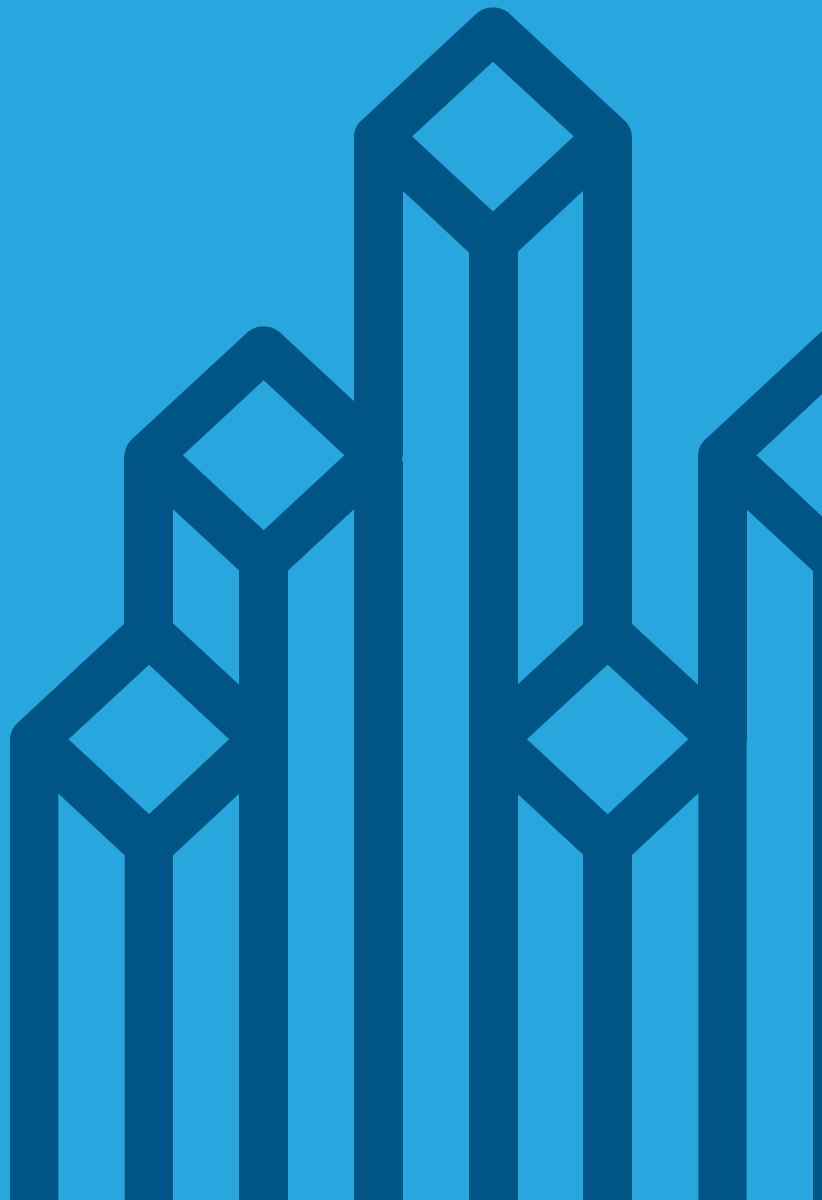
cloudera®

Cloudera Training for Apache HBase



201505

cloudera®

HBase Performance Tuning

Chapter 11

Course Chapters

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

HBase Performance Tuning

In this chapter you will learn

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

Chapter Topics

HBase Performance Tuning

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

Column Family Design

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

Column Family Considerations

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

Column Family Attributes

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

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

COMPRESSION

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

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

BLOCKSIZE

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

Bloom Filters

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

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

Bloom Filter Considerations

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

Chapter Topics

HBase Performance Tuning

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

Schema Fundamentals (1)

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

Schema Fundamentals (2)

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

Tall-Narrow vs Flat-Wide Tables

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

Secondary Indexes

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

Secondary Indexes – HBase Options

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

Chapter Topics

HBase Performance Tuning

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

Caching Metadata and Data

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

Configuring the L1 and L2 Caches

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

Chapter Topics

HBase Performance Tuning

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

HBase Memory Considerations

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

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

Chapter Topics

HBase Performance Tuning

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

Problems with Hotspotting

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

Approaches to Address Hotspotting

- **Salting**
  - Place a small, calculated hash in front of the real data to randomize the row key
  - e.g., `<salt><timestamp>` instead of just `<timestamp>`
- **Promoted Field Keys**
  - A field is moved in front of the incremental or timestamp field
  - e.g., `<sourceid><timestamp>` instead of `<timestamp><sourceid>`
- **Pseudo-Random**
  - Process the data using a one-way hash like MD5
  - e.g., `<md5(timestamp)>` instead of just `<timestamp>`

Chapter Topics

HBase Performance Tuning

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

Regions

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

Pre-Splitting Regions

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

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

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

Region Splits

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

Chapter Topics

HBase Performance Tuning

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

Hands-On Exercise: Detecting Hotspots

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

Chapter Topics

HBase Performance Tuning

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

Key Points

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

cloudera®

HBase Administration and Cluster Management

Chapter 12

Course Chapters

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

HBase Administration and Cluster Management

In this chapter you will learn

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

Aside: Existing Cluster, Using Cloudera Manager

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

Chapter Topics

HBase Administration and Cluster Management

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

HBase Master

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

HBase Master Background Processes

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

RegionServer (1)

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

RegionServer (2)

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

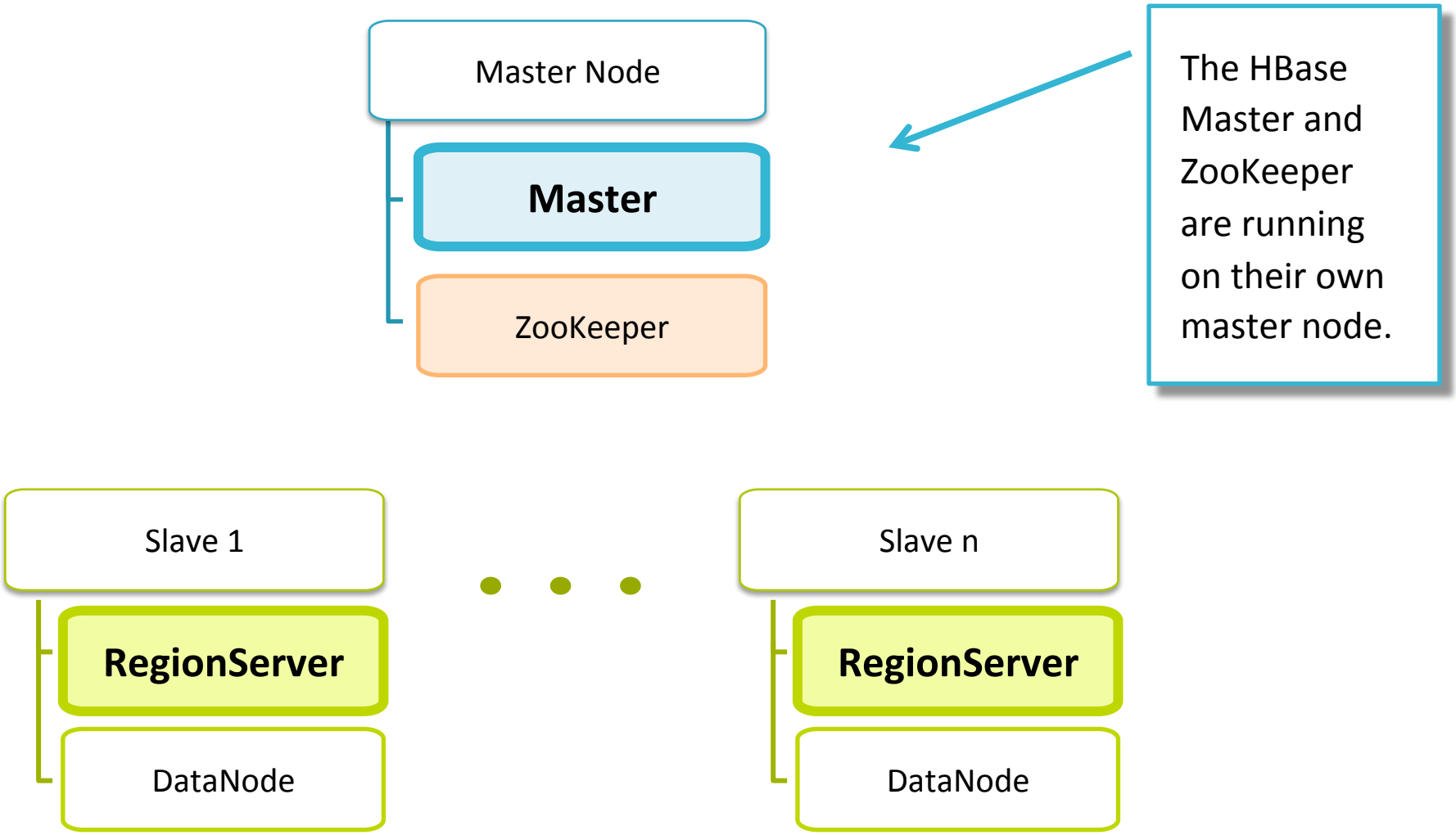
RegionServer Background Processes

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

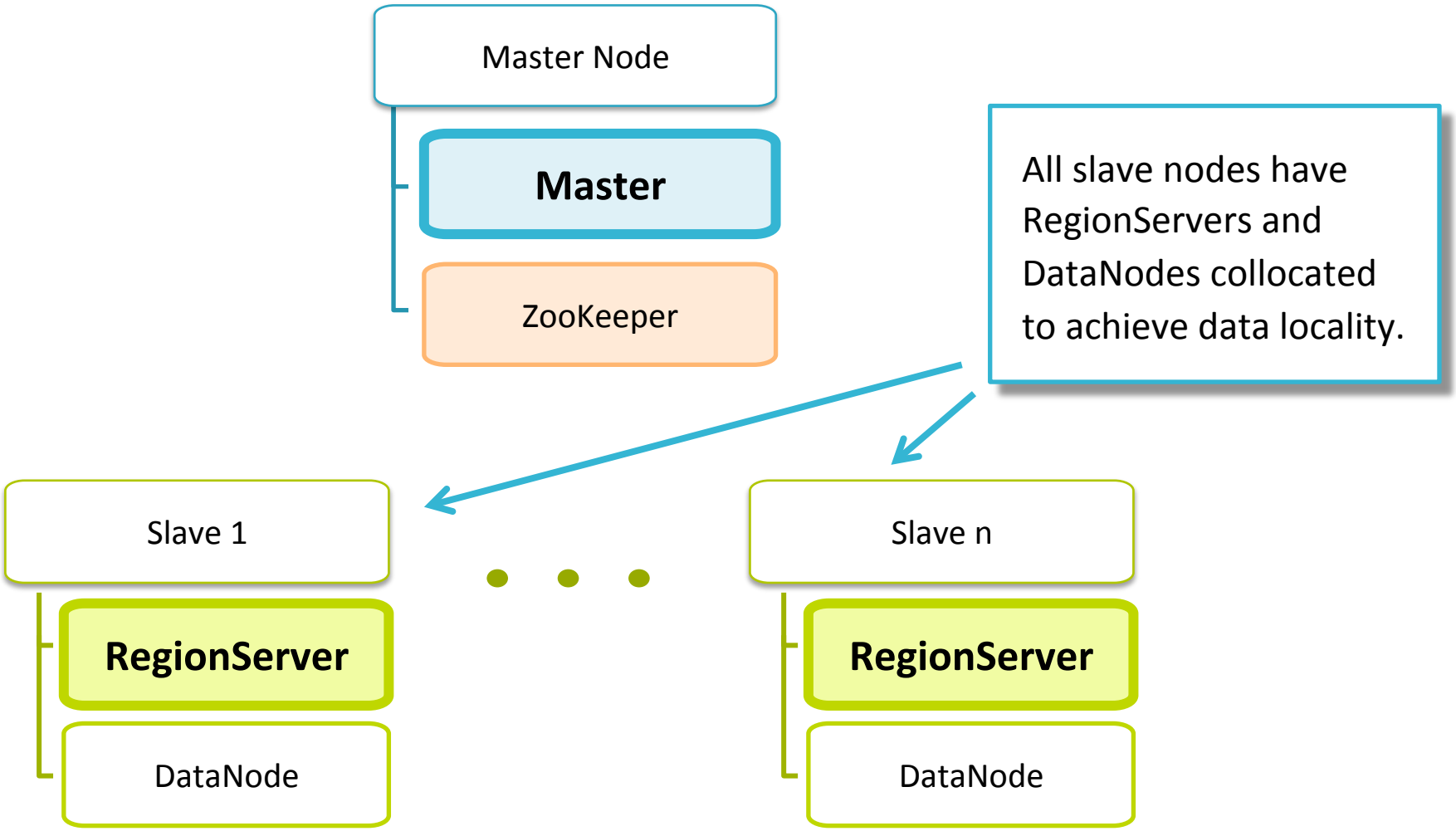
Placement of HBase Daemons

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

Small HBase Cluster Diagram: Master Node



Small HBase Cluster Diagram: Slave Nodes



Chapter Topics

HBase Administration and Cluster Management

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

What is ZooKeeper?

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

ZooKeeper Data

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

Highly Available HBase

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

ZooKeeper

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

ZooKeeper Considerations

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

Chapter Topics

HBase Administration and Cluster Management

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

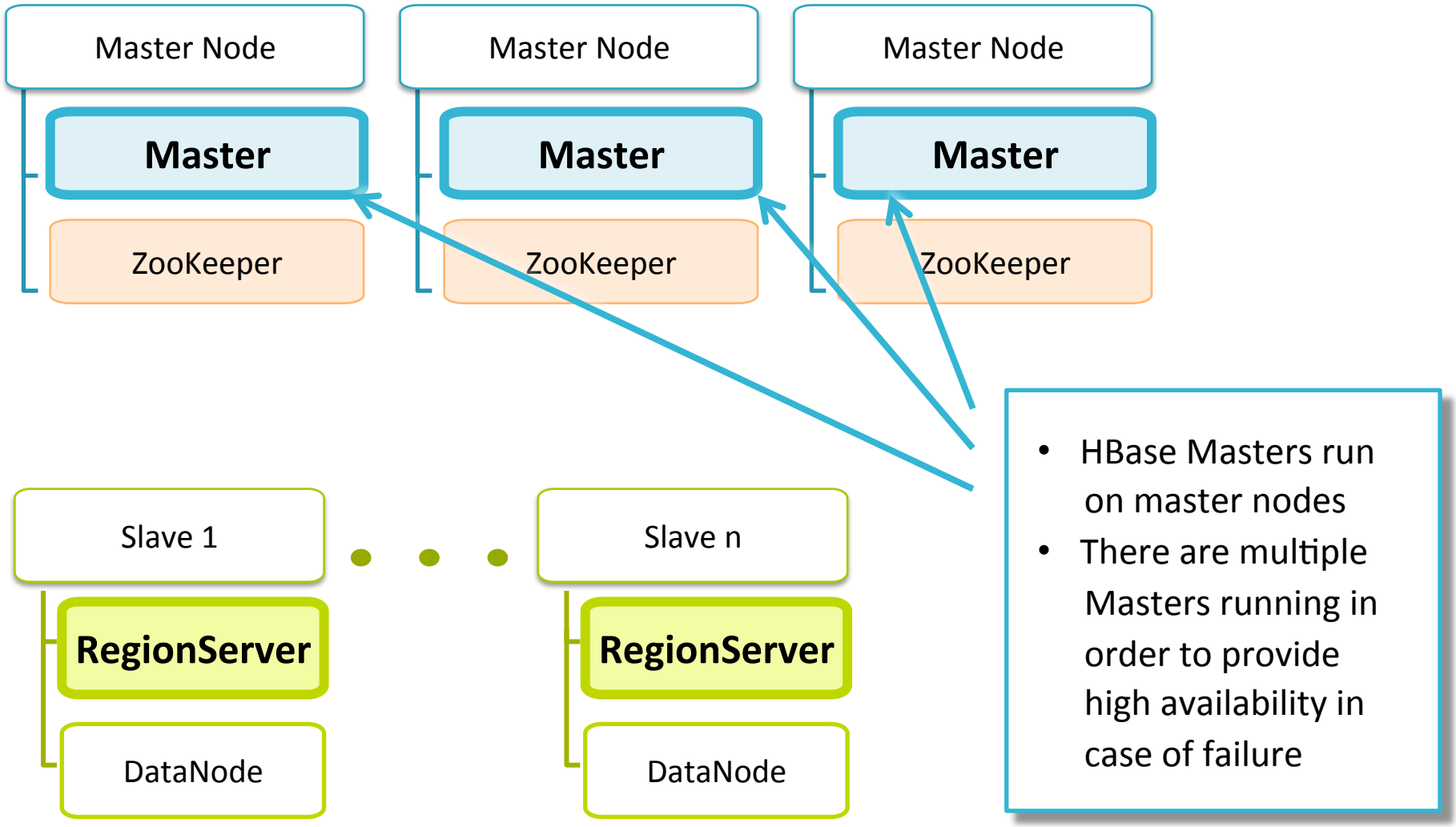
Detecting Failures with ZooKeeper (1)

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

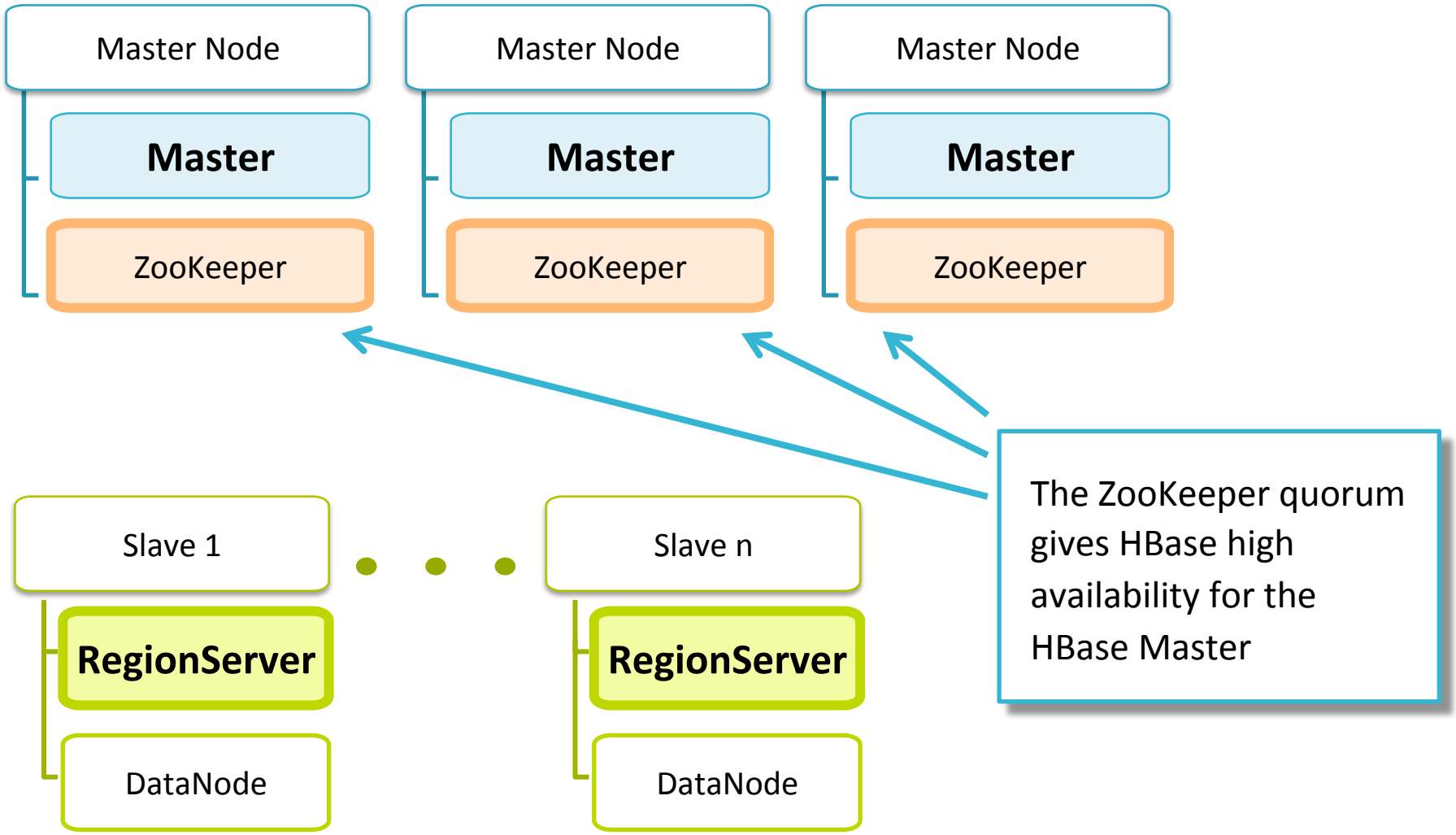
Detecting Failures with ZooKeeper (2)

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

Highly Available HBase Cluster Diagram: HBase Masters



Highly Available HBase Cluster Diagram: ZooKeeper Quorum



Chapter Topics

HBase Administration and Cluster Management

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

What the Balancer Does

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

How to Use the Balancer

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

Using the HDFS Balancer: Caution

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

Chapter Topics

HBase Administration and Cluster Management

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

HBase Errors

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

Fixing HBase (1)

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

Fixing HBase (2)

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

Fixing HBase with `hbck` (1)

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

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

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

```
Status: OK
```

Fixing HBase with `hbck` (2)

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

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

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

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

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

Fixing HBase with `hbck` (3)

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

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

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

Chapter Topics

HBase Administration and Cluster Management

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

HBase Security

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

Configuring HBase Security

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

HBase Shell – Granting Access (1)

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

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

HBase Shell – Granting Access (2)

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

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

- **To grant read and write permissions for a table:**

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

- **To grant read permissions for a specific column descriptor:**

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

- **To revoke a permission for a user:**

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

Chapter Topics

HBase Administration and Cluster Management

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

Key Points

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

cloudera®

HBase Replication and Backup

Chapter 13

Course Chapters

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

HBase Replication and Backup

In this chapter you will learn

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

Chapter Topics

HBase Replication and Backup

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

Replication and Backup

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

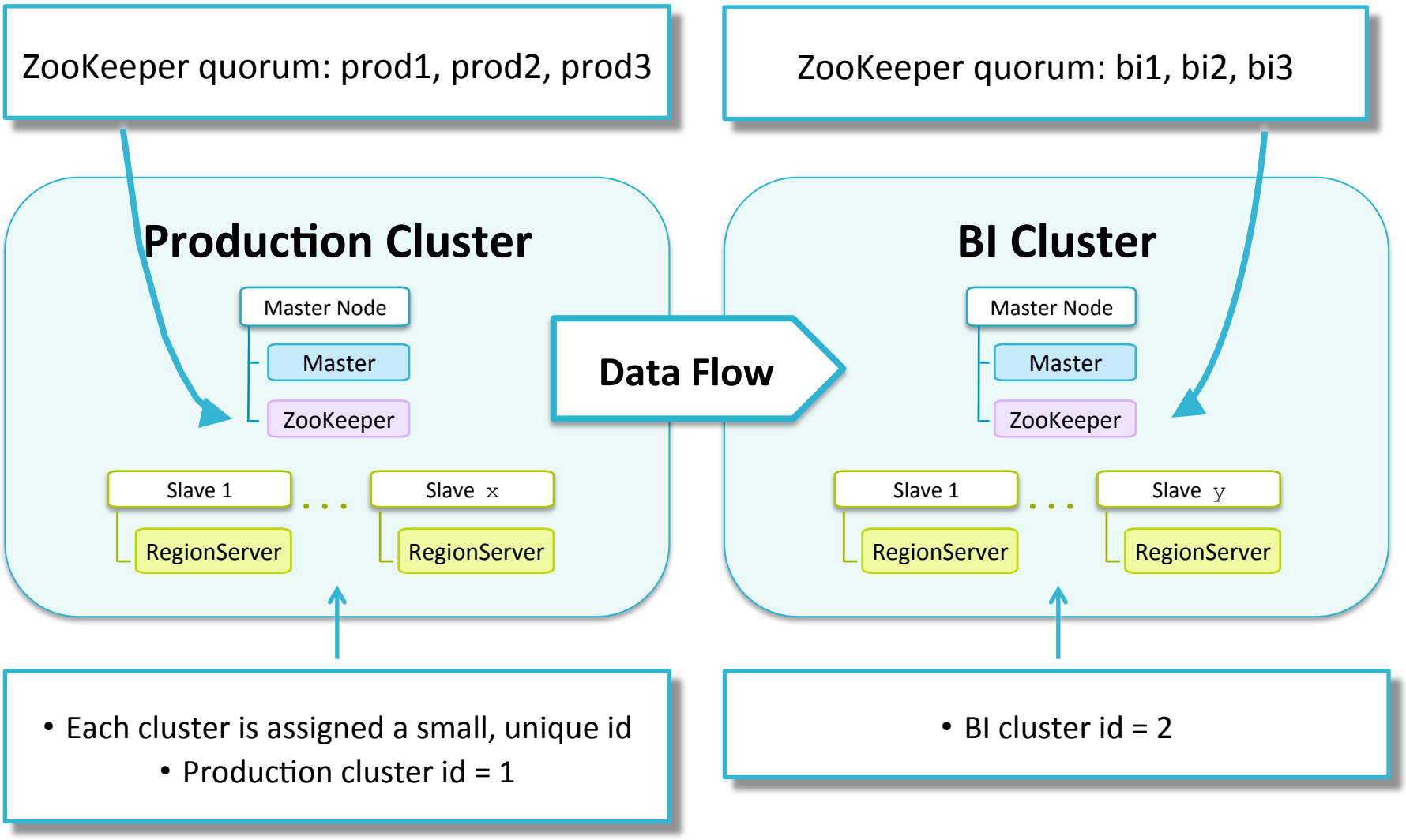
HBase Replication

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

HBase Replication Caveats

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

HBase Master-Slave Replication Diagram



Setting Up Replication (1)

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

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

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

Setting Up Replication (2)

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

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

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

Types Of Replication

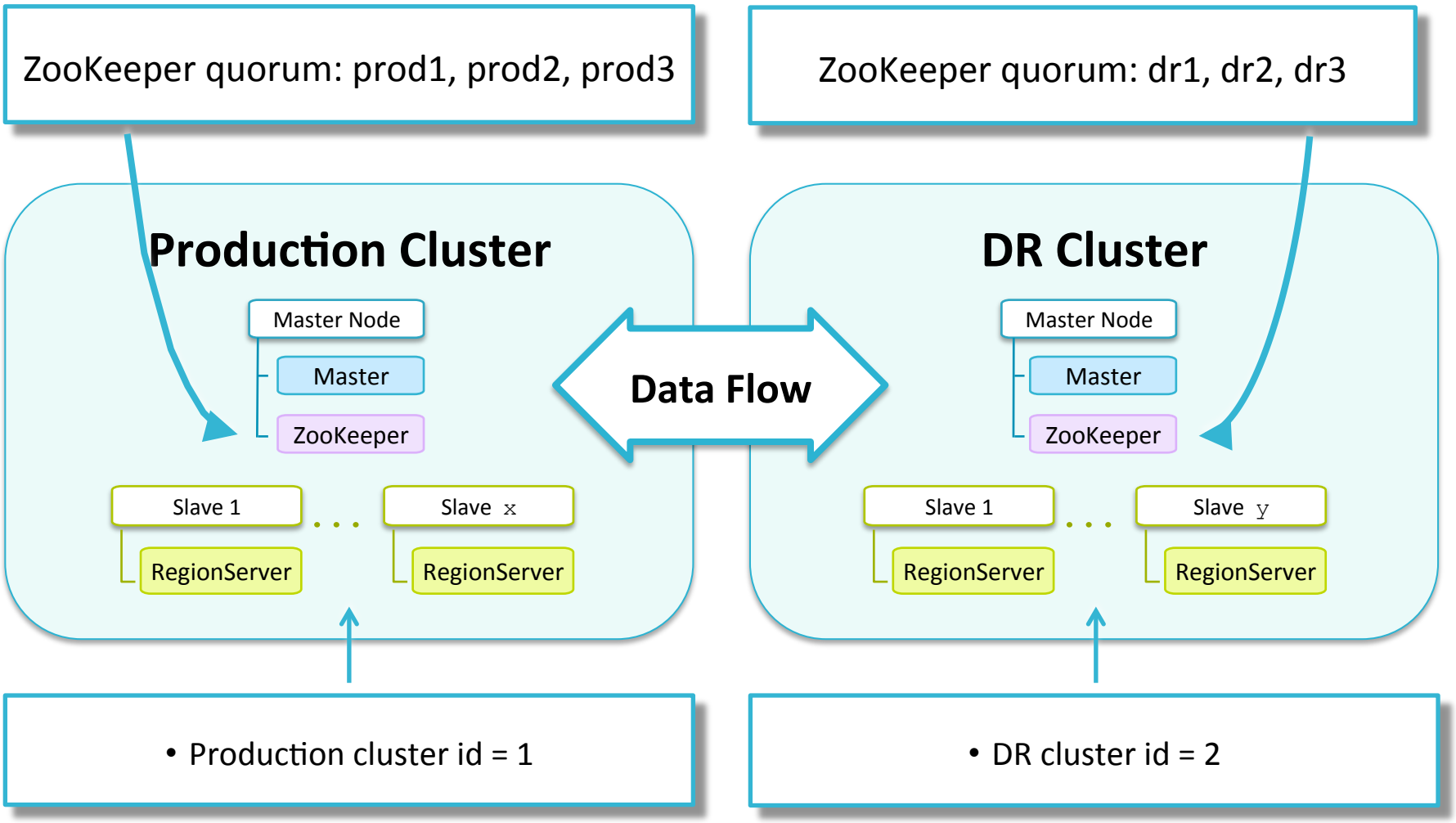
- **There are three types of replication strategies**
  - Master-slave replication is unidirectional from master to slave
  - Master-master replication is bidirectional between clusters
  - Cyclic replication is for clusters with > 2 clusters to replicate
- **To configure master-master:**
  - Turn on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

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

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

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

HBase Master-Master Replication Diagram



Verifying Replication (1)

- **Check that peers are set up correctly**

```
hbase> list_peers
```

  - Verify that the peers are pointing to the correct cluster id
  - Verify that all peers are listed to replicate to

- **Run the VerifyReplication to verify that each row was replicated correctly**

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

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

Verifying Replication (2)

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

Chapter Topics

HBase Replication and Backup

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

HBase Backups

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

CopyTable

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

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

Backup and Restore

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

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

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

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

ImportTSV (1)

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

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

ImportTSV (2)

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

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

- **ImportTSV issues a `put` per row**
  - Can be configured to bulk load by creating an HFile as the output
  - Use `importtsv.bulk.output` and supply the path in HDFS to write to and the table name

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

Bulk Loading

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

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

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

Full Backup

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

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

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

Snapshot

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

Using Snapshots (1)

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

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

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

Using Snapshots (2)

- **To create a new table from a snapshot:**

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

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

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

  - The table must be disabled beforehand

- **To list existing snapshots:**

```
hbase> list_snapshots
```

Exporting Snapshots

- **To export a snapshot to another cluster:**

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

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

Snapshot Caveats

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

MapReduce Over Snapshots

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

Backup Methods

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

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

Backups and Replication

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

Chapter Topics

HBase Replication and Backup

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

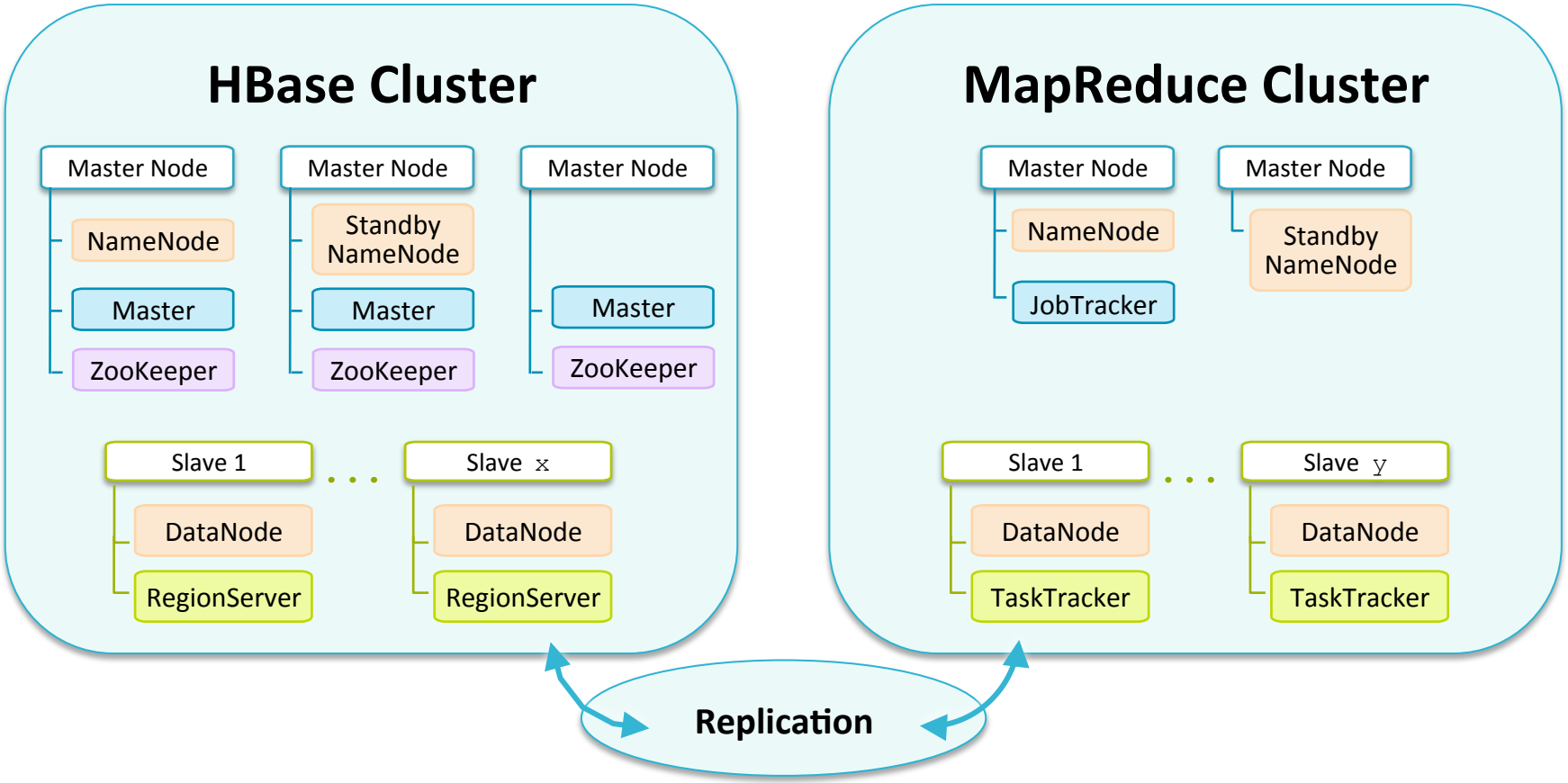
MapReduce Considerations (1)

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

MapReduce Considerations (2)

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

MapReduce and Low Latency HBase Cluster Diagram



- The HBase cluster may need to satisfy low latency requirements.
- Best practice: Deploy separate HBase and MapReduce clusters.

Request Throttling

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

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

Chapter Topics

HBase Replication and Backup

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

Hands-On Exercise: Administration

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

Chapter Topics

HBase Replication and Backup

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

Key Points

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

cloudera®

Using Hive and Impala with HBase

Chapter 14

Course Chapters

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

The HBase Ecosystem

In this chapter you will learn

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

Chapter Topics

The HBase Ecosystem

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

Hive: Motivation

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

Hive: Introduction

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

The Hive and HBase Data Model

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

Starting The Hive Shell

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

```
$ hive
```

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

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

Hive Basics: Creating External HBase Tables

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

hive> DESCRIBE user;
```

Basic SELECT Queries

- **Hive supports most familiar SELECT syntax**

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

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

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

Joining Tables

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

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

Improved Scan Performance

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

Impala

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

Impala on HBase

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

Chapter Topics

The HBase Ecosystem

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

Hands-On Exercise: Hive and HBase

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

Chapter Topics

The HBase Ecosystem

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

Key Points

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

cloudera®

Conclusion

Chapter 15

Conclusion

During this class, you have learned

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

Thank You!

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

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

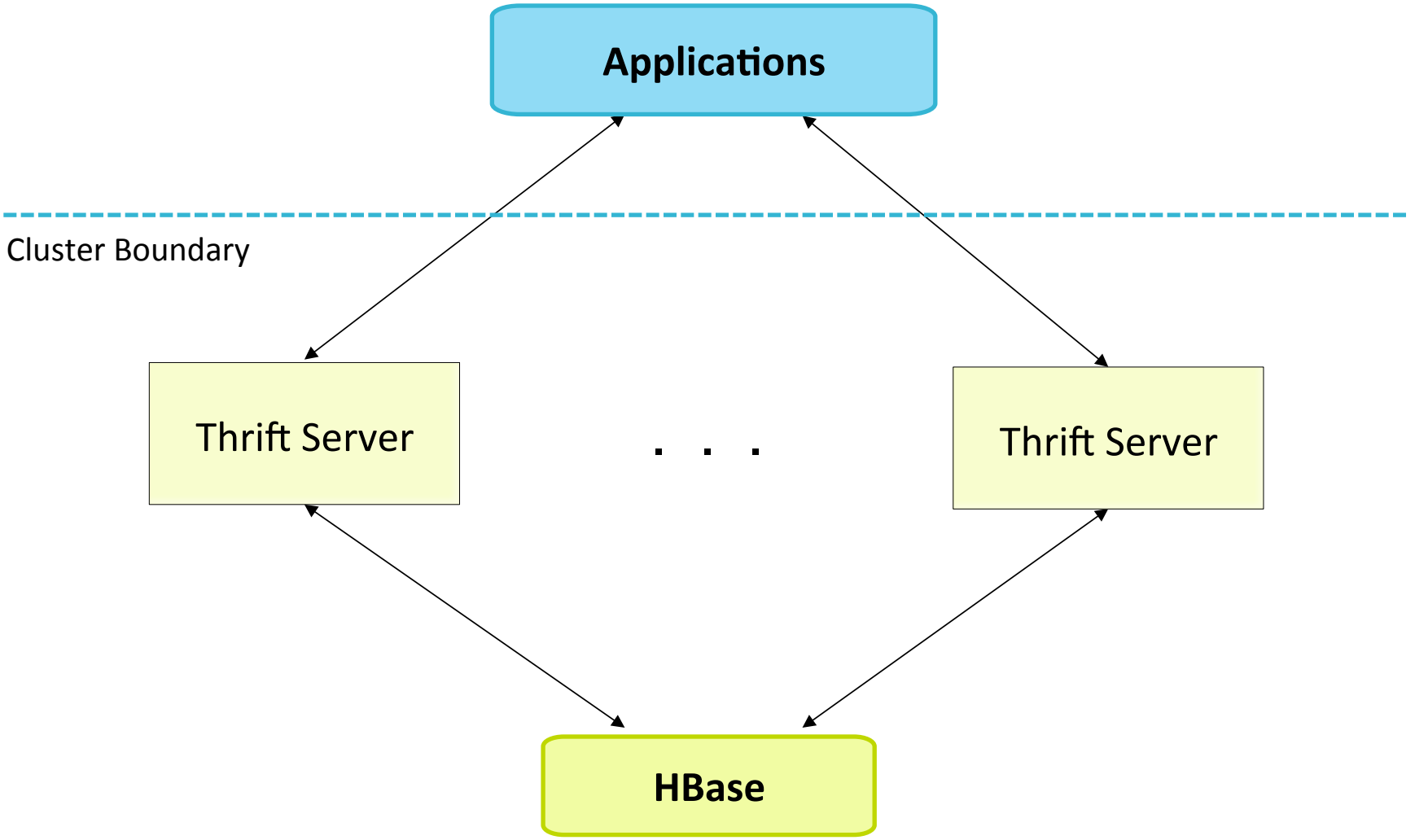
Reprise: Accessing Data in HBase

- **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**

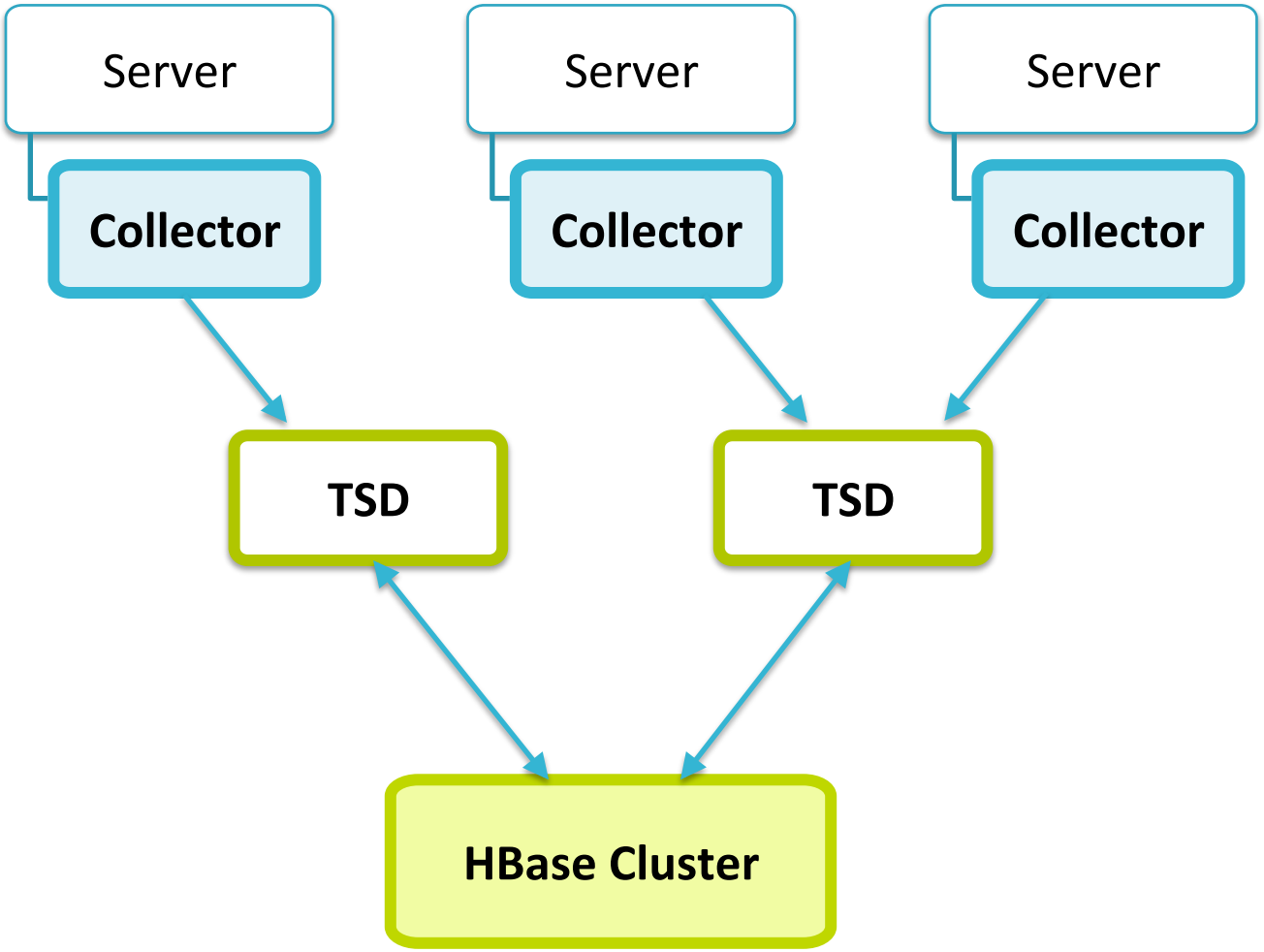
OpenTSDB

- **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:**

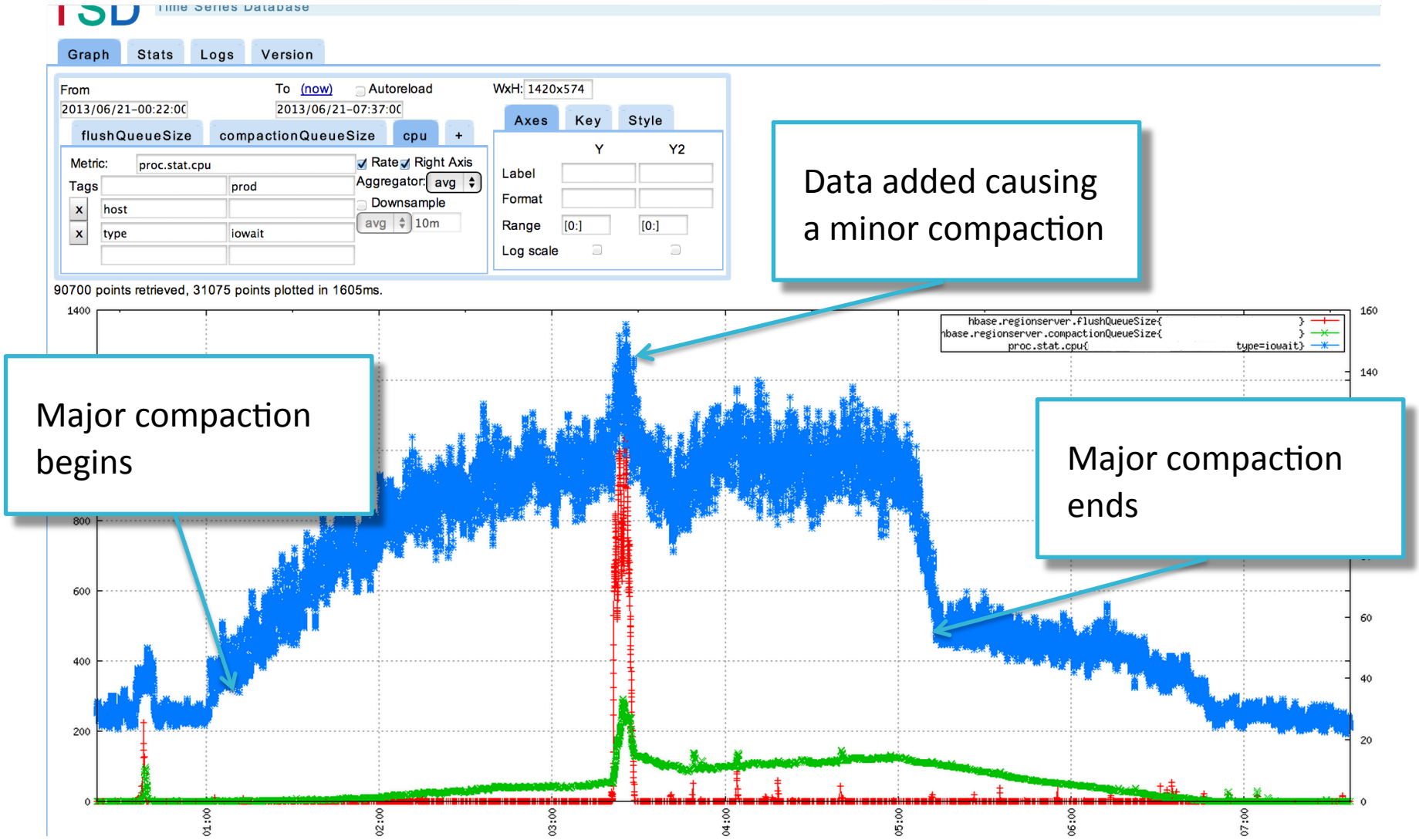
```
env COMPRESSION=lzo HBASE_HOME=path/to/hbase-0.98.6
./src/create_table.sh
```

- **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>`