the cluster

What is a Table?

- HBase tables are comprised of rows, columns, and column families
- Every row has a *row key* for fast lookup
- Columns hold the data for the table
- Each column belongs to a particular *column family*
- A table has one or more column families

Chapter Topics

HBase Tables

- HBase Concepts
- **HBase Table Fundamentals**
- Thinking About Table Design
- Hands-On Exercise: HBase Data Import
- Conclusion

About HBase Tables

- **HBase is essentially a distributed, sorted map**
- **Distributed: HBase is designed to use multiple machines to store and serve table data**
- **Sorted Map: HBase stores table data as a map, and guarantees that adjacent keys will be stored next to each other on disk**

Example Application Data (1)

- **In our examples, we will use a table that holds**
 - User contact information
 - Profile photos
 - User sign-in information such as username and password
 - Settings or preferences for multiple applications
- **The table will be designed to provide access to the data based on the username**

Example Application Data (2)

- **Not every field will have a value**
 - This might happen if the application that stores data to a given field has not run
 - Alternatively, the user may have elected not to provide all the information
- **For now we will focus on the contact information and profile photo**
 - Contact information
 - First Name
 - Last Name
 - Profile photo
 - Image that the user uploads

Rows

- **Tables are comprised of rows, columns, and column families**
- **Every row has a *row key***
 - A row key is analogous to a primary key in a traditional RDBMS
 - Rows are stored sorted by row key to enable speedy retrieval of data

Columns

- **Columns hold the data for the table**
 - Columns can be created on the fly
 - A column exists for a particular row only if the row has data in that column
 - The table's *cells* (row-column intersection) are arbitrary arrays of bytes
- **Each column in an HBase table belongs to a particular *column family***
 - A collection of columns
- **A table has one or more column families**
 - Created as part of the table definition

HBase Columns and Column Families

- All columns belonging to the same column family have the same prefix
 - e.g., contactinfo:fname and contactinfo:lname
 - The “:” delimits the column family from the qualifier (column name)
- Tuning and storage settings can be specified for each column family
 - For example, the number of versions of each cell which will be stored
 - More on this later
- A column family can have any number of columns
 - Columns within a family are sorted and stored together

HBase Tables: A Conceptual View

- A column is referenced using its **column family** and **column name** (or *qualifier*)
- Separate column families are useful for
 - Data that is not frequently accessed together
 - Data that uses different column family options
 - e.g., compression

Row Key	contactinfo		profilephoto
	fname	Iname	image
jdupont	Jean	Dupont	
jsmith	John	Smith	<smith.jpg>
mrossi	Mario	Rossi	<mario.jpg>

Diagram illustrating the structure of the HBase table:

- Column Families: contactinfo, profilephoto
- Column Names: fname, Iname, image
- Rows: jdupont, jsmith, mrossi

Storing Data in Tables

- **Data in HBase tables is stored as byte arrays**
 - Anything that can be converted to an array of bytes can be stored
 - Strings, numbers, complex objects, images, etc.
- **Cell size**
 - Practical limit on the size of values
 - In general, cell size should not consistently be above 10MB

Table Storage On Disk: Basics

- Data is physically stored on disk on a per-column family basis
- Empty cells are not stored

File

Row Key	contactinfo	
	fname	Iname
jdupont	Jean	Dupont
jsmith	John	Smith
mrossi	Mario	Rossi

File

Row Key	profilephoto
	image
jsmith	<smith.jpg>
mrossi	<mario.jpg>

Data Storage Within a Column Family

- HBase table features

- Row key + column + timestamp --> value
- Row key and value are just bytes
- Can store anything that can be serialized into a byte array

Sorted by
Row Key
and Column

Row Key	Column	Timestamp	Cell Value
jdupont	contactinfo:fname	1273746289103	Jean
	contactinfo:lname	1273878447049	Dupont
	contactinfo:fname	1273516197868	John
	contactinfo:lname	1273871824184	Smith
	contactinfo:fname	1273616297446	Mario
	contactinfo:lname	1273616297446	Rossi

HBase Operations (1)

- **All rows in HBase are identified by a row key**
 - This is like the primary key in a relational database
- **Get/Scan retrieves data**
 - A Get retrieves a single row using the row key
 - A Scan retrieves all rows
 - A Scan can be constrained to retrieve all rows between a start row key and an end row key
- **Put inserts data**
 - A Put adds a new row identified by a row key
 - Multiple Put calls can be run to insert multiple rows with different row keys

HBase Operations (2)

- **Delete marks data as having been deleted**

- A Delete removes the row identified by a row key
 - The data is not removed from HDFS during the call but is marked for deletion
 - Physical deletion from HDFS happens later

- **Increment allows atomic counters**

- Cells containing a value stored as a 64-bit integer (a long)
 - Increment allows the value to be initially set, or incremented if it already has a value
 - Atomicity allows for concurrent access from multiple clients without fear of corruption by a write from another process

Chapter Topics

HBase Tables

- HBase Concepts
- HBase Table Fundamentals
- **Thinking About Table Design**
- Hands-On Exercise: HBase Data Import
- Conclusion

Row Key is the Only Indexed Column

- RDBMSs can have as many index columns as required
- In HBase, we have just one indexed column – the row key
- Significant effort goes into the row key planning for HBase tables
 - We rely on the row key to provide quick access to data for all applications that use a given table

Features Comparison: HBase vs. RDBMS

	RDBMS	HBase
Data layout	Row- or column-oriented	Column family-oriented
Transactions	Yes	Single row only
Query language	SQL	get/put/scan
Security	Authentication/Authorization	Access control at per-cell level, also at cluster, table, or row level
Indexes	Yes	Row key only
Max data size	TBs	PB+
Read/write throughput limits	1000s queries/second	Millions of queries/second

Replacing RDBMSs with HBase

- Replacing an RDBMS-based application with HBase requires significant re-architecting
- Some major differences
 - Data layout
 - Data access

Comparison with RDBMS Table Design

- **Joins**

- In relational databases, one would typically normalize tables and use joins to retrieve data
 - HBase does not support explicit joins
 - Instead, a lookup by row key joins data from column families

- **Scaling**

- Relational tables can scale through partitioning or sharding data
 - HBase automatically partitions data into smaller pieces

HBase vs. RDBMS Schema Design

- **Steps for designing an RDBMS schema**
 - Determine all the types of data to be stored
 - Relationship-centric: Determine relationships between data elements
 - Create tables, columns, and foreign keys to maintain relationships

- **Steps for designing an HBase schema**
 - Data-centric: Identify ways in which data will be accessed
 - Identify types of data to be stored
 - Create data layouts and keys

Chapter Topics

HBase Tables

- HBase Concepts
- HBase Table Fundamentals
- Thinking About Table Design
- **Hands-On Exercise: HBase Data Import**
- Conclusion

Hands-On Exercise: HBase Data Import

- In this Hands-On Exercise, you will try out HBase by running pre-written programs to import data from MySQL into HBase
- Please refer to the Exercise Manual

Chapter Topics

HBase Tables

- HBase Concepts
- HBase Table Fundamentals
- Thinking About Table Design
- Hands-On Exercise: HBase Data Import
- **Conclusion**

Key Points

- **Tables are comprised of rows, columns, and column families**
- **Every row has a single row key**



HBase Shell

Chapter 4



Course Chapters

- Introduction
- Introduction to Hadoop and HBase
- HBase Tables
- **HBase Shell**
- HBase Architecture Fundamentals
- HBase Schema Design
- Basic Data Access with the HBase API
- More Advanced HBase API Features
- HBase on the Cluster
- HBase Reads and Writes
- HBase Performance Tuning
- HBase Administration and Cluster Management
- HBase Replication and Backup
- Using Hive and Impala with HBase
- Conclusion
- Appendix: Using Python and Thrift to Access HBase Data
- Appendix: OpenTSDB

HBase Shell

In this chapter you will learn

- How to use the HBase shell
- How to access table data in HBase
- Basic HBase administration calls

Chapter Topics

HBase Shell

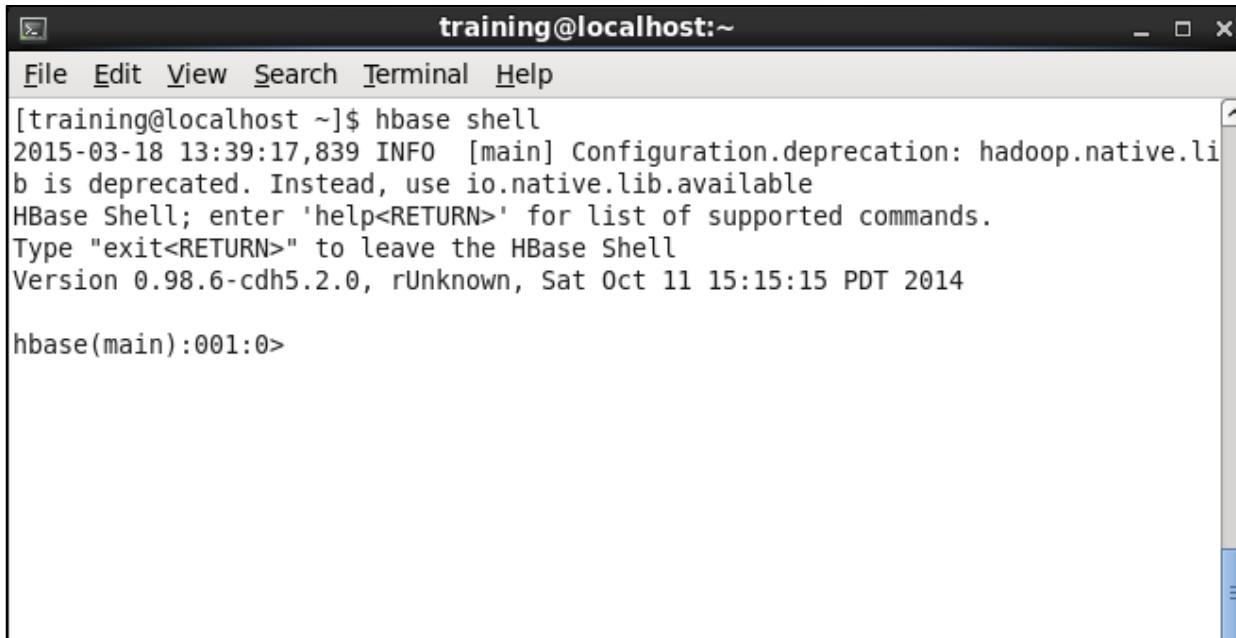
- **Creating Tables with the HBase Shell**
- Working with Tables
- Hands-On Exercise: Using the HBase Shell
- Working with Table Data
- Hands-On Exercise: Data Access in the HBase Shell
- Conclusion

HBase Shell Basics

- **The HBase Shell is an interactive shell for sending commands to HBase**
- **HBase shell uses JRuby**
 - Wraps Java client calls in Ruby
 - Allows Ruby syntax to be used for commands
 - Makes parameter usage a little different than most shells
 - Command parameters are single quoted ('')

Running the HBase Shell

```
$ hbase shell
```



A screenshot of a terminal window titled "training@localhost:~". The window contains the following text:

```
[training@localhost ~]$ hbase shell
2015-03-18 13:39:17,839 INFO  [main] Configuration.deprecation: hadoop.native.lib is deprecated. Instead, use io.native.lib.available
HBase Shell; enter 'help<RETURN>' for list of supported commands.
Type "exit<RETURN>" to leave the HBase Shell
Version 0.98.6-cdh5.2.0, rUnknown, Sat Oct 11 15:15:15 PDT 2014

hbase(main):001:0>
```

Basic Commands

- Get help:

```
hbase> help
```

- Get HBase status:

```
hbase> status
3 servers, 0 dead, 1.3333 average load
```

- Get version:

```
hbase(main):001:0> version
0.98.6-cdh5.2.0, rUnknown, Sat Oct 11 15:15:15 PDT 2014
```

Shell Command Syntax

- Shell commands are often followed by parameters

```
hbase> command 'parameter1', 'parameter2'
```

- More advanced parameters use Ruby hashes
 - Ruby hash syntax is { *PARAM* => '*stringvalue*' }
 - The Ruby '*=>*' operator is called a hash rocket
- Commands with advanced parameters look like this:

```
hbase> command 'parameter1', {PARAMETER2 => 'stringvalue',  
PARAMETER3 => intvalue}
```

- Example:

```
hbase> create 'movie', {NAME => 'desc', VERSIONS => 5}
```

Shell Scripting

- The HBase Shell works in interactive and batch modes
 - Allows a script to be written in JRuby/Ruby and passed into the shell
- Passing scripts to the HBase Shell

```
$ hbase shell pathtorubyscript.rb
```

- Shell scripts can take command line parameters to expand functionality

```
$ hbase shell  
hbase> require 'pathtorubyscript.rb'  
hbase> myRubyFunction 'parameter1'
```

HBase Table Creation

- **Only tables and column families need to be specified at table creation**
 - You must supply names for the table and column family/families
 - Every table must have at least one column family
 - Any optional settings can be changed later
- **Through a new namespace feature, HBase tables can be grouped**
 - Similar to the ‘database’ or ‘schema’ concept in relational database systems
- **Table creation is different from traditional RDBMS table creation**
 - No columns or strict relationships need to be created
 - No constraints or foreign key relationships are specified
 - No database namespace needs to be supplied

Creating Tables

- General form:

```
create 'tablename', {NAME => 'colfam' [, options]} [, {...}]
```

- Examples:

```
create 'movie', {NAME => 'desc'}
create 'movie', {NAME => 'desc', VERSIONS => 2}
create 'movie', {NAME => 'desc'}, {NAME => 'media'}
```

- Shorthand (with default options):

```
create 'movie', 'desc', 'media'
```

Managing HBase Namespaces

- HBase provides the capability to define and manage namespaces
- Multiple tables can belong to a single namespace
 - Define the table's namespace when the table is created
- Namespaces can be created, dropped, or altered:

```
create_namespace 'namespaceName'  
  
drop_namespace 'namespaceName'  
  
alter_namespace 'namespaceName' , {METHOD => 'set' ,  
'PROPERTY_NAME' => 'PROPERTY_VALUE' }
```

- There are two predefined special namespaces
 - hbase – A system namespace that holds HBase internal tables
 - default – A namespace for tables with no explicit namespace defined

Creating Tables in Namespaces

- General form:

```
create 'namespace:tablename', {NAME => 'colfam' [, options] } [,  
{...}]
```

- Examples:

```
create_namespace 'entertainment'  
create 'entertainment:movie', {NAME => 'desc'}
```

Chapter Topics

HBase Shell

- Creating Tables with the HBase Shell
- **Working with Tables**
- Hands-On Exercise: Using the HBase Shell
- Working with Table Data
- Hands-On Exercise: Data Access in the HBase Shell
- Conclusion

Listing and Describing Tables

- List all the tables in HBase

```
hbase> list
```

- Give extended details about a table

- Provides all column families in a table, their properties, and values

```
hbase> describe 'movie'
```

Disabling and Enabling Tables

- **Disabling a table puts it in a maintenance state**
 - Allows various maintenance commands to be run
 - Prevents all client access
 - May take up to several minutes for a table to disable

```
hbase> disable 'movie'
```

- **Take the table out of maintenance state**

```
hbase> enable 'movie'
```

Deleting

- **The drop command removes the table from HBase and deletes all its files in HDFS**
 - The table must be disabled first

```
hbase> disable 'movie'  
hbase> drop 'movie'
```

- **The truncate command deletes every row in the table**
 - Table and column family schema are unaffected
 - It is not necessary to manually disable the table first

```
hbase> truncate 'movie'
```

Altering Tables

- **Tables can be changed after creation**
 - The entire table can be modified
 - Column families can be added, modified, or deleted
- **For some operations, tables must be disabled while the changes are being applied**
 - Re-enable the table after changes are complete
 - As of Hbase 0.92, schema changes such as column family changes do not require the table to be disabled
 - By default, online schema changes are enabled
 - The `hbase.online.schema.update.enable` property is set to true

Adding, Deleting, and Modifying Column Families

- Add a new column family to an existing table

```
hbase> alter 'movie', NAME => 'media'
```

- Permanently delete an existing column family

```
hbase> alter 'movie', NAME => 'media', METHOD => 'delete'
```

- Modify an existing column family with new parameters

```
hbase> alter 'movie', NAME => 'desc', VERSIONS => 5
```

Altering Column Families Asynchronously

- You can use the shell command `alter_async` to alter a column family schema
 - Non-blocking form of `alter` command
 - The `alter_async` command does not wait for all regions to receive the schema changes

```
hbase> alter_async 'movie', NAME => 'desc', VERSIONS => 6
```

- Use `alter_status` to check update status of regions

Chapter Topics

HBase Shell

- Creating Tables with the HBase Shell
- Working with Tables
- **Hands-On Exercise: Using the HBase Shell**
- Working with Table Data
- Hands-On Exercise: Data Access in the HBase Shell
- Conclusion

Hands-On Exercise: Using the HBase Shell

- In this Hands-On Exercise, you will use the HBase Shell to perform basic table operations
- Please refer to the Exercise Manual

Chapter Topics

HBase Shell

- Creating Tables with the HBase Shell
- Working with Tables
- Hands-On Exercise: Using the HBase Shell
- **Working with Table Data**
- Hands-On Exercise: Data Access in the HBase Shell
- Conclusion

Gets and Puts

■ Get

- For retrieving a single row
- Must specify the exact row key to retrieve

■ Put

- Inserts a new row when the row key does not yet exist in the table
- If the row key does exist, put updates the existing value and the associated timestamp
 - If the column family is defined to keep just one version of each column value (default), only this new value is retained
 - If the column family is defined to keep n versions of each column value, up to n versions including the new value are retained
- Updates to specific column descriptors leave the row's other columns unchanged

Inserting Rows

- Insert a row with `put`

- General form:

```
hbase> put 'tablename', 'rowkey', 'colfam:col',  
'value' [,timestamp]
```

- Examples:

```
hbase> put 'movie', 'row1', 'desc:title', 'Home Alone'  
hbase> put 'movie', 'row1', 'desc:title', 'Home Alone 2',  
1274032629663
```

Retrieving Rows

- Retrieve a row with **get**
- General form:

```
hbase> get 'tablename', 'rowkey' [,options]
```

- Examples:

```
hbase> get 'movie', 'row1'  
hbase> get 'movie', 'row1', {COLUMN => 'desc:title'}  
hbase> get 'movie', 'row1', {COLUMN => 'desc:title',  
                           VERSIONS => 2}  
hbase> get 'movie', 'row1', {COLUMN => ['desc']}
```

Scans

- **A Scan is used when:**
 - The exact row key is not known
 - A group of rows needs to be accessed
- **Scans can be bounded by a start and stop row key**
 - The start row key is included in the results
 - The stop row is not included in the results and the Scan will exhaust its data upon hitting the stop row key
- **Scans can be limited to certain column families or column descriptors**
- **Scans also support Filters, to reduce the number of rows returned to the client**
 - More later

Scanning

- A scan without a start and stop row will scan the entire table
- Example: with a start row of **jordena** and an end row of **turnerb**
 - The scan will return all rows starting at **jordena** and *not* include **turnerb**

Row key	Users Table
aaronsona	fname: Aaron lname: Aaronson
harrise	fname: Ernest lname: Harris
jordena	fname: Adam lname: Jorden
laytonb	fname: Bennie lname: Layton
millerb	fname: Billie lname: Miller
nununezw	fname: Willam lname: Nunez
rossw	fname: William lname: Ross
sperberp	fname: Phyllis lname: Sperber
turnerb	fname: Brian lname: Turner
walkerm	fname: Martin lname: Walker
zykowskiz	fname: Zeph lname: Zykowski

Scanning Rows

- Retrieve a group of rows with `scan`
- General form:

```
hbase> scan 'tablename' [,options]
```

- Examples:

```
hbase> scan 'movie'  
hbase> scan 'movie', {LIMIT => 10}  
hbase> scan 'movie', {STARTROW => 'row1',  
    STOPROW => 'row5'}  
hbase> scan 'movie', {COLUMNS =>  
    ['desc:title', 'media:type']}
```

Removing Rows

- Delete a single column descriptor with **delete**
 - Adding a timestamp deletes that version of the column and all previous versions

```
hbase> delete 'tablename', 'rowkey', 'colfam:col'  
[,timestamp]
```

- Delete the entire row with **deleteall**

```
hbase> deleteall 'movie', 'row1'
```

- Delete all rows in the table with **truncate**

```
hbase> truncate 'movie'
```

Deleting Data

- When a row is deleted, HBase does not actually remove the row, it only marks the row for deletion
 - Actual deletion happens later (more information on this in a subsequent chapter)
- Rows are selected for deletion by specifying the row key
- Deletes can be performed on a particular column family or specific column descriptors
 - In this case, all other data for the row will remain intact

What Really Happens When You Alter Data

- **Put can be used to alter existing data**
- **Put always creates a new version of a cell and assigns a timestamp to it**
- **By default the system uses the server's currentTimeMillis as the timestamp**
- **When you retrieve a row, the most recent version (the version with the largest timestamp) will be returned**
- **You can override the system's timestamp with your own**
 - You can assign a time in the past or in the future
 - You can choose to assign a value that represents something other than time
- **To overwrite an existing value, perform a Put to exactly the same row, column, and version as the cell you would overwrite**

Versions

- By default, HBase keeps one version of each cell
 - The number of versions retained is configurable on a per-Column Family basis
- For example, if you specify three versions be retained, three rows are kept
 - Rows are sorted by their timestamp (in descending order)

Sorted in
descending
Timestamp
order

Row Key	Column Key	Timestamp	Cell Value
jsmith	profilephoto:image	1371851677671	<work.jpg>
jsmith	profilephoto:image	930001926438	<college.jpg>
jsmith	profilephoto:image	866929926351	<highschool.jpg>

Chapter Topics

HBase Shell

- Creating Tables with the HBase Shell
- Working with Tables
- Hands-On Exercise: Using the HBase Shell
- Working with Table Data
- **Hands-On Exercise: Data Access in the HBase Shell**
- Conclusion

Hands-On Exercise: Data Access in the HBase Shell

- In this Hands-On Exercise, you will use the HBase Shell to put, get, and delete rows
- Please refer to the Exercise Manual

Chapter Topics

HBase Shell

- Creating Tables with the HBase Shell
- Working with Tables
- Hands-On Exercise: Using the HBase Shell
- Working with Table Data
- Modifying Data in an HBase Table
- Hands-On Exercise: Data Access in the HBase Shell
- **Conclusion**

Key Points

- Every row has a single row key
- The HBase shell is a simple way to work with HBase tables and data



HBase Architecture Fundamentals

Chapter 5



Course Chapters

- Introduction
- Introduction to Hadoop and HBase
- HBase Tables
- HBase Shell
- **HBase Architecture Fundamentals**
- HBase Schema Design
- Basic Data Access with the HBase API
- More Advanced HBase API Features
- HBase on the Cluster
- HBase Reads and Writes
- HBase Performance Tuning
- HBase Administration and Cluster Management
- HBase Replication and Backup
- Using Hive and Impala with HBase
- Conclusion
- Appendix: Using Python and Thrift to Access HBase Data
- Appendix: OpenTSDB

HBase Architecture Fundamentals

In this chapter you will learn

- **the major components of the HBase architecture**
- **The role of each HBase component**
- **HBase and Data Locality**

Chapter Topics

HBase Architecture Fundamentals

- **HBase Regions**
- HBase Cluster Architecture
- HBase and HDFS Data Locality
- Conclusion

HBase Regions

- HBase tables are split into *Regions*
 - Sections of the table
 - Similar to sharding or partitioning in a traditional RDBMS
- Regions are served to clients by *RegionServer* daemons

HBase RegionServers

- A RegionServer usually runs on each slave node in the cluster
- A RegionServer will typically serve multiple regions
 - The regions it serves may belong to a number of different tables
 - It is very unlikely that one RegionServer will serve all regions for a particular table

Regions

- Tables are broken into smaller pieces called *regions*
- A region contains a series of rows spanning from the start row key to the end row key defined for that region
- A region is served by a RegionServer

Row key	Users Table	
aaronsona	fname: Aaron	Iname: Aaronson
harrise	fname: Ernest	Iname: Harris
jordena	fname: Adam	Iname: Jorden
laytonb	fname: Bennie	Iname: Layton
millerb	fname: Billie	Iname: Miller
nununezw	fname: Willam	Iname: Nunez
rossw	fname: William	Iname: Ross
sperberp	fname: Phyllis	Iname: Sperber
turnerb	fname: Brian	Iname: Turner
walkerm	fname: Martin	Iname: Walker
zykowskiz	fname: Zeph	Iname: Zykowski

Users Table Divided into Regions

Row key	Users Table – Region 1	
aaronsona	fname: Aaron	Iname: Aaronson
harrise	fname: Ernest	Iname: Harris
Row key	Users Table – Region 2	
jordena	fname: Adam	Iname: Jorden
laytonb	fname: Bonnie	Iname: Layton
Row key	Users Table – Region 3	
millerb	fname: Billie	Iname: Miller
nununezw	fname: William	Iname: Nuñez
Row key	Users Table – Region 4	
rossw	sperberp	fname: Phyllis Iname: Sperber
	turnerb	fname: Brian Iname: Turner
	walkerm	fname: Martin Iname: Walker
	zykowskiz	fname: Zeph Iname: Zykowski

Regions Served by RegionServers

Row key	Users Table – Region 1	
aaronsona	fname: Aaron	Iname: Aaronson
RegionServer 1	Row key	
harrisde	sperberp	fname: Phyllis Iname: Sperber
	turnerb	fname: Brian Iname: Turner
	walkerm	fname: Martin Iname: Walker
	zykowskiz	fname: Zeph Iname: Zykowski

Row key	Users Table – Region 2	
jordena	fname: Adam	Iname: Jordan
RegionServer 2	Row key	
laytonb	millerb	fname: Billie Iname: Miller
	nununezw	fname: Willam Iname: Nunez
	rossw	fname: William Iname: Ross

Chapter Topics

HBase Architecture Fundamentals

- HBase Regions
- **HBase Cluster Architecture**
- HBase and HDFS Data Locality
- Conclusion

Major Daemons in an HBase Cluster

- **RegionServer**

- Responsible for serving and managing regions

- **Master**

- Monitors all RegionServer instances in the cluster
 - Interface for all metadata changes

- **ZooKeeper**

- A centralized service used to maintain configuration information for HBase

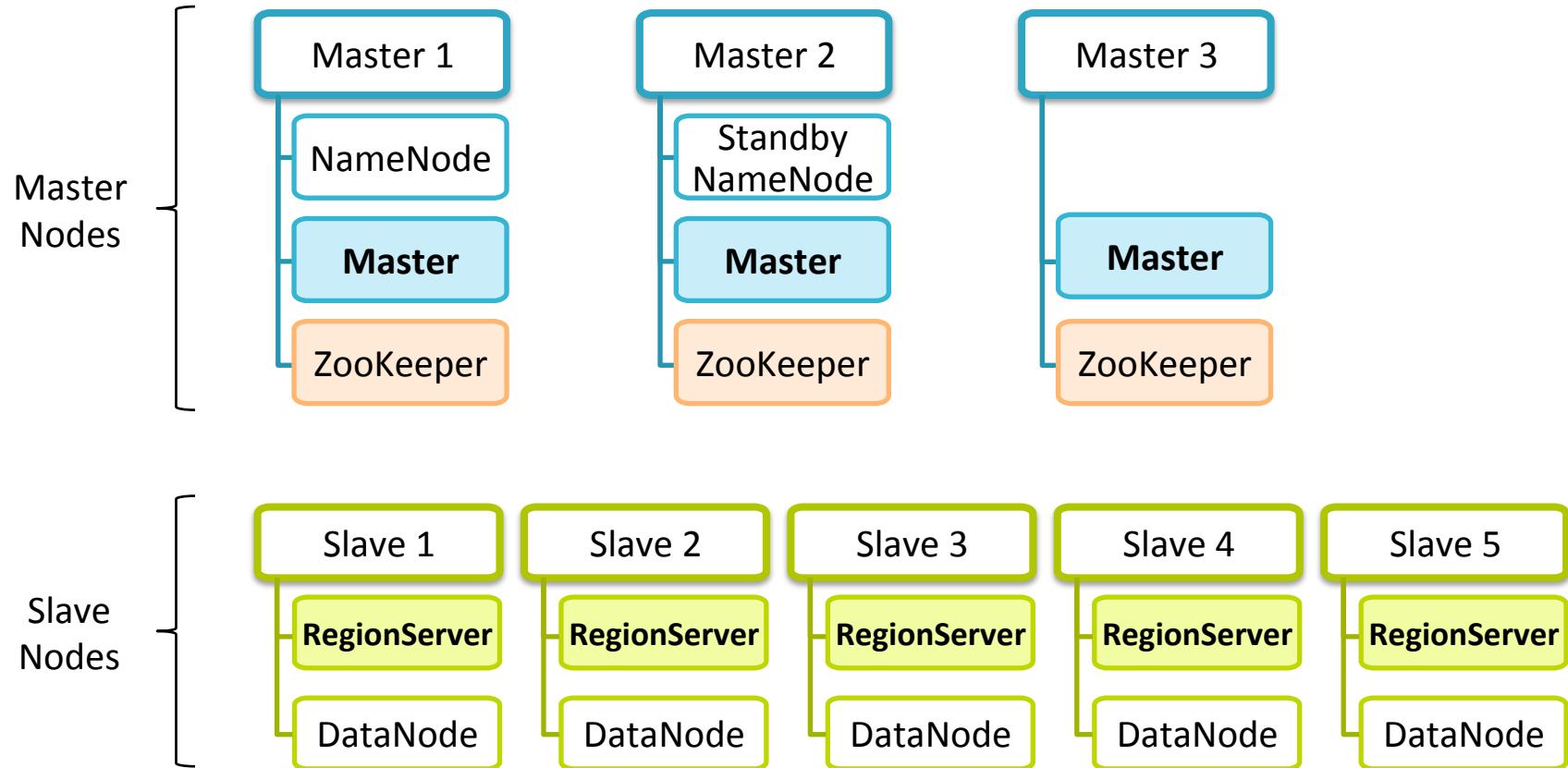
- **NameNode**

- Keeps track of HDFS metadata

- **DataNode**

- Keeps track of HDFS blocks

Placement of Daemons and Services in a Typical HBase Cluster



The HBase Master

- **HBase Master is a daemon which coordinates the RegionServers**
 - Determines which Regions are managed by each RegionServer
 - These assignments will change as data gets added or deleted
 - Handles new table creation and other housekeeping operations

HBase and ZooKeeper

- **An HBase cluster can have multiple Masters for high availability**
 - Only one Master controls the cluster
 - The ZooKeeper service handles coordination of the Masters
- **The ZooKeeper service runs on master nodes on the cluster**
 - Upon startup all Masters connect to ZooKeeper
 - They compete to run the cluster
 - The first Master to connect ‘wins’ control
 - If the controlling Master fails, the remaining Masters will compete again to run the cluster

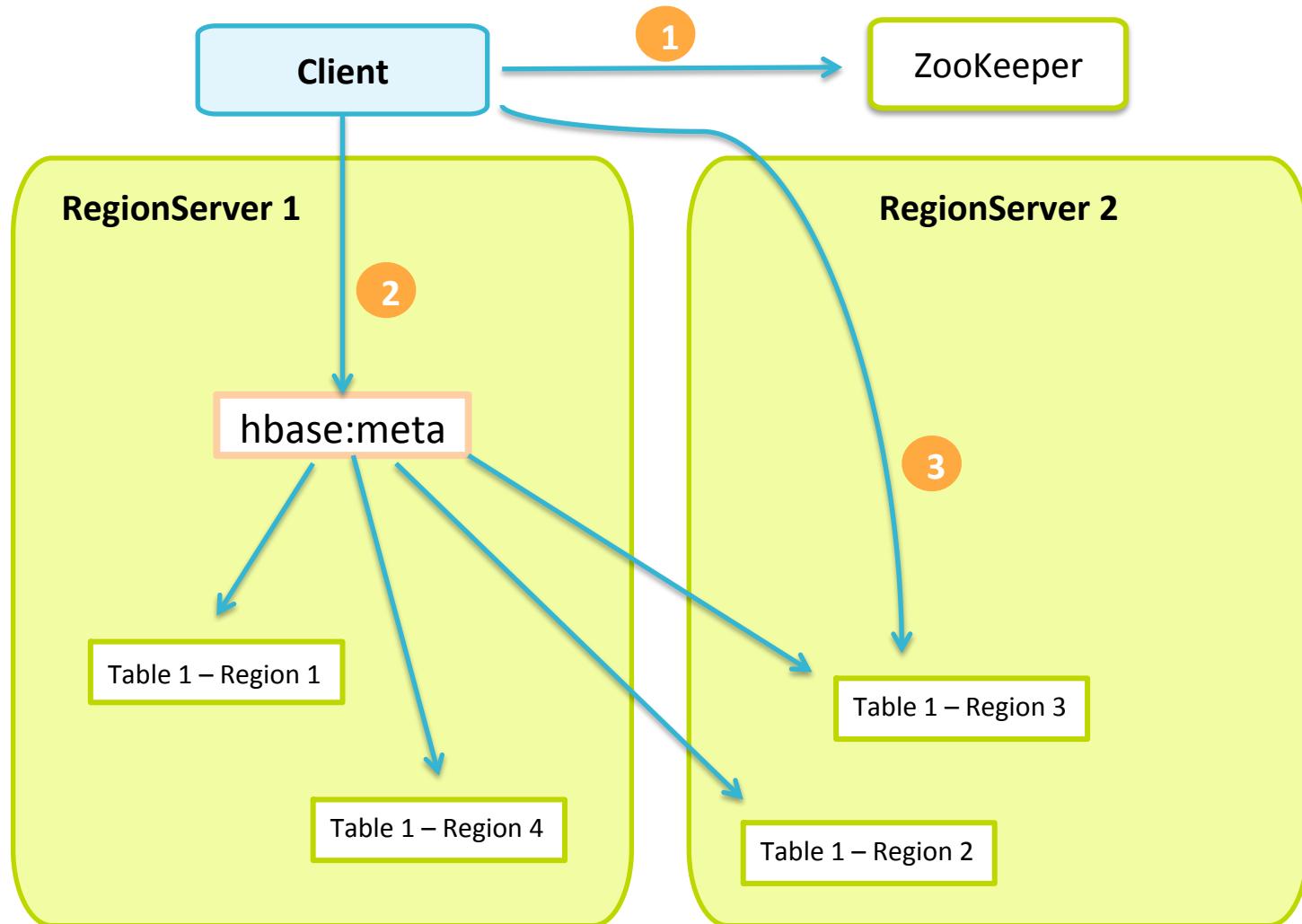
Table Types

- **Userspace tables**
 - HBase tables created with the HBase API or HBase shell
 - The `Users` table in the previous section is a userspace table
- **The catalog table, `hbase:meta`**
 - Special table used and accessed only by HBase
 - Keeps track of the locations of RegionServers and regions
 - `hbase:meta` is an HBase table but it is filtered out of the HBase shell's `list` command
- **The location of the `hbase:meta` table is found through a lookup in ZooKeeper**

The hbase:meta Table

- **The first query from a client is to ZooKeeper to find the location of hbase:meta**
- **The second query is to hbase:meta**
 - hbase:meta lists all regions and their locations
 - The hbase:meta table is never split into regions
- **The third query is to the RegionServer where the region for the data is held**
- **Results of the first two queries are cached by the client**

Querying for Regions



Monitoring the Cluster with Hadoop and HBase Web UIs

- **All Hadoop daemons contain a Web server**
 - Exposes information over a well-known port
 - The type of information and content are specific to the daemon
- **Some important ones for HBase are:**
 - HBase Master `http://<master_address>:60010`
 - RegionServer `http://<regionserver_address>:60030`
- **Looking at the NameNode is useful for HBase**
 - NameNode `http://<namenode_address>:50070`

Diagnosing Problems Using Log Files

- All log files are written to `/var/log/hbase` by default
- Log files can be viewed using the daemons' Web interfaces
 - The Web interfaces can be used to dynamically set the logging level
- Many issues can be diagnosed using the logs
 - Thrift and REST errors are only logged in the log file and are not always sent to the client

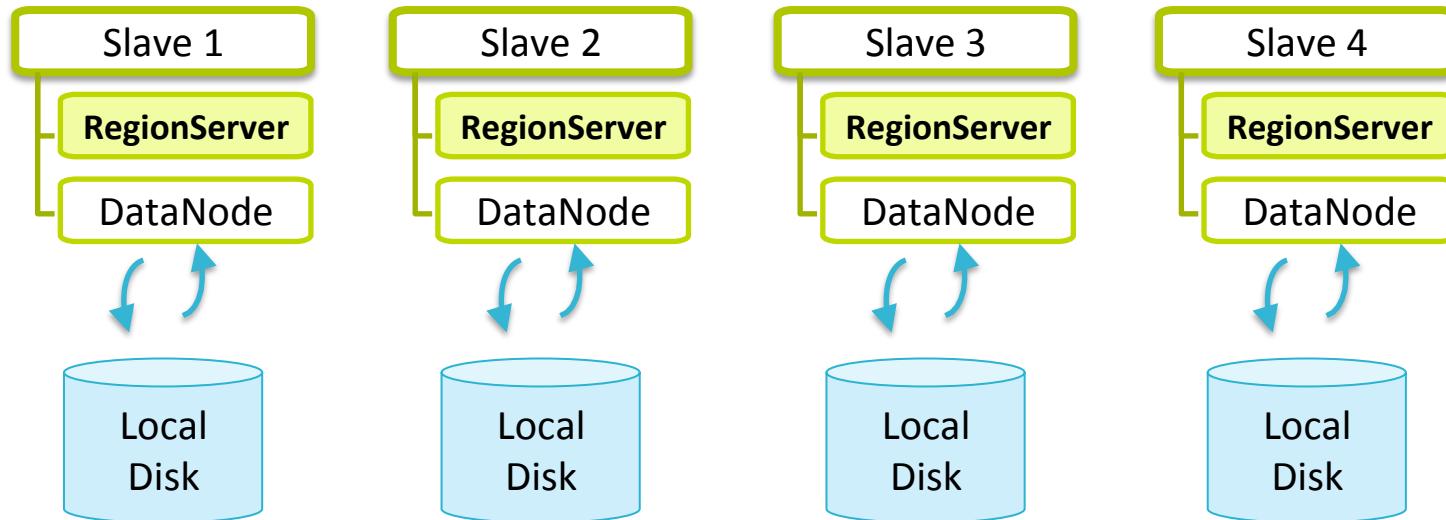
Chapter Topics

HBase Architecture Fundamentals

- HBase Regions
- HBase Cluster Architecture
- **HBase and HDFS Data Locality**
- Conclusion

HBase and HDFS

- **The HBase RegionServer writes data to HDFS on its local disk**
 - HDFS will replicate the data to other nodes in the cluster
 - Replication ensures the data remains available even if a node fails



Chapter Topics

HBase Tables

- HBase Regions
- HBase Cluster Architecture
- HBase and HDFS Data Locality
- **Conclusion**

Key Points

- Tables are broken up into regions by row key
- HBase tables are split into regions and served by RegionServers
- RegionServers are coordinated by the HBase Master
- HDFS and ZooKeeper provide high availability for HBase
- The `hbase:meta` table is used to figure out which RegionServer is serving a region based on the row key



HBase Schema Design

Chapter 6



Course Chapters

- Introduction
- Introduction to Hadoop and HBase
- HBase Tables
- HBase Shell
- HBase Architecture Fundamentals
- **HBase Schema Design**
- Basic Data Access with the HBase API
- More Advanced HBase API Features
- HBase on the Cluster
- HBase Reads and Writes
- HBase Performance Tuning
- HBase Installation and Basic Administration
- HBase Cluster Management
- HBase Replication and Backup
- Using Hive and Impala with HBase
- Conclusion
- Appendix: Using Python and Thrift to Access HBase Data
- Appendix: OpenTSDB

HBase Schema Design

In this chapter you will learn

- What to consider when designing an HBase schema
- How to deal with, and avoid, hotspotting
- What issues to consider for row key design

Chapter Topics

HBase Schema Design

- **General Design Considerations**
- Application-Centric Design
- Designing HBase Row Keys
- Other HBase Table Features
- Hands-On Exercise: Using MIN VERSIONS and Time-To-Live
- Conclusion

What is an HBase Schema?

- An HBase schema includes all of the following considerations:
 - How many column families should the table have?
 - Which data should be assigned to which column family?
 - How many columns should be assigned to each column family?
 - How should the columns be named?
 - What type of information should be stored in the cells?
 - How many versions of each cell do we require?
 - What should the row key structure be?

HBase vs. RDBMS (1)

- **HBase is not a relational database**
- **Schema design in databases generally favors normalization**
 - A normalized layout has little to no redundant data
- **RDBMSs generally have lots of smaller tables**
 - These tables are normalized to facilitate joins
- **HBase typically has only a few, large tables**
 - These tables are denormalized to avoid joins

HBase vs. RDBMS (2)

- **HBase tables have a small number of column families**
 - Within each column family you may have hundreds or thousands of columns
 - RDBMS tables typically have far fewer columns in a table
- **HBase only has a single row key**
 - RDBMSs can have as many indexed columns as required
- **Schema design for HBase is much different than for an RDBMS**
 - Design and layout of data is different
 - More effort goes into the row key planning for HBase
- **Retrofitting HBase into existing code is not a trivial task**

Chapter Topics

HBase Schema Design

- General Design Considerations
- **Application-Centric Design**
- Designing HBase Row Keys
- Other HBase Table Features
- Hands-On Exercise: Using MIN VERSIONS and Time-To-Live
- Conclusion

Application-Centric, not Data-Centric

- **When designing for an RDBMS, examine the relationships in the data**
 - Break up data into tables and columns
 - Create relationships between tables, e.g., 1 to 1, 1 to many, and many to many
- **When designing for HBase, consider how the application(s) will access the data**
 - What information is available when performing a get?
 - Which pieces of information will be accessed together frequently?
 - Will the load be spread evenly across RegionServers?

Thinking About Access Patterns is Important

- **The access pattern is an integral part of designing for HBase**
 - How an application accesses data will inform the design
 - Identify the type of data that is being accessed
- **Most applications can be characterized as either read-heavy or write-heavy**
 - Some applications accessing a particular table may be read-heavy while others are write-heavy
- **The access pattern is further defined by whether an application will modify existing data**
 - Write-heavy does not mean that the same piece of data is being overwritten
 - Some applications are write-heavy and do not change existing data

Read-Heavy Access Patterns

- **A read-heavy application needs to be optimized for efficient reading**
- **Avoid joining tables**
 - Place all of the data that the application needs in a single row or in a set of contiguous rows
 - Denormalize one to many relationships by pre-materializing them
- **With denormalized data, multiple copies of a data item may appear in many rows**
 - Any updates must be applied to all copies of the data item
- **Optimize the row key and column family layouts**
 - Row keys need to allow for quick reading mainly via gets
 - Column families must be optimized to use the Block Cache efficiently

Write Heavy Access Patterns

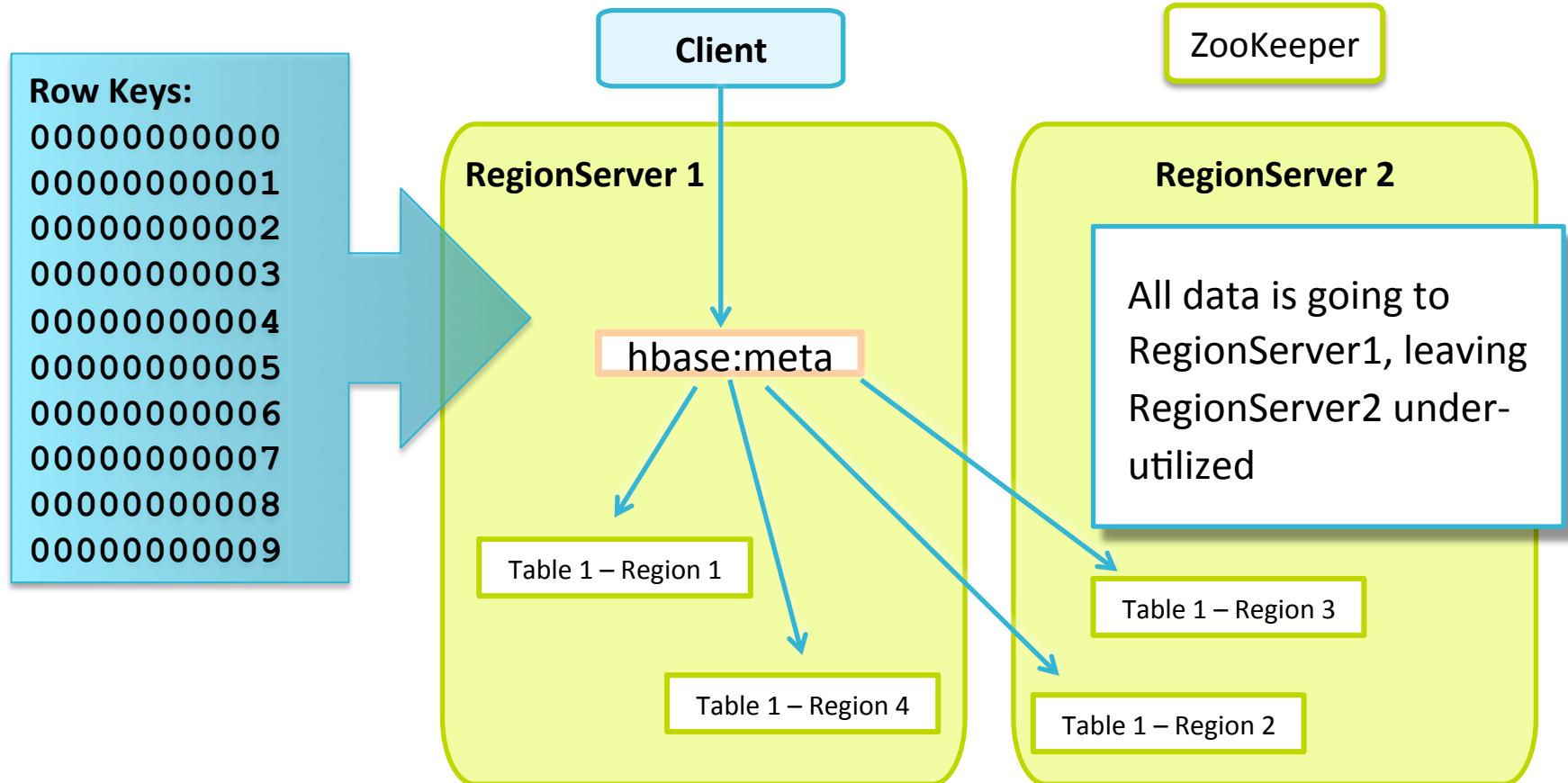
- A write-heavy application needs to be optimized to write as fast as possible
- Optimize the row key to spread the load evenly across RegionServers
 - Using only one RegionServer's resources is inefficient
- Normalized tables may result in faster updates
 - Denormalized data may require many rows to be updated, rather than just a single row
 - Read performance must also be considered when making this decision, since a join must be performed if data is normalized

Hybrid Access Patterns

- An application or group of applications accessing a table may be both read- and write-heavy
- Consider what the cluster or table is primarily used for
 - Is the table primarily used by one application and used less often by another application?
 - Which has a heavier load, the read or the write?
 - Is the write workload updating or adding new rows?
 - Can you optimize certain settings to improve the writes or the reads?

Time Series and Sequential Data Types

- Time series and sequential row keys prevent even RegionServer loading



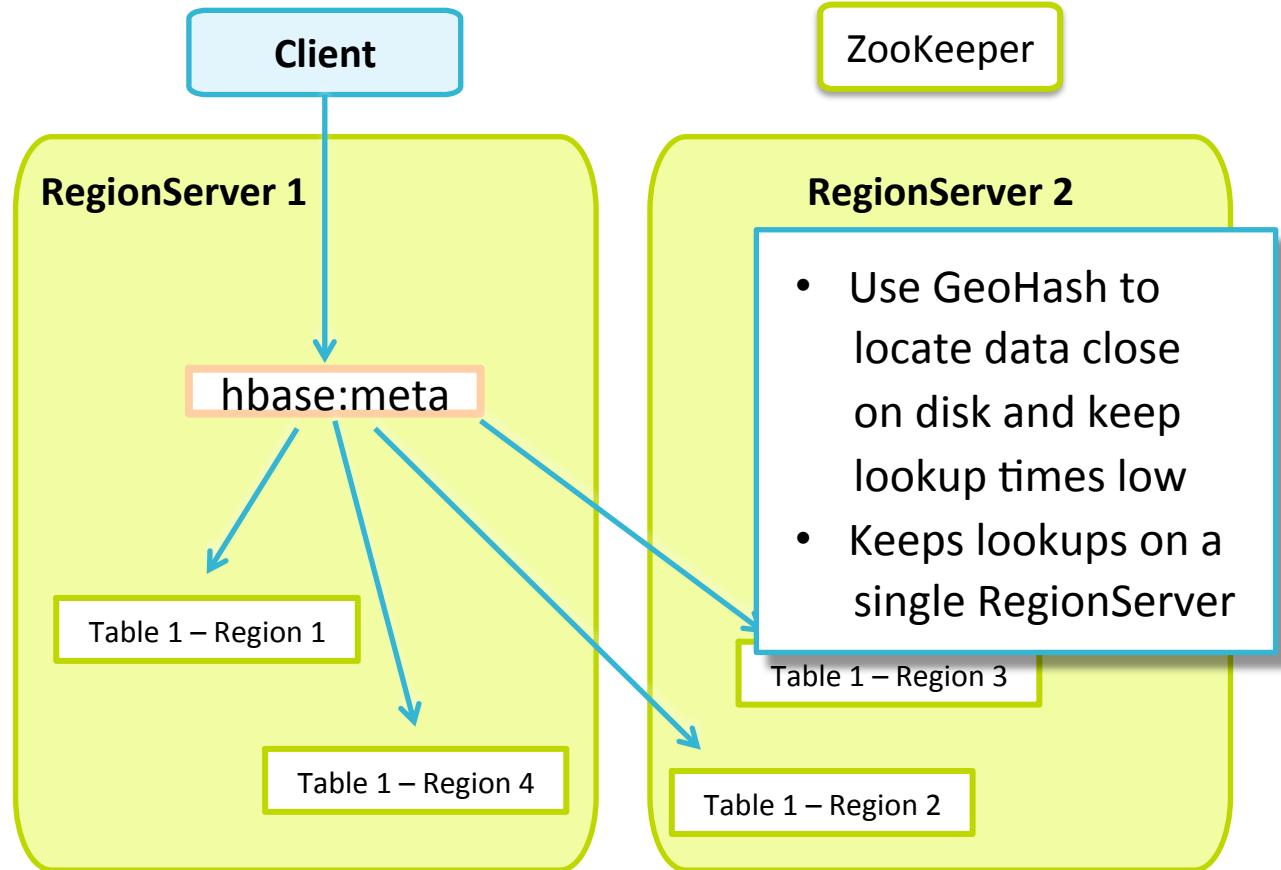
Geospatial Data Types

- Geospatial data requires row keys that allow lookups based on location

Latitude and longitude:

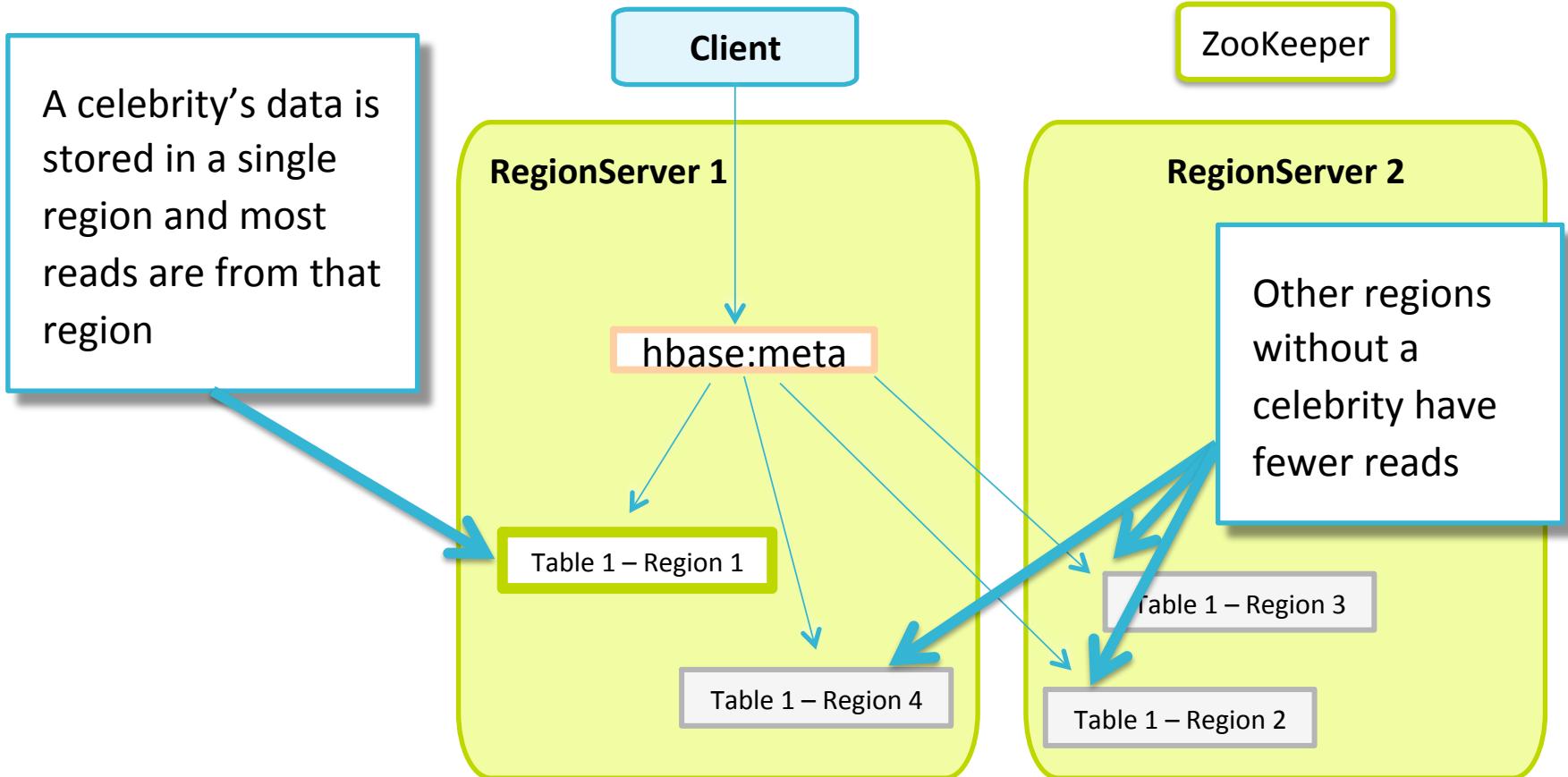
39.5272° N,
119.8219° W

Query:
What is the nearest coffee shop to my latitude and longitude?



Social Data Types

- Social data access patterns are different for celebrities vs. non-celebrities



Chapter Topics

HBase Schema Design

- General Design Considerations
- Application-Centric Design
- **Designing HBase Row Keys**
- Other HBase Table Features
- Hands-On Exercise: Using MIN VERSIONS and Time-To-Live
- Conclusion

Row Keys

- **HBase row keys are often composites**
 - The data in the key is made up of several different pieces of data
 - For example, instead of just <timestamp>, they are often <timestamp><source><event>
 - All pieces of data are combined into a single row key
 - Each piece of data allows more granularity for scans

Row Key Design

- **Row keys cannot be changed**
 - A row must be deleted and then re-inserted with the new key
- **Rows are sorted as they are put into the table**
 - Nothing is sorted during an operation like a scan
- **Keys are ordered lexicographically**
 - E.g., 1, 10, 100, 11, 12, 13 . . . 2, 20, 21, . . .
 - You must left pad with zeroes to preserve natural ordering if using numbers for your row key
- **Selecting the appropriate row keys for your application is critical for performance**

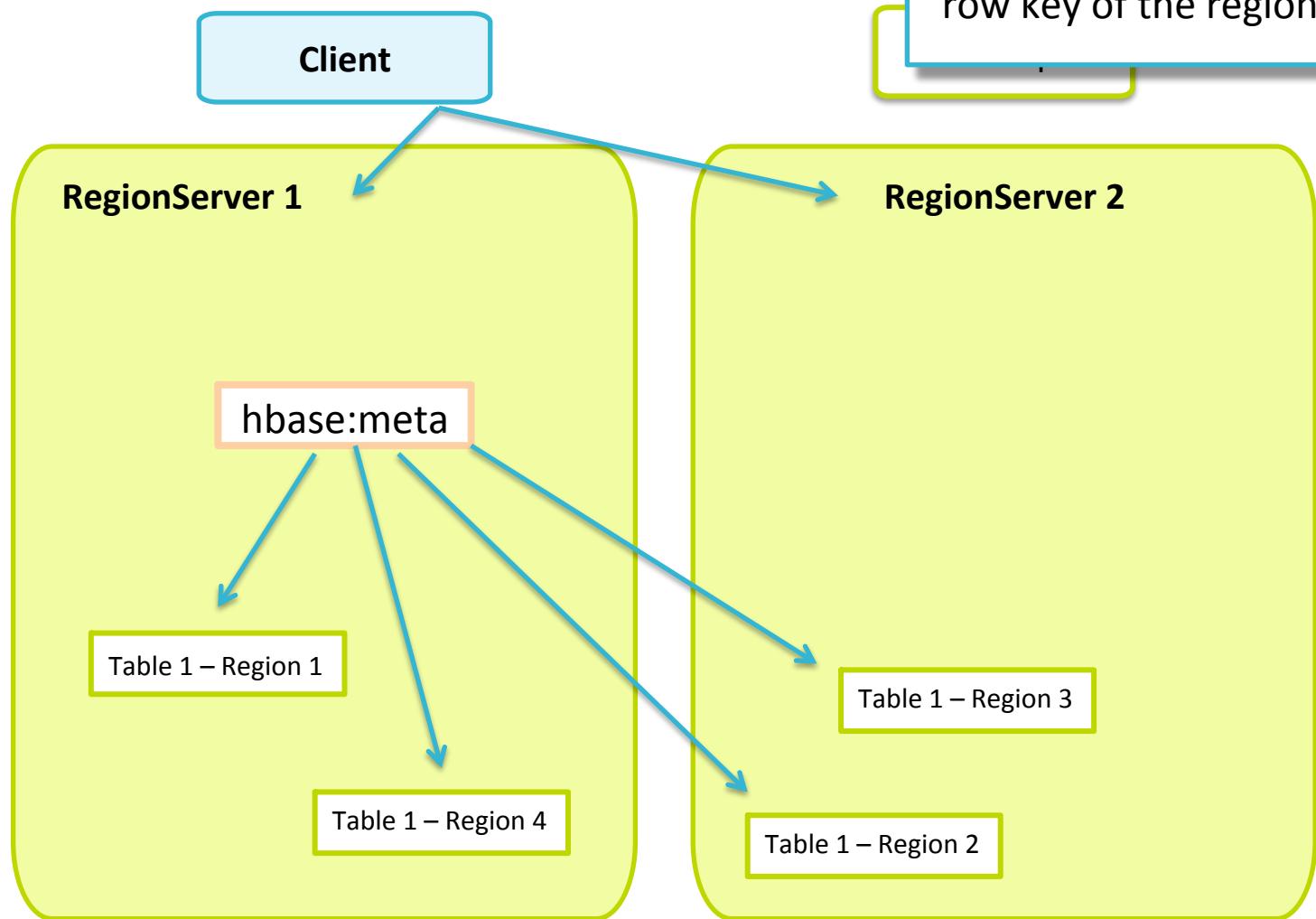
Row Key Performance

- **Querying based on row keys has the highest performance**
 - Row key queries have the least amount of processing to perform
 - Timerange-bound and column family reads can skip store files
- **Querying on column values has the lowest performance**
 - Value-based filtering is a full table scan
 - Each column's value must be checked during a scan
- **Typically, commonly queried data is added to the row key in order to achieve the fastest access**

Row Keys for Easy Data Retrieval

- **Each region serves a portion of the row keys**
 - Region 1, for example, will serve a portion of the rows for a given table
- **A given RegionServer manages and serves several regions**
 - Heavy usage on even one region causes higher latency for other regions served by the same RegionServer
 - We can counter the high latency by moving a highly used region to another RegionServer

Read and Write Paths



Client issues calls to a RegionServer based on the row key of the region.

Potential Hot Spot Problems

- **Hotspotting occurs when a small number of RegionServers are handling the majority of the load**
 - This causes an uneven use of the cluster's resources
- **Hotspotting can happen for different reasons**
 - An automatic or pre-split region is not optimal
 - The row key is sequential or time series
 - The regions for a table are not distributed around the cluster efficiently

Sequential Row Key Type

■ Sequential

- Incremental id or time series
- e.g., <timestamp> or <incrementalid>
- Best scan performance for reading one row after another
- Worst for write performance because all writes are likely to hit one RegionServer

Timestamp Row Key:

20130704-0130
20130704-0134
20130704-0246
20130704-0252
20130704-0414
20130704-0533
20130704-0721
20130704-0929

Incremental Row Key:

0000000000
0000000001
0000000002
0000000003
0000000004
0000000005
0000000006
0000000007

Salted Row Key Type

- **Salted**

- Place a small, calculated hash in front of the real data to randomize the row key
- e.g., <salt><timestamp> instead of just <timestamp>
- Still allows scanning by ignoring the salt
- Improves the write performance because the salt prefix distributes the writes across multiple RegionServers

Original Row Key:

0000000000
00000000001
00000000002
00000000003
00000000004
00000000005
00000000006
00000000007



Salted Row Key:

0 : 00000000000
1 : 00000000001
2 : 00000000002
3 : 00000000003
4 : 00000000004
5 : 00000000005
6 : 00000000006
7 : 00000000007

Promoted Field Row Key Type

■ Promoted Field Keys

- A field is moved in front of the incremental or timestamp field
- E.g. <sourceid><timestamp> instead of <timestamp><sourceid>
- Still allows scanning by ignoring or using the promoted field
- Improves the write performance because the promoted field prefix distributes the writes across multiple RegionServers

Original Row Key:

0000000000:775
0000000001:314
0000000002:314
0000000003:310
0000000004:916
0000000005:925
0000000006:775



Promoted Row Key:

775:0000000000
314:0000000001
314:0000000002
310:0000000003
916:0000000004
925:0000000005
775:0000000006

Random Row Key Type

■ Random

- Process the data using a one-way hash like MD5
- E.g. <md5(timestamp)> instead of just <timestamp>
- Worst read performance because all values will need to be read
- Best write performance because the randomized row key distributes the writes evenly across all RegionServers
- Note: Strictly speaking, the row key is not random, since it is derived from the original data

Original Row Key:

0000000000

0000000001

0000000002

0000000003

0000000004

0000000005

0000000006



Random MD5 Hashed Row Key:

645a8aca5a5b84527c57ee2f153f1946

d67f0826d4c0aa7e3ea5861616a822b2

c93c5cedf7fba468e0fe2c845837abc7

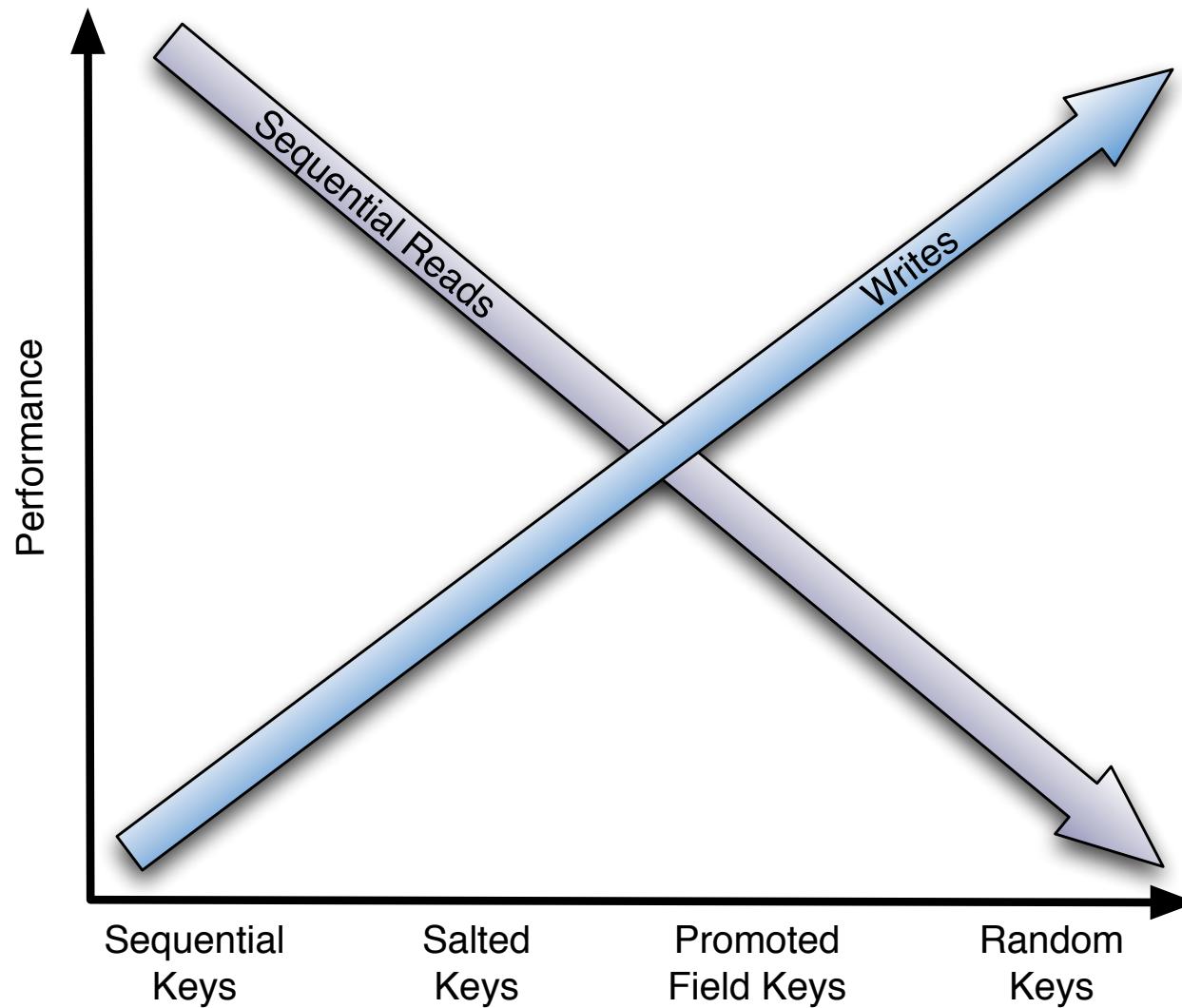
6a1ae0e285acaf40dc30d13b702e6470

e57ea6134fc5278023292f1941dff865

63b307e583982c0746a5617e94f12dca

0d51268ce5ae7eed7e1cccd6d3859d033

Key Design Tradeoff



Partial Key Scans: Movie Example

Key	Description
<year>	Scan over all rows for a given year (e.g., "1997", "1979", etc.)
<year><genre>	Scan over all rows for a given year for the given genre
<year><genre><subgenre>	Scan over all rows for a given year, a given genre with a given subgenre
<year><genre><subgenre><moviename>	Scan over all rows for a given year, a given genre with a given subgenre with the movie name

Time Series or Incremental Data

- **Monotonically increasing row keys such as time series or incremental values**
- **Options for using such data depend on read and write use cases**
 - If gets are always random and never scanned in order
 - Use random row keys
 - Salting and promoted keys are an option where scans are used but write speed is still a problem
 - Use bulk import for sequential keys to keep write speeds from being an issue

Reverse Timestamps

- **How to quickly find the most recent version of a value?**
 - Append the reverse timestamp to the row key
 - Same groups will be located together
 - Within each group, rows will be sorted so the most recent insert will be located at the top
 - Example:

```
<group_id><Long.MAX_VALUE - System.currentTimeMillis()>
```

Chapter Topics

HBase Schema Design

- General Design Considerations
- Application-Centric Design
- Designing HBase Row Keys
- **Other HBase Table Features**
- Hands-On Exercise: Using MIN VERSIONS and Time-To-Live
- Conclusion

VERSIONS

- **By default, only one version of a cell is retained**
 - HBase automatically adds a timestamp to each version of a cell
 - Only the most recently committed cell(s) are retained
- **The maximum number of versions to retain is configurable**
 - The VERSIONS property can be configured on a per column family basis

MIN VERSIONS

- You can configure MIN VERSIONS to specify the minimum number of versions of a cell to retain
 - By default, MIN VERSIONS is set to zero, thus disabling the feature
 - Any other value for MIN VERSIONS will cause that number of versions to be retained

Time-To-Live

- **MIN VERSIONS** is used in conjunction with Time-To-Live (TTL)
- The TTL attribute is the time-to-live for a particular cell and is used as an expiration value
 - A row is kept until a user deletes it, but an expired row will be deleted automatically
 - TTL is given in seconds and is configured on a per-column family basis
 - The default value of TTL is FOREVER

Chapter Topics

HBase Schema Design

- General Design Considerations
- Application-Centric Design
- Designing HBase Row Keys
- Other HBase Table Features
- **Hands-On Exercise: Using MIN VERSIONS and Time-To-Live**
- Conclusion

Hands-On Exercise: Using MIN VERSIONS and Time-To-Live

- In this Hands-On Exercise, you will learn how to use the MIN VERSIONS and Time-To-Live features for managing data retention.
- Please refer to the Exercise Manual

Chapter Topics

HBase Schema Design

- General Design Considerations
- Application-Centric Design
- Designing HBase Row Keys
- Other HBase Table Features
- Hands-On Exercise: Using MIN VERSIONS and Time-To-Live
- **Conclusion**

Key Points

- HBase is not a relational database
- Schema design takes into consideration all aspects of a table's design
- HBase favors denormalization to avoid the expense of joins
- HBase development is application-centric not data-centric
- A lot of thought should go into the row key design
- Hotspotting can affect performance



Basic Data Access with the HBase API

Chapter 7



Course Chapters

- Introduction
- Introduction to Hadoop and HBase
- HBase Tables
- HBase Shell
- HBase Architecture Fundamentals
- HBase Schema Design
- **Basic Data Access with the HBase API**
- More Advanced HBase API Features
- HBase on the Cluster
- HBase Reads and Writes
- HBase Performance Tuning
- HBase Administration and Cluster Management
- HBase Replication and Backup
- Using Hive and Impala with HBase
- Conclusion
- Appendix: Using Python and Thrift to Access HBase Data
- Appendix: OpenTSDB

Basic Data Access with the HBase API

In this chapter you will learn

- How to access data with the Java HBase API
- How to use the Java API for administration
- Basic HBase administration calls
- How to add and update data with the API
- How to use the Scan API

Chapter Topics

Basic Data Access with the HBase API

- **Options to Access HBase Data**
- Creating and Deleting HBase Tables
- Retrieving Data with Get
- Retrieving Data with Scan
- Inserting and Updating Data
- Deleting Data
- Hands-On Exercise: Using the Developer API
- Conclusion

Accessing Data in HBase

- There are several different ways of accessing data in HBase depending on your programming language and use case
- The Java API is the only ‘first class citizen’ for HBase
 - It is the preferred method for accessing HBase
- For non-Java languages there are the Thrift and REST interfaces
 - The Thrift interface is the preferred method for non-Java access in HBase
 - The REST interface allows data access using HTTP calls
- The HBase Shell can also be used to access data
 - This can be used for smaller, ad hoc queries

HBase Java API

- **Administrative HBase tasks are commonly performed using the HBase shell, but a Java API is also available**
- **The Java API is the only first class citizen, other languages are supported through ecosystem projects**
 - Some ecosystem projects augment the Java API with new features
- **Using the HBase API in a Java program is easy**
 - Simply add the HBase Jars to the classpath
 - Instantiate the HBase objects just like any other object

HBase and Byte Arrays

- Many HBase Java API methods require a byte array as the value
- HBase has a utility class called **Bytes** to help convert data to and from a byte array
 - Has various `Bytes.to*` methods that convert primitives and Strings to a byte array and vice versa

```
byte[] stringArray = Bytes.toBytes("somestring");
byte[] intArray   = Bytes.toBytes(1337);
byte[] doubleArray = Bytes.toBytes(3.14159D);
```

- Converting from byte arrays back to original type:

```
String someString = Bytes.toString(stringArray);
int    myInt     = Bytes.toInt(intArray);
double pi        = Bytes.toDouble(doubleArray);
```

Simple Examples Using the HBase Java API (1)

- The admin object provides access to the administration calls for HBase
- List all tables in HBase, including detailed information for each table

```
HTableDescriptor[] descriptors = admin.listTables();
```

- Get detailed information for a specific table
 - View all column families in table, their properties, and values
 - Use the HTableDescriptor object to access all information about table settings

```
HTableDescriptor descriptor =
    admin.getTableDescriptor(TableName.valueOf("movie"));
```

Simple Examples Using the HBase Java API (2)

- **Disable a table to put it in the maintenance state**

- Allows various maintenance commands to be run
- Prevents all client access
- May take up to several minutes for a table to disable

```
admin.disableTable(TableName.valueOf("movie"));
```

- **Enable a table to take it out of maintenance state**

```
admin.enableTable(TableName.valueOf("movie"));
```

Chapter Topics

Basic Data Access with the HBase API

- Options to Access HBase Data
- **Creating and Deleting HBase Tables**
- Retrieving Data with Get
- Retrieving Data with Scan
- Inserting and Updating Data
- Deleting Data
- Hands-On Exercise: Using the Developer API
- Conclusion

Using the Administration API to Create a Table: Complete Code

```
Configuration configuration =
HBaseConfiguration.create();
HBaseAdmin admin = new HBaseAdmin(configuration);

HTableDescriptor descriptor = new
    HTableDescriptor(TableName.valueOf("movie"));
HColumnDescriptor columnDescriptor = new
    HColumnDescriptor(Bytes.toBytes("desc"));

descriptor.addFamily(columnDescriptor);
admin.createTable(descriptor);

admin.close();
```

Using the Administration API to Create a Table: Initial Steps

```
Configuration configuration =  
HBaseConfiguration.create();  
HBaseAdmin admin = new HBaseAdmin(configuration);
```

HTable table = null;
HColumnDescriptor columnDescriptor = null;
admin.createTable(table, columnDescriptor);
admin.close();

- Create an instance of the Configuration object
- This object will read in the Hadoop and HBase configuration files
- It will be passed into the HBaseAdmin class as a parameter.
- The HBaseAdmin class provides access to the administration calls for HBase

Using the Administration API to Create a Table: Descriptors

```
Configuration configuration =  
HBaseConfiguration.create();  
HBaseAdmin admin = new HBaseAdmin(configuration);  
  
HTableDescriptor descriptor = new  
    HTableDescriptor(TableName.valueOf("movie"));  
HColumnDescriptor columnDescriptor = new  
    HColumnDescriptor(Bytes.toBytes("desc"));  
  
descriptor.addFamily(columnDescriptor);  
admin.createTable(descriptor);  
  
admin.close();
```

- The `HTableDescriptor` object contains the description of the table to be created. This includes the table name and other parameters
- The `HColumnDescriptor` object contains the specification of the column family, including any names and parameters

Using the Administration API to Create a Table: Creating the Table

```
Configuration configuration =
```

```
HBaseConfiguration.createDefaultConfiguration();
```

```
HBaseAdmin admin =
```

```
HTableDescriptor descriptor =
```

- The HColumnDescriptor gets added to the HTableDescriptor object, which adds a column family to the table being created
- createTable is called to create the table in HBase
- The HBaseAdmin connection must then be closed

```
descriptor.addFamily(columnDescriptor);
```

```
admin.createTable(descriptor);
```

```
admin.close();
```

Using the Administration API to Delete a Table: Code Example

```
Configuration configuration =
HBaseConfiguration.create();
HBaseAdmin admin = new HBaseAdmin(configuration);

admin.disableTable(TableName.valueOf("movie"));

Boolean isDisabled =
admin.isTableDisabled(TableName.valueOf("movie"));
if (true == isDisabled) {
    admin.deleteTable(TableName.valueOf("movie"));
}
else {
    // Disable failed
}

admin.close();
```

Chapter Topics

Basic Data Access with the HBase API

- Options to Access HBase Data
- Creating and Deleting HBase Tables
- **Retrieving Data with Get**
- Retrieving Data with Scan
- Inserting and Updating Data
- Deleting Data
- Hands-On Exercise: Using the Developer API
- Conclusion

Connecting to HBase (1)

- All connections use the **HTableInterface** class to connect to a table in HBase
 - Each connection only connects to a single table
 - Not thread safe
 - Create a separate instance if using multiple threads
 - Must call `close()` once done with connection
- **HTableInterface** provides all access to data in HBase

Connecting to HBase (2)

- Code to connect to a table and close the connection:

```
HConnection conn =  
    HConnectionManager.createConnection(config);  
  
HTableInterface table =  
conn.getTable(TableName.valueOf("movie"));  
  
...  
  
table.close();  
  
conn.close();
```

Getting Data

- **Data can be accessed in the HBase Shell, via the Java API, through scripting, or using alternate interfaces**
 - Java and alternate interfaces are discussed later in the chapter
- **Get**
 - Used to retrieve a single row
 - Must know the exact row key to retrieve

Retrieving Rows: Complete Code

```
Configuration cfg = HBaseConfiguration.create();
HConnection conn = HConnectionManager.createConnection(cfg);
HTableInterface table =
conn.getTable(TableName.valueOf("movie"));

Get g = new Get(Bytes.toBytes("rowkey1"));
Result r = table.get(g);
String rowKey = Bytes.toString(r.getRow());

byte[] byteArray = r.getValue(Bytes.toBytes("desc"),
Bytes.toBytes("title"));
String columnValue = Bytes.toString(byteArray);

table.close();
conn.close();
```

Retrieving Rows: HTable Initialization

```
Configuration cfg = HBaseConfiguration.create();
HConnection conn = HConnectionManager.createConnection(cfg);
HTableInterface table =
conn.getTable(TableName.valueOf("movie"));
```

```
Get g :  
Result  
String  
byte[]  
    Byt  
String columnValue = Bytes.toString(columnArray),
```

```
table.close();  
conn.close();
```

- The Configuration object needs to be instantiated.
It will read in the Hadoop and HBase configuration files
- The Configuration object is passed into the HConnectionManager class
- We then connect to the table “movie”

Retrieving Rows: Get and Result

```
Configuration cfg = HBaseConfiguration.create();
HConnection conn = HConnectionManager.createConnection(cfg);
HTableInterface table =
conn.getTable(TableName.valueOf("movie"));

Get g = new Get(Bytes.toBytes("rowkey1"));
Result r = table.get(g);
String rowKey = Bytes.toString(r.getRow());
```

by

St

ta

cc

- The Get object needs to be instantiated with the exact name of the row key it will retrieve
- The Get object is passed to the HTableInterface table object
- The get method returns a Result object containing the row's data
- The row key can be retrieved from the Result object

Retrieving Rows: Extracting and Transforming Values

Conf

HCon

HTab

conn

Get

Resu

Stri

- The Result object also gives access to the values in the row
- These values are accessed using the column family and column descriptor
- These values come back as byte arrays that then need to be transformed back into their original type
- When finished, close the table and connection

```
byte[] byteArray = r.getValue(Bytes.toBytes("desc"),
    Bytes.toBytes("title"));
String columnValue = Bytes.toString(byteArray);

table.close();
conn.close();
```

Getting Previous Versions of a Cell in HBase

- Multiple versions of a cell can be accessed on a per-column basis

```
Get g = new Get(Bytes.toBytes("rowkey1"));  
g.setMaxVersions(3);  
  
Result r = table.get(g);  
  
List<Cell> columnVersions = r.getColumnCells  
    (FAMILY_BYTES, COLUMN_BYTES);  
  
for (Cell cell : columnVersions) {  
    String columnValue =  
        Bytes.toString(CellUtil.cloneValue(cell));  
    System.out.println("The value at:" +  
        cell.getTimestamp()  
        + " is " + columnValue);  
}
```

Chapter Topics

Basic Data Access with the HBase API

- Options to Access HBase Data
- Creating and Deleting HBase Tables
- Retrieving Data with Get
- **Retrieving Data with Scan**
- Inserting and Updating Data
- Deleting Data
- Hands-On Exercise: Using the Developer API
- Conclusion

Scans

- **The HBase API supports table scans**
- **Recall that a Scan is useful when the exact row key is not known, or when a group of rows needs to be accessed**
- **Scans can be bounded by a start and stop row key**
 - The start row key is included in the results
 - The stop row is not included in the results

Scanning: Complete Code

```
Scan s = new Scan();
ResultScanner rs = table.getScanner(s);

for (Result r : rs) {
    String rowKey = Bytes.toString(r.getRow());
    byte[] b = r.getValue(FAMILY_BYTES, COLUMN_BYTES);
    String user = Bytes.toString(b);
}

rs.close();
```

Scanning: Scan and ResultScanner

```
Scan s = new Scan();  
ResultScanner rs = table.getScanner(s);  
  
for (Result r : rs) {  
    String id = r.getId();  
    byte[] bytes = r.getValue("info", "content");  
    String content = new String(bytes);  
}  
rs.close();
```

The Scan object is created. It will scan all rows. The scan is executed on the table and a ResultScanner object is returned

Scanning: Iterating

```
Scan s = new Scan();  
ResultScanner rs = table.getScanner(s);  
  
for (Result r : rs) {  
    String rowKey = Bytes.toString(r.getRow());  
    byte[] b = r.getValue(FAMILY_BYTES, COLUMN_BYTES);  
    String user = Bytes.toString(b);  
}  
rs.close();
```

Using a `for` loop, you iterate through all `Result` objects in the `ResultScanner`. Each `Result` can be used to get the values.

Reducing Scan Results Returned

- Scan results can be reduced by specifying start and stop row keys
 - The result returned is start row key *inclusive*, and stop row key *exclusive*

```
Scan s = new Scan();  
String startRowKey = "1";  
String endRowKey = "20";  
s.setStartRow(Bytes.toBytes(startRowKey));  
s.setStopRow(Bytes.toBytes(endRowKey));
```

- Further reduce scan results by specifying the columns required

```
Scan s = new Scan();  
s.addColumn(Bytes.toBytes("info"), Bytes.toBytes("name"));
```

Scanner Caching

- Scan results can be retrieved in batches to improve performance
 - Performance will improve but memory usage will increase

```
Scan s = new Scan();  
s.setCaching(20);
```

Chapter Topics

Basic Data Access with the HBase API

- Options to Access HBase Data
- Creating and Deleting HBase Tables
- Retrieving Data with Get
- Retrieving Data with Scan
- **Inserting and Updating Data**
- Deleting Data
- Hands-On Exercise: Using the Developer API
- Conclusion

Adding and Updating Data

- **Recall: HBase does not distinguish an insert from an update**
- **Put is used to both insert new rows and update existing rows**
 - An insert occurs when performing a Put on a row key that does not yet exist
 - An update of a row occurs when a Put is performed on an existing row
- **Updates can occur on specific column descriptors, leaving the row's other columns unchanged**

Adding Data: Complete Code

```
Configuration cfg = HBaseConfiguration.create();
HConnection conn = HConnectionManager.createConnection(cfg);
HTableInterface table =
conn.getTable(TableName.valueOf("movie"));

Put p = new Put(Bytes.toBytes("rowkey1"));
p.add(FAMILY_BYTES, COLUMN_BYTES, Bytes.toBytes("E.T."));

table.put(p);

table.close();
conn.close();
```

Adding Data: Put Object

```
Configuration cfg = HBaseConfiguration.create();
HConnection conn = HConnectionManager.createConnection(cfg);
HTableInterface table =
conn.getTable(TableName.valueOf("movie"));

Put p = new Put(Bytes.toBytes("rowkey1"));
p.add(FAMILY_BYTES, COLUMN_BYTES, Bytes.toBytes("E.T."));

table.put(p);
table.close();
conn.close();
```

- Instantiate the Put object with the row key that uniquely identifies the row
- Each value that will be stored in HBase needs a separate add call. Each add should specify the column family and column descriptor for that value
- All values must be converted to byte arrays

Adding Data: Finishing Up

```
Configuration cfg = HBaseConfiguration.create();
HConnection conn = HConnectionManager.createConnection(cfg);
HTableInterface table =
conn.getTable(TableName.valueOf("movie"));

Put p = new Put(Bytes.toBytes("rowkey1"));
p.add(FAMILY_BYTES, COLUMN_BYTES, Bytes.toBytes("E.T."));

table.put(p);
```

- The Put object is added to the table with a table.put call
- This adds the row specified in the Put object
- The same table object can be used to do several puts

Chapter Topics

Basic Data Access with the HBase API

- Options to Access HBase Data
- Creating and Deleting HBase Tables
- Retrieving Data with Get
- Retrieving Data with Scan
- Inserting and Updating Data
- **Deleting Data**
- Hands-On Exercise: Using the Developer API
- Conclusion

Deleting Data

- Rows can be deleted through the HBase Shell, via the Java API, or using Thrift
- Recall that HBase marks rows for deletion, and the actual deletion occurs at a later time
- Multiple deletes can be performed by batching them together in a list

Removing Rows

- A `Delete` object will delete the entire row across all column families
- Methods can be called on the `Delete` object to only delete certain column families or column descriptors

```
Configuration cfg = HBaseConfiguration.create();
HConnection conn = HConnectionManager.createConnection(cfg);
HTableInterface table =
conn.getTable(TableName.valueOf("movie"));

Delete deleteRow = new Delete(Bytes.toBytes("rowkey1"));
table.delete(deleteRow);

table.close();
conn.close();
```

Chapter Topics

Basic Data Access with the HBase API

- Options to Access HBase Data
- Creating and Deleting HBase Tables
- Retrieving Data with Get
- Retrieving Data with Scan
- Inserting and Updating Data
- Deleting Data
- **Hands-On Exercise: Using the Developer API**
- Conclusion

Hands-On Exercise: Using the Developer API

- In this Hands-On Exercise, you will use the Developer API to put, get, and scan tables
- Please refer to the Exercise Manual

Chapter Topics

Basic Data Access with the HBase API

- Options to Access HBase Data
- Creating and Deleting HBase Tables
- Retrieving Data with Get
- Retrieving Data with Scan
- Inserting and Updating Data
- Deleting Data
- Hands-On Exercise: Using the Developer API
- **Conclusion**

Key Points

- HBase has a Java API, which is the only first class citizen
- HBase can be accessed with the HBase shell, Java API, Thrift, and REST interfaces
- All administrative functions can also be performed with the Java API
- Rows can be accessed and deleted using the row key
- The HBase APIs allow data to be added or updated
- Scans allow a program to read all rows or rows in a range



More Advanced HBase API Features

Chapter 8



Course Chapters

- Introduction
- Introduction to Hadoop and HBase
- HBase Tables
- HBase Shell
- HBase Architecture Fundamentals
- HBase Schema Design
- Basic Data Access with the HBase API
- **More Advanced HBase API Features**
- HBase on the Cluster
- HBase Reads and Writes
- HBase Performance Tuning
- HBase Administration and Cluster Management
- HBase Replication and Backup
- Using Hive and Impala with HBase
- Conclusion
- Appendix: Using Python and Thrift to Access HBase Data
- Appendix: OpenTSDB

More Advanced HBase API Features

In this chapter you will learn

- **How to use Filters with the Scan API**
- **How and why to use atomic operations**
- **How to use coprocessors**

Chapter Topics

More Advanced HBase API Features

- **Filtering Scans**
- Hands-On Exercise: HBase Filters
- Best Practices
- HBase Coprocessors
- Hands-On Exercise: Example Using Atomic Counters
- Conclusion

Filters

- **Not all queries can be performed with just a row key**
 - Using scans passes back all rows to the client
 - Client must perform all logic once the data is received to determine which rows are the ones desired
- **Scans can be augmented with Filters**
 - Filters allow logic to be run on RegionServers before the data is returned
 - RegionServers run the logic on the rows and only send what passes the test
 - Causes less data to be sent over the wire
- **Scans with Filters can be used in the HBase Shell, via the Java API, or using Thrift**

Using Filters

- **Filters are made up of:**
 - A Filter that specifies what piece of information to process, such as a column family or a value
 - A ByteArrayComparable that defines how to process the data
 - A CompareOp that defines the operand for the result
- **HBase contains many built-in filters**
 - Allows you to use filters without having to write a new one
 - Several filters can be combined using the built-in FilterList filter
- **You can create your own Filter**
 - The class must implement the Filter interface
 - A JAR with the class needs to be on every RegionServer and in its classpath

ByteArrayComparable

- **ByteArrayComparable tells HBase how to process a Filter**
 - The Filter decides which piece of information is fed into the ByteArrayComparable, such as a column family or value
- **Many ByteArrayComparable subclasses are built into HBase**
 - BinaryComparator does binary matches of the specified value
 - NullComparator checks if the value is null or not
 - RegexStringComparator runs a regular expression
 - SubstringComparator checks to see if a portion of the string matches
 - Etc.

CompareOp

- **CompareOp tells HBase how to evaluate the result of a Filter**
 - Gives the operands for the `ByteArrayComparable`
- **Possible values are: `EQUAL`, `GREATER`, `GREATER_OR_EQUAL`, `LESS`, `LESS_OR_EQUAL`, `NOT_EQUAL`, `NO_OP`**
- **NO_OP is used for testing**
 - It ensures that everything on the server side will be scanned, and the scan executes as quickly as possible since no rows are returned in the result
- **Some `ByteArrayComparable` classes only support certain CompareOps**

Built-in Filters (1)

- **RowFilter** filters based on the row keys

```
RowFilter rf = new RowFilter(  
    CompareFilter.CompareOp.EQUAL, new  
    RegexStringComparator("h[aeu]ll{o}"));
```

- **PrefixFilter** filters based on the row keys beginning with a prefix

```
PrefixFilter pf = new PrefixFilter(rowBytes);
```

- **QualifierFilter** filters based on the column descriptor name

```
QualifierFilter qf = new QualifierFilter(  
    CompareFilter.CompareOp.EQUAL, new  
    SubstringComparator("column"));
```

Built-in Filters (2)

- **ValueFilter** compares each value to the specified comparator using the compare operator
 - When the comparison result is true, it returns the key-value

```
ValueFilter vf = new ValueFilter(  
    CompareFilter.CompareOp.EQUAL,  
    new BinaryComparator(Bytes.toBytes("value")));
```

Built-in Filters (3)

- **SingleColumnValueFilter** checks for a value in a specific column family and column descriptor
 - If the specified column is not found, all columns of that row are emitted
 - Set the `filterIfColumnMissing` parameter to `true` to emit nothing when the column is not found
 - If the column is found and
 - the comparison returns true, all columns of that row are emitted
 - the comparison returns false, the row is not emitted

```
SingleColumnValueFilter scvf = new  
    SingleColumnValueFilter(FAMILY_BYTES, COLUMN_BYTES,  
    CompareOp.NOT_EQUAL,  
    new BinaryComparator(Bytes.toBytes("value")));
```

Scan with Filter: Complete Code

```
byte[] startRow = Bytes.toBytes("startrow");
byte[] stopRow = Bytes.toBytes("stoprow");

Scan s = new Scan(startRow, stopRow);

BinaryComparator b = new BinaryComparator(
    Bytes.toBytes("VALUE"));
SingleColumnValueFilter f =
    new SingleColumnValueFilter(FAMILY_BYTES,
        COLUMN_BYTES, CompareOp.EQUAL, b);
s.setFilter(f);

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

Scan with Filter: Comparator

```
byte[] startRow = Bytes.toBytes("startrow");  
byte[] stopRow = Bytes.toBytes("stoprow");
```

```
Scan s = new Scan(startRow, stopRow);
```

Binary
Byt

Single
new

- Scan takes a start row and a stop row as arguments
 - Both need to be passed in as byte arrays
 - Scan begins at or after the start row (it is inclusive)
 - Scan is exclusive of the end row

```
COLUMN_BYTES, compareOp.EQUAL, b);  
s.setFilter(f);
```

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

Scan with Filter: Comparator

```
byte []  
byte []  
Scan s
```

The BinaryComparator object is initialized to evaluate the value; comparison is performed on the bytes defined in the Filter. The byte array from the string passed into the constructor is used as the value for the comparison.

```
BinaryComparator b = new BinaryComparator(  
    Bytes.toBytes("VALUE"));  
SingleColumnValueFilter f =  
    new SingleColumnValueFilter(FAMILY_BYT  
    ES,  
    COLUMN_BYT  
    ES, CompareOp.EQUAL, b);  
s.setFilter(f);  
  
ResultScanner rs = table.getScanner(s);
```

Scan with Filter: Filter

```
byte[] b = new byte[10];  
byte[] b1 = new byte[10];  
  
Scan s = new Scan();  
s.addColumn(FAMILY, COLUMN);  
BinaryComparator bc = Bytes.toBytes("bc");  
Byt  
ScanFilter f = new SingleColumnValueFilter(FAMILY_BYT  
The SingleColumnValueFilter is instantiated. Since  
the filter deals with a specific column family and column  
descriptor, those must be passed in as arguments. The  
CompareOp needs to be specified to evaluate the return  
from the comparator. The comparator itself is the last  
argument. Finally, the filter is set in the scanner object.
```

```
SingleColumnValueFilter f =  
    new SingleColumnValueFilter(FAMILY_BYT  
    COLUMN_BYT, CompareOp.EQUAL, b);  
s.setFilter(f);
```

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

Filter Lists

- Several filters can be grouped together and nested
- Java API:

```
ArrayList<Filter> filters = new ArrayList<Filter>();  
filters.add(filter1);  
filters.add(filter2);  
FilterList filterList = new FilterList(  
    FilterList.Operator.MUST_PASS_ALL, filters);
```

- The `FilterList.Operator` tells the filter list how to evaluate the results
 - `FilterList.Operator.MUST_PASS_ALL` works like an AND
 - `FilterList.Operator.MUST_PASS_ONE` works like an OR

Chapter Topics

More Advanced HBase API Features

- Filtering Scans
- **Hands-On Exercise: HBase Filters**
- Best Practices
- HBase Coprocessors
- Hands-On Exercise: Example Using Atomic Counters
- Conclusion

Hands-On Exercise: HBase Filters

- In this Hands-On Exercise, you will use Filters with scans to find rows with certain values
- Please refer to the Exercise Manual

Chapter Topics

More Advanced HBase API Features

- Filtering Scans
- Hands-On Exercise: HBase Filters
- **Best Practices**
- HBase Coprocessors
- Hands-On Exercise: Example Using Atomic Counters
- Conclusion

Overuse of `toBytes()`

- The previous example had too many `toBytes()` for column families and column descriptors
 - Hard to read, and inefficient if the name is used more than once
 - Best practice: only hard code names in one place
- Keep String and byte array versions of column families and descriptors

```
public static final String FAMILY = "desc";
public static final byte[] FAMILY_BYTES =
    Bytes.toBytes(FAMILY);
```

- Using byte arrays in code:

```
byte[] byteArray = result.getValue(FAMILY_BYTES,
    COLUMN_BYTES);
```

Retrieving as Little as Possible

- Try to constrain Gets to the least amount of column families or column descriptors possible
 - Speeds up queries and transfer times by dealing with less data
 - One column family or column descriptor could be large
- Getting all column descriptors for a column family

```
Get get = new Get(Bytes.toBytes("rowkey1"));  
get.addFamily(FAMILY_BYTES);
```

- Getting a specific column descriptor in a column family

```
Get get = new Get(Bytes.toBytes("rowkey1"));  
get.addColumn(FAMILY_BYTES, COLUMN_BYTES);
```

Batching Gets

- Allows multiple actions of the same type to be performed in a single call
 - Batch all Get objects together in a single API call

```
ArrayList<Get> getsList = new ArrayList<Get>();  
  
getsList.add(new Get(Bytes.toBytes("rowkey1")));  
getsList.add(new Get(Bytes.toBytes("rowkey2")));  
  
Result[] results = table.get(getsList);  
  
for (Result r : results) {  
    // Do something with rows...  
}
```

Batching Puts

- **Batching allows multiple Puts to be performed in a single call**
 - Improves performance by minimizing RegionServer calls
 - Best practice when dealing with many Puts at once

```
ArrayList<Put> putsList = new ArrayList<Put>();  
...  
putsList.add(p1);  
putsList.add(p2);  
...  
table.put(putsList);
```

- **Similar to Get and Put, calls to the Delete API can also be batched**

HBase Counters

- HBase can atomically increment 64-bit values
 - All calls return the value as a 64-bit integer or long
- HBase shell:

```
hbase> incr 'movie', 'row1', 'metrics:ticketsSold', 1
```

- Java API:

```
long incrementAmount = 100L;  
long newValue =  
    table.incrementColumnValue(Bytes.toBytes("row1"),  
    FAMILY_BYTES, COLUMN_BYTES, incrementAmount);
```

Atomic Puts and Deletes

- **Performing a Get followed by a Put is not atomic**
 - Another user could update the cell between the two operations
 - To update data or state atomically, use `checkAndPut()` and `checkAndDelete()`

```
Put p = new Put(rowkey);
p.add(FAMILY_BYTES, COLUMN_BYTES, newvaluebytes);
boolean success = table.checkAndPut(rowkey, FAMILY_BYTES,
    COLUMN_BYTES, valuebytes, p);
// Optionally retry the operation if success equals false

Delete d = new Delete(rowkey);
boolean deleted = userTable.checkAndDelete(rowkey,
    FAMILY_BYTES, COLUMN_BYTES, valuebytes, d);
// Optionally retry the operation if deleted equals false
```

Chapter Topics

More Advanced HBase API Features

- Filtering Scans
- Hands-On Exercise: HBase Filters
- Best Practices
- **HBase Coprocessors**
- Hands-On Exercise: Example Using Atomic Counters
- Conclusion

What is a Coprocessor?

- HBase has a feature similar to stored procedures in a traditional RDBMS called Coprocessors
 - These come in two types: *Endpoints* and *Observers*
- *Endpoints* allow you to add functionality to HBase that is exposed as a new method
 - Similar to a stored procedure in an RDBMS
- *Observers* can hook into HBase as it performs various operations
 - Similar to a trigger in an RDBMS

Examples of Coprocessor Use

- **Coprocessors can be used for implementing**

- Access control
- Secondary indexes
- Optimized search
- Reduced result sets
- Data Aggregation
- Real time analytics
- etc.

Coprocessors Warning

- **WARNING: Coprocessors are advanced options that can cause performance and stability problems in HBase!**
- **Coprocessors run in the same process space as the RegionServer**
 - A bug in your Coprocessor can cause the RegionServer to fail
 - This failure can cascade through your **entire cluster** as the regions are migrated due to other RegionServers failing
- **Debugging and supporting issues as a result of Coprocessors is extremely difficult**
 - When isolating an issue, disable Coprocessors to see if that narrows down the issue

Endpoints

- **Endpoints allow new methods to be exposed on a RegionServer**
- **Some algorithms are better done on the RegionServer**
 - Algorithms that do sums, counts, or averages could use endpoints
 - ‘Group by’ clauses can be implemented with an endpoint
 - Execute more complicated logic than can be created with a Filter
- **Running code in an endpoint reduces the network traffic back to a client**
 - Otherwise all data has to be passed back and the client will then need to run the algorithm

Observers

- **Observers hook into a RegionServer's lifecycle as each request happens**
 - Observers run on the RegionServer between the client and the RegionServer
- **pre and post methods allow an Observer to watch as a request performs an action**
 - Operations like Get, Put and Scan all have pre and post methods to perform an operation before and after the call
- **Observers allow automatic functionality when doing an operation. For example:**
 - More granular security checks could be run by checking pre before an operation
 - Secondary indexes could be created by performing an operation after a Put

Chapter Topics

More Advanced HBase API Features

- Filtering Scans
- Hands-On Exercise: HBase Filters
- Best Practices
- HBase Coprocessors
- **Hands-On Exercise: Example Using Atomic Counters**
- Conclusion

Hands-On Exercise: Hands-On Exercise: Example Using Atomic Counters

- In this Hands-On Exercise you will try out atomic counters.
- Please refer to the Exercise Manual

Chapter Topics

More Advanced HBase API Features

- Filtering Scans
- Hands-On Exercise: HBase Filters
- Best Practices
- HBase Coprocessors
- Hands-On Exercise: Example Using Atomic Counters
- **Conclusion**

Key Points

- Filters can be used with scans to evaluate data
- Using a Filter can dramatically reduce the amount of data passed back to the client
- Atomic operations ensure data read is not modified by another program before your update has occurred



HBase on the Cluster

Chapter 9



Course Chapters

- Introduction
- Introduction to Hadoop and HBase
- HBase Tables
- HBase Shell
- HBase Architecture Fundamentals
- HBase Schema Design
- Basic Data Access with the HBase API
- More Advanced HBase API Features
- **HBase on the Cluster**
- HBase Reads and Writes
- HBase Performance Tuning
- HBase Administration and Cluster Management
- HBase Replication and Backup
- Using Hive and Impala with HBase
- Conclusion
- Appendix: Using Python and Thrift to Access HBase Data
- Appendix: OpenTSDB

HBase on the Cluster

In this chapter you will learn

- How HBase scales
- How HBase deals with deletes and table growth

Chapter Topics

HBase on the Cluster

- **How HBase uses HDFS**
- Hands-On Exercise: Exploring HBase
- Compaction and Splits
- Hands-On Exercise: Flushes and Compactions
- Conclusion

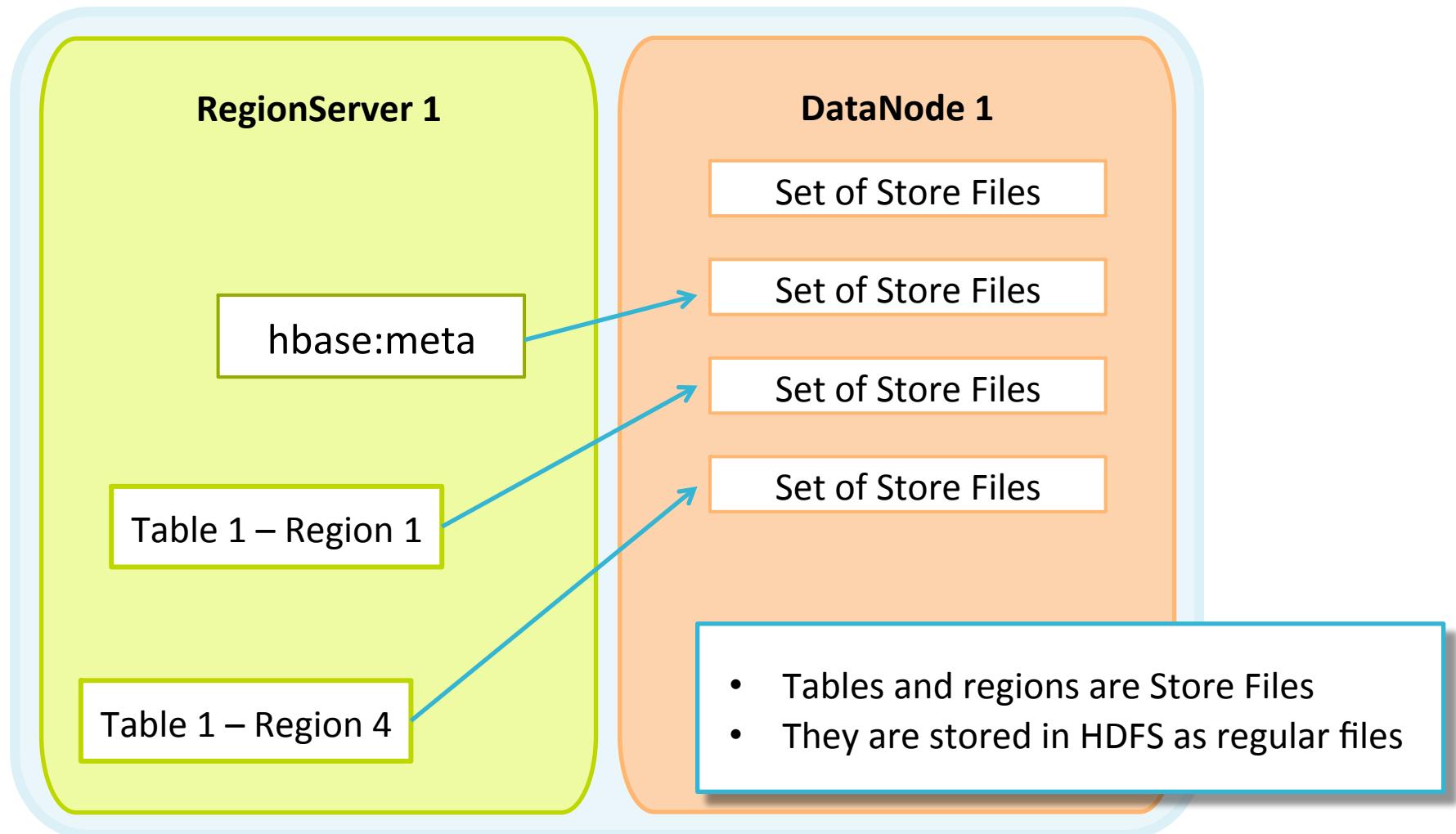
HBase and HDFS

- **HBase uses HDFS for the storage of all table data**
 - Regions are stored as files in HDFS
- **HDFS provides**
 - High Availability for NameNodes to avoid a single point of failure
 - Durability for data because it is stored three times on multiple DataNodes
 - Storage scalability through the addition of DataNodes
- **Regions can be written to and read from anywhere in HDFS, allowing RegionServers to run anywhere on the cluster**

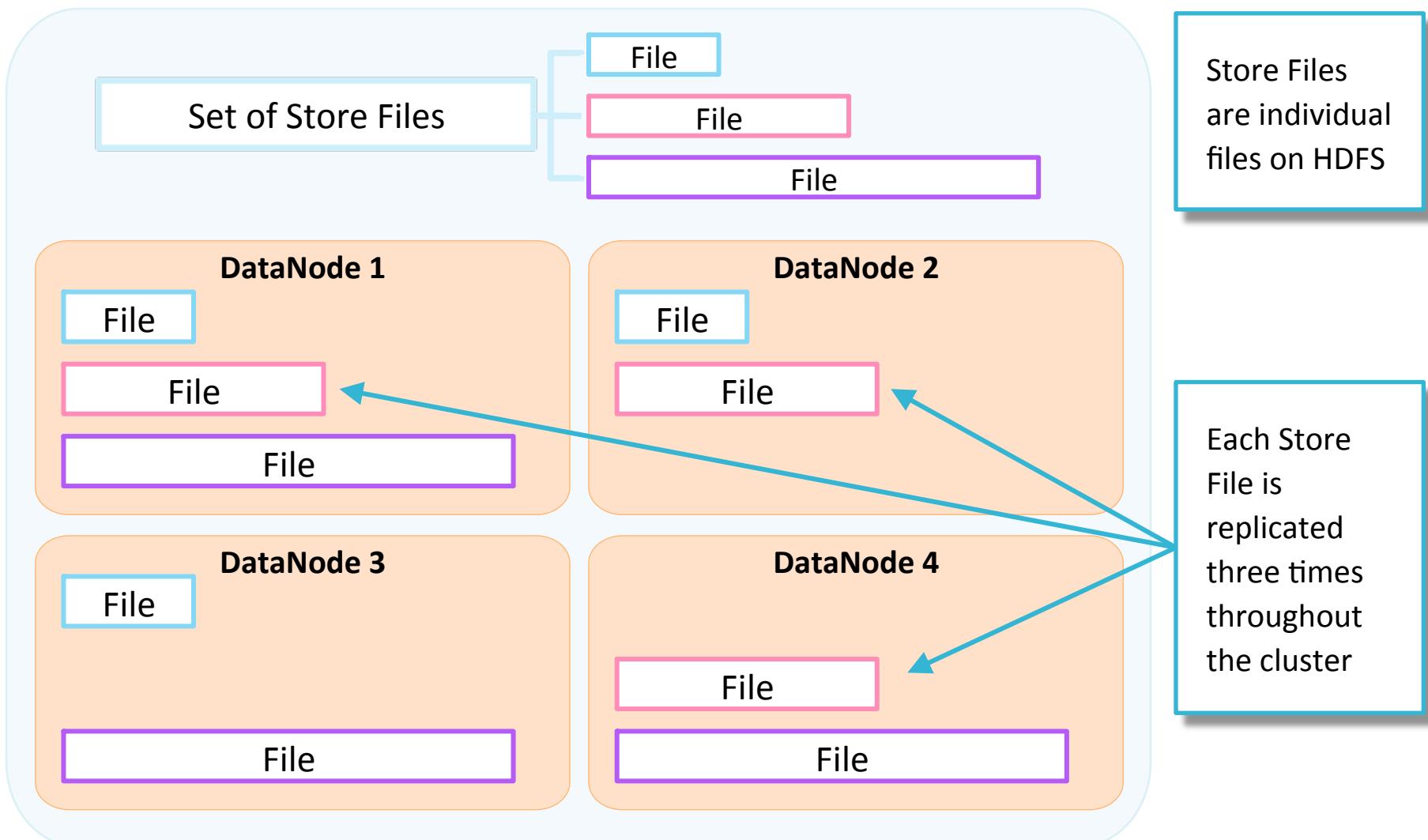
Store Files

- **HBase tables are stored permanently in HDFS**
- **A region's column families are divided into Stores**
 - A Store corresponds to a column family for a table for a given region
- **Store file**
 - Actual data storage of the data that makes up a table
 - Uses an HBase-specific file format called HFile

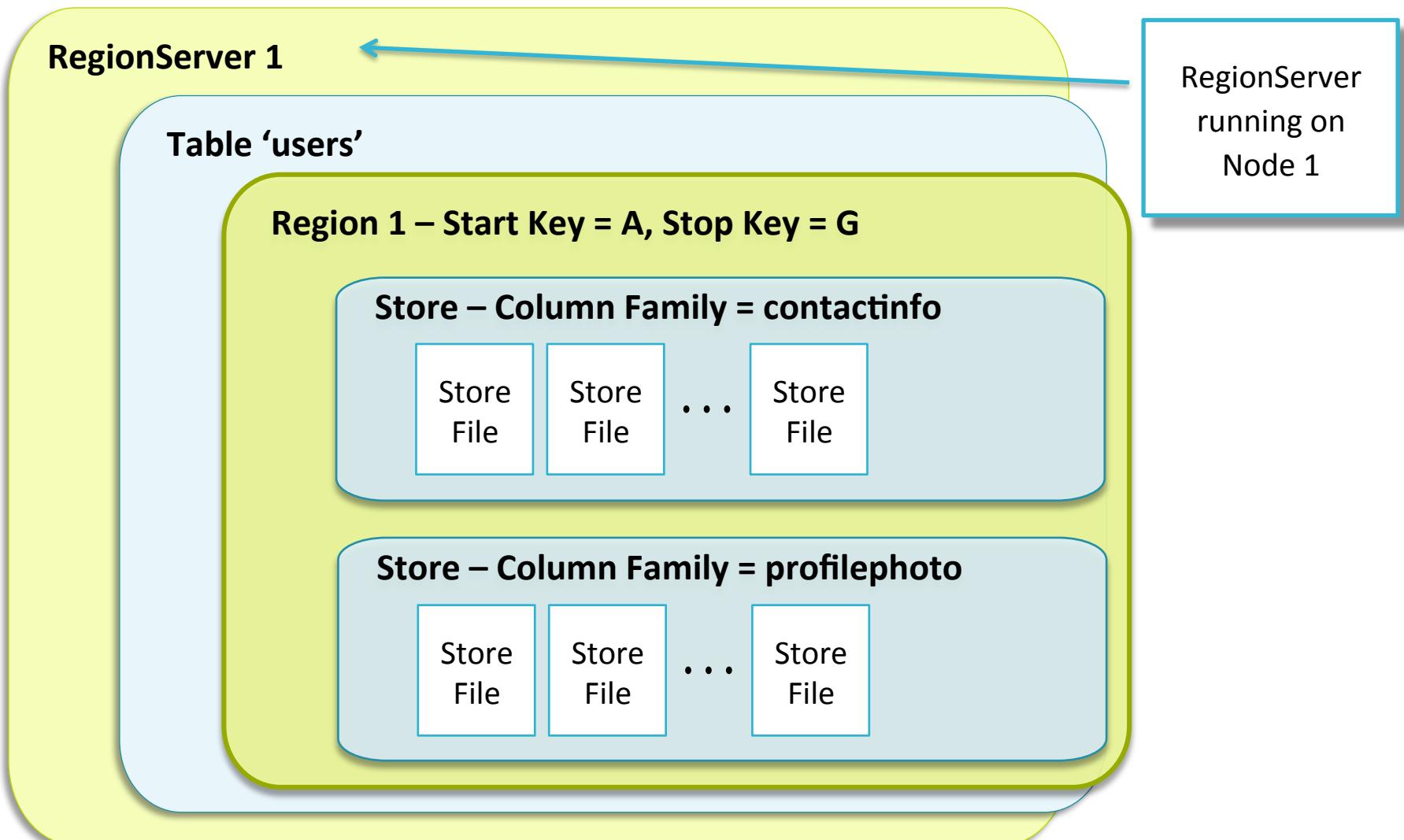
Store Files as HDFS Files



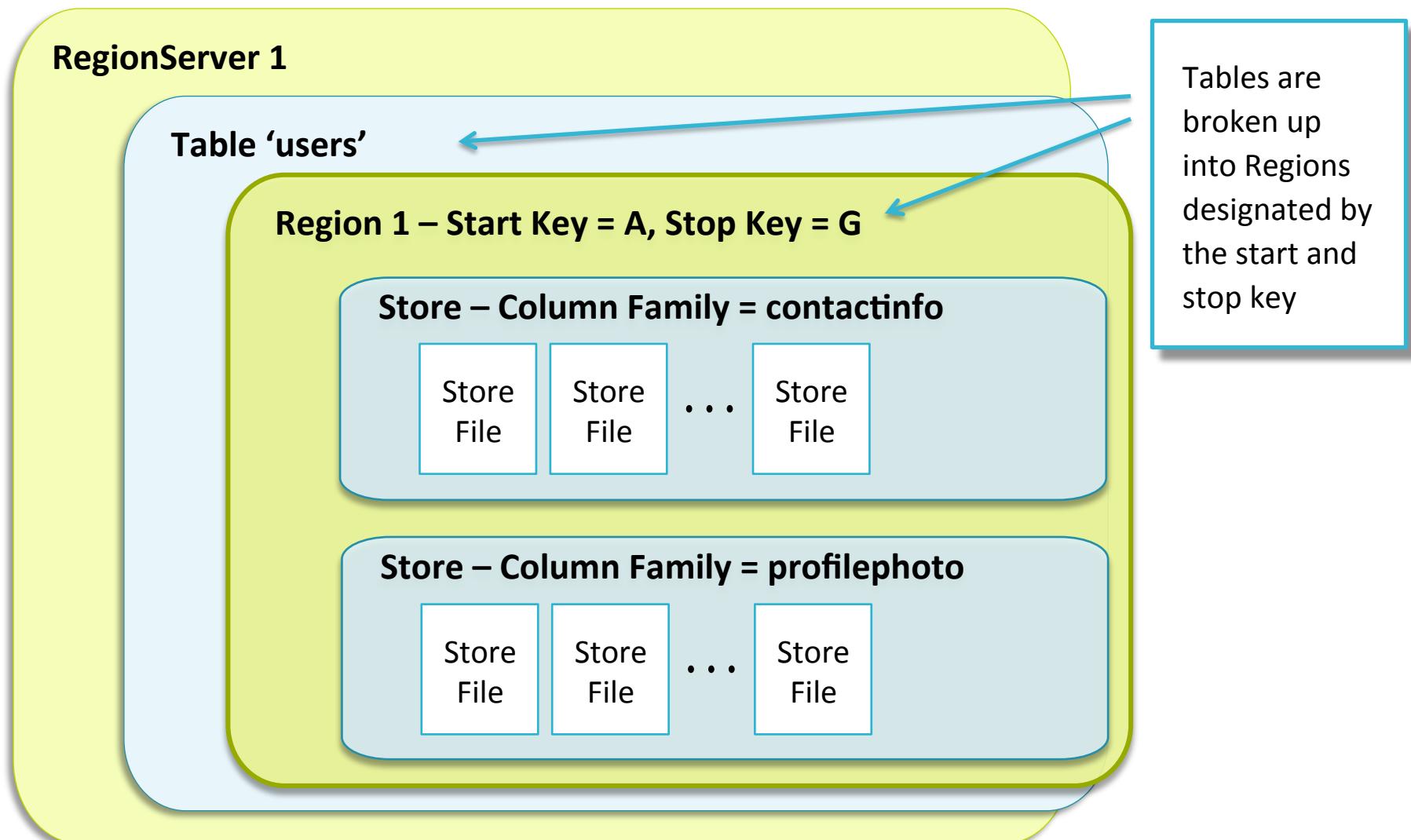
Store File Replication



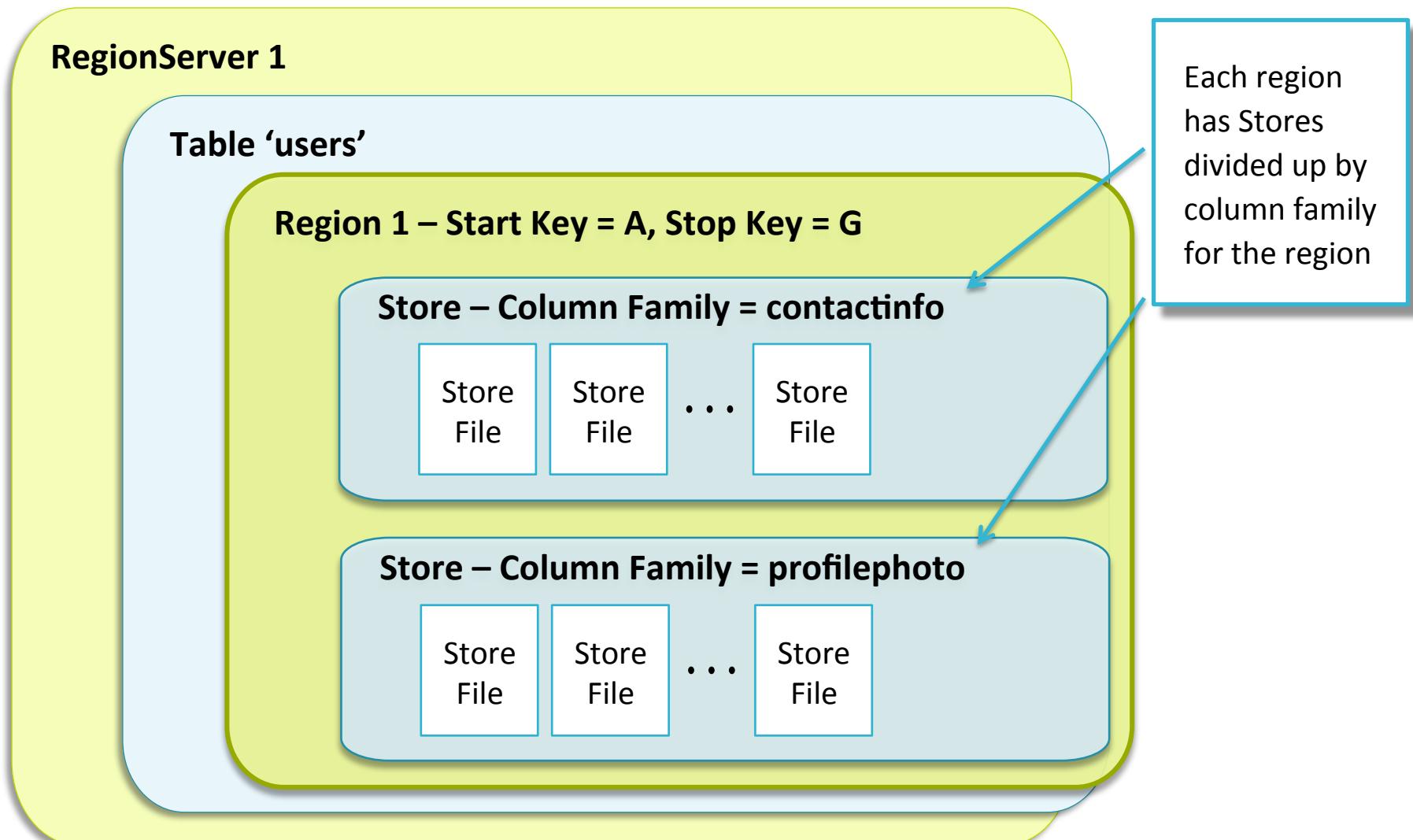
Store Files: RegionServers



Store Files: Tables and Regions



Store Files: Column Families



Store Files: Store File Storage

RegionServer 1

Table 'users'

Region 1 – Start Key = A, Stop Key = G

Store – Column Family = contactinfo



Store – Column Family = profilephoto



Each Store has files in HDFS called Store Files where the column family's data is stored

Chapter Topics

HBase on the Cluster

- How HBase uses HDFS
- **Hands-On Exercise: Exploring HBase**
- Compaction and Splits
- Hands-On Exercise: Flushes and Compactions
- Conclusion

Hands-On Exercise: Exploring HBase

- In this Hands-On Exercise, you will explore Regions, the `hbase:meta` table, and HBase file layouts
- Please refer to the Exercise Manual

Chapter Topics

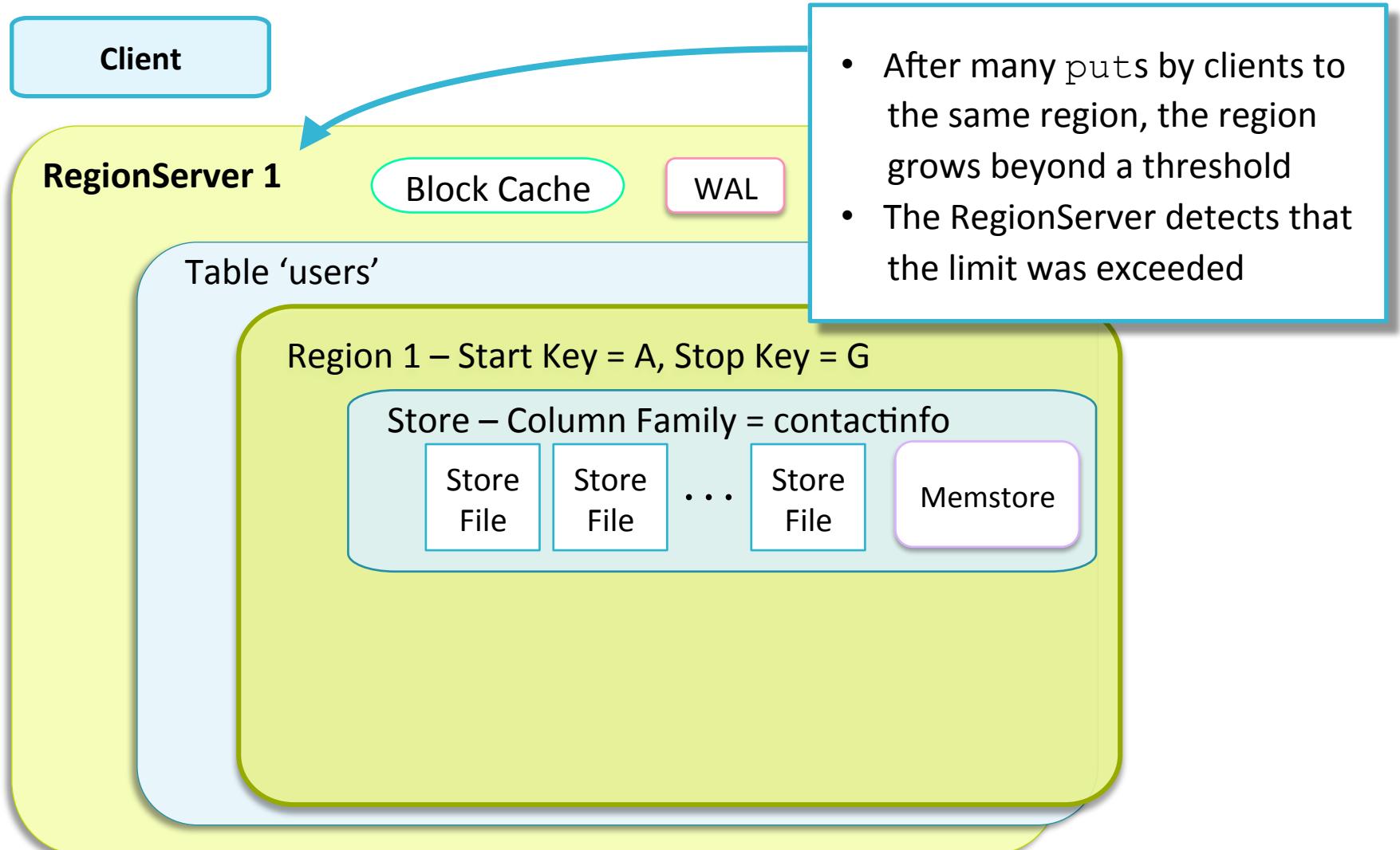
HBase on the Cluster

- How HBase uses HDFS
- Hands-On Exercise: Exploring HBase
- **Compaction and Splits**
- Hands-On Exercise: Flushes and Compactions
- Conclusion

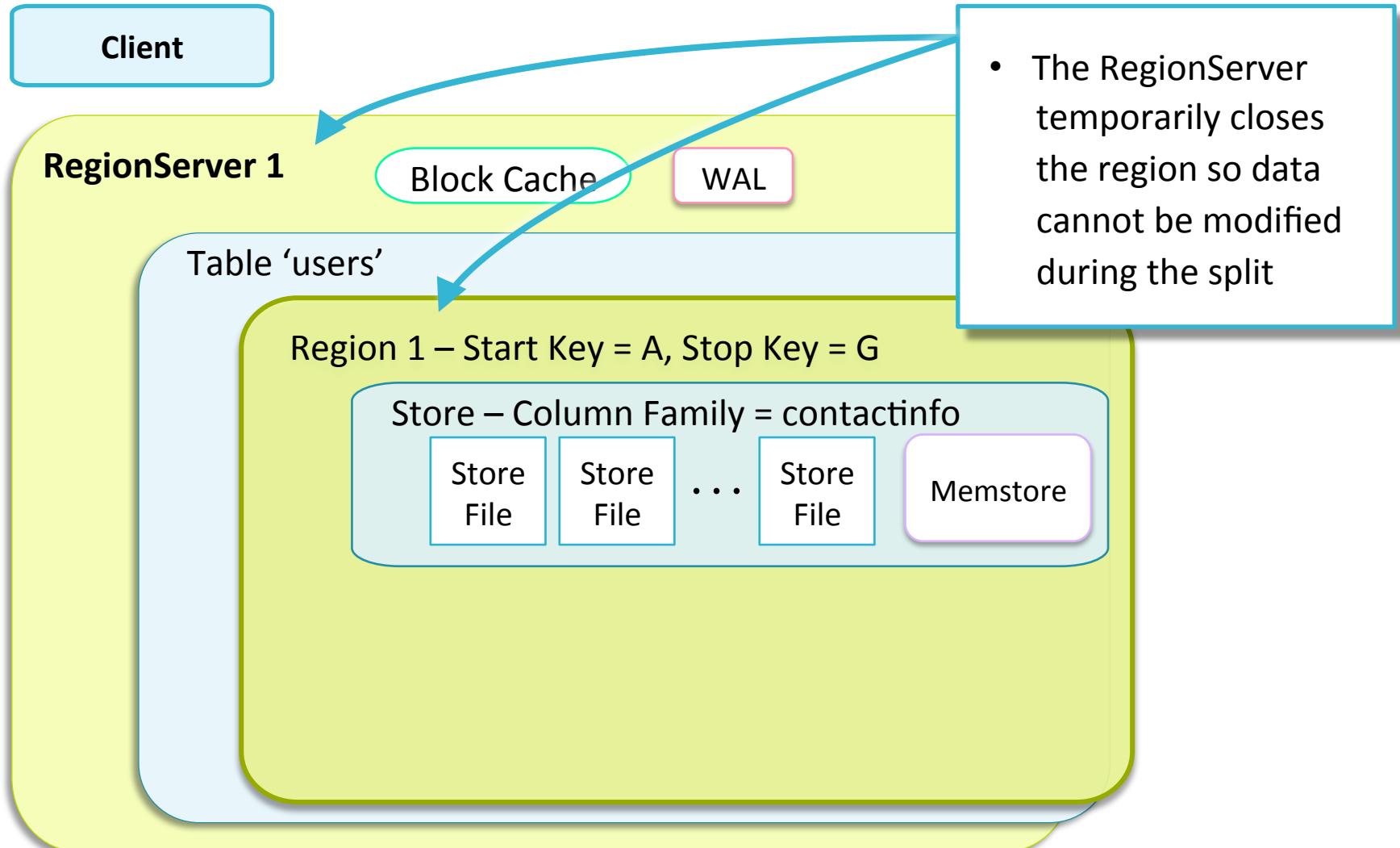
Region Splits

- When regions get too big they are automatically split
 - HBase creates two “daughter” reference files
 - Daughter files only contain the key where region was split
 - At major compaction the original data files are rewritten into separate files in the new region directory
 - Small reference files and original region are removed

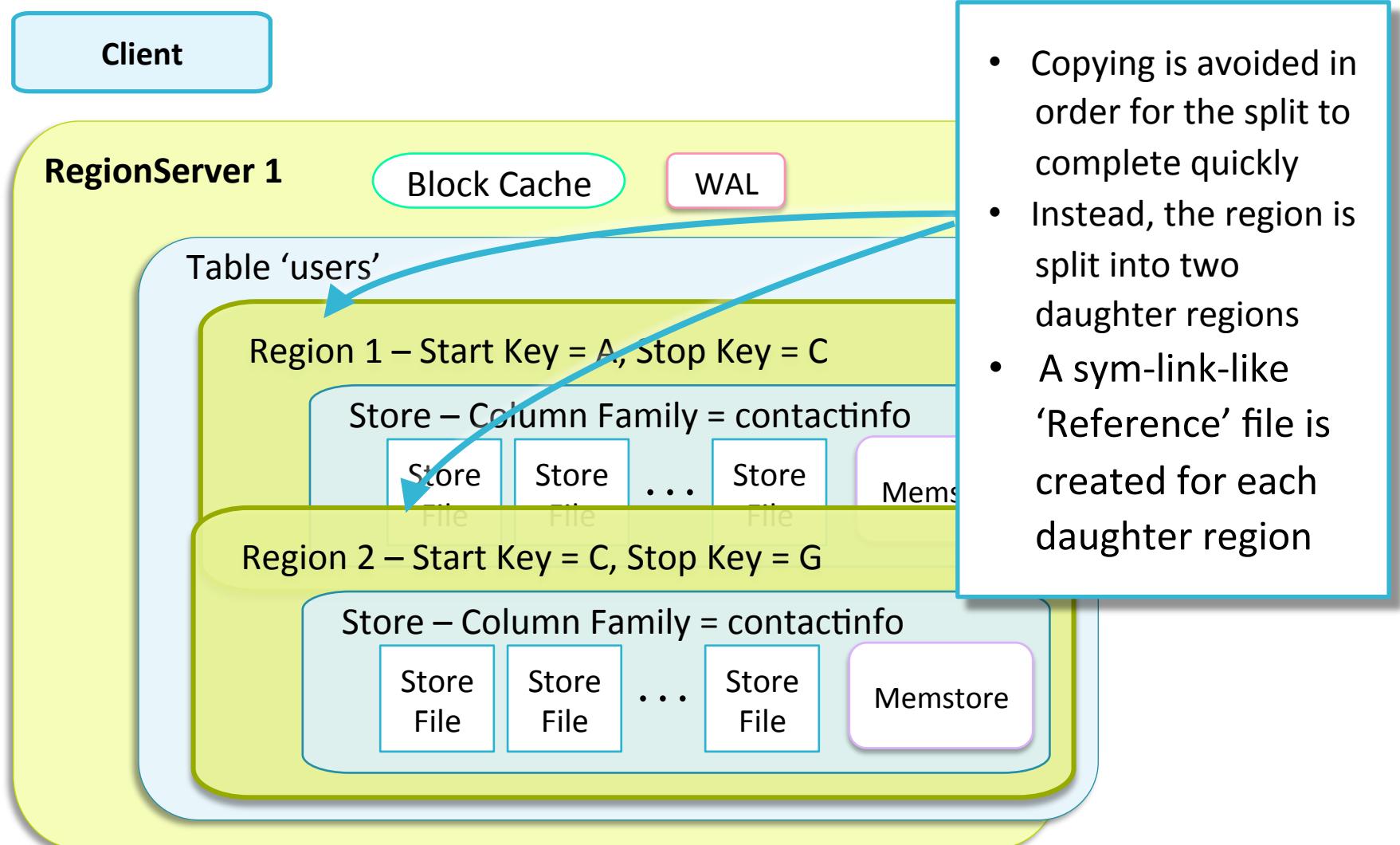
Region Split: Limit Exceeded



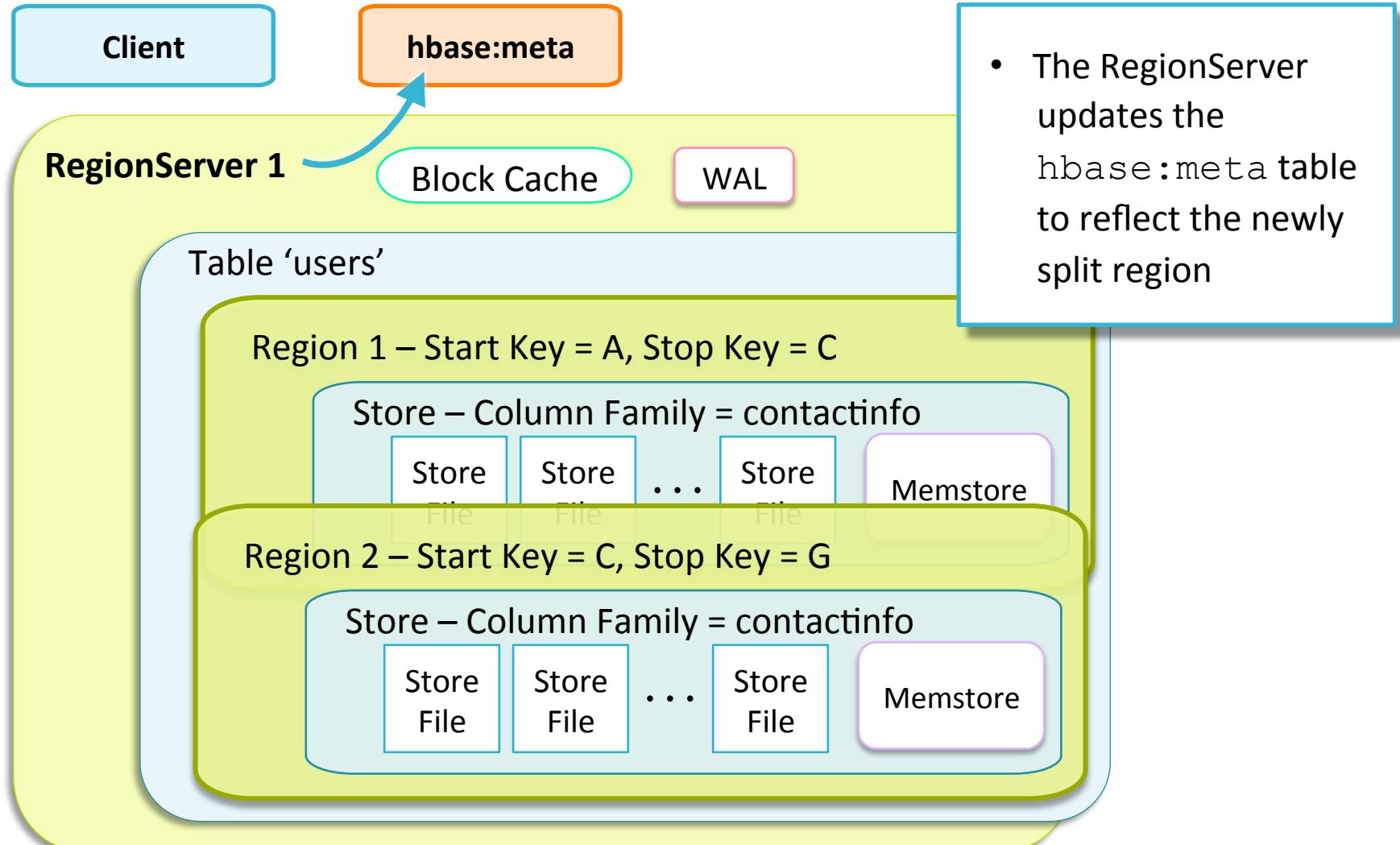
Region Split: Region Closed



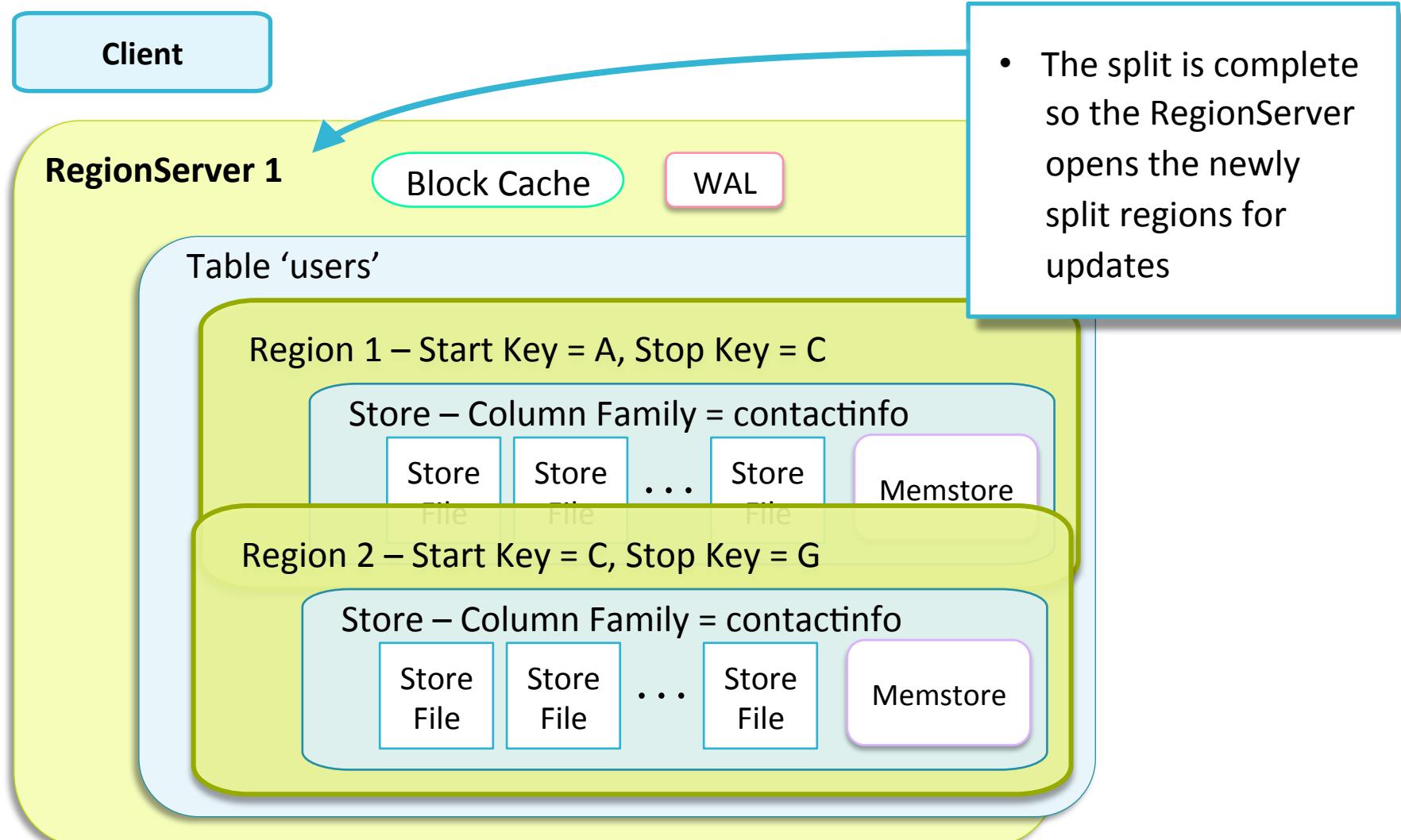
Region Split: Daughter Regions



Region Split: hbase:meta Update



Region Split: Split Complete



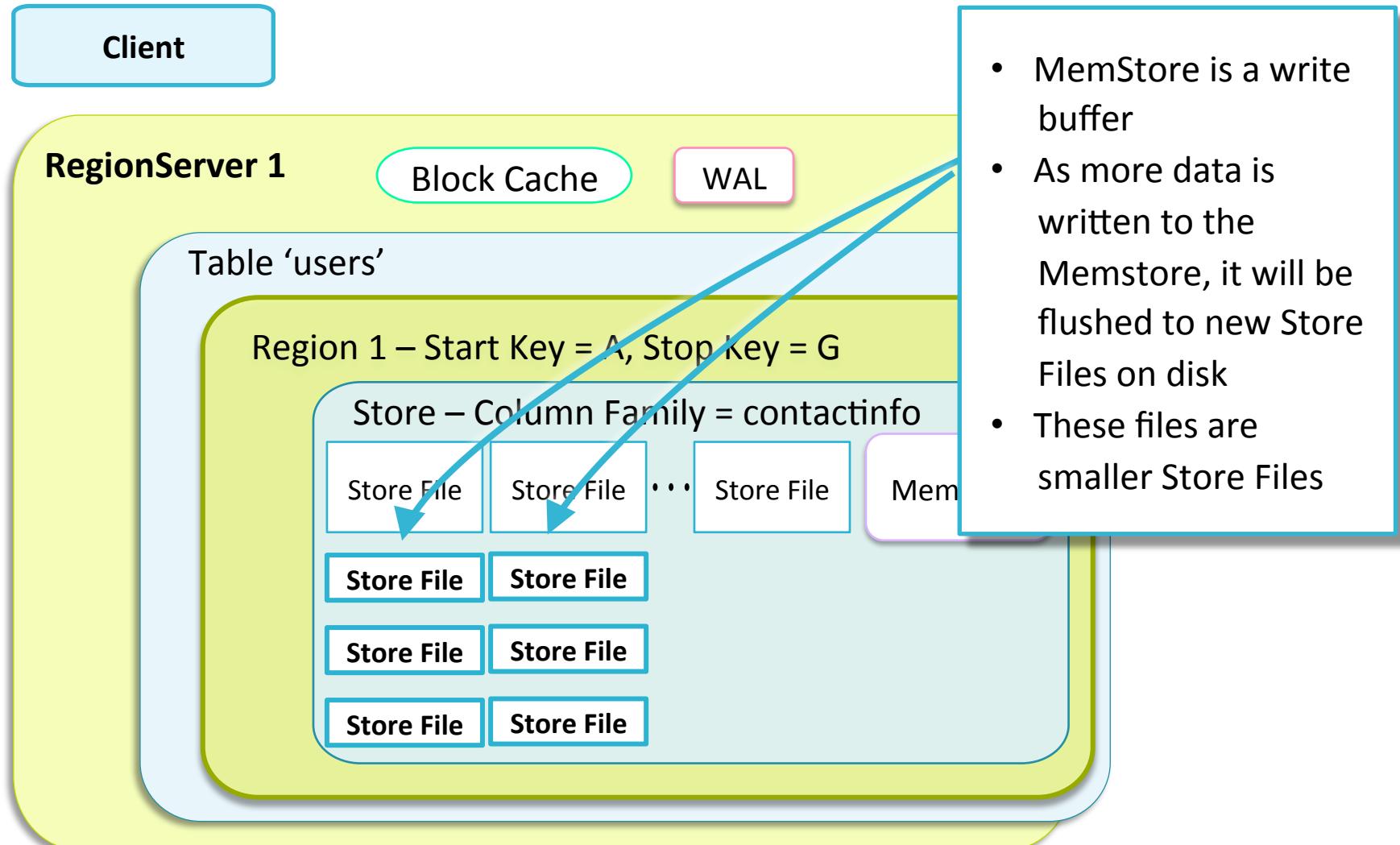
Region Size

- **Region size**
 - Basic element of availability and distribution
 - You do not want a high number of regions for a small amount of data
 - High region count (e.g., > 3000) can impact performance
 - Low region count prevents parallel scalability
 - A lower number of regions is preferred, generally in the range of 20 to low-hundreds per RegionServer
- **Load Balancer**
 - Automatically moves regions around to balance cluster load
 - Period when this runs is configurable
- **Data Distribution**
 - Data is distributed across regions based on row keys

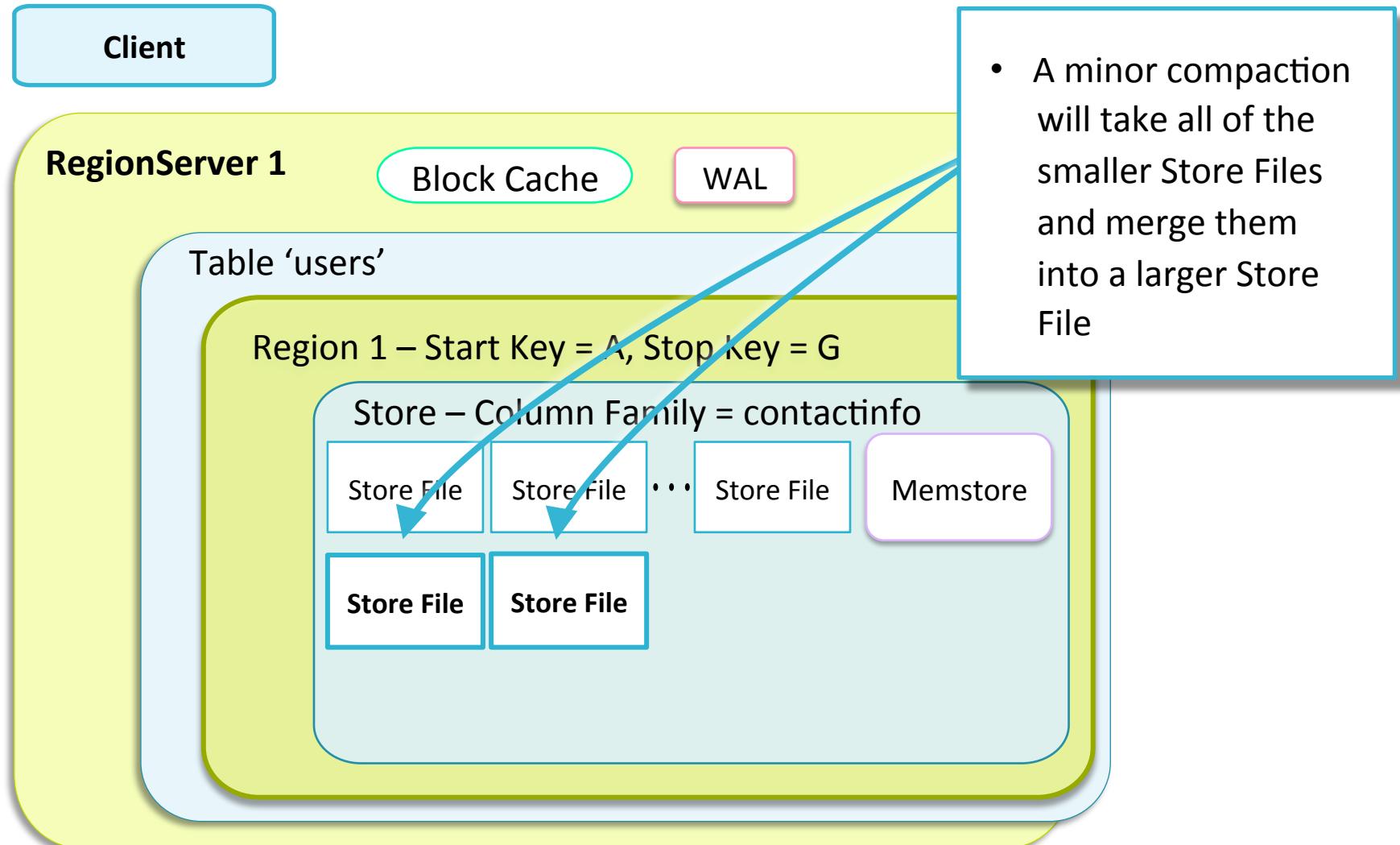
Minor Compactions

- **Minor compaction combines several Store Files into a single file**
- **Usually runs after three store files have accumulated**
 - This is not guaranteed; additional complex logic is also involved
- **Can be configured with `hbase.hstore.compactionThreshold`**
 - Larger number results in fewer compactions
 - But each compaction will take longer
- **Memstore cannot flush to disk during compaction**
 - If Memstore runs out of memory, clients will block/timeout

Minor Compactions: Small Store Files



Minor Compactions: Store Files



Major Compactions

- Major compaction reads all the Store Files for a region and writes to a single Store File
- Deleted rows and expired versions are removed
- Major compactions happen once per week by default
 - Can be configured with `hbase.hregion.majorcompaction`
 - Set to the value in milliseconds between major compactions
 - Set to 0 to disable automatic major compactions
- A staggered approach is implemented to prevent all RegionServers from compacting at the same time

Major Compactions: Changes and Updates to Tables

Row key	Timestamp	Users Table	
aaronsonj	1378139353	fname: Jim	Iname: Aaronson
aaronsonj	1378138123	fname: Jimbob	Iname: Aaronson
aaronsonj	1378138100	fname: Jamie	Iname: Aaronson
aaronsonj	1378137001	fname: James	Iname: Aaronson
millerb	1378428766	fname: Billie	Iname: Miller
nunezw	1376287655	fname: Willam	Iname: Nu
rossw	1372867889	fname: William	Iname: Ro
sperberp	1376738290	fname: Phyllis	Iname: Sp
turnerb	1378135555	fname: Brian	Iname: Tu
walkerm	1378101012	fname: Martin	Iname: Walker
zykowskiz	1372987903	fname: Zeph	Iname: Zykowski

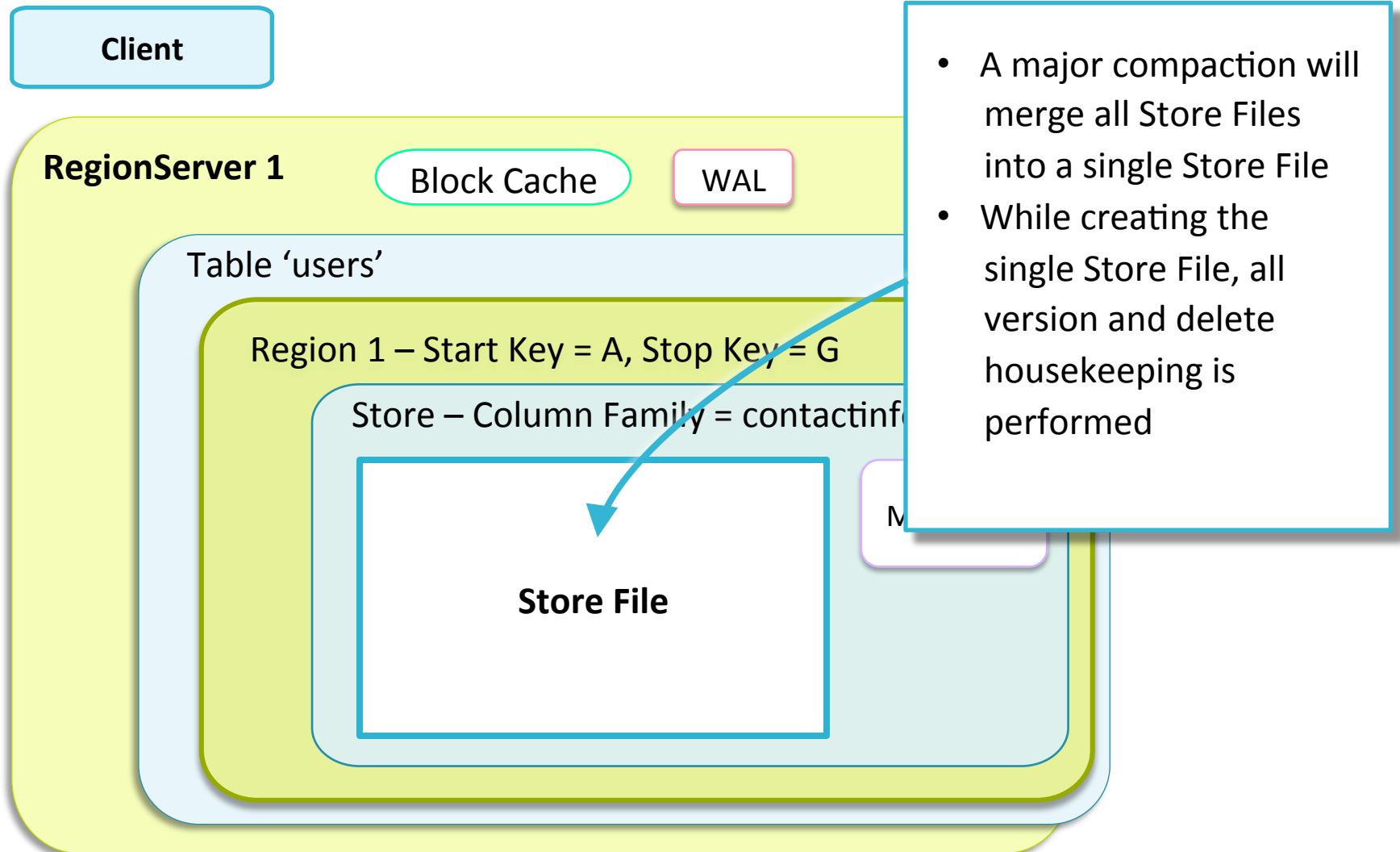
A major compaction will remove any rows that exceed the configured number of versions for a cell

Major Compactions: Deletes to Tables

Row key	Timestamp	Users Table	
aaronsonj	1378139353	fname: Jim	Iname: Aaronson
aaronsonj	1378138123	fname: Jimbob	Iname: Aaronson
aaronsonj	1378138100	fname: Jamie	Iname: Aaronson
aaronsonj	1378137001	fname: James	Iname: Aaronson
millerb	1378428766	fname: Billie	Iname: Miller
nunezw	1376287655	fname: Willam	Iname: Nunez
rossw	1372867889	fname: William	Iname: Ross
sperberp	1376738290	fname: Phyllis	Iname: Sperber
turnerb	1378135555	fname: Brian	Iname: Turner
walkerm	1378101012	fname: Martin	Iname: Walker
zykowskiz	1372987903	fname: Zeph	Iname: Zykowski

A major compaction will remove any rows that were deleted or marked as tombstoned

Major Compactions: Single Store File



Major Compactions: Best Practices

- **Major Compaction is a heavyweight operation**
 - Should be run when load is low if possible
- **Recommendation: set `hbase.hregion.majorcompaction` to a high value and perform major compactions via a script**
 - Setting to 0 runs the risk of major compactions never happening if the script fails
- **`hbase.hregion.majorcompaction.jitter` ensures that not all major compactions happen at once**
 - On each Region Server, `hbase.hregion.majorcompaction` is multiplied by a random fraction inside the bounds of this maximum
 - This is added to `hbase.hregion.majorcompaction` to determine when the next major compaction is to run
 - Ensures that not all major compactions run simultaneously
 - Default is 0.5 (50%)

RegionServer Data Locality

- **The HDFS client writes three replicas of each block by default**
 - First replica is written to the local node
 - Second and third replicas are written to two other nodes in a different rack
- **Region splits and reassessments could result in a RegionServer needing to access non-local Store Files**
 - HDFS will automatically pull the data across the network
 - Reading or writing across the network is not as efficient as local hard disk
- **HBase eventually achieves locality for a region**
 - Gradually after a flush
 - Completely after a major compaction

Chapter Topics

HBase on the Cluster

- How HBase uses HDFS
- Hands-On Exercise: Exploring HBase
- Compaction and Splits
- **Hands-On Exercise: Flushes and Compactions**
- Conclusion

Hands-On Exercise: Flushes and Compactions

- In this Hands-On Exercise, you will learn how to manually flush tables and compact HBase files
- Please refer to the Exercise Manual

Chapter Topics

HBase on the Cluster

- How HBase uses HDFS
- Hands-On Exercise: Exploring HBase
- Compaction and Splits
- Hands-On Exercise: Flushes and Compactions
- **Conclusion**

Key Points

- A region's files are stored in HDFS as **Store Files**
- Regions that grow beyond a certain size are split into two regions
- Some **Store Files** are merged together during **minor compactions**
- A **major compaction** merges all **Store Files** and performs housekeeping



HBase Reads and Writes

Chapter 10



Course Chapters

- Introduction
- Introduction to Hadoop and HBase
- HBase Tables
- HBase Shell
- HBase Architecture Fundamentals
- HBase Schema Design
- Basic Data Access with the HBase API
- More Advanced HBase API Features
- HBase on the Cluster
- **HBase Reads and Writes**
- HBase Performance Tuning
- HBase Administration and Cluster Management
- HBase Replication and Backup
- Using Hive and Impala with HBase
- Conclusion
- Appendix: Using Python and Thrift to Access HBase Data
- Appendix: OpoenTSDB

HBase Reads and Writes

In this chapter you will learn

- How a write in HBase works
- How a read in HBase works
- How to configure the HBase block cache

Chapter Topics

HBase Reads and Writes

- **How HBase Writes Data**
- How HBase Reads Data
- Block Caches for Reading
- Conclusion

MemStore

RegionServer 1

Table 'users'

Region 1 – Start Key = A, Stop Key = G

Store – Column Family = contactinfo

Store File

Store File

...

Store File

MemStore

Store – Column Family = profilephoto

Store File

Store File

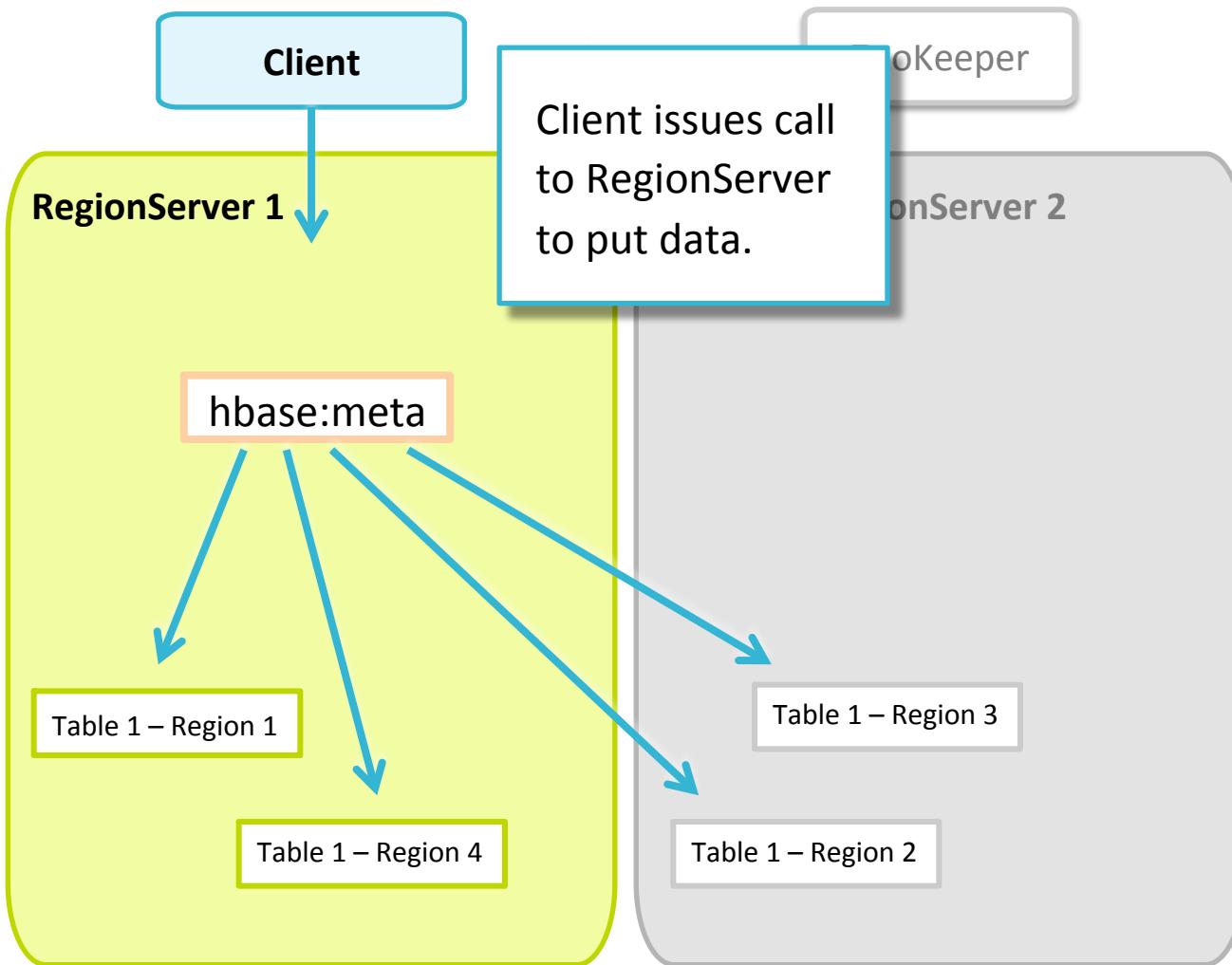
...

Store File

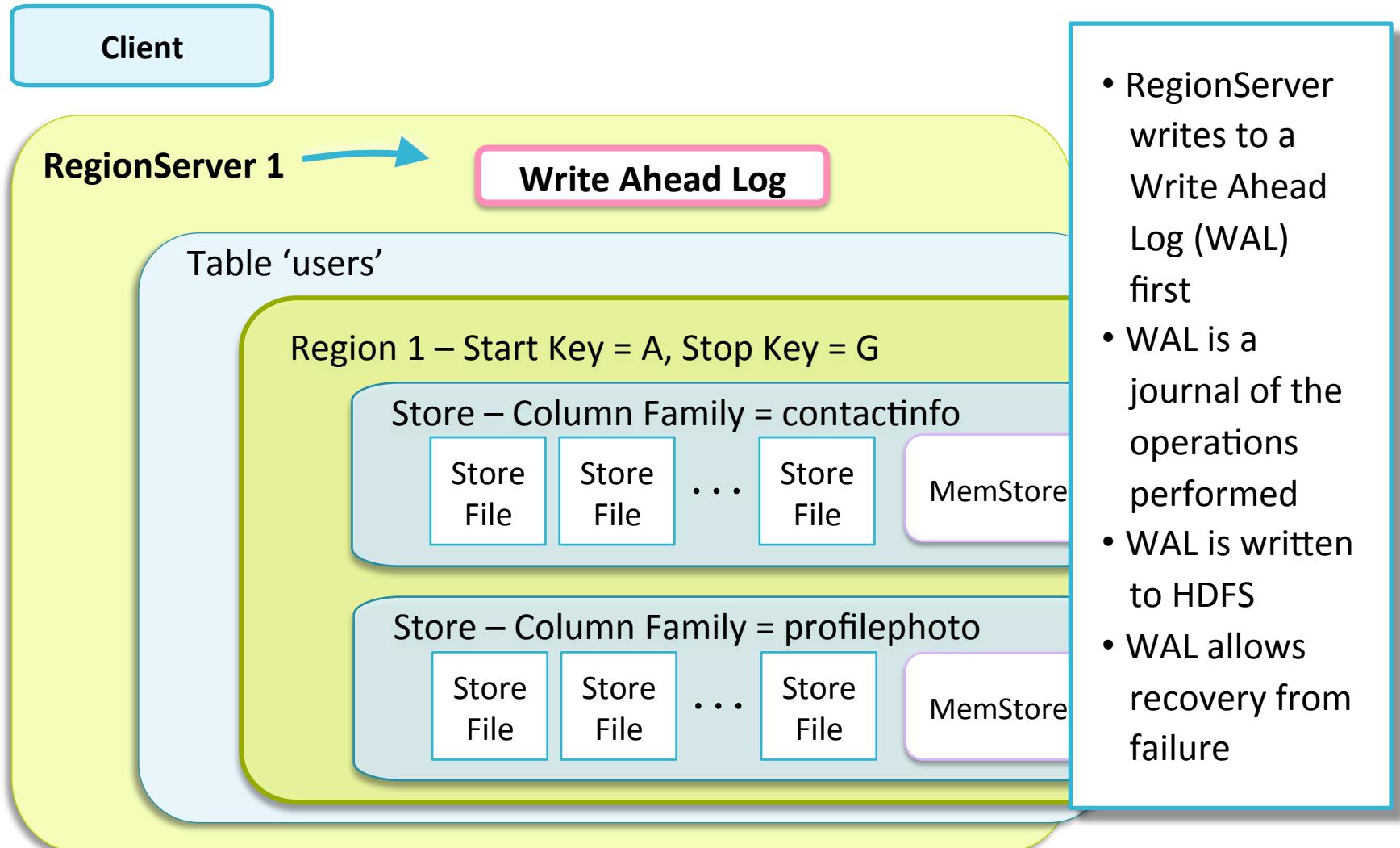
MemStore

MemStores help buffer writes in HBase to improve write performance

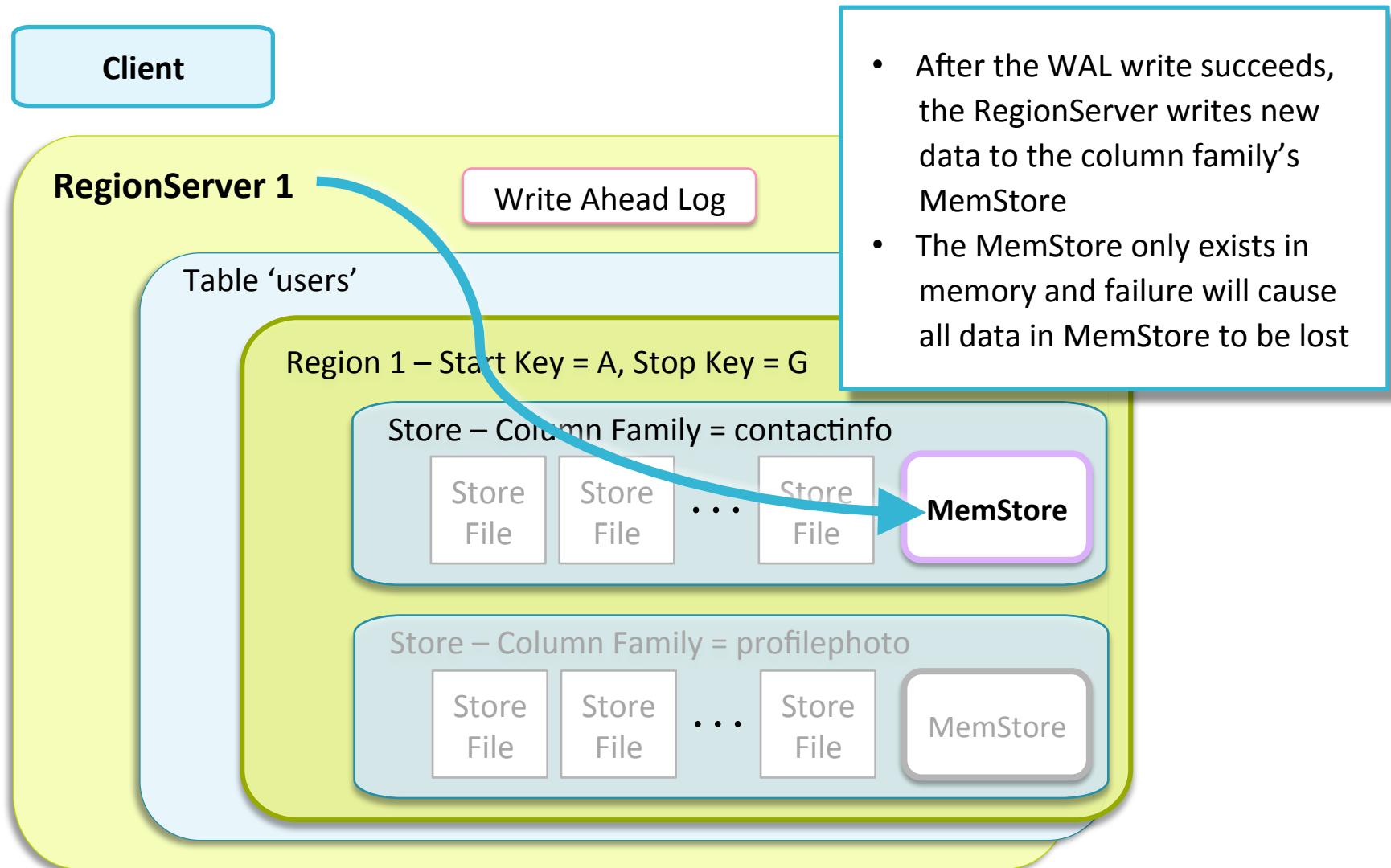
Write Path: Client Put



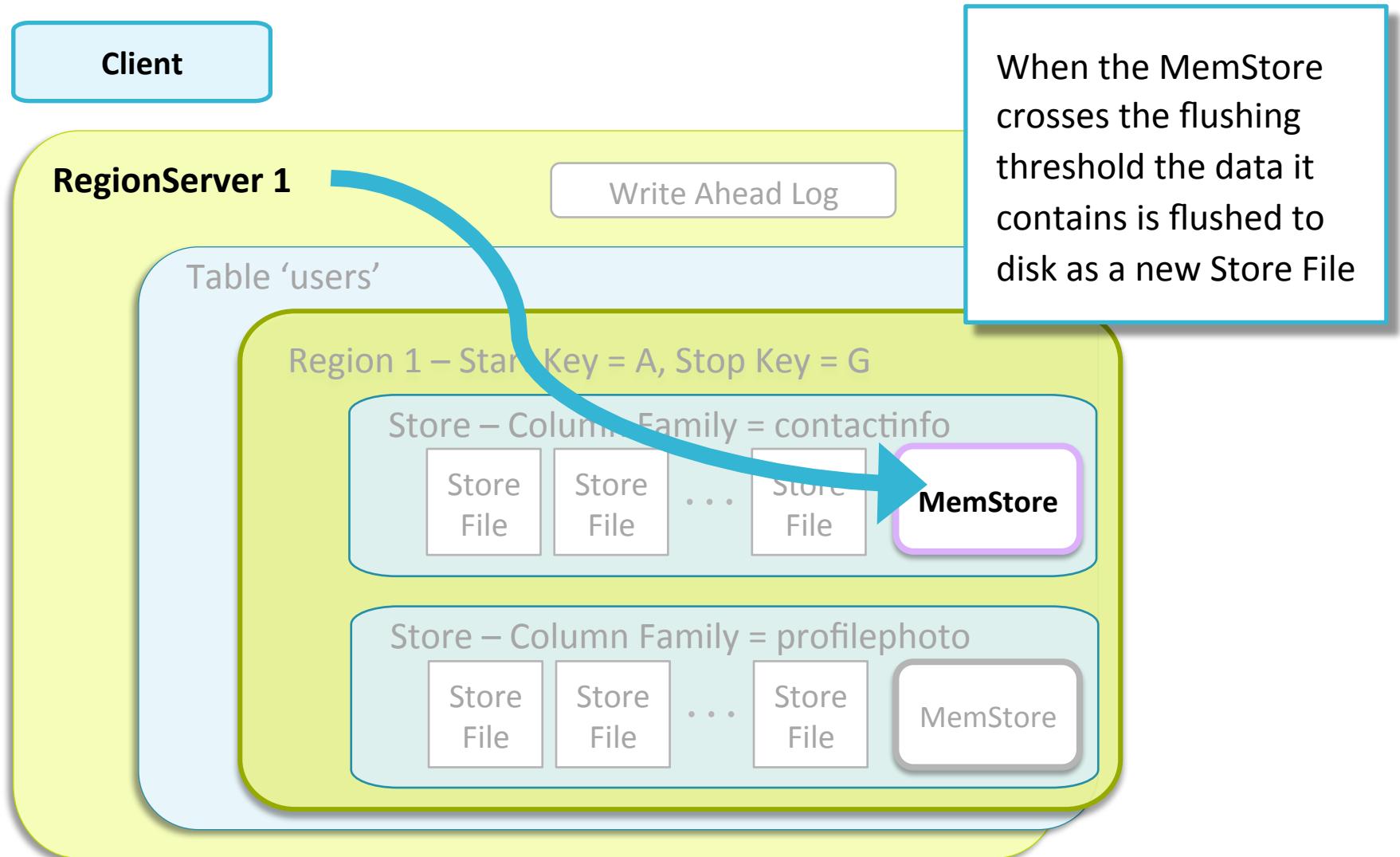
Write Path: Write Ahead Log



Write Path: MemStore Write



Write Path: MemStore Flush



Writing Data to Storage

- Client issues a **put** request to the RegionServer
- Details are handed to the appropriate region instance
- Possible to set WAL usage with the `writeToWAL` method
 - When `writeToWAL(true)` the data is written to the WAL, then the MemStore
 - When `writeToWAL(false)` the data is not written to the WAL and only to the MemStore
 - Default is `writeToWAL(true)`
 - If MemStore is full then it is flushed to disk
 - Flush is performed by a separate RegionServer thread

HBase File Types (1)

- **HLog**

- Used for the WAL data format
- Stored as a standard Hadoop SequenceFile
- Stores HLogKeys as well as actual data
- HLogKeys are used to replay not-yet-persisted data after a server crash

HBase File Types (2)

- **HFiles are the format used for Store Files**
 - HFiles are broken into smaller pieces called *blocks*
 - Default block size is 64KB
 - An entire block is read whenever a read operation is performed
- **Note: HBase blocks are not the same thing as HDFS blocks**
- **A single HFile is made up of many blocks**
 - An HFile can be gigabytes in size in HDFS
- **HBase Block sizes are configured at the column family level**
 - A smaller block size improves random access speeds
 - A larger block size improves scan performance

Data Storage

- **MemStore is flushed to a Store File when one of the following occurs:**
 - `hbase.regionserver.global.MemStore.upperLimit` is reached
 - Default is 0.4, or 40% of heap
 - `hbase.hregion.MemStore.flush.size` is reached
 - Default is 128MiB
 - `hbase.hregion.preclose.flush.size` is reached and the region is being closed
 - Default is 5MiB
- **Flushes can block updates**
 - This happens when the size in bytes of the memstore reaches :
`hbase.hregion.MemStore.block.multiplier * hbase.hregion.MemStore.flush.size`
 - Default values:
 - `hbase.hregion.MemStore.block.multiplier = 2`
 - `hbase.hregion.MemStore.flush.size = 134217728`

RegionServer Crash Restart Process

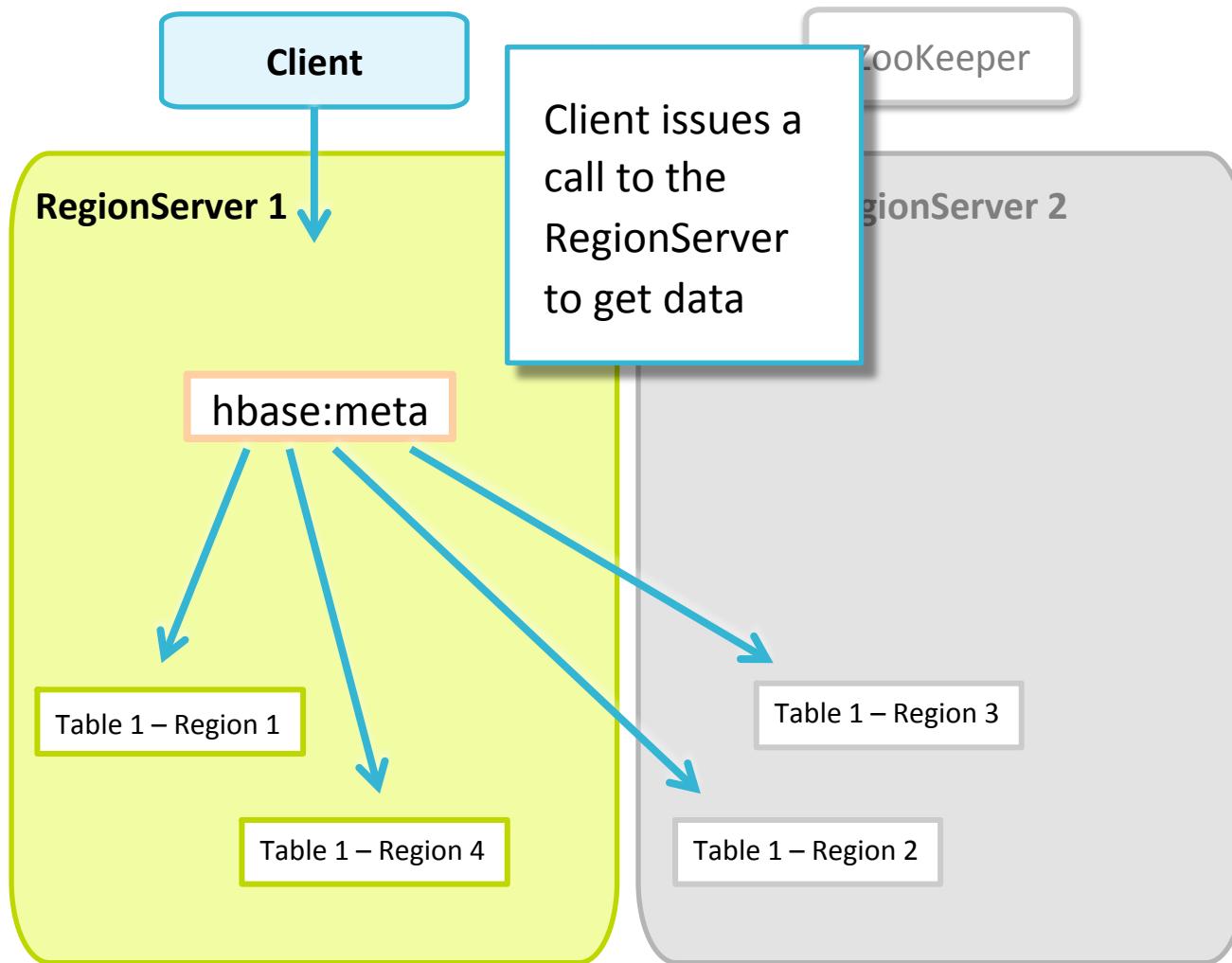
- **If a RegionServer crashes, since the MemStore is in memory its data is lost**
 - However, recent updates would also be in the WAL, stored in HDFS
 - WAL is used to reapply the recent changes
- **Region store files (HFiles) are stored in HDFS**
 - HDFS replicates storage blocks automatically by default
 - All nodes in the cluster can access files stored in HDFS
- **Master detects the RegionServer failure**
 - The Master coordinates the process of log splitting
 - The WAL splitting is performed by RegionServers on the cluster
 - The Master reassigned regions to other RegionServers
 - After splitting has completed the RegionServers apply the relevant portion(s) of the WAL

Chapter Topics

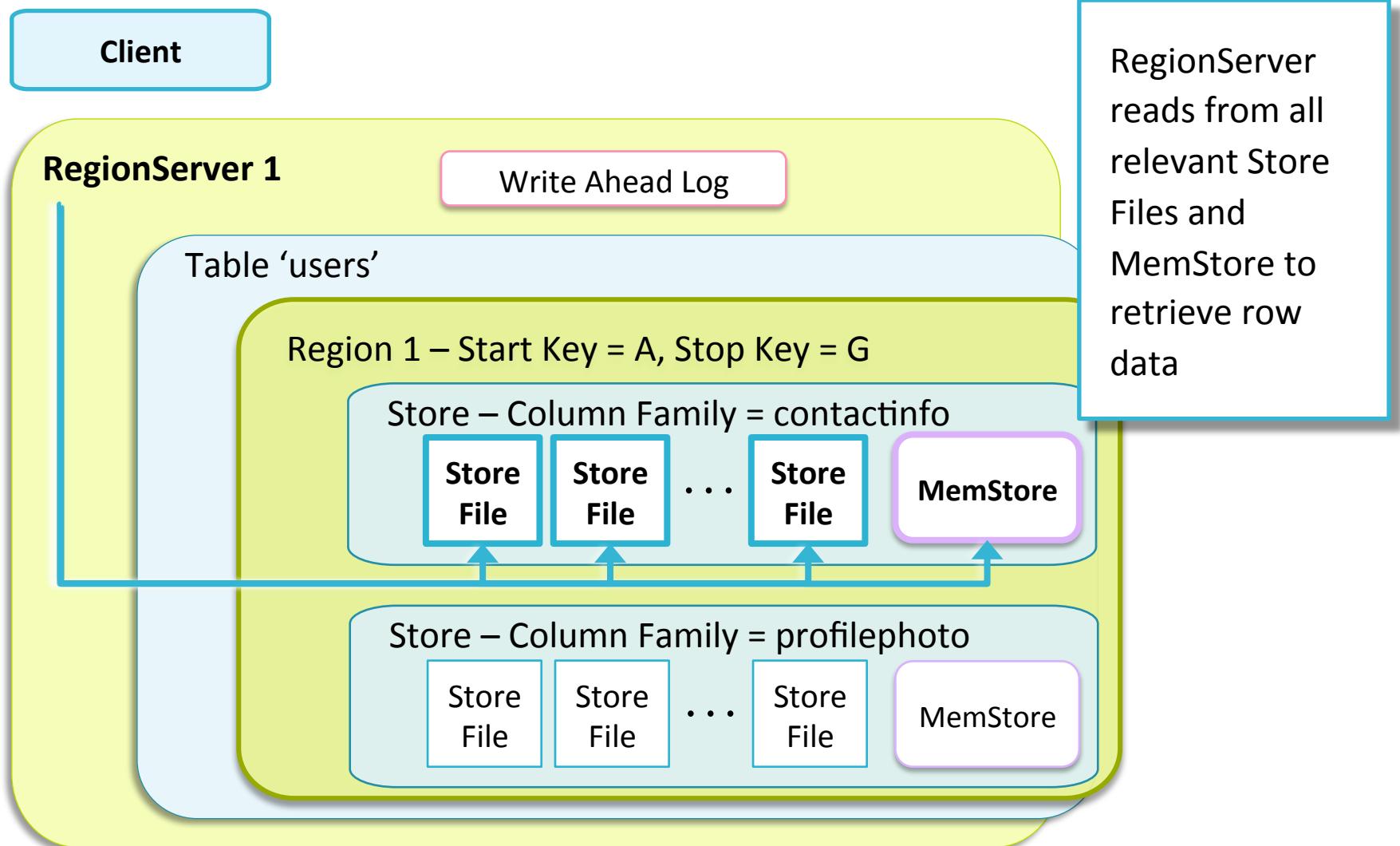
HBase Reads and Writes

- How HBase Writes Data
- **How HBase Reads Data**
- Block Caches for Reading
- Conclusion

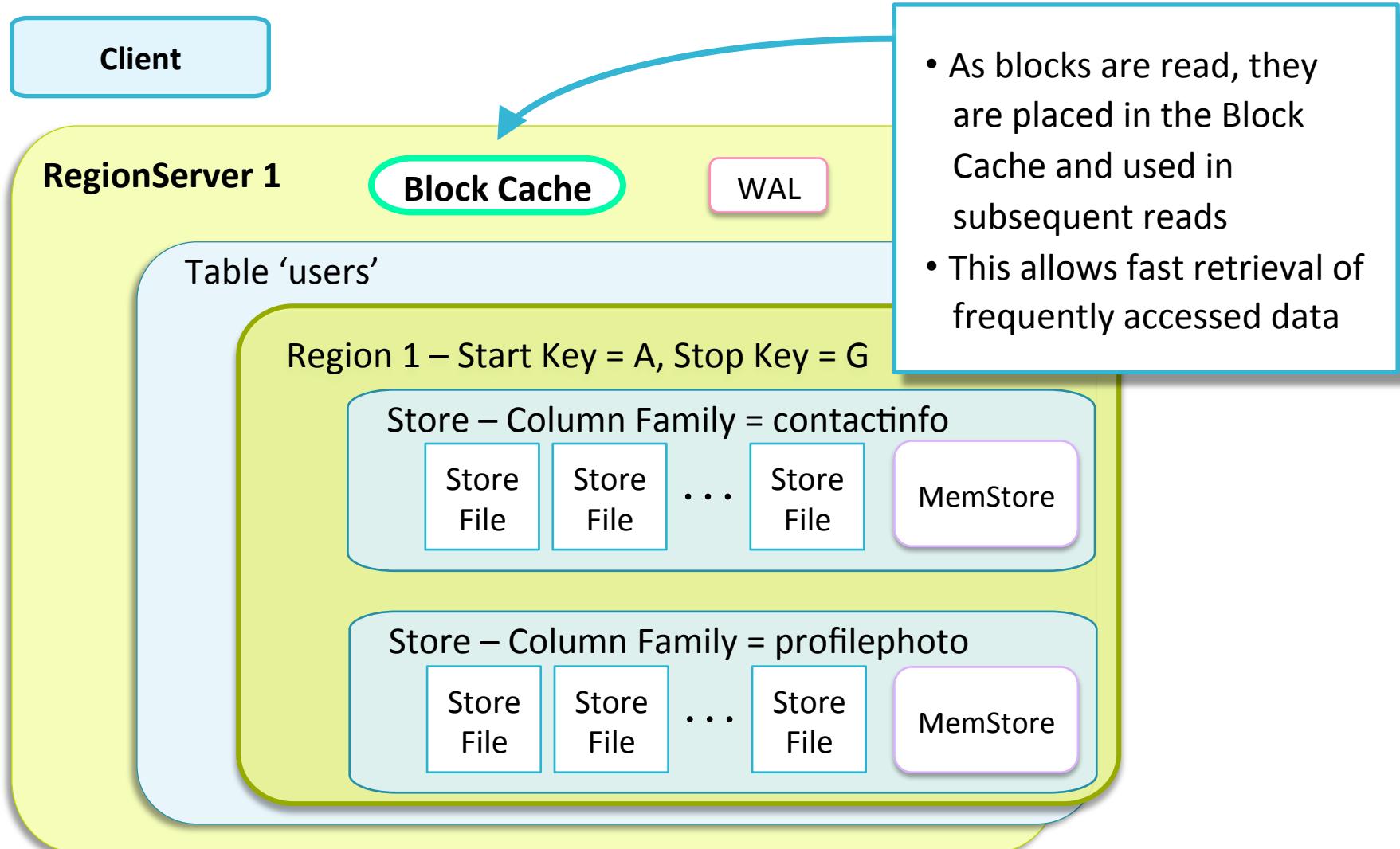
Read Path: Client Get



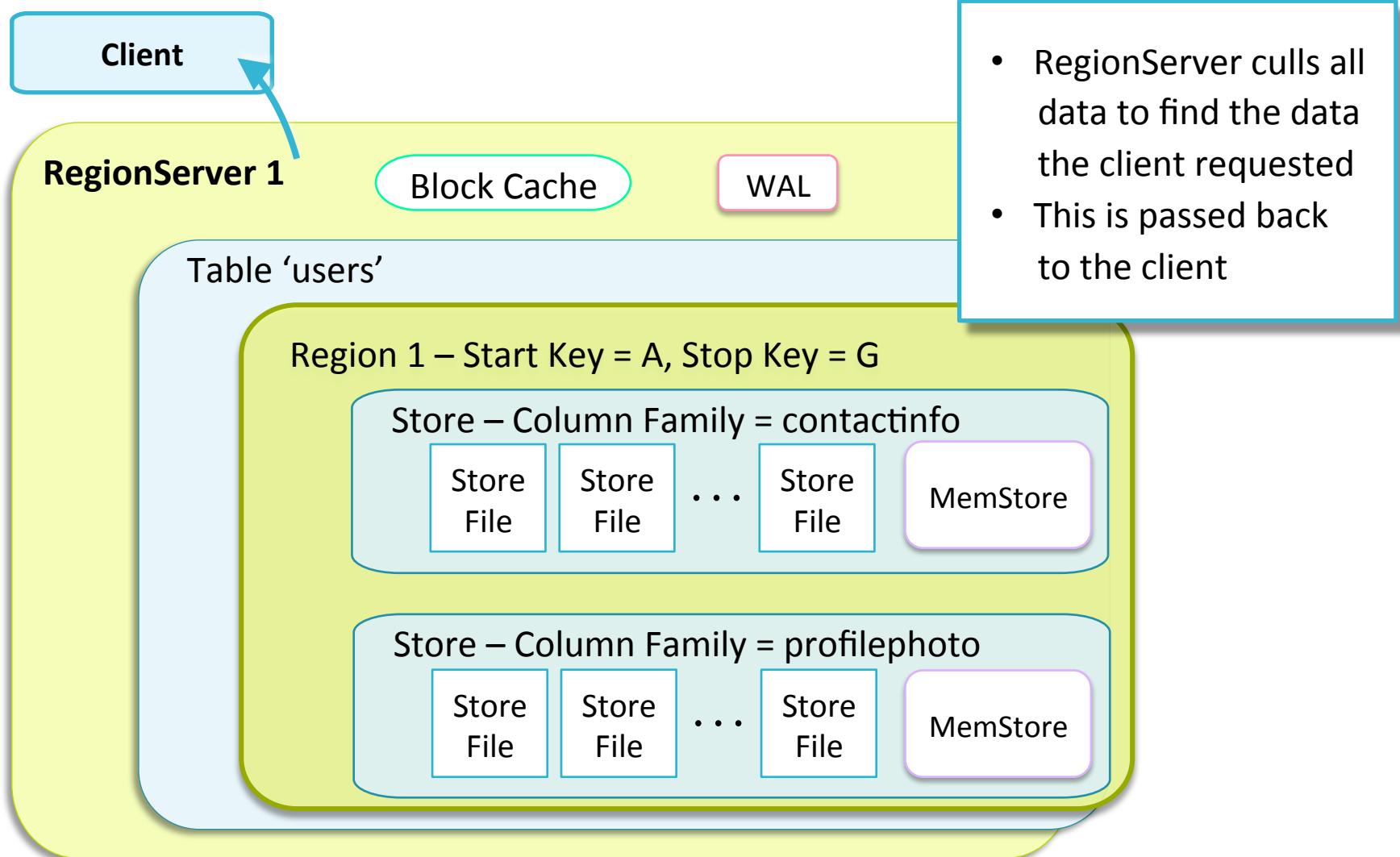
Read Path: Reading Store Files and MemStore



Read Path: Block Cache



Read Path: Find RegionServer



Chapter Topics

HBase Reads and Writes

- How HBase Writes Data
- How HBase Reads Data
- **Block Caches for Reading**
- Conclusion

Block Cache

- The HBase block cache facilitates fast reads by caching metadata and column family data reads
- The two types of block cache implementations in HBase are LruBlockCache and BucketCache
- **LruBlockCache is the original HBase block cache**
 - Uses only onheap memory (JVM heap) for its cache
- **BucketCache is a new block cache option that is not limited to onheap memory**
 - Can be configured offheap whereby it uses memory outside the JVM heap
- The eviction policy of both caches is based on the Least Recently Used (LRU) policy

Caching Operation

- There are multiple configurations for LruBlockCache and BucketCache

LruBlockCache	<ul style="list-style-type: none">• LruBlockCache caches both metadata and column family data reads• This is the default operation when BucketCache is disabled
Combined Cache	<ul style="list-style-type: none">• LruBlockCache stores metadata only• BucketCache stores column family reads• This is the default operation once BucketCache is enabled
Raw L1+L2	<ul style="list-style-type: none">• Level 1 cache (LruBlockCache) caches both metadata and column family reads• Blocks evicted from Level 1 cache flow to the Level 2 cache (BucketCache)• This operation is enabled by setting <code>hbase.bucketcache.combinedcache.enabled</code> to <code>false</code> in <code>hbase-site.xml</code> (default is <code>true</code>)

Configuring Caching

- Setting the `IN_MEMORY` configuration for a Column Family to `true` ensures that data from the column family is only evicted from the cache when absolutely necessary

BLOCKCACHE	IN_MEMORY	Result
False	False	Never cache
True	False	Regular LRU caching
True	True	Caching with priority

Enabling and Configuring BucketCache

- **BucketCache is enabled by setting `hbase.bucketcache.size` to a value greater than zero**
- **BucketCache is deployed offheap by default**
 - To use onheap memory, set `hbase.bucketcache.ioengine` in `hbase-site.xml` to `heap`
 - To use file-based cache, set `hbase.bucketcache.ioengine` to `file:PATH_TO_FILE`
- **Using BucketCache helps to reduce JVM garbage collection pressure experienced by the RegionServers**
 - This is because fewer objects are stored onheap by LruBlockCache
 - In offheap mode, BucketCache manages all memory allocations (no GC involved)

Disabling Caching

- You can disable caching of data reads on a per-column family basis
 - From the HBase shell: Set BLOCKCACHE to false for each column family whose reads should not be cached
 - From the Java API: Use the `setCacheBlocks (false)` method to bypass the BlockCache for a given scan or get operation
- It is not possible to disable metadata caching

LruBlockCache – A Detailed Look

- **The default block size for the LruBlockCache is 64KB**
- **LruBlockCache caches the following metadata**
 - hbase:meta table data that identifies which RegionServers are serving specific regions
 - HFile indexes which allow HBase to quickly find data within an HFile
 - Bloom filters if enabled (they help reduce lookup times)
- **When the LruBlockCache is full, the least recently used block will be evicted according to access priority (see next slide)**

LruBlockCache – Access Priorities

- Access priority groups are used by the caching system when selecting blocks to evict
- Blocks in the LruBlockCache are grouped into one of three access priority groups

Access Priority Groups	
Single	<ul style="list-style-type: none">• Typical priority assigned to a block upon first load from HDFS• First blocks to be evicted
Multi	<ul style="list-style-type: none">• A given block is promoted from single to multi access priority group on second access of the block• Blocks are evicted if there are no single access priority blocks to evict
In-memory	<ul style="list-style-type: none">• Assign in-memory priority by setting <code>IN_MEMORY</code> to true for a given column family• Blocks in this group are the last to be evicted• Catalog tables have in-memory access priority by default

BucketCache – A Detailed Look

- **BucketCache manages areas of memory organized as ‘buckets’**
 - Less garbage collection when BucketCache is used because it manages BlockCache allocations, not the GC
 - If the BucketCache is deployed offheap, the memory is not managed by the GC at all
- **Offheap BucketCache**
 - Fetches are slower with offheap cache than with onheap
 - Latency is less variable over time because there is less garbage collection

Block Cache Reporting

- The RegionServer web interface at port 60030 shows details about the block cache
 - Current configuration settings
 - Current usage
 - Time-in-the-cache
 - Block counts
 - Block types

Chapter Topics

HBase Reads and Writes

- How HBase Writes Data
- How HBase Reads Data
- Block Caches for Reading
- **Conclusion**

Key Points

- Reads and writes are distributed across RegionServers based on the row key
- HBase provides two types of read caches for onheap or offheap caching



HBase Performance Tuning

Chapter 11



Course Chapters

- Introduction
- Introduction to Hadoop and HBase
- HBase Tables
- HBase Shell
- HBase Architecture Fundamentals
- HBase Schema Design
- Basic Data Access with the HBase API
- More Advanced HBase API Features
- HBase on the Cluster
- HBase Reads and Writes
- **HBase Performance Tuning**
- HBase Administration and Cluster Management
- HBase Replication and Backup
- Using Hive and Impala with HBase
- Conclusion
- Appendix: Using Python and Thrift to Access HBase Data
- Appendix: OpenTSDB

HBase Performance Tuning

In this chapter you will learn

- **How to work with column families**
- **What to consider when designing the schema**
- **Some issues to consider when determining optimal memory size for the RegionServer daemons**

Chapter Topics

HBase Performance Tuning

- **Column Family Considerations**
- Schema Design Considerations
- Configuring for Caching
- Memory Considerations
- Dealing with Time Series and Sequential Data
- Pre-Splitting Regions
- Hands-On Exercise: Detecting Hot Spots
- Conclusion

Column Family Design

- **Column family names must use printable characters**
- **Use short column family and descriptor names**
 - Each row in the HFile contains both, so long names waste space
- **Recommendation: No more than three column families per table**
- **Column families allow for separation of data**
 - Design column families such that data that is accessed simultaneously is stored in the same column family

Column Family Considerations

- **Flushing and compaction occur per region**
 - Compaction is triggered by the number of store files per column family
 - If one column family is large (and therefore has a lot of files) the other column families will also be flushed from the Memstore
 - The more column families, the greater the I/O load
- **Attributes such as compression, Bloom filters and replication are set on a per column family basis**

Column Family Attributes

Attribute	Possible values	Default
COMPRESSION	NONE, GZ, LZO, SNAPPY	NONE
VERSIONS	1+	1
TTL	1-2147483647 (seconds)	FOREVER (special value, means the data is never deleted)
MIN VERSIONS	0+	0
BLOCKSIZE	1 byte - 2GB	64K
IN_MEMORY	true, false	false
BLOCKCACHE	true, false	true
BLOOMFILTER	NONE, ROW, ROWCOL	NONE

- Note: other attributes exist; these are the most common

COMPRESSION

- **Compression is recommended for most column families**
 - Not recommended for column families storing already compressed data such as JPEG or PNG
- **Compression codecs including GZIP, LZO, and Snappy are available**
 - Your choice of codec is a tradeoff between size and compression time
- **To enable compression on a column family with the codec of your choice:**

```
alter 'movie', { NAME => 'desc', COMPRESSION => 'SNAPPY' }
```

BLOCKSIZE

- **BLOCKSIZE specifies the minimum amount of data read during any read request**
 - Large values generally improve performance for scans
 - A small value is preferred if the workload typically consists of random reads

Bloom Filters

- A Bloom filter is a data structure which allows an existence check for a particular piece of data
 - The result of an existence check is a ‘no’ or a ‘maybe’
 - Can definitively say if a piece of data does not exist; cannot definitively say that a piece of data *does* exist
- HBase supports using Bloom filters to improve read performance
 - Eliminates the need to read every store file
 - Allows the RegionServer to skip files that do not contain the row or row and column
 - Cannot guarantee that a row *is* in the file
 - To enable:

```
alter 'movie', { NAME => 'desc', BLOOMFILTER => 'ROW' }
```

Bloom Filter Considerations

- **Good use cases**
 - Access patterns with lots of misses during reads
 - Speed up reads by cutting down on Store File reads
 - Update data in batches so that rows are in a few Store Files
- **Bad use cases**
 - All of the rows are updated regularly and the rows are spread across most Store Files
- **Bloom filters are kept in the Store File and add minimal overhead to storage**

Chapter Topics

HBase Performance Tuning

- Column Family Considerations
- **Schema Design Considerations**
- Configuring for Caching
- Memory Considerations
- Dealing with Time Series and Sequential Data
- Pre-Splitting Regions
- Hands-On Exercise: Detecting Hot Spots
- Conclusion

Schema Fundamentals (1)

- **Schema design brings together all fundamental aspects of HBase design**
 - Designing the keys (row and column)
 - Segregating data into column families
 - Choosing appropriate compression, block size, etc.
- **Similar techniques are needed to scale most systems**
 - Optimizing indexes, partitioning data, consistent hashing

Schema Fundamentals (2)

- **You must engineer to work around the limitations of HBase's architecture**
- **Joins are very costly**
 - Performed on the client side
 - Avoid wherever possible
 - Data should be denormalized to avoid joins
 - Code must be written to update denormalized data everywhere
- **Row keys must be more intelligent**
 - Take advantage of the sorting, and optimize reads

Tall-Narrow vs Flat-Wide Tables

- **There are two major approaches to table layouts**
 - Tall-Narrow tables have few columns, but more rows
 - Flat-Wide tables have many columns in a single row
- **Both layouts have the same storage footprint**
- **Tall-Narrow tables have less atomicity because all data is not in the same row**
- **Flat-Wide tables have rows that do not split**
 - You might end up with one row per region if the rows are extremely wide
- **Tall-Narrow tables typically put more details into the row key**
 - You can use partial key scans to reconstruct all of the data
- **Suggestion: Make tables tall if the access pattern is mainly scans, wide if gets**

Secondary Indexes

- What if you want to query on something other than the row key?
- If you are performing many ad-hoc queries you may need an RDBMS
- RDBMS vs. HBase
 - Both require additional space and processing
 - RDBMS is more advanced for index management
 - HBase scales better at larger data volumes

Secondary Indexes – HBase Options

- **Run a filter query using the API**
 - Not good for a full scan on a large table
- **Create a secondary index**
 - Create a secondary index as another table
 - Periodically update the table via a MapReduce job
 - Alternatively, dual-write while publishing data to the cluster
- **Create summary tables**
 - Good for very wide time-range data
 - Typically generated via MapReduce jobs to pre-compute otherwise on-the-fly queried data
 - For example, count the number of distinct instances of a value in a table and write those summarized counts into another table

Chapter Topics

HBase Performance Tuning

- Column Family Considerations
- Schema Design Considerations
- **Configuring for Caching**
- Memory Considerations
- Dealing with Time Series and Sequential Data
- Pre-Splitting Regions
- Hands-On Exercise: Detecting Hot Spots
- Conclusion

Caching Metadata and Data

- **The LruBlockCache is an onheap, memory-based cache used for reads**
 - Uses an LRU (least recently used) algorithm to evict data when the cache is full
- **The new BucketCache feature allows for setup of an L1 cache and an L2 cache**
 - Uses a similar algorithm to that used by LruBlockCache
 - The BucketCache is enabled by default
- **Disable caching of data for a given column family by setting BLOCKCACHE to false**
 - Avoids ‘polluting’ the cache with data from seldom-accessed column families

Configuring the L1 and L2 Caches

- **The new BucketCache feature allows setup of an L1 cache and an L2 cache that operate together in one of three ways**
 1. The default option dedicates the L1 cache to storing metadata only, while the L2 cache stores column family data
 2. Data read from specific column families can be directed to the L1 cache by setting `CACHE_DATA_IN_L1` to true
 3. Finally, set `BUCKET_CACHE_COMBINED_KEY` to false to operate BucketCache as a strict L2 cache to the L1 cache
- **Many factors must be considered when configuring caching**
 - On-heap vs. off-heap
 - Nature of the application (e.g., read-heavy vs. write-heavy)
 - Cache configuration requires tuning to achieve performance outcomes

Chapter Topics

HBase Performance Tuning

- Column Family Considerations
- Schema Design Considerations
- Configuring for Caching
- **Memory Considerations**
- Dealing with Time Series and Sequential Data
- Pre-Splitting Regions
- Hands-On Exercise: Detecting Hot Spots
- Conclusion

HBase Memory Considerations

- **Recommendation: Give the HBase Master JVM 1GB of RAM**
- **Recommendation: Give the RegionServer JVMs between 12GB and 16GB of RAM**
 - Larger heap sizes can cause garbage collection to take too long
 - Other Java garbage collection settings need to be configured
- **The remaining system memory is used by the kernel for caching**
- **Memory settings are configured in `hbase-env.sh`:**

```
export HBASE_REGIONSERVER_OPTS="-Xmx12g -Xms12g -Xmn128m  
-XX:+UseParNewGC -XX:+UseConcMarkSweepGC  
-XX:CMSInitiatingOccupancyFraction=70"  
  
export HBASE_MASTER_OPTS="-Xmx1000m"
```

Chapter Topics

HBase Performance Tuning

- Column Family Considerations
- Schema Design Considerations
- Configuring for Caching
- Memory Considerations
- **Dealing with Time Series and Sequential Data**
- Pre-Splitting Regions
- Hands-On Exercise: Detecting Hot Spots
- Conclusion

Problems with Hotspotting

- **Hotspotting occurs when a small number of RegionServers are handling the majority of the load**
 - This causes an uneven use of the cluster's resources
- **Hotspotting can happen if the row key is sequential or time series**
 - All writes will typically be to the same region

Approaches to Address Hotspotting

■ Salting

- Place a small, calculated hash in front of the real data to randomize the row key
- e.g., <salt><timestamp> instead of just <timestamp>

■ Promoted Field Keys

- A field is moved in front of the incremental or timestamp field
- e.g., <sourceid><timestamp> instead of <timestamp><sourceid>

■ Pseudo-Random

- Process the data using a one-way hash like MD5
- e.g., <md5 (timestamp) > instead of just <timestamp>

Chapter Topics

HBase Performance Tuning

- Column Family Considerations
- Schema Design Considerations
- Configuring for Caching
- Memory Considerations
- Dealing with Time Series and Sequential Data
- **Pre-Splitting Regions**
- Hands-On Exercise: Detecting Hot Spots
- Conclusion

Regions

- **We prefer fewer, larger regions over many, smaller regions**
- **Recommendation: RegionServers should serve between 20 to low hundreds of regions**
 - Too many regions on a RegionServer can cause scaling issues
- **The number of regions per RegionServer is not explicitly configured**
 - It is determined by the maximum region size and the amount of data
- **The size at which a region is split is the number of regions that are on this server that all are part of the same table, squared, times the region flush size or the maximum region split size, whichever is smaller**
 - Maximum region split size is configured in `hbase-site.xml` with the `hbase.hregion.max.filesize` parameter
 - 10GB is recommended as a starting value
 - Monitor, and modify as necessary

Pre-Splitting Regions

- **Regions can be pre-split to improve performance**
 - Pre-splitting a table and then loading data is more efficient
 - Example using the HBase shell:

```
hbase> create 'myTable', 'cf1', 'cf2', {SPLITS => ['A',  
'M', 'Z']}
```

- **Example: Bulk loading 100GB of data to a new table**
 - Start with a filesize of 10GB
 - Pre-split the regions so that each region gets ~10GB of data
 - If more data will be added, pre-split into 10-20 regions to avoid immediate splitting as more data is added

Region Splits

- **Region splits are an offline process**
 - The region is briefly taken offline during the split
- **Manually splitting a region can alleviate hotspotting in a table**
- **Splitting can be turned off by setting filesize to a large value such as 100GB**
 - Care must be taken verify that your splitting process continues to function
- **Regions can be merged together**
 - This is an online process initiated at the HBase shell using the command `merge_region`

Chapter Topics

HBase Performance Tuning

- Column Family Considerations
- Schema Design Considerations
- Configuring for Caching
- Memory Considerations
- Dealing with Time Series and Sequential Data
- Pre-Splitting Regions
- **Hands-On Exercise: Detecting Hot Spots**
- Conclusion

Hands-On Exercise: Detecting Hotspots

- In this Hands-On Exercise, you will test several row key types and detect the hotspots
- Please refer to the Exercise Manual

Chapter Topics

HBase Performance Tuning

- Column Family Considerations
- Schema Design Considerations
- Configuring for Caching
- Memory Considerations
- Dealing with Time Series and Sequential Data
- Pre-Splitting Regions
- Hands-On Exercise: Detecting Hot Spots
- **Conclusion**

Key Points

- Tables are typically either Tall-Narrow or Flat-Wide
- Compression and Bloom filters can optimize a column family without code changes
- Schema design takes into consideration all aspects of a table's design
- Tables can be used to provide secondary indexes



HBase Administration and Cluster Management

Chapter 12



Course Chapters

- Introduction
- Introduction to Hadoop and HBase
- HBase Tables
- HBase Shell
- HBase Architecture Fundamentals
- HBase Schema Design
- Basic Data Access with the HBase API
- More Advanced HBase API Features
- HBase on the Cluster
- HBase Reads and Writes
- HBase Performance Tuning
- **HBase Administration and Cluster Management**
- HBase Replication and Backup
- Using Hive and Impala with HBase
- Conclusion
- Appendix: Using Python and Thrift to Access HBase Data
- Appendix: OpenTSDB

HBase Administration and Cluster Management

In this chapter you will learn

- The role of each HBase daemon
- How ZooKeeper adds high availability
- How to administer HBase
- Security in HBase

Aside: Existing Cluster, Using Cloudera Manager

- This course assumes that you have an existing Hadoop cluster, managed by Cloudera Manager
- Installation of HBase is performed by adding the HBase Service to the cluster

Chapter Topics

HBase Administration and Cluster Management

- **HBase Daemons**
- ZooKeeper Considerations
- HBase High Availability
- Using the HBase Balancer
- Fixing HBase Tables with hbck
- HBase Security
- Conclusion

HBase Master

- **The HBase Master handles many critical functions for an HBase cluster**
- **Coordinates the RegionServers and manages failovers**
 - Handles reassignment of regions to other RegionServers
- **Handles table and column family changes and additions**
 - Updates hbase:meta accordingly
- **Manages region changes such as finalizing a region split or assignment**
- **Multiple Masters can run at the same time**
 - Only one Master is active
 - The other Masters are competing to become active if the active Master fails

HBase Master Background Processes

- HBase has a LoadBalancer process for spreading a table's regions across many RegionServers
- The Catalog Janitor process checks for unused regions to garbage collect
- The Log Cleaner process deletes old WAL files

RegionServer (1)

- **The RegionServer handles the data movement**
 - Reads all data for gets and scans and passes the data back to the client
 - Stores all data from puts from clients
 - Records all deletes from clients
- **Handles all compactions**
 - Both major and minor compactions
- **Handles all region splitting**

RegionServer (2)

- **Maintains the Block Cache**
 - Catalog tables, indexes, bloom filters, and data blocks are all maintained in the Block Cache
- **Manages the Memstore and WAL**
 - Puts and Deletes are written to the WAL then added to the Memstore
 - The Memstore is occasionally flushed
- **Receives new regions from the Master to serve**
 - Replays the WAL if necessary

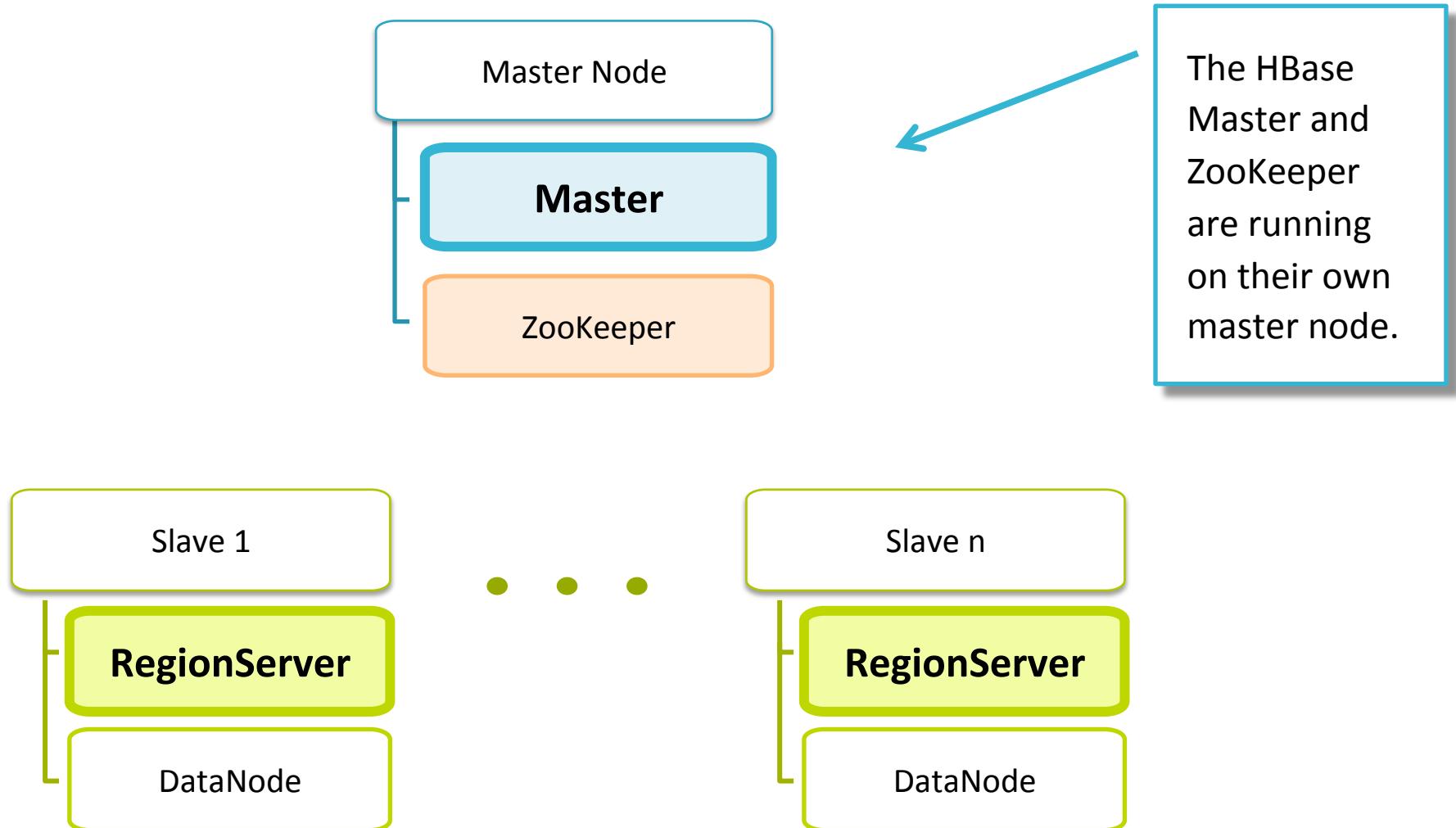
RegionServer Background Processes

- **The RegionServer has a CompactSplitThread process**
 - Looks for regions that need to be split
 - Larger than the maximum size
 - Handles the minor compactions
- **The MajorCompactionChecker checks to see if a major compaction needs to be run**
- **The MemstoreFlusher checks if the Memstore is too full and needs to be flushed to disk**
- **The LogRoller closes the WAL and creates a new file**

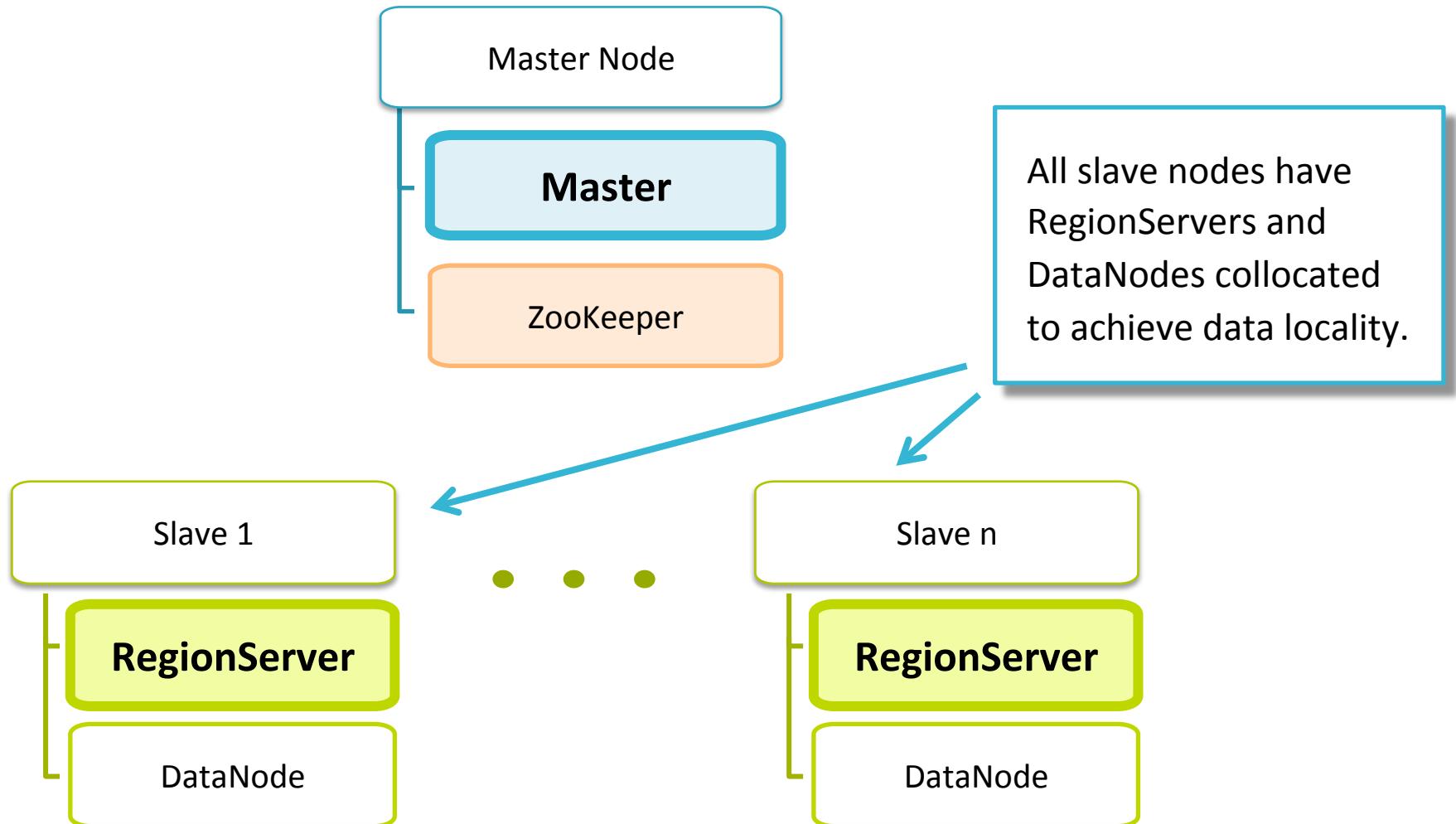
Placement of HBase Daemons

- A proof of concept cluster can have all master nodes collocated on a single node
 - This includes the HBase Master, ZooKeeper, NameNode and Secondary NameNode
- A small production cluster should have the HBase Master and ZooKeeper on a separate node from other master nodes in the cluster
 - For a production cluster, consider deploying a highly available cluster with a backup HBase Master
- Clusters of any size will have the RegionServers and DataNodes collocated on the same nodes
 - This gives the RegionServer data locality when reading and writing data

Small HBase Cluster Diagram: Master Node



Small HBase Cluster Diagram: Slave Nodes



Chapter Topics

HBase Administration and Cluster Management

- HBase Daemons
- **ZooKeeper Considerations**
- HBase High Availability
- Using the HBase Balancer
- Fixing HBase Tables with hbck
- HBase Security
- Conclusion

What is ZooKeeper?

- **ZooKeeper is a distributed coordination service**
 - It allows configuration information and locking to be distributed across many nodes
 - The distributed nature of ZooKeeper allows services to use ZooKeeper for difficult, distributed problems
- **A group of ZooKeeper nodes that are configured to work together is called a *quorum***
 - All data is kept in sync across all nodes in the quorum
 - Having several nodes in a quorum makes ZooKeeper highly available
- **HBase makes extensive use of ZooKeeper to maintain state and configuration information**

ZooKeeper Data

- **ZooKeeper presents data as a shared, hierachal namespace**
 - Similar to a file and directory structure
 - These files and directories are called *znodes*
- **ZooKeeper stores all data in memory**
 - This allows ZooKeeper to respond as quickly as possible
 - Transactions logs and snapshots of the data are stored to disk
- **All data is replicated across the entire quorum**

Highly Available HBase

- **HBase uses ZooKeeper to be highly available**
- **The ZooKeeper quorum allows multiple HBase Masters to run**
 - Each HBase Master competes for a lock that is managed by ZooKeeper
 - If a Master fails to acquire the lock in a time, the other HBase Masters will vie for the lock
 - The ZooKeeper quorum should not be collocated with DataNodes, RegionServers or NodeManagers (MR2)
- **A stand-alone ZooKeeper quorum will need to be started**
 - All nodes and clients must be able to access ZooKeeper ensemble
 - A quorum of 3 has high availability
 - A quorum of 5 has high availability and extra nodes for maintenance
- **Other services like HDFS will need to have high availability enabled**

ZooKeeper

- **By default, HBase provides a managed ZooKeeper installation**
 - HBase will start and stop a ZooKeeper process with the Master
 - For production HBase clusters a stand-alone ZooKeeper cluster should be used
 - This is the default when HBase is installed using Cloudera Manager

ZooKeeper Considerations

- **ZooKeeper is very sensitive to latency and clock skew**
 - Need to have sufficient bandwidth and a network that is not over-subscribed
 - Need to configure NTP to keep clock times synchronized across the cluster
- **If collocating the HBase Master daemon with ZooKeeper, ZooKeeper should have a separate disk**
 - Reduces I/O contention and improves ZooKeeper performance
 - Any shared resources should be as high-performing as possible

Chapter Topics

HBase Administration and Cluster Management

- HBase Daemons
- ZooKeeper Considerations
- **HBase High Availability**
- Using the HBase Balancer
- Fixing HBase Tables with hbck
- HBase Security
- Conclusion

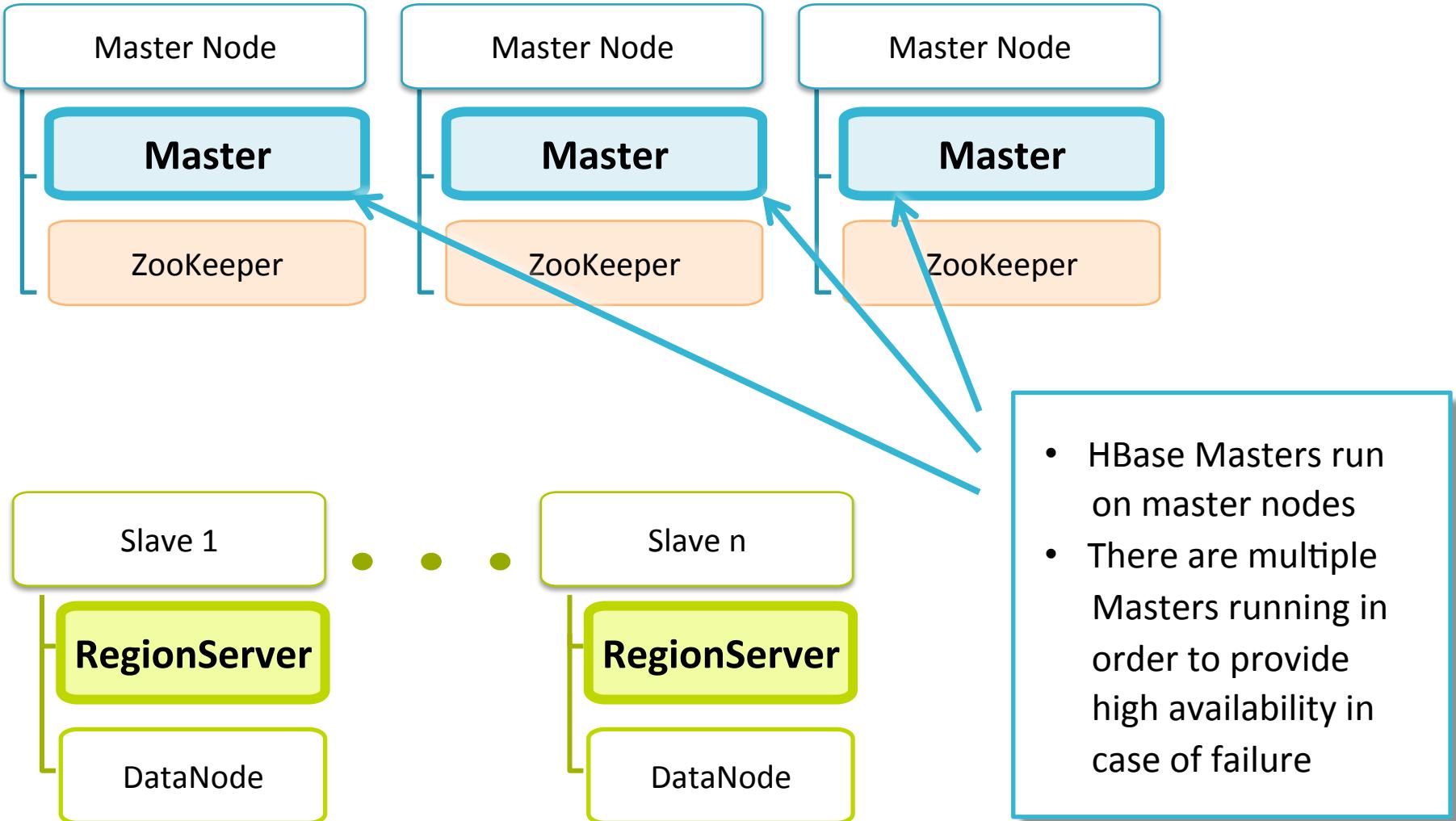
Detecting Failures with ZooKeeper (1)

- **HBase uses ZooKeeper to detect failures**
 - This includes both HBase Master and RegionServer failures
- **There is a significant amount of development work currently being done on detecting and recovering from failures faster**
 - This is called MTTR (Mean Time To Recovery)
 - HBase committers are working to lower MTTR to under one minute
 - Reducing the MTTR is difficult because of the size of data and memory usage

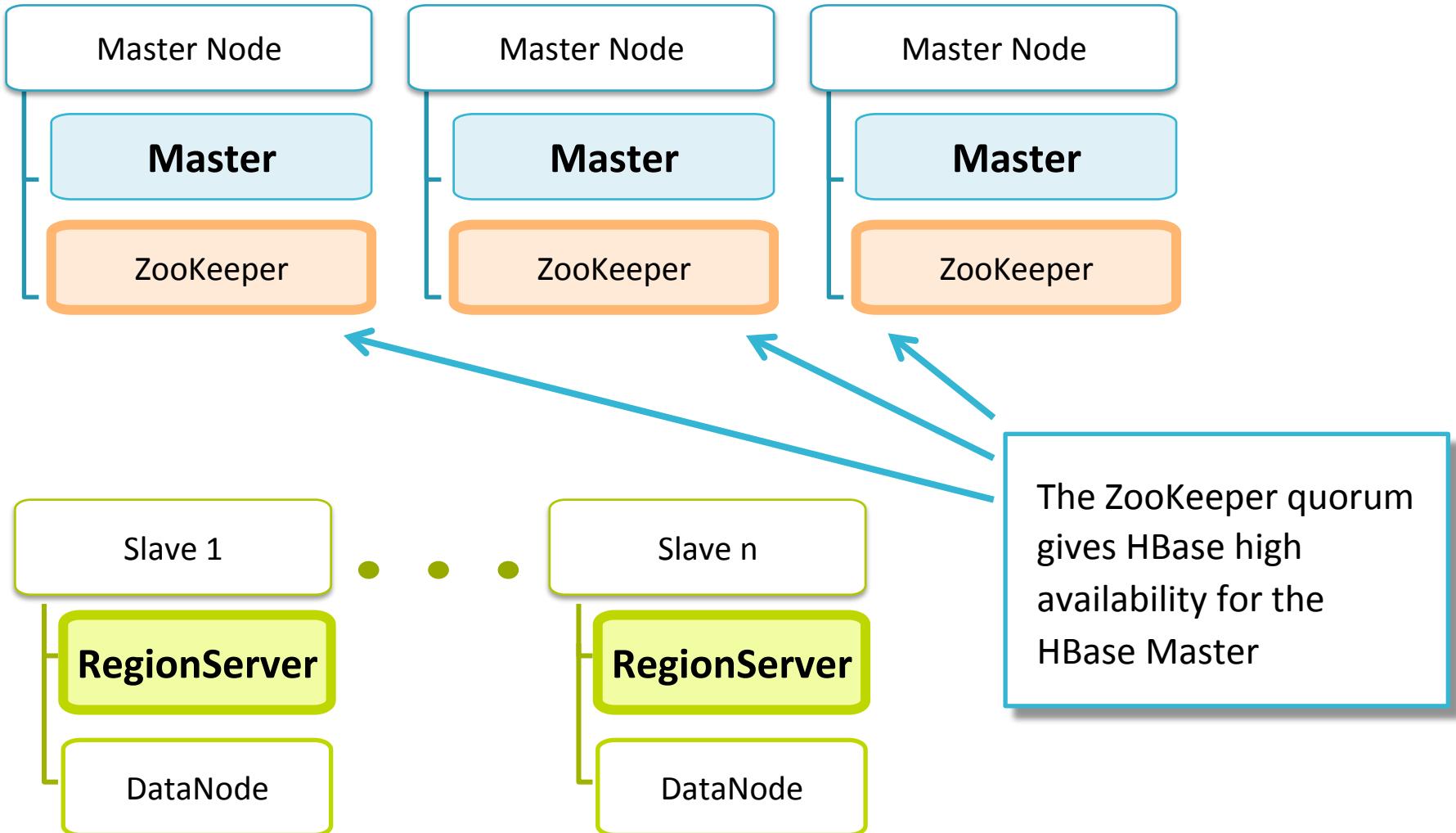
Detecting Failures with ZooKeeper (2)

- **A JVM garbage collection can cause long pauses**
 - During these pauses, nothing can execute in the JVM
 - These pauses can cause a RegionServer to timeout
 - This will cause it to be treated as if it had failed
 - Effort should be taken to decrease garbage collection times
- **HBase's default timeout is 90 seconds**
 - The RegionServer must check in with ZooKeeper before this timeout or it will be perceived as a RegionServer crash
 - Recommendation: in Cloudera Manager, reduce this to 60 seconds

Highly Available HBase Cluster Diagram: HBase Masters



Highly Available HBase Cluster Diagram: ZooKeeper Quorum



Chapter Topics

HBase Administration and Cluster Management

- HBase Daemons
- ZooKeeper Considerations
- HBase High Availability
- **Using the HBase Balancer**
- Fixing HBase Tables with hbck
- HBase Security
- Conclusion

What the Balancer Does

- There are two balancers in a Hadoop cluster that runs HBase
- HBase has a load balancer that balances regions served by a RegionServer
- HDFS has a balancer that balances the blocks in HDFS by moving them to different DataNodes
 - WARNING: the HDFS balancer is not HBase-aware and will temporarily destroy RegionServer data locality!
- Adding new nodes may require you to run the HDFS balancer
 - The RegionServers will have data locality again after a major compaction

How to Use the Balancer

- **HBase has a LoadBalancer process**
 - Balances based on the number of regions served by a RegionServer
 - The balancer is table aware and will spread a table's regions across many RegionServers
 - Note: does not take into account system resources
- **The balancer runs every five minutes by default**
- **During a move, the table is briefly offline**
- **Regions that are moved will lose data locality until a major compaction**

Using the HDFS Balancer: Caution

- **The HDFS Balancer moves disk blocks on the cluster to ensure that each DataNode is equally utilized**
- **The HDFS balancer is likely to cause loss of HBase data locality**
 - Blocks which were local to a RegionServer may be moved to another node
 - Running a major compaction will regain data locality
 - Also, as flushes are performed, data slowly becomes local again

Chapter Topics

HBase Administration and Cluster Management

- HBase Daemons
- ZooKeeper Considerations
- HBase High Availability
- Using the HBase Balancer
- **Fixing HBase Tables with hbck**
- HBase Security
- Conclusion

HBase Errors

- HBase has strong requirements for how data is served
- Every region must be assigned and deployed to one RegionServer
 - All catalog tables must have correct RegionServer information
- Every table's row key must resolve to just one region
- Data about a region is written to HDFS and HBase catalog tables
 - HBase can fix itself by reading in the data from HDFS
 - Then update the catalog tables to match
- These inconsistencies are rare and can happen due to bugs or crashes

Fixing HBase (1)

- **Some issues can be fixed online while others have to be repaired offline**
 - An offline check means that HBase is running, but there is no activity on the cluster
- **Inconsistencies can happen at the table and region level**
 - A region's metadata is not stored in HDFS (offline)
 - A region can overlap with another region (offline)
 - A region is not assigned to a RegionServer (online)
 - A region is served by more than one RegionServer (online)
 - A portion of a table's row keys are not contained within a region (online)
 - `hbase:meta` shows an incorrect assignment for a region (online)

Fixing HBase (2)

- **Some inconsistencies may be transient**
 - An error can appear during a transition period but the next check will not show any problems
- **Checks should be run before upgrading to a new version of HBase**
- **Each repair operation is a different level of severity and difficulty to fix**
 - The offline issues are the most complicated to fix
- **General steps to repair an issue:**
 - Copy the table to a separate cluster
 - Run the fix in a separate cluster first to resolve and verify any issues
 - Run the fix in the original cluster

Fixing HBase with hbck (1)

- **HBase has a consistency check utility called hbck**
 - Similar to filesystem check utilities
- **To run hbck:**

```
$ sudo -u hbase hbase hbck
```

- Will check that all tables and regions pass the HBase requirements but will not make any changes
 - Run hbck as the hbase user to have file access permissions
- **If there are no issues with HBase the final hbck output will say:**

```
Status: OK
```

Fixing HBase with hbck (2)

- If there are issues with HBase the final hbck output will say, e.g.:

```
2 inconsistencies detected.  
Status: INCONSISTENT
```

- Inconsistencies can be transient and caused by a normal HBase operation
 - Rerun the check after some time to verify
- Inconsistencies can be fixed with hbck run in repair mode

```
$ sudo -u hbase hbase hbck -repair
```

- There are other more specific repair options available
 - Check the documentation
- Warning: Only attempt hbck –repair after all other resources are exhausted

Fixing HBase with hbck (3)

- After a repair, hbck will run another consistency check to verify all issues are fixed
- A successful repair should output:

```
0 inconsistencies detected.  
Status: OK
```

- If the repair shows inconsistencies after a repair:
 - Run another hbck check to verify
 - Check for exceptions or errors in the output
 - Consult the documentation

Chapter Topics

HBase Administration and Cluster Management

- HBase Daemons
- ZooKeeper Considerations
- HBase High Availability
- Using the HBase Balancer
- Fixing HBase Tables with hbck
- **HBase Security**
- Conclusion

HBase Security

- HBase has built-in security available to limit access and permissions
- Enabling security requires a Kerberos-enabled Hadoop cluster
- Users are granted certain permissions
 - Permissions are granted globally or at a namespace, table, column family, column descriptor, or per-cell level
 - The users are given access permissions as read, write, execute, create or admin privileges
- Enabling security may decrease maximum throughput by 5-10%

Configuring HBase Security

- HBase must be configured to use a secure RPC engine
- The Kerberos principles must be added to the HBase configuration
- Masters, RegionServers and clients must use a secure ZooKeeper connection
- The HBase coprocessor that controls access must be enabled
- User access to HBase must be granted
- See the Cloudera HBase documentation for more information

HBase Shell – Granting Access (1)

- **Users are granted access to resources in the HBase shell**
 - Permissions are similar to Unix's chmod command
- **The user permissions are:**
 - R - read from a table
 - W - write to a table
 - X - execute a coprocessor on a table
 - C - create, alter, and delete tables
 - A - Administer or manage tables
- **To grant a permission:**

```
hbase> grant 'username', 'permissions', <table>, <colfam>,
      <coldesc>
```

HBase Shell – Granting Access (2)

- To grant create, alter, and delete permissions for all tables:

```
hbase> grant 'username', 'C'
```

- To grant read and write permissions for a table:

```
hbase> grant 'username', 'RW', 'movie'
```

- To grant read permissions for a specific column descriptor:

```
hbase> grant 'username', 'R', 'movie', 'desc', 'title'
```

- To revoke a permission for a user:

```
hbase> revoke 'username', 'RW'
```

Chapter Topics

HBase Administration and Cluster Management

- HBase Daemons
- ZooKeeper Considerations
- HBase High Availability
- Using the HBase Balancer
- Fixing HBase Tables with hbck
- HBase Security
- **Conclusion**

Key Points

- Users can be given access to specific tables, column families, and column descriptors via Kerberos authentication
- The HBase Masters and RegionServers perform many operations in the background
- ZooKeeper helps to provides HBase high availability
- HBase inconsistencies can be repaired with hbck



HBase Replication and Backup

Chapter 13



Course Chapters

- Introduction
- Introduction to Hadoop and HBase
- HBase Tables
- HBase Shell
- HBase Architecture Fundamentals
- HBase Schema Design
- Basic Data Access with the HBase API
- More Advanced HBase API Features
- HBase on the Cluster
- HBase Reads and Writes
- HBase Performance Tuning
- HBase Administration and Cluster Management
- **HBase Replication and Backup**
- Using Hive and Impala with HBase
- Conclusion
- Appendix: Using Python and Thrift to Access HBase Data
- Appendix: OpenTSDB

HBase Replication and Backup

In this chapter you will learn

- **How to set up HBase Replication**
- **How to back up data**
- **How to deal with MapReduce and HBase on the same cluster**

Chapter Topics

HBase Replication and Backup

- **HBase Replication**
- HBase Backup
- MapReduce and HBase Clusters
- Hands-On Exercise: Administration
- Conclusion

Replication and Backup

- **There are several strategies for HBase replication**
 - Have the applications do puts and deletes to all clusters
 - Use HBase's built-in replication
 - Manually copy tables between clusters
- **HBase supports inter-cluster replication**
 - Newly added and deleted data is automatically added to another cluster
 - Replication is done on a per-column family basis

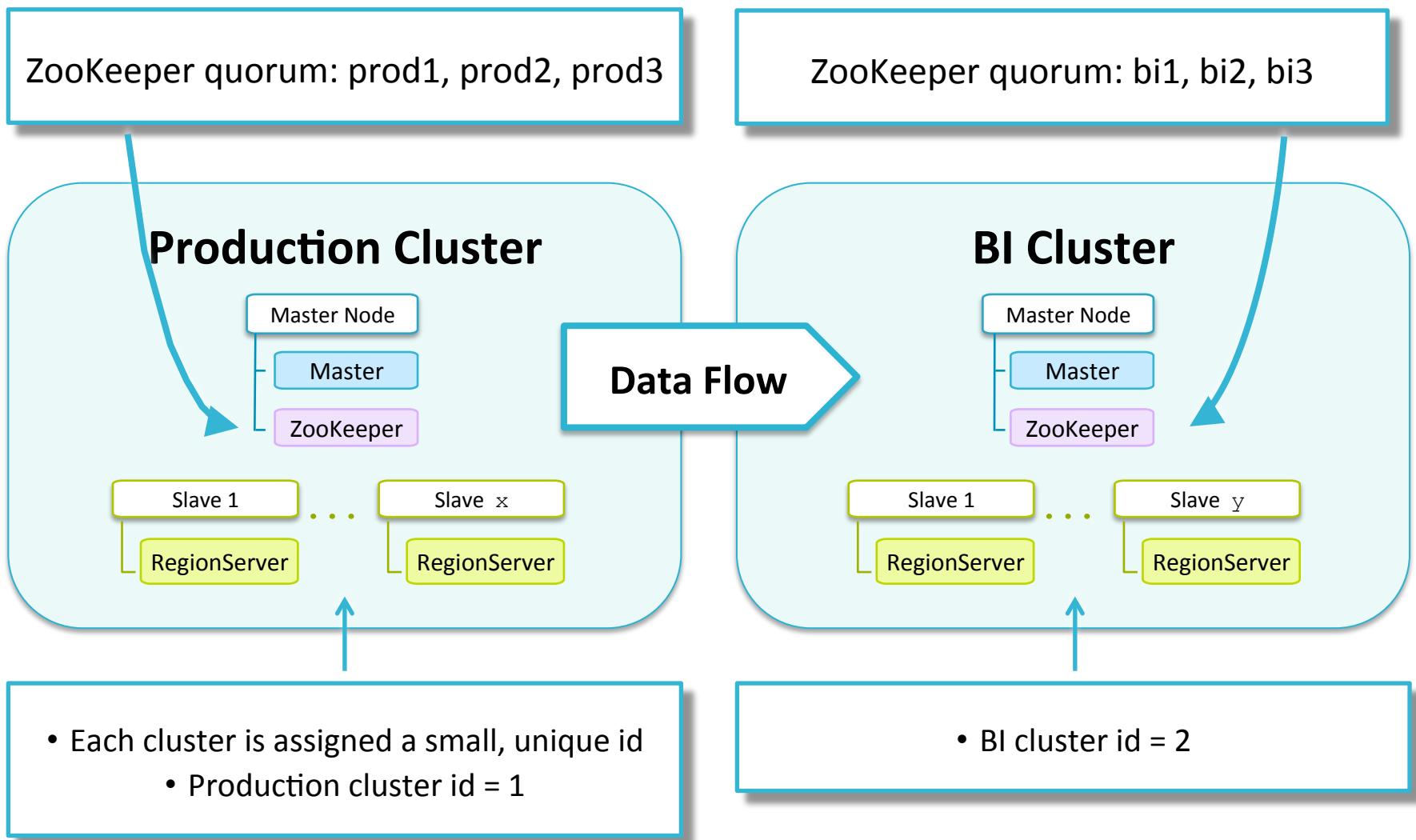
HBase Replication

- **HBase replicates data by shipping the WAL**
 - The copy is asynchronous and eventually consistent
 - Only writes to the WAL will be shipped
 - puts that bypass the WAL will not be replicated
 - Bulk imports will not be replicated (as they disable the WAL)
 - Updates will be delivered at least once and be atomic
- **HBase uses ZooKeeper to maintain state about the replication**
 - The list of clusters to replicate to
 - The list of WAL logs to replicate and the position in the log that has been replicated

HBase Replication Caveats

- **Clusters are usually 1-2 seconds apart**
 - Assuming there is adequate bandwidth
 - If bandwidth is insufficient, the ‘backup’ cluster can start to fall behind
- **Replication will start as soon as the command is run**
- **Column family setting changes will not be replicated**
- **Exercise caution when stopping replication**
 - Data written or modified during the stoppage will not be replicated once replication is restarted
- **If the connection between the clusters is down, replication will automatically resume when the connection is restored**

HBase Master-Slave Replication Diagram



Setting Up Replication (1)

- All column families to replicated must to be configured
 - The same table and column family must exist on the target HBase cluster
 - Not all column families need to be replicated
- When creating a column family:

```
hbase> create 't1',  
          {NAME => 'fam1', REPLICATION_SCOPE => '1'}
```

- REPLICATION_SCOPE is either set to a ‘1’ or ‘0’ to turn replication on or off, respectively
- Setting defaults to 0 (off)

Setting Up Replication (2)

- HBase must be configured to enable replication
 - This setting is enabled in Cloudera Manager
- HBase needs to know which cluster to replicate to

```
hbase> add_peer '2', "bi1,bi2,bi3:2181:/hbase"
```

- The first argument is the peer id which is a small, unique id to identify a cluster
- The second argument is the ZooKeeper quorum in a comma separated format followed by the root znode for HBase
- Peers are not configured in a configuration file
 - Peer information is kept in ZooKeeper

Types Of Replication

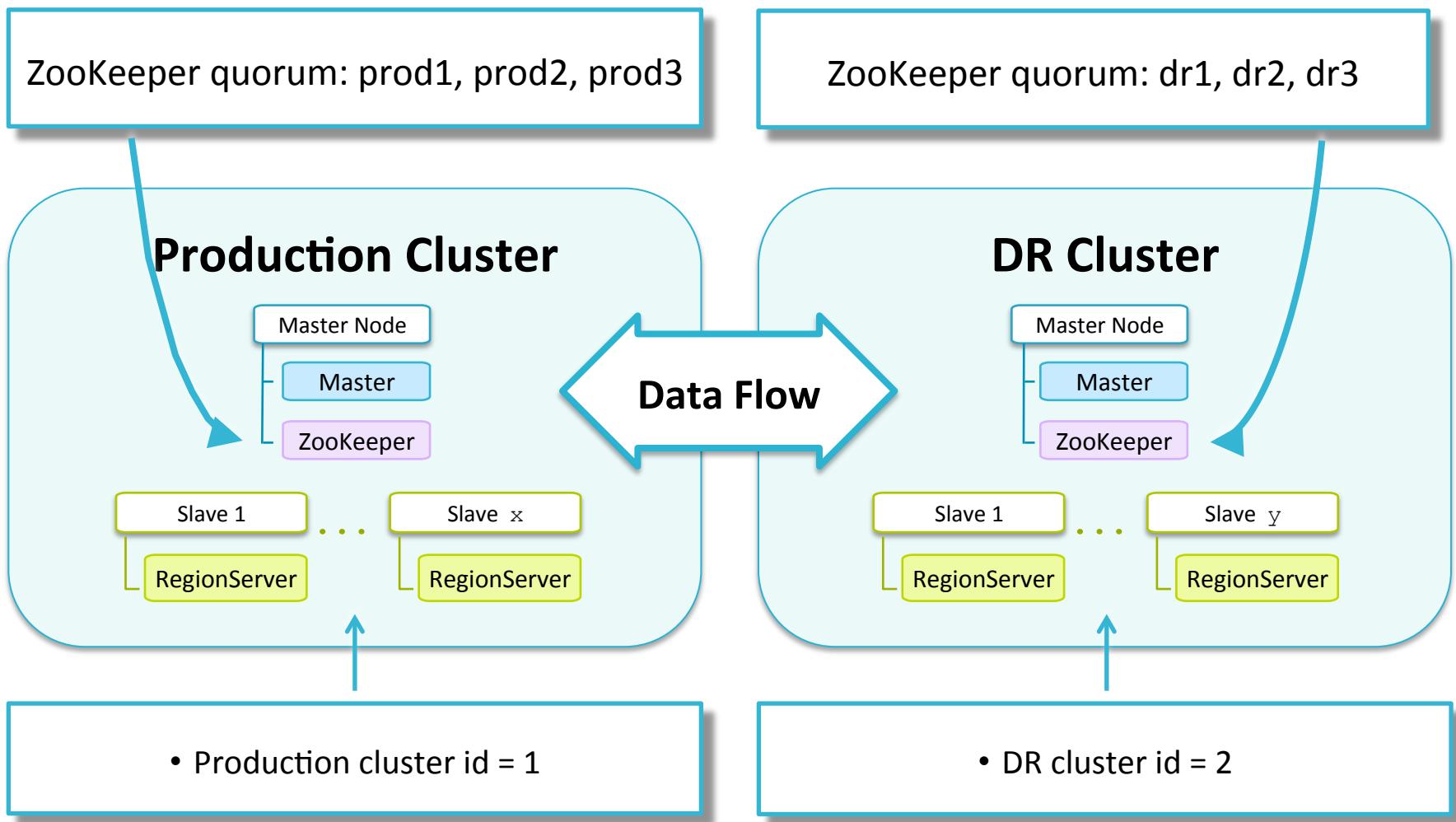
- **There are three types of replication strategies**
 - Master-slave replication is unidirectional from master to slave
 - Master-master replication is bidirectional between clusters
 - Cyclic replication is for clusters with > 2 clusters to replicate
- **To configure master-master:**
 - Turn on replication in both clusters
 - In the Production Cluster, add the DR cluster as a peer:

```
hbase> add_peer '2', "dr1,dr2,dr3:2181:/hbase"
```

- In the DR Cluster, add the Prod cluster as a peer:

```
hbase> add_peer '1', "prod1,prod2,prod3:2181:/hbase"
```

HBase Master-Master Replication Diagram



Verifying Replication (1)

- Check that peers are set up correctly

```
hbase> list_peers
```

- Verify that the peers are pointing to the correct cluster id
- Verify that all peers are listed to replicate to
- Run the VerifyReplication to verify that each row was replicated correctly

```
$ hbase  
org.apache.hadoop.hbase.mapreduce.replication.VerifyReplication  
[--starttime=timestamp1] [--stoptime=timestamp]  
[--families=comma separated list of families]  
<peerId> <tablename>
```

- Will print counters for GOODROWS and BADROWS
- The MapReduce job's log will show the differences

Verifying Replication (2)

- Consider running continuous verification to check that replication is working
 - HBase does not have a built-in way to do this
 - Many companies write their own program to handle this
 - Create a custom program that writes a timestamp to a known table and row (also known as a ‘canary program’)
 - The canary program on the other cluster reads the timestamp and verifies the timestamp is not stale
 - The canary program should have configurable times between writes and time windows for stale checks

Chapter Topics

HBase Replication and Backup

- HBase Replication
- **HBase Backup**
- MapReduce and HBase Clusters
- Hands-On Exercise: Administration
- Conclusion

HBase Backups

- **There are various methods to back up data in HBase**
- **Certain backups methods only save the data**
 - The table, column families, and any settings must recreated manually
- **Other backups represent a point in time (PIT)**
 - Changes during and after the backup are not saved
- **Some backups can happen while the cluster is online**
 - Others require all HBase processes to be stopped

CopyTable

- **CopyTable copies a table within a cluster, or to another cluster**
- **To make a copy of a table on the same cluster, first create the target table with the same column families as the original table, then:**

```
$ hbase org.apache.hadoop.hbase.mapreduce.CopyTable \
--new.name=targettable originaltable
```

Backup and Restore

- **HBase has built-in programs to import and export tables**
 - Uses MapReduce to create a full point-in-time backup of the table
- **To export a table to HDFS**

```
$ hbase org.apache.hadoop.hbase.mapreduce.Export table hdfspath
```

- Only data is backed up
 - Not the metadata such as column families
- **To import a table from HDFS**
 - Create the target HBase table
 - Then run:

```
$ hbase org.apache.hadoop.hbase.mapreduce.Import table hdfspath
```

ImportTSV (1)

- **ImportTSV allows more advanced imports of data**
 - Data must be delimited, newline-terminated text files
- **The column family and column descriptor for each column must be configured with `importtsv.columns`**
- **If the wrong number of columns are given, the job will fail with an error**
 - Check the MapReduce job's error log for details
- **Follow these steps to try out ImportTSV:**
 - Put the input data into HDFS
 - Create the target table, or use `importtsv.bulk.output`
 - Use `HBASE_ROW_KEY` to designate which column will be the row key

```
$ hbase org.apache.hadoop.hbase.mapreduce.ImportTsv \
-Dimporttsv.columns=HBASE_ROW_KEY,fam1:col table hdfspath
```

ImportTSV (2)

- The default delimiter is a tab character and can be configured with `importtsv.separator`

```
$ hbase org.apache.hadoop.hbase.mapreduce.ImportTsv \
-Dimporttsv.columns=HBASE_ROW_KEY,fam1:col \
'-Dimporttsv.separator=' table hdfspath
```

- ImportTSV issues a put per row

- Can be configured to bulk load by creating an HFile as the output
- Use `importtsv.bulk.output` and supply the path in HDFS to write to and the table name

```
$ hbase org.apache.hadoop.hbase.mapreduce.ImportTsv \
-Dimporttsv.columns=HBASE_ROW_KEY,fam1:col \
-Dimporttsv.bulk.output=/path/for/output table hdfspath
```

Bulk Loading

- **LoadIncrementalHFiles** is used to import the HFiles created by **ImportTSV**

```
$ hbase org.apache.hadoop.hbase.mapreduce.LoadIncrementalHFiles \
  /path/to/output table
```

- Bulk loading allows efficient loading of data
 - Especially helpful for time series and sequential data
 - Individual puts are not run
 - Bypasses the WAL, Memstore, and memory issues
 - Hotspotting is avoided by importing all data at once
- Can only be used in batches or loading incremental data
- Note: Bulk loaded data will not be replicated
- Tables remain online while the bulk load is performed

Full Backup

- A full backup can be performed by copying the HBase directory in HDFS
 - Can only be done offline
 - All HBase daemons must be stopped to prevent changes during the copy
 - Should be done with the HDFS distcp command
 - The distcp command uses MapReduce to perform the copy on several nodes

```
$ hadoop distcp /hbase /hbasebackup
```

- Restore the backup by changing the `hbase.rootdir` in `hbase-site.xml` to the backup path
 - The HBase daemons must be restarted for this to take effect

Snapshot

- A snapshot is a collection of metadata required to reconstitute the data near a particular point in time
- Snapshots create metadata-only backups of HBase tables
 - These are references to the files which contain the table data
 - They do not copy any data
 - Snapshots are stored in HDFS
 - Snapshots are not deleted
 - They must be manually deleted when no longer required
- Snapshots can be taken while the cluster is online

Using Snapshots (1)

- To enable snapshots, snapshotting should be enabled in Cloudera Manager
- To create a snapshot:

```
hbase> snapshot 'table', 'snapshotName'
```

- Snapshot names can contain alphanumeric, underscore, or period characters

Using Snapshots (2)

- To create a new table from a snapshot:

```
hbase> clone_snapshot 'snapshotName', 'newtablename'
```

- To restore a snapshot to the table it was taken from:

```
hbase> restore_snapshot 'snapshotName'
```

- The table must be disabled beforehand

- To list existing snapshots:

```
hbase> list_snapshots
```

Exporting Snapshots

- To export a snapshot to another cluster:

```
hbase org.apache.hadoop.hbase.snapshot.ExportSnapshot \
-snapshot 'snapshotName' -copy-to hdfs://binamenode:8082/hbase
```

- The export snapshot operation copies both data and metadata to the target cluster
- Data and metadata are copied directly from HDFS to HDFS without using the RegionServer

Snapshot Caveats

- **Files that make up the snapshot will not be deleted**
 - Verify that there is enough storage to store the entire table at the time of the snapshot
- **Merging regions referenced by a snapshot causes data loss on the snapshot and on cloned tables**
- **Restoring a table with replication enabled for the restored table results in the two clusters being out of sync**
 - The table is not restored on the replica

MapReduce Over Snapshots

- **It is possible to run a MapReduce job over a snapshot from HBase**
 - Useful for running resource-intensive MapReduce jobs that can tolerate potentially-stale data
 - Snapshot can be copied to a different cluster, avoiding load on the main production cluster
 - A new API, TableSnapshotInputFormat, is provided
- **Security implications for a file exported out of the scope of HBase**
 - Bypass of ACLs, visibility labels, and encryption in your HBase cluster
 - Permissions of the underlying filesystem and server govern access to the snapshot

Backup Methods

	Snapshot	distcp	CopyTable	Export
Online	Yes	No	Yes	Yes
Consistency	PIT	Full	PIT	PIT
Metadata	Yes	Yes	No	No
Data	Yes if exporting snapshot	Yes	Yes	Yes

- **Snapshot, CopyTable, and Export methods offer ‘point in time’ consistency, while distcp offers full consistency**
- **To achieve full consistency, distcp requires the cluster to be offline during copying**

Backups and Replication

- **Backups and replication perform different functions**
- **Backup: contains the complete dataset**
 - Allows a full recovery of data
 - Contains the dataset for a specific point in time
- **Replication: virtually real-time duplication of a cluster**
 - User error on one cluster will immediately be replicated to the other!

Chapter Topics

HBase Replication and Backup

- HBase Replication
- HBase Backup
- **MapReduce and HBase Clusters**
- Hands-On Exercise: Administration
- Conclusion

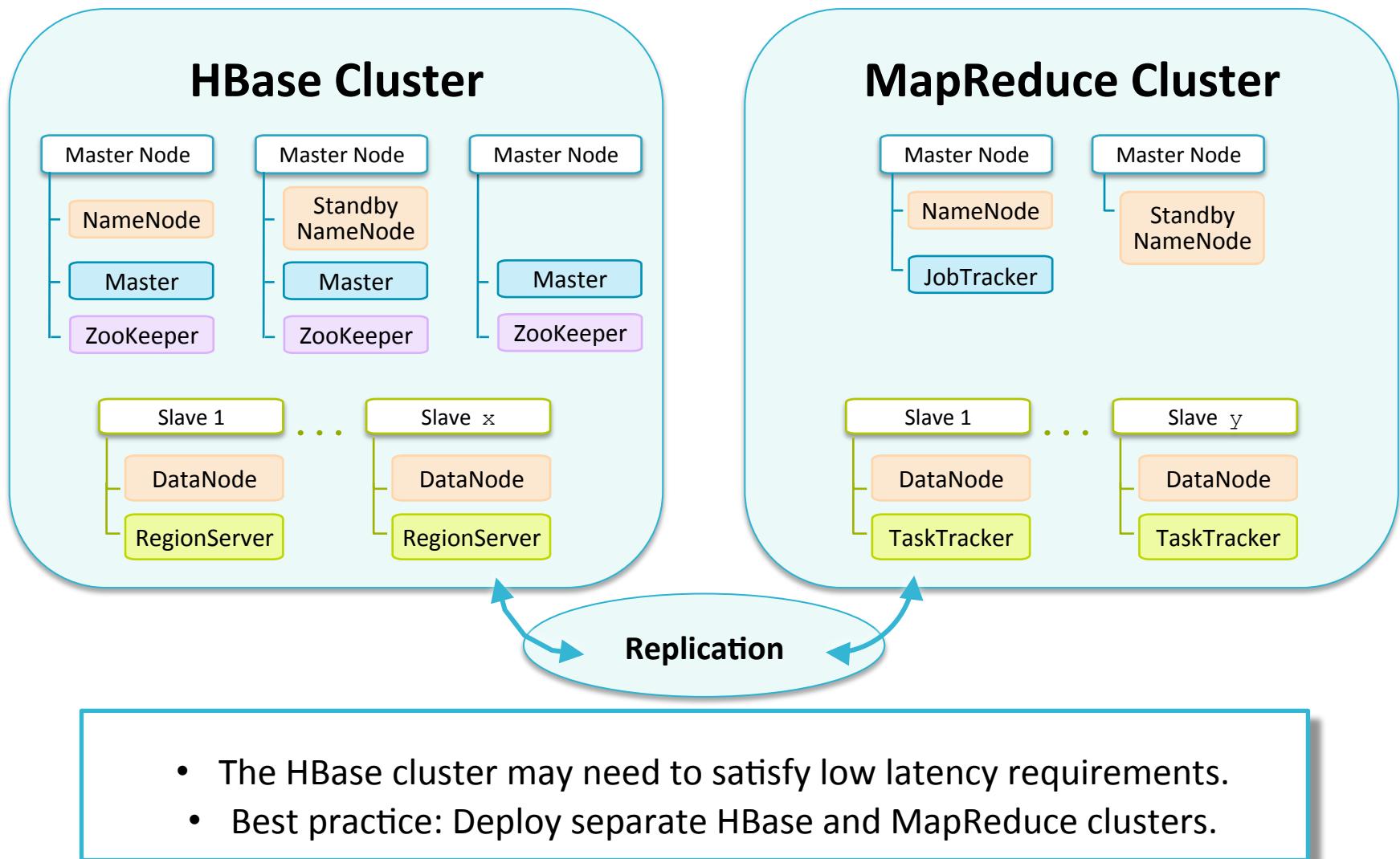
MapReduce Considerations (1)

- **MapReduce adds more daemons to the cluster**
 - MR1: JobTracker for coordinating MR jobs and TaskTrackers for running the tasks
 - MR2: ResourceManager for managing resources on nodes, ApplicationMasters and NodeManagers to run the tasks on slave nodes
- **MapReduce workflows typically need another 20-30% of disk space for temporary data**

MapReduce Considerations (2)

- **Be careful if you are running many MapReduce jobs on the same cluster as your HBase installation**
 - Heavy MapReduce workload can cause problems for HBase
- **Recommendation: significantly decrease the resources allocated to MapReduce tasks on any slave node which is also running as a RegionServer**
- **Monitor nodes to ensure they are not being starved of disk I/O, RAM, or network bandwidth**
- **Consider using two clusters:**
 - One for HBase and MapReduce jobs which use data from HBase
 - One for general MapReduce jobs

MapReduce and Low Latency HBase Cluster Diagram



Request Throttling

- MapReduce will typically run as fast as possible when using HBase as a source
 - Can cause latency issues for other real-time requests
- CDH 5.2 and above introduce ‘Request Throttling’
 - Can control the number of requests or the amount of data returned
 - Configurable by user or by table or namespace
 - Example: Throttle table ‘movies’ to 100 requests/second
 - Example: Throttle user ‘jim’ on table ‘hits’ to 2MB/sec
 - Configured from the HBase shell

```
hbase> set_quota TYPE=>THROTTLE, TABLE=>'movies',  
LIMIT=>'100req/sec'  
  
hbase> set_quota TYPE=>THROTTLE, USER=>'jim', TABLE=>'hits',  
LIMIT=>'2M/sec'
```

Chapter Topics

HBase Replication and Backup

- HBase Replication
- HBase Backup
- MapReduce and HBase Clusters
- **Hands-On Exercise: Administration**
- Conclusion

Hands-On Exercise: Administration

- In this Hands-On Exercise, you will perform various administrative tasks
- Please refer to the Exercise Manual

Chapter Topics

HBase Replication and Backup

- HBase Replication
- HBase Backup
- MapReduce and HBase Clusters
- Hands-On Exercise: Administration
- **Conclusion**

Key Points

- HBase clusters can be automatically replicated
- There are multiple ways to back up an HBase table
- Running MapReduce on an HBase cluster can degrade real-time performance



Using Hive and Impala with HBase

Chapter 14



Course Chapters

- Introduction
- Introduction to Hadoop and HBase
- HBase Tables
- HBase Shell
- HBase Architecture Fundamentals
- HBase Schema Design
- Basic Data Access with the HBase API
- More Advanced HBase API Features
- HBase on the Cluster
- HBase Reads and Writes
- HBase Performance Tuning
- HBase Administration and Cluster Management
- HBase Replication and Backup
- **Using Hive and Impala with HBase**
- Conclusion
- Appendix: Using Python and Thrift to Access HBase Data
- Appendix: OpenTSDB

The HBase Ecosystem

In this chapter you will learn

- **How to access HBase with Hive and Impala**

Chapter Topics

The HBase Ecosystem

- **Using Hive and Impala with HBase**
- Hands-On Exercise: Hive and HBase
- Conclusion

Hive: Motivation

- HBase code is typically written in Java or some other language using Thrift
- Requires:
 - A good programmer
 - Who understands how HBase works
 - Who understands the problem they're trying to solve
 - Who has enough time to write, maintain, and test the code
- What's needed is a higher-level abstraction to access HBase
 - Providing the ability to query the data without needing to know the HBase APIs intimately
 - Hive addresses these needs

Hive: Introduction

- **Hive is an open source Apache project**
 - Provides a very SQL-like language
 - Can be used by people who know SQL
 - Under the covers, generates MapReduce jobs that run on the Hadoop cluster

The Hive and HBase Data Model

- **Hive ‘layers’ table definitions on top of a table in HBase**
 - Hive is not just limited to layering on top of HBase
 - Can be used with standard files stored in HDFS
 - We will be focusing on using Hive and HBase together
- **Tables**
 - Columns in Hive map to column descriptors in HBase
 - Column types are int, float, string, boolean, and so on
- **IMPORTANT! Because Hive generates MapReduce jobs, using Hive to access HBase is a batch processing operation**
 - Hive does *not* provide a real-time SQL-like interface to HBase!

Starting The Hive Shell

- To launch the Hive shell, start a terminal and run:

```
$ hive
```

- You must add some JAR files to work with HBase:

```
hive> add jar /usr/lib/hive/lib/zookeeper.jar;
hive> add jar /usr/lib/hive/lib/hive-hbase-
handler-0.13.1-cdh5.2.0.jar;
hive> add jar /usr/lib/hive/lib/guava-11.0.2.jar;
hive> add jar /usr/lib/hive/lib/hbase-client.jar;
hive> add jar /usr/lib/hive/lib/hbase-common.jar;
hive> add jar /usr/lib/hive/lib/hbase-hadoop-
compat.jar;
hive> add jar /usr/lib/hive/lib/hbase-hadoop2-
compat.jar;
hive> add jar /usr/lib/hive/lib/hbase-protocol.jar;
hive> add jar /usr/lib/hive/lib/hbase-server.jar;
hive> add jar /usr/lib/hive/lib/htrace-core.jar;
```

Hive Basics: Creating External HBase Tables

```
hive> CREATE EXTERNAL TABLE user
  (rowkey string, fname string, lname string)
STORED BY
'org.apache.hadoop.hive.hbase.HBaseStorageHandler'
WITH SERDEPROPERTIES ("hbase.columns.mapping" =
":key,contactinfo:fname,contactinfo:lname")
TBLPROPERTIES
("hbase.table.name" = "user");

hive> DESCRIBE user;
```

Basic SELECT Queries

- Hive supports most familiar SELECT syntax

```
hive> SELECT * FROM user LIMIT 10;

hive> SELECT * FROM user
      WHERE rowkey = "jsmith";

hive> SELECT count(*) FROM user
      WHERE fname like "J%"
      and lname like "S%";
```

Joining Tables

- **Joining datasets is a time consuming operation with the HBase API**
 - We have seen this throughout the course
- **In Hive, it's easy!**

```
SELECT user.fname, user.lname, order.orderid  
      FROM user JOIN order  
    ON (user.rowkey = order.rowkey);
```

Improved Scan Performance

- You can achieve better scan performance by telling HBase to fetch more rows at a time from the server
 - This comes at the cost of requiring a greater amount of memory to be allocated
- To increase the number of rows fetched at a time, set `hbase.client.scanner.cachingNumber` to the desired number of rows
- The default value for `hbase.client.scanner.caching` is 100

Impala

- Like Hive, Impala provides a higher-level abstraction to access data stored in HBase
 - Impala provides a SQL-like language
 - Impala can access data stored in HBase, but it can also access data stored in HDFS
- Impala executes queries very quickly because it is not batch-oriented

Impala on HBase

- **Like Hive, Impala ‘layers’ a table definitions on top of a table in HBase**
 - Impala and Hive share the same metastore database
 - Once the table is created in Hive, Impala can query or insert into it
- **Impala supports most familiar SELECT syntax**
 - Full-table scans are efficient for regular Impala tables in HDFS, but less efficient when using Impala with HBase

Chapter Topics

The HBase Ecosystem

- Using Hive and Impala with HBase
- **Hands-On Exercise: Hive and HBase**
- Conclusion

Hands-On Exercise: Hive and HBase

- In this Hands-On Exercise, you will run Hive queries on the dataset
- Please refer to the Exercise Manual

Chapter Topics

The HBase Ecosystem

- Using Hive with HBase
- Hands-On Exercise: Hive and HBase
- **Conclusion**

Key Points

- Hive and Impala allow users to access HBase data using a SQL-like syntax
- Note that they do not provide a completely real-time SQL interface to HBase



Conclusion

Chapter 15



Conclusion

During this class, you have learned

- **The core technologies of Apache HBase**
- **How HBase and HDFS work together**
- **How to work with the HBase shell and Java API**
- **The HBase storage and cluster architecture**
- **The fundamentals of HBase administration**
- **Advanced features of the HBase API**
- **The importance of schema design in HBase**
- **How to use Hive and Impala with HBase**

Thank You!

- **Thank you for attending the course!**
- **If you have any questions or comments, please contact us via
<http://www.cloudera.com>**



Accessing Data with Python and Thrift

Appendix A

Course Chapters

- Introduction
- Introduction to Hadoop and HBase
- HBase Tables
- HBase Shell
- HBase Architecture Fundamentals
- HBase Schema Design
- Basic Data Access with the HBase API
- More Advanced HBase API Features
- HBase on the Cluster
- HBase Reads and Writes
- HBase Performance Tuning
- HBase Administration and Cluster Management
- HBase Replication and Backup
- Using Hive and Impala with HBase
- Conclusion
- **Appendix: Using Python and Thrift to Access HBase Data**
- Appendix: OpenTSDB

Using Python and Thrift to Access Data

In this chapter you will learn

- **How to access data in HBase using Python and the Thrift interface**

Chapter Topics

Using Python and Thrift to Access Data

- **Thrift Usage**
- Working with Tables
- Getting and Putting Data
- Scanning Data
- Deleting Data
- Counters
- Filters
- Hands-On Exercise: Using the Developer API with Python and Thrift
- Conclusion

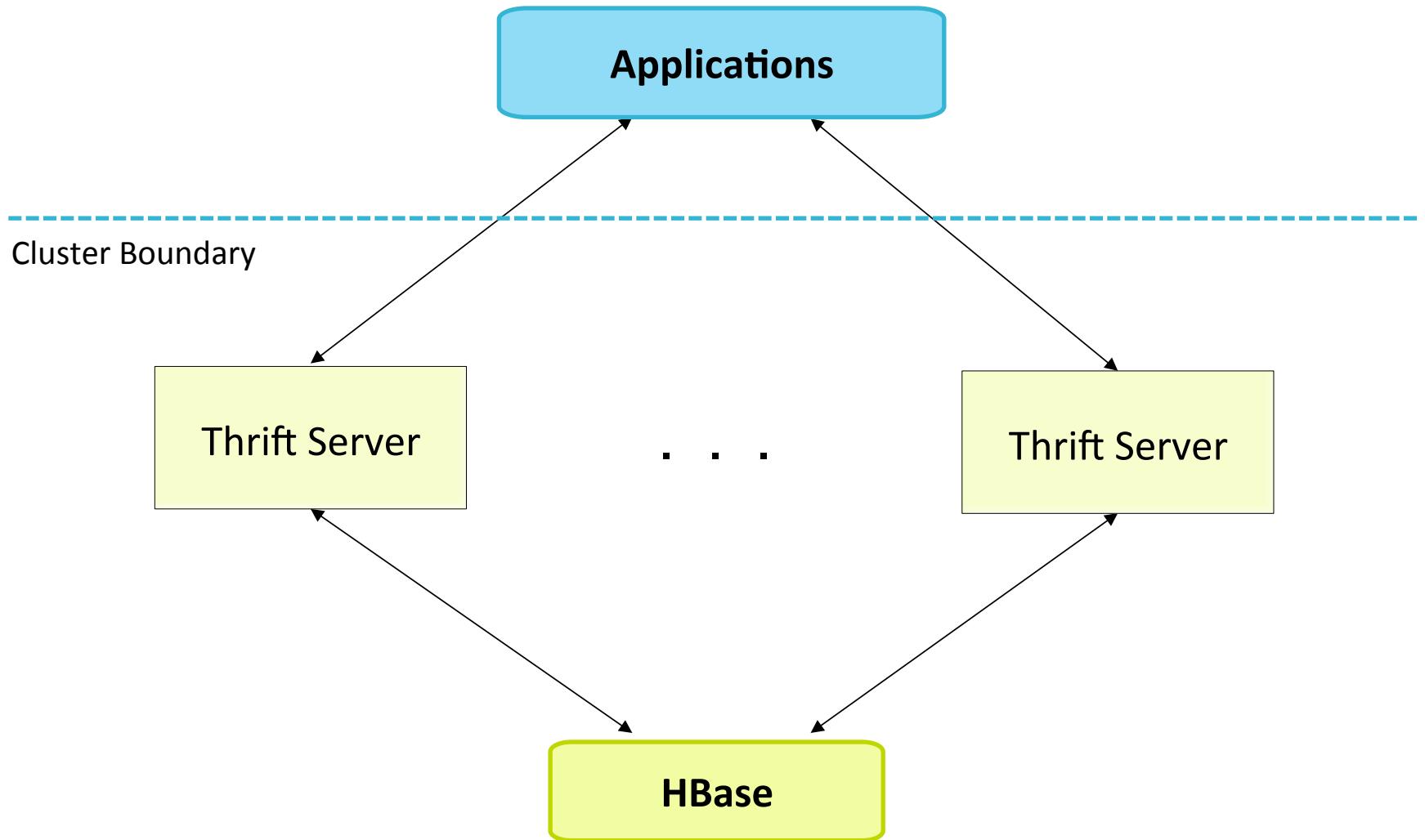
Reprise: Accessing Data in HBase

- **There are several different ways of accessing data in HBase depending on your language and use case**
- **The Java API is the only first class citizen for HBase**
 - The Java API is the preferred method for accessing HBase
- **For non-Java languages there are the Thrift and REST interfaces**
 - The Thrift interface is the preferred method for non-Java access in HBase
 - The REST interface allows data access using HTTP calls
- **The HBase Shell can also be used to access data**
 - This can be used for smaller, ad hoc queries

Apache Thrift and HBase

- **Thrift is a framework for creating cross-language services**
 - Open source Apache project, originally developed at Facebook
 - Supports 14 languages including Java, C++, Python, PHP, Ruby, C#
- **Most common way for non-Java programs to access HBase**
 - Uses a binary protocol and does not need encoding or decoding
 - Provides the most language-friendly way to access HBase

HBase Thrift Diagram



Using Thrift With Python

- Before you can use Python with HBase you will need to first install Thrift
- Next, generate HBase Thrift bindings for Python:

```
thrift -gen py /path/to/hbase/source/hbase-0.98.6-cdh5.2.0/src/  
main/resources/org/apache/hadoop/hbase/thrift/Hbase.thrift
```

- The generated bindings files must be moved to the Python project's directory

Install the Thrift Library

- The Python Thrift library must be installed

```
sudo easy_install thrift==0.9.0
```

- Or copied from the Thrift source

```
cp -r /path/to/thrift/thrift-0.9.0/lib/py/src/* ./thrift/
```

Python Connection Code: Complete Code

```
from thrift.transport import TSocket
from thrift.protocol import TBinaryProtocol
from thrift.transport import TTransport
from hbase import Hbase

# Connect to HBase Thrift server
transport = TTransport.TBufferedTransport(
    TSocket.TSocket(host, port))
protocol = TBinaryProtocol.TBinaryProtocolAccelerated(transport)

# Create and open the client connection
client = Hbase.Client(protocol)
transport.open()

# Do something

transport.close()
```

Python Connection Code: Imports

```
from thrift.transport import TSocket
from thrift.protocol import TBinaryProtocol
from thrift.transport import TTransport
from hbase import Hbase

# Conn
# The Thrift and HBase modules must be imported
transport = TSocket.TSocket(host, port))
protocol = TBinaryProtocol.TBinaryProtocolAccelerated(transport)

# Create and open the client connection
client = Hbase.Client(protocol)
transport.open()

# Do something

transport.close()
```

Python Connection Code: Connection

```
from thrift.transport import TSocket
from thrift.protocol import TBinaryProtocol
from thrift.transport import TTransport
from hbase import Hbase

# Connect to HBase Thrift server
transport = TTransport.TBufferedTransport(
    TSocket.TSocket(host, port))
protocol = TBinaryProtocol.TBinaryProtocolAccelerated(transport)

# Create client
client = Hbase.Client(transport)

# Do some stuff

transport.close()
```

The TTransport object creates the socket connection between the client and the Thrift server.
The TBinaryProtocol object creates the protocol that defines the line protocol for communication.

Python Connection Code: Client Connection

```
from thrift.transport import TSocket
from thrift.protocol import TBinaryProtocol
from t
from h
# Conn
transp
    TS
protocol = TBinaryProtocol.TBinaryProtocolAccelerated(transport)

# Create and open the client connection
client = Hbase.Client(protocol)
transport.open()

# Do something

transport.close()
```

The Client object takes the Protocol object as a parameter. It is used to communicate with the Thrift server for HBase. The transport is opened so that the socket is opened.

Python Connection Code: Using the Connection

```
from thrift.transport import TSocket
from thrift.protocol import TBinaryProtocol
from thrift.transport import TTransport
from hbase import Hbase

# Connect to HBase Thrift server
transport = TSocket.TSocket('localhost', 9090)
protocol = TBinaryProtocol.TBinaryProtocol(transport)
client = Hbase.Client(protocol)

# Create a connection
client.open()

# Do something

transport.close()
```

All calls to HBase are done using the Client object. The setup and initialization work is done, and the Client object can now be used now. Once all HBase interaction is complete, the Transport object must be closed.

Chapter Topics

Using Python and Thrift to Access Data

- Thrift Usage
- **Working with Tables**
- Getting and Putting Data
- Scanning Data
- Deleting Data
- Counters
- Filters
- Hands-On Exercise: Using the Developer API with Python and Thrift
- Conclusion

Working with Tables

- List all HBase tables

```
tables = client.getTableNames()  
  
print "Tables:"  
for t in tables:  
    print t
```

Create and Delete Tables

- Create an HBase table

```
client.createTable('pytest_table',  
                   [Hbase.ColumnDescriptor('colFam1')])
```

- Delete the table

```
client.disableTable('pytest_table');  
client.deleteTable('pytest_table');
```

Chapter Topics

Using Python and Thrift to Access Data

- Thrift Usage
- Working with Tables
- **Getting and Putting Data**
- Scanning Data
- Deleting Data
- Counters
- Filters
- Hands-On Exercise: Using the Developer API with Python and Thrift
- Conclusion

Reprise: Getting Data

- **Data can be accessed in the HBase Shell, via the Java API, through scripting, or using alternate interfaces**
 - Java and alternate interfaces are discussed later in the chapter
- **Get**
 - Used to retrieve a single row
 - Must know the exact row key to retrieve

Getting Rows with Python

- Get row for specific rowkey

```
row = client.getRows("movie", [ "45" ])
```

- Get rows for multiple rowkeys

```
rowKeys = [ "45", "67", "78", "190", "2001" ]
rows = client.getRows('movie', rowKeys)
```

Getting Cell Versions with Python

- Get most recent version of a cell:

```
columnVersions = client.getVer("movie", "rowkey1",
    "desc:title", 1)
```

- Request multiple versions of a cell:

```
columnVersions = client.getVer("movie", "rowkey1",
    "desc:title", 3)

for column in columnVersions:
    print "The value at:" + str(column.timestamp) +
        " was:" + column.value
```

Reprise: Adding and Updating Data

- **Recall: HBase does not distinguish an insert from an update**
- **Put is used to both insert new rows and update existing rows**
 - An insert occurs when performing a Put on a row key that does not yet exist
 - An update of a row occurs when a Put is performed on an existing row
- **Updates can occur on specific column descriptors, leaving the row's other columns unchanged**

Python Puts and Batch Puts

- Put row in HBase over Thrift

```
mutations = [  
    Hbase.Mutation(column='desc:title',  
    value='Singing in the Rain')]  
client.mutateRow('movie', 'rowkey1', mutations)
```

- Batching multiple row puts

```
mutationsbatch = [  
    Hbase.BatchMutation(row="rowkey1",mutations=mutations1),  
    Hbase.BatchMutation(row="rowkey2",mutations=mutations2)]  
]  
client.mutateRows('movie', mutationsbatch)
```

Chapter Topics

Using Python and Thrift to Access Data

- Thrift Usage
- Working with Tables
- Getting and Putting Data
- **Scanning Data**
- Deleting Data
- Counters
- Filters
- Hands-On Exercise: Using the Developer API with Python and Thrift
- Conclusion

Reprise: Scans

- **The HBase API supports table scans**
- **Recall that a Scan is useful when the exact row key is not known, or when a group of rows needs to be accessed**
- **Scans can be bounded by a start and stop row key**
 - The start row key is included in the results
 - The stop row is not included in the results

Python Scan Code: Complete Code

```
scanInfo = Hbase.TScan(startRow="1", stopRow="rowkey_5")
scannerId = client.scannerOpenWithScan("pyMovies", scanInfo)

row = client.scannerGet(scannerId)

while row:
    print row
rows= client.scannerGet(scannerId)

client.scannerClose(scannerId)
```

Python Scan Code: Open Scanner

```
scanInfo = Hbase.TScan(startRow="1", stopRow="rowkey_5")
scannerId = client.scannerOpenWithScan("pyMovies", scanInfo)

row =
while
    prin
rows= client.scannerGet(scannerId)

client.scannerClose(scannerId)
```

Call scannerOpen to create a Scan object on the Thrift server. This returns a scanner id that uniquely identifies the scanner on the server.

Python Scan Code: Get the List

```
scanInfo = Hbase.TScan(startRow="1", stopRow="rowkey_5")
scannerId = client.scannerOpenWithScan("pyMovies", scanInfo)

row = client.scannerGet(scannerId)

while
    prin
rows=
```

The scannerGet method needs to be called with the unique id. This returns a row of results.

```
client.scannerClose(scannerId)
```

Python Scan Code: Iterating Through

```
scanInfo = Hbase.TScan(startRow="1", stopRow="rowkey_5")
scannerId = client.scannerOpenWithScan("pyMovies", scanInfo)

row = client.scannerGet(scannerId)

while row:
    print row
rows= client.scannerGet(scannerId)

client
```

The `while` loop continues as long as the scanner returns a new row with another call to `scannerGet`.

Python Scan Code: Closing the Scanner

```
scanInfo = Hbase.TScan(startRow="1", stopRow="rowkey_5")
scannerId = client.scannerOpenWithScan("pyMovies", scanInfo)

row = client.scannerGet(scannerId)

while row:
    print row
rows= client.scannerGet(scannerId)

client.scannerClose(scannerId)
```

The `scannerClose` method call is very important. This closes the Scan object on the Thrift server. Not calling this method can leak Scan objects on the server.

Scanner Caching

- Scan results can be retrieved in batches to improve performance
 - Performance will improve but memory usage will increase

```
rowsArray = client.scannerGetList(scannerId,10)

index=0

while rowsArray:

    index+=1

    print "\n%d. Number of results: [%d]" % (index,len(rowsArray))

    for item in rowsArray:

        print "Result: [%s]" % item

    rowsArray = client.scannerGetList(scannerId, 10)

client.scannerClose(scannerId)
```

Chapter Topics

Using Python and Thrift to Access Data

- Thrift Usage
- Working with Tables
- Getting and Putting Data
- Scanning Data
- **Deleting Data**
- Counters
- Filters
- Hands-On Exercise: Using the Developer API with Python and Thrift
- Conclusion

Reprise: Deleting Data

- **Rows can be deleted through HBase Shell, via the Java API, or using Thrift**
- **Recall that HBase marks rows for deletion, and the actual deletion occurs at a later time**
- **Multiple deletes can be performed by batching them together in a list**

Python Deletes

- Delete an entire row from HBase over Thrift

```
client.deleteAllRow("movie", "rowkey1")
```

- The best way to see all available Thrift methods is to open the `Hbase.thrift` file
 - It contains the listing and descriptions of all methods, structures, arguments, and return types

Chapter Topics

Using Python and Thrift to Access Data

- Thrift Usage
- Working with Tables
- Getting and Putting Data
- Scanning Data
- Deleting Data
- **Counters**
- Filters
- Hands-On Exercise: Using the Developer API with Python and Thrift
- Conclusion

Counters

- HBase can atomically increment counters
 - All calls return the value as a 64-bit integer or long

```
client.atomicIncrement('movie', 'rowKey1',
                      'metrics:ticketsSold', amount);
```

Chapter Topics

Using Python and Thrift to Access Data

- Thrift Usage
- Working with Tables
- Getting and Putting Data
- Scanning Data
- Deleting Data
- Counters
- **Filters**
- Hands-On Exercise: Using the Developer API with Python and Thrift
- Conclusion

Reprise: Filters

- **Not all queries can be performed with just a row key**
 - Using scans passes back all rows to the client
 - Client must perform all logic once the data is received to determine which rows are the ones desired
- **Scans can be augmented with Filters**
 - Filters allow logic to be run on RegionServers before the data is returned
 - RegionServers run the logic on the rows and only send what passes the logic
 - Causes less data to be sent over the wire
- **Scans with Filters can be used in the HBase Shell, via the Java API, or using Thrift**

Reprise: Using Filters

- **HBase contains many built-in filters**
 - Allows you to use filters without having to write a new one
 - Filters can be combined
 - In addition, you can create custom filters

Python Scan with Filter: Complete Code

```
filter = 'SingleColumnValueFilter (\'COLUMN_FAMILY\',  
    \'COLUMN_DESCRIPTOR\', =, \'binary:VALUE\')'  
  
scan = Hbase.TScan(startRow="startrow",  
    stopRow="stoprow", filterString=filter)  
scannerId = client.scannerOpenWithScan("tablename",  
    scan)  
  
rowList = client.scannerGetList(scannerId, numRows)
```

Python Scan with Filter: Filter String

```
filter = 'SingleColumnValueFilter (\'COLUMN_FAMILY\',\'  
\'COLUMN_DESCRIPTOR\', =, \'binary:VALUE\')'
```

scan = A string is created that gives the arguments to the filter and which filter to use. The string starts with the name of the filter. Following the Java class parameters, the next arguments are the column family and column descriptor. The CompareOp follows but uses the actual signs instead of words. The string ends with the comparator. The comparator's names are slightly different and the BinaryComparator is only "binary", followed by a colon, then the value to match.

Python Scan with Filter: Adding Filter String

```
filter = 'SingleColumnValueFilter (\'COLUMN_FAMILY\',  
    \'COLUMN_DESCRIPTOR\', =, \'binary:VALUE\')'  
  
scan = Hbase.TScan(startRow="startrow",  
    stopRow="stoprow", filterString=filter)  
scanner = scanner.withScan(scan)  
rowList = scanner.getScanner().next()  
rowList[0].get("row")",  
    rowList[0].get("value"))  
scannedRows = scannedRows + 1  
print "Scanned %d rows" % scannedRows
```

The filter string is passed to Thrift in the TScan object using the filterString key in the parameters. The scan is constrained to the start and stop rows.

Python Filter Lists

- Several filters can be grouped together and nested using parenthesis
 - Similar to grouping and nesting in a SQL where clause

```
filters = 'SingleColumnValueFilter (\'FAMILY\'\n    \'\COLUMN1\', =, \'binary:value\', true, true) AND\nSingleColumnValueFilter (\'FAMILY\', \'\COLUMN2\',\n    =, \'binary:value2\', true, true)'
```

- Evaluations can use AND and OR

Chapter Topics

Using Python and Thrift to Access Data

- Thrift Usage
- Working with Tables
- Getting and Putting Data
- Scanning Data
- Deleting Data
- Counters
- Filters
- **Hands-On Exercise: Using the Developer API with Python and Thrift**
- Conclusion

Hands-On Exercise: Using the Developer API with Python and Thrift

- In this Hands-On Exercise, you will use Python and Thrift to access data in HBase tables
- Please refer to the Exercise Manual

Chapter Topics

Using Python and Thrift to Access Data

- Thrift Usage
- Working with Tables
- Getting and Putting Data
- Scanning Data
- Deleting Data
- Counters
- Filters
- Hands-On Exercise: Using the Developer API with Python and Thrift
- **Conclusion**

Key Points

- HBase can be accessed from Python using the Thrift interface
- Rows can be accessed and deleted using the row key



OpenTSDB

Appendix B



Course Chapters

- Introduction
- Introduction to Hadoop and HBase
- HBase Tables
- HBase Shell
- HBase Architecture Fundamentals
- HBase Schema Design
- Basic Data Access with the HBase API
- More Advanced HBase API Features
- HBase on the Cluster
- HBase Reads and Writes
- HBase Performance Tuning
- HBase Administration and Cluster Management
- HBase Replication and Backup
- Using Hive and Impala with HBase
- Conclusion
- Appendix: Using Python and Thrift to Access HBase Data
- **Appendix: OpenTSDB**

The HBase Ecosystem

In this appendix you will learn

- **The key features of OpenTSDB**

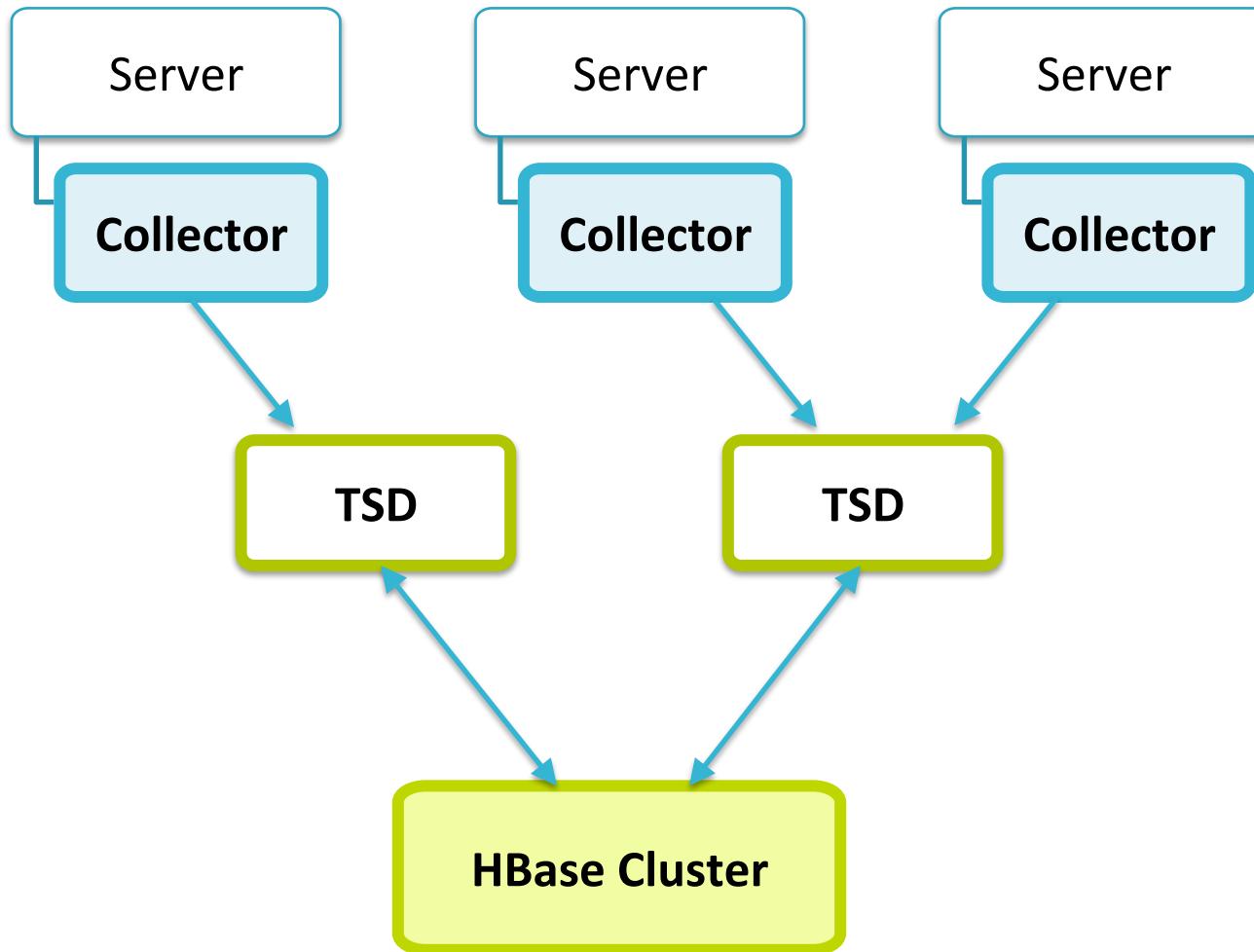
OpenTSDB

- A scalable, distributed Time Series Database
- Allows various metrics for computer systems to be collected, analyzed, and graphed
- Uses HBase for storage and queries
 - Makes use of HBase's row keys to allow for quick retrieval of data
- See <http://www.opentsdb.net> for more information

OpenTSDB Use Cases

- **Metrics collection**
 - Collect metrics from thousands of hosts and applications
 - StumbleUpon collects over one billion data points per day
 - Box and Tumblr collect tens of billions per day
- **Check SLA times**
- **Correlate outages to events in the cluster**
- **Obtain real-time statistics about infrastructure and services**

OpenTSDB Architecture



Compiling OpenTSDB

- Clone from GitHub:

```
git clone git://github.com/OpenTSDB/opentsdb.git
```

- Build:

```
cd opentsdb  
./build.sh
```

- OpenTSDB requires these libraries: **JDK 1.6, asynchbase, Guava, logback, Netty, SLF4J, suasync, ZooKeeper**

Starting OpenTSDB

- Create tables in HBase:

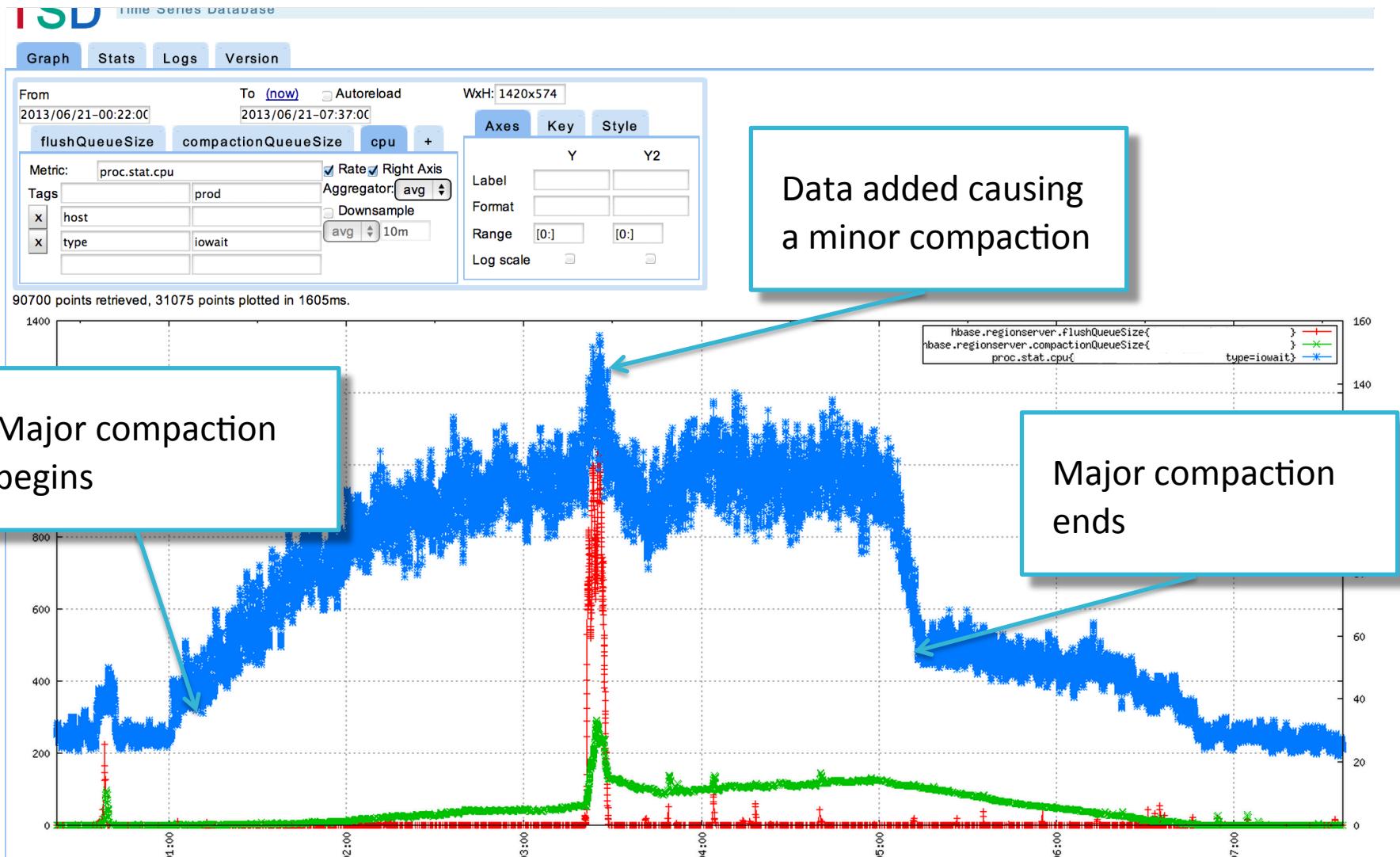
```
env COMPRESSION=lzo HBASE_HOME=path/to/hbase-0.98.6  
./src/create_table.sh
```

- Start TSD (Time Series Daemon):

```
./build/tsdb tsd --port=4242  
--staticroot=build/staticroot --cachedir=/tmp/tsdtmp  
--zkquorum=zkhost1,zkhost2,zkhost3
```

- The TSD's web interface will start on localhost port 4242
- `cachedir` should be a `tmpfs` for best performance
- `zkquorum` is a comma separated list of the ZooKeeper quorum hosts

OpenTSDB Web Interface



OpenTSDB Data Types

- OpenTSDB stores several types of data: metric, timestamp, value, and tags
- The **metric** field is a string that describes the piece of data being captured
 - The string is user-defined
 - e.g., mysql.bytes_received or proc.loadavg.1m
- The **timestamp** field is a value in milliseconds since the Unix epoch
- The **value** field is the value of the metric at the timestamp
- The **tags** field contains arbitrary, informative strings about the data
 - A tag string might be a hostname or a cluster name
 - You can use the tag value to distinguish between the data coming from two services on the same host

OpenTSDB Metrics

- Metrics must be registered before using them:

```
./tsdb mkmetric mysql.bytes_received mysql.bytes_sent
```

- Metrics need to be registered because they are used as a primary key
 - The metrics are not stored as the string, but are stored using the primary key
- New tags do not need to be registered beforehand
 - Tag names and values are not stored as their strings either

Data Collection Script: Example

```
#!/bin/bash
set -e
while true; do
    mysql -u USERNAME -pPASSWORD --batch -N \
--execute "SHOW STATUS LIKE 'bytes%'" \
| awk -F"\t" -v now=`date +%s` -v host=`hostname` \
'{ print "put mysql." tolower($1) " " now " " $2 " " \
host=" host }'
sleep 15
done | nc -w 30 host.name.of.tsdb PORT
```

This script runs in an infinite loop with a 15 second sleep. It runs a status command in MySQL and pipes that to awk for formatting. The nc command passes the data to OpenTSDB

OpenTSDB Row Key and Schema

- Time series data presents a problem for HBase
- OpenTSDB uses a promoted key to avoid RegionServer hotspotting
- When metrics are registered they are given a 3-byte value
 - The 3-byte value is promoted ahead of the timestamp
- Timestamps are rounded down by OpenTSDB to the nearest hour
 - All metrics for the same hour and tags are stored in the same row
 - Each value is stored as a different column
 - The column qualifier is the timestamp remainder after subtracting the hour

OpenTSDB Row Key:

```
<metricid><roundedtimestamp><tagnameid><tagvalueid>
```