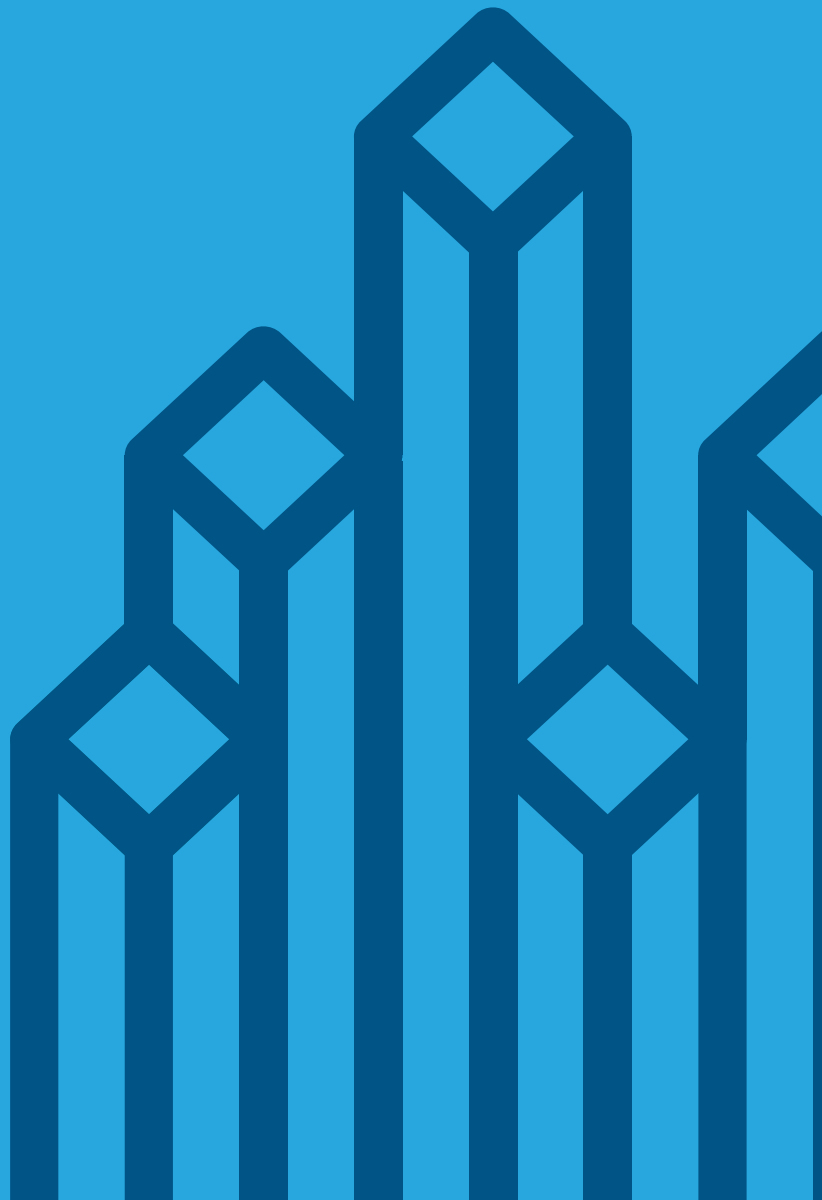
cloudera®

Cloudera Training for Apache HBase



201505

cloudera®

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	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

Users Table Divided into Regions

Row key	Users Table – Region 1	
aaronsona	fname: Aaron	lname: Aaronson
harrise	fname: Ernest	lname: Harris

Row key	Users Table – Region 2	
jordena	fname: Adam	lname: Jorden
laytonb	fname: Bennie	lname: Layton

Row key	Users Table – Region 3	
millerb	fname: Billie	lname: Miller
nununezw	fname: William	lname: Nunez
rossw	fname: Willie	lname: Ross

Row key	Users Table – Region 4	
sperberp	fname: Phyllis	lname: Sperber
turnerb	fname: Brian	lname: Turner
walkerm	fname: Martin	lname: Walker
zykowskiz	fname: Zeph	lname: Zykowski

Regions Served by RegionServers

RegionServer 1

Row key	Users Table – Region 1	
aaronsona	fname: Aaron	lname: Aaronson
harris		

Row key	Users Table – Region 4	
sperberp	fname: Phyllis	lname: Sperber
turnerb	fname: Brian	lname: Turner
walkerm	fname: Martin	lname: Walker
zykowskiz	fname: Zeph	lname: Zykowski

RegionServer 2

Row key	Users Table – Region 2	
jordena	fname: Adam	lname: Jorden
laytonb		

Row key	Users Table – Region 3	
millerb	fname: Billie	lname: Miller
nununezw	fname: Willam	lname: Nunez
rossw	fname: William	lname: Ross

Chapter Topics

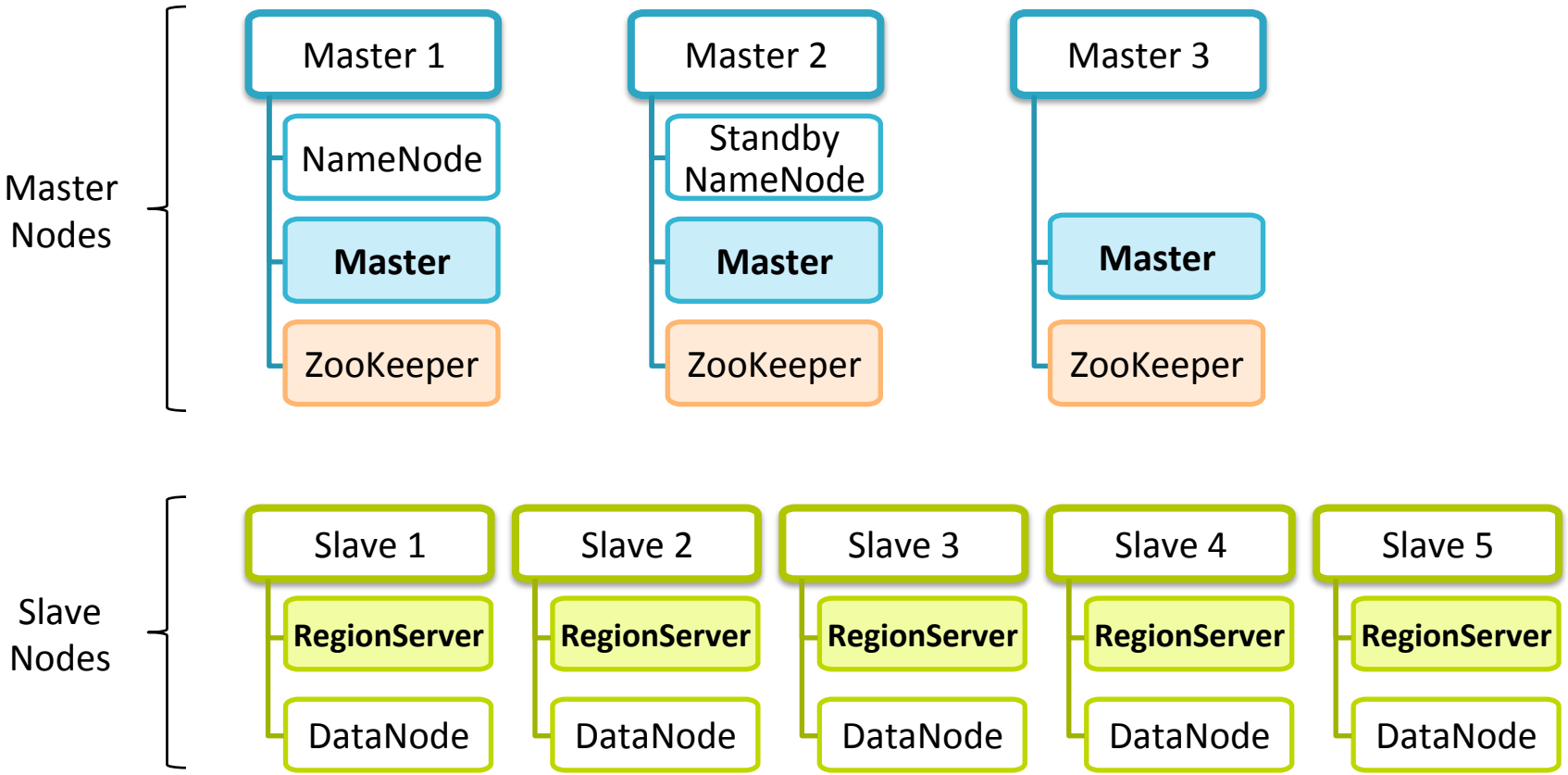
HBase Architecture Fundamentals

- 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

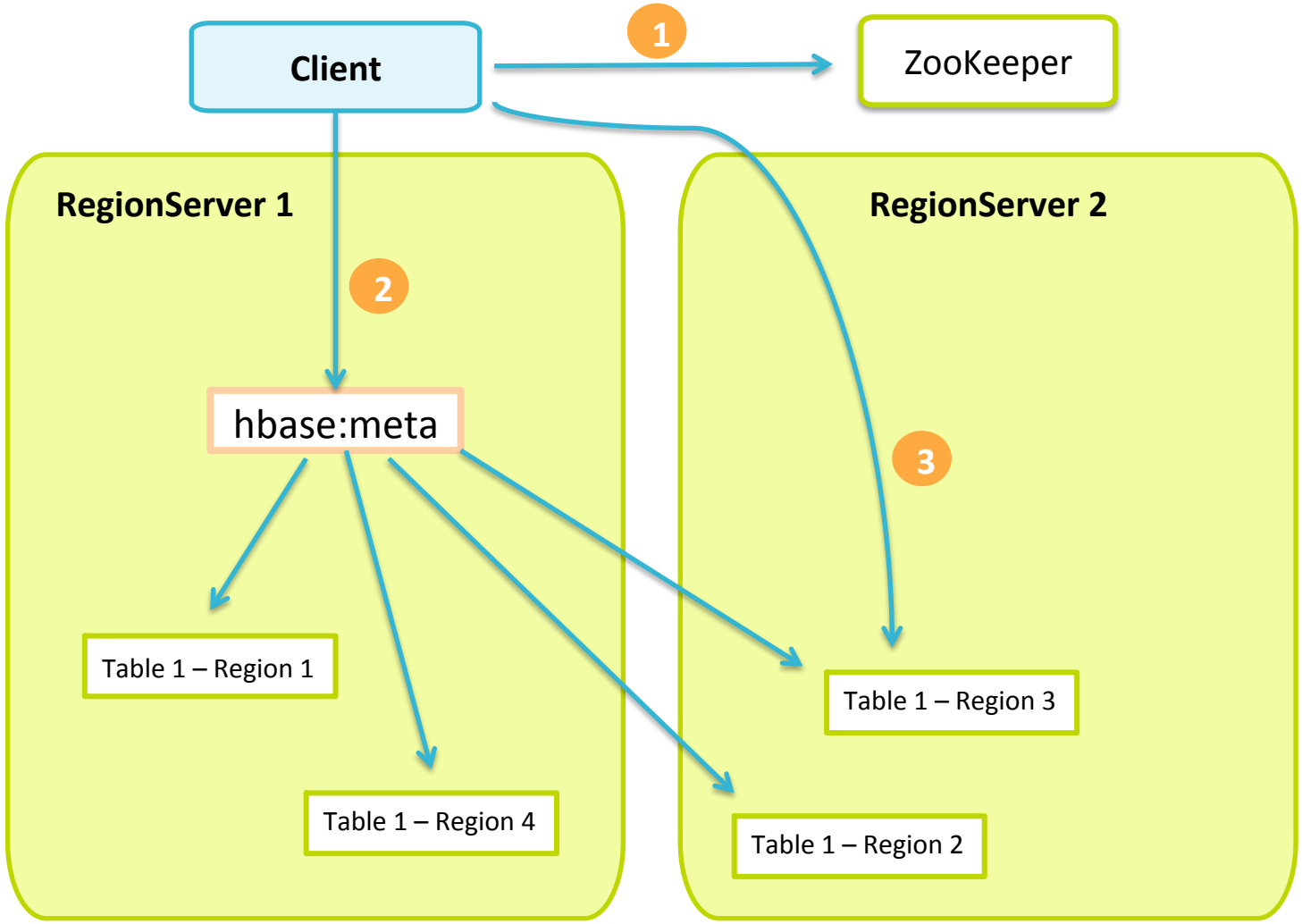
Table Types

- **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

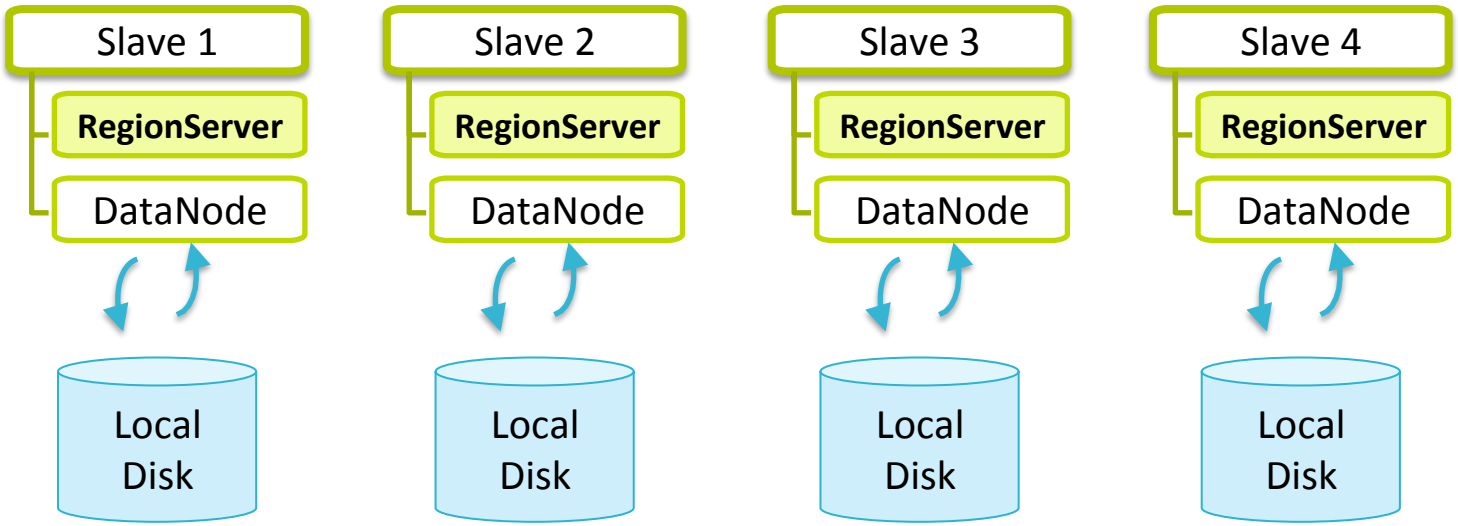
Chapter Topics

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**

cloudera®

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**
  - Amy 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

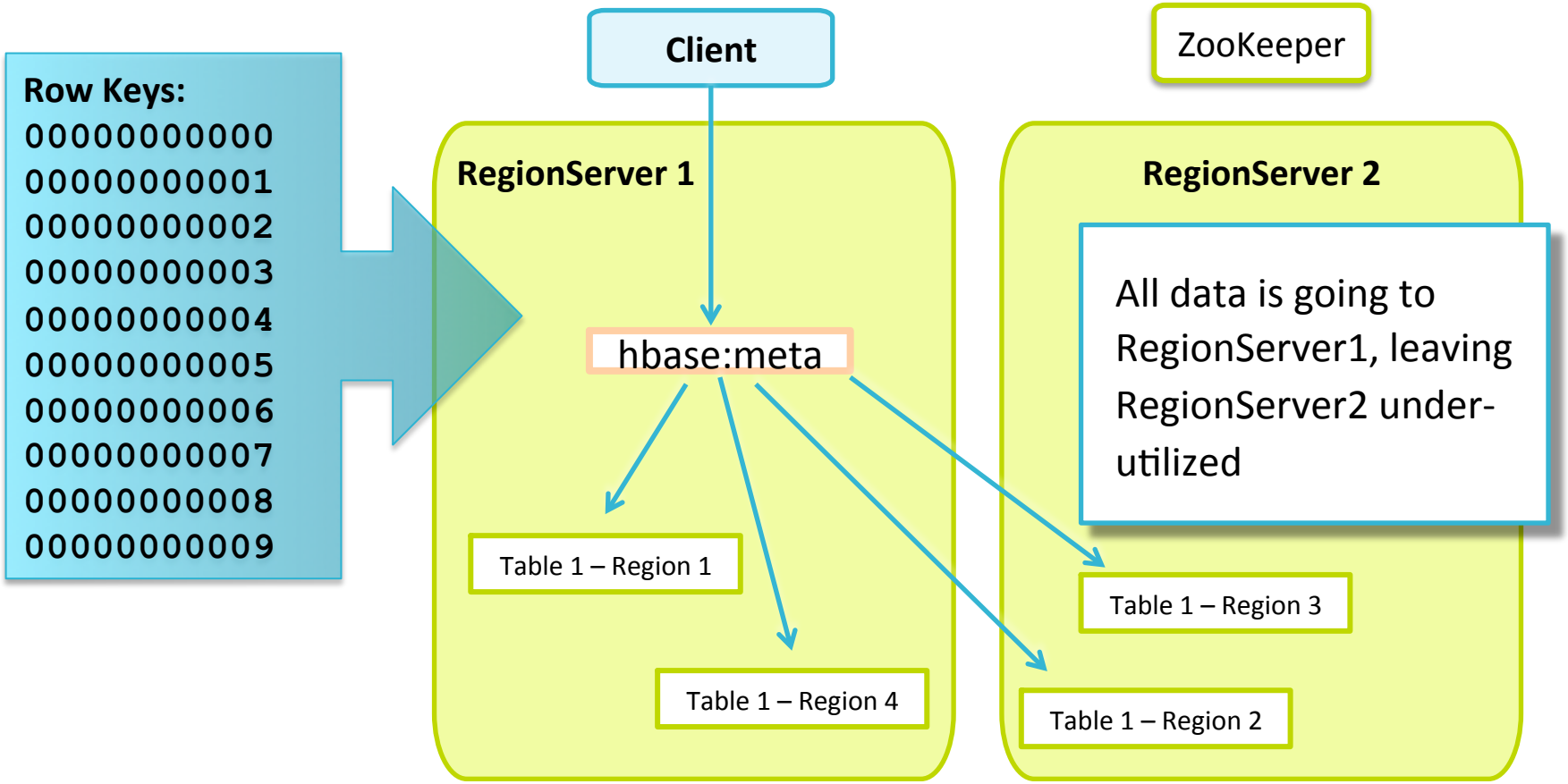
- **A write-heavy application needs to be optimized to write as fast as possible**
- **Optimize the row key to spread the load evenly across RegionServers**
  - Using only one RegionServer's resources is inefficient
- **Normalized tables may result in faster updates**
  - Denormalized data may require 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**

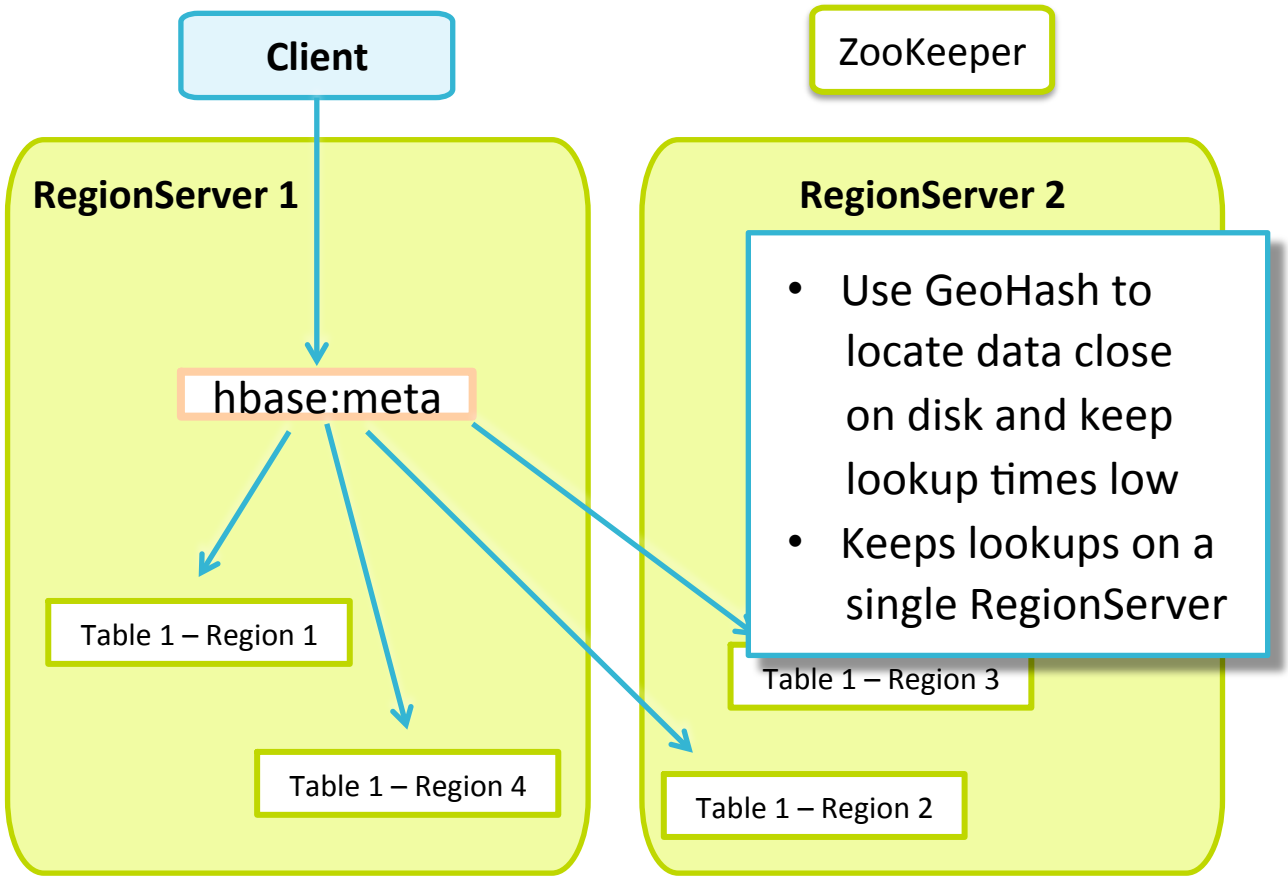


Geospatial Data Types

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

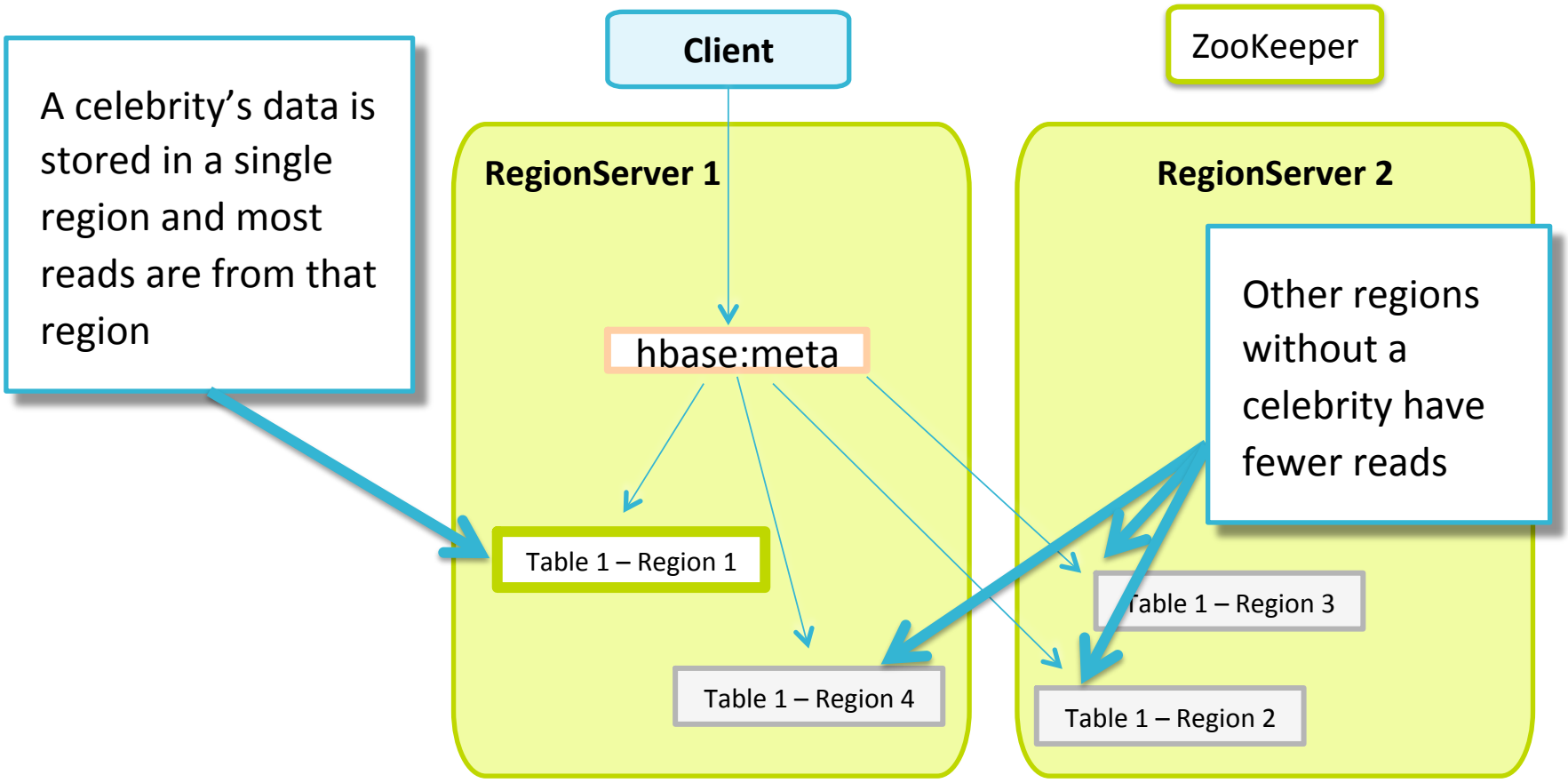
Latitude and longitude:
`39.5272° N, 119.8219° W`

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 `scan`s

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

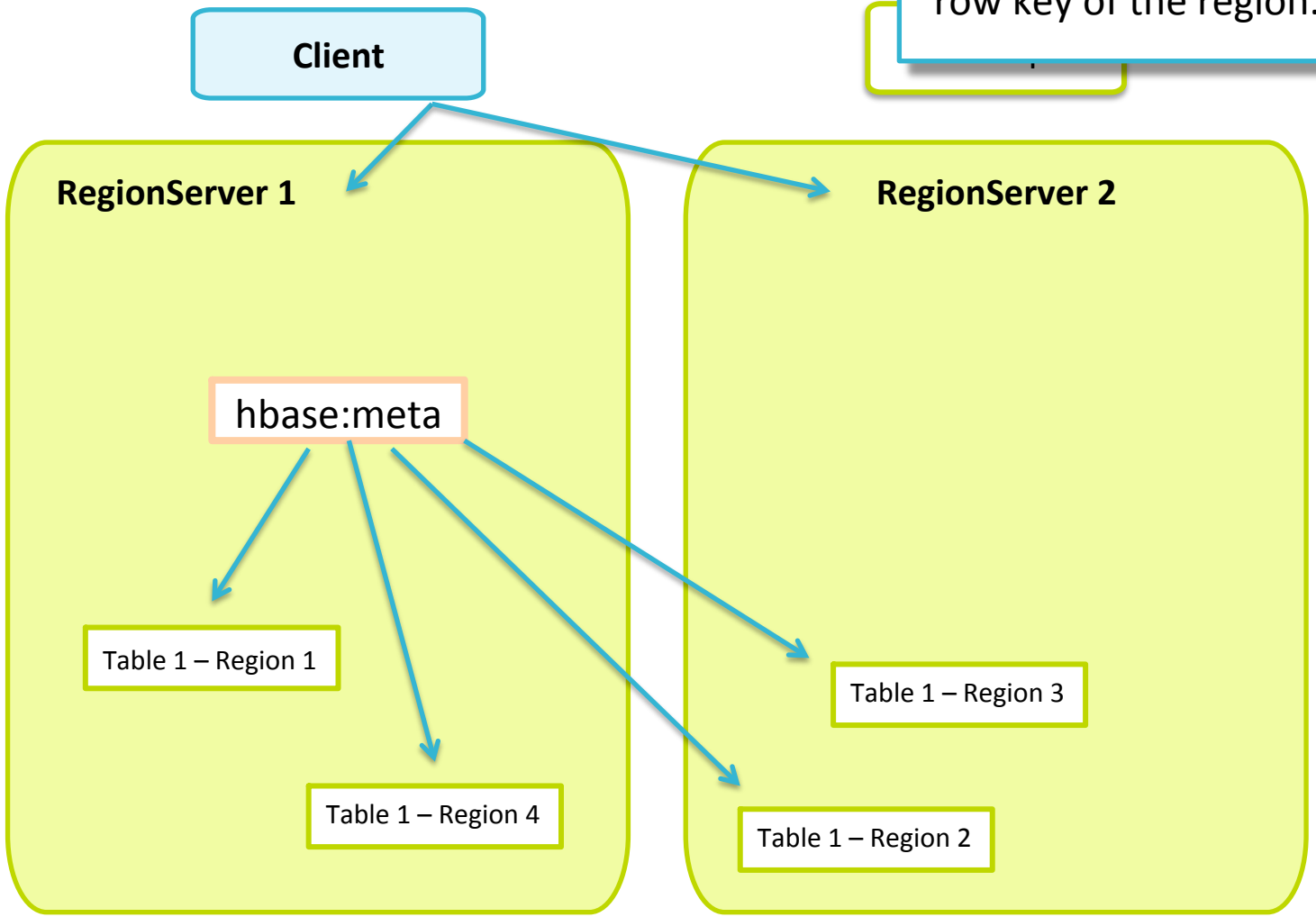
- **Querying based on row keys has the highest performance**
  - Row key queries have the least amount of processing to perform
  - Timerange-bound and column family reads can skip store files
- **Querying on column values has the lowest performance**
  - Value-based filtering is a full table scan
  - Each column's value must 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:	Incremental Row Key:
20130704-0130	00000000000
20130704-0134	00000000001
20130704-0246	00000000002
20130704-0252	00000000003
20130704-0414	00000000004
20130704-0533	00000000005
20130704-0721	00000000006
20130704-0929	00000000007

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:

```
00000000000
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:

```
00000000000:775
00000000001:314
00000000002:314
00000000003:310
00000000004:916
00000000005:925
00000000006:775
```

Promoted Row Key:

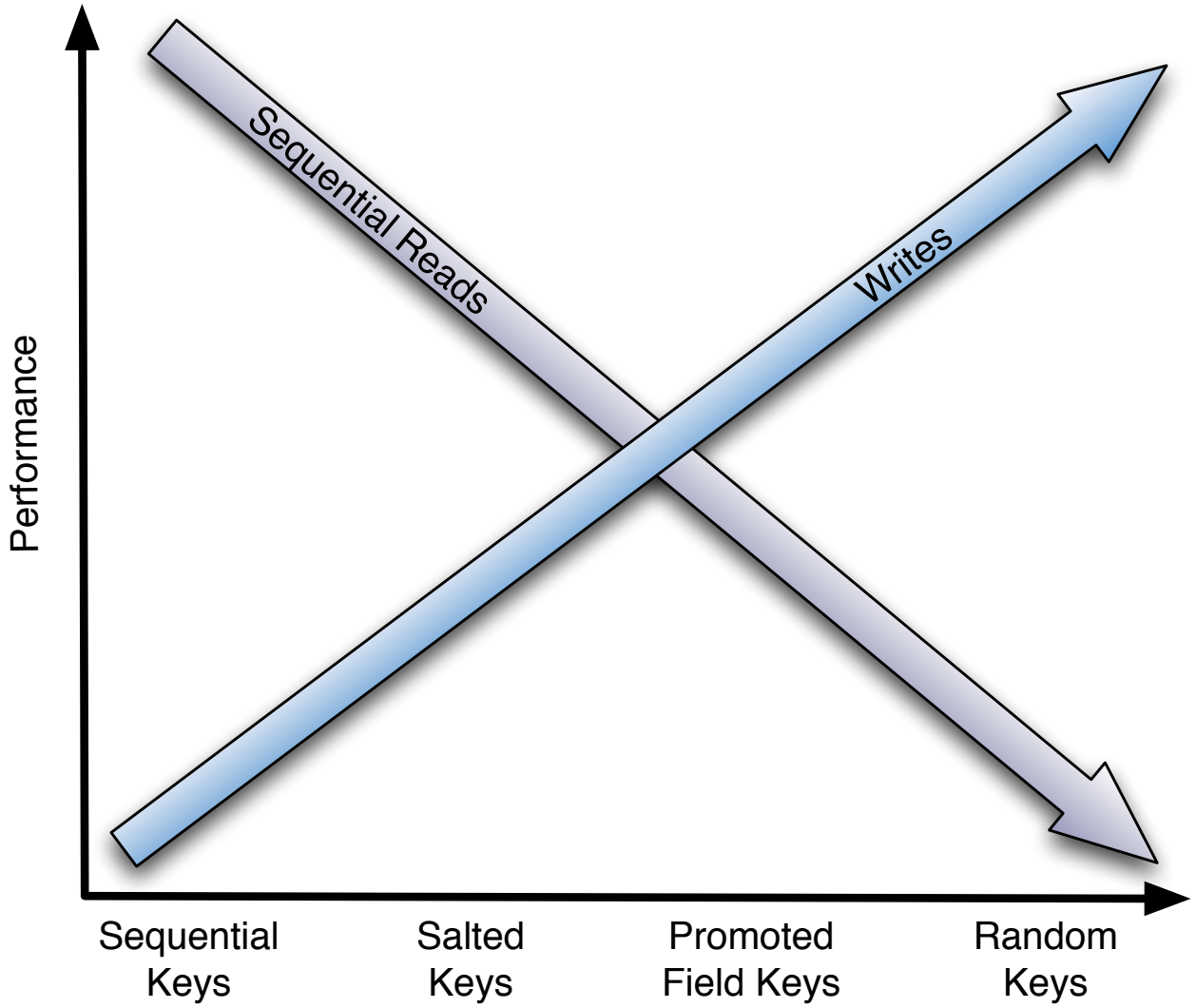
```
775:00000000000
314:00000000001
314:00000000002
310:00000000003
916:00000000004
925:00000000005
775:00000000006
```

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:	Random MD5 Hashed Row Key:
00000000000	645a8aca5a5b84527c57ee2f153f1946
00000000001	d67f0826d4c0aa7e3ea5861616a822b2
00000000002	c93c5cedf7fba468e0fe2c845837abc7
00000000003	6a1ae0e285acaf40dc30d13b702e6470
00000000004	e57ea6134fc5278023292f1941dff865
00000000005	63b307e583982c0746a5617e94f12dca
00000000006	0d51268ce5ae7eed7e1ccd6d3859d033

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**

cloudera®

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);

HTa

HCo

des
adm

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"));

des
adm

adm
```

- 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 =
HBase
HBase

HTabl
    H
HColu
    H

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

admin.close();
```

- 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

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(byteArray);

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
co
```

- 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                                                    g);
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 (R
    Str
    byt
    Str
}
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.clo
```

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.p

table.
conn.c
```

- 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);

table.
conn.c
```

- 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**

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

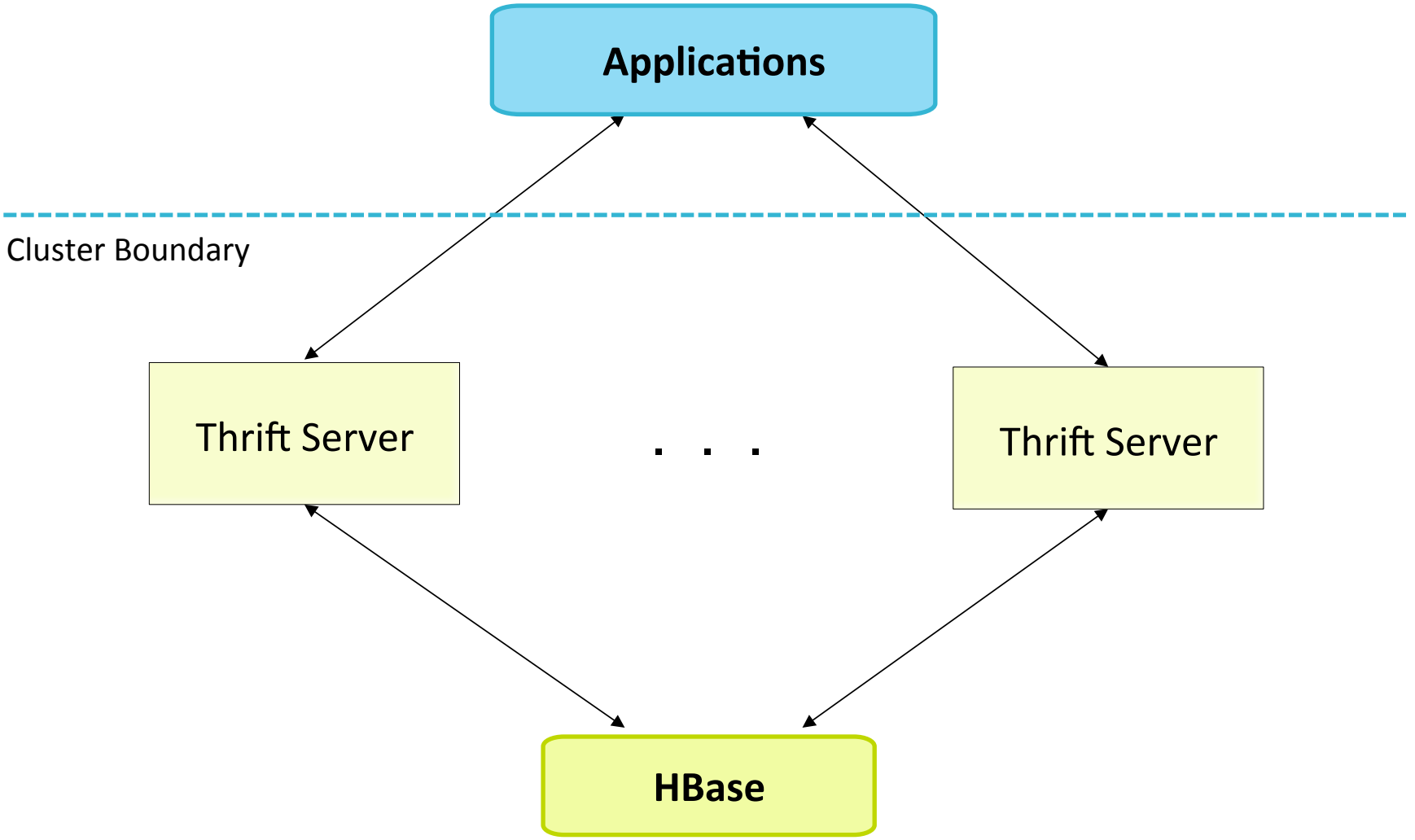
Reprise: Accessing Data in HBase

- **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**

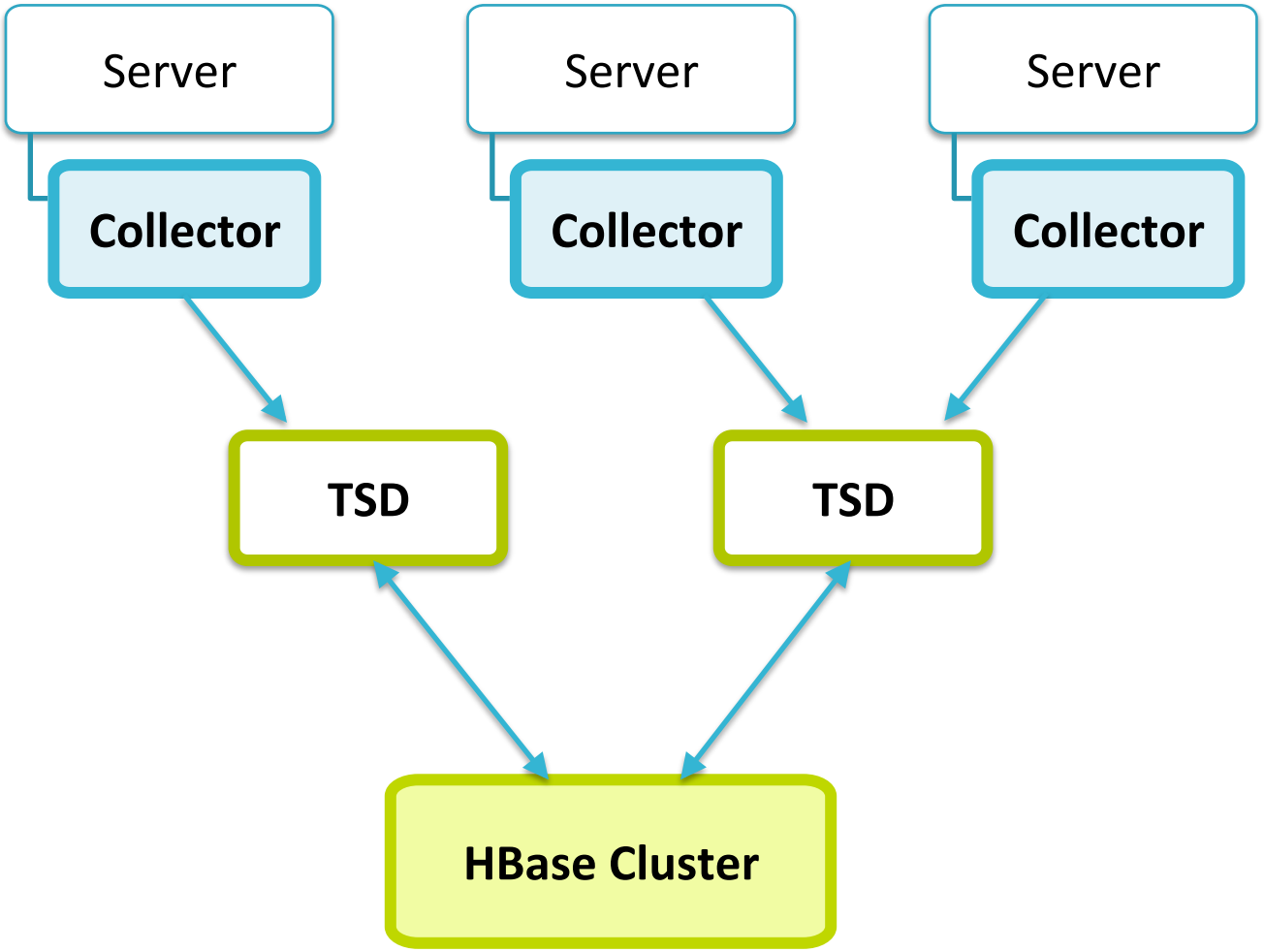
OpenTSDB

- **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:**

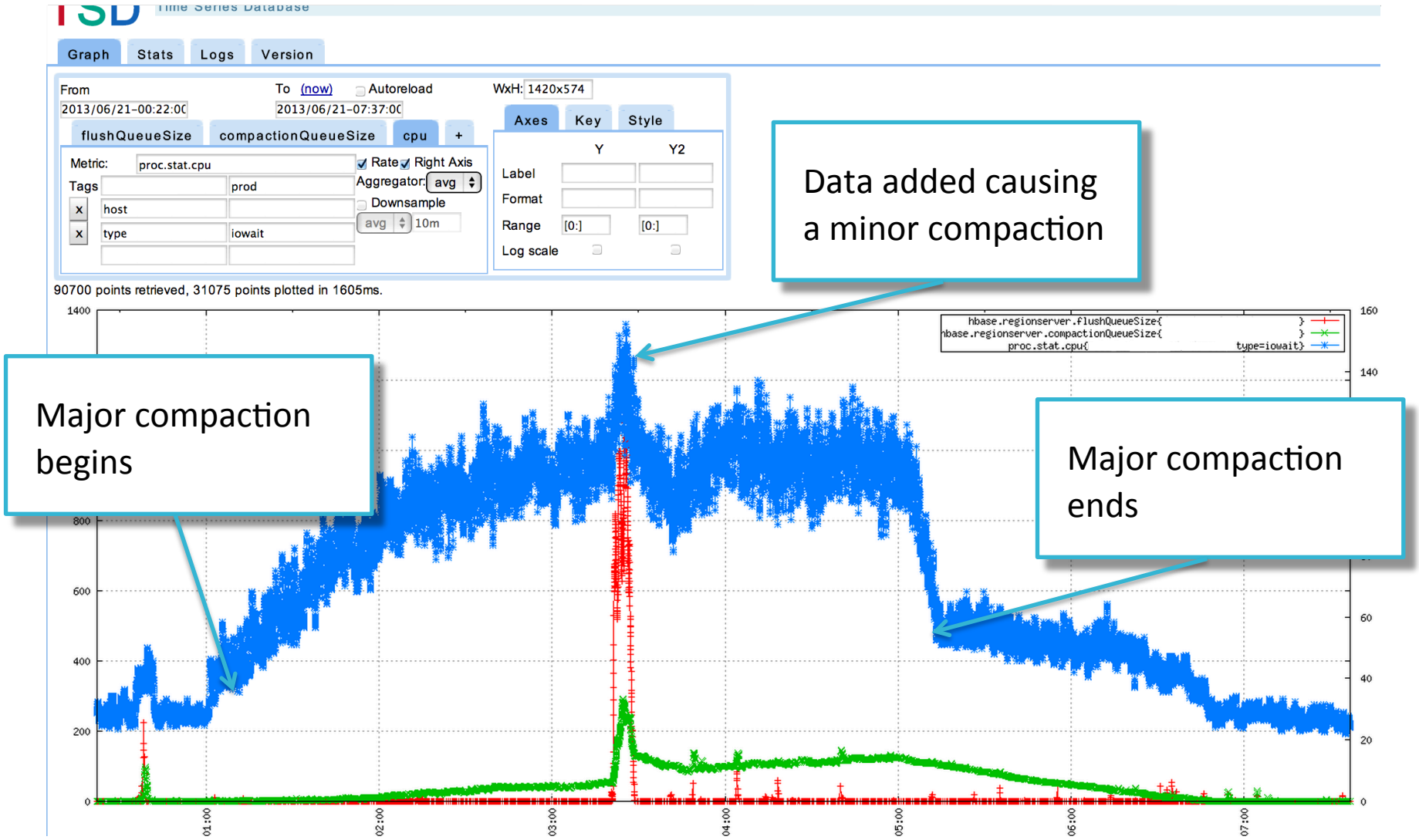
```
env COMPRESSION=lzo HBASE_HOME=path/to/hbase-0.98.6
./src/create_table.sh
```

- **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>`