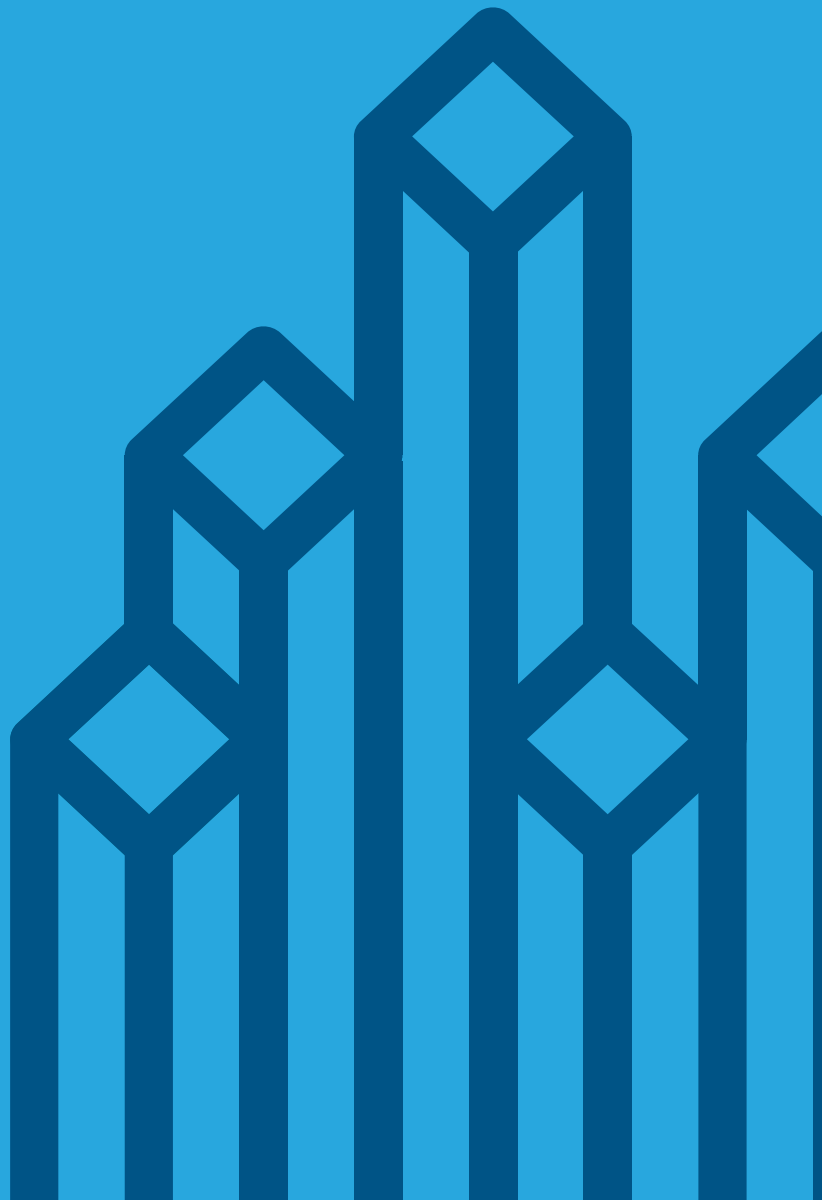
cloudera®

Cloudera Training for Apache HBase



201505

cloudera®

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)

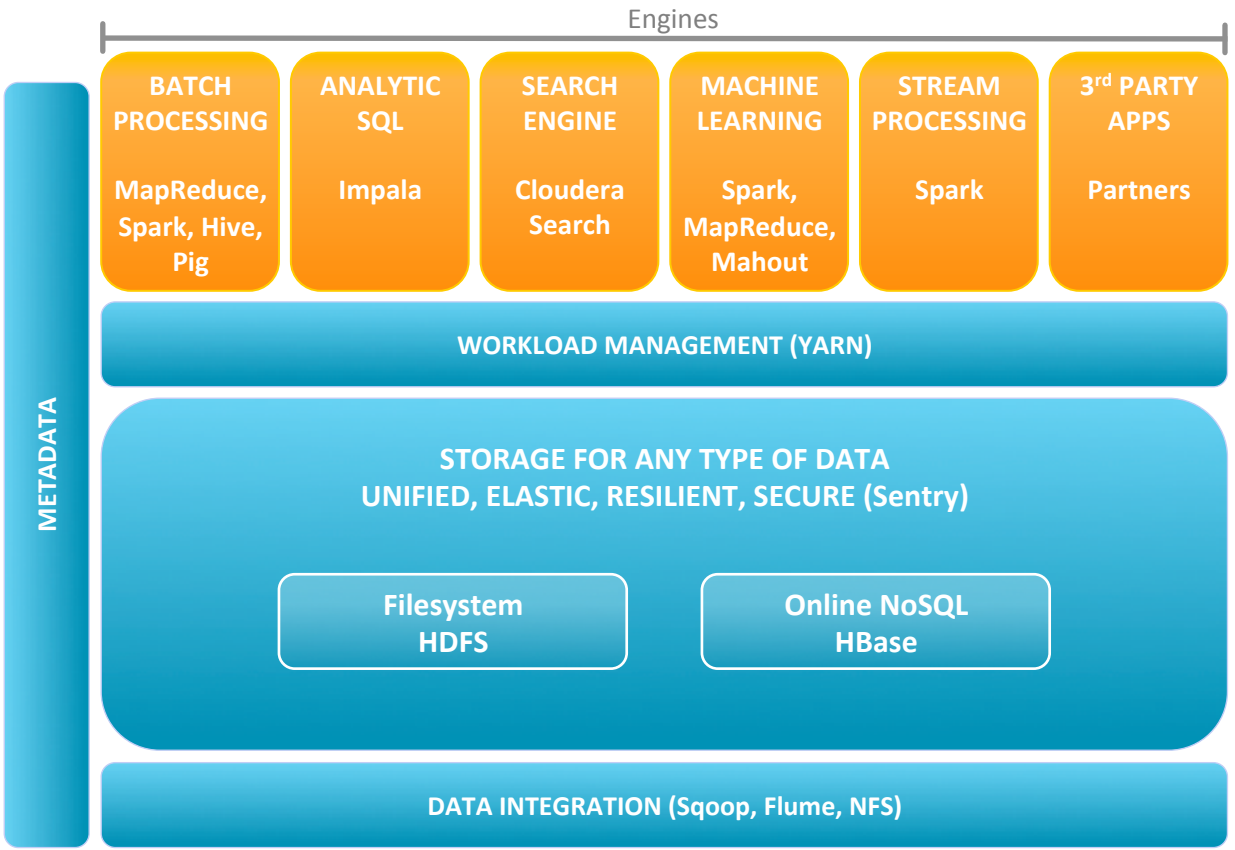
- **The leader in Apache Hadoop-based software and services**
- **Founded by leading experts on Hadoop from Facebook, Yahoo, Google, and Oracle**
- **Provides support, consulting, training, and certification for Hadoop users**
- **Staff includes committers to virtually all Hadoop projects**
- **Many 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**

CDH

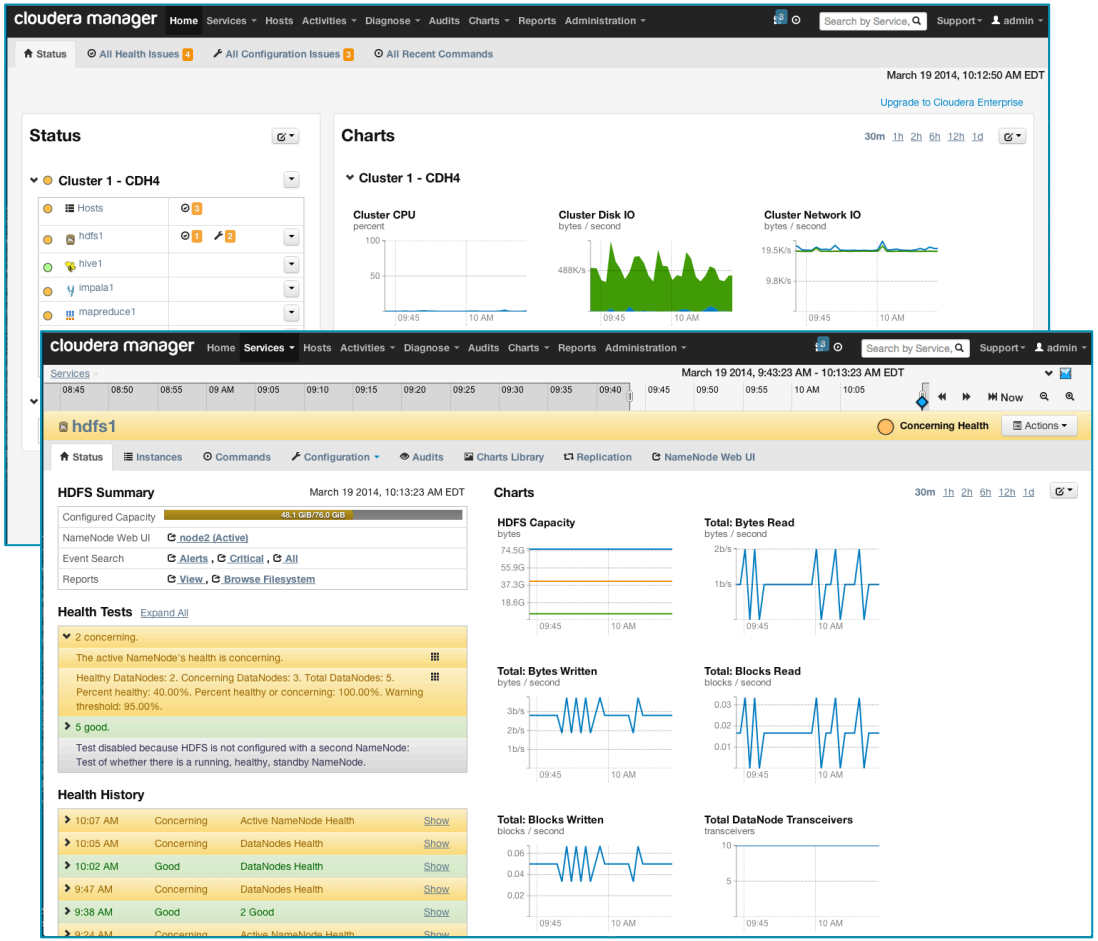
- 100% open source, enterprise-ready distribution of Hadoop and related projects
- The most complete, tested, and widely-deployed distribution of Hadoop
- Integrates all key Hadoop ecosystem projects



- Available as RPMs and Ubuntu/Debian/SuSE packages, or as a tarball

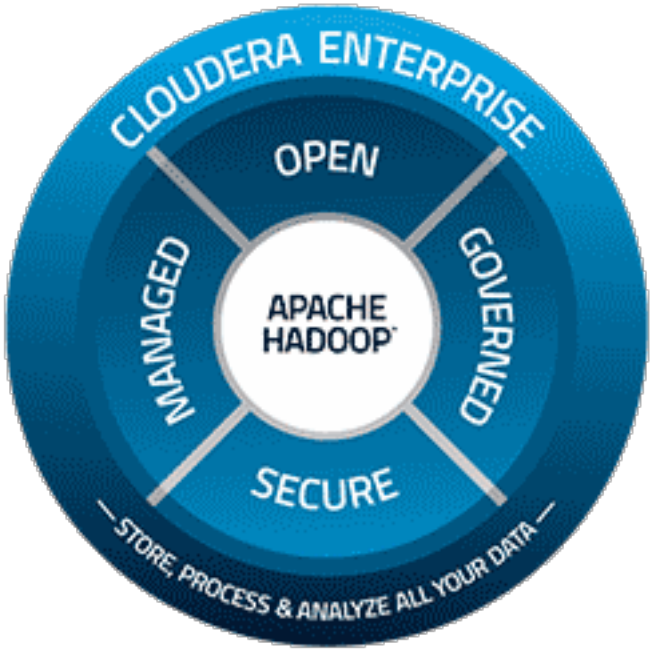
Cloudera Express

- **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?

cloudera®

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

Simple Entities

- Geolocation data
- Search index building

Characteristics

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

Graph Data

- Sessionization
  - Financial transactions
  - Click streams
  - Network traffic

Characteristics

- Batch/realtime, random reads/writes

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

cloudera®

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	**lname**	**image**
jdupont	Jean	Dupont	
jsmith	John	Smith	<smith.jpg>
mrossi	Mario	Rossi	<mario.jpg>

Column Families

Column Names

Rows

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	lname
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
jdupont	contactinfo:lname	1273878447049	Dupont
jsmith	contactinfo:fname	1273516197868	John
jsmith	contactinfo:lname	1273871824184	Smith
mrossi	contactinfo:fname	1273616297446	Mario
mrossi	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**

cloudera®

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

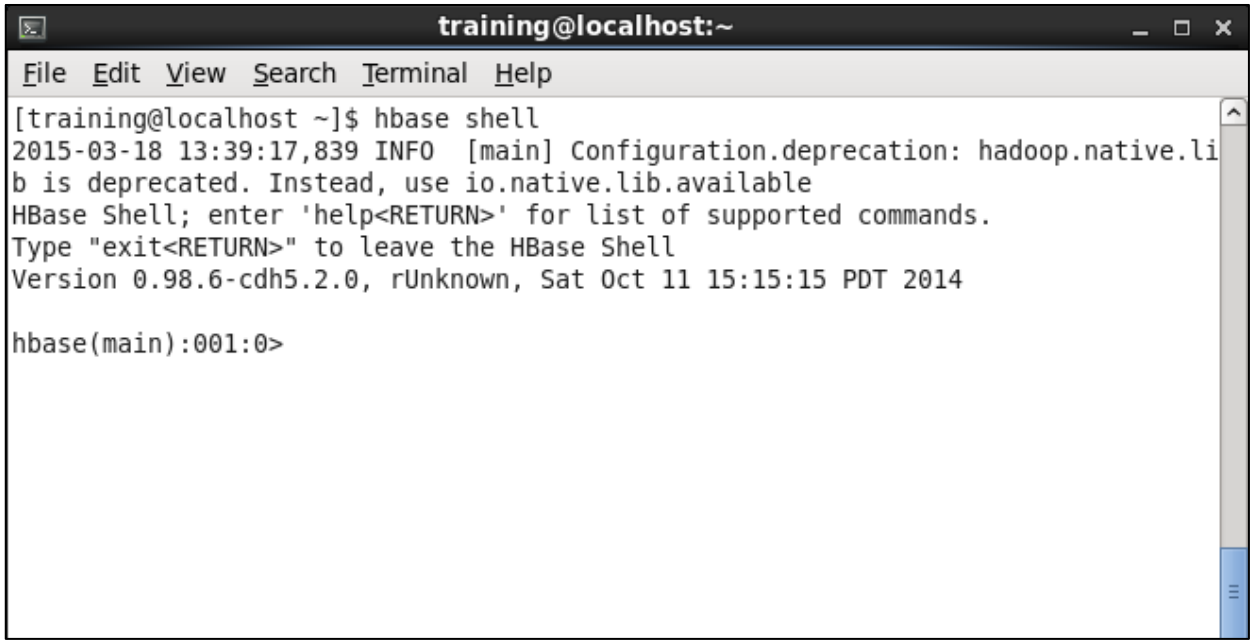
- **Creating Tables with the HBase Shell**
- Working with Tables
- Hands-On Exercise: Using the HBase Shell
- Working with Table Data
- Hands-On Exercise: Data Access in the HBase Shell
- Conclusion

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
```



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**

cloudera®

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

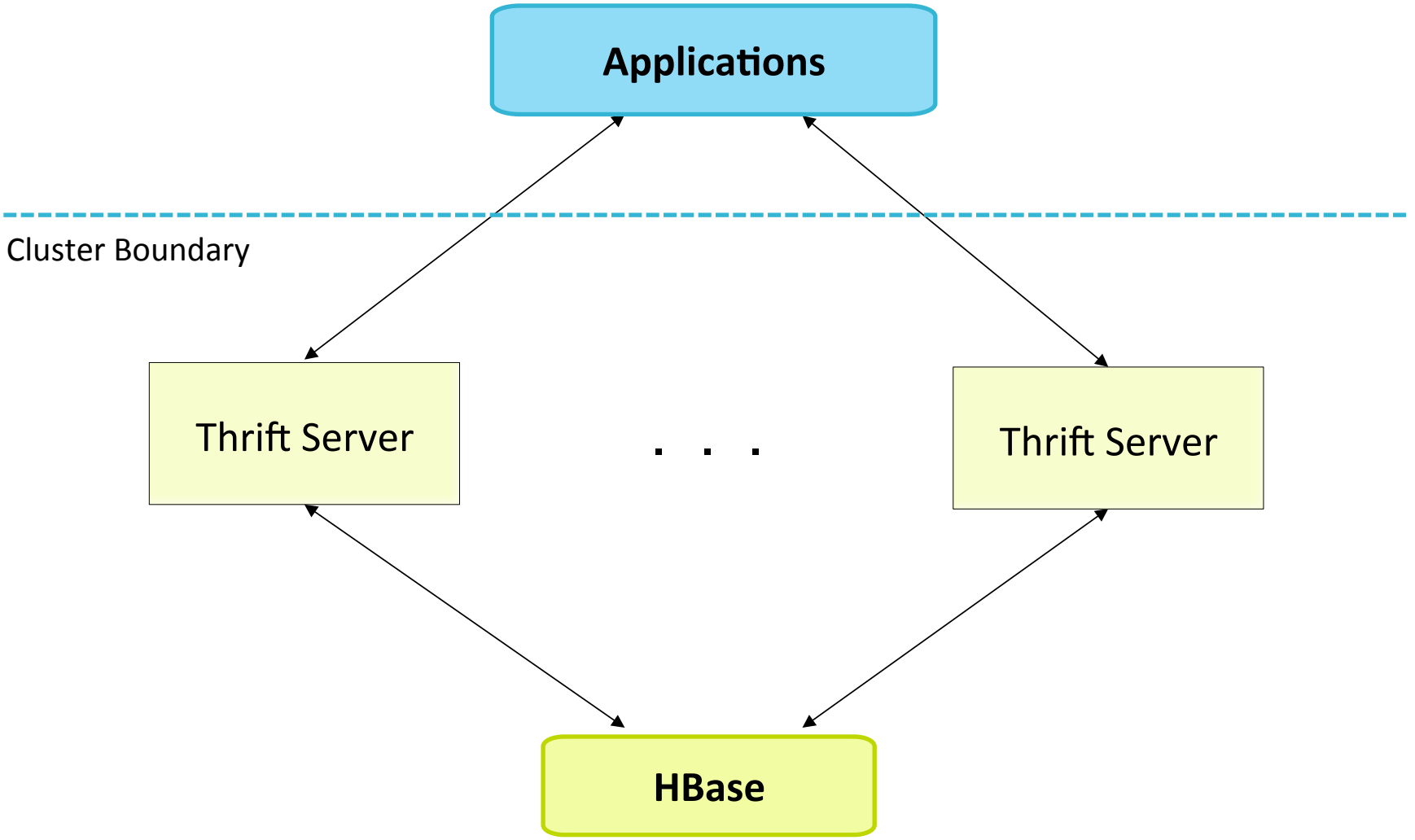
Reprise: Accessing Data in HBase

- **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
transp
    TSocket.TSocket(host, port))
protocol = TBinaryProtocol.TBinaryProtocolAccelerated(transport)

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

# Do something

transport.close()
```

The Thrift and HBase modules must be imported

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)

# Crea
client
transp

# Do s

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
transp
    TS
protoc                                                    ort)

# Crea
client
transport.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=

client.scannerClose(scannerId)
```

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

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 =
   stc
scanne                                                    ",
   sca

rowLis                                                    )
```

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)
scanne                                          ",
   sca

rowLis                                          )
```

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\'
    \'COLUMN1\', =, \'binary:value\', true, true) AND
  SingleColumnValueFilter (\'FAMILY\', \'COLUMN2\',
    =, \'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**

cloudera®

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**

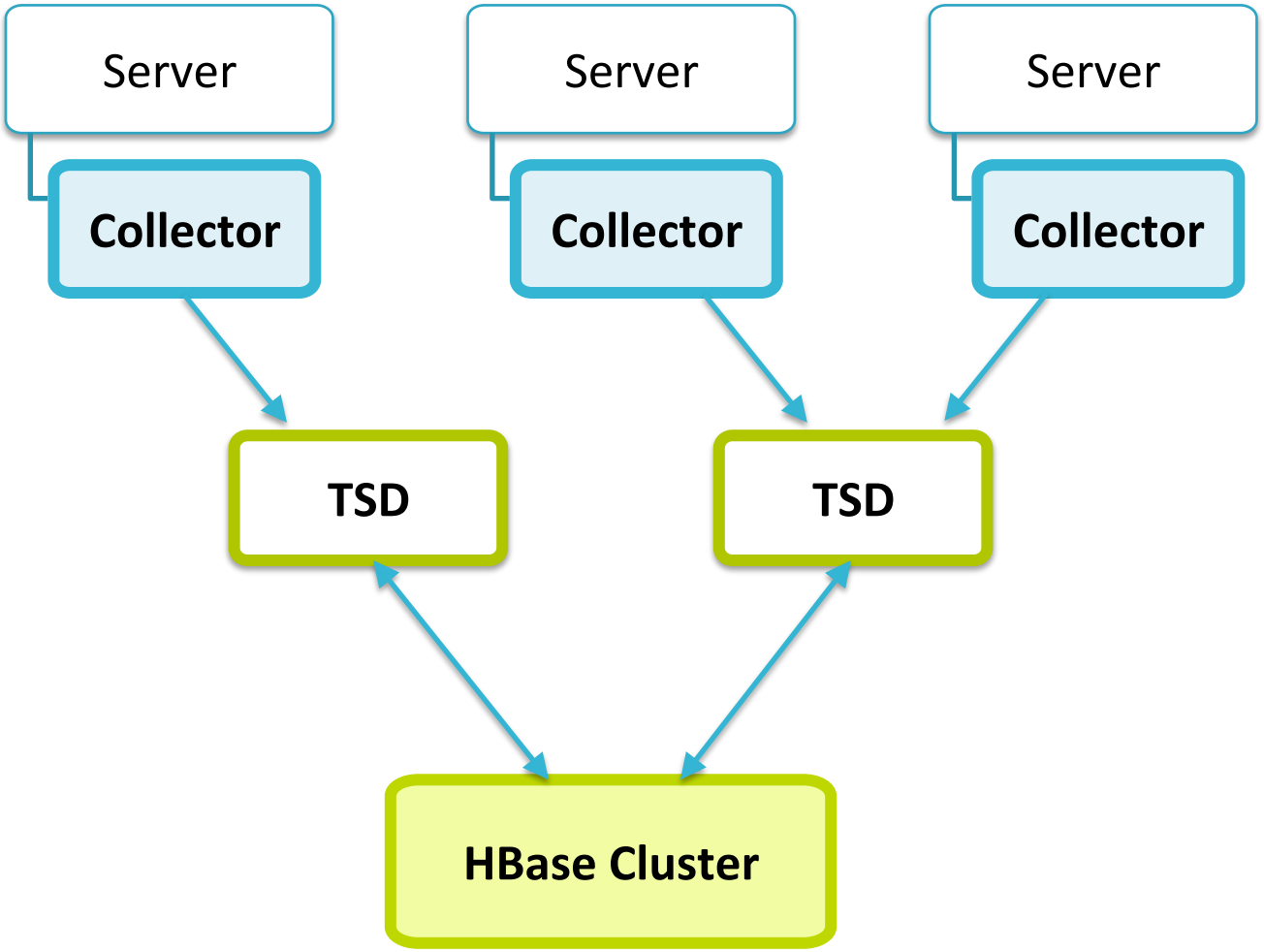
OpenTSDB

- **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:**

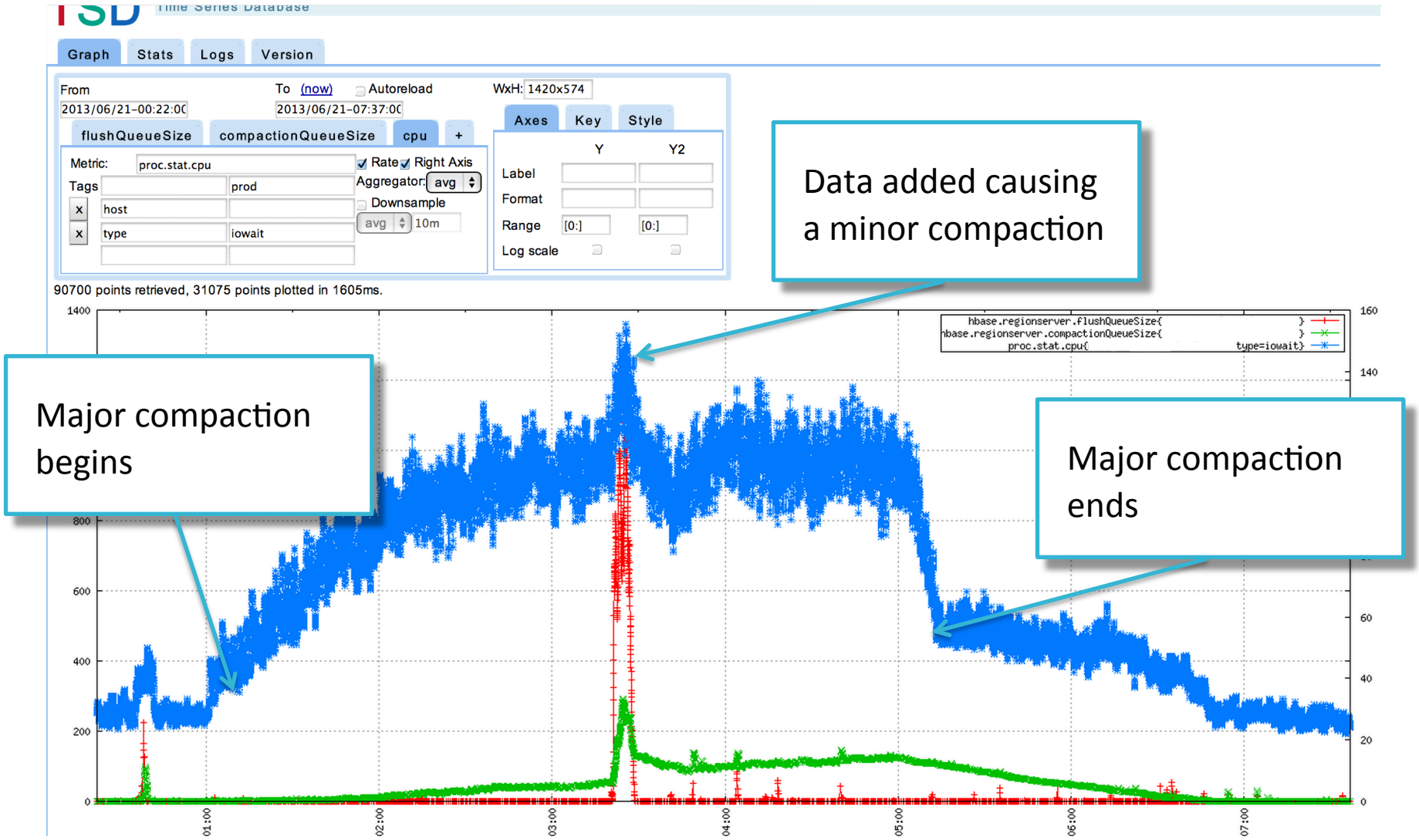
```
env COMPRESSION=lzo HBASE_HOME=path/to/hbase-0.98.6
./src/create_table.sh
```

- **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.tsd 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>`