

Apache HBase® Reference Guide

Version 4.0.0-alpha-1-SNAPSHOT

Apache HBase Team

Contents

Preface

1. About This Guide
2. Contributing to the Documentation
3. Heads-up if this is your first foray into the world of distributed computing...
4. Reporting Bugs
5. Support and Testing Expectations

Getting Started

1. Quick Start - Standalone HBase
2. JDK Version Requirements
3. Get Started with HBase
4. Pseudo-Distributed for Local Testing
5. Fully Distributed for Production
6. Where to go next

Configuration

1. Configuration Files
2. Basic Prerequisites
3. Java
4. Hadoop
5. ZooKeeper Requirements
6. HBase run modes: Standalone and Distributed
7. Standalone HBase
8. Distributed
9. Fully-distributed
10. Choosing between the Classic Package and the BYO Hadoop Package
11. Advantages of the BYO Hadoop package:
12. Running and Confirming Your Installation
13. Default Configuration
14. *hbase-site.xml* and *hbase-default.xml*
15. HBase Default configuration
16. hbase-env.sh
17. *log4j2.properties*
18. Client configuration and dependencies connecting to an HBase cluster
19. Timeout settings
20. Example Configurations
21. Basic Distributed HBase Install
22. The Important Configurations

23. Required Configurations
24. Recommended Configurations
25. Other Configurations
26. Dynamic Configuration

Upgrading

1. HBase version number and compatibility
2. Aspirational Semantic Versioning
3. HBase API Surface
4. Rolling Upgrades
5. Rollback
6. Caveats
7. All service rollback
8. Rollback after HDFS rollback and ZooKeeper downgrade
9. Rollback after HDFS downgrade
10. Upgrade Paths
11. Upgrade from 2.x to 3.x
12. Upgrade from 2.0.x-2.2.x to 2.3+
13. Upgrade from 2.0 or 2.1 to 2.2+
14. Upgrading from 1.x to 2.x
15. Upgrading to 1.7.1+
16. Upgrading from pre-1.4 to 1.4+
17. Upgrading from pre-1.3 to 1.3+
18. Upgrading to 1.x

The Apache HBase Shell

1. Scripting with Ruby
2. Running the Shell in Non-Interactive Mode
3. HBase Shell in OS Scripts
4. Read HBase Shell Commands from a Command File
5. Passing VM Options to the Shell
6. Overriding configuration starting the HBase Shell
7. Shell Tricks

Data Model

1. HBase Data Model Terminology
2. Conceptual View
3. Physical View
4. Namespace
5. Table

6. Row
7. Column Family
8. Cells
9. Data Model Operations
10. Versions
11. Sort Order
12. Column Metadata
13. Joins
14. ACID

HBase and Schema Design

1. Schema Creation
2. Table Schema Rules Of Thumb

RegionServer Sizing Rules of Thumb

1. On the number of column families
2. Rowkey Design
3. Number of Versions
4. Supported Datatypes
5. Joins
6. Time To Live (TTL)
7. Keeping Deleted Cells
8. Secondary Indexes and Alternate Query Paths
9. Constraints
10. Schema Design Case Studies
11. Operational and Performance Configuration Options
12. Special Cases

HBase and MapReduce

1. HBase, MapReduce, and the CLASSPATH
2. MapReduce Scan Caching
3. Bundled HBase MapReduce Jobs
4. HBase as a MapReduce Job Data Source and Data Sink
5. Writing HFiles Directly During Bulk Import
6. RowCounter Example
7. Map-Task Splitting
8. HBase MapReduce Examples
9. Accessing Other HBase Tables in a MapReduce Job
10. Speculative Execution

Security

1. Web UI Security
2. Using Secure HTTP (HTTPS) for the Web UI
3. Disable cache in HBase UI
4. Using SPNEGO for Kerberos authentication with Web UIs
5. Defining administrators of the Web UI with SPNEGO
6. Using LDAP authentication with Web UIs
7. Defining Administrators of the Web UI with LDAP
8. Other UI security-related configuration
9. Secure Client Access to Apache HBase
10. Prerequisites
11. Server-side Configuration for Secure Operation
12. Client-side Configuration for Secure Operation
13. Client-side Configuration for Secure Operation - Thrift Gateway
14. Configure the Thrift Gateway to Authenticate on Behalf of the Client
15. Configure the Thrift Gateway to Use the `doAs` Feature
16. Client-side Configuration for Secure Operation - REST Gateway
17. REST Gateway Impersonation Configuration
18. Simple User Access to Apache HBase
19. Simple versus Secure Access
20. Prerequisites
21. Server-side Configuration for Simple User Access Operation
22. Client-side Configuration for Simple User Access Operation
23. Transport Level Security (TLS) in HBase RPC communication
24. Server side configuration
25. Client side configuration
26. Creating self-signed certificates
27. Upgrading existing non-TLS cluster with no downtime
28. Enable automatic certificate reloading
29. Additional configuration
30. Securing Access to HDFS and ZooKeeper
31. Securing ZooKeeper Data
32. Securing File System (HDFS) Data
33. Securing Access To Your Data
34. Tags
35. Access Control Labels (ACLs)
36. Visibility Labels
37. Transparent Encryption of Data At Rest

38. Secure Enable
39. Security Configuration Example
Architecture
1. Resources
2. Overview
3. NoSQL?
4. When Should I Use HBase?
5. What Is The Difference Between HBase and Hadoop/HDFS?
6. Catalog Tables
7. hbase:meta
8. Startup Sequencing
9. Client
10. Cluster Connections
11. WriteBuffer and Batch Methods
12. Asynchronous Client
13. Asynchronous Admin
14. External Clients
15. Master Registry (new as of 2.3.0)
16. Rpc Connection Registry (new as of 2.5.0)
17. Connection URI
18. Client Request Filters
19. Structural
20. Column Value
21. Column Value Comparators
22. KeyValue Metadata
23. RowKey
24. Utility
25. Master
26. Startup Behavior
27. Runtime Impact
28. Interface
29. Processes
30. MasterProcWAL
31. RegionServer
32. Interface
33. Processes
34. Coprocessors

35. Block Cache
36. RegionServer Splitting Implementation
37. Write Ahead Log (WAL)
38. Regions
39. Considerations for Number of Regions
40. Region-RegionServer Assignment
41. Region-RegionServer Locality
42. Region Splits
43. Manual Region Splitting
44. Online Region Merges
45. Store
46. Bulk Loading
47. Overview
48. Bulk Load Architecture
49. See Also
50. Advanced Usage
51. Bulk Loading Replication
52. HDFS
53. NameNode
54. DataNode
55. Timeline-consistent High Available Reads
56. Introduction
57. Timeline Consistency
58. Tradeoffs
59. Where is the code
60. Propagating writes to region replicas
61. Store File TTL
62. Region replication for META table's region
63. Memory accounting
64. Secondary replica failover
65. Configuration properties
66. User Interface
67. Creating a table with region replication
68. Read API and Usage
69. Storing Medium-sized Objects (MOB)
70. Configuring Columns for MOB
71. Testing MOB

72. MOB architecture
73. MOB Optimization Tasks
74. MOB Troubleshooting
75. MOB Upgrade Considerations
76. Scan Over Snapshot
77. TableSnapshotScanner
78. TableSnapshotInputFormat
79. Permission to access snapshot and data files

In-memory Compaction
1. Overview
2. Enabling

RegionServer Offheap Read/Write Path
1. Overview
2. Offheap read-path
3. Read block from HDFS to offheap directly
4. Offheap write-path

Backup and Restore
1. Overview
2. Terminology
3. Planning
4. First-time configuration steps
5. Backup and Restore commands
6. Administration of Backup Images
7. Additional Topics
8. Configuration keys
9. Best Practices
10. Scenario: Safeguarding Application Datasets on Amazon S3
11. Security of Backup Data
12. Technical Details of Incremental Backup and Restore
13. A Warning on File System Growth
14. Capacity Planning
15. Limitations of the Backup and Restore Utility

Synchronous Replication
1. Background
2. Design
3. Operation and maintenance

Apache HBase APIs

1. Examples

Apache HBase External APIs

1. REST
2. Thrift
3. C/C++ Apache HBase Client
4. Using Java Data Objects (JDO) with HBase
5. Scala
6. Jython

Thrift API and Filter Language

1. Filter Language

HBase and Spark

1. Basic Spark
2. Spark Streaming
3. Bulk Load
4. SparkSQL/DataFrames

Apache HBase Coprocessors

1. Coprocessor Overview
2. Types of Coprocessors
3. Loading Coprocessors
4. Examples
5. Guidelines For Deploying A Coprocessor
6. Restricting Coprocessor Usage

Apache HBase Performance Tuning

1. Operating System
2. Network
3. Java
4. HBase Configurations
5. ZooKeeper
6. Schema Design
7. HBase General Patterns
8. Writing to HBase
9. Reading from HBase
10. Deleting from HBase
11. HDFS
12. Amazon EC2
13. Collocating HBase and MapReduce
14. Case Studies

Profiler Servlet

1. Background
2. Prerequisites
3. Usage
4. UI
5. Notes

Troubleshooting and Debugging Apache HBase

1. General Guidelines
2. Logs
3. Log Locations
4. Log Levels
5. Handling Multiple SLF4J Bindings to Prevent Overridden Log Configurations
6. JVM Garbage Collection Logs
7. Resources
8. Tools
9. Client
10. MapReduce
11. NameNode
12. Network
13. RegionServer
14. Master
15. ZooKeeper
16. Amazon EC2
17. HBase and Hadoop version issues
18. HBase and HDFS
19. Running unit or integration tests
20. Case Studies
21. Cryptographic Features
22. Operating System Specific Issues
23. JDK Issues

Case Studies

1. Overview
2. Schema Design
3. Performance/Troubleshooting

Operational Management

1. HBase Tools and Utilities
2. Canary

3. RegionSplitter
4. Health Checker
5. Driver
6. HBase `hbck`
7. HBase `HBCK2`
8. HFile Tool
9. WAL Tools
10. Compression Tool
11. CopyTable
12. HashTable/SyncTable
13. Export
14. Import
15. ImportTsv
16. CompleteBulkLoad
17. WALPlayer
18. RowCounter
19. CellCounter
20. mlockall
21. Offline Compaction Tool
22. `hbase clean`
23. `hbase pe`
24. `hbase ltt`
25. Pre-Upgrade validator
26. Data Block Encoding Tool
27. HBase Conf Tool
28. Node Management
29. Metrics & Monitoring
30. HBase Metrics
31. HBase Monitoring
32. Cluster Replication
33. Serial Replication
34. Running Multiple Workloads on a Single Cluster
35. Quotas
36. Request Queues
37. Multiple-Typed Queues
38. Space Quotas
39. Disabling Automatic Space Quota Deletion

40. HBase Snapshots with Space Quotas
41. Backup & Snapshots
42. HBase Backup
43. HBase Snapshots
44. Storing Snapshots in an Amazon S3 Bucket
45. Storing Snapshots in Microsoft Azure Blob Storage
46. Storing Snapshots in Aliyun Object Storage Service
47. Table Rename
48. Region & Capacity Management
49. Region Management
50. Capacity Planning and Region Sizing
51. RegionServer Grouping
52. Region Normalizer
53. Auto Region Reopen

Building and Developing

1. Getting Involved
2. Mailing Lists
3. Slack
4. Internet Relay Chat (IRC)
5. Jira
6. Apache HBase Repositories
7. IDEs
8. Eclipse
9. IntelliJ IDEA
10. Other IDEs
11. Building Apache HBase
12. Basic Compile
13. Build On Linux Aarch64
14. Releasing Apache HBase
15. Making a Release Candidate
16. Voting on Release Candidates
17. Baseline Verifications for Voting Release Candidates
18. Additional Verifications for Voting Release Candidates
19. Announcing Releases
20. Generating the HBase Reference Guide
21. Updating hbase.apache.org
22. Contributing to hbase.apache.org

23. Publishing hbase.apache.org
24. Tests
25. Apache HBase Modules
26. Unit Tests
27. Running tests
28. Writing Tests
29. Integration Tests
30. Developer Guidelines
31. Branches
32. Policy for Fix Version in JIRA
33. Code Standards
34. Invariants
35. Running In-Situ
36. Adding Metrics
37. Git Best Practices
38. Submitting Patches
39. The hbase-thirdparty dependency and shading/relocation
40. Development of HBase-related Maven archetypes

Unit Testing HBase Applications
1. JUnit
2. Mockito
3. MRUnit
4. Integration Testing with an HBase Mini-Cluster

Protobuf in HBase
1. Protobuf

Procedure Framework (Pv2)
1. Procedures
2. Subprocedures
3. ProcedureExecutor
4. Nonces
5. Wait/Wake/Suspend/Yield
6. Locking
7. Procedure Types
8. References

AMv2 Description for Devs
1. Background
2. New System

3. Procedures Detail
4. UI
5. Logging
6. Implementation Notes
7. New Configs
8. Tools

ZooKeeper

1. Using existing ZooKeeper ensemble
2. SASL Authentication with ZooKeeper
3. TLS connection to ZooKeeper

Community

1. Decisions
2. Community Roles
3. Commit Message format

hbtop

1. Usage
2. Others

Tracing

1. Overview
2. Usage

Store File Tracking

1. Overview
2. Available Implementations
3. Usage

Bulk Data Generator Tool

1. Usage
2. Overview

Appendix: Contributing to Documentation

1. Contributing to Documentation
2. Contributing to Other Strings
3. Editing the HBase Website and Documentation
4. Publishing the HBase Website and Documentation
5. MDX and Fumadocs Components
6. Auto-Generated Content
7. Images in the Documentation
8. Adding a New Section to the Documentation
9. Unique Headings Requirement

10. Common Documentation Issues
FAQ
1. General
2. Upgrading
3. Architecture
4. Configuration
5. Schema Design / Data Access
6. MapReduce
7. Performance and Troubleshooting
8. Amazon EC2
9. Operations
10. HBase in Action
Access Control Matrix
1. Interpreting the ACL Matrix Table
2. ACL Matrix
Compression and Data Block Encoding In HBase
1. Changes Take Effect Upon Compaction
2. Block Compressors
3. Data Block Encoding Types
4. Which Compressor or Data Block Encoder To Use
5. Making use of Hadoop Native Libraries in HBase
6. Compressor Configuration, Installation, and Use
7. Enable Data Block Encoding
SQL over HBase
YCSB
HFile Format
1. HBase File Format (version 1)
2. HBase file format with inline blocks (version 2)
3. HBase File Format with Security Enhancements (version 3)
Other Information About HBase
1. HBase Videos
2. HBase Presentations (Slides)
3. HBase Papers
4. HBase Sites
5. HBase Books
6. Hadoop Books
HBase History

Apache Software Foundation
1. ASF Development Process
2. ASF Board Reporting
Apache HBase Orca
0.95 RPC Specification
1. Goals
2. TODO
3. RPC
4. Notes
Known Incompatibilities Among HBase Versions
1. HBase 2.0 Incompatible Changes
2. Footnotes

Preface

Herein you will find either the definitive documentation on an HBase topic as of its standing when the referenced HBase version shipped, or it will point to the location in Javadoc or JIRA where the pertinent information can be found.

About This Guide

This reference guide is a work in progress. The source for this guide can be found in the *hbase-website/app/pages/_docs/docs/_mdx/(multi-page)* directory of the HBase source. This reference guide is marked up using MDX (just extended markdown) powered by Fumadocs from which the finished guide is generated as part of the 'site' build target. Run

```
mvn site
```

to generate this documentation. Amendments and improvements to the documentation are welcomed. Click this link to file a new documentation bug against Apache HBase with some values pre-selected.

Contributing to the Documentation

For an overview and suggestions to get started contributing to the documentation, see the relevant section later in this documentation.

Heads-up if this is your first foray into the world of distributed computing...

If this is your first foray into the wonderful world of Distributed Computing, then you are in for some interesting times. First off, distributed systems are hard; making a distributed system hum requires a disparate skillset that spans systems (hardware and software) and networking.

Your cluster's operation can hiccup because of any of a myriad set of reasons from bugs in HBase itself through misconfigurations — misconfiguration of HBase but also operating system misconfigurations — through to hardware problems whether it be a bug in your

network card drivers or an underprovisioned RAM bus (to mention two recent examples of hardware issues that manifested as "HBase is slow"). You will also need to do a recalibration if up to this your computing has been bound to a single box. Here is one good starting point: Fallacies of Distributed Computing.

That said, you are welcome.
It's a fun place to be.
Yours, the HBase Community.

Reporting Bugs

Please use JIRA to report non-security-related bugs.

To protect existing HBase installations from new vulnerabilities, please **do not** use JIRA to report security-related bugs. Instead, send your report to the mailing list private@hbase.apache.org, which allows anyone to send messages, but restricts who can read them. Someone on that list will contact you to follow up on your report.

Support and Testing Expectations

The phrases *supported*, *not supported*, *tested*, and *not tested* occur several places throughout this guide. In the interest of clarity, here is a brief explanation of what is generally meant by these phrases, in the context of HBase.

> Commercial technical support for Apache HBase is provided by many Hadoop vendors. This is not the sense in which the term *support* is used in the context of the Apache HBase project. The Apache HBase team assumes no responsibility for your HBase clusters, your configuration, or your data.

Supported

In the context of Apache HBase, *supported* means that HBase is designed to work in the way described, and deviation from the defined behavior or functionality should be reported as a bug.

Not Supported

In the context of Apache HBase, *not supported* means that a use case or use pattern is not expected to work and should be considered an antipattern. If you think this designation should be reconsidered for a given feature or use pattern, file a JIRA or start a discussion on one of the mailing lists.

Tested

In the context of Apache HBase, *tested* means that a feature is covered by unit or integration tests, and has been proven to work as expected.

Not Tested

In the context of Apache HBase, *not tested* means that a feature or use pattern may or may not work in a given way, and may or may not corrupt your data or cause operational issues. It is an unknown, and there are no guarantees. If you can provide proof that a feature designated as *not tested* does work in a given way, please submit the tests and/or the metrics so that other users can gain certainty about such features or use patterns.

Getting Started

Quick Start - Standalone HBase

This section describes the setup of a single-node standalone HBase. A *standalone* instance has all HBase daemons — the Master, RegionServers, and ZooKeeper — running in a single JVM persisting to the local filesystem. It is our most basic deploy profile. We will show you how to create a table in HBase using the `hbase shell` CLI, insert rows into the table, perform put and scan operations against the table, enable or disable the table, and start and stop HBase.

Apart from downloading HBase, this procedure should take less than 10 minutes.

JDK Version Requirements

HBase requires that a JDK be installed. See Java for information about supported JDK versions.

Get Started with HBase

Procedure: Download, Configure, and Start HBase in Standalone Mode

1. Choose a download site from this list of Apache Download Mirrors. Click on the suggested top link. This will take you to a mirror of *HBase Releases*. Click on the folder named *stable* and then download the binary file that ends in *.tar.gz* to your local filesystem. Do not download the file ending in *src.tar.gz* for now.

2. Extract the downloaded file, and change to the newly-created directory.

   ```
   tar xzvf <FILE_NAME>.tar.gz
   cd <DIRECTORY_NAME>
   ```

3. You must set the `JAVA_HOME` environment variable before starting HBase. To make this easier, HBase lets you set it within the *conf/hbase-env.sh* file. You must locate

where Java is installed on your machine, and one way to find this is by using the *whereis java* command. Once you have the location, edit the *conf/hbase-env.sh* file and uncomment the line starting with *#export JAVA_HOME=*, and then set it to your Java installation path.

Example extract from hbase-env.sh where JAVA_HOME is set

```
# Set environment variables here.
# The java implementation to use.
export JAVA_HOME=/usr/jdk64/jdk1.8.0_112
```

4 The *bin/start-hbase.sh* script is provided as a convenient way to start HBase. Issue the command, and if all goes well, a message is logged to standard output showing that HBase started successfully. You can use the `jps` command to verify that you have one running process called `HMaster`. In standalone mode HBase runs all daemons within this single JVM, i.e. the HMaster, a single HRegionServer, and the ZooKeeper daemon. Go to ***http://localhost:16010*** to view the HBase Web UI.

Procedure: Use HBase For the First Time

1 **Connect to HBase**

Connect to your running instance of HBase using the `hbase shell` command, located in the *bin/* directory of your HBase install. In this example, some usage and version information that is printed when you start HBase Shell has been omitted. The HBase Shell prompt ends with a `>` character.

```
$ ./bin/hbase shell
hbase(main):001:0>
```

2 **Display HBase Shell Help Text**

Type `help` and press Enter, to display some basic usage information for HBase Shell, as well as several example commands. Notice that table names, rows, columns all must be enclosed in quote characters.

3 **Create a table**

Use the `create` command to create a new table. You must specify the table name and the ColumnFamily name.

```
hbase(main):001:0> create 'test', 'cf'
0 row(s) in 0.4170 seconds

=> Hbase::Table - test
```

4 List Information About your Table

Use the `list` command to confirm your table exists

```
hbase(main):002:0> list 'test'
TABLE
test
1 row(s) in 0.0180 seconds

=> ["test"]
```

Now use the `describe` command to see details, including configuration defaults

```
hbase(main):003:0> describe 'test'
Table test is ENABLED
test
COLUMN FAMILIES DESCRIPTION
{NAME => 'cf', VERSIONS => '1', EVICT_BLOCKS_ON_CLOSE => 'false', NEW_VERSIO
N_BEHAVIOR => 'false', KEEP_DELETED_CELLS => 'FALSE', CACHE_DATA_ON_WRITE =>
'false', DATA_BLOCK_ENCODING => 'NONE', TTL => 'FOREVER', MIN_VERSIONS =>
'0', REPLICATION_SCOPE => '0', BLOOMFILTER => 'ROW', CACHE_INDEX_ON_WRITE =>
'f
alse', IN_MEMORY => 'false', CACHE_BLOOMS_ON_WRITE => 'false', PREFETCH_BLOC
KS_ON_OPEN => 'false', COMPRESSION => 'NONE', BLOCKCACHE => 'true', BLOCKSIZ
E
 => '65536'}
1 row(s)
Took 0.9998 seconds
```

5 Put data into your table

To put data into your table, use the `put` command.

```
hbase(main):003:0> put 'test', 'row1', 'cf:a', 'value1'
0 row(s) in 0.0850 seconds

hbase(main):004:0> put 'test', 'row2', 'cf:b', 'value2'
0 row(s) in 0.0110 seconds

hbase(main):005:0> put 'test', 'row3', 'cf:c', 'value3'
0 row(s) in 0.0100 seconds
```

Here, we insert three values, one at a time. The first insert is at `row1`, column `cf:a`, with a value of `value1`. Columns in HBase are comprised of a column family prefix, `cf` in this example, followed by a colon and then a column qualifier suffix, `a` in this case.

6 **Scan the table for all data at once**

One of the ways to get data from HBase is to scan. Use the `scan` command to scan the table for data. You can limit your scan, but for now, all data is fetched.

```
hbase(main):006:0> scan 'test'
ROW                                   COLUMN+CELL
 row1                                 column=cf:a, timestamp=142176248576
8, value=value1
 row2                                 column=cf:b, timestamp=142176249178
5, value=value2
 row3                                 column=cf:c, timestamp=142176249621
0, value=value3
3 row(s) in 0.0230 seconds
```

7 **Get a single row of data**

To get a single row of data at a time, use the `get` command.

```
hbase(main):007:0> get 'test', 'row1'
COLUMN                                CELL
 cf:a                                 timestamp=1421762485768, value=valu
e1
1 row(s) in 0.0350 seconds
```

8 **Disable a table**

If you want to delete a table or change its settings, as well as in some other situations, you need to disable the table first, using the `disable` command. You can re-enable it using the `enable` command.

```
hbase(main):008:0> disable 'test'
0 row(s) in 1.1820 seconds

hbase(main):009:0> enable 'test'
0 row(s) in 0.1770 seconds
```

Disable the table again if you tested the `enable` command above:

```
hbase(main):010:0> disable 'test'
0 row(s) in 1.1820 seconds
```

9 **Drop the table**

To drop (delete) a table, use the `drop` command.

```
hbase(main):011:0> drop 'test'
0 row(s) in 0.1370 seconds
```

10 **Exit the HBase Shell**

To exit the HBase Shell and disconnect from your cluster, use the `quit` command. HBase is still running in the background.

Procedure: Stop HBase

1 In the same way that the *bin/start-hbase.sh* script is provided to conveniently start all HBase daemons, the *bin/stop-hbase.sh* script stops them.

```
$ ./bin/stop-hbase.sh
stopping hbase....................
```

2 After issuing the command, it can take several minutes for the processes to shut down. Use the `jps` to be sure that the HMaster and HRegionServer processes are shut down.

The above has shown you how to start and stop a standalone instance of HBase. In the next sections we give a quick overview of other modes of hbase deploy.

Pseudo-Distributed for Local Testing

After working your way through quickstart standalone mode, you can re-configure HBase to run in pseudo-distributed mode. Pseudo-distributed mode means that HBase still runs completely on a single host, but each HBase daemon (HMaster, HRegionServer, and ZooKeeper) runs as a separate process: in standalone mode all daemons ran in one jvm process/instance. By default, unless you configure the `hbase.rootdir` property as described in quickstart, your data is still stored in */tmp/*. In this walk-through, we store

your data in HDFS instead, assuming you have HDFS available. You can skip the HDFS configuration to continue storing your data in the local filesystem.

> This procedure assumes that you have configured Hadoop and HDFS on your local system and/or a remote system, and that they are running and available. It also assumes you are using Hadoop 2. The guide on Setting up a Single Node Cluster in the Hadoop documentation is a good starting point.

1. **Stop HBase if it is running**

   If you have just finished quickstart and HBase is still running, stop it. This procedure will create a totally new directory where HBase will store its data, so any databases you created before will be lost.

2. **Configure HBase**

   Edit the *hbase-site.xml* configuration. First, add the following property which directs HBase to run in distributed mode, with one JVM instance per daemon.

   ```
   <property>
     <name>hbase.cluster.distributed</name>
     <value>true</value>
   </property>
   ```

   Next, add a configuration for `hbase.rootdir`, pointing to the address of your HDFS instance, using the `hdfs:////` URI syntax. In this example, HDFS is running on the localhost at port 8020.

   ```
   <property>
     <name>hbase.rootdir</name>
     <value>hdfs://localhost:9000/hbase</value>
   </property>
   ```

   You do not need to create the directory in HDFS. HBase will do this for you. If you create the directory, HBase will attempt to do a migration, which is not what you want.

   Finally, remove existing configuration for `hbase.tmp.dir` and `hbase.unsafe.stream.capability.enforce`.

3 **Start HBase**

Use the *bin/start-hbase.sh* command to start HBase. If your system is configured correctly, the `jps` command should show the HMaster and HRegionServer processes running.

4 **Check the HBase directory in HDFS**

If everything worked correctly, HBase created its directory in HDFS. In the configuration above, it is stored in */hbase/* on HDFS. You can use the `hadoop fs` command in Hadoop's *bin/* directory to list this directory.

```
$ ./bin/hadoop fs -ls /hbase
Found 7 items
drwxr-xr-x   - hbase users          0 2014-06-25 18:58 /hbase/.tmp
drwxr-xr-x   - hbase users          0 2014-06-25 21:49 /hbase/WALs
drwxr-xr-x   - hbase users          0 2014-06-25 18:48 /hbase/corrupt
drwxr-xr-x   - hbase users          0 2014-06-25 18:58 /hbase/data
-rw-r--r--   3 hbase users         42 2014-06-25 18:41 /hbase/hbase.id
-rw-r--r--   3 hbase users          7 2014-06-25 18:41 /hbase/hbase.version
drwxr-xr-x   - hbase users          0 2014-06-25 21:49 /hbase/oldWALs
```

5 **Create a table and populate it with data**

You can use the HBase Shell to create a table, populate it with data, scan and get values from it, using the same procedure as in shell exercises.

6 **Start and stop a backup HBase Master (HMaster) server**

> Running multiple HMaster instances on the same hardware does not make sense in a production environment, in the same way that running a pseudo-distributed cluster does not make sense for production. This step is offered for testing and learning purposes only.

The HMaster server controls the HBase cluster. You can start up to 9 backup HMaster servers, which makes 10 total HMasters, counting the primary. To start a backup HMaster, use the `local-master-backup.sh`. For each backup master you want to start, add a parameter representing the port offset for that master. Each HMaster uses two ports (16000 and 16010 by default). The port offset is added to these ports, so using an offset of 2, the backup HMaster would use ports 16002 and 16012. The following command starts 3 backup servers using ports 16002/16012, 16003/16013, and 16005/16015.

```
./bin/local-master-backup.sh start 2 3 5
```

To kill a backup master without killing the entire cluster, you need to find its process ID (PID). The PID is stored in a file with a name like */tmp/hbase-USER-X-master.pid*. The only contents of the file is the PID. You can use the `kill -9` command to kill that PID. The following command will kill the master with port offset 1, but leave the cluster running:

```
cat /tmp/hbase-testuser-1-master.pid |xargs kill -9
```

7 **Start and stop additional RegionServers**

The HRegionServer manages the data in its StoreFiles as directed by the HMaster. Generally, one HRegionServer runs per node in the cluster. Running multiple HRegionServers on the same system can be useful for testing in pseudo-distributed mode. The `local-regionservers.sh` command allows you to run multiple RegionServers. It works in a similar way to the `local-master-backup.sh` command, in that each parameter you provide represents the port offset for an instance. Each RegionServer requires two ports, and the default ports are 16020 and 16030. Since HBase version 1.1.0, HMaster doesn't use region server ports, this leaves 10 ports (16020 to 16029 and 16030 to 16039) to be used for RegionServers. For supporting additional RegionServers, set environment variables HBASE_RS_BASE_PORT and HBASE_RS_INFO_BASE_PORT to appropriate values before running script `local-regionservers.sh`. e.g. With values 16200 and 16300 for base ports, 99 additional RegionServers can be supported, on a server. The following command starts four additional RegionServers, running on sequential ports starting at 16022/16032 (base ports 16020/16030 plus 2).

```
./bin/local-regionservers.sh start 2 3 4 5
```

To stop a RegionServer manually, use the `local-regionservers.sh` command with the `stop` parameter and the offset of the server to stop.

```
./bin/local-regionservers.sh stop 3
```

8 **Stop HBase**

You can stop HBase the same way as in the quickstart procedure, using the *bin/stop-hbase.sh* command.

Fully Distributed for Production

In reality, you need a fully-distributed configuration to fully test HBase and to use it in real-world scenarios. In a distributed configuration, the cluster contains multiple nodes, each of which runs one or more HBase daemon. These include primary and backup Master instances, multiple ZooKeeper nodes, and multiple RegionServer nodes.

This advanced quickstart adds two more nodes to your cluster. The architecture will be as follows:

Node Name	Master	ZooKeeper	RegionServer
node-a.example.com	yes	yes	no
node-b.example.com	backup	yes	yes
node-c.example.com	no	yes	yes

Distributed Cluster Demo Architecture

This quickstart assumes that each node is a virtual machine and that they are all on the same network. It builds upon the previous quickstart, Pseudo-Distributed for Local Testing, assuming that the system you configured in that procedure is now `node-a`. Stop HBase on `node-a` before continuing.

> Be sure that all the nodes have full access to communicate, and that no firewall rules are in place which could prevent them from talking to each other. If you see any errors like `no route to host`, check your firewall.

Procedure: Configure Passwordless SSH Access

`node-a` needs to be able to log into `node-b` and `node-c` (and to itself) in order to start the daemons. The easiest way to accomplish this is to use the same username on all hosts, and configure password-less SSH login from `node-a` to each of the others.

1. **On `node-a`, generate a key pair**

   While logged in as the user who will run HBase, generate a SSH key pair, using the following command:

   ```
   ssh-keygen -t rsa
   ```

   If the command succeeds, the location of the key pair is printed to standard output. The default name of the public key is *id_rsa.pub*.

2. **Create the directory that will hold the shared keys on the other nodes**

   On `node-b` and `node-c`, log in as the HBase user and create a *.ssh/* directory in the user's home directory, if it does not already exist. If it already exists, be aware that it may already contain other keys.

3. **Copy the public key to the other nodes**

   Securely copy the public key from `node-a` to each of the nodes, by using the `scp` or some other secure means. On each of the other nodes, create a new file called *.ssh/authorized_keys if it does not already exist*, and append the contents of the *id_rsa.pub* file to the end of it. Note that you also need to do this for `node-a` itself.

   ```
   cat id_rsa.pub >> ~/.ssh/authorized_keys
   ```

4. **Test password-less login**

   If you performed the procedure correctly, you should not be prompted for a password when you SSH from `node-a` to either of the other nodes using the same username.

5. Since `node-b` will run a backup Master, repeat the procedure above, substituting `node-b` everywhere you see `node-a`. Be sure not to overwrite your existing *.ssh/authorized_keys* files, but concatenate the new key onto the existing file using the `>>` operator rather than the `>` operator.

Procedure: Prepare `node-a`

`node-a` will run your primary master and ZooKeeper processes, but no RegionServers. Stop the RegionServer from starting on `node-a`.

1. Edit *conf/regionservers* and remove the line which contains `localhost`. Add lines with the hostnames or IP addresses for `node-b` and `node-c`.

   Even if you did want to run a RegionServer on `node-a`, you should refer to it by the hostname the other servers would use to communicate with it. In this case, that would be `node-a.example.com`. This enables you to distribute the configuration to each node of your cluster any hostname conflicts. Save the file.

2. **Configure HBase to use `node-b` as a backup master**

   Create a new file in *conf/* called *backup-masters*, and add a new line to it with the hostname for `node-b`. In this demonstration, the hostname is `node-b.example.com`.

3. **Configure ZooKeeper**

   In reality, you should carefully consider your ZooKeeper configuration. You can find out more about configuring ZooKeeper in **zookeeper** section. This configuration will direct HBase to start and manage a ZooKeeper instance on each node of the cluster.

   On `node-a`, edit *conf/hbase-site.xml* and add the following properties.

   ```
   <property>
   <name>hbase.zookeeper.quorum</name>
   <value>node-a.example.com,node-b.example.com,node-c.example.com</value>
   </property>
   <property>
   <name>hbase.zookeeper.property.dataDir</name>
   <value>/usr/local/zookeeper</value>
   </property>
   ```

4. Everywhere in your configuration that you have referred to `node-a` as `localhost`, change the reference to point to the hostname that the other nodes will use to refer to `node-a`. In these examples, the hostname is `node-a.example.com`.

Procedure: Prepare `node-b` and `node-c`

`node-b` will run a backup master server and a ZooKeeper instance.

1. **Download and unpack HBase**

   Download and unpack HBase to `node-b`, just as you did for the standalone and pseudo-distributed quickstarts.

2 **Copy the configuration files from `node-a` to `node-b`.and `node-c`**

Each node of your cluster needs to have the same configuration information. Copy the contents of the *conf/* directory to the *conf/* directory on `node-b` and `node-c`.

Procedure: Start and Test Your Cluster

1 **Be sure HBase is not running on any node**

If you forgot to stop HBase from previous testing, you will have errors. Check to see whether HBase is running on any of your nodes by using the `jps` command. Look for the processes `HMaster`, `HRegionServer`, and `HQuorumPeer`. If they exist, kill them.

2 **Start the cluster**

On `node-a`, issue the `start-hbase.sh` command. Your output will be similar to that below.

```
$ bin/start-hbase.sh
node-c.example.com: starting zookeeper, logging to /home/hbuser/hbase-0.98.3
-hadoop2/bin/../logs/hbase-hbuser-zookeeper-node-c.example.com.out
node-a.example.com: starting zookeeper, logging to /home/hbuser/hbase-0.98.3
-hadoop2/bin/../logs/hbase-hbuser-zookeeper-node-a.example.com.out
node-b.example.com: starting zookeeper, logging to /home/hbuser/hbase-0.98.3
-hadoop2/bin/../logs/hbase-hbuser-zookeeper-node-b.example.com.out
starting master, logging to /home/hbuser/hbase-0.98.3-hadoop2/bin/../logs/hb
ase-hbuser-master-node-a.example.com.out
node-c.example.com: starting regionserver, logging to /home/hbuser/hbase-0.9
8.3-hadoop2/bin/../logs/hbase-hbuser-regionserver-node-c.example.com.out
node-b.example.com: starting regionserver, logging to /home/hbuser/hbase-0.9
8.3-hadoop2/bin/../logs/hbase-hbuser-regionserver-node-b.example.com.out
node-b.example.com: starting master, logging to /home/hbuser/hbase-0.98.3-ha
doop2/bin/../logs/hbase-hbuser-master-nodeb.example.com.out
```

ZooKeeper starts first, followed by the master, then the RegionServers, and finally the backup masters.

3 **Verify that the processes are running**

On each node of the cluster, run the `jps` command and verify that the correct processes are running on each server. You may see additional Java processes running on your servers as well, if they are used for other purposes.

`node-a` `jps` Output

```
$ jps
20355 Jps
20071 HQuorumPeer
20137 HMaster
```

`node-b` `jps` Output

```
$ jps
15930 HRegionServer
16194 Jps
15838 HQuorumPeer
16010 HMaster
```

`node-c` `jps` Output

```
$ jps
13901 Jps
13639 HQuorumPeer
13737 HRegionServer
```

> The `HQuorumPeer` process is a ZooKeeper instance which is controlled and started by HBase. If you use ZooKeeper this way, it is limited to one instance per cluster node and is appropriate for testing only. If ZooKeeper is run outside of HBase, the process is called `QuorumPeer`. For more about ZooKeeper configuration, including using an external ZooKeeper instance with HBase, see zookeeper section.

4 **Browse to the Web UI**

If everything is set up correctly, you should be able to connect to the UI for the Master `http://node-a.example.com:16010/` or the secondary master at `http://node-b.example.com:16010/` using a web browser. If you can connect via `localhost` but not from another host, check your firewall rules. You can see the web UI for each of the RegionServers at port 16030 of their IP addresses, or by clicking their links in the web UI for the Master.

5 **Test what happens when nodes or services disappear**

With a three-node cluster you have configured, things will not be very resilient. You can still test the behavior of the primary Master or a RegionServer by killing the associated processes and watching the logs.

Where to go next

The next chapter, configuration, gives more information about the different HBase run modes, system requirements for running HBase, and critical configuration areas for setting up a distributed HBase cluster.

Configuration

This chapter expands upon the Getting Started chapter to further explain configuration of Apache HBase. Please read this chapter carefully, especially the Basic Prerequisites to ensure that your HBase testing and deployment goes smoothly. Familiarize yourself with Support and Testing Expectations as well.

Configuration Files

Apache HBase uses the same configuration system as Apache Hadoop. All configuration files are located in the *conf/* directory, which needs to be kept in sync for each node on your cluster.

backup-masters
Not present by default. A plain-text file which lists hosts on which the Master should start a backup Master process, one host per line.

hadoop-metrics2-hbase.properties
Used to connect HBase Hadoop's Metrics2 framework. See the Hadoop Wiki entry for more information on Metrics2. Contains only commented-out examples by default.

hbase-env.cmd and *hbase-env.sh*
Script for Windows and Linux / Unix environments to set up the working environment for HBase, including the location of Java, Java options, and other environment variables. The file contains many commented-out examples to provide guidance.

hbase-policy.xml
The default policy configuration file used by RPC servers to make authorization decisions on client requests. Only used if HBase security is enabled.

hbase-site.xml
The main HBase configuration file. This file specifies configuration options which override HBase's default configuration. You can view (but do not edit) the default configuration file at *hbase-common/src/main/resources/hbase-default.xml*. You can also view the entire effective configuration for your cluster (defaults and overrides) in the **HBase Configuration** tab of the HBase Web UI.

log4j2.properties
Configuration file for HBase logging via `log4j2`.

regionservers
A plain-text file containing a list of hosts which should run a RegionServer in your HBase cluster. By default, this file contains the single entry `localhost`. It should contain a list of hostnames or IP addresses, one per line, and should only contain `localhost` if each node in your cluster will run a RegionServer on its `localhost` interface.

> When you edit XML, it is a good idea to use an XML-aware editor to be sure that your syntax is correct and your XML is well-formed. You can also use the `xmllint` utility to check that your XML is well-formed. By default, `xmllint` re-flows and prints the XML to standard output. To check for well-formedness and only print output if errors exist, use the command `xmllint -noout filename.xml`.

> When running in distributed mode, after you make an edit to an HBase configuration, make sure you copy the contents of the *conf/* directory to all nodes of the cluster. HBase will not do this for you. Use a configuration management tool for managing and copying the configuration files to your nodes. For most configurations, a restart is needed for servers to pick up changes. Dynamic configuration is an exception to this, to be described later below.

Basic Prerequisites

Java

HBase runs on the Java Virtual Machine, thus all HBase deployments require a JVM runtime.

The following table summarizes the recommendations of the HBase community with respect to running on various Java versions. The ✅ symbol indicates a base level of testing and willingness to help diagnose and address issues you might run into; these are the expected deployment combinations. An entry of ⚠️ means that there may be challenges with this combination, and you should look for more information before deciding to pursue this as your deployment strategy. The ❌ means this combination does not work; either an older Java version is considered deprecated by the HBase community, or this combination is known to not work. For combinations of newer JDK with older HBase releases, it's likely there are known compatibility issues that cannot be addressed under our compatibility guarantees, making the combination impossible. In some cases,

specific guidance on limitations (e.g. whether compiling / unit tests work, specific operational issues, etc) are also noted. Assume any combination not listed here is considered ❌.

⚠️ HBase recommends downstream users rely only on JDK releases that are marked as Long-Term Supported (LTS), either from the OpenJDK project or vendors. At the time of this writing, the following JDK releases are NOT LTS releases and are NOT tested or advocated for use by the Apache HBase community: JDK9, JDK10, JDK12, JDK13, and JDK14. Community discussion around this decision is recorded on HBASE-20264.

ℹ️ At this time, all testing performed by the Apache HBase project runs on the HotSpot variant of the JVM. When selecting your JDK distribution, please take this into consideration.

Java support by release line

HBase Version	JDK 6	JDK 7	JDK 8	JDK 11	JDK 17
HBase 2.6	❌	❌	✅	✅	✅
HBase 2.5	❌	❌	✅	✅	⚠️*
HBase 2.4	❌	❌	✅	✅	❌
HBase 2.3	❌	❌	✅	⚠️*	❌
HBase 2.0-2.2	❌	❌	✅	❌	❌
HBase 1.2+	❌	✅	✅	❌	❌
HBase 1.0-1.1	❌	✅	⚠️	❌	❌
HBase 0.98	✅	✅	⚠️	❌	❌
HBase 0.94	✅	✅	❌	❌	❌

⚠️ Preliminary support for JDK11 is introduced with HBase 2.3.0, and for JDK17 is introduced with HBase 2.5.x. We will compile and run test suites with JDK11/17 in pre commit checks and nightly checks. We will mark the support as ✅ as long as we have run some ITs with the JDK version and also there are users in the community use the JDK version in real production clusters.

For JDK11/JDK17 support in HBase, please refer to HBASE-22972 and HBASE-26038

For JDK11/JDK17 support in Hadoop, which may also affect HBase, please refer to HADOOP-15338 and HADOOP-17177

> You must set `JAVA_HOME` on each node of your cluster. *hbase-env.sh* provides a handy mechanism to do this.

Operating System Utilities

ssh

HBase uses the Secure Shell (ssh) command and utilities extensively to communicate between cluster nodes. Each server in the cluster must be running `ssh` so that the Hadoop and HBase daemons can be managed. You must be able to connect to all nodes via SSH, including the local node, from the Master as well as any backup Master, using a shared key rather than a password. You can see the basic methodology for such a set-up in Linux or Unix systems at "Procedure: Configure Passwordless SSH Access" chapter. If your cluster nodes use OS X, see the section, SSH: Setting up Remote Desktop and Enabling Self-Login on the Hadoop wiki.

DNS

HBase uses the local hostname to self-report its IP address.

NTP

The clocks on cluster nodes should be synchronized. A small amount of variation is acceptable, but larger amounts of skew can cause erratic and unexpected behavior. Time synchronization is one of the first things to check if you see unexplained problems in your cluster. It is recommended that you run a Network Time Protocol (NTP) service, or another time-synchronization mechanism on your cluster and that all nodes look to the same service for time synchronization. See the Basic NTP Configuration at *The Linux Documentation Project (TLDP)* to set up NTP.

Limits on Number of Files and Processes (ulimit)

Apache HBase is a database. It requires the ability to open a large number of files at once. Many Linux distributions limit the number of files a single user is allowed to open to `1024` (or `256` on older versions of OS X). You can check this limit on your servers by running the command `ulimit -n` when logged in as the user which runs HBase. See the Troubleshooting section for some of the problems you may experience if the limit is too low. You may also notice errors such as the following:

```
2010-04-06 03:04:37,542 INFO org.apache.hadoop.hdfs.DFSClient: Exception increate
BlockOutputStream java.io.EOFException
```

```
2010-04-06 03:04:37,542 INFO org.apache.hadoop.hdfs.DFSClient: Abandoning block b
lk_-6935524980745310745_1391901
```

It is recommended to raise the ulimit to at least 10,000, but more likely 10,240, because the value is usually expressed in multiples of 1024. Each ColumnFamily has at least one StoreFile, and possibly more than six StoreFiles if the region is under load. The number of open files required depends upon the number of ColumnFamilies and the number of regions. The following is a rough formula for calculating the potential number of open files on a RegionServer.

Calculate the Potential Number of Open Files:

```
(StoreFiles per ColumnFamily) x (regions per RegionServer)
```

For example, assuming that a schema had 3 ColumnFamilies per region with an average of 3 StoreFiles per ColumnFamily, and there are 100 regions per RegionServer, the JVM will open `3 * 3 * 100 = 900` file descriptors, not counting open JAR files, configuration files, and others. Opening a file does not take many resources, and the risk of allowing a user to open too many files is minimal.

Another related setting is the number of processes a user is allowed to run at once. In Linux and Unix, the number of processes is set using the `ulimit -u` command. This should not be confused with the `nproc` command, which controls the number of CPUs available to a given user. Under load, a `ulimit -u` that is too low can cause OutOfMemoryError exceptions.

Configuring the maximum number of file descriptors and processes for the user who is running the HBase process is an operating system configuration, rather than an HBase configuration. It is also important to be sure that the settings are changed for the user that actually runs HBase. To see which user started HBase, and that user's ulimit configuration, look at the first line of the HBase log for that instance.

Example: `ulimit` Settings on Ubuntu

To configure ulimit settings on Ubuntu, edit */etc/security/limits.conf*, which is a space-delimited file with four columns. Refer to the man page for *limits.conf* for details about the format of this file. In the following example, the first line sets both soft and hard limits for the number of open files (nofile) to 32768 for the operating system user with the username hadoop. The second line sets the number of processes to 32000 for the same user.

```
hadoop  -       nofile  32768
hadoop  -       nproc   32000
```

The settings are only applied if the Pluggable Authentication Module (PAM) environment is directed to use them. To configure PAM to use these limits, be sure that the */etc/pam.d/common-session* file contains the following line:

```
session required  pam_limits.so
```

Linux Shell

All of the shell scripts that come with HBase rely on the GNU Bash shell.

Windows

Running production systems on Windows machines is not recommended.

Hadoop

The following table summarizes the versions of Hadoop supported with each version of HBase. Older versions not appearing in this table are considered unsupported and likely missing necessary features, while newer versions are untested but may be suitable.

Based on the version of HBase, you should select the most appropriate version of Hadoop. You can use Apache Hadoop, or a vendor's distribution of Hadoop. No distinction is made here. See the Hadoop wiki for information about vendors of Hadoop.

> Comparing to Hadoop 1.x, Hadoop 2.x is faster and includes features, such as short-circuit reads (see Leveraging local data), which will help improve your HBase random read profile. Hadoop 2.x also includes important bug fixes that will improve your overall HBase experience. HBase does not support running with earlier versions of Hadoop. See the table below for requirements specific to different HBase versions.
>
> Today, Hadoop 3.x is recommended as the last Hadoop 2.x release 2.10.2 was released years ago, and there is no release for Hadoop 2.x for a very long time, although the Hadoop community does not officially EOL Hadoop 2.x yet.

Use the following legend to interpret these tables:

- ✅ = Tested to be fully-functional

- ❌ = Known to not be fully-functional, or there are CVEs so we drop the support in newer minor releases
- ⚠️ = Not tested, may/may-not function

	HBase-2.5.x	HBase-2.6.x
Hadoop-2.10.[0-1]	❌	❌
Hadoop-2.10.2+	✅	✅
Hadoop-3.1.0	❌	❌
Hadoop-3.1.1+	❌	❌
Hadoop-3.2.[0-2]	❌	❌
Hadoop-3.2.3+	✅	❌
Hadoop-3.3.[0-1]	❌	❌
Hadoop-3.3.[2-4]	✅	❌
Hadoop-3.3.5+	✅	✅
Hadoop-3.4.0+	✅ (2.5.11+)	✅ (2.6.2+)

Hadoop version support matrix for active release lines

	HBase-2.3.x	HBase-2.4.x
Hadoop-2.10.x	✅	✅
Hadoop-3.1.0	❌	❌
Hadoop-3.1.1+	✅	✅
Hadoop-3.2.x	✅	✅
Hadoop-3.3.x	✅	✅

Hadoop version support matrix for EOM 2.3+ release lines

	HBase-2.0.x	HBase-2.1.x	HBase-2.2.x
Hadoop-2.6.1+	✅	❌	❌
Hadoop-2.7.[0-6]	❌	❌	❌
Hadoop-2.7.7+	✅	✅	❌

	HBase-2.0.x	HBase-2.1.x	HBase-2.2.x
Hadoop-2.8.[0-2]	❌	❌	❌
Hadoop-2.8.[3-4]	✅	✅	❌
Hadoop-2.8.5+	✅	✅	✅
Hadoop-2.9.[0-1]	⚠️	❌	❌
Hadoop-2.9.2+	⚠️	⚠️	✅
Hadoop-3.0.[0-2]	❌	❌	❌
Hadoop-3.0.3+	❌	✅	❌
Hadoop-3.1.0	❌	❌	❌
Hadoop-3.1.1+	❌	✅	✅

Hadoop version support matrix for EOM 2.x release lines

	HBase-1.5.x	HBase-1.6.x	HBase-1.7.x
Hadoop-2.7.7+	✅	❌	❌
Hadoop-2.8.[0-4]	❌	❌	❌
Hadoop-2.8.5+	✅	✅	✅
Hadoop-2.9.[0-1]	❌	❌	❌
Hadoop-2.9.2+	✅	✅	✅
Hadoop-2.10.x	⚠️	✅	✅

Hadoop version support matrix for EOM 1.5+ release lines

	HBase-1.0.x (Hadoop 1.x is NOT supported)	HBase-1.1.x	HBase-1.2.x	HBase-1.3.x	HBase-1.4.x
Hadoop-2.4.x	✅	✅	✅	✅	❌
Hadoop-2.5.x	✅	✅	✅	✅	❌
Hadoop-2.6.0	❌	❌	❌	❌	❌

	HBase-1.0.x (Hadoop 1.x is NOT supported)	HBase-1.1.x	HBase-1.2.x	HBase-1.3.x	HBase-1.4.x
Hadoop-2.6.1+	⚠️	⚠️	✅	✅	❌
Hadoop-2.7.0	❌	❌	❌	❌	❌
Hadoop-2.7.1+	⚠️	⚠️	✅	✅	✅

Hadoop version support matrix for EOM 1.x release lines

	HBase-0.92.x	HBase-0.94.x	HBase-0.96.x	HBase-0.98.x (Support for Hadoop 1.1+ is deprecated.)
Hadoop-0.20.205	✅	❌	❌	❌
Hadoop-0.22.x	✅	❌	❌	❌
Hadoop-1.0.x	❌	❌	❌	❌
Hadoop-1.1.x	⚠️	✅	✅	⚠️
Hadoop-0.23.x	❌	✅	⚠️	❌
Hadoop-2.0.x-alpha	❌	⚠️	❌	❌
Hadoop-2.1.0-beta	❌	⚠️	✅	❌
Hadoop-2.2.0	❌	⚠️	✅	✅
Hadoop-2.3.x	❌	⚠️	✅	✅
Hadoop-2.4.x	❌	⚠️	✅	✅
Hadoop-2.5.x	❌	⚠️	✅	✅

Hadoop version support matrix for EOM pre-1.0 release lines

Starting around the time of Hadoop version 2.7.0, the Hadoop PMC got into the habit of calling out new minor releases on their major version 2 release line as not stable / production ready. As such, HBase expressly advises downstream users to avoid running on top of these releases. Note that additionally the 2.8.1 release was given the same caveat by the Hadoop PMC. For reference, see the release announcements for Apache Hadoop 2.7.0, Apache Hadoop 2.8.0, Apache Hadoop 2.8.1, and Apache Hadoop 2.9.0.

The Hadoop PMC called out the 3.1.0 release as not stable / production ready. As such, HBase expressly advises downstream users to avoid running on top of this release. For reference, see the release announcement for Hadoop 3.1.0.

Because HBase depends on Hadoop, it bundles Hadoop jars under its *lib* directory. The bundled jars are ONLY for use in stand-alone mode. In distributed mode, it is *critical* that the version of Hadoop that is out on your cluster match what is under HBase. Replace the hadoop jars found in the HBase lib directory with the equivalent hadoop jars from the version you are running on your cluster to avoid version mismatch issues. Make sure you replace the jars under HBase across your whole cluster. Hadoop version mismatch issues have various manifestations. Check for mismatch if HBase appears hung.

Hadoop 3 Support for the HBase Binary Releases and Maven Artifacts

For HBase 2.5.1 and earlier, the official HBase binary releases and Maven artifacts were built with Hadoop 2.x.

Starting with HBase 2.5.2, HBase provides binary releases and Maven artifacts built with both Hadoop 2.x and Hadoop 3.x. The Hadoop 2 artifacts do not have any version suffix, the Hadoop 3 artifacts add the `-hadoop-3` suffix to the version. i.e. `hbase-2.5.2-bin.tar.gz.asc` is the Binary release built with Hadoop2, and `hbase-2.5.2-hadoop3-bin.tar.gz` is the release built with Hadoop 3.

Hadoop 3 version policy

Each HBase release has a default Hadoop 3 version. This is used when the Hadoop 3 version is not specified during build, and for building the official binary releases and artifacts. Generally when a new minor version is released (i.e. 2.5.0) the default version is set to the latest supported Hadoop 3 version at the start of the release process.

Up to HBase 2.5.10 and 2.6.1 even if HBase added support for newer Hadoop 3 releases in a patch release, the default Hadoop 3 version (and the one used in the official binary

releases) was not updated. This simplified upgrading, but meant that HBase releases often included old unfixed CVEs both from Hadoop and Hadoop's dependencies, even when newer Hadoop releases with fixes were available.

Starting with HBase 2.5.11 and 2.6.2, the default Hadoop 3 version is always set to the latest supported Hadoop 3 version, and is also used for the `-hadoop3` binary releases and artifacts. This will drastically reduce the number of known CVEs shipped in the HBase binary releases, and make sure that all fixes and improvements in Hadoop are included.

`dfs.datanode.max.transfer.threads`

An HDFS DataNode has an upper bound on the number of files that it will serve at any one time. Before doing any loading, make sure you have configured Hadoop's *conf/hdfs-site.xml*, setting the `dfs.datanode.max.transfer.threads` value to at least the following:

```
<property>
  <name>dfs.datanode.max.transfer.threads</name>
  <value>4096</value>
</property>
```

Be sure to restart your HDFS after making the above configuration.

Not having this configuration in place makes for strange-looking failures. One manifestation is a complaint about missing blocks. For example:

```
10/12/08 20:10:31 INFO hdfs.DFSClient: Could not obtain block
          blk_XXXXXXXXXXXXXXXXXXXXXX_YYYYYYYY from any node: java.io.IOException:
No live nodes
          contain current block. Will get new block locations from namenode and r
etry...
```

See also Case Studies and note that this property was previously known as `dfs.datanode.max.xcievers` (e.g. Hadoop HDFS: Deceived by Xciever).

ZooKeeper Requirements

An Apache ZooKeeper quorum is required. The exact version depends on your version of HBase, though the minimum ZooKeeper version is 3.4.x due to the `useMulti` feature made default in 1.0.0 (see HBASE-16598).

HBase run modes: Standalone and Distributed

HBase has two run modes: **standalone** and **distributed**. Out of the box, HBase runs in standalone mode. Whatever your mode, you will need to configure HBase by editing files in the HBase *conf* directory. At a minimum, you must edit `conf/hbase-env.sh` to tell HBase which java to use. In this file you set HBase environment variables such as the heapsize and other options for the `JVM`, the preferred location for log files, etc. Set `JAVA_HOME` to point at the root of your java install.

Standalone HBase

This is the default mode. Standalone mode is what is described in the **quickstart** section. In standalone mode, HBase does not use HDFS — it uses the local filesystem instead — and it runs all HBase daemons and a local ZooKeeper all up in the same JVM. ZooKeeper binds to a well-known port so clients may talk to HBase.

Standalone HBase over HDFS

A sometimes useful variation on standalone hbase has all daemons running inside the one JVM but rather than persist to the local filesystem, instead they persist to an HDFS instance.

You might consider this profile when you are intent on a simple deploy profile, the loading is light, but the data must persist across node comings and goings. Writing to HDFS where data is replicated ensures the latter.

To configure this standalone variant, edit your *hbase-site.xml* setting *hbase.rootdir* to point at a directory in your HDFS instance but then set *hbase.cluster.distributed* to *false*. For example:

```
<configuration>
  <property>
    <name>hbase.rootdir</name>
    <value>hdfs://namenode.example.org:9000/hbase</value>
  </property>
  <property>
    <name>hbase.cluster.distributed</name>
    <value>false</value>
  </property>
```

```
</configuration>
```

Distributed

Distributed mode can be subdivided into distributed but all daemons run on a single node — a.k.a. *pseudo-distributed* — and *fully-distributed* where the daemons are spread across all nodes in the cluster. The *pseudo-distributed* vs. *fully-distributed* nomenclature comes from Hadoop.

Pseudo-distributed mode can run against the local filesystem or it can run against an instance of the *Hadoop Distributed File System* (HDFS). Fully-distributed mode can ONLY run on HDFS. See the Hadoop documentation for how to set up HDFS. A good walk-through for setting up HDFS on Hadoop 2 can be found at https://web.archive.org/web/20221007121526/https://www.alexjf.net/blog/distributed-systems/hadoop-yarn-installation-definitive-guide/.

Pseudo-distributed

> A quickstart has been added to the quickstart chapter. See quickstart-pseudo. Some of the information that was originally in this section has been moved there.

A pseudo-distributed mode is simply a fully-distributed mode run on a single host. Use this HBase configuration for testing and prototyping purposes only. Do not use this configuration for production or for performance evaluation.

Fully-distributed

By default, HBase runs in stand-alone mode. Both stand-alone mode and pseudo-distributed mode are provided for the purposes of small-scale testing. For a production environment, distributed mode is advised. In distributed mode, multiple instances of HBase daemons run on multiple servers in the cluster.

Just as in pseudo-distributed mode, a fully distributed configuration requires that you set the `hbase.cluster.distributed` property to `true`. Typically, the `hbase.rootdir` is configured to point to a highly-available HDFS filesystem.

In addition, the cluster is configured so that multiple cluster nodes enlist as RegionServers, ZooKeeper QuorumPeers, and backup HMaster servers. These configuration basics are all demonstrated in quickstart-fully-distributed.

Distributed RegionServers

Typically, your cluster will contain multiple RegionServers all running on different servers, as well as primary and backup Master and ZooKeeper daemons. The *conf/regionservers* file on the master server contains a list of hosts whose RegionServers are associated with this cluster. Each host is on a separate line. All hosts listed in this file will have their RegionServer processes started and stopped when the master server starts or stops.

ZooKeeper and HBase

See the ZooKeeper section for ZooKeeper setup instructions for HBase.

Example: Distributed HBase Cluster

This is a bare-bones *conf/hbase-site.xml* for a distributed HBase cluster. A cluster that is used for real-world work would contain more custom configuration parameters. Most HBase configuration directives have default values, which are used unless the value is overridden in the *hbase-site.xml*. See "Configuration Files" for more information.

```
<configuration>
  <property>
    <name>hbase.rootdir</name>
    <value>hdfs://namenode.example.org:9000/hbase</value>
  </property>
  <property>
    <name>hbase.cluster.distributed</name>
    <value>true</value>
  </property>
  <property>
    <name>hbase.zookeeper.quorum</name>
    <value>node-a.example.com,node-b.example.com,node-c.example.com</value>
  </property>
</configuration>
```

This is an example *conf/regionservers* file, which contains a list of nodes that should run a RegionServer in the cluster. These nodes need HBase installed and they need to use the same contents of the *conf/* directory as the Master server.

```
node-a.example.com
node-b.example.com
node-c.example.com
```

This is an example *conf/backup-masters* file, which contains a list of each node that should run a backup Master instance. The backup Master instances will sit idle unless the main Master becomes unavailable.

```
node-b.example.com
node-c.example.com
```

Distributed HBase Quickstart

See quickstart-fully-distributed for a walk-through of a simple three-node cluster configuration with multiple ZooKeeper, backup HMaster, and RegionServer instances.

Procedure: HDFS Client Configuration

1. Of note, if you have made HDFS client configuration changes on your Hadoop cluster, such as configuration directives for HDFS clients, as opposed to server-side configurations, you must use one of the following methods to enable HBase to see and use these configuration changes:
    - Add a pointer to your `HADOOP_CONF_DIR` to the `HBASE_CLASSPATH` environment variable in *hbase-env.sh*.
    - Add a copy of *hdfs-site.xml* (or *hadoop-site.xml*) or, better, symlinks, under `$HBASE_HOME/conf`, or
    - if only a small set of HDFS client configurations, add them to *hbase-site.xml*.

An example of such an HDFS client configuration is `dfs.replication`. If for example, you want to run with a replication factor of 5, HBase will create files with the default of 3 unless you do the above to make the configuration available to HBase.

Choosing between the Classic Package and the BYO Hadoop Package

Starting with HBase 3.0, HBase includes two binary packages. The classic package includes both the HBase and Hadoop components, while the Hadoop-less "Bring Your Own Hadoop" package omits the Hadoop components, and uses the files from an existing Hadoop installation. The classic binary package filename is named `hbase-VERSION-bin.tar.gz` (e.g. *hbase-3.0.0-bin.tar.gz*), while the Hadoop-less package is `hbase-byo-hadoop-VERSION-bin.tar.gz` (e.g. *hbase-byo-hadoop-3.0.0-bin.tar.gz*).

If the cluster nodes already have Hadoop installed, you can use the Hadoop-less package. In this case you need to make sure that the `HADOOP_HOME` environment variable is set and points to the Hadoop installation. The easiest way to ensure this is to set it in *hbase-env.sh*. You still need to make sure that the Hadoop configuration files are present on the HBase classpath, as described above.

Advantages of the BYO Hadoop package:

- There is no need to replace the Hadoop libraries.
- It is easier to upgrade Hadoop and HBase independently (as long as compatible versions are used).
- Both the package and installed size are about 100 MB smaller.

Running and Confirming Your Installation

Make sure HDFS is running first. Start and stop the Hadoop HDFS daemons by running *bin/start-hdfs.sh* over in the `HADOOP_HOME` directory. You can ensure it started properly by testing the `put` and `get` of files into the Hadoop filesystem. HBase does not normally use the MapReduce or YARN daemons. These do not need to be started.

If you are managing your own ZooKeeper, start it and confirm it's running, else HBase will start up ZooKeeper for you as part of its start process.

Start HBase with the following command:

```
bin/start-hbase.sh
```

Run the above from the `HBASE_HOME` directory.

You should now have a running HBase instance. HBase logs can be found in the *logs* subdirectory. Check them out especially if HBase had trouble starting.

HBase also puts up a UI listing vital attributes. By default it's deployed on the Master host at port 16010 (HBase RegionServers listen on port 16020 by default and put up an informational HTTP server at port 16030). If the Master is running on a host named `master.example.org` on the default port, point your browser at http://master.example.org:16010 to see the web interface.

Once HBase has started, see the shell exercises documentation for how to create tables, add data, scan your insertions, and finally disable and drop your tables.

To stop HBase after exiting the HBase shell enter

```
$ ./bin/stop-hbase.sh
stopping hbase..............
```

Shutdown can take a moment to complete. It can take longer if your cluster is comprised of many machines. If you are running a distributed operation, be sure to wait until HBase has shut down completely before stopping the Hadoop daemons.

Default Configuration

hbase-site.xml and *hbase-default.xml*

Just as in Hadoop where you add site-specific HDFS configuration to the *hdfs-site.xml* file, for HBase, site specific customizations go into the file *conf/hbase-site.xml*. For the list of configurable properties, see hbase default configurations below or view the raw *hbase-default.xml* source file in the HBase source code at *src/main/resources*.

Not all configuration options make it out to *hbase-default.xml*. Some configurations would only appear in source code; the only way to identify these changes are through code review.

Currently, changes here will require a cluster restart for HBase to notice the change.

HBase Default configuration

The documentation below is generated using the default hbase configuration file, *hbase-default.xml*, as source.

`hbase.tmp.dir`

Description: Temporary directory on the local filesystem. Change this setting to point to a location more permanent than '/tmp', the usual resolve for java.io.tmpdir, as the '/tmp' directory is cleared on machine restart.
Default: `${java.io.tmpdir}/hbase-${user.name}`

`hbase.rootdir`

Description: The directory shared by region servers and into which HBase persists. The URL should be 'fully-qualified' to include the filesystem scheme. For example, to specify the HDFS directory '/hbase' where the HDFS instance's namenode is running at namenode.example.org on port 9000, set this value to: hdfs://namenode.example.org:9000/hbase. By default, we write to whatever ${hbase.tmp.dir} is set too -- usually /tmp -- so change this configuration or else all data will be lost on machine restart.
Default: `${hbase.tmp.dir}/hbase`

`hbase.cluster.distributed`

Description: The mode the cluster will be in. Possible values are false for standalone mode and true for distributed mode. If false, startup will run all HBase and ZooKeeper daemons together in the one JVM.
Default: `false`

`hbase.zookeeper.quorum`

Description: Comma separated list of servers in the ZooKeeper ensemble (This config. should have been named hbase.zookeeper.ensemble). For example, "host1.mydomain.com,host2.mydomain.com,host3.mydomain.com". By default this is set to localhost for local and pseudo-distributed modes of operation. For a fully-distributed setup, this should be set to a full list of ZooKeeper ensemble servers. If HBASE_MANAGES_ZK is set in hbase-env.sh this is the list of servers which hbase will

start/stop ZooKeeper on as part of cluster start/stop. Client-side, we will take this list of ensemble members and put it together with the hbase.zookeeper.property.clientPort config. and pass it into zookeeper constructor as the connectString parameter.
Default: `127.0.0.1`

`zookeeper.recovery.retry.maxsleeptime`

Description: Max sleep time before retry zookeeper operations in milliseconds, a max time is needed here so that sleep time won't grow unboundedly
Default: `60000`

`hbase.local.dir`

Description: Directory on the local filesystem to be used as a local storage.
Default: `${hbase.tmp.dir}/local/`

`hbase.master.port`

Description: The port the HBase Master should bind to.
Default: `16000`

`hbase.master.info.port`

Description: The port for the HBase Master web UI. Set to -1 if you do not want a UI instance run.
Default: `16010`

`hbase.master.info.bindAddress`

Description: The bind address for the HBase Master web UI
Default: `0.0.0.0`

`hbase.master.logcleaner.plugins`

Description: A comma-separated list of BaseLogCleanerDelegate invoked by the LogsCleaner service. These WAL cleaners are called in order, so put the cleaner that prunes the most files in front. To implement your own BaseLogCleanerDelegate, just put it in HBase's classpath and add the fully qualified class name here. Always add the above default log cleaners in the list.
Default: `org.apache.hadoop.hbase.master.cleaner.TimeToLiveLogCleaner,org.apache.hadoop.hbase.master.cleaner.TimeToLiveProcedureWALCleaner,org.apache.hadoop.hbase.master.cleaner.TimeToLiveMasterLocalStoreWALCleaner`

`hbase.master.logcleaner.ttl`

Description: How long a WAL remain in the archive ({hbase.rootdir}/oldWALs) directory, after which it will be cleaned by a Master thread. The value is in milliseconds.
Default: `600000`

`hbase.master.hfilecleaner.plugins`

Description: A comma-separated list of BaseHFileCleanerDelegate invoked by the HFileCleaner service. These HFiles cleaners are called in order, so put the cleaner that prunes the most files in front. To implement your own BaseHFileCleanerDelegate, just put it in HBase's classpath and add the fully qualified class name here. Always add the above default hfile cleaners in the list as they will be overwritten in hbase-site.xml.
Default: `org.apache.hadoop.hbase.master.cleaner.TimeToLiveHFileCleaner,org.apache.hadoop.hbase.master.cleaner.TimeToLiveMasterLocalStoreHFileCleaner`

`hbase.master.infoserver.redirect`

Description: Whether or not the Master listens to the Master web UI port (hbase.master.info.port) and redirects requests to the web UI server shared by the Master and RegionServer. Config. makes sense when Master is serving Regions (not the default).
Default: `true`

`hbase.master.fileSplitTimeout`

Description: Splitting a region, how long to wait on the file-splitting step before aborting the attempt. Default: 600000. This setting used to be known as hbase.regionserver.fileSplitTimeout in hbase-1.x. Split is now run master-side hence the rename (If a 'hbase.master.fileSplitTimeout' setting found, will use it to prime the current 'hbase.master.fileSplitTimeout' Configuration.
Default: `600000`

`hbase.regionserver.port`

Description: The port the HBase RegionServer binds to.
Default: `16020`

`hbase.regionserver.info.port`

Description: The port for the HBase RegionServer web UI Set to -1 if you do not want the RegionServer UI to run.
Default: `16030`

`hbase.regionserver.info.bindAddress`

Description: The address for the HBase RegionServer web UI

Default: `0.0.0.0`

`hbase.regionserver.info.port.auto`

Description: Whether or not the Master or RegionServer UI should search for a port to bind to. Enables automatic port search if hbase.regionserver.info.port is already in use. Useful for testing, turned off by default.

Default: `false`

`hbase.regionserver.handler.count`

Description: Count of RPC Listener instances spun up on RegionServers. Same property is used by the Master for count of master handlers. Too many handlers can be counter-productive. Make it a multiple of CPU count. If mostly read-only, handlers count close to cpu count does well. Start with twice the CPU count and tune from there.

Default: `30`

`hbase.ipc.server.callqueue.handler.factor`

Description: Factor to determine the number of call queues. A value of 0 means a single queue shared between all the handlers. A value of 1 means that each handler has its own queue.

Default: `0.1`

`hbase.ipc.server.callqueue.read.ratio`

Description: Split the call queues into read and write queues. The specified interval (which should be between 0.0 and 1.0) will be multiplied by the number of call queues. A value of 0 indicate to not split the call queues, meaning that both read and write requests will be pushed to the same set of queues. A value lower than 0.5 means that there will be less read queues than write queues. A value of 0.5 means there will be the same number of read and write queues. A value greater than 0.5 means that there will be more read queues than write queues. A value of 1.0 means that all the queues except one are used to dispatch read requests. Example: Given the total number of call queues being 10 a read.ratio of 0 means that: the 10 queues will contain both read/write requests. a read.ratio of 0.3 means that: 3 queues will contain only read requests and 7 queues will contain only write requests. a read.ratio of 0.5 means that: 5 queues will contain only read requests and 5 queues will contain only write requests. a read.ratio of 0.8 means that: 8 queues will

contain only read requests and 2 queues will contain only write requests. a read.ratio of 1 means that: 9 queues will contain only read requests and 1 queues will contain only write requests.
Default: `0`

`hbase.ipc.server.callqueue.scan.ratio`

Description: Given the number of read call queues, calculated from the total number of call queues multiplied by the callqueue.read.ratio, the scan.ratio property will split the read call queues into small-read and long-read queues. A value lower than 0.5 means that there will be less long-read queues than short-read queues. A value of 0.5 means that there will be the same number of short-read and long-read queues. A value greater than 0.5 means that there will be more long-read queues than short-read queues A value of 0 or 1 indicate to use the same set of queues for gets and scans. Example: Given the total number of read call queues being 8 a scan.ratio of 0 or 1 means that: 8 queues will contain both long and short read requests. a scan.ratio of 0.3 means that: 2 queues will contain only long-read requests and 6 queues will contain only short-read requests. a scan.ratio of 0.5 means that: 4 queues will contain only long-read requests and 4 queues will contain only short-read requests. a scan.ratio of 0.8 means that: 6 queues will contain only long-read requests and 2 queues will contain only short-read requests.
Default: `0`

`hbase.regionserver.msginterval`

Description: Interval between messages from the RegionServer to Master in milliseconds.
Default: `3000`

`hbase.regionserver.logroll.period`

Description: Period at which we will roll the commit log regardless of how many edits it has.
Default: `3600000`

`hbase.regionserver.logroll.errors.tolerated`

Description: The number of consecutive WAL close errors we will allow before triggering a server abort. A setting of 0 will cause the region server to abort if closing the current WAL writer fails during log rolling. Even a small value (2 or 3) will allow a region server to ride over transient HDFS errors.
Default: `2`

`hbase.regionserver.free.heap.min.memory.size`

Description: Defines the minimum amount of heap memory that must remain free for the RegionServer to start, specified in bytes or human-readable formats like '512m' for megabytes or '4g' for gigabytes. If not set, the default is 20% of the total heap size. To disable the check entirely, set this value to 0. If the combined memory usage of memstore and block cache exceeds (total heap - this value), the RegionServer will fail to start.
Default: `(empty)`

`hbase.regionserver.global.memstore.size`

Description: Maximum size of all memstores in a region server before new updates are blocked and flushes are forced. Defaults to 40% of heap (0.4). Updates are blocked and flushes are forced until size of all memstores in a region server hits hbase.regionserver.global.memstore.size.lower.limit. The default value in this configuration has been intentionally left empty in order to honor the old hbase.regionserver.global.memstore.upperLimit property if present.
Default: `(empty)`

`hbase.regionserver.global.memstore.memory.size`

Description: Maximum size of all memstores in a region server before new updates are blocked and flushes are forced, specified in bytes or human-readable formats like '10m' for megabytes or '10g' for gigabytes. This configuration allows setting an absolute memory size instead of a percentage of the maximum heap. Takes precedence over hbase.regionserver.global.memstore.size if both are specified.
Default: `(empty)`

`hbase.regionserver.global.memstore.size.lower.limit`

Description: Maximum size of all memstores in a region server before flushes are forced. Defaults to 95% of hbase.regionserver.global.memstore.size (0.95). A 100% value for this value causes the minimum possible flushing to occur when updates are blocked due to memstore limiting. The default value in this configuration has been intentionally left empty in order to honor the old hbase.regionserver.global.memstore.lowerLimit property if present.
Default: `(empty)`

`hbase.systemtables.compacting.memstore.type`

Description: Determines the type of memstore to be used for system tables like META, namespace tables etc. By default NONE is the type and hence we use the default memstore for all the system tables. If we need to use compacting memstore for system tables then set this property to BASIC/EAGER

Default: `NONE`

`hbase.regionserver.optionalcacheflushinterval`

Description: Maximum amount of time an edit lives in memory before being automatically flushed. Default 1 hour. Set it to 0 to disable automatic flushing.

Default: `3600000`

`hbase.regionserver.dns.interface`

Description: The name of the Network Interface from which a region server should report its IP address.

Default: `default`

`hbase.regionserver.dns.nameserver`

Description: The host name or IP address of the name server (DNS) which a region server should use to determine the host name used by the master for communication and display purposes.

Default: `default`

`hbase.regionserver.region.split.policy`

Description: A split policy determines when a region should be split. The various other split policies that are available currently are BusyRegionSplitPolicy, ConstantSizeRegionSplitPolicy, DisabledRegionSplitPolicy, DelimitedKeyPrefixRegionSplitPolicy, KeyPrefixRegionSplitPolicy, and SteppingSplitPolicy. DisabledRegionSplitPolicy blocks manual region splitting.

Default: `org.apache.hadoop.hbase.regionserver.SteppingSplitPolicy`

`hbase.regionserver.regionSplitLimit`

Description: Limit for the number of regions after which no more region splitting should take place. This is not hard limit for the number of regions but acts as a guideline for the regionserver to stop splitting after a certain limit. Default is set to 1000.

Default: `1000`

`zookeeper.session.timeout`

Description: ZooKeeper session timeout in milliseconds. It is used in two different ways. First, this value is used in the ZK client that HBase uses to connect to the ensemble. It is also used by HBase when it starts a ZK server and it is passed as the 'maxSessionTimeout'. See https://zookeeper.apache.org/doc/current/zookeeperProgrammers.html#ch_zkSessions. For example, if an HBase region server connects to a ZK ensemble that's also managed by HBase, then the session timeout will be the one specified by this configuration. But, a region server that connects to an ensemble managed with a different configuration will be subjected that ensemble's maxSessionTimeout. So, even though HBase might propose using 90 seconds, the ensemble can have a max timeout lower than this and it will take precedence. The current default maxSessionTimeout that ZK ships with is 40 seconds, which is lower than HBase's.
Default: `90000`

`zookeeper.znode.parent`

Description: Root ZNode for HBase in ZooKeeper. All of HBase's ZooKeeper files that are configured with a relative path will go under this node. By default, all of HBase's ZooKeeper file paths are configured with a relative path, so they will all go under this directory unless changed.
Default: `/hbase`

`zookeeper.znode.acl.parent`

Description: Root ZNode for access control lists.
Default: `acl`

`hbase.zookeeper.dns.interface`

Description: The name of the Network Interface from which a ZooKeeper server should report its IP address.
Default: `default`

`hbase.zookeeper.dns.nameserver`

Description: The host name or IP address of the name server (DNS) which a ZooKeeper server should use to determine the host name used by the master for communication and display purposes.
Default: `default`

`hbase.zookeeper.peerport`

Description: Port used by ZooKeeper peers to talk to each other. See https://zookeeper.apache.org/doc/r3.4.10/zookeeperStarted.html#sc_RunningReplicatedZooKeeper for more information.
Default: `2888`

`hbase.zookeeper.leaderport`

Description: Port used by ZooKeeper for leader election. See https://zookeeper.apache.org/doc/r3.4.10/zookeeperStarted.html#sc_RunningReplicatedZooKeeper for more information.
Default: `3888`

`hbase.zookeeper.property.initLimit`

Description: Property from ZooKeeper's config zoo.cfg. The number of ticks that the initial synchronization phase can take.
Default: `10`

`hbase.zookeeper.property.syncLimit`

Description: Property from ZooKeeper's config zoo.cfg. The number of ticks that can pass between sending a request and getting an acknowledgment.
Default: `5`

`hbase.zookeeper.property.dataDir`

Description: Property from ZooKeeper's config zoo.cfg. The directory where the snapshot is stored.
Default: `${hbase.tmp.dir}/zookeeper`

`hbase.zookeeper.property.clientPort`

Description: Property from ZooKeeper's config zoo.cfg. The port at which the clients will connect.
Default: `2181`

`hbase.zookeeper.property.maxClientCnxns`

Description: Property from ZooKeeper's config zoo.cfg. Limit on number of concurrent connections (at the socket level) that a single client, identified by IP address, may make to a single member of the ZooKeeper ensemble. Set high to avoid zk connection issues

running standalone and pseudo-distributed.
Default: `300`

`hbase.client.write.buffer`

Description: Default size of the BufferedMutator write buffer in bytes. A bigger buffer takes more memory -- on both the client and server side since server instantiates the passed write buffer to process it -- but a larger buffer size reduces the number of RPCs made. For an estimate of server-side memory-used, evaluate hbase.client.write.buffer * hbase.regionserver.handler.count
Default: `2097152`

`hbase.client.pause`

Description: General client pause value. Used mostly as value to wait before running a retry of a failed get, region lookup, etc. See hbase.client.retries.number for description of how we backoff from this initial pause amount and how this pause works w/ retries.
Default: `100`

`hbase.client.pause.server.overloaded`

Description: Pause time when encountering an exception indicating a server is overloaded, CallQueueTooBigException or CallDroppedException. Set this property to a higher value than hbase.client.pause if you observe frequent CallQueueTooBigException or CallDroppedException from the same RegionServer and the call queue there keeps filling up. This config used to be called hbase.client.pause.cqtbe, which has been deprecated as of 2.5.0.
Default: `(empty)`

`hbase.client.retries.number`

Description: Maximum retries. Used as maximum for all retryable operations such as the getting of a cell's value, starting a row update, etc. Retry interval is a rough function based on hbase.client.pause. At first we retry at this interval but then with backoff, we pretty quickly reach retrying every ten seconds. See HConstants#RETRY_BACKOFF for how the backup ramps up. Change this setting and hbase.client.pause to suit your workload.
Default: `15`

`hbase.client.max.total.tasks`

Description: The maximum number of concurrent mutation tasks a single HTable instance will send to the cluster.
Default: `100`

`hbase.client.max.perserver.tasks`

Description: The maximum number of concurrent mutation tasks a single HTable instance will send to a single region server.
Default: `2`

`hbase.client.max.perregion.tasks`

Description: The maximum number of concurrent mutation tasks the client will maintain to a single Region. That is, if there is already hbase.client.max.perregion.tasks writes in progress for this region, new puts won't be sent to this region until some writes finishes.
Default: `1`

`hbase.client.perserver.requests.threshold`

Description: The max number of concurrent pending requests for one server in all client threads (process level). Exceeding requests will be thrown ServerTooBusyException immediately to prevent user's threads being occupied and blocked by only one slow region server. If you use a fix number of threads to access HBase in a synchronous way, set this to a suitable value which is related to the number of threads will help you. See https://issues.apache.org/jira/browse/HBASE-16388 for details.
Default: `2147483647`

`hbase.client.scanner.caching`

Description: Number of rows that we try to fetch when calling next on a scanner if it is not served from (local, client) memory. This configuration works together with hbase.client.scanner.max.result.size to try and use the network efficiently. The default value is Integer.MAX_VALUE by default so that the network will fill the chunk size defined by hbase.client.scanner.max.result.size rather than be limited by a particular number of rows since the size of rows varies table to table. If you know ahead of time that you will not require more than a certain number of rows from a scan, this configuration should be set to that row limit via Scan#setCaching. Higher caching values will enable faster scanners but will eat up more memory and some calls of next may take longer and longer times when the cache is empty. Do not set this value such that the time between

invocations is greater than the scanner timeout; i.e. hbase.client.scanner.timeout.period
Default: `2147483647`

`hbase.client.keyvalue.maxsize`

Description: Specifies the combined maximum allowed size of a KeyValue instance. This is to set an upper boundary for a single entry saved in a storage file. Since they cannot be split it helps avoiding that a region cannot be split any further because the data is too large. It seems wise to set this to a fraction of the maximum region size. Setting it to zero or less disables the check.
Default: `10485760`

`hbase.server.keyvalue.maxsize`

Description: Maximum allowed size of an individual cell, inclusive of value and all key components. A value of 0 or less disables the check. The default value is 10MB. This is a safety setting to protect the server from OOM situations.
Default: `10485760`

`hbase.client.scanner.timeout.period`

Description: Client scanner lease period in milliseconds.
Default: `60000`

`hbase.client.localityCheck.threadPoolSize`

Description:
Default: `2`

`hbase.bulkload.retries.number`

Description: Maximum retries. This is maximum number of iterations to atomic bulk loads are attempted in the face of splitting operations 0 means never give up.
Default: `10`

`hbase.compaction.after.bulkload.enable`

Description: Request Compaction after bulkload immediately. If bulkload is continuous, the triggered compactions may increase load, bring about performance side effect.
Default: `false`

`hbase.master.balancer.maxRitPercent`

Description: The max percent of regions in transition when balancing. The default value is 1.0. So there are no balancer throttling. If set this config to 0.01, It means that there are at most 1% regions in transition when balancing. Then the cluster's availability is at least 99% when balancing.
Default: `1.0`

`hbase.balancer.period`

Description: Period at which the region balancer runs in the Master, in milliseconds.
Default: `300000`

`hbase.master.oldwals.dir.updater.period`

Description: Period at which the oldWALs directory size calculator/updater will run in the Master, in milliseconds.
Default: `300000`

`hbase.regions.slop`

Description: The load balancer can trigger for several reasons. This value controls one of those reasons. Run the balancer if any regionserver has a region count outside the range of average +/- (average * slop) regions. If the value of slop is negative, disable sloppiness checks. The balancer can still run for other reasons, but sloppiness will not be one of them. If the value of slop is 0, run the balancer if any server has a region count more than 1 from the average. If the value of slop is 100, run the balancer if any server has a region count greater than 101 times the average. The default value of this parameter is 0.2, which runs the balancer if any server has a region count less than 80% of the average, or greater than 120% of the average. Note that for the default StochasticLoadBalancer, this does not guarantee any balancing actions will be taken, but only that the balancer will attempt to run.
Default: `0.2`

`hbase.normalizer.period`

Description: Period at which the region normalizer runs in the Master, in milliseconds.
Default: `300000`

`hbase.normalizer.split.enabled`

Description: Whether to split a region as part of normalization.
Default: `true`

`hbase.normalizer.merge.enabled`

Description: Whether to merge a region as part of normalization.
Default: `true`

`hbase.normalizer.merge.min.region.count`

Description: The minimum number of regions in a table to consider it for merge normalization.
Default: `3`

`hbase.normalizer.merge.min_region_age.days`

Description: The minimum age for a region to be considered for a merge, in days.
Default: `3`

`hbase.normalizer.merge.min_region_size.mb`

Description: The minimum size for a region to be considered for a merge, in whole MBs.
Default: `1`

`hbase.normalizer.merge.merge_request_max_number_of_regions`

Description: The maximum number of region count in a merge request for merge normalization.
Default: `100`

`hbase.table.normalization.enabled`

Description: This config is used to set default behaviour of normalizer at table level. To override this at table level one can set NORMALIZATION_ENABLED at table descriptor level and that property will be honored
Default: `false`

`hbase.server.thread.wakefrequency`

Description: In master side, this config is the period used for FS related behaviors: checking if hdfs is out of safe mode, setting or checking hbase.version file, setting or checking hbase.id file. Using default value should be fine. In regionserver side, this config

is used in several places: flushing check interval, compaction check interval, wal rolling check interval. Specially, admin can tune flushing and compaction check interval by hbase.regionserver.flush.check.period and hbase.regionserver.compaction.check.period. (in milliseconds)
Default: `10000`

`hbase.regionserver.flush.check.period`

Description: It determines the flushing check period of PeriodicFlusher in regionserver. If unset, it uses hbase.server.thread.wakefrequency as default value. (in milliseconds)
Default: `${hbase.server.thread.wakefrequency}`

`hbase.regionserver.compaction.check.period`

Description: It determines the compaction check period of CompactionChecker in regionserver. If unset, it uses hbase.server.thread.wakefrequency as default value. (in milliseconds)
Default: `${hbase.server.thread.wakefrequency}`

`hbase.server.versionfile.writeattempts`

Description: How many times to retry attempting to write a version file before just aborting. Each attempt is separated by the hbase.server.thread.wakefrequency milliseconds.
Default: `3`

`hbase.hregion.memstore.flush.size`

Description: Memstore will be flushed to disk if size of the memstore exceeds this number of bytes. Value is checked by a thread that runs every hbase.server.thread.wakefrequency.
Default: `134217728`

`hbase.hregion.percolumnfamilyflush.size.lower.bound.min`

Description: If FlushLargeStoresPolicy is used and there are multiple column families, then every time that we hit the total memstore limit, we find out all the column families whose memstores exceed a "lower bound" and only flush them while retaining the others in memory. The "lower bound" will be "hbase.hregion.memstore.flush.size / column_family_number" by default unless value of this property is larger than that. If none of the families have their memstore size more than lower bound, all the memstores will be

flushed (just as usual).
Default: `16777216`

`hbase.hregion.preclose.flush.size`

Description: If the memstores in a region are this size or larger when we go to close, run a "pre-flush" to clear out memstores before we put up the region closed flag and take the region offline. On close, a flush is run under the close flag to empty memory. During this time the region is offline and we are not taking on any writes. If the memstore content is large, this flush could take a long time to complete. The preflush is meant to clean out the bulk of the memstore before putting up the close flag and taking the region offline so the flush that runs under the close flag has little to do.
Default: `5242880`

`hbase.hregion.memstore.block.multiplier`

Description: Block updates if memstore has hbase.hregion.memstore.block.multiplier times hbase.hregion.memstore.flush.size bytes. Useful preventing runaway memstore during spikes in update traffic. Without an upper-bound, memstore fills such that when it flushes the resultant flush files take a long time to compact or split, or worse, we OOME.
Default: `4`

`hbase.hregion.memstore.mslab.enabled`

Description: Enables the MemStore-Local Allocation Buffer, a feature which works to prevent heap fragmentation under heavy write loads. This can reduce the frequency of stop-the-world GC pauses on large heaps.
Default: `true`

`hbase.hregion.memstore.mslab.chunksize`

Description: The maximum byte size of a chunk in the MemStoreLAB. Unit: bytes
Default: `2097152`

`hbase.regionserver.offheap.global.memstore.size`

Description: The amount of off-heap memory all MemStores in a RegionServer may use. A value of 0 means that no off-heap memory will be used and all chunks in MSLAB will be HeapByteBuffer, otherwise the non-zero value means how many megabyte of off-heap memory will be used for chunks in MSLAB and all chunks in MSLAB will be

DirectByteBuffer. Unit: megabytes.
Default: `0`

`hbase.hregion.memstore.mslab.max.allocation`

Description: The maximal size of one allocation in the MemStoreLAB, if the desired byte size exceed this threshold then it will be just allocated from JVM heap rather than MemStoreLAB.
Default: `262144`

`hbase.hregion.max.filesize`

Description: Maximum file size. If the sum of the sizes of a region's HFiles has grown to exceed this value, the region is split in two. There are two choices of how this option works, the first is when any store's size exceed the threshold then split, and the other is overall region's size exceed the threshold then split, it can be configed by hbase.hregion.split.overallfiles.
Default: `10737418240`

`hbase.hregion.split.overallfiles`

Description: If we should sum overall region files size when check to split.
Default: `true`

`hbase.hregion.majorcompaction`

Description: Time between major compactions, expressed in milliseconds. Set to 0 to disable time-based automatic major compactions. User-requested and size-based major compactions will still run. This value is multiplied by hbase.hregion.majorcompaction.jitter to cause compaction to start at a somewhat-random time during a given window of time. The default value is 7 days, expressed in milliseconds. If major compactions are causing disruption in your environment, you can configure them to run at off-peak times for your deployment, or disable time-based major compactions by setting this parameter to 0, and run major compactions in a cron job or by another external mechanism.
Default: `604800000`

`hbase.hregion.majorcompaction.jitter`

Description: A multiplier applied to hbase.hregion.majorcompaction to cause compaction to occur a given amount of time either side of hbase.hregion.majorcompaction. The smaller the number, the closer the compactions will happen to the

hbase.hregion.majorcompaction interval.
Default: `0.50`

`hbase.hstore.compactionThreshold`

Description: If more than or equal to this number of StoreFiles exist in any one Store (one StoreFile is written per flush of MemStore), a compaction is run to rewrite all StoreFiles into a single StoreFile. Larger values delay compaction, but when compaction does occur, it takes longer to complete.
Default: `3`

`hbase.regionserver.compaction.enabled`

Description: Enable/disable compactions on by setting true/false. We can further switch compactions dynamically with the compaction_switch shell command.
Default: `true`

`hbase.hstore.flusher.count`

Description: The number of flush threads. With fewer threads, the MemStore flushes will be queued. With more threads, the flushes will be executed in parallel, increasing the load on HDFS, and potentially causing more compactions.
Default: `2`

`hbase.hstore.blockingStoreFiles`

Description: If more than this number of StoreFiles exist in any one Store (one StoreFile is written per flush of MemStore), updates are blocked for this region until a compaction is completed, or until hbase.hstore.blockingWaitTime has been exceeded.
Default: `16`

`hbase.hstore.blockingWaitTime`

Description: The time for which a region will block updates after reaching the StoreFile limit defined by hbase.hstore.blockingStoreFiles. After this time has elapsed, the region will stop blocking updates even if a compaction has not been completed.
Default: `90000`

`hbase.hstore.compaction.min`

Description: The minimum number of StoreFiles which must be eligible for compaction before compaction can run. The goal of tuning hbase.hstore.compaction.min is to avoid

ending up with too many tiny StoreFiles to compact. Setting this value to 2 would cause a minor compaction each time you have two StoreFiles in a Store, and this is probably not appropriate. If you set this value too high, all the other values will need to be adjusted accordingly. For most cases, the default value is appropriate (empty value here, results in 3 by code logic). In previous versions of HBase, the parameter hbase.hstore.compaction.min was named hbase.hstore.compactionThreshold.

Default: `(empty)`

`hbase.hstore.compaction.max`

Description: The maximum number of StoreFiles which will be selected for a single minor compaction, regardless of the number of eligible StoreFiles. Effectively, the value of hbase.hstore.compaction.max controls the length of time it takes a single compaction to complete. Setting it larger means that more StoreFiles are included in a compaction. For most cases, the default value is appropriate.

Default: `10`

`hbase.hstore.compaction.min.size`

Description: A StoreFile (or a selection of StoreFiles, when using ExploringCompactionPolicy) smaller than this size will always be eligible for minor compaction. HFiles this size or larger are evaluated by hbase.hstore.compaction.ratio to determine if they are eligible. Because this limit represents the "automatic include" limit for all StoreFiles smaller than this value, this value may need to be reduced in write-heavy environments where many StoreFiles in the 1-2 MB range are being flushed, because every StoreFile will be targeted for compaction and the resulting StoreFiles may still be under the minimum size and require further compaction. If this parameter is lowered, the ratio check is triggered more quickly. This addressed some issues seen in earlier versions of HBase but changing this parameter is no longer necessary in most situations. Default: 128 MB expressed in bytes.

Default: `134217728`

`hbase.hstore.compaction.max.size`

Description: A StoreFile (or a selection of StoreFiles, when using ExploringCompactionPolicy) larger than this size will be excluded from compaction. The effect of raising hbase.hstore.compaction.max.size is fewer, larger StoreFiles that do not get compacted often. If you feel that compaction is happening too often without much benefit, you can try raising this value. Default: the value of LONG.MAX_VALUE, expressed

in bytes.
Default: `9223372036854775807`

`hbase.hstore.compaction.ratio`

Description: For minor compaction, this ratio is used to determine whether a given StoreFile which is larger than hbase.hstore.compaction.min.size is eligible for compaction. Its effect is to limit compaction of large StoreFiles. The value of hbase.hstore.compaction.ratio is expressed as a floating-point decimal. A large ratio, such as 10, will produce a single giant StoreFile. Conversely, a low value, such as .25, will produce behavior similar to the BigTable compaction algorithm, producing four StoreFiles. A moderate value of between 1.0 and 1.4 is recommended. When tuning this value, you are balancing write costs with read costs. Raising the value (to something like 1.4) will have more write costs, because you will compact larger StoreFiles. However, during reads, HBase will need to seek through fewer StoreFiles to accomplish the read. Consider this approach if you cannot take advantage of Bloom filters. Otherwise, you can lower this value to something like 1.0 to reduce the background cost of writes, and use Bloom filters to control the number of StoreFiles touched during reads. For most cases, the default value is appropriate.
Default: `1.2F`

`hbase.hstore.compaction.ratio.offpeak`

Description: Allows you to set a different (by default, more aggressive) ratio for determining whether larger StoreFiles are included in compactions during off-peak hours. Works in the same way as hbase.hstore.compaction.ratio. Only applies if hbase.offpeak.start.hour and hbase.offpeak.end.hour are also enabled.
Default: `5.0F`

`hbase.hstore.time.to.purge.deletes`

Description: The amount of time to delay purging of delete markers with future timestamps. If unset, or set to 0, all delete markers, including those with future timestamps, are purged during the next major compaction. Otherwise, a delete marker is kept until the major compaction which occurs after the marker's timestamp plus the value of this setting, in milliseconds.
Default: `0`

`hbase.offpeak.start.hour`

Description: The start of off-peak hours, expressed as an integer between 0 and 23, inclusive. Set to -1 to disable off-peak.
Default: `-1`

`hbase.offpeak.end.hour`

Description: The end of off-peak hours, expressed as an integer between 0 and 23, inclusive. Set to -1 to disable off-peak.
Default: `-1`

`hbase.regionserver.thread.compaction.throttle`

Description: There are two different thread pools for compactions, one for large compactions and the other for small compactions. This helps to keep compaction of lean tables (such as hbase:meta) fast. If a compaction is larger than this threshold, it goes into the large compaction pool. In most cases, the default value is appropriate. Default: 2 x hbase.hstore.compaction.max x hbase.hregion.memstore.flush.size (which defaults to 128MB). The value field assumes that the value of hbase.hregion.memstore.flush.size is unchanged from the default.
Default: `2684354560`

`hbase.regionserver.majorcompaction.pagecache.drop`

Description: Specifies whether to drop pages read/written into the system page cache by major compactions. Setting it to true helps prevent major compactions from polluting the page cache, which is almost always required, especially for clusters with low/moderate memory to storage ratio.
Default: `true`

`hbase.regionserver.minorcompaction.pagecache.drop`

Description: Specifies whether to drop pages read/written into the system page cache by minor compactions. Setting it to true helps prevent minor compactions from polluting the page cache, which is most beneficial on clusters with low memory to storage ratio or very write heavy clusters. You may want to set it to false under moderate to low write workload when bulk of the reads are on the most recently written data.
Default: `true`

`hbase.hstore.compaction.kv.max`

Description: The maximum number of KeyValues to read and then write in a batch when flushing or compacting. Set this lower if you have big KeyValues and problems with Out Of Memory Exceptions Set this higher if you have wide, small rows.
Default: `10`

`hbase.storescanner.parallel.seek.enable`

Description: Enables StoreFileScanner parallel-seeking in StoreScanner, a feature which can reduce response latency under special conditions.
Default: `false`

`hbase.storescanner.parallel.seek.threads`

Description: The default thread pool size if parallel-seeking feature enabled.
Default: `10`

`hfile.block.cache.policy`

Description: The eviction policy for the L1 block cache (LRU or TinyLFU).
Default: `LRU`

`hfile.block.cache.size`

Description: Percentage of maximum heap (-Xmx setting) to allocate to block cache used by a StoreFile. Default of 0.4 means allocate 40%. Set to 0 to disable but it's not recommended; you need at least enough cache to hold the storefile indices.
Default: `0.4`

`hfile.block.cache.memory.size`

Description: Defines the maximum heap memory allocated for the HFile block cache, specified in bytes or human-readable formats like '10m' for megabytes or '10g' for gigabytes. This configuration allows setting an absolute memory size instead of a percentage of the maximum heap. Takes precedence over hfile.block.cache.size if both are specified.
Default: `(empty)`

`hfile.block.index.cacheonwrite`

Description: This allows to put non-root multi-level index blocks into the block cache at the time the index is being written.

Default: `false`

`hfile.index.block.max.size`

Description: When the size of a leaf-level, intermediate-level, or root-level index block in a multi-level block index grows to this size, the block is written out and a new block is started.
Default: `131072`

`hbase.bucketcache.ioengine`

Description: Where to store the contents of the bucketcache. One of: offheap, file, files, mmap or pmem. If a file or files, set it to file(s):PATH_TO_FILE. mmap means the content will be in an mmaped file. Use mmap:PATH_TO_FILE. 'pmem' is bucket cache over a file on the persistent memory device. Use pmem:PATH_TO_FILE. See https://hbase.apache.org/docs/architecture/regionserver#off-heap-block-cache for more information.
Default: `(empty)`

`hbase.hstore.compaction.throughput.lower.bound`

Description: The target lower bound on aggregate compaction throughput, in bytes/sec. Allows you to tune the minimum available compaction throughput when the PressureAwareCompactionThroughputController throughput controller is active. (It is active by default.)
Default: `52428800`

`hbase.hstore.compaction.throughput.higher.bound`

Description: The target upper bound on aggregate compaction throughput, in bytes/sec. Allows you to control aggregate compaction throughput demand when the PressureAwareCompactionThroughputController throughput controller is active. (It is active by default.) The maximum throughput will be tuned between the lower and upper bounds when compaction pressure is within the range [0.0, 1.0]. If compaction pressure is 1.0 or greater the higher bound will be ignored until pressure returns to the normal range.
Default: `104857600`

`hbase.bucketcache.size`

Description: It is the total capacity in megabytes of BucketCache. Default: 0.0
Default: `(empty)`

`hbase.bucketcache.bucket.sizes`

Description: A comma-separated list of sizes for buckets for the bucketcache. Can be multiple sizes. List block sizes in order from smallest to largest. The sizes you use will depend on your data access patterns. Must be a multiple of 256 else you will run into 'java.io.IOException: Invalid HFile block magic' when you go to read from cache. If you specify no values here, then you pick up the default bucketsizes set in code (See BucketAllocator#DEFAULT_BUCKET_SIZES).
Default: `(empty)`

`hfile.format.version`

Description: The HFile format version to use for new files. Version 3 adds support for tags in hfiles (See https://hbase.apache.org/docs/security/data-access#tags). Also see the configuration 'hbase.replication.rpc.codec'.
Default: `3`

`hfile.block.bloom.cacheonwrite`

Description: Enables cache-on-write for inline blocks of a compound Bloom filter.
Default: `false`

`io.storefile.bloom.block.size`

Description: The size in bytes of a single block ("chunk") of a compound Bloom filter. This size is approximate, because Bloom blocks can only be inserted at data block boundaries, and the number of keys per data block varies.
Default: `131072`

`hbase.rs.cacheblocksonwrite`

Description: Whether an HFile block should be added to the block cache when the block is finished.
Default: `false`

`hbase.rpc.timeout`

Description: This is for the RPC layer to define how long (millisecond) HBase client applications take for a remote call to time out. It uses pings to check connections but will eventually throw a TimeoutException.
Default: `60000`

`hbase.client.operation.timeout`

Description: Operation timeout is a top-level restriction (millisecond) that makes sure a blocking operation in Table will not be blocked more than this. In each operation, if rpc request fails because of timeout or other reason, it will retry until success or throw RetriesExhaustedException. But if the total time being blocking reach the operation timeout before retries exhausted, it will break early and throw SocketTimeoutException.
Default: `1200000`

`hbase.client.connection.metacache.invalidate-interval.ms`

Description: Interval in milliseconds of checking and invalidating meta cache when table disabled or dropped, when set to zero means disable checking, suggest set it to 24h or a higher value, because disable/delete table usually not very frequently.
Default: `0`

`hbase.cells.scanned.per.heartbeat.check`

Description: The number of cells scanned in between heartbeat checks. Heartbeat checks occur during the processing of scans to determine whether or not the server should stop scanning in order to send back a heartbeat message to the client. Heartbeat messages are used to keep the client-server connection alive during long running scans. Small values mean that the heartbeat checks will occur more often and thus will provide a tighter bound on the execution time of the scan. Larger values mean that the heartbeat checks occur less frequently
Default: `10000`

`hbase.rpc.shortoperation.timeout`

Description: This is another version of "hbase.rpc.timeout". For those RPC operation within cluster, we rely on this configuration to set a short timeout limitation for short operation. For example, short rpc timeout for region server's trying to report to active master can benefit quicker master failover process.
Default: `10000`

`hbase.ipc.client.tcpnodelay`

Description: Set no delay on rpc socket connections. See http://docs.oracle.com/javase/1.5.0/docs/api/java/net/Socket.html#getTcpNoDelay()
Default: `true`

`hbase.unsafe.regionserver.hostname`

Description: This config is for experts: don't set its value unless you really know what you are doing. When set to a non-empty value, this represents the (external facing) hostname for the underlying server. See https://issues.apache.org/jira/browse/HBASE-12954 for details.

Default: `(empty)`

`hbase.unsafe.regionserver.hostname.disable.master.reversedns`

Description: This config is for experts: don't set its value unless you really know what you are doing. When set to true, regionserver will use the current node hostname for the servername and HMaster will skip reverse DNS lookup and use the hostname sent by regionserver instead. Note that this config and hbase.unsafe.regionserver.hostname are mutually exclusive. See https://issues.apache.org/jira/browse/HBASE-18226 for more details.

Default: `false`

`hbase.master.keytab.file`

Description: Full path to the kerberos keytab file to use for logging in the configured HMaster server principal.

Default: `(empty)`

`hbase.master.kerberos.principal`

Description: Ex. "hbase/_HOST@EXAMPLE.COM". The kerberos principal name that should be used to run the HMaster process. The principal name should be in the form: user/hostname@DOMAIN. If "_HOST" is used as the hostname portion, it will be replaced with the actual hostname of the running instance.

Default: `(empty)`

`hbase.regionserver.keytab.file`

Description: Full path to the kerberos keytab file to use for logging in the configured HRegionServer server principal.

Default: `(empty)`

`hbase.regionserver.kerberos.principal`

Description: Ex. "hbase/_HOST@EXAMPLE.COM". The kerberos principal name that should be used to run the HRegionServer process. The principal name should be in the

form: user/hostname@DOMAIN. If "_HOST" is used as the hostname portion, it will be replaced with the actual hostname of the running instance. An entry for this principal must exist in the file specified in hbase.regionserver.keytab.file

Default: `(empty)`

`hadoop.policy.file`

Description: The policy configuration file used by RPC servers to make authorization decisions on client requests. Only used when HBase security is enabled.

Default: `hbase-policy.xml`

`hbase.superuser`

Description: List of users or groups (comma-separated), who are allowed full privileges, regardless of stored ACLs, across the cluster. Only used when HBase security is enabled. Group names should be prefixed with "@".

Default: `(empty)`

`hbase.auth.key.update.interval`

Description: The update interval for master key for authentication tokens in servers in milliseconds. Only used when HBase security is enabled.

Default: `86400000`

`hbase.auth.token.max.lifetime`

Description: The maximum lifetime in milliseconds after which an authentication token expires. Only used when HBase security is enabled.

Default: `604800000`

`hbase.ipc.client.fallback-to-simple-auth-allowed`

Description: When a client is configured to attempt a secure connection, but attempts to connect to an insecure server, that server may instruct the client to switch to SASL SIMPLE (unsecure) authentication. This setting controls whether or not the client will accept this instruction from the server. When false (the default), the client will not allow the fallback to SIMPLE authentication, and will abort the connection.

Default: `false`

`hbase.ipc.server.fallback-to-simple-auth-allowed`

Description: When a server is configured to require secure connections, it will reject connection attempts from clients using SASL SIMPLE (unsecure) authentication. This setting allows secure servers to accept SASL SIMPLE connections from clients when the client requests. When false (the default), the server will not allow the fallback to SIMPLE authentication, and will reject the connection. WARNING: This setting should ONLY be used as a temporary measure while converting clients over to secure authentication. It MUST BE DISABLED for secure operation.
Default: `false`

`hbase.unsafe.client.kerberos.hostname.disable.reversedns`

Description: This config is for experts: don't set its value unless you really know what you are doing. When set to true, HBase client using SASL Kerberos will skip reverse DNS lookup and use provided hostname of the destination for the principal instead. See https://issues.apache.org/jira/browse/HBASE-25665 for more details.
Default: `false`

`hbase.display.keys`

Description: When this is set to true the webUI and such will display all start/end keys as part of the table details, region names, etc. When this is set to false, the keys are hidden.
Default: `true`

`hbase.coprocessor.enabled`

Description: Enables or disables coprocessor loading. If 'false' (disabled), any other coprocessor related configuration will be ignored.
Default: `true`

`hbase.coprocessor.user.enabled`

Description: Enables or disables user (aka. table) coprocessor loading. If 'false' (disabled), any table coprocessor attributes in table descriptors will be ignored. If "hbase.coprocessor.enabled" is 'false' this setting has no effect.
Default: `true`

`hbase.coprocessor.region.classes`

Description: A comma-separated list of region observer or endpoint coprocessors that are loaded by default on all tables. For any override coprocessor method, these classes will

be called in order. After implementing your own Coprocessor, add it to HBase's classpath and add the fully qualified class name here. A coprocessor can also be loaded on demand by setting HTableDescriptor or the HBase shell.
Default: `(empty)`

`hbase.coprocessor.master.classes`

Description: A comma-separated list of org.apache.hadoop.hbase.coprocessor.MasterObserver coprocessors that are loaded by default on the active HMaster process. For any implemented coprocessor methods, the listed classes will be called in order. After implementing your own MasterObserver, just put it in HBase's classpath and add the fully qualified class name here.
Default: `(empty)`

`hbase.coprocessor.abortonerror`

Description: Set to true to cause the hosting server (master or regionserver) to abort if a coprocessor fails to load, fails to initialize, or throws an unexpected Throwable object. Setting this to false will allow the server to continue execution but the system wide state of the coprocessor in question will become inconsistent as it will be properly executing in only a subset of servers, so this is most useful for debugging only.
Default: `true`

`hbase.rest.port`

Description: The port for the HBase REST server.
Default: `8080`

`hbase.rest.readonly`

Description: Defines the mode the REST server will be started in. Possible values are: false: All HTTP methods are permitted - GET/PUT/POST/DELETE. true: Only the GET method is permitted.
Default: `false`

`hbase.rest.threads.max`

Description: The maximum number of threads of the REST server thread pool. Threads in the pool are reused to process REST requests. This controls the maximum number of requests processed concurrently. It may help to control the memory used by the REST server to avoid OOM issues. If the thread pool is full, incoming requests will be queued up

and wait for some free threads.
Default: `100`

`hbase.rest.threads.min`

Description: The minimum number of threads of the REST server thread pool. The thread pool always has at least these number of threads so the REST server is ready to serve incoming requests.
Default: `2`

`hbase.rest.support.proxyuser`

Description: Enables running the REST server to support proxy-user mode.
Default: `false`

`hbase.defaults.for.version.skip`

Description: Set to true to skip the 'hbase.defaults.for.version' check. Setting this to true can be useful in contexts other than the other side of a maven generation; i.e. running in an IDE. You'll want to set this boolean to true to avoid seeing the RuntimeException complaint: "hbase-default.xml file seems to be for and old version of HBase (${hbase.version}), this version is X.X.X-SNAPSHOT"
Default: `false`

`hbase.table.lock.enable`

Description: Set to true to enable locking the table in zookeeper for schema change operations. Table locking from master prevents concurrent schema modifications to corrupt table state.
Default: `true`

`hbase.table.max.rowsize`

Description: Maximum size of single row in bytes (default is 1 Gb) for Get'ting or Scan'ning without in-row scan flag set. If row size exceeds this limit RowTooBigException is thrown to client.
Default: `1073741824`

`hbase.thrift.minWorkerThreads`

Description: The "core size" of the thread pool. New threads are created on every connection until this many threads are created.

Default: `16`

`hbase.thrift.maxWorkerThreads`

Description: The maximum size of the thread pool. When the pending request queue overflows, new threads are created until their number reaches this number. After that, the server starts dropping connections.
Default: `1000`

`hbase.thrift.maxQueuedRequests`

Description: The maximum number of pending Thrift connections waiting in the queue. If there are no idle threads in the pool, the server queues requests. Only when the queue overflows, new threads are added, up to hbase.thrift.maxQueuedRequests threads.
Default: `1000`

`hbase.regionserver.thrift.framed`

Description: Use Thrift TFramedTransport on the server side. This is the recommended transport for thrift servers and requires a similar setting on the client side. Changing this to false will select the default transport, vulnerable to DoS when malformed requests are issued due to THRIFT-601.
Default: `false`

`hbase.regionserver.thrift.framed.max_frame_size_in_mb`

Description: Default frame size when using framed transport, in MB
Default: `2`

`hbase.regionserver.thrift.compact`

Description: Use Thrift TCompactProtocol binary serialization protocol.
Default: `false`

`hbase.rootdir.perms`

Description: FS Permissions for the root data subdirectory in a secure (kerberos) setup. When master starts, it creates the rootdir with this permissions or sets the permissions if it does not match.
Default: `700`

`hbase.wal.dir.perms`

Description: FS Permissions for the root WAL directory in a secure(kerberos) setup. When master starts, it creates the WAL dir with this permissions or sets the permissions if it does not match.
Default: `700`

`hbase.data.umask.enable`

Description: Enable, if true, that file permissions should be assigned to the files written by the regionserver
Default: `false`

`hbase.data.umask`

Description: File permissions that should be used to write data files when hbase.data.umask.enable is true
Default: `000`

`hbase.snapshot.enabled`

Description: Set to true to allow snapshots to be taken / restored / cloned.
Default: `true`

`hbase.snapshot.restore.take.failsafe.snapshot`

Description: Set to true to take a snapshot before the restore operation. The snapshot taken will be used in case of failure, to restore the previous state. At the end of the restore operation this snapshot will be deleted
Default: `true`

`hbase.snapshot.restore.failsafe.name`

Description: Name of the failsafe snapshot taken by the restore operation. You can use the {snapshot.name}, {table.name} and {restore.timestamp} variables to create a name based on what you are restoring.
Default: `hbase-failsafe-{snapshot.name}-{restore.timestamp}`

`hbase.snapshot.working.dir`

Description: Location where the snapshotting process will occur. The location of the completed snapshots will not change, but the temporary directory where the snapshot process occurs will be set to this location. This can be a separate filesystem than the root

directory, for performance increase purposes. See HBASE-21098 for more information
Default: `(empty)`

`hbase.server.compactchecker.interval.multiplier`

Description: The number that determines how often we scan to see if compaction is necessary. Normally, compactions are done after some events (such as memstore flush), but if region didn't receive a lot of writes for some time, or due to different compaction policies, it may be necessary to check it periodically. The interval between checks is hbase.server.compactchecker.interval.multiplier multiplied by hbase.server.thread.wakefrequency.
Default: `1000`

`hbase.lease.recovery.timeout`

Description: How long we wait on dfs lease recovery in total before giving up.
Default: `900000`

`hbase.lease.recovery.dfs.timeout`

Description: How long between dfs recover lease invocations. Should be larger than the sum of the time it takes for the namenode to issue a block recovery command as part of datanode; dfs.heartbeat.interval and the time it takes for the primary datanode, performing block recovery to timeout on a dead datanode; usually dfs.client.socket-timeout. See the end of HBASE-8389 for more.
Default: `64000`

`hbase.column.max.version`

Description: New column family descriptors will use this value as the default number of versions to keep.
Default: `1`

`dfs.client.read.shortcircuit`

Description: If set to true, this configuration parameter enables short-circuit local reads.
Default: `(empty)`

`dfs.domain.socket.path`

Description: This is a path to a UNIX domain socket that will be used for communication between the DataNode and local HDFS clients, if dfs.client.read.shortcircuit is set to true. If

the string "_PORT" is present in this path, it will be replaced by the TCP port of the DataNode. Be careful about permissions for the directory that hosts the shared domain socket; dfsclient will complain if open to other users than the HBase user.
Default: `(empty)`

`hbase.dfs.client.read.shortcircuit.buffer.size`

Description: If the DFSClient configuration dfs.client.read.shortcircuit.buffer.size is unset, we will use what is configured here as the short circuit read default direct byte buffer size. DFSClient native default is 1MB; HBase keeps its HDFS files open so number of file blocks * 1MB soon starts to add up and threaten OOME because of a shortage of direct memory. So, we set it down from the default. Make it > the default hbase block size set in the HColumnDescriptor which is usually 64k.
Default: `131072`

`hbase.regionserver.checksum.verify`

Description: If set to true (the default), HBase verifies the checksums for hfile blocks. HBase writes checksums inline with the data when it writes out hfiles. HDFS (as of this writing) writes checksums to a separate file than the data file necessitating extra seeks. Setting this flag saves some on i/o. Checksum verification by HDFS will be internally disabled on hfile streams when this flag is set. If the hbase-checksum verification fails, we will switch back to using HDFS checksums (so do not disable HDFS checksums! And besides this feature applies to hfiles only, not to WALs). If this parameter is set to false, then hbase will not verify any checksums, instead it will depend on checksum verification being done in the HDFS client.
Default: `true`

`hbase.hstore.bytes.per.checksum`

Description: Number of bytes in a newly created checksum chunk for HBase-level checksums in hfile blocks.
Default: `16384`

`hbase.hstore.checksum.algorithm`

Description: Name of an algorithm that is used to compute checksums. Possible values are NULL, CRC32, CRC32C.
Default: `CRC32C`

`hbase.client.scanner.max.result.size`

Description: Maximum number of bytes returned when calling a scanner's next method. Note that when a single row is larger than this limit the row is still returned completely. The default value is 2MB, which is good for 1ge networks. With faster and/or high latency networks this value should be increased.
Default: `2097152`

`hbase.server.scanner.max.result.size`

Description: Maximum number of bytes returned when calling a scanner's next method. Note that when a single row is larger than this limit the row is still returned completely. The default value is 100MB. This is a safety setting to protect the server from OOM situations.
Default: `104857600`

`hbase.status.published`

Description: This setting activates the publication by the master of the status of the region server. When a region server dies and its recovery starts, the master will push this information to the client application, to let them cut the connection immediately instead of waiting for a timeout.
Default: `false`

`hbase.status.publisher.class`

Description: Implementation of the status publication with a multicast message.
Default: `org.apache.hadoop.hbase.master.ClusterStatusPublisher$MulticastPublisher`

`hbase.status.listener.class`

Description: Implementation of the status listener with a multicast message.
Default: `org.apache.hadoop.hbase.client.ClusterStatusListener$MulticastListener`

`hbase.status.multicast.address.ip`

Description: Multicast address to use for the status publication by multicast.
Default: `226.1.1.3`

`hbase.status.multicast.address.port`

Description: Multicast port to use for the status publication by multicast.
Default: `16100`

`hbase.dynamic.jars.dir`

Description: The directory from which the custom filter JARs can be loaded dynamically by the region server without the need to restart. However, an already loaded filter/co-processor class would not be un-loaded. See HBASE-1936 for more details. Does not apply to coprocessors.
Default: `${hbase.rootdir}/lib`

`hbase.security.authentication`

Description: Controls whether or not secure authentication is enabled for HBase. Possible values are 'simple' (no authentication), and 'kerberos'.
Default: `simple`

`hbase.rest.filter.classes`

Description: Servlet filters for REST service.
Default: `org.apache.hadoop.hbase.rest.filter.GzipFilter`

`hbase.master.loadbalancer.class`

Description: Class used to execute the regions balancing when the period occurs. See the class comment for more on how it works http://hbase.apache.org/devapidocs/org/apache/hadoop/hbase/master/balancer/StochasticLoadBalancer.html It replaces the DefaultLoadBalancer as the default (since renamed as the SimpleLoadBalancer).
Default: `org.apache.hadoop.hbase.master.balancer.StochasticLoadBalancer`

`hbase.master.loadbalance.bytable`

Description: Factor Table name when the balancer runs. Default: false.
Default: `false`

`hbase.master.normalizer.class`

Description: Class used to execute the region normalization when the period occurs. See the class comment for more on how it works http://hbase.apache.org/devapidocs/org/apache/hadoop/hbase/master/normalizer/SimpleRegionNormalizer.html
Default: `org.apache.hadoop.hbase.master.normalizer.SimpleRegionNormalizer`

`hbase.rest.csrf.enabled`

Description: Set to true to enable protection against cross-site request forgery (CSRF)
Default: `false`

`hbase.rest-csrf.browser-useragents-regex`

Description: A comma-separated list of regular expressions used to match against an HTTP request's User-Agent header when protection against cross-site request forgery (CSRF) is enabled for REST server by setting hbase.rest.csrf.enabled to true. If the incoming User-Agent matches any of these regular expressions, then the request is considered to be sent by a browser, and therefore CSRF prevention is enforced. If the request's User-Agent does not match any of these regular expressions, then the request is considered to be sent by something other than a browser, such as scripted automation. In this case, CSRF is not a potential attack vector, so the prevention is not enforced. This helps achieve backwards-compatibility with existing automation that has not been updated to send the CSRF prevention header.
Default: `^Mozilla.*,^Opera.*`

`hbase.security.exec.permission.checks`

Description: If this setting is enabled and ACL based access control is active (the AccessController coprocessor is installed either as a system coprocessor or on a table as a table coprocessor) then you must grant all relevant users EXEC privilege if they require the ability to execute coprocessor endpoint calls. EXEC privilege, like any other permission, can be granted globally to a user, or to a user on a per table or per namespace basis. For more information on coprocessor endpoints, see the coprocessor section of the HBase online manual. For more information on granting or revoking permissions using the AccessController, see the security section of the HBase online manual.
Default: `false`

`hbase.procedure.regionserver.classes`

Description: A comma-separated list of org.apache.hadoop.hbase.procedure.RegionServerProcedureManager procedure managers that are loaded by default on the active HRegionServer process. The lifecycle methods (init/start/stop) will be called by the active HRegionServer process to perform the specific globally barriered procedure. After implementing your own RegionServerProcedureManager, just put it in HBase's classpath and add the fully qualified

class name here.
Default: `(empty)`

`hbase.procedure.master.classes`

Description: A comma-separated list of org.apache.hadoop.hbase.procedure.MasterProcedureManager procedure managers that are loaded by default on the active HMaster process. A procedure is identified by its signature and users can use the signature and an instant name to trigger an execution of a globally barriered procedure. After implementing your own MasterProcedureManager, just put it in HBase's classpath and add the fully qualified class name here.
Default: `(empty)`

`hbase.coordinated.state.manager.class`

Description: Fully qualified name of class implementing coordinated state manager.
Default: `org.apache.hadoop.hbase.coordination.ZkCoordinatedStateManager`

`hbase.regionserver.storefile.refresh.period`

Description: The period (in milliseconds) for refreshing the store files for the secondary regions. 0 means this feature is disabled. Secondary regions sees new files (from flushes and compactions) from primary once the secondary region refreshes the list of files in the region (there is no notification mechanism). But too frequent refreshes might cause extra Namenode pressure. If the files cannot be refreshed for longer than HFile TTL (hbase.master.hfilecleaner.ttl) the requests are rejected. Configuring HFile TTL to a larger value is also recommended with this setting.
Default: `0`

`hbase.region.replica.replication.enabled`

Description: Whether asynchronous WAL replication to the secondary region replicas is enabled or not. We have a separated implementation for replicating the WAL without using the general inter-cluster replication framework, so now we will not add any replication peers.
Default: `false`

`hbase.http.filter.initializers`

Description: A comma separated list of class names. Each class in the list must extend org.apache.hadoop.hbase.http.FilterInitializer. The corresponding Filter will be initialized.

Then, the Filter will be applied to all user facing jsp and servlet web pages. The ordering of the list defines the ordering of the filters. The default StaticUserWebFilter add a user principal as defined by the hbase.http.staticuser.user property.
Default: `org.apache.hadoop.hbase.http.lib.StaticUserWebFilter`

`hbase.security.visibility.mutations.checkauths`

Description: This property if enabled, will check whether the labels in the visibility expression are associated with the user issuing the mutation
Default: `false`

`hbase.http.max.threads`

Description: The maximum number of threads that the HTTP Server will create in its ThreadPool.
Default: `16`

`hbase.http.metrics.servlets`

Description: Comma separated list of servlet names to enable for metrics collection. Supported servlets are jmx, metrics, prometheus
Default: `jmx,metrics,prometheus`

`hbase.replication.rpc.codec`

Description: The codec that is to be used when replication is enabled so that the tags are also replicated. This is used along with HFileV3 which supports tags in them. If tags are not used or if the hfile version used is HFileV2 then KeyValueCodec can be used as the replication codec. Note that using KeyValueCodecWithTags for replication when there are no tags causes no harm.
Default: `org.apache.hadoop.hbase.codec.KeyValueCodecWithTags`

`hbase.replication.source.maxthreads`

Description: The maximum number of threads any replication source will use for shipping edits to the sinks in parallel. This also limits the number of chunks each replication batch is broken into. Larger values can improve the replication throughput between the master and slave clusters. The default of 10 will rarely need to be changed.
Default: `10`

`hbase.http.staticuser.user`

Description: The user name to filter as, on static web filters while rendering content. An example use is the HDFS web UI (user to be used for browsing files).
Default: `dr.stack`

`hbase.regionserver.handler.abort.on.error.percent`

Description: The percent of region server RPC threads failed to abort RS. -1 Disable aborting; 0 Abort if even a single handler has died; 0.x Abort only when this percent of handlers have died; 1 Abort only all of the handers have died.
Default: `0.5`

`hbase.mob.file.cache.size`

Description: Number of opened file handlers to cache. A larger value will benefit reads by providing more file handlers per mob file cache and would reduce frequent file opening and closing. However, if this is set too high, this could lead to a "too many opened file handlers" The default value is 1000.
Default: `1000`

`hbase.mob.cache.evict.period`

Description: The amount of time in seconds before the mob cache evicts cached mob files. The default value is 3600 seconds.
Default: `3600`

`hbase.mob.cache.evict.remain.ratio`

Description: The ratio (between 0.0 and 1.0) of files that remains cached after an eviction is triggered when the number of cached mob files exceeds the hbase.mob.file.cache.size. The default value is 0.5f.
Default: `0.5f`

`hbase.master.mob.cleaner.period`

Description: The period that MobFileCleanerChore runs. The unit is second. The default value is one day. The MOB file name uses only the date part of the file creation time in it. We use this time for deciding TTL expiry of the files. So the removal of TTL expired files might be delayed. The max delay might be 24 hrs.
Default: `86400`

`hbase.mob.major.compaction.region.batch.size`

Description: The max number of a MOB table regions that is allowed in a batch of the mob compaction. By setting this number to a custom value, users can control the overall effect of a major compaction of a large MOB-enabled table. Default is 0 - means no limit - all regions of a MOB table will be compacted at once

Default: `0`

`hbase.mob.compaction.chore.period`

Description: The period that MobCompactionChore runs. The unit is second. The default value is one week.

Default: `604800`

`hbase.snapshot.master.timeout.millis`

Description: Timeout for master for the snapshot procedure execution.

Default: `300000`

`hbase.snapshot.region.timeout`

Description: Timeout for regionservers to keep threads in snapshot request pool waiting.

Default: `300000`

`hbase.rpc.rows.warning.threshold`

Description: Number of rows in a batch operation above which a warning will be logged. If hbase.client.write.buffer.maxmutations is not set, this will be used as fallback for that setting.

Default: `5000`

`hbase.master.wait.on.service.seconds`

Description: Default is 5 minutes. Make it 30 seconds for tests. See HBASE-19794 for some context.

Default: `30`

`hbase.master.cleaner.snapshot.interval`

Description: Snapshot Cleanup chore interval in milliseconds. The cleanup thread keeps running at this interval to find all snapshots that are expired based on TTL and delete them.

Default: `1800000`

`hbase.master.snapshot.ttl`

Description: Default Snapshot TTL to be considered when the user does not specify TTL while creating snapshot. Default value 0 indicates FOREVERE - snapshot should not be automatically deleted until it is manually deleted
Default: `0`

`hbase.master.regions.recovery.check.interval`

Description: Regions Recovery Chore interval in milliseconds. This chore keeps running at this interval to find all regions with configurable max store file ref count and reopens them.
Default: `1200000`

`hbase.regions.recovery.store.file.ref.count`

Description: Very large number of ref count on a compacted store file indicates that it is a ref leak on that object(compacted store file). Such files can not be removed after it is invalidated via compaction. Only way to recover in such scenario is to reopen the region which can release all resources, like the refcount, leases, etc. This config represents Store files Ref Count threshold value considered for reopening regions. Any region with compacted store files ref count > this value would be eligible for reopening by master. Here, we get the max refCount among all refCounts on all compacted away store files that belong to a particular region. Default value -1 indicates this feature is turned off. Only positive integer value should be provided to enable this feature.
Default: `-1`

`hbase.regionserver.slowlog.ringbuffer.size`

Description: Default size of ringbuffer to be maintained by each RegionServer in order to store online slowlog responses. This is an in-memory ring buffer of requests that were judged to be too slow in addition to the responseTooSlow logging. The in-memory representation would be complete. For more details, please look into Doc Section: Get Slow Response Log from shell
Default: `256`

`hbase.regionserver.slowlog.buffer.enabled`

Description: Indicates whether RegionServers have ring buffer running for storing Online Slow logs in FIFO manner with limited entries. The size of the ring buffer is indicated by config: hbase.regionserver.slowlog.ringbuffer.size The default value is false, turn this on

and get latest slowlog responses with complete data.
Default: `false`

`hbase.regionserver.slowlog.systable.enabled`

Description: Should be enabled only if hbase.regionserver.slowlog.buffer.enabled is enabled. If enabled (true), all slow/large RPC logs would be persisted to system table hbase:slowlog (in addition to in-memory ring buffer at each RegionServer). The records are stored in increasing order of time. Operators can scan the table with various combination of ColumnValueFilter. More details are provided in the doc section: "Get Slow/Large Response Logs from System table hbase:slowlog"
Default: `false`

`hbase.master.metafixer.max.merge.count`

Description: Maximum regions to merge at a time when we fix overlaps noted in CJ consistency report, but avoid merging 100 regions in one go!
Default: `64`

`hbase.rpc.rows.size.threshold.reject`

Description: If value is true, RegionServer will abort batch requests of Put/Delete with number of rows in a batch operation exceeding threshold defined by value of config: hbase.rpc.rows.warning.threshold. The default value is false and hence, by default, only warning will be logged. This config should be turned on to prevent RegionServer from serving very large batch size of rows and this way we can improve CPU usages by discarding too large batch request.
Default: `false`

`hbase.namedqueue.provider.classes`

Description: Default values for NamedQueueService implementors. This comma separated full class names represent all implementors of NamedQueueService that we would like to be invoked by LogEvent handler service. One example of NamedQueue service is SlowLogQueueService which is used to store slow/large RPC logs in ringbuffer at each RegionServer. All implementors of NamedQueueService should be found under package: "org.apache.hadoop.hbase.namequeues.impl"
Default: `org.apache.hadoop.hbase.namequeues.impl.SlowLogQueueService,org.apache.hadoop.hbase.namequeues.impl.BalancerDecisionQueueService,org.apache.hadoop.hbase.namequeues.imp`

`l.BalancerRejectionQueueService,org.apache.hadoop.hbase.namequeues.WALEventTrackerQueueService`

`hbase.master.balancer.decision.buffer.enabled`

Description: Indicates whether active HMaster has ring buffer running for storing balancer decisions in FIFO manner with limited entries. The size of the ring buffer is indicated by config: hbase.master.balancer.decision.queue.size
Default: `false`

`hbase.master.balancer.rejection.buffer.enabled`

Description: Indicates whether active HMaster has ring buffer running for storing balancer rejection in FIFO manner with limited entries. The size of the ring buffer is indicated by config: hbase.master.balancer.rejection.queue.size
Default: `false`

`hbase.locality.inputstream.derive.enabled`

Description: If true, derive StoreFile locality metrics from the underlying DFSInputStream backing reads for that StoreFile. This value will update as the DFSInputStream's block locations are updated over time. Otherwise, locality is computed on StoreFile open, and cached until the StoreFile is closed.
Default: `false`

`hbase.locality.inputstream.derive.cache.period`

Description: If deriving StoreFile locality metrics from the underlying DFSInputStream, how long should the derived values be cached for. The derivation process may involve hitting the namenode, if the DFSInputStream's block list is incomplete.
Default: `60000`

hbase-env.sh

Set HBase environment variables in this file. Examples include options to pass the JVM on start of an HBase daemon such as heap size and garbage collector configs. You can also set configurations for HBase configuration, log directories, niceness, ssh options, where to locate process pid files, etc. Open the file at *conf/hbase-env.sh* and peruse its content. Each option is fairly well documented. Add your own environment variables here if you want them read by HBase daemons on startup.

Changes here will require a cluster restart for HBase to notice the change.

log4j2.properties

Since version 2.5.0, HBase has upgraded to Log4j2, so the configuration file name and format has changed. Read more in Apache Log4j2.

Edit this file to change rate at which HBase files are rolled and to change the level at which HBase logs messages.

Changes here will require a cluster restart for HBase to notice the change though log levels can be changed for particular daemons via the HBase UI.

Client configuration and dependencies connecting to an HBase cluster

If you are running HBase in standalone mode, you don't need to configure anything for your client to work provided that they are all on the same machine.

Starting release 3.0.0, the default connection registry has been switched to a rpc based implementation. Refer to Rpc Connection Registry (new as of 2.5.0). Depending on your HBase version, following is the expected minimal client configuration.

Up until 2.x.y releases

In 2.x.y releases, the default connection registry was based on ZooKeeper as the source of truth. This means that the clients always looked up ZooKeeper znodes to fetch the required metadata. For example, if an active master crashed and the a new master is elected, clients looked up the master znode to fetch the active master address (similarly for meta locations). This meant that the clients needed to have access to ZooKeeper and need to know the ZooKeeper ensemble information before they can do anything. This can be configured in the client configuration xml as follows:

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<configuration>
  <property>
    <name>hbase.zookeeper.quorum</name>
```

```
    <value>example1,example2,example3</value>
    <description>Zookeeper ensemble information</description>
  </property>
</configuration>
```

Starting from 3.0.0 release

The default implementation was switched to a rpc based connection registry. With this implementation, by default clients contact the active or stand-by master RPC end points to fetch the connection registry information. This means that the clients should have access to the list of active and master end points before they can do anything. This can be configured in the client configuration xml as follows:

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<configuration>
  <property>
    <name>hbase.masters</name>
    <value>example1,example2,example3</value>
    <description>List of master rpc end points for the hbase cluster.</descriptio
n>
  </property>
</configuration>
```

The configuration value for *hbase.masters* is a comma separated list of *host:port* values. If no port value is specified, the default of *16000* is assumed.

Of course you are free to specify bootstrap nodes other than masters, like:

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<property>
  <name>hbase.client.bootstrap.servers</name>
  <value>server1:16020,server2:16020,server3:16020</value>
</property>
```

The configuration value for *hbase.client.bootstrap.servers* is a comma separated list of *host:port* values. Notice that port must be specified here.

Usually these configurations are kept out in the *hbase-site.xml* and is picked up by the client from the `CLASSPATH`.

If you are configuring an IDE to run an HBase client, you should include the *conf/* directory on your classpath so *hbase-site.xml* settings can be found (or add *src/test/resources* to pick up the hbase-site.xml used by tests).

For Java applications using Maven, including the hbase-shaded-client module is the recommended dependency when connecting to a cluster:

```
<dependency>
  <groupId>org.apache.hbase</groupId>
  <artifactId>hbase-shaded-client</artifactId>
  <version>2.0.0</version>
</dependency>
```

Java client configuration

The configuration used by a Java client is kept in an HBaseConfiguration instance.

The factory method on HBaseConfiguration, `HBaseConfiguration.create();`, on invocation, will read in the content of the first *hbase-site.xml* found on the client's `CLASSPATH`, if one is present (Invocation will also factor in any *hbase-default.xml* found; an *hbase-default.xml* ships inside the *hbase.X.X.X.jar*). It is also possible to specify configuration directly without having to read from a *hbase-site.xml*.

For example, to set the ZooKeeper ensemble or bootstrap nodes for the cluster programmatically do as follows:

```
Configuration config = HBaseConfiguration.create();
config.set("hbase.zookeeper.quorum", "localhost");  // Until 2.x.y versions
// ---- or ----
config.set("hbase.client.bootstrap.servers", "localhost:1234"); // Starting 3.0.0
version
```

Timeout settings

HBase provides a wide variety of timeout settings to limit the execution time of various remote operations.

- hbase.rpc.timeout
- hbase.rpc.read.timeout

- hbase.rpc.write.timeout
- hbase.client.operation.timeout
- hbase.client.meta.operation.timeout
- hbase.client.scanner.timeout.period

The `hbase.rpc.timeout` property limits how long a single RPC call can run before timing out. To fine tune read or write related RPC timeouts set `hbase.rpc.read.timeout` and `hbase.rpc.write.timeout` configuration properties. In the absence of these properties `hbase.rpc.timeout` will be used.

A higher-level timeout is `hbase.client.operation.timeout` which is valid for each client call. When an RPC call fails for instance for a timeout due to `hbase.rpc.timeout` it will be retried until `hbase.client.operation.timeout` is reached. Client operation timeout for system tables can be fine tuned by setting `hbase.client.meta.operation.timeout` configuration value. When this is not set its value will use `hbase.client.operation.timeout`.

Timeout for scan operations is controlled differently. Use `hbase.client.scanner.timeout.period` property to set this timeout.

Example Configurations

Basic Distributed HBase Install

Here is a basic configuration example for a distributed ten node cluster: * The nodes are named `example0`, `example1`, etc., through node `example9` in this example. * The HBase Master and the HDFS NameNode are running on the node `example0`. * RegionServers run on nodes `example1`-`example9`. * A 3-node ZooKeeper ensemble runs on `example1`, `example2`, and `example3` on the default ports. * ZooKeeper data is persisted to the directory */export/zookeeper*.

Below we show what the main configuration files — *hbase-site.xml*, *regionservers*, and *hbase-env.sh* — found in the HBase *conf* directory might look like.

hbase-site.xml

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<configuration>
  <property>
    <name>hbase.zookeeper.quorum</name>
    <value>example1,example2,example3</value>
    <description>The directory shared by RegionServers.</description>
  </property>
  <property>
    <name>hbase.zookeeper.property.dataDir</name>
    <value>/export/zookeeper</value>
    <description>Property from ZooKeeper config zoo.cfg.
      The directory where the snapshot is stored.
    </description>
  </property>
  <property>
    <name>hbase.rootdir</name>
    <value>hdfs://example0:9000/hbase</value>
    <description>The directory shared by RegionServers.</description>
  </property>
  <property>
    <name>hbase.cluster.distributed</name>
    <value>true</value>
    <description>The mode the cluster will be in. Possible values are
      false: standalone and pseudo-distributed setups with managed ZooKeeper
      true: fully-distributed with unmanaged ZooKeeper Quorum (see hbase-env.sh)
    </description>
  </property>
</configuration>
```

regionservers

In this file you list the nodes that will run RegionServers. In our case, these nodes are `example1` - `example9`.

```
example1
example2
example3
example4
example5
example6
example7
example8
example9
```

hbase-env.sh

The following lines in the *hbase-env.sh* file show how to set the `JAVA_HOME` environment variable (required for HBase) and set the heap to 4 GB (rather than the default value of 1 GB). If you copy and paste this example, be sure to adjust the `JAVA_HOME` to suit your environment.

```
# The java implementation to use.
export JAVA_HOME=/usr/java/jdk1.8.0/

# The maximum amount of heap to use. Default is left to JVM default.
export HBASE_HEAPSIZE=4G
```

Use rsync to copy the content of the *conf* directory to all nodes of the cluster.

The Important Configurations

Required Configurations

Review the os and hadoop sections.

Big Cluster Configurations

If you have a cluster with a lot of regions, it is possible that a Regionserver checks in briefly after the Master starts while all the remaining RegionServers lag behind. This first server to check in will be assigned all regions which is not optimal. To prevent the above scenario from happening, up the `hbase.master.wait.on.regionservers.mintostart` property from its default value of 1. See HBASE-6389 Modify the conditions to ensure that Master waits for sufficient number of Region Servers before starting region assignments for more detail.

Recommended Configurations

ZooKeeper Configuration

`zookeeper.session.timeout`

The default timeout is 90 seconds (specified in milliseconds). This means that if a server crashes, it will be 90 seconds before the Master notices the crash and starts recovery. You might need to tune the timeout down to a minute or even less so the Master notices failures sooner. Before changing this value, be sure you have your JVM garbage collection configuration under control, otherwise, a long garbage collection that lasts beyond the ZooKeeper session timeout will take out your RegionServer. (You might be fine with this — you probably want recovery to start on the server if a RegionServer has been in GC for a long period of time).

To change this configuration, edit *hbase-site.xml*, copy the changed file across the cluster and restart.

We set this value high to save our having to field questions up on the mailing lists asking why a RegionServer went down during a massive import. The usual cause is that their JVM is untuned and they are running into long GC pauses. Our thinking is that while users are getting familiar with HBase, we'd save them having to know all of its intricacies. Later when they've built some confidence, then they can play with configuration such as this.

Number of ZooKeeper Instances

See zookeeper.

HDFS Configurations

`dfs.datanode.failed.volumes.tolerated`

This is the "...number of volumes that are allowed to fail before a DataNode stops offering service. By default, any volume failure will cause a datanode to shutdown" from the *hdfs-default.xml* description. You might want to set this to about half the amount of your available disks.

`hbase.regionserver.handler.count`

This setting defines the number of threads that are kept open to answer incoming requests to user tables. The rule of thumb is to keep this number low when the payload per request approaches the MB (big puts, scans using a large cache) and high when the payload is

small (gets, small puts, ICVs, deletes). The total size of the queries in progress is limited by the setting `hbase.ipc.server.max.callqueue.size`.

It is safe to set that number to the maximum number of incoming clients if their payload is small, the typical example being a cluster that serves a website since puts aren't typically buffered and most of the operations are gets.

The reason why it is dangerous to keep this setting high is that the aggregate size of all the puts that are currently happening in a region server may impose too much pressure on its memory, or even trigger an OutOfMemoryError. A RegionServer running on low memory will trigger its JVM's garbage collector to run more frequently up to a point where GC pauses become noticeable (the reason being that all the memory used to keep all the requests' payloads cannot be trashed, no matter how hard the garbage collector tries). After some time, the overall cluster throughput is affected since every request that hits that RegionServer will take longer, which exacerbates the problem even more.

You can get a sense of whether you have too little or too many handlers by rpc.logging on an individual RegionServer then tailing its logs (Queued requests consume memory).

Configuration for large memory machines

HBase ships with a reasonable, conservative configuration that will work on nearly all machine types that people might want to test with. If you have larger machines — HBase has 8G and larger heap — you might find the following configuration options helpful. TODO.

Compression

You should consider enabling ColumnFamily compression. There are several options that are near-frictionless and in most all cases boost performance by reducing the size of StoreFiles and thus reducing I/O.

See compression for more information.

Configuring the size and number of WAL files

HBase uses [wal]/docs/architecture/regionserver#write-ahead-log-wal to recover the memstore data that has not been flushed to disk in case of an RS failure. These WAL files

should be configured to be slightly smaller than HDFS block (by default a HDFS block is 64Mb and a WAL file is ~60Mb).

HBase also has a limit on the number of WAL files, designed to ensure there's never too much data that needs to be replayed during recovery. This limit needs to be set according to memstore configuration, so that all the necessary data would fit. It is recommended to allocate enough WAL files to store at least that much data (when all memstores are close to full). For example, with 16Gb RS heap, default memstore settings (0.4), and default WAL file size (~60Mb), 16Gb*0.4/60, the starting point for WAL file count is ~109. However, as all memstores are not expected to be full all the time, less WAL files can be allocated.

Managed Splitting

HBase generally handles splitting of your regions based upon the settings in your *hbase-default.xml* and *hbase-site.xml* configuration files. Important settings include `hbase.regionserver.region.split.policy`, `hbase.hregion.max.filesize`, `hbase.regionserver.regionSplitLimit`. A simplistic view of splitting is that when a region grows to `hbase.hregion.max.filesize`, it is split. For most usage patterns, you should use automatic splitting. See manual region splitting decisions for more information about manual region splitting.

Instead of allowing HBase to split your regions automatically, you can choose to manage the splitting yourself. Manually managing splits works if you know your keyspace well, otherwise let HBase figure where to split for you. Manual splitting can mitigate region creation and movement under load. It also makes it so region boundaries are known and invariant (if you disable region splitting). If you use manual splits, it is easier doing staggered, time-based major compactions to spread out your network IO load.

Disable Automatic Splitting

To disable automatic splitting, you can set region split policy in either cluster configuration or table configuration to be `org.apache.hadoop.hbase.regionserver.DisabledRegionSplitPolicy`

> If you disable automatic splits to diagnose a problem or during a period of fast data growth, it is recommended to re-enable them when your situation becomes more stable. The potential benefits of managing region splits yourself are not undisputed.

Determine the Optimal Number of Pre-Split Regions

The optimal number of pre-split regions depends on your application and environment. A good rule of thumb is to start with 10 pre-split regions per server and watch as data grows over time. It is better to err on the side of too few regions and perform rolling splits later. The optimal number of regions depends upon the largest StoreFile in your region. The size of the largest StoreFile will increase with time if the amount of data grows. The goal is for the largest region to be just large enough that the compaction selection algorithm only compacts it during a timed major compaction. Otherwise, the cluster can be prone to compaction storms with a large number of regions under compaction at the same time. It is important to understand that the data growth causes compaction storms and not the manual split decision.

If the regions are split into too many large regions, you can increase the major compaction interval by configuring `HConstants.MAJOR_COMPACTION_PERIOD`. The `org.apache.hadoop.hbase.util.RegionSplitter` utility also provides a network-IO-safe rolling split of all regions.

Managed Compactions

By default, major compactions are scheduled to run once in a 7-day period.

If you need to control exactly when and how often major compaction runs, you can disable managed major compactions. See the entry for `hbase.hregion.majorcompaction` in the compaction.parameters table for details.

> ⚠ Major compactions are absolutely necessary for StoreFile clean-up. Do not disable them altogether. You can run major compactions manually via the HBase shell or via the Admin API.

For more information about compactions and the compaction file selection process, see compaction

Speculative Execution

Speculative Execution of MapReduce tasks is on by default, and for HBase clusters it is generally advised to turn off Speculative Execution at a system-level unless you need it for a specific case, where it can be configured per-job. Set the properties `mapreduce.map.speculative` and `mapreduce.reduce.speculative` to false.

Other Configurations

Balancer

The balancer is a periodic operation which is run on the master to redistribute regions on the cluster. It is configured via `hbase.balancer.period` and defaults to 300000 (5 minutes).

See master.processes.loadbalancer for more information on the LoadBalancer.

Disabling Blockcache

Do not turn off block cache (You'd do it by setting `hfile.block.cache.size` to zero). Currently, we do not do well if you do this because the RegionServer will spend all its time loading HFile indices over and over again. If your working set is such that block cache does you no good, at least size the block cache such that HFile indices will stay up in the cache (you can get a rough idea on the size you need by surveying RegionServer UIs; you'll see index block size accounted near the top of the webpage).

Nagle's or the small package problem

If a big 40ms or so occasional delay is seen in operations against HBase, try the Nagle's setting. For example, see the user mailing list thread, Inconsistent scan performance with caching set to 1 and the issue cited therein where setting `notcpdelay` improved scan speeds. You might also see the graphs on the tail of HBASE-7008 Set scanner caching to a better default where our Lars Hofhansl tries various data sizes w/ Nagle's on and off measuring the effect.

Better Mean Time to Recover (MTTR)

This section is about configurations that will make servers come back faster after a fail. See the Deveraj Das and Nicolas Liochon blog post Introduction to HBase Mean Time to Recover (MTTR) for a brief introduction.

The issue HBASE-8354 forces Namenode into loop with lease recovery requests is messy but has a bunch of good discussion toward the end on low timeouts and how to cause faster recovery including citation of fixes added to HDFS. Read the Varun Sharma comments. The below suggested configurations are Varun's suggestions distilled and tested. Make sure you are running on a late-version HDFS so you have the fixes he refers

to and himself adds to HDFS that help HBase MTTR (e.g. HDFS-3703, HDFS-3712, and HDFS-4791 — Hadoop 2 for sure has them and late Hadoop 1 has some). Set the following in the RegionServer.

```
<property>
  <name>hbase.lease.recovery.dfs.timeout</name>
  <value>23000</value>
  <description>How much time we allow elapse between calls to recover lease.
  Should be larger than the dfs timeout.</description>
</property>
<property>
  <name>dfs.client.socket-timeout</name>
  <value>10000</value>
  <description>Down the DFS timeout from 60 to 10 seconds.</description>
</property>
```

And on the NameNode/DataNode side, set the following to enable 'staleness' introduced in HDFS-3703, HDFS-3912.

```
<property>
  <name>dfs.client.socket-timeout</name>
  <value>10000</value>
  <description>Down the DFS timeout from 60 to 10 seconds.</description>
</property>
<property>
  <name>dfs.datanode.socket.write.timeout</name>
  <value>10000</value>
  <description>Down the DFS timeout from 8 * 60 to 10 seconds.</description>
</property>
<property>
  <name>ipc.client.connect.timeout</name>
  <value>3000</value>
  <description>Down from 60 seconds to 3.</description>
</property>
<property>
  <name>ipc.client.connect.max.retries.on.timeouts</name>
  <value>2</value>
  <description>Down from 45 seconds to 3 (2 == 3 retries).</description>
</property>
<property>
  <name>dfs.namenode.avoid.read.stale.datanode</name>
  <value>true</value>
  <description>Enable stale state in hdfs</description>
</property>
<property>
  <name>dfs.namenode.stale.datanode.interval</name>
  <value>20000</value>
  <description>Down from default 30 seconds</description>
</property>
<property>
  <name>dfs.namenode.avoid.write.stale.datanode</name>
```

```
  <value>true</value>
  <description>Enable stale state in hdfs</description>
</property>
```

JMX

JMX (Java Management Extensions) provides built-in instrumentation that enables you to monitor and manage the Java VM. To enable monitoring and management from remote systems, you need to set system property `com.sun.management.jmxremote.port` (the port number through which you want to enable JMX RMI connections) when you start the Java VM. See the official documentation for more information. Historically, besides above port mentioned, JMX opens two additional random TCP listening ports, which could lead to port conflict problem. (See HBASE-10289 for details)

As an alternative, you can use the coprocessor-based JMX implementation provided by HBase. To enable it, add below property in *hbase-site.xml*:

```
<property>
  <name>hbase.coprocessor.regionserver.classes</name>
  <value>org.apache.hadoop.hbase.JMXListener</value>
</property>
```

DO NOT set `com.sun.management.jmxremote.port` for Java VM at the same time.

Currently it supports Master and RegionServer Java VM. By default, the JMX listens on TCP port 10102, you can further configure the port using below properties:

```
<property>
  <name>regionserver.rmi.registry.port</name>
  <value>61130</value>
</property>
<property>
  <name>regionserver.rmi.connector.port</name>
  <value>61140</value>
</property>
```

The registry port can be shared with connector port in most cases, so you only need to configure `regionserver.rmi.registry.port`. However, if you want to use SSL communication, the 2 ports must be configured to different values.

By default the password authentication and SSL communication is disabled. To enable password authentication, you need to update *hbase-env.sh* like below:

```
export HBASE_JMX_BASE="-Dcom.sun.management.jmxremote.authenticate=true
\
                       -Dcom.sun.management.jmxremote.password.file=your_password
_file   \
                       -Dcom.sun.management.jmxremote.access.file=your_access_fil
e"

export HBASE_MASTER_OPTS="$HBASE_MASTER_OPTS $HBASE_JMX_BASE "
export HBASE_REGIONSERVER_OPTS="$HBASE_REGIONSERVER_OPTS $HBASE_JMX_BASE "
```

See example password/access file under `$JRE_HOME/lib/management`.

To enable SSL communication with password authentication, follow below steps:

```
#1. generate a key pair, stored in myKeyStore
keytool -genkey -alias jconsole -keystore myKeyStore

#2. export it to file jconsole.cert
keytool -export -alias jconsole -keystore myKeyStore -file jconsole.cert

#3. copy jconsole.cert to jconsole client machine, import it to jconsoleKeyStore
keytool -import -alias jconsole -keystore jconsoleKeyStore -file jconsole.cert
```

And then update *hbase-env.sh* like below:

```
export HBASE_JMX_BASE="-Dcom.sun.management.jmxremote.ssl=true
\
                       -Djavax.net.ssl.keyStore=/home/tianq/myKeyStore
\
                       -Djavax.net.ssl.keyStorePassword=your_password_in_step_1
\
                       -Dcom.sun.management.jmxremote.authenticate=true
\
                       -Dcom.sun.management.jmxremote.password.file=your_password
file \
                       -Dcom.sun.management.jmxremote.access.file=your_access_fil
e"

export HBASE_MASTER_OPTS="$HBASE_MASTER_OPTS $HBASE_JMX_BASE "
export HBASE_REGIONSERVER_OPTS="$HBASE_REGIONSERVER_OPTS $HBASE_JMX_BASE "
```

Finally start `jconsole` on the client using the key store:

```
jconsole -J-Djavax.net.ssl.trustStore=/home/tianq/jconsoleKeyStore
```

> To enable the HBase JMX implementation on Master, you also need to add below property in *hbase-site.xml*:

```
<property>
  <name>hbase.coprocessor.master.classes</name>
  <value>org.apache.hadoop.hbase.JMXListener</value>
</property>
```

The corresponding properties for port configuration are `master.rmi.registry.port` (by default 10101) and `master.rmi.connector.port` (by default the same as registry.port)

Dynamic Configuration

It is possible to change a subset of the configuration without requiring a server restart. In the HBase shell, the operations `update_config`, `update_all_config` and `update_rsgroup_config` will prompt a server, all servers or all servers in the RSGroup to reload configuration.

Only a subset of all configurations can currently be changed in the running server. Here are those configurations:

Configurations that support dynamically change

Key
hbase.ipc.server.fallback-to-simple-auth-allowed
hbase.cleaner.scan.dir.concurrent.size
hbase.coprocessor.master.classes
hbase.coprocessor.region.classes
hbase.coprocessor.regionserver.classes
hbase.coprocessor.user.region.classes
hbase.regionserver.thread.compaction.large
hbase.regionserver.thread.compaction.small
hbase.regionserver.thread.split

Key
hbase.regionserver.throughput.controller
hbase.regionserver.thread.hfilecleaner.throttle
hbase.regionserver.hfilecleaner.large.queue.size
hbase.regionserver.hfilecleaner.small.queue.size
hbase.regionserver.hfilecleaner.large.thread.count
hbase.regionserver.hfilecleaner.small.thread.count
hbase.regionserver.hfilecleaner.thread.timeout.msec
hbase.regionserver.hfilecleaner.thread.check.interval.msec
hbase.regionserver.flush.throughput.controller
hbase.hstore.compaction.max.size
hbase.hstore.compaction.max.size.offpeak
hbase.hstore.compaction.min.size
hbase.hstore.compaction.min
hbase.hstore.compaction.max
hbase.hstore.compaction.ratio
hbase.hstore.compaction.ratio.offpeak
hbase.regionserver.thread.compaction.throttle
hbase.hregion.majorcompaction
hbase.hregion.majorcompaction.jitter
hbase.hstore.min.locality.to.skip.major.compact
hbase.hstore.compaction.date.tiered.max.storefile.age.millis
hbase.hstore.compaction.date.tiered.incoming.window.min
hbase.hstore.compaction.date.tiered.window.policy.class
hbase.hstore.compaction.date.tiered.single.output.for.minor.compaction
hbase.hstore.compaction.date.tiered.window.factory.class
hbase.offpeak.start.hour
hbase.offpeak.end.hour

Key
hbase.oldwals.cleaner.thread.size
hbase.oldwals.cleaner.thread.timeout.msec
hbase.oldwals.cleaner.thread.check.interval.msec
hbase.procedure.worker.keep.alive.time.msec
hbase.procedure.worker.add.stuck.percentage
hbase.procedure.worker.monitor.interval.msec
hbase.procedure.worker.stuck.threshold.msec
hbase.regions.slop
hbase.regions.overallSlop
hbase.balancer.tablesOnMaster
hbase.balancer.tablesOnMaster.systemTablesOnly
hbase.util.ip.to.rack.determiner
hbase.ipc.server.max.callqueue.length
hbase.ipc.server.priority.max.callqueue.length
hbase.ipc.server.callqueue.type
hbase.ipc.server.callqueue.codel.target.delay
hbase.ipc.server.callqueue.codel.interval
hbase.ipc.server.callqueue.codel.lifo.threshold
hbase.master.balancer.stochastic.maxSteps
hbase.master.balancer.stochastic.stepsPerRegion
hbase.master.balancer.stochastic.maxRunningTime
hbase.master.balancer.stochastic.runMaxSteps
hbase.master.balancer.stochastic.numRegionLoadsToRemember
hbase.master.loadbalance.bytable
hbase.master.balancer.stochastic.minCostNeedBalance
hbase.master.balancer.stochastic.localityCost
hbase.master.balancer.stochastic.rackLocalityCost

Key
hbase.master.balancer.stochastic.readRequestCost
hbase.master.balancer.stochastic.writeRequestCost
hbase.master.balancer.stochastic.memstoreSizeCost
hbase.master.balancer.stochastic.storefileSizeCost
hbase.master.balancer.stochastic.regionReplicaHostCostKey
hbase.master.balancer.stochastic.regionReplicaRackCostKey
hbase.master.balancer.stochastic.regionCountCost
hbase.master.balancer.stochastic.primaryRegionCountCost
hbase.master.balancer.stochastic.moveCost
hbase.master.balancer.stochastic.moveCost.offpeak
hbase.master.balancer.stochastic.maxMovePercent
hbase.master.balancer.stochastic.tableSkewCost
hbase.master.regions.recovery.check.interval
hbase.regions.recovery.store.file.ref.count
hbase.rsgroup.fallback.enable

Upgrading

You cannot skip major versions when upgrading. If you are upgrading from version 0.98.x to 2.x, you must first go from 0.98.x to 1.2.x and then go from 1.2.x to 2.x.

Review Apache HBase Configuration, in particular Hadoop. Familiarize yourself with Support and Testing Expectations.

HBase version number and compatibility

Aspirational Semantic Versioning

Starting with the 1.0.0 release, HBase is working towards Semantic Versioning for its release versioning. In summary:

Given a version number MAJOR.MINOR.PATCH, increment the:

- MAJOR version when you make incompatible API changes,
- MINOR version when you add functionality in a backwards-compatible manner, and
- PATCH version when you make backwards-compatible bug fixes.
- Additional labels for pre-release and build metadata are available as extensions to the MAJOR.MINOR.PATCH format.

Compatibility Dimensions

In addition to the usual API versioning considerations HBase has other compatibility dimensions that we need to consider.

Client-Server wire protocol compatibility

- Allows updating client and server out of sync.
- We could only allow upgrading the server first. I.e. the server would be backward compatible to an old client, that way new APIs are OK.
- Example: A user should be able to use an old client to connect to an upgraded cluster.

Server-Server protocol compatibility

- Servers of different versions can co-exist in the same cluster.
- The wire protocol between servers is compatible.

- Workers for distributed tasks, such as replication and log splitting, can co-exist in the same cluster.
- Dependent protocols (such as using ZK for coordination) will also not be changed.
- Example: A user can perform a rolling upgrade.

File format compatibility

- Support file formats backward and forward compatible
- Example: File, ZK encoding, directory layout is upgraded automatically as part of an HBase upgrade. User can downgrade to the older version and everything will continue to work.

Client API compatibility

- Allow changing or removing existing client APIs.
- An API needs to be deprecated for a whole major version before we will change/remove it.
  - An example: An API was deprecated in 2.0.1 and will be marked for deletion in 4.0.0. On the other hand, an API deprecated in 2.0.0 can be removed in 3.0.0.
  - Occasionally mistakes are made and internal classes are marked with a higher access level than they should. In these rare circumstances, we will accelerate the deprecation schedule to the next major version (i.e., deprecated in 2.2.x, marked `IA.Private` 3.0.0). Such changes are communicated and explained via release note in Jira.
- APIs available in a patch version will be available in all later patch versions. However, new APIs may be added which will not be available in earlier patch versions.
- New APIs introduced in a patch version will only be added in a source compatible way: i.e. code that implements public APIs will continue to compile. [1]
  - Example: A user using a newly deprecated API does not need to modify application code with HBase API calls until the next major version. *

Client Binary compatibility

- Client code written to APIs available in a given patch release can run unchanged (no recompilation needed) against the new jars of later patch versions.
- Client code written to APIs available in a given patch release might not run against the old jars from an earlier patch version.

    - Example: Old compiled client code will work unchanged with the new jars.
- If a Client implements an HBase Interface, a recompile MAY be required upgrading to a newer minor version (See release notes for warning about incompatible changes). All effort will be made to provide a default implementation so this case should not arise.

Server-Side Limited API compatibility (taken from Hadoop)

- Internal APIs are marked as Stable, Evolving, or Unstable
- This implies binary compatibility for coprocessors and plugins (pluggable classes, including replication) as long as these are only using marked interfaces/classes.
- Example: Old compiled Coprocessor, Filter, or Plugin code will work unchanged with the new jars.

Dependency Compatibility

- An upgrade of HBase will not require an incompatible upgrade of a dependent project, except for Apache Hadoop.
- An upgrade of HBase will not require an incompatible upgrade of the Java runtime.
- Example: Upgrading HBase to a version that supports *Dependency Compatibility* won't require that you upgrade your Apache ZooKeeper service.
- Example: If your current version of HBase supported running on JDK 8, then an upgrade to a version that supports *Dependency Compatibility* will also run on JDK 8.

> Previously, we tried to maintain dependency compatibility for the underly Hadoop service but over the last few years this has proven untenable. While the HBase project attempts to maintain support for older versions of Hadoop, we drop the "supported" designator for minor versions that fail to continue to see releases. Additionally, the Hadoop project has its own set of compatibility guidelines, which means in some cases having to update to a newer supported minor release might break some of our compatibility promises.

Operational Compatibility

- Metric changes
- Behavioral changes of services
- JMX APIs exposed via the `/jmx/` endpoint

Summary

- A patch upgrade is a drop-in replacement. Any change that is not Java binary and source compatible would not be allowed.[2] Downgrading versions within patch releases

may not be compatible.

- A minor upgrade requires no application/client code modification. Ideally it would be a drop-in replacement but client code, coprocessors, filters, etc might have to be recompiled if new jars are used.
- A major upgrade allows the HBase community to make breaking changes.

Compatibility Matrix: 3

	Major	Minor	Patch
Client-Server wire Compatibility	N	Y	Y
Server-Server Compatibility	N	Y	Y
File Format Compatibility	N [4]	Y	Y
Client API Compatibility	N	Y	Y
Client Binary Compatibility	N	N	Y
Server-Side Limited API Compatibility			
→ Stable	N	Y	Y
→ Evolving	N	N	Y
→ Unstable	N	N	N
Dependency Compatibility	N	Y	Y
Operational Compatibility	N	N	Y

⚠ HBase 1.7.0 release violated client-server wire compatibility guarantees and was subsequently withdrawn after the incompatibilities were reported and fixed in 1.7.1. If you are considering an upgrade to 1.7.x line, see Upgrading to 1.7.1+.

HBase API Surface

HBase has a lot of API points, but for the compatibility matrix above, we differentiate between Client API, Limited Private API, and Private API. HBase uses Apache Yetus Audience Annotations to guide downstream expectations for stability.

- InterfaceAudience (javadocs): captures the intended audience, possible values include:
  - Public: safe for end users and external projects
  - LimitedPrivate: used for internals we expect to be pluggable, such as coprocessors
  - Private: strictly for use within HBase itself Classes which are defined as `IA.Private` may be used as parameters or return values for interfaces which are declared `IA.LimitedPrivate`. Treat the `IA.Private` object as opaque; do not try to access its methods or fields directly.
- InterfaceStability (javadocs): describes what types of interface changes are permitted. Possible values include:
  - Stable: the interface is fixed and is not expected to change
  - Evolving: the interface may change in future minor versions
  - Unstable: the interface may change at any time

Please keep in mind the following interactions between the `InterfaceAudience` and `InterfaceStability` annotations within the HBase project:

- `IA.Public` classes are inherently stable and adhere to our stability guarantees relating to the type of upgrade (major, minor, or patch).
- `IA.LimitedPrivate` classes should always be annotated with one of the given `InterfaceStability` values. If they are not, you should presume they are `IS.Unstable`.
- `IA.Private` classes should be considered implicitly unstable, with no guarantee of stability between releases.

HBase Client API

HBase Client API consists of all the classes or methods that are marked with InterfaceAudience.Public interface. All main classes in hbase-client and dependent modules have either InterfaceAudience.Public, InterfaceAudience.LimitedPrivate, or

InterfaceAudience.Private marker. Not all classes in other modules (hbase-server, etc) have the marker. If a class is not annotated with one of these, it is assumed to be a InterfaceAudience.Private class.

HBase LimitedPrivate API

LimitedPrivate annotation comes with a set of target consumers for the interfaces. Those consumers are coprocessors, phoenix, replication endpoint implementations or similar. At this point, HBase only guarantees source and binary compatibility for these interfaces between patch versions.

HBase Private API

All classes annotated with InterfaceAudience.Private or all classes that do not have the annotation are for HBase internal use only. The interfaces and method signatures can change at any point in time. If you are relying on a particular interface that is marked Private, you should open a jira to propose changing the interface to be Public or LimitedPrivate, or an interface exposed for this purpose.

Binary Compatibility

When we say two HBase versions are compatible, we mean that the versions are wire and binary compatible. Compatible HBase versions means that clients can talk to compatible but differently versioned servers. It means too that you can just swap out the jars of one version and replace them with the jars of another, compatible version and all will just work. Unless otherwise specified, HBase point versions are (mostly) binary compatible. You can safely do rolling upgrades between binary compatible versions; i.e. across maintenance releases: e.g. from 1.4.4 to 1.4.6. See "Does compatibility between versions also mean binary compatibility?" discussion on the HBase dev mailing list.

Rolling Upgrades

A rolling upgrade is the process by which you update the servers in your cluster a server at a time. You can rolling upgrade across HBase versions if they are binary or wire compatible. See the "Rolling Upgrade Between Versions that are Binary/Wire Compatible" section below for more on what this means. Coarsely, a rolling upgrade is a graceful stop each server, update the software, and then restart. You do this for each server in the

cluster. Usually you upgrade the Master first and then the RegionServers. See Rolling Restart for tools that can help use the rolling upgrade process.

For example, in the below, HBase was symlinked to the actual HBase install. On upgrade, before running a rolling restart over the cluster, we changed the symlink to point at the new HBase software version and then ran

```
$ HADOOP_HOME=~/hadoop-2.6.0-CRC-SNAPSHOT ~/hbase/bin/rolling-restart.sh --config
~/conf_hbase
```

The rolling-restart script will first gracefully stop and restart the master, and then each of the RegionServers in turn. Because the symlink was changed, on restart the server will come up using the new HBase version. Check logs for errors as the rolling upgrade proceeds.

Rolling Upgrade Between Versions that are Binary/Wire Compatible

Unless otherwise specified, HBase minor versions are binary compatible. You can do a rolling upgrades between HBase point versions. For example, you can go to 1.4.4 from 1.4.6 by doing a rolling upgrade across the cluster replacing the 1.4.4 binary with a 1.4.6 binary.

In the minor version-particular sections below, we call out where the versions are wire/protocol compatible and in this case, it is also possible to do a rolling upgrade.

Rollback

Sometimes things don't go as planned when attempting an upgrade. This section explains how to perform a *rollback* to an earlier HBase release. Note that this should only be needed between Major and some Minor releases. You should always be able to *downgrade* between HBase Patch releases within the same Minor version. These instructions may require you to take steps before you start the upgrade process, so be sure to read through this section beforehand.

Caveats

Rollback vs Downgrade

This section describes how to perform a *rollback* on an upgrade between HBase minor and major versions. In this document, rollback refers to the process of taking an upgraded cluster and restoring it to the old version *while losing all changes that have occurred since upgrade*. By contrast, a cluster *downgrade* would restore an upgraded cluster to the old version while maintaining any data written since the upgrade. We currently only offer instructions to rollback HBase clusters. Further, rollback only works when these instructions are followed prior to performing the upgrade.

When these instructions talk about rollback vs downgrade of prerequisite cluster services (i.e. HDFS), you should treat leaving the service version the same as a degenerate case of downgrade.

Replication

Unless you are doing an all-service rollback, the HBase cluster will lose any configured peers for HBase replication. If your cluster is configured for HBase replication, then prior to following these instructions you should document all replication peers. After performing the rollback you should then add each documented peer back to the cluster. Note also that data written to the cluster since the upgrade may or may not have already been replicated to any peers. Determining which, if any, peers have seen replication data as well as rolling back the data in those peers is out of the scope of this guide.

Data Locality

Unless you are doing an all-service rollback, going through a rollback procedure will likely destroy all locality for Region Servers. You should expect degraded performance until after the cluster has had time to go through compactions to restore data locality. Optionally, you can force a compaction to speed this process up at the cost of generating cluster load.

Configurable Locations

The instructions below assume default locations for the HBase data directory and the HBase znode. Both of these locations are configurable and you should verify the value used in your cluster before proceeding. In the event that you have a different value, just

replace the default with the one found in your configuration * HBase data directory is configured via the key 'hbase.rootdir' and has a default value of '/hbase'. * HBase znode is configured via the key 'zookeeper.znode.parent' and has a default value of '/hbase'.

All service rollback

If you will be performing a rollback of both the HDFS and ZooKeeper services, then HBase's data will be rolled back in the process.

Requirements

- Ability to rollback HDFS and ZooKeeper

Before upgrade

No additional steps are needed pre-upgrade. As an extra precautionary measure, you may wish to use distcp to back up the HBase data off of the cluster to be upgraded. To do so, follow the steps in the 'Before upgrade' section of 'Rollback after HDFS downgrade' but copy to another HDFS instance instead of within the same instance.

Performing a rollback

1. Stop HBase
2. Perform a rollback for HDFS and ZooKeeper (HBase should remain stopped)
3. Change the installed version of HBase to the previous version
4. Start HBase
5. Verify HBase contents — use the HBase shell to list tables and scan some known values.

Rollback after HDFS rollback and ZooKeeper downgrade

If you will be rolling back HDFS but going through a ZooKeeper downgrade, then HBase will be in an inconsistent state. You must ensure the cluster is not started until you

complete this process.

Requirements

- Ability to rollback HDFS
- Ability to downgrade ZooKeeper

Before upgrade

No additional steps are needed pre-upgrade. As an extra precautionary measure, you may wish to use distcp to back up the HBase data off of the cluster to be upgraded. To do so, follow the steps in the 'Before upgrade' section of 'Rollback after HDFS downgrade' but copy to another HDFS instance instead of within the same instance.

Performing a rollback

1. Stop HBase
2. Perform a rollback for HDFS and a downgrade for ZooKeeper (HBase should remain stopped)
3. Change the installed version of HBase to the previous version
4. Clean out ZooKeeper information related to HBase. WARNING: This step will permanently destroy all replication peers. Please see the section on HBase Replication under Caveats for more information.

   Clean HBase information out of ZooKeeper:

   ```
   [hpnewton@gateway_node.example.com ~]$ zookeeper-client -server zookeeper1.e
   xample.com:2181,zookeeper2.example.com:2181,zookeeper3.example.com:2181
   Welcome to ZooKeeper!
   JLine support is disabled
   rmr /hbase
   quit
   Quitting...
   ```

5. Start HBase
6. Verify HBase contents—use the HBase shell to list tables and scan some known values.

Rollback after HDFS downgrade

If you will be performing an HDFS downgrade, then you'll need to follow these instructions regardless of whether ZooKeeper goes through rollback, downgrade, or reinstallation.

Requirements

- Ability to downgrade HDFS
- Pre-upgrade cluster must be able to run MapReduce jobs
- HDFS super user access
- Sufficient space in HDFS for at least two copies of the HBase data directory

Before upgrade

Before beginning the upgrade process, you must take a complete backup of HBase's backing data. The following instructions cover backing up the data within the current HDFS instance. Alternatively, you can use the distcp command to copy the data to another HDFS cluster.

1. Stop the HBase cluster
2. Copy the HBase data directory to a backup location using the distcp command as the HDFS super user (shown below on a security enabled cluster)

   Using distcp to backup the HBase data directory:

   ```
   [hpnewton@gateway_node.example.com ~]$ kinit -k -t hdfs.keytab hdfs@EXAMPLE.
   COM
   [hpnewton@gateway_node.example.com ~]$ hadoop distcp /hbase /hbase-pre-upgra
   de-backup
   ```

3. Distcp will launch a mapreduce job to handle copying the files in a distributed fashion. Check the output of the distcp command to ensure this job completed successfully.

Performing a rollback

1. Stop HBase

2. Perform a downgrade for HDFS and a downgrade/rollback for ZooKeeper (HBase should remain stopped)

3. Change the installed version of HBase to the previous version

4. Restore the HBase data directory from prior to the upgrade. Restore the HBase data directory from prior to the upgrade as the HDFS super user (shown below on a security enabled cluster). If you backed up your data on another HDFS cluster instead of locally, you will need to use the distcp command to copy it back to the current HDFS cluster.

   Restore the HBase data directory:

```
[hpnewton@gateway_node.example.com ~]$ kinit -k -t hdfs.keytab hdfs@EXAMPLE.
COM
[hpnewton@gateway_node.example.com ~]$ hdfs dfs -mv /hbase /hbase-upgrade-ro
llback
[hpnewton@gateway_node.example.com ~]$ hdfs dfs -mv /hbase-pre-upgrade-backu
p /hbase
```

5. Clean out ZooKeeper information related to HBase. WARNING: This step will permanently destroy all replication peers. Please see the section on HBase Replication under Caveats for more information.

   Clean HBase information out of ZooKeeper:

```
[hpnewton@gateway_node.example.com ~]$ zookeeper-client -server zookeeper1.e
xample.com:2181,zookeeper2.example.com:2181,zookeeper3.example.com:2181
Welcome to ZooKeeper!
JLine support is disabled
rmr /hbase
quit
Quitting...
```

6. Start HBase

7. Verify HBase contents–use the HBase shell to list tables and scan some known values.

Upgrade Paths

Upgrade from 2.x to 3.x

The RegionServer Grouping feature has been reimplemented. See section Migrating From Old Implementation in Apache HBase Operational Management for more details.

The `hbase:namespace` table has been removed and fold into `hbase:meta`. See Data Model for more details.

There is no special consideration upgrading to hbase-2.4.x from 2.3.x. And for earlier versions, just follow the Upgrade from 2.0.x-2.2.x to 2.3+ guide. In general, 2.2.x should be rolling upgradeable, for 2.1.x or 2.0.x, you will need to clear the Upgrade from 2.0 or 2.1 to 2.2+ hurdle first.

Upgrade from 2.0.x-2.2.x to 2.3+

There is no special consideration upgrading to hbase-2.3.x from earlier versions. From 2.2.x, it should be rolling upgradeable. From 2.1.x or 2.0.x, you will need to clear the Upgrade from 2.0 or 2.1 to 2.2+ hurdle first.

Upgraded ZooKeeper Dependency Version

Our dependency on Apache ZooKeeper has been upgraded to 3.5.7 (HBASE-24132), as 3.4.x is EOL. The newer 3.5.x client is compatible with the older 3.4.x server. However, if you're using HBase in stand-alone mode and perform an in-place upgrade, there are some upgrade steps documented by the ZooKeeper community. This doesn't impact a production deployment, but would impact a developer's local environment.

New In-Master Procedure Store

Of note, HBase 2.3.0 changes the in-Master Procedure Store implementation. It was a dedicated custom store to instead use a standard HBase Region (HBASE-23326). The migration from the old to new format is automatic run by the new 2.3.0 Master on startup. The old *MasterProcWALs* dir which hosted the old custom implementation files in *${hbase.rootdir}* is deleted on successful migration. A new *MasterProc* sub-directory replaces it to host the Store files and WALs for the new Procedure Store in-Master Region.

The in-Master Region is unusual in that it writes to an alternate location at *${hbase.rootdir}/MasterProc* rather than under *${hbase.rootdir}/data* in the filesystem and the special Procedure Store in-Master Region is hidden from all clients other than the active Master itself. Otherwise, it is like any other with the Master process running flushes and compactions, archiving WALs when over-flushed, and so on. Its files are readable by standard Region and Store file tooling for triage and analysis as long as they are pointed to the appropriate location in the filesystem.

Notice that, after the migration, you should make sure to not start an active master with old code, as it can not recognize the new procedure store. So it is suggested to upgrade backup master(s) to new 2.3 first, and then upgrade the active master. And unless explicitly mentioned, this is the suggested way for all upgrading, i.e, upgrading backup master(s) first, then active master, and then region servers.

Upgrade from 2.0 or 2.1 to 2.2+

HBase 2.2+ uses a new Procedure form assigning/unassigning/moving Regions. It does not process HBase 2.1 and 2.0's Unassign/Assign Procedure types. Upgrade requires that we first drain the Master Procedure Store of old style Procedures before starting the new 2.2 Master. So you need to make sure that before you kill the old version (2.0 or 2.1) Master, there is no region in transition. And once the new version (2.2+) Master is up, you can rolling upgrade RegionServers one by one.

And there is a more safer way if you are running 2.1.1+ or 2.0.3+ cluster. It need four steps to upgrade Master.

1. Shutdown both active and standby Masters (Your cluster will continue to server reads and writes without interruption).
2. Set the property hbase.procedure.upgrade-to-2-2 to true in hbase-site.xml for the Master, and start only one Master, still using the 2.1.1+ (or 2.0.3+) version.
3. Wait until the Master quits. Confirm that there is a 'UPGRADE OK: All existed procedures have been finished, quit...' message in the Master log as the cause of the shutdown. The Procedure Store is now empty.
4. Start new Masters with the new 2.2+ version.

Then you can rolling upgrade RegionServers one by one. See HBASE-21075 for more details.

In case these steps are not done, on starting 2.2+ master, you would see the following exception in the master logs:

```
org.apache.hadoop.hbase.HBaseIOException: Unsupported procedure type class org.apache.h
adoop.hbase.master.assignment.UnassignProcedure found
```

Upgrading from 1.x to 2.x

In this section we will first call out significant changes compared to the prior stable HBase release and then go over the upgrade process. Be sure to read the former with care so you avoid surprises.

Changes of Note!

First we'll cover deployment / operational changes that you might hit when upgrading to HBase 2.0+. After that we'll call out changes for downstream applications. Please note that Coprocessors are covered in the operational section. Also note that this section is not meant to convey information about new features that may be of interest to you. For a complete summary of changes, please see the CHANGES.txt file in the source release artifact for the version you are planning to upgrade to.

Update to basic prerequisite minimums in HBase 2.0+

As noted in the Basic Prerequisites section, HBase 2.0+ requires a minimum of Java 8 and Hadoop 2.6. The HBase community recommends ensuring you have already completed any needed upgrades in prerequisites prior to upgrading your HBase version.

HBCK must match HBase server version

You **must not** use an HBase 1.x version of HBCK against an HBase 2.0+ cluster. HBCK is strongly tied to the HBase server version. Using the HBCK tool from an earlier release against an HBase 2.0+ cluster will destructively alter said cluster in unrecoverable ways.

As of HBase 2.0, HBCK (A.K.A *HBCK1* or *hbck1*) is a read-only tool that can report the status of some non-public system internals but will often misread state because it does not understand the workings of hbase2.

To read about HBCK's replacement, see Operations and Management documentation.

⚠ Related, before you upgrade, ensure that *hbck1* reports no `INCONSISTENCIES`. Fixing hbase1-type inconsistencies post-upgrade is an involved process.

Configuration settings no longer in HBase 2.0+

The following configuration settings are no longer applicable or available. For details, please see the detailed release notes.

- hbase.config.read.zookeeper.config (see upgrade2.0.zkconfig for migration details)
- hbase.zookeeper.useMulti (HBase now always uses ZK's multi functionality)
- hbase.rpc.client.threads.max
- hbase.rpc.client.nativetransport
- hbase.fs.tmp.dir
- hbase.bucketcache.combinedcache.enabled
- hbase.bucketcache.ioengine no longer supports the 'heap' value.
- hbase.bulkload.staging.dir
- hbase.balancer.tablesOnMaster wasn't removed, strictly speaking, but its meaning has fundamentally changed and users should not set it. See the section upgrade2.0.regions.on.master for details.
- hbase.master.distributed.log.replay See the section upgrade2.0.distributed.log.replay for details
- hbase.regionserver.disallow.writes.when.recovering See the section upgrade2.0.distributed.log.replay for details
- hbase.regionserver.wal.logreplay.batch.size See the section upgrade2.0.distributed.log.replay for details
- hbase.master.catalog.timeout
- hbase.regionserver.catalog.timeout
- hbase.metrics.exposeOperationTimes
- hbase.metrics.showTableName
- hbase.online.schema.update.enable (HBase now always supports this)
- hbase.thrift.htablepool.size.max

Configuration properties that were renamed in HBase 2.0+

The following properties have been renamed. Attempts to set the old property will be ignored at run time.

Old name	New name
hbase.rpc.server.nativetransport	hbase.netty.nativetransport
hbase.netty.rpc.server.worker.count	hbase.netty.worker.count
hbase.hfile.compactions.discharger.interval	hbase.hfile.compaction.discharger.interval
hbase.hregion.percolumnfamilyflush.size.lower.bound	hbase.hregion.percolumnfamilyflush.size.lower.bound.min

Configuration settings with different defaults in HBase 2.0+

The following configuration settings changed their default value. Where applicable, the value to set to restore the behavior of HBase 1.2 is given.

- hbase.security.authorization now defaults to false. set to true to restore same behavior as previous default.
- hbase.client.retries.number is now set to 10. Previously it was 35. Downstream users are advised to use client timeouts as described in Configuration instead.
- hbase.client.serverside.retries.multiplier is now set to 3. Previously it was 10. Downstream users are advised to use client timeouts as described in Configuration instead.
- hbase.master.fileSplitTimeout is now set to 10 minutes. Previously it was 30 seconds.
- hbase.regionserver.logroll.multiplier is now set to 0.5. Previously it was 0.95. This change is tied with the following doubling of block size. Combined, these two configuration changes should make for WALs of about the same size as those in hbase-1.x but there should be less incidence of small blocks because we fail to roll the WAL before we hit the blocksize threshold. See HBASE-19148 for discussion.
- hbase.regionserver.hlog.blocksize defaults to 2x the HDFS default block size for the WAL dir. Previously it was equal to the HDFS default block size for the WAL dir.
- hbase.client.start.log.errors.counter changed to 5. Previously it was 9.
- hbase.ipc.server.callqueue.type changed to 'fifo'. In HBase versions 1.0 - 1.2 it was 'deadline'. In prior and later 1.x versions it already defaults to 'fifo'.

- hbase.hregion.memstore.chunkpool.maxsize is 1.0 by default. Previously it was 0.0. Effectively, this means previously we would not use a chunk pool when our memstore is onheap and now we will. See the section Long GC pauses for more information about the MSLAB chunk pool.
- hbase.master.cleaner.interval is now set to 10 minutes. Previously it was 1 minute.
- hbase.master.procedure.threads will now default to 1/4 of the number of available CPUs, but not less than 16 threads. Previously it would be number of threads equal to number of CPUs.
- hbase.hstore.blockingStoreFiles is now 16. Previously it was 10.
- hbase.http.max.threads is now 16. Previously it was 10.
- hbase.client.max.perserver.tasks is now 2. Previously it was 5.
- hbase.normalizer.period is now 5 minutes. Previously it was 30 minutes.
- hbase.regionserver.region.split.policy is now SteppingSplitPolicy. Previously it was IncreasingToUpperBoundRegionSplitPolicy.
- replication.source.ratio is now 0.5. Previously it was 0.1.

"Master hosting regions" feature broken and unsupported

The feature "Master acts as region server" and associated follow-on work available in HBase 1.y is non-functional in HBase 2.y and should not be used in a production setting due to deadlock on Master initialization. Downstream users are advised to treat related configuration settings as experimental and the feature as inappropriate for production settings.

A brief summary of related changes:

- Master no longer carries regions by default
- hbase.balancer.tablesOnMaster is a boolean, default false (if it holds an HBase 1.x list of tables, will default to false)
- hbase.balancer.tablesOnMaster.systemTablesOnly is boolean to keep user tables off master. default false
- those wishing to replicate old list-of-servers config should deploy a stand-alone RegionServer process and then rely on Region Server Groups

"Distributed Log Replay" feature broken and removed

The Distributed Log Replay feature was broken and has been removed from HBase 2.y+. As a consequence all related configs, metrics, RPC fields, and logging have also been removed. Note that this feature was found to be unreliable in the run up to HBase 1.0, defaulted to being unused, and was effectively removed in HBase 1.2.0 when we started ignoring the config that turns it on (HBASE-14465). If you are currently using the feature, be sure to perform a clean shutdown, ensure all DLR work is complete, and disable the feature prior to upgrading.

prefix-tree encoding removed

The prefix-tree encoding was removed from HBase 2.0.0 (HBASE-19179). It was (late!) deprecated in hbase-1.2.7, hbase-1.4.0, and hbase-1.3.2.

This feature was removed because it as not being actively maintained. If interested in reviving this sweet facility which improved random read latencies at the expensive of slowed writes, write the HBase developers list at *dev at hbase dot apache dot org*.

The prefix-tree encoding needs to be removed from all tables before upgrading to HBase 2.0+. To do that first you need to change the encoding from PREFIX_TREE to something else that is supported in HBase 2.0. After that you have to major compact the tables that were using PREFIX_TREE encoding before. To check which column families are using incompatible data block encoding you can use Pre-Upgrade Validator.

Changed metrics

The following metrics have changed names:

- Metrics previously published under the name "AssignmentManger" [sic] are now published under the name "AssignmentManager"

The following metrics have changed their meaning:

- The metric 'blockCacheEvictionCount' published on a per-region server basis no longer includes blocks removed from the cache due to the invalidation of the hfiles they are from (e.g. via compaction).
- The metric 'totalRequestCount' increments once per request; previously it incremented by the number of `Actions` carried in the request; e.g. if a request was a `multi` made of four Gets and two Puts, we'd increment 'totalRequestCount' by six; now we increment by one regardless. Expect to see lower values for this metric in hbase-2.0.0.

- The 'readRequestCount' now counts reads that return a non-empty row where in older hbases, we'd increment 'readRequestCount' whether a Result or not. This change will flatten the profile of the read-requests graphs if requests for non-existent rows. A YCSB read-heavy workload can do this dependent on how the database was loaded.

The following metrics have been removed:

- Metrics related to the Distributed Log Replay feature are no longer present. They were previously found in the region server context under the name 'replay'. See the section "Distributed Log Replay" feature broken and removed for details.

The following metrics have been added:

- 'totalRowActionRequestCount' is a count of region row actions summing reads and writes.

Changed logging

HBase-2.0.0 now uses slf4j as its logging frontend. Previously, we used log4j (1.2). For most the transition should be seamless; slf4j does a good job interpreting *log4j.properties* logging configuration files such that you should not notice any difference in your log system emissions.

That said, your *log4j.properties* may need freshening. See HBASE-20351 for example, where a stale log configuration file manifest as netty configuration being dumped at DEBUG level as preamble on every shell command invocation.

ZooKeeper configs no longer read from zoo.cfg

HBase no longer optionally reads the 'zoo.cfg' file for ZooKeeper related configuration settings. If you previously relied on the 'hbase.config.read.zookeeper.config' config for this functionality, you should migrate any needed settings to the hbase-site.xml file while adding the prefix 'hbase.zookeeper.property.' to each property name.

Changes in permissions

The following permission related changes either altered semantics or defaults:

- Permissions granted to a user now merge with existing permissions for that user, rather than over-writing them. (see the release note on HBASE-17472 for details)
- Region Server Group commands (added in 1.4.0) now require admin privileges.

Most Admin APIs don't work against an HBase 2.0+ cluster from pre-HBase 2.0 clients

A number of admin commands are known to not work when used from a pre-HBase 2.0 client. This includes an HBase Shell that has the library jars from pre-HBase 2.0. You will need to plan for an outage of use of admin APIs and commands until you can also update to the needed client version.

The following client operations do not work against HBase 2.0+ cluster when executed from a pre-HBase 2.0 client:

- list_procedures
- split
- merge_region
- list_quotas
- enable_table_replication
- disable_table_replication
- Snapshot related commands

Deprecated in 1.0 admin commands have been removed.

The following commands that were deprecated in 1.0 have been removed. Where applicable the replacement command is listed.

- The 'hlog' command has been removed. Downstream users should rely on the 'wal' command instead.

Region Server memory consumption changes.

Users upgrading from versions prior to HBase 1.4 should read the instructions in section Region Server memory consumption changes..

Additionally, HBase 2.0 has changed how memstore memory is tracked for flushing decisions. Previously, both the data size and overhead for storage were used to calculate utilization against the flush threshold. Now, only data size is used to make these per-region decisions. Globally the addition of the storage overhead is used to make decisions about forced flushes.

Web UI for splitting and merging operate on row prefixes

Previously, the Web UI included functionality on table status pages to merge or split based on an encoded region name. In HBase 2.0, instead this functionality works by taking a row prefix.

Special upgrading for Replication users from pre-HBase 1.4

User running versions of HBase prior to the 1.4.0 release that make use of replication should be sure to read the instructions in the section Replication peer's TableCFs config.

HBase shell changes

The HBase shell command relies on a bundled JRuby instance. This bundled JRuby been updated from version 1.6.8 to version 9.1.10.0. The represents a change from Ruby 1.8 to Ruby 2.3.3, which introduces non-compatible language changes for user scripts.

The HBase shell command now ignores the '--return-values' flag that was present in early HBase 1.4 releases. Instead the shell always behaves as though that flag were passed. If you wish to avoid having expression results printed in the console you should alter your IRB configuration.

Coprocessor APIs have changed in HBase 2.0+

All Coprocessor APIs have been refactored to improve supportability around binary API compatibility for future versions of HBase. If you or applications you rely on have custom HBase coprocessors, you should read the release notes for HBASE-18169 for details of changes you will need to make prior to upgrading to HBase 2.0+.

For example, if you had a BaseRegionObserver in HBase 1.2 then at a minimum you will need to update it to implement both RegionObserver and RegionCoprocessor and add the method

```
...
  @Override
  public Optional<RegionObserver> getRegionObserver() {
    return Optional.of(this);
  }
...
```

For more information, see Upgrading Coprocessors to 2.0.

HBase 2.0+ can no longer write HFile v2 files.

HBase has simplified our internal HFile handling. As a result, we can no longer write HFile versions earlier than the default of version 3. Upgrading users should ensure that hfile.format.version is not set to 2 in hbase-site.xml before upgrading. Failing to do so will cause Region Server failure. HBase can still read HFiles written in the older version 2 format.

HBase 2.0+ can no longer read Sequence File based WAL file.

HBase can no longer read the deprecated WAL files written in the Apache Hadoop Sequence File format. The hbase.regionserver.hlog.reader.impl and hbase.regionserver.hlog.writer.impl configuration entries should be set to use the Protobuf based WAL reader / writer classes. This implementation has been the default since HBase 0.96, so legacy WAL files should not be a concern for most downstream users.

Starting from 2.6.0, the hbase.regionserver.hlog.reader.impl and hbase.regionserver.hlog.writer.impl configuration entries are removed since the only valid values are protobuf based reader/writer. Setting them in *hbase-site.xml* will have no real effect.

A clean cluster shutdown should ensure there are no WAL files. If you are unsure of a given WAL file's format you can use the `hbase wal` command to parse files while the HBase cluster is offline. In HBase 2.0+, this command will not be able to read a Sequence File based WAL.

Change in behavior for filters

The Filter ReturnCode NEXT_ROW has been redefined as skipping to next row in current family, not to next row in all family. it's more reasonable, because ReturnCode is a concept in store level, not in region level.

Downstream HBase 2.0+ users should use the shaded client

Downstream users are strongly urged to rely on the Maven coordinates org.apache.hbase:hbase-shaded-client for their runtime use. This artifact contains all the needed implementation details for talking to an HBase cluster while minimizing the number of third party dependencies exposed.

Note that this artifact exposes some classes in the org.apache.hadoop package space (e.g. o.a.h.configuration.Configuration) so that we can maintain source compatibility with our public API. Those classes are included so that they can be altered to use the same

relocated third party dependencies as the rest of the HBase client code. In the event that you need to **also** use Hadoop in your code, you should ensure all Hadoop related jars precede the HBase client jar in your classpath.

Downstream HBase 2.0+ users of MapReduce must switch to new artifact

Downstream users of HBase's integration for Apache Hadoop MapReduce must switch to relying on the org.apache.hbase:hbase-shaded-mapreduce module for their runtime use. Historically, downstream users relied on either the org.apache.hbase:hbase-server or org.apache.hbase:hbase-shaded-server artifacts for these classes. Both uses are no longer supported and in the vast majority of cases will fail at runtime.

Note that this artifact exposes some classes in the org.apache.hadoop package space (e.g. o.a.h.configuration.Configuration) so that we can maintain source compatibility with our public API. Those classes are included so that they can be altered to use the same relocated third party dependencies as the rest of the HBase client code. In the event that you need to **also** use Hadoop in your code, you should ensure all Hadoop related jars precede the HBase client jar in your classpath.

Significant changes to runtime classpath

A number of internal dependencies for HBase were updated or removed from the runtime classpath. Downstream client users who do not follow the guidance in Downstream HBase 2.0+ users should use the shaded client will have to examine the set of dependencies Maven pulls in for impact. Downstream users of LimitedPrivate Coprocessor APIs will need to examine the runtime environment for impact.

Multiple breaking changes to source and binary compatibility for client API

The Java client API for HBase has a number of changes that break both source and binary compatibility for details see the Compatibility Check Report for the release you'll be upgrading to.

Tracing implementation changes

The backing implementation of HBase's tracing features was updated from Apache HTrace 3 to HTrace 4, which includes several breaking changes. While HTrace 3 and 4 can coexist in the same runtime, they will not integrate with each other, leading to disjoint trace information.

The internal changes to HBase during this upgrade were sufficient for compilation, but it has not been confirmed that there are no regressions in tracing functionality. Please

consider this feature experimental for the immediate future.

If you previously relied on client side tracing integrated with HBase operations, it is recommended that you upgrade your usage to HTrace 4 as well.

After the Apache HTrace project moved to the Attic/retired, the traces in HBase are left broken and unmaintained since HBase 2.0. A new project HBASE-22120 will replace HTrace with OpenTelemetry. It will be shipped in 3.0.0 release. Please see the Tracing documentation for more details.

HFile lose forward compatibility

HFiles generated by 2.0.0, 2.0.1, 2.1.0 are not forward compatible to 1.4.6-, 1.3.2.1-, 1.2.6.1-, and other inactive releases. Why HFile lose compatibility is hbase in new versions (2.0.0, 2.0.1, 2.1.0) use protobuf to serialize/deserialize TimeRangeTracker (TRT) while old versions use DataInput/DataOutput. To solve this, We have to put HBASE-21012 to 2.x and put HBASE-21013 in 1.x. For more information, please check HBASE-21008.

Performance

You will likely see a change in the performance profile on upgrade to hbase-2.0.0 given read and write paths have undergone significant change. On release, writes may be slower with reads about the same or much better, dependent on context. Be prepared to spend time re-tuning (See Performance). Performance is also an area that is now under active review so look forward to improvement in coming releases (See HBASE-20188 TESTING Performance).

Integration Tests and Kerberos

Integration Tests (`IntegrationTests*`) used to rely on the Kerberos credential cache for authentication against secured clusters. This used to lead to tests failing due to authentication failures when the tickets in the credential cache expired. As of hbase-2.0.0 (and hbase-1.3.0+), the integration test clients will make use of the configuration properties `hbase.client.keytab.file` and `hbase.client.kerberos.principal`. They are required. The clients will perform a login from the configured keytab file and automatically refresh the credentials in the background for the process lifetime (See HBASE-16231).

Default Compaction Throughput

HBase 2.x comes with default limits to the speed at which compactions can execute. This limit is defined per RegionServer. In previous versions of HBase earlier than 1.5, there was

no limit to the speed at which a compaction could run by default. Applying a limit to the throughput of a compaction should ensure more stable operations from RegionServers.

Take care to notice that this limit is *per RegionServer*, not *per compaction*.

The throughput limit is defined as a range of bytes written per second, and is allowed to vary within the given lower and upper bound. RegionServers observe the current throughput of a compaction and apply a linear formula to adjust the allowed throughput, within the lower and upper bound, with respect to external pressure. For compactions, external pressure is defined as the number of store files with respect to the maximum number of allowed store files. The more store files, the higher the compaction pressure.

Configuration of this throughput is governed by the following properties.

- The lower bound is defined by `hbase.hstore.compaction.throughput.lower.bound` and defaults to 50 MB/s (`52428800`).
- The upper bound is defined by `hbase.hstore.compaction.throughput.higher.bound` and defaults to 100 MB/s (`104857600`).

To revert this behavior to the unlimited compaction throughput of earlier versions of HBase, please set the following property to the implementation that applies no limits to compactions.

`hbase.regionserver.throughput.controller=org.apache.hadoop.hbase.regionserver.throttle.NoLimitThroughputController`

Upgrading Coprocessors to 2.0

Coprocessors have changed substantially in 2.0 ranging from top level design changes in class hierarchies to changed/removed methods, interfaces, etc. (Parent jira: HBASE-18169 Coprocessor fix and cleanup before 2.0.0 release). Some of the reasons for such widespread changes:

1. Pass Interfaces instead of Implementations; e.g. TableDescriptor instead of HTableDescriptor and Region instead of HRegion (HBASE-18241 Change client.Table and client.Admin to not use HTableDescriptor).
2. Design refactor so implementers need to fill out less boilerplate and so we can do more compile-time checking (HBASE-17732)

3. Purge Protocol Buffers from Coprocessor API (HBASE-18859, HBASE-16769, etc)
4. Cut back on what we expose to Coprocessors removing hooks on internals that were too private to expose (for eg. HBASE-18453 CompactionRequest should not be exposed to user directly; HBASE-18298 RegionServerServices Interface cleanup for CP expose; etc)

To use coprocessors in 2.0, they should be rebuilt against new API otherwise they will fail to load and HBase processes will die.

Suggested order of changes to upgrade the coprocessors:

1. Directly implement observer interfaces instead of extending Base*Observer classes. Change `Foo extends BaseXXXObserver` to `Foo implements XXXObserver`. (HBASE-17312).
2. Adapt to design change from Inheritence to Composition (HBASE-17732) by following this example.
3. getTable() has been removed from the CoprocessorEnvrionment, coprocessors should self-manage Table instances.

Some examples of writing coprocessors with new API can be found in hbase-example module here.

Lastly, if an api has been changed/removed that breaks you in an irreparable way, and if there's a good justification to add it back, bring it our notice dev@hbase.apache.org.

Rolling Upgrade from 1.x to 2.x

Rolling upgrades are currently an experimental feature. They have had limited testing. There are likely corner cases as yet uncovered in our limited experience so you should be careful if you go this route. The stop/upgrade/start described in the next section, Upgrade process from 1.x to 2.x, is the safest route.

That said, the below is a prescription for a rolling upgrade of a 1.4 cluster.

Pre-Requirements

- Upgrade to the latest 1.4.x release. Pre 1.4 releases may also work but are not tested, so please upgrade to 1.4.3+ before upgrading to 2.x, unless you are an expert and familiar with the region assignment and crash processing. See the section Upgrading from pre-1.4 to 1.4+ on how to upgrade to 1.4.x.

- Make sure that the zk-less assignment is enabled, i.e, set `hbase.assignment.usezk` to `false`. This is the most important thing. It allows the 1.x master to assign/unassign regions to/from 2.x region servers. See the release note section of HBASE-11059 on how to migrate from zk based assignment to zk less assignment.
- Before you upgrade, ensure that *hbck1* reports no `INCONSISTENCIES`. Fixing hbase1-type inconsistencies post-upgrade is an involved process.
- We have tested rolling upgrading from 1.4.3 to 2.1.0, but it should also work if you want to upgrade to 2.0.x.

Instructions

1. Unload a region server and upgrade it to 2.1.0. With HBASE-17931 in place, the meta region and regions for other system tables will be moved to this region server immediately. If not, please move them manually to the new region server. This is very important because
   - The schema of meta region is hard coded, if meta is on an old region server, then the new region servers can not access it as it does not have some families, for example, table state.
   - Client with lower version can communicate with server with higher version, but not vice versa. If the meta region is on an old region server, the new region server will use a client with higher version to communicate with a server with lower version, this may introduce strange problems.
2. Rolling upgrade all other region servers.
3. Upgrading masters.

It is OK that during the rolling upgrading there are region server crashes. The 1.x master can assign regions to both 1.x and 2.x region servers, and HBASE-19166 fixed a problem so that 1.x region server can also read the WALs written by 2.x region server and split them.

> Please read the Changes of Note! section carefully before rolling upgrading. Make sure that you do not use the removed features in 2.0, for example, the prefix-tree encoding, the old hfile format, etc. They could both fail the upgrading and leave the cluster in an intermediate state and hard to recover.

> If you have success running this prescription, please notify the dev list with a note on your experience and/or update the above with any deviations you may have taken so others going

this route can benefit from your efforts.

Upgrade process from 1.x to 2.x

To upgrade an existing HBase 1.x cluster, you should:

- Ensure that *hbck1* reports no `INCONSISTENCIES`. Fixing hbase1-type inconsistencies post-upgrade is an involved process. Fix all *hbck1* complaints before proceeding.
- Clean shutdown of existing 1.x cluster
- Update coprocessors
- Upgrade Master roles first
- Upgrade RegionServers
- (Eventually) Upgrade Clients

Upgrading to 1.7.1+

HBase release 1.7.0 introduced an incompatible table metadata serialization format that broke the minor release compatibility guarantees. The issue was reported in HBASE-26021 and the problematic serialization patch was reverted in HBase 1.7.1. Some important notes about 1.7.x upgrades below.

- If you are considering an upgrade to 1.7.x version, skip 1.7.0 completely and upgrade to 1.7.1+ version. 1.7.0 was withdrawn and removed from the Apache sites.
- If you already installed a 1.7.0 cluster from scratch and are looking to migrate to 1.7.1+, you cannot follow the regular rolling upgrade procedures due to broken compatibility contracts. Instead shutdown the cluster and reboot with 1.7.1+ binaries. Newer versions detect any existing tables with incompatible serialization and rewrite them using the correct format at bootstrap.
- If you are already on 1.7.1+ version, everything is good and no additional steps need to be performed.

Upgrading from pre-1.4 to 1.4+

Region Server memory consumption changes.

Users upgrading from versions prior to HBase 1.4 should be aware that the estimates of heap usage by the memstore objects (KeyValue, object and array header sizes, etc) have been made more accurate for heap sizes up to 32G (using CompressedOops), resulting in them dropping by 10-50% in practice. This also results in less number of flushes and compactions due to "fatter" flushes. YMMV. As a result, the actual heap usage of the memstore before being flushed may increase by up to 100%. If configured memory limits for the region server had been tuned based on observed usage, this change could result in worse GC behavior or even OutOfMemory errors. Set the environment property (not hbase-site.xml) "hbase.memorylayout.use.unsafe" to false to disable.

Replication peer's TableCFs config

Before 1.4, the table name can't include namespace for replication peer's TableCFs config. It was fixed by add TableCFs to ReplicationPeerConfig which was stored on Zookeeper. So when upgrade to 1.4, you have to update the original ReplicationPeerConfig data on Zookeeper firstly. There are four steps to upgrade when your cluster have a replication peer with TableCFs config.

- Disable the replication peer.
- If master has permission to write replication peer znode, then rolling update master directly. If not, use TableCFsUpdater tool to update the replication peer's config.

```
$ bin/hbase org.apache.hadoop.hbase.replication.master.TableCFsUpdater update
```

- Rolling update regionservers.
- Enable the replication peer.

Notes:

- Can't use the old client(before 1.4) to change the replication peer's config. Because the client will write config to Zookeeper directly, the old client will miss TableCFs config. And the old client write TableCFs config to the old tablecfs znode, it will not work for new version regionserver.

Raw scan now ignores TTL

Doing a raw scan will now return results that have expired according to TTL settings.

Upgrading from pre-1.3 to 1.3+

If running Integration Tests under Kerberos, see upgrade2.0.it.kerberos.

Upgrading to 1.x

Please consult the documentation published specifically for the version of HBase that you are upgrading to for details on the upgrade process.

The Apache HBase Shell

The Apache HBase Shell is (J)Ruby's IRB with some HBase particular commands added. Anything you can do in IRB, you should be able to do in the HBase Shell.

To run the HBase shell, do as follows:

```
$ ./bin/hbase shell
```

Type `help` and then `<RETURN>` to see a listing of shell commands and options. Browse at least the paragraphs at the end of the help output for the gist of how variables and command arguments are entered into the HBase shell; in particular note how table names, rows, and columns, etc., must be quoted.

See shell exercises for example basic shell operation.

Here is a nicely formatted listing of all shell commands by Rajeshbabu Chintaguntla.

Scripting with Ruby

For examples scripting Apache HBase, look in the HBase *bin* directory. Look at the files that end in **.rb*. To run one of these files, do as follows:

```
$ ./bin/hbase org.jruby.Main PATH_TO_SCRIPT
```

Running the Shell in Non-Interactive Mode

A new non-interactive mode has been added to the HBase Shell (HBASE-11658). Non-interactive mode captures the exit status (success or failure) of HBase Shell commands and passes that status back to the command interpreter. If you use the normal interactive mode, the HBase Shell will only ever return its own exit status, which will nearly always be `0` for success.

To invoke non-interactive mode, pass the `-n` or `--non-interactive` option to HBase Shell.

HBase Shell in OS Scripts

You can use the HBase shell from within operating system script interpreters like the Bash shell which is the default command interpreter for most Linux and UNIX distributions. The following guidelines use Bash syntax, but could be adjusted to work with C-style shells such as csh or tcsh, and could probably be modified to work with the Microsoft Windows script interpreter as well. Submissions are welcome.

> Spawning HBase Shell commands in this way is slow, so keep that in mind when you are deciding when combining HBase operations with the operating system command line is appropriate.

Passing Commands to the HBase Shell

You can pass commands to the HBase Shell in non-interactive mode (see **hbase.shell.noninteractive**) using the `echo` command and the `|` (pipe) operator. Be sure to escape characters in the HBase commands which would otherwise be interpreted by the shell. Some debug-level output has been truncated from the example below.

```
$ echo "describe 'test1'" | ./hbase shell -n

Version 0.98.3-hadoop2, rd5e65a9144e315bb0a964e7730871af32f5018d5, Sat May 31 19:
56:09 PDT 2014

describe 'test1'

DESCRIPTION                                          ENABLED
 'test1', {NAME => 'cf', DATA_BLOCK_ENCODING => 'NON true
 E', BLOOMFILTER => 'ROW', REPLICATION_SCOPE => '0',
  VERSIONS => '1', COMPRESSION => 'NONE', MIN_VERSIO
 NS => '0', TTL => 'FOREVER', KEEP_DELETED_CELLS =>
 'false', BLOCKSIZE => '65536', IN_MEMORY => 'false'
 , BLOCKCACHE => 'true'}
1 row(s) in 3.2410 seconds
```

To suppress all output, echo it to */dev/null:*

```
$ echo "describe 'test'" | ./hbase shell -n > /dev/null 2>&1
```

Checking the Result of a Scripted Command

Since scripts are not designed to be run interactively, you need a way to check whether your command failed or succeeded. The HBase shell uses the standard convention of returning a value of `0` for successful commands, and some non-zero value for failed commands. Bash stores a command's return value in a special environment variable called `$?`. Because that variable is overwritten each time the shell runs any command, you should store the result in a different, script-defined variable.

This is a naive script that shows one way to store the return value and make a decision based upon it.

```
#!/bin/bash

echo "describe 'test'" | ./hbase shell -n > /dev/null 2>&1
status=$?
echo "The status was " $status
if ($status == 0); then
  echo "The command succeeded"
else
  echo "The command may have failed."
fi
return $status
```

Checking for Success or Failure In Scripts

Getting an exit code of `0` means that the command you scripted definitely succeeded. However, getting a non-zero exit code does not necessarily mean the command failed. The command could have succeeded, but the client lost connectivity, or some other event obscured its success. This is because RPC commands are stateless. The only way to be sure of the status of an operation is to check. For instance, if your script creates a table, but returns a non-zero exit value, you should check whether the table was actually created before trying again to create it.

Read HBase Shell Commands from a Command File

You can enter HBase Shell commands into a text file, one command per line, and pass that file to the HBase Shell.

Example Command File

```
create 'test', 'cf'
list 'test'
put 'test', 'row1', 'cf:a', 'value1'
put 'test', 'row2', 'cf:b', 'value2'
put 'test', 'row3', 'cf:c', 'value3'
put 'test', 'row4', 'cf:d', 'value4'
scan 'test'
get 'test', 'row1'
disable 'test'
enable 'test'
```

Directing HBase Shell to Execute the Commands

Pass the path to the command file as the only argument to the `hbase shell` command. Each command is executed and its output is shown. If you do not include the `exit` command in your script, you are returned to the HBase shell prompt. There is no way to programmatically check each individual command for success or failure. Also, though you see the output for each command, the commands themselves are not echoed to the screen so it can be difficult to line up the command with its output.

```
$ ./hbase shell ./sample_commands.txt
0 row(s) in 3.4170 seconds

TABLE
test
1 row(s) in 0.0590 seconds

0 row(s) in 0.1540 seconds

0 row(s) in 0.0080 seconds

0 row(s) in 0.0060 seconds

0 row(s) in 0.0060 seconds

ROW                   COLUMN+CELL
 row1                 column=cf:a, timestamp=1407130286968, value=value1
 row2                 column=cf:b, timestamp=1407130286997, value=value2
 row3                 column=cf:c, timestamp=1407130287007, value=value3
 row4                 column=cf:d, timestamp=1407130287015, value=value4
4 row(s) in 0.0420 seconds

COLUMN                CELL
 cf:a                 timestamp=1407130286968, value=value1
1 row(s) in 0.0110 seconds
```

```
0 row(s) in 1.5630 seconds

0 row(s) in 0.4360 seconds
```

Passing VM Options to the Shell

You can pass VM options to the HBase Shell using the `HBASE_SHELL_OPTS` environment variable. You can set this in your environment, for instance by editing *~/.bashrc*, or set it as part of the command to launch HBase Shell. The following example sets several garbage-collection-related variables, just for the lifetime of the VM running the HBase Shell. The command should be run all on a single line, but is broken by the `\` character, for readability.

```
$ HBASE_SHELL_OPTS="-verbose:gc -XX:+PrintGCApplicationStoppedTime -XX:+PrintGCDa
teStamps \
  -XX:+PrintGCDetails -Xloggc:$HBASE_HOME/logs/gc-hbase.log" ./bin/hbase shell
```

Overriding configuration starting the HBase Shell

As of hbase-2.0.5/hbase-2.1.3/hbase-2.2.0/hbase-1.4.10/hbase-1.5.0, you can pass or override hbase configuration as specified in `hbase-*.xml` by passing your key/values prefixed with `-D` on the command-line as follows:

```
$ ./bin/hbase shell -Dhbase.zookeeper.quorum=ZK0.remote.cluster.example.org,ZK1.r
emote.cluster.example.org,ZK2.remote.cluster.example.org -Draining=false
...
hbase(main):001:0> @shell.hbase.configuration.get("hbase.zookeeper.quorum")
=> "ZK0.remote.cluster.example.org,ZK1.remote.cluster.example.org,ZK2.remote.clus
ter.example.org"
hbase(main):002:0> @shell.hbase.configuration.get("raining")
=> "false"
```

Shell Tricks

Table variables

HBase 0.95 adds shell commands that provides jruby-style object-oriented references for tables. Previously all of the shell commands that act upon a table have a procedural style that always took the name of the table as an argument. HBase 0.95 introduces the ability to assign a table to a jruby variable. The table reference can be used to perform data read write operations such as puts, scans, and gets well as admin functionality such as disabling, dropping, describing tables.

For example, previously you would always specify a table name:

```
hbase(main):000:0> create 't', 'f'
0 row(s) in 1.0970 seconds
hbase(main):001:0> put 't', 'rold', 'f', 'v'
0 row(s) in 0.0080 seconds

hbase(main):002:0> scan 't'
ROW                                COLUMN+CELL
 rold                              column=f:, timestamp=1378473207660, value=v
1 row(s) in 0.0130 seconds

hbase(main):003:0> describe 't'
DESCRIPTION
ENABLED
 't', {NAME => 'f', DATA_BLOCK_ENCODING => 'NONE', BLOOMFILTER => 'ROW', REPLICAT
ION_ true
 SCOPE => '0', VERSIONS => '1', COMPRESSION => 'NONE', MIN_VERSIONS => '0', TTL =
> '2
 147483647', KEEP_DELETED_CELLS => 'false', BLOCKSIZE => '65536', IN_MEMORY => 'f
alse
 ', BLOCKCACHE => 'true'}
1 row(s) in 1.4430 seconds

hbase(main):004:0> disable 't'
0 row(s) in 14.8700 seconds

hbase(main):005:0> drop 't'
0 row(s) in 23.1670 seconds

hbase(main):006:0>
```

Now you can assign the table to a variable and use the results in jruby shell code.

```
hbase(main):007 > t = create 't', 'f'
0 row(s) in 1.0970 seconds
```

```
=> Hbase::Table - t
hbase(main):008 > t.put 'r', 'f', 'v'
0 row(s) in 0.0640 seconds
hbase(main):009 > t.scan
ROW                          COLUMN+CELL
 r                           column=f:, timestamp=1331865816290, value=v
1 row(s) in 0.0110 seconds
hbase(main):010:0> t.describe
DESCRIPTION
ENABLED
 't', {NAME => 'f', DATA_BLOCK_ENCODING => 'NONE', BLOOMFILTER => 'ROW', REPLICAT
ION_ true
 SCOPE => '0', VERSIONS => '1', COMPRESSION => 'NONE', MIN_VERSIONS => '0', TTL =
> '2
 147483647', KEEP_DELETED_CELLS => 'false', BLOCKSIZE => '65536', IN_MEMORY => 'f
alse
 ', BLOCKCACHE => 'true'}
1 row(s) in 0.0210 seconds
hbase(main):038:0> t.disable
0 row(s) in 6.2350 seconds
hbase(main):039:0> t.drop
0 row(s) in 0.2340 seconds
```

If the table has already been created, you can assign a Table to a variable by using the get_table method:

```
hbase(main):011 > create 't','f'
0 row(s) in 1.2500 seconds

=> Hbase::Table - t
hbase(main):012:0> tab = get_table 't'
0 row(s) in 0.0010 seconds

=> Hbase::Table - t
hbase(main):013:0> tab.put 'r1' ,'f', 'v'
0 row(s) in 0.0100 seconds
hbase(main):014:0> tab.scan
ROW                              COLUMN+CELL
 r1                              column=f:, timestamp=1378473876949, value=v
1 row(s) in 0.0240 seconds
hbase(main):015:0>
```

The list functionality has also been extended so that it returns a list of table names as strings. You can then use jruby to script table operations based on these names. The list_snapshots command also acts similarly.

```
hbase(main):016 > tables = list('t.*')
TABLE
t
1 row(s) in 0.1040 seconds
```

```
=> ["t"]
hbase(main):017:0> tables.map { |t| disable t ; drop  t}
0 row(s) in 2.2510 seconds

=> [nil]
hbase(main):018:0>
```

irbrc

Create an *.irbrc* file for yourself in your home directory. Add customizations. A useful one is command history so commands are save across Shell invocations:

```
$ more .irbrc
require 'irb/ext/save-history'
IRB.conf[:SAVE_HISTORY] = 100
IRB.conf[:HISTORY_FILE] = "#{ENV['HOME']}/.irb-save-history"
```

If you'd like to avoid printing the result of evaluting each expression to stderr, for example the array of tables returned from the "list" command:

```
$ echo "IRB.conf[:ECHO] = false" >>~/.irbrc
```

See the `ruby` documentation of *.irbrc* to learn about other possible configurations.

LOG data to timestamp

To convert the date '08/08/16 20:56:29' from an hbase log into a timestamp, do:

```
hbase(main):021:0> import java.text.SimpleDateFormat
hbase(main):022:0> import java.text.ParsePosition
hbase(main):023:0> SimpleDateFormat.new("yy/MM/dd HH:mm:ss").parse("08/08/16 20:5
6:29", ParsePosition.new(0)).getTime() => 1218920189000
```

To go the other direction:

```
hbase(main):021:0> import java.util.Date
hbase(main):022:0> Date.new(1218920189000).toString() => "Sat Aug 16 20:56:29 UTC
2008"
```

To output in a format that is exactly like that of the HBase log format will take a little messing with SimpleDateFormat.

Query Shell Configuration

```
hbase(main):001:0> @shell.hbase.configuration.get("hbase.rpc.timeout")
=> "60000"
```

To set a config in the shell:

```
hbase(main):005:0> @shell.hbase.configuration.setInt("hbase.rpc.timeout", 61010)
hbase(main):006:0> @shell.hbase.configuration.get("hbase.rpc.timeout")
=> "61010"
```

Pre-splitting tables with the HBase Shell

You can use a variety of options to pre-split tables when creating them via the HBase Shell `create` command.

The simplest approach is to specify an array of split points when creating the table. Note that when specifying string literals as split points, these will create split points based on the underlying byte representation of the string. So when specifying a split point of '10', we are actually specifying the byte split point '\x31\30'.

The split points will define `n+1` regions where `n` is the number of split points. The lowest region will contain all keys from the lowest possible key up to but not including the first split point key. The next region will contain keys from the first split point up to, but not including the next split point key. This will continue for all split points up to the last. The last region will be defined from the last split point up to the maximum possible key.

```
hbase>create 't1','f',SPLITS => ['10','20','30']
```

In the above example, the table 't1' will be created with column family 'f', pre-split to four regions. Note the first region will contain all keys from '\x00' up to '\x30' (as '\x31' is the ASCII code for '1').

You can pass the split points in a file using following variation. In this example, the splits are read from a file corresponding to the local path on the local filesystem. Each line in the file specifies a split point key.

```
hbase>create 't14','f',SPLITS_FILE=>'splits.txt'
```

The other options are to automatically compute splits based on a desired number of regions and a splitting algorithm. HBase supplies algorithms for splitting the key range based on uniform splits or based on hexadecimal keys, but you can provide your own splitting algorithm to subdivide the key range.

```
# create table with four regions based on random bytes keys
hbase>create 't2','f1', { NUMREGIONS => 4 , SPLITALGO => 'UniformSplit' }

# create table with five regions based on hex keys
hbase>create 't3','f1', { NUMREGIONS => 5, SPLITALGO => 'HexStringSplit' }
```

As the HBase Shell is effectively a Ruby environment, you can use simple Ruby scripts to compute splits algorithmically.

```
# generate splits for long (Ruby fixnum) key range from start to end key
hbase(main):070:0> def gen_splits(start_key,end_key,num_regions)
hbase(main):071:1>   results=[]
hbase(main):072:1>   range=end_key-start_key
hbase(main):073:1>   incr=(range/num_regions).floor
hbase(main):074:1>   for i in 1 .. num_regions-1
hbase(main):075:2>     results.push([i*incr+start_key].pack("N"))
hbase(main):076:2>   end
hbase(main):077:1>   return results
hbase(main):078:1> end
hbase(main):079:0>
hbase(main):080:0> splits=gen_splits(1,2000000,10)
=> ["\000\003\r@", "\000\006\032\177", "\000\t'\276", "\000\f4\375", "\000\017B
<", "\000\022O{", "\000\025\\\272", "\000\030i\371", "\000\ew8"]
hbase(main):081:0> create 'test_splits','f',SPLITS=>splits
0 row(s) in 0.2670 seconds

=> Hbase::Table - test_splits
```

Note that the HBase Shell command `truncate` effectively drops and recreates the table with default options which will discard any pre-splitting. If you need to truncate a pre-split table, you must drop and recreate the table explicitly to re-specify custom split options.

Debug

Shell debug switch

You can set a debug switch in the shell to see more output — e.g. more of the stack trace on exception — when you run a command:

```
hbase> debug <RETURN>
```

DEBUG log level

To enable DEBUG level logging in the shell, launch it with the `-d` option.

```
$ ./bin/hbase shell -d
```

Commands

count

Count command returns the number of rows in a table. It's quite fast when configured with the right CACHE

```
hbase> count '<tablename>', CACHE => 1000
```

The above count fetches 1000 rows at a time. Set CACHE lower if your rows are big. Default is to fetch one row at a time.

Data Model

HBase Data Model Terminology

Table

An HBase table consists of multiple rows.

Row

A row in HBase consists of a row key and one or more columns with values associated with them. Rows are sorted alphabetically by the row key as they are stored. For this reason, the design of the row key is very important. The goal is to store data in such a way that related rows are near each other. A common row key pattern is a website domain. If your row keys are domains, you should probably store them in reverse (org.apache.www, org.apache.mail, org.apache.jira). This way, all of the Apache domains are near each other in the table, rather than being spread out based on the first letter of the subdomain.

Column

A column in HBase consists of a column family and a column qualifier, which are delimited by a `:` (colon) character.

Column Family

Column families physically colocate a set of columns and their values, often for performance reasons. Each column family has a set of storage properties, such as whether its values should be cached in memory, how its data is compressed or its row keys are encoded, and others. Each row in a table has the same column families, though a given row might not store anything in a given column family.

Column Qualifier

A column qualifier is added to a column family to provide the index for a given piece of data. Given a column family `content`, a column qualifier might be `content:html`, and another might be `content:pdf`. Though column families are fixed at table creation, column qualifiers are mutable and may differ greatly between rows.

Cell

A cell is a combination of row, column family, and column qualifier, and contains a value and a timestamp, which represents the value's version.

Timestamp

A timestamp is written alongside each value, and is the identifier for a given version of a value. By default, the timestamp represents the time on the RegionServer when the data was written, but you can specify a different timestamp value when you put data into the cell.

Conceptual View

You can read a very understandable explanation of the HBase data model in the blog post Understanding HBase and BigTable by Jim R. Wilson. Another good explanation is available in the PDF Introduction to Basic Schema Design by Amandeep Khurana.

It may help to read different perspectives to get a solid understanding of HBase schema design. The linked articles cover the same ground as the information in this section.

The following example is a slightly modified form of the one on page 2 of the BigTable paper. There is a table called `webtable` that contains two rows (`com.cnn.www` and `com.example.www`) and three column families named `contents`, `anchor`, and `people`. In this example, for the first row (`com.cnn.www`), `anchor` contains two columns (`anchor:cssnsi.com`, `anchor:my.look.ca`) and `contents` contains one column (`contents:html`). This example contains 5 versions of the row with the row key `com.cnn.www`, and one version of the row with the row key `com.example.www`. The `contents:html` column qualifier contains the entire HTML of a given website. Qualifiers of the `anchor` column family each contain the external site which links to the site represented by the row, along with the text it used in the anchor of its link. The `people` column family represents people associated with the site.

> **Column Names**
>
> By convention, a column name is made of its column family prefix and a *qualifier*. For example, the column *contents:html* is made up of the column family `contents` and the `html` qualifier. The colon character (`:`) delimits the column family from the column family *qualifier*.

Table `webtable`

Row Key	Time Stamp	ColumnFamily `contents`	ColumnFamily `anchor`	ColumnFamily `people`
"com.cnn.www"	t9		anchor:cnnsi.com = "CNN"	
"com.cnn.www"	t8		anchor:my.look.ca = "CNN.com"	
"com.cnn.www"	t6	contents:html = "<html>..."		
"com.cnn.www"	t5	contents:html = "<html>..."		
"com.cnn.www"	t3	contents:html = "<html>..."		
"com.example.www"	t5	contents:html = "<html>..."		people:author = "John Doe"

Cells in this table that appear to be empty do not take space, or in fact exist, in HBase. This is what makes HBase "sparse." A tabular view is not the only possible way to look at data in HBase, or even the most accurate. The following represents the same information as a multi-dimensional map. This is only a mock-up for illustrative purposes and may not be strictly accurate.

```
{
  "com.cnn.www": {
    contents: {
      t6: contents:html: "<html>..."
      t5: contents:html: "<html>..."
      t3: contents:html: "<html>..."
    }
    anchor: {
      t9: anchor:cnnsi.com = "CNN"
      t8: anchor:my.look.ca = "CNN.com"
    }
    people: {}
  }
  "com.example.www": {
    contents: {
      t5: contents:html: "<html>..."
    }
    anchor: {}
    people: {
      t5: people:author: "John Doe"
```

```
    }
  }
}
```

Physical View

Although at a conceptual level tables may be viewed as a sparse set of rows, they are physically stored by column family. A new column qualifier (column_family:column_qualifier) can be added to an existing column family at any time.

ColumnFamily `anchor`

Row Key	Time Stamp	Column Family `anchor`
"com.cnn.www"	t9	`anchor:cnnsi.com = "CNN"`
"com.cnn.www"	t8	`anchor:my.look.ca = "CNN.com"`

ColumnFamily `contents`

Row Key	Time Stamp	ColumnFamily `contents:`
"com.cnn.www"	t6	contents:html = "<html>..."
"com.cnn.www"	t5	contents:html = "<html>..."
"com.cnn.www"	t3	contents:html = "<html>..."

The empty cells shown in the conceptual view are not stored at all. Thus a request for the value of the `contents:html` column at time stamp `t8` would return no value. Similarly, a request for an `anchor:my.look.ca` value at time stamp `t9` would return no value. However, if no timestamp is supplied, the most recent value for a particular column would be returned. Given multiple versions, the most recent is also the first one found, since timestamps are stored in descending order. Thus a request for the values of all columns in the row `com.cnn.www` if no timestamp is specified would be: the value of `contents:html` from timestamp `t6`, the value of `anchor:cnnsi.com` from timestamp `t9`, the value of `anchor:my.look.ca` from timestamp `t8`.

For more information about the internals of how Apache HBase stores data, see regions.arch.

Namespace

A namespace is a logical grouping of tables analogous to a database in relation database systems. This abstraction lays the groundwork for upcoming multi-tenancy related features:

- Quota Management (HBASE-8410) - Restrict the amount of resources (i.e. regions, tables) a namespace can consume.
- Namespace Security Administration (HBASE-9206) - Provide another level of security administration for tenants.
- Region server groups (HBASE-6721) - A namespace/table can be pinned onto a subset of RegionServers thus guaranteeing a coarse level of isolation.

Namespace management

A namespace can be created, removed or altered. Namespace membership is determined during table creation by specifying a fully-qualified table name of the form:

```
<table namespace>:<table qualifier>
```

Examples

```
#Create a namespace
create_namespace 'my_ns'
```

```
#create my_table in my_ns namespace
create 'my_ns:my_table', 'fam'
```

```
#drop namespace
drop_namespace 'my_ns'
```

```
#alter namespace
alter_namespace 'my_ns', {METHOD => 'set', 'PROPERTY_NAME' => 'PROPERTY_VALUE'}
```

Predefined namespaces

There are two predefined special namespaces:

- hbase - system namespace, used to contain HBase internal tables
- default - tables with no explicit specified namespace will automatically fall into this namespace

Examples #datamodel-predefined-namespaces-examples

```
#namespace=foo and table qualifier=bar
create 'foo:bar', 'fam'

#namespace=default and table qualifier=bar
create 'bar', 'fam'
```

About hbase:namespace table

We used to have a system table called `hbase:namespace` for storing the namespace information.

It introduced some painful bugs in the past, especially that it may hang the master startup thus hang the whole cluster. This is because meta table also has a namespace, so it depends on namespace table. But namespace table also depends on meta table as meta table stores the location of all regions. This is a cyclic dependency so sometimes namespace and meta table will wait for each other to online and hang the master start up.

It is not easy to fix so in 3.0.0, we decided to completely remove the `hbase:namespace` table and fold its content into the `ns` family in `hbase:meta` table. When upgrading from 2.x to 3.x, the migration will be done automatically and the `hbase:namespace` table will be disabled after the migration is done. You are free to leave it there for sometime and finally drop it.

For more tails, please see https://issues.apache.org/jira/browse/HBASE-21154.

Table

Tables are declared up front at schema definition time.

Row

Row keys are uninterpreted bytes. Rows are lexicographically sorted with the lowest order appearing first in a table. The empty byte array is used to denote both the start and end of a tables' namespace.

Column Family

Columns in Apache HBase are grouped into *column families*. All column members of a column family have the same prefix. For example, the columns *courses:history* and *courses:math* are both members of the *courses* column family. The colon character (`:`) delimits the column family from the column family qualifier. The column family prefix must be composed of *printable* characters. The qualifying tail, the column family *qualifier*, can be made of any arbitrary bytes. Column families must be declared up front at schema definition time whereas columns do not need to be defined at schema time but can be conjured on the fly while the table is up and running.

Physically, all column family members are stored together on the filesystem. Because tunings and storage specifications are done at the column family level, it is advised that all column family members have the same general access pattern and size characteristics.

Cells

A *{row, column, version}* tuple exactly specifies a `cell` in HBase. Cell content is uninterpreted bytes

Data Model Operations

The four primary data model operations are Get, Put, Scan, and Delete. Operations are applied via Table instances.

Get

Get returns attributes for a specified row. Gets are executed via Table.get

Put

Put either adds new rows to a table (if the key is new) or can update existing rows (if the key already exists). Puts are executed via Table.put (non-writeBuffer) or Table.batch (non-writeBuffer)

Scans

Scan allow iteration over multiple rows for specified attributes.

The following is an example of a Scan on a Table instance. Assume that a table is populated with rows with keys "row1", "row2", "row3", and then another set of rows with the keys "abc1", "abc2", and "abc3". The following example shows how to set a Scan instance to return the rows beginning with "row".

```
public static final byte[] CF = "cf".getBytes();
public static final byte[] ATTR = "attr".getBytes();
...

Table table = ...      // instantiate a Table instance

Scan scan = new Scan();
scan.addColumn(CF, ATTR);
scan.setStartStopRowForPrefixScan(Bytes.toBytes("row"));
ResultScanner rs = table.getScanner(scan);
try {
  for (Result r = rs.next(); r != null; r = rs.next()) {
    // process result...
  }
} finally {
  rs.close();  // always close the ResultScanner!
}
```

Note that generally the easiest way to specify a specific stop point for a scan is by using the InclusiveStopFilter class.

Delete

Delete removes a row from a table. Deletes are executed via Table.delete.

HBase does not modify data in place, and so deletes are handled by creating new markers called *tombstones*. These tombstones, along with the dead values, are cleaned up on major compactions.

See version.delete for more information on deleting versions of columns, and see compaction for more information on compactions.

Versions

A *{row, column, version}* tuple exactly specifies a `cell` in HBase. It's possible to have an unbounded number of cells where the row and column are the same but the cell address differs only in its version dimension.

While rows and column keys are expressed as bytes, the version is specified using a long integer. Typically this long contains time instances such as those returned by `java.util.Date.getTime()` or `System.currentTimeMillis()`, that is: *the difference, measured in milliseconds, between the current time and midnight, January 1, 1970 UTC*.

The HBase version dimension is stored in decreasing order, so that when reading from a store file, the most recent values are found first.

There is a lot of confusion over the semantics of `cell` versions, in HBase. In particular:

- If multiple writes to a cell have the same version, only the last written is fetchable.
- It is OK to write cells in a non-increasing version order.

Below we describe how the version dimension in HBase currently works. See HBASE-2406 for discussion of HBase versions. Bending time in HBase makes for a good read on the version, or time, dimension in HBase. It has more detail on versioning than is provided here.

As of this writing, the limitation *Overwriting values at existing timestamps* mentioned in the article no longer holds in HBase. This section is basically a synopsis of this article by Bruno Dumon.

Specifying the Number of Versions to Store

The maximum number of versions to store for a given column is part of the column schema and is specified at table creation, or via an `alter` command, via `HColumnDescriptor.DEFAULT_VERSIONS`. Prior to HBase 0.96, the default number of versions kept was `3`, but in 0.96 and newer has been changed to `1`.

Example: Modify the Maximum Number of Versions for a Column Family

This example uses HBase Shell to keep a maximum of 5 versions of all columns in column family `f1`. You could also use ColumnFamilyDescriptorBuilder.

```
hbase> alter 't1', NAME => 'f1', VERSIONS => 5
```

Example: Modify the Minimum Number of Versions for a Column Family

You can also specify the minimum number of versions to store per column family. By default, this is set to 0, which means the feature is disabled. The following example sets the minimum number of versions on all columns in column family `f1` to `2`, via HBase Shell. You could also use ColumnFamilyDescriptorBuilder.

```
hbase> alter 't1', NAME => 'f1', MIN_VERSIONS => 2
```

Starting with HBase 0.98.2, you can specify a global default for the maximum number of versions kept for all newly-created columns, by setting `hbase.column.max.version` in *hbase-site.xml*. See hbase.column.max.version.

Versions and HBase Operations

In this section we look at the behavior of the version dimension for each of the core HBase operations.

Get/Scan

Gets are implemented on top of Scans. The below discussion of Get applies equally to Scans.

By default, i.e. if you specify no explicit version, when doing a `get`, the cell whose version has the largest value is returned (which may or may not be the latest one written, see later). The default behavior can be modified in the following ways:

- to return more than one version, see Get.readVersions(int)
- to return versions other than the latest, see Get.setTimeRange(long,long)

To retrieve the latest version that is less than or equal to a given value, thus giving the 'latest' state of the record at a certain point in time, just use a range from 0 to the desired version and set the max versions to 1.

Default Get Example

The following Get will only retrieve the current version of the row

```
public static final byte[] CF = "cf".getBytes();
public static final byte[] ATTR = "attr".getBytes();
...
Get get = new Get(Bytes.toBytes("row1"));
Result r = table.get(get);
byte[] b = r.getValue(CF, ATTR);  // returns current version of value
```

Versioned Get Example

The following Get will return the last 3 versions of the row.

```
public static final byte[] CF = "cf".getBytes();
public static final byte[] ATTR = "attr".getBytes();
...
Get get = new Get(Bytes.toBytes("row1"));
get.setMaxVersions(3);  // will return last 3 versions of row
Result r = table.get(get);
byte[] b = r.getValue(CF, ATTR);  // returns current version of value
List<Cell> cells = r.getColumnCells(CF, ATTR);  // returns all versions of this c
olumn
```

Put

Doing a put always creates a new version of a `cell`, at a certain timestamp. By default the system uses the server's `currentTimeMillis`, but you can specify the version (= the long integer) yourself, on a per-column level. This means you could assign a time in the past or the future, or use the long value for non-time purposes.

To overwrite an existing value, do a put at exactly the same row, column, and version as that of the cell you want to overwrite.

Implicit Version Example

The following Put will be implicitly versioned by HBase with the current time.

```
public static final byte[] CF = "cf".getBytes();
public static final byte[] ATTR = "attr".getBytes();
...
Put put = new Put(Bytes.toBytes(row));
put.add(CF, ATTR, Bytes.toBytes( data));
table.put(put);
```

Explicit Version Example

The following Put has the version timestamp explicitly set.

```
public static final byte[] CF = "cf".getBytes();
public static final byte[] ATTR = "attr".getBytes();
...
Put put = new Put( Bytes.toBytes(row));
long explicitTimeInMs = 555;  // just an example
put.add(CF, ATTR, explicitTimeInMs, Bytes.toBytes(data));
table.put(put);
```

Caution: the version timestamp is used internally by HBase for things like time-to-live calculations. It's usually best to avoid setting this timestamp yourself. Prefer using a separate timestamp attribute of the row, or have the timestamp as a part of the row key, or both.

Cell Version Example

The following Put uses a method getCellBuilder() to get a CellBuilder instance that already has relevant Type and Row set.

```
public static final byte[] CF = "cf".getBytes();
public static final byte[] ATTR = "attr".getBytes();
...

Put put = new Put(Bytes.toBytes(row));
put.add(put.getCellBuilder().setQualifier(ATTR)
   .setFamily(CF)
   .setValue(Bytes.toBytes(data))
   .build());
table.put(put);
```

Delete

There are three different types of internal delete markers. See Lars Hofhansl's blog for discussion of his attempt adding another, Scanning in HBase: Prefix Delete Marker.

- Delete: for a specific version of a column.
- Delete column: for all versions of a column.
- Delete family: for all columns of a particular ColumnFamily

When deleting an entire row, HBase will internally create a tombstone for each ColumnFamily (i.e., not each individual column).

Deletes work by creating *tombstone* markers. For example, let's suppose we want to delete a row. For this you can specify a version, or else by default the `currentTimeMillis` is used. What this means is *delete all cells where the version is less than or equal to this version*. HBase never modifies data in place, so for example a delete will not immediately delete (or mark as deleted) the entries in the storage file that correspond to the delete condition. Rather, a so-called *tombstone* is written, which will mask the deleted values. When HBase does a major compaction, the tombstones are processed to actually remove the dead values, together with the tombstones themselves. If the version you specified when deleting a row is larger than the version of any value in the row, then you can consider the complete row to be deleted.

For an informative discussion on how deletes and versioning interact, see the thread **Put w/timestamp → Deleteall → Put w/ timestamp fails** up on the user mailing list.

Also see **keyvalue** for more information on the internal KeyValue format.

Delete markers are purged during the next major compaction of the store, unless the `KEEP_DELETED_CELLS` option is set in the column family (See **Keeping Deleted Cells**). To keep the deletes for a configurable amount of time, you can set the delete TTL via the `hbase.hstore.time.to.purge.deletes` property in *hbase-site.xml*. If `hbase.hstore.time.to.purge.deletes` is not set, or set to 0, all delete markers, including those with timestamps in the future, are purged during the next major compaction. Otherwise, a delete marker with a timestamp in the future is kept until the major compaction which occurs after the time represented by the marker's timestamp plus the value of `hbase.hstore.time.to.purge.deletes`, in milliseconds.

> This behavior represents a fix for an unexpected change that was introduced in HBase 0.94, and was fixed in **HBASE-10118**. The change has been backported to HBase 0.94 and newer branches.

Optional New Version and Delete behavior in HBase-2.0.0

In `hbase-2.0.0`, the operator can specify an alternate version and delete treatment by setting the column descriptor property `NEW_VERSION_BEHAVIOR` to true (To set a property on a column family descriptor, you must first disable the table and then alter the column family descriptor; see **Keeping Deleted Cells** for an example of editing an attribute on a column family descriptor).

The 'new version behavior', undoes the limitations listed below whereby a `Delete` ALWAYS overshadows a `Put` if at the same location — i.e. same row, column family, qualifier and timestamp -- regardless of which arrived first. Version accounting is also changed as deleted versions are considered toward total version count. This is done to ensure results are not changed should a major compaction intercede. See `HBASE-15968` and linked issues for discussion.

Running with this new configuration currently costs; we factor the Cell MVCC on every compare so we burn more CPU. The slow down will depend. In testing we've seen between 0% and 25% degradation.

If replicating, it is advised that you run with the new serial replication feature (See `HBASE-94 65`; the serial replication feature did NOT make it into `hbase-2.0.0` but should arrive in a subsequent hbase-2.x release) as now the order in which Mutations arrive is a factor.

Current Limitations

The below limitations are addressed in hbase-2.0.0. See the section above, Optional New Version and Delete behavior in HBase-2.0.0.

Deletes mask Puts

Deletes mask puts, even puts that happened after the delete was entered. See HBASE-2256. Remember that a delete writes a tombstone, which only disappears after then next major compaction has run. Suppose you do a delete of everything `<= T`. After this you do a new put with a timestamp `<= T`. This put, even if it happened after the delete, will be masked by the delete tombstone. Performing the put will not fail, but when you do a get you will notice the put did have no effect. It will start working again after the major compaction has run. These issues should not be a problem if you use always-increasing versions for new puts to a row. But they can occur even if you do not care about time: just do delete and put immediately after each other, and there is some chance they happen within the same millisecond.

Major compactions change query results

...create three cell versions at t1, t2 and t3, with a maximum-versions setting of 2. So when getting all versions, only the values at t2 and t3 will be returned. But if you delete the version at t2 or t3, the one at t1 will appear again. Obviously, once a major

compaction has run, such behavior will not be the case anymore... (See *Garbage Collection* in Bending time in HBase.)

Sort Order

All data model operations HBase return data in sorted order. First by row, then by ColumnFamily, followed by column qualifier, and finally timestamp (sorted in reverse, so newest records are returned first).

Column Metadata

There is no store of column metadata outside of the internal KeyValue instances for a ColumnFamily. Thus, while HBase can support not only a wide number of columns per row, but a heterogeneous set of columns between rows as well, it is your responsibility to keep track of the column names.

The only way to get a complete set of columns that exist for a ColumnFamily is to process all the rows. For more information about how HBase stores data internally, see keyvalue.

Joins

Whether HBase supports joins is a common question on the dist-list, and there is a simple answer: it doesn't, at not least in the way that RDBMS' support them (e.g., with equi-joins or outer-joins in SQL). As has been illustrated in this chapter, the read data model operations in HBase are Get and Scan.

However, that doesn't mean that equivalent join functionality can't be supported in your application, but you have to do it yourself. The two primary strategies are either denormalizing the data upon writing to HBase, or to have lookup tables and do the join between HBase tables in your application or MapReduce code (and as RDBMS' demonstrate, there are several strategies for this depending on the size of the tables, e.g., nested loops vs. hash-joins). So which is the best approach? It depends on what you are trying to do, and as such there isn't a single answer that works for every use case.

ACID

See ACID Semantics. Lars Hofhansl has also written a note on ACID in HBase.

HBase and Schema Design

A good introduction on the strength and weaknesses modelling on the various non-rdbms datastores is to be found in Ian Varley's Master thesis, No Relation: The Mixed Blessings of Non-Relational Databases. It is a little dated now but a good background read if you have a moment on how HBase schema modeling differs from how it is done in an RDBMS. Also, read keyvalue for how HBase stores data internally, and the section on schema.casestudies.

The documentation on the Cloud Bigtable website, Designing Your Schema, is pertinent and nicely done and lessons learned there equally apply here in HBase land; just divide any quoted values by ~10 to get what works for HBase: e.g. where it says individual values can be ~10MBs in size, HBase can do similar — perhaps best to go smaller if you can — and where it says a maximum of 100 column families in Cloud Bigtable, think ~10 when modeling on HBase.

See also Robert Yokota's HBase Application Archetypes (an update on work done by other HBasers), for a helpful categorization of use cases that do well on top of the HBase model.

Schema Creation

HBase schemas can be created or updated using the Apache HBase Shell or by using Admin in the Java API.

Tables must be disabled when making ColumnFamily modifications, for example:

```
Configuration config = HBaseConfiguration.create();
Admin admin = new Admin(conf);
TableName table = TableName.valueOf("myTable");

admin.disableTable(table);

HColumnDescriptor cf1 = ...;
admin.addColumn(table, cf1);      // adding new ColumnFamily
HColumnDescriptor cf2 = ...;
admin.modifyColumn(table, cf2);    // modifying existing ColumnFamily

admin.enableTable(table);
```

See client dependencies for more information about configuring client connections.

> Online schema changes are supported in the 0.92.x codebase, but the 0.90.x codebase requires the table to be disabled.

Schema Updates

When changes are made to either Tables or ColumnFamilies (e.g. region size, block size), these changes take effect the next time there is a major compaction and the StoreFiles get re-written.

See store for more information on StoreFiles.

Table Schema Rules Of Thumb

There are many different data sets, with different access patterns and service-level expectations. Therefore, these rules of thumb are only an overview. Read the rest of this chapter to get more details after you have gone through this list.

- Aim to have regions sized between 10 and 50 GB.
- Aim to have cells no larger than 10 MB, or 50 MB if you use mob. Otherwise, consider storing your cell data in HDFS and store a pointer to the data in HBase.
- A typical schema has between 1 and 3 column families per table. HBase tables should not be designed to mimic RDBMS tables.
- Around 50-100 regions is a good number for a table with 1 or 2 column families. Remember that a region is a contiguous segment of a column family.
- Keep your column family names as short as possible. The column family names are stored for every value (ignoring prefix encoding). They should not be self-documenting and descriptive like in a typical RDBMS.
- If you are storing time-based machine data or logging information, and the row key is based on device ID or service ID plus time, you can end up with a pattern where older data regions never have additional writes beyond a certain age. In this type of situation, you end up with a small number of active regions and a large number of older regions which have no new writes. For these situations, you can tolerate a larger number of regions because your resource consumption is driven by the active regions only.

- If only one column family is busy with writes, only that column family accomulates memory. Be aware of write patterns when allocating resources.

RegionServer Sizing Rules of Thumb

Lars Hofhansl wrote a great blog post about RegionServer memory sizing. The upshot is that you probably need more memory than you think you need. He goes into the impact of region size, memstore size, HDFS replication factor, and other things to check.

> *"Personally I would place the maximum disk space per machine that can be served exclusively with HBase around 6T, unless you have a very read-heavy workload. In that case the Java heap should be 32GB (20G regions, 128M memstores, the rest defaults).*
>
> *— Lars Hofhansl http://hadoop-hbase.blogspot.com/2013/01/hbase-region-server-memory-sizing.html"*

On the number of column families

HBase currently does not do well with anything above two or three column families so keep the number of column families in your schema low. Currently, flushing is done on a per Region basis so if one column family is carrying the bulk of the data bringing on flushes, the adjacent families will also be flushed even though the amount of data they carry is small. When many column families exist the flushing interaction can make for a bunch of needless i/o (To be addressed by changing flushing to work on a per column family basis). In addition, compactions triggered at table/region level will happen per store too.

Try to make do with one column family if you can in your schemas. Only introduce a second and third column family in the case where data access is usually column scoped; i.e. you query one column family or the other but usually not both at the one time.

Cardinality of ColumnFamilies

Where multiple ColumnFamilies exist in a single table, be aware of the cardinality (i.e., number of rows). If ColumnFamilyA has 1 million rows and ColumnFamilyB has 1 billion rows, ColumnFamilyA's data will likely be spread across many, many regions (and RegionServers). This makes mass scans for ColumnFamilyA less efficient.

Rowkey Design

Hotspotting

Rows in HBase are sorted lexicographically by row key. This design optimizes for scans, allowing you to store related rows, or rows that will be read together, near each other. However, poorly designed row keys are a common source of **hotspotting**. Hotspotting occurs when a large amount of client traffic is directed at one node, or only a few nodes, of a cluster. This traffic may represent reads, writes, or other operations. The traffic overwhelms the single machine responsible for hosting that region, causing performance degradation and potentially leading to region unavailability. This can also have adverse effects on other regions hosted by the same region server as that host is unable to service the requested load. It is important to design data access patterns such that the cluster is fully and evenly utilized.

To prevent hotspotting on writes, design your row keys such that rows that truly do need to be in the same region are, but in the bigger picture, data is being written to multiple regions across the cluster, rather than one at a time. Some common techniques for avoiding hotspotting are described below, along with some of their advantages and drawbacks.

Salting

Salting in this sense has nothing to do with cryptography, but refers to adding random data to the start of a row key. In this case, salting refers to adding a randomly-assigned prefix to the row key to cause it to sort differently than it otherwise would. The number of possible prefixes correspond to the number of regions you want to spread the data across. Salting can be helpful if you have a few "hot" row key patterns which come up over and over amongst other more evenly-distributed rows. Consider the following example, which shows that salting can spread write load across multiple RegionServers, and illustrates some of the negative implications for reads.

Salting Example:

Suppose you have the following list of row keys, and your table is split such that there is one region for each letter of the alphabet. Prefix 'a' is one region, prefix 'b' is another. In this table, all rows starting with 'f' are in the same region. This example focuses on rows with keys like the following:

```
foo0001
```

```
foo0002
foo0003
foo0004
```

Now, imagine that you would like to spread these across four different regions. You decide to use four different salts: `a`, `b`, `c`, and `d`. In this scenario, each of these letter prefixes will be on a different region. After applying the salts, you have the following rowkeys instead. Since you can now write to four separate regions, you theoretically have four times the throughput when writing that you would have if all the writes were going to the same region.

```
a-foo0003
b-foo0001
c-foo0004
d-foo0002
```

Then, if you add another row, it will randomly be assigned one of the four possible salt values and end up near one of the existing rows.

```
a-foo0003
b-foo0001
c-foo0003
c-foo0004
d-foo0002
```

Since this assignment will be random, you will need to do more work if you want to retrieve the rows in lexicographic order. In this way, salting attempts to increase throughput on writes, but has a cost during reads.

Hashing

Instead of a random assignment, you could use a one-way **hash** that would cause a given row to always be "salted" with the same prefix, in a way that would spread the load across the RegionServers, but allow for predictability during reads. Using a deterministic hash allows the client to reconstruct the complete rowkey and use a Get operation to retrieve that row as normal.

Hashing Example:

Given the same situation in the salting example above, you could instead apply a one-way hash that would cause the row with key `foo0003` to always, and predictably, receive the `a`

prefix. Then, to retrieve that row, you would already know the key. You could also optimize things so that certain pairs of keys were always in the same region, for instance.

Reversing the Key

A third common trick for preventing hotspotting is to reverse a fixed-width or numeric row key so that the part that changes the most often (the least significant digit) is first. This effectively randomizes row keys, but sacrifices row ordering properties.

See https://communities.intel.com/community/itpeernetwork/datastack/blog/2013/11/10/discussion-on-designing-hbase-tables, and article on Salted Tables from the Phoenix project, and the discussion in the comments of HBASE-11682 for more information about avoiding hotspotting.

Monotonically Increasing Row Keys/Timeseries Data

In the HBase chapter of Tom White's book Hadoop: The Definitive Guide (O'Reilly) there is a an optimization note on watching out for a phenomenon where an import process walks in lock-step with all clients in concert pounding one of the table's regions (and thus, a single node), then moving onto the next region, etc. With monotonically increasing row-keys (i.e., using a timestamp), this will happen. See this comic by IKai Lan on why monotonically increasing row keys are problematic in BigTable-like datastores: monotonically increasing values are bad. The pile-up on a single region brought on by monotonically increasing keys can be mitigated by randomizing the input records to not be in sorted order, but in general it's best to avoid using a timestamp or a sequence (e.g. 1, 2, 3) as the row-key.

If you do need to upload time series data into HBase, you should study OpenTSDB as a successful example. It has a page describing the schema it uses in HBase. The key format in OpenTSDB is effectively [metric_type][event_timestamp], which would appear at first glance to contradict the previous advice about not using a timestamp as the key. However, the difference is that the timestamp is not in the *lead* position of the key, and the design assumption is that there are dozens or hundreds (or more) of different metric types. Thus, even with a continual stream of input data with a mix of metric types, the Puts are distributed across various points of regions in the table.

See schema.casestudies for some rowkey design examples.

Try to minimize row and column sizes

In HBase, values are always freighted with their coordinates; as a cell value passes through the system, it'll be accompanied by its row, column name, and timestamp - always. If your rows and column names are large, especially compared to the size of the cell value, then you may run up against some interesting scenarios. One such is the case described by Marc Limotte at the tail of HBASE-3551 (recommended!). Therein, the indices that are kept on HBase storefiles (see HFile Format) to facilitate random access may end up occupying large chunks of the HBase allotted RAM because the cell value coordinates are large. Mark in the above cited comment suggests upping the block size so entries in the store file index happen at a larger interval or modify the table schema so it makes for smaller rows and column names. Compression will also make for larger indices. See the thread a question storefileIndexSize up on the user mailing list.

Most of the time small inefficiencies don't matter all that much. Unfortunately, this is a case where they do. Whatever patterns are selected for ColumnFamilies, attributes, and rowkeys they could be repeated several billion times in your data.

See keyvalue for more information on HBase stores data internally to see why this is important.

Column Families

Try to keep the ColumnFamily names as small as possible, preferably one character (e.g. "d" for data/default).

See KeyValue for more information on how HBase stores data internally.

Attributes

Although verbose attribute names (e.g., "myVeryImportantAttribute") are easier to read, prefer shorter attribute names (e.g., "via") to store in HBase.

See keyvalue for more information on HBase stores data internally to see why this is important.

Rowkey Length

Keep them as short as is reasonable such that they can still be useful for required data access (e.g. Get vs. Scan). A short key that is useless for data access is not better than a longer key with better get/scan properties. Expect tradeoffs when designing rowkeys.

Byte Patterns

A long is 8 bytes. You can store an unsigned number up to 18,446,744,073,709,551,615 in those eight bytes. If you stored this number as a String — presuming a byte per character — you need nearly 3x the bytes.

Not convinced? Below is some sample code that you can run on your own.

```
// long
//
long l = 1234567890L;
byte[] lb = Bytes.toBytes(l);
System.out.println("long bytes length: " + lb.length);   // returns 8

String s = String.valueOf(l);
byte[] sb = Bytes.toBytes(s);
System.out.println("long as string length: " + sb.length);    // returns 10

// hash
//
MessageDigest md = MessageDigest.getInstance("MD5");
byte[] digest = md.digest(Bytes.toBytes(s));
System.out.println("md5 digest bytes length: " + digest.length);    // returns 16

String sDigest = new String(digest);
byte[] sbDigest = Bytes.toBytes(sDigest);
System.out.println("md5 digest as string length: " + sbDigest.length);    // retu
rns 26
```

Unfortunately, using a binary representation of a type will make your data harder to read outside of your code. For example, this is what you will see in the shell when you increment a value:

```
hbase(main):001:0> incr 't', 'r', 'f:q', 1
COUNTER VALUE = 1

hbase(main):002:0> get 't', 'r'
COLUMN                                        CELL
 f:q                                          timestamp=1369163040570, value=\x00
\x00\x00\x00\x00\x00\x00\x01
1 row(s) in 0.0310 seconds
```

The shell makes a best effort to print a string, and it this case it decided to just print the hex. The same will happen to your row keys inside the region names. It can be okay if you know what's being stored, but it might also be unreadable if arbitrary data can be put in the same cells. This is the main trade-off.

Reverse Timestamps

> HBASE-4811 implements an API to scan a table or a range within a table in reverse, reducing the need to optimize your schema for forward or reverse scanning. This feature is available in HBase 0.98 and later. See Scan.setReversed() for more information.

A common problem in database processing is quickly finding the most recent version of a value. A technique using reverse timestamps as a part of the key can help greatly with a special case of this problem. Also found in the HBase chapter of Tom White's book Hadoop: The Definitive Guide (O'Reilly), the technique involves appending (`Long.MAX_VALUE - timestamp`) to the end of any key, e.g. [key][reverse_timestamp].

The most recent value for [key] in a table can be found by performing a Scan for [key] and obtaining the first record. Since HBase keys are in sorted order, this key sorts before any older row-keys for [key] and thus is first.

This technique would be used instead of using Number of Versions where the intent is to hold onto all versions "forever" (or a very long time) and at the same time quickly obtain access to any other version by using the same Scan technique.

Rowkeys and ColumnFamilies

Rowkeys are scoped to ColumnFamilies. Thus, the same rowkey could exist in each ColumnFamily that exists in a table without collision.

Immutability of Rowkeys

Rowkeys cannot be changed. The only way they can be "changed" in a table is if the row is deleted and then re-inserted. This is a fairly common question on the HBase dist-list so it pays to get the rowkeys right the first time (and/or before you've inserted a lot of data).

Relationship Between RowKeys and Region Splits

If you pre-split your table, it is *critical* to understand how your rowkey will be distributed across the region boundaries. As an example of why this is important, consider the example of using displayable hex characters as the lead position of the key (e.g., "0000000000000000" to "ffffffffffffffff"). Running those key ranges through `Bytes.split`

(which is the split strategy used when creating regions in `Admin.createTable(byte[] startKey, byte[] endKey, numRegions)` for 10 regions will generate the following splits...

```
48 48 48 48 48 48 48 48 48 48 48 48 48 48 48 48                                 //
0
54 -10 -10 -10 -10 -10 -10 -10 -10 -10 -10 -10 -10 -10 -10 -10                  //
6
61 -67 -67 -67 -67 -67 -67 -67 -67 -67 -67 -67 -67 -67 -67 -68                  //
=
68 -124 -124 -124 -124 -124 -124 -124 -124 -124 -124 -124 -124 -124 -124 -126   //
D
75 75 75 75 75 75 75 75 75 75 75 75 75 75 72                                   //
K
82 18 18 18 18 18 18 18 18 18 18 18 18 18 14                                   //
R
88 -40 -40 -40 -40 -40 -40 -40 -40 -40 -40 -40 -40 -40 -40 -44                  //
X
95 -97 -97 -97 -97 -97 -97 -97 -97 -97 -97 -97 -97 -97 -97 -102                //
_
102 102 102 102 102 102 102 102 102 102 102 102 102 102 102 102                //
f
```

(note: the lead byte is listed to the right as a comment.) Given that the first split is a '0' and the last split is an 'f', everything is great, right? Not so fast.

The problem is that all the data is going to pile up in the first 2 regions and the last region thus creating a "lumpy" (and possibly "hot") region problem. To understand why, refer to an ASCII Table. '0' is byte 48, and 'f' is byte 102, but there is a huge gap in byte values (bytes 58 to 96) that will *never appear in this keyspace* because the only values are [0-9] and [a-f]. Thus, the middle regions will never be used. To make pre-splitting work with this example keyspace, a custom definition of splits (i.e., and not relying on the built-in split method) is required.

Lesson #1: Pre-splitting tables is generally a best practice, but you need to pre-split them in such a way that all the regions are accessible in the keyspace. While this example demonstrated the problem with a hex-key keyspace, the same problem can happen with *any* keyspace. Know your data.

Lesson #2: While generally not advisable, using hex-keys (and more generally, displayable data) can still work with pre-split tables as long as all the created regions are accessible in the keyspace.

To conclude this example, the following is an example of how appropriate splits can be pre-created for hex-keys:

```
public static boolean createTable(Admin admin, HTableDescriptor table, byte[][] s
plits)
throws IOException {
  try {
    admin.createTable( table, splits );
    return true;
  } catch (TableExistsException e) {
    logger.info("table " + table.getNameAsString() + " already exists");
    // the table already exists...
    return false;
  }
}

public static byte[][] getHexSplits(String startKey, String endKey, int numRegion
s) {
  byte[][] splits = new byte[numRegions-1][];
  BigInteger lowestKey = new BigInteger(startKey, 16);
  BigInteger highestKey = new BigInteger(endKey, 16);
  BigInteger range = highestKey.subtract(lowestKey);
  BigInteger regionIncrement = range.divide(BigInteger.valueOf(numRegions));
  lowestKey = lowestKey.add(regionIncrement);
  for(int i=0; i < numRegions-1;i++) {
    BigInteger key = lowestKey.add(regionIncrement.multiply(BigInteger.valueOf
(i)));
    byte[] b = String.format("%016x", key).getBytes();
    splits[i] = b;
  }
  return splits;
}
```

Number of Versions

Maximum Number of Versions

The maximum number of row versions to store is configured per column family via HColumnDescriptor. The default for max versions is 1. This is an important parameter because as described in Data Model section HBase does *not* overwrite row values, but rather stores different values per row by time (and qualifier). Excess versions are removed during major compactions. The number of max versions may need to be increased or decreased depending on application needs.

It is not recommended setting the number of max versions to an exceedingly high level (e.g., hundreds or more) unless those old values are very dear to you because this will

greatly increase StoreFile size.

Minimum Number of Versions

Like maximum number of row versions, the minimum number of row versions to keep is configured per column family via ColumnFamilyDescriptorBuilder. The default for min versions is 0, which means the feature is disabled. The minimum number of row versions parameter is used together with the time-to-live parameter and can be combined with the number of row versions parameter to allow configurations such as "keep the last T minutes worth of data, at most N versions, *but keep at least M versions around*" (where M is the value for minimum number of row versions, M<N). This parameter should only be set when time-to-live is enabled for a column family and must be less than the number of row versions.

Supported Datatypes

HBase supports a "bytes-in/bytes-out" interface via Put and Result, so anything that can be converted to an array of bytes can be stored as a value. Input could be strings, numbers, complex objects, or even images as long as they can rendered as bytes.

There are practical limits to the size of values (e.g., storing 10-50MB objects in HBase would probably be too much to ask); search the mailing list for conversations on this topic. All rows in HBase conform to the Data Model, and that includes versioning. Take that into consideration when making your design, as well as block size for the ColumnFamily.

Counters

One supported datatype that deserves special mention are "counters" (i.e., the ability to do atomic increments of numbers). See Increment in `Table`.

Synchronization on counters are done on the RegionServer, not in the client.

Joins

If you have multiple tables, don't forget to factor in the potential for Joins into the schema design.

Time To Live (TTL)

ColumnFamilies can set a TTL length in seconds, and HBase will automatically delete rows once the expiration time is reached. This applies to *all* versions of a row - even the current one. The TTL time encoded in the HBase for the row is specified in UTC.

Store files which contains only expired rows are deleted on minor compaction. Setting `hbase.store.delete.expired.storefile` to `false` disables this feature. Setting minimum number of versions to other than 0 also disables this.

See HColumnDescriptor for more information.

Recent versions of HBase also support setting time to live on a per cell basis. See HBASE-10560 for more information. Cell TTLs are submitted as an attribute on mutation requests (Appends, Increments, Puts, etc.) using Mutation#setTTL. If the TTL attribute is set, it will be applied to all cells updated on the server by the operation. There are two notable differences between cell TTL handling and ColumnFamily TTLs:

- Cell TTLs are expressed in units of milliseconds instead of seconds.
- A cell TTLs cannot extend the effective lifetime of a cell beyond a ColumnFamily level TTL setting.

Keeping Deleted Cells

By default, delete markers extend back to the beginning of time. Therefore, Get or Scan operations will not see a deleted cell (row or column), even when the Get or Scan operation indicates a time range before the delete marker was placed.

ColumnFamilies can optionally keep deleted cells. In this case, deleted cells can still be retrieved, as long as these operations specify a time range that ends before the timestamp of any delete that would affect the cells. This allows for point-in-time queries even in the presence of deletes.

Deleted cells are still subject to TTL and there will never be more than "maximum number of versions" deleted cells. A new "raw" scan options returns all deleted rows and the delete markers.

Change the Value of `KEEP_DELETED_CELLS` Using HBase Shell:

```
hbase> hbase> alter 't1', NAME => 'f1', KEEP_DELETED_CELLS => true
```

Change the Value of `KEEP_DELETED_CELLS` Using the API:

```
...
HColumnDescriptor.setKeepDeletedCells(true);
...
```

Let us illustrate the basic effect of setting the `KEEP_DELETED_CELLS` attribute on a table.

First, without:

```
create 'test', {NAME=>'e', VERSIONS=>2147483647}
put 'test', 'r1', 'e:c1', 'value', 10
put 'test', 'r1', 'e:c1', 'value', 12
put 'test', 'r1', 'e:c1', 'value', 14
delete 'test', 'r1', 'e:c1',  11

hbase(main):017:0> scan 'test', {RAW=>true, VERSIONS=>1000}
ROW                                                  COLUMN+CELL
 r1                                                  column=e:c1, timestamp=14, value
=value
 r1                                                  column=e:c1, timestamp=12, value
=value
 r1                                                  column=e:c1, timestamp=11, type=
DeleteColumn
 r1                                                  column=e:c1, timestamp=10, value
=value
1 row(s) in 0.0120 seconds

hbase(main):018:0> flush 'test'
0 row(s) in 0.0350 seconds

hbase(main):019:0> scan 'test', {RAW=>true, VERSIONS=>1000}
ROW                                                  COLUMN+CELL
 r1                                                  column=e:c1, timestamp=14, value
=value
 r1                                                  column=e:c1, timestamp=12, value
=value
 r1                                                  column=e:c1, timestamp=11, type=
DeleteColumn
1 row(s) in 0.0120 seconds

hbase(main):020:0> major_compact 'test'
0 row(s) in 0.0260 seconds

hbase(main):021:0> scan 'test', {RAW=>true, VERSIONS=>1000}
ROW                                                  COLUMN+CELL
```

```
 r1                                                  column=e:c1, timestamp=14, value
=value
 r1                                                  column=e:c1, timestamp=12, value
=value
1 row(s) in 0.0120 seconds
```

Notice how delete cells are let go.

Now let's run the same test only with `KEEP_DELETED_CELLS` set on the table (you can do table or per-column-family):

```
hbase(main):005:0> create 'test', {NAME=>'e', VERSIONS=>2147483647, KEEP_DELETED_
CELLS => true}
0 row(s) in 0.2160 seconds

=> Hbase::Table - test
hbase(main):006:0> put 'test', 'r1', 'e:c1', 'value', 10
0 row(s) in 0.1070 seconds

hbase(main):007:0> put 'test', 'r1', 'e:c1', 'value', 12
0 row(s) in 0.0140 seconds

hbase(main):008:0> put 'test', 'r1', 'e:c1', 'value', 14
0 row(s) in 0.0160 seconds

hbase(main):009:0> delete 'test', 'r1', 'e:c1',  11
0 row(s) in 0.0290 seconds

hbase(main):010:0> scan 'test', {RAW=>true, VERSIONS=>1000}
ROW
COLUMN+CELL
 r1
column=e:c1, timestamp=14, value=value
 r1
column=e:c1, timestamp=12, value=value
 r1
column=e:c1, timestamp=11, type=DeleteColumn
 r1
column=e:c1, timestamp=10, value=value
1 row(s) in 0.0550 seconds

hbase(main):011:0> flush 'test'
0 row(s) in 0.2780 seconds

hbase(main):012:0> scan 'test', {RAW=>true, VERSIONS=>1000}
ROW
COLUMN+CELL
 r1
column=e:c1, timestamp=14, value=value
 r1
column=e:c1, timestamp=12, value=value
```

```
 r1
column=e:c1, timestamp=11, type=DeleteColumn
 r1
column=e:c1, timestamp=10, value=value
1 row(s) in 0.0620 seconds

hbase(main):013:0> major_compact 'test'
0 row(s) in 0.0530 seconds

hbase(main):014:0> scan 'test', {RAW=>true, VERSIONS=>1000}
ROW
COLUMN+CELL
 r1
column=e:c1, timestamp=14, value=value
 r1
column=e:c1, timestamp=12, value=value
 r1
column=e:c1, timestamp=11, type=DeleteColumn
 r1
column=e:c1, timestamp=10, value=value
1 row(s) in 0.0650 seconds
```

KEEP*DELETED_CELLS is to avoid removing Cells from HBase when the _only* reason to remove them is the delete marker. So with KEEP_DELETED_CELLS enabled deleted cells would get removed if either you write more versions than the configured max, or you have a TTL and Cells are in excess of the configured timeout, etc.

Secondary Indexes and Alternate Query Paths

This section could also be titled "what if my table rowkey looks like *this* but I also want to query my table like *that*." A common example on the dist-list is where a row-key is of the format "user-timestamp" but there are reporting requirements on activity across users for certain time ranges. Thus, selecting by user is easy because it is in the lead position of the key, but time is not.

There is no single answer on the best way to handle this because it depends on...

- Number of users
- Data size and data arrival rate
- Flexibility of reporting requirements (e.g., completely ad-hoc date selection vs. pre-configured ranges)
- Desired execution speed of query (e.g., 90 seconds may be reasonable to some for an ad-hoc report, whereas it may be too long for others)

and solutions are also influenced by the size of the cluster and how much processing power you have to throw at the solution. Common techniques are in sub-sections below. This is a comprehensive, but not exhaustive, list of approaches.

It should not be a surprise that secondary indexes require additional cluster space and processing. This is precisely what happens in an RDBMS because the act of creating an alternate index requires both space and processing cycles to update. RDBMS products are more advanced in this regard to handle alternative index management out of the box. However, HBase scales better at larger data volumes, so this is a feature trade-off.

Pay attention to Apache HBase Performance Tuning when implementing any of these approaches.

Additionally, see the David Butler response in this dist-list thread HBase, mail # user - Stargate+hbase

Filter Query

Depending on the case, it may be appropriate to use Client Request Filters. In this case, no secondary index is created. However, don't try a full-scan on a large table like this from an application (i.e., single-threaded client).

Periodic-Update Secondary Index

A secondary index could be created in another table which is periodically updated via a MapReduce job. The job could be executed intra-day, but depending on load-strategy it could still potentially be out of sync with the main data table.

See mapreduce.example.readwrite for more information.

Dual-Write Secondary Index

Another strategy is to build the secondary index while publishing data to the cluster (e.g., write to data table, write to index table). If this is approach is taken after a data table already exists, then bootstrapping will be needed for the secondary index with a MapReduce job (see secondary.indexes.periodic).

Summary Tables

Where time-ranges are very wide (e.g., year-long report) and where the data is voluminous, summary tables are a common approach. These would be generated with MapReduce jobs into another table.

See mapreduce.example.summary for more information.

Coprocessor Secondary Index

Coprocessors act like RDBMS triggers. These were added in 0.92. For more information, see coprocessors

Constraints

HBase currently supports 'constraints' in traditional (SQL) database parlance. The advised usage for Constraints is in enforcing business rules for attributes in the table (e.g. make sure values are in the range 1-10). Constraints could also be used to enforce referential integrity, but this is strongly discouraged as it will dramatically decrease the write throughput of the tables where integrity checking is enabled. Extensive documentation on using Constraints can be found at Constraint since version 0.94.

Schema Design Case Studies

The following will describe some typical data ingestion use-cases with HBase, and how the rowkey design and construction can be approached. Note: this is just an illustration of potential approaches, not an exhaustive list. Know your data, and know your processing requirements.

It is highly recommended that you read the rest of the HBase and Schema Design first, before reading these case studies.

The following case studies are described:

- Log Data / Timeseries Data
- Log Data / Timeseries on Steroids
- Customer/Order

- Tall/Wide/Middle Schema Design
- List Data

Case Study - Log Data and Timeseries Data

Assume that the following data elements are being collected.

- Hostname
- Timestamp
- Log event
- Value/message

We can store them in an HBase table called LOG_DATA, but what will the rowkey be? From these attributes the rowkey will be some combination of hostname, timestamp, and log-event - but what specifically?

Timestamp In The Rowkey Lead Position

The rowkey `[timestamp][hostname][log-event]` suffers from the monotonically increasing rowkey problem described in Monotonically Increasing Row Keys/Timeseries Data.

There is another pattern frequently mentioned in the dist-lists about "bucketing" timestamps, by performing a mod operation on the timestamp. If time-oriented scans are important, this could be a useful approach. Attention must be paid to the number of buckets, because this will require the same number of scans to return results.

```
long bucket = timestamp % numBuckets;
```

to construct:

```
[bucket][timestamp][hostname][log-event]
```

As stated above, to select data for a particular timerange, a Scan will need to be performed for each bucket. 100 buckets, for example, will provide a wide distribution in the keyspace but it will require 100 Scans to obtain data for a single timestamp, so there are trade-offs.

Host In The Rowkey Lead Position

The rowkey `[hostname][log-event][timestamp]` is a candidate if there is a large-ish number of hosts to spread the writes and reads across the keyspace. This approach would be useful if scanning by hostname was a priority.

Timestamp, or Reverse Timestamp?

If the most important access path is to pull most recent events, then storing the timestamps as reverse-timestamps (e.g., `timestamp = Long.MAX_VALUE – timestamp`) will create the property of being able to do a Scan on `[hostname][log-event]` to obtain the most recently captured events.

Neither approach is wrong, it just depends on what is most appropriate for the situation.

> HBASE-4811 implements an API to scan a table or a range within a table in reverse, reducing the need to optimize your schema for forward or reverse scanning. This feature is available in HBase 0.98 and later. See Scan.setReversed() for more information.

Variable Length or Fixed Length Rowkeys?

It is critical to remember that rowkeys are stamped on every column in HBase. If the hostname is `a` and the event type is `e1` then the resulting rowkey would be quite small. However, what if the ingested hostname is `myserver1.mycompany.com` and the event type is `com.package1.subpackage2.subsubpackage3.ImportantService`?

It might make sense to use some substitution in the rowkey. There are at least two approaches: hashed and numeric. In the Hostname In The Rowkey Lead Position example, it might look like this:

Composite Rowkey With Hashes:

- [MD5 hash of hostname] = 16 bytes
- [MD5 hash of event-type] = 16 bytes
- [timestamp] = 8 bytes

Composite Rowkey With Numeric Substitution:

For this approach another lookup table would be needed in addition to LOG_DATA, called LOG_TYPES. The rowkey of LOG_TYPES would be:

- `[type]` (e.g., byte indicating hostname vs. event-type)
- `[bytes]` variable length bytes for raw hostname or event-type.

A column for this rowkey could be a long with an assigned number, which could be obtained by using an HBase counter

So the resulting composite rowkey would be:

- [substituted long for hostname] = 8 bytes
- [substituted long for event type] = 8 bytes
- [timestamp] = 8 bytes

In either the Hash or Numeric substitution approach, the raw values for hostname and event-type can be stored as columns.

Case Study - Log Data and Timeseries Data on Steroids

This effectively is the OpenTSDB approach. What OpenTSDB does is re-write data and pack rows into columns for certain time-periods. For a detailed explanation, see: http://opentsdb.net/schema.html, and Lessons Learned from OpenTSDB from HBaseCon2012.

But this is how the general concept works: data is ingested, for example, in this manner...

```
[hostname][log-event][timestamp1]
[hostname][log-event][timestamp2]
[hostname][log-event][timestamp3]
```

with separate rowkeys for each detailed event, but is re-written like this...

```
[hostname][log-event][timerange]
```

and each of the above events are converted into columns stored with a time-offset relative to the beginning timerange (e.g., every 5 minutes). This is obviously a very advanced processing technique, but HBase makes this possible.

Case Study - Customer/Order

Assume that HBase is used to store customer and order information. There are two core record-types being ingested: a Customer record type, and Order record type.

The Customer record type would include all the things that you'd typically expect:

- Customer number
- Customer name
- Address (e.g., city, state, zip)
- Phone numbers, etc.

The Order record type would include things like:

- Customer number
- Order number
- Sales date
- A series of nested objects for shipping locations and line-items (see Order Object Design for details)

Assuming that the combination of customer number and sales order uniquely identify an order, these two attributes will compose the rowkey, and specifically a composite key such as:

```
[customer number][order number]
```

for an ORDER table. However, there are more design decisions to make: are the *raw* values the best choices for rowkeys?

The same design questions in the Log Data use-case confront us here. What is the keyspace of the customer number, and what is the format (e.g., numeric? alphanumeric?) As it is advantageous to use fixed-length keys in HBase, as well as keys that can support a reasonable spread in the keyspace, similar options appear:

Composite Rowkey With Hashes:

- [MD5 of customer number] = 16 bytes

- [MD5 of order number] = 16 bytes

Composite Numeric/Hash Combo Rowkey:

- [substituted long for customer number] = 8 bytes
- [MD5 of order number] = 16 bytes

Single Table? Multiple Tables?

A traditional design approach would have separate tables for CUSTOMER and SALES. Another option is to pack multiple record types into a single table (e.g., CUSTOMER++).

Customer Record Type Rowkey:

- [customer-id]
- [type] = type indicating `1` for customer record type

Order Record Type Rowkey:

- [customer-id]
- [type] = type indicating `2` for order record type
- [order]

The advantage of this particular CUSTOMER++ approach is that organizes many different record-types by customer-id (e.g., a single scan could get you everything about that customer). The disadvantage is that it's not as easy to scan for a particular record-type.

Order Object Design

Now we need to address how to model the Order object. Assume that the class structure is as follows:

Order

an Order can have multiple ShippingLocations

LineItem

a ShippingLocation can have multiple LineItems

there are multiple options on storing this data.

Completely Normalized

With this approach, there would be separate tables for ORDER, SHIPPING_LOCATION, and LINE_ITEM.

The ORDER table's rowkey was described above: schema.casestudies.custorder

The SHIPPING_LOCATION's composite rowkey would be something like this:

- `[order-rowkey]`
- `[shipping location number]` (e.g., 1st location, 2nd, etc.)

The LINE_ITEM table's composite rowkey would be something like this:

- `[order-rowkey]`
- `[shipping location number]` (e.g., 1st location, 2nd, etc.)
- `[line item number]` (e.g., 1st lineitem, 2nd, etc.)

Such a normalized model is likely to be the approach with an RDBMS, but that's not your only option with HBase. The cons of such an approach is that to retrieve information about any Order, you will need:

- Get on the ORDER table for the Order
- Scan on the SHIPPING_LOCATION table for that order to get the ShippingLocation instances
- Scan on the LINE_ITEM for each ShippingLocation

granted, this is what an RDBMS would do under the covers anyway, but since there are no joins in HBase you're just more aware of this fact.

Single Table With Record Types

With this approach, there would exist a single table ORDER that would contain

The Order rowkey was described above: schema.casestudies.custorder

- `[order-rowkey]`
- `[ORDER record type]`

The ShippingLocation composite rowkey would be something like this:

- `[order-rowkey]`
- `[SHIPPING record type]`
- `[shipping location number]` (e.g., 1st location, 2nd, etc.)

The LineItem composite rowkey would be something like this:

- `[order-rowkey]`
- `[LINE record type]`
- `[shipping location number]` (e.g., 1st location, 2nd, etc.)
- `[line item number]` (e.g., 1st lineitem, 2nd, etc.)

Denormalized

A variant of the Single Table With Record Types approach is to denormalize and flatten some of the object hierarchy, such as collapsing the ShippingLocation attributes onto each LineItem instance.

The LineItem composite rowkey would be something like this:

- `[order-rowkey]`
- `[LINE record type]`
- `[line item number]` (e.g., 1st lineitem, 2nd, etc., care must be taken that there are unique across the entire order)

and the LineItem columns would be something like this:

- itemNumber
- quantity
- price
- shipToLine1 (denormalized from ShippingLocation)
- shipToLine2 (denormalized from ShippingLocation)
- shipToCity (denormalized from ShippingLocation)
- shipToState (denormalized from ShippingLocation)

- shipToZip (denormalized from ShippingLocation)

The pros of this approach include a less complex object hierarchy, but one of the cons is that updating gets more complicated in case any of this information changes.

Object BLOB

With this approach, the entire Order object graph is treated, in one way or another, as a BLOB. For example, the ORDER table's rowkey was described above: schema.casestudies.custorder, and a single column called "order" would contain an object that could be deserialized that contained a container Order, ShippingLocations, and LineItems.

There are many options here: JSON, XML, Java Serialization, Avro, Hadoop Writables, etc. All of them are variants of the same approach: encode the object graph to a byte-array. Care should be taken with this approach to ensure backward compatibility in case the object model changes such that older persisted structures can still be read back out of HBase.

Pros are being able to manage complex object graphs with minimal I/O (e.g., a single HBase Get per Order in this example), but the cons include the aforementioned warning about backward compatibility of serialization, language dependencies of serialization (e.g., Java Serialization only works with Java clients), the fact that you have to deserialize the entire object to get any piece of information inside the BLOB, and the difficulty in getting frameworks like Hive to work with custom objects like this.

Case Study - "Tall/Wide/Middle" Schema Design Smackdown

This section will describe additional schema design questions that appear on the dist-list, specifically about tall and wide tables. These are general guidelines and not laws - each application must consider its own needs.

Rows vs. Versions

A common question is whether one should prefer rows or HBase's built-in-versioning. The context is typically where there are "a lot" of versions of a row to be retained (e.g., where it is significantly above the HBase default of 1 max versions). The rows-approach would require storing a timestamp in some portion of the rowkey so that they would not overwrite with each successive update.

Preference: Rows (generally speaking).

Rows vs. Columns

Another common question is whether one should prefer rows or columns. The context is typically in extreme cases of wide tables, such as having 1 row with 1 million attributes, or 1 million rows with 1 columns apiece.

Preference: Rows (generally speaking). To be clear, this guideline is in the context is in extremely wide cases, not in the standard use-case where one needs to store a few dozen or hundred columns. But there is also a middle path between these two options, and that is "Rows as Columns."

Rows as Columns

The middle path between Rows vs. Columns is packing data that would be a separate row into columns, for certain rows. OpenTSDB is the best example of this case where a single row represents a defined time-range, and then discrete events are treated as columns. This approach is often more complex, and may require the additional complexity of re-writing your data, but has the advantage of being I/O efficient. For an overview of this approach, see schema.casestudies.log-steroids.

Case Study - List Data

The following is an exchange from the user dist-list regarding a fairly common question: how to handle per-user list data in Apache HBase.

- QUESTION *

We're looking at how to store a large amount of (per-user) list data in HBase, and we were trying to figure out what kind of access pattern made the most sense. One option is store the majority of the data in a key, so we could have something like:

```
<FixedWidthUserName><FixedWidthValueId1>:"" (no value)
<FixedWidthUserName><FixedWidthValueId2>:"" (no value)
<FixedWidthUserName><FixedWidthValueId3>:"" (no value)
```

The other option we had was to do this entirely using:

```
<FixedWidthUserName><FixedWidthPageNum0>:<FixedWidthLength><FixedIdNextPageNum><V
alueId1><ValueId2><ValueId3>...
```

```
<FixedWidthUserName><FixedWidthPageNum1>:<FixedWidthLength><FixedIdNextPageNum><V
alueId1><ValueId2><ValueId3>...
```

where each row would contain multiple values. So in one case reading the first thirty values would be:

```
scan { STARTROW => 'FixedWidthUsername' LIMIT => 30}
```

And in the second case it would be

```
get 'FixedWidthUserName\x00\x00\x00\x00'
```

The general usage pattern would be to read only the first 30 values of these lists, with infrequent access reading deeper into the lists. Some users would have ⇐ 30 total values in these lists, and some users would have millions (i.e. power-law distribution)

The single-value format seems like it would take up more space on HBase, but would offer some improved retrieval / pagination flexibility. Would there be any significant performance advantages to be able to paginate via gets vs paginating with scans?

My initial understanding was that doing a scan should be faster if our paging size is unknown (and caching is set appropriately), but that gets should be faster if we'll always need the same page size. I've ended up hearing different people tell me opposite things about performance. I assume the page sizes would be relatively consistent, so for most use cases we could guarantee that we only wanted one page of data in the fixed-page-length case. I would also assume that we would have infrequent updates, but may have inserts into the middle of these lists (meaning we'd need to update all subsequent rows).

Thanks for help / suggestions / follow-up questions.

- ANSWER *

If I understand you correctly, you're ultimately trying to store triples in the form "user, valueid, value", right? E.g., something like:

```
"user123, firstname, Paul",
"user234, lastname, Smith"
```

(But the usernames are fixed width, and the valueids are fixed width).

And, your access pattern is along the lines of: "for user X, list the next 30 values, starting with valueid Y". Is that right? And these values should be returned sorted by valueid?

The tl;dr version is that you should probably go with one row per user+value, and not build a complicated intra-row pagination scheme on your own unless you're really sure it is needed.

Your two options mirror a common question people have when designing HBase schemas: should I go "tall" or "wide"? Your first schema is "tall": each row represents one value for one user, and so there are many rows in the table for each user; the row key is user + valueid, and there would be (presumably) a single column qualifier that means "the value". This is great if you want to scan over rows in sorted order by row key (thus my question above, about whether these ids are sorted correctly). You can start a scan at any user+valueid, read the next 30, and be done. What you're giving up is the ability to have transactional guarantees around all the rows for one user, but it doesn't sound like you need that. Doing it this way is generally recommended (see here).

Your second option is "wide": you store a bunch of values in one row, using different qualifiers (where the qualifier is the valueid). The simple way to do that would be to just store ALL values for one user in a single row. I'm guessing you jumped to the "paginated" version because you're assuming that storing millions of columns in a single row would be bad for performance, which may or may not be true; as long as you're not trying to do too much in a single request, or do things like scanning over and returning all of the cells in the row, it shouldn't be fundamentally worse. The client has methods that allow you to get specific slices of columns.

Note that neither case fundamentally uses more disk space than the other; you're just "shifting" part of the identifying information for a value either to the left (into the row key, in option one) or to the right (into the column qualifiers in option 2). Under the covers, every key/value still stores the whole row key, and column family name. (If this is a bit confusing, take an hour and watch Lars George's excellent video about understanding HBase schema design: http://www.youtube.com/watch?v=_HLoH_PgrLk).

A manually paginated version has lots more complexities, as you note, like having to keep track of how many things are in each page, re-shuffling if new values are inserted, etc. That seems significantly more complex. It might have some slight speed advantages (or disadvantages!) at extremely high throughput, and the only way to really know that would

be to try it out. If you don't have time to build it both ways and compare, my advice would be to start with the simplest option (one row per user+value). Start simple and iterate! :)

Operational and Performance Configuration Options

Tune HBase Server RPC Handling

- Set `hbase.regionserver.handler.count` (in `hbase-site.xml`) to cores x spindles for concurrency.
- Optionally, split the call queues into separate read and write queues for differentiated service. The parameter `hbase.ipc.server.callqueue.handler.factor` specifies the number of call queues:
  - `0` means a single shared queue
  - `1` means one queue for each handler.
  - A value between `0` and `1` allocates the number of queues proportionally to the number of handlers. For instance, a value of `.5` shares one queue between each two handlers.
- Use `hbase.ipc.server.callqueue.read.ratio` (`hbase.ipc.server.callqueue.read.share` in 0.98) to split the call queues into read and write queues:
  - `0.5` means there will be the same number of read and write queues
  - `< 0.5` for more write than read
  - `> 0.5` for more read than write
- Set `hbase.ipc.server.callqueue.scan.ratio` (HBase 1.0+) to split read call queues into small-read and long-read queues:
  - 0.5 means that there will be the same number of short-read and long-read queues
  - `< 0.5` for more short-read
  - `> 0.5` for more long-read

Disable Nagle for RPC

Disable Nagle's algorithm. Delayed ACKs can add up to ~200ms to RPC round trip time. Set the following parameters:

- In Hadoop's `core-site.xml`:
  - `ipc.server.tcpnodelay = true`
  - `ipc.client.tcpnodelay = true`
- In HBase's `hbase-site.xml`:
  - `hbase.ipc.client.tcpnodelay = true`
  - `hbase.ipc.server.tcpnodelay = true`

Limit Server Failure Impact

Detect regionserver failure as fast as reasonable. Set the following parameters:

- In `hbase-site.xml`, set `zookeeper.session.timeout` to 30 seconds or less to bound failure detection (20-30 seconds is a good start).
  - Note: Zookeeper clients negotiate a session timeout with the server during client init. Server enforces this timeout to be in the range [`minSessionTimeout`, `maxSessionTimeout`] and both these timeouts (measured in milliseconds) are configurable in Zookeeper service configuration. If not configured, these default to 2 * `tickTime` and 20 * `tickTime` respectively (`tickTime` is the basic time unit used by ZooKeeper, as measured in milliseconds. It is used to regulate heartbeats, timeouts etc.). Refer to Zookeeper documentation for additional details.
- Detect and avoid unhealthy or failed HDFS DataNodes: in `hdfs-site.xml` and `hbase-site.xml`, set the following parameters:
  - `dfs.namenode.avoid.read.stale.datanode = true`
  - `dfs.namenode.avoid.write.stale.datanode = true`

Optimize on the Server Side for Low Latency

Skip the network for local blocks when the RegionServer goes to read from HDFS by exploiting HDFS's Short-Circuit Local Reads facility. Note how setup must be done both at the datanode and on the dfsclient ends of the conneciton — i.e. at the RegionServer and how both ends need to have loaded the hadoop native `.so` library. After configuring your hadoop setting *dfs.client.read.shortcircuit* to *true* and configuring the *dfs.domain.socket.path* path for the datanode and dfsclient to share and restarting, next configure the regionserver/dfsclient side.

- In `hbase-site.xml`, set the following parameters:
  - `dfs.client.read.shortcircuit = true`
  - `dfs.client.read.shortcircuit.skip.checksum = true` so we don't double checksum (HBase does its own checksumming to save on i/os).
  - `dfs.domain.socket.path` to match what was set for the datanodes.
  - `dfs.client.read.shortcircuit.buffer.size = 131072` Important to avoid OOME — hbase has a default it uses if unset, see `hbase.dfs.client.read.shortcircuit.buffer.size`; its default is 131072.
- Ensure data locality. In `hbase-site.xml`, set `hbase.hstore.min.locality.to.skip.major.compact = 0.7` (Meaning that 0.7 <= n <= 1)
- Make sure DataNodes have enough handlers for block transfers. In `hdfs-site.xml`, set the following parameters:
  - `dfs.datanode.max.xcievers >= 8192`
  - `dfs.datanode.handler.count =` number of spindles

Check the RegionServer logs after restart. You should only see complaint if misconfiguration. Otherwise, shortcircuit read operates quietly in background. It does not provide metrics so no optics on how effective it is but read latencies should show a marked improvement, especially if good data locality, lots of random reads, and dataset is larger than available cache.

Other advanced configurations that you might play with, especially if shortcircuit functionality is complaining in the logs, include `dfs.client.read.shortcircuit.streams.cach`

`e.size` and `dfs.client.socketcache.capacity`. Documentation is sparse on these options. You'll have to read source code.

RegionServer metric system exposes HDFS short circuit read metrics `shortCircuitBytesRead`. Other HDFS read metrics, including `totalBytesRead` (The total number of bytes read from HDFS), `localBytesRead` (The number of bytes read from the local HDFS DataNode), `zeroCopyBytesRead` (The number of bytes read through HDFS zero copy) are available and can be used to troubleshoot short-circuit read issues.

For more on short-circuit reads, see Colin's old blog on rollout, **How Improved Short-Circuit Local Reads Bring Better Performance and Security to Hadoop**. The **HDFS-347** issue also makes for an interesting read showing the HDFS community at its best (caveat a few comments).

JVM Tuning

Tune JVM GC for low collection latencies

- Use the CMS collector: `-XX:+UseConcMarkSweepGC`
- Keep eden space as small as possible to minimize average collection time. Example:

```
-XX:CMSInitiatingOccupancyFraction=70
```

- Optimize for low collection latency rather than throughput: `-Xmn512m`
- Collect eden in parallel: `-XX:+UseParNewGC`
- Avoid collection under pressure: `-XX:+UseCMSInitiatingOccupancyOnly`
- Limit per request scanner result sizing so everything fits into survivor space but doesn't tenure. In `hbase-site.xml`, set `hbase.client.scanner.max.result.size` to 1/8th of eden space (with -`Xmn512m` this is ~51MB)
- Set `max.result.size` x `handler.count` less than survivor space

OS-Level Tuning

- Turn transparent huge pages (THP) off:

```
echo never > /sys/kernel/mm/transparent_hugepage/enabled
echo never > /sys/kernel/mm/transparent_hugepage/defrag
```

- Set `vm.swappiness = 0`
- Set `vm.min_free_kbytes` to at least 1GB (8GB on larger memory systems)
- Disable NUMA zone reclaim with `vm.zone_reclaim_mode = 0`

Special Cases

For applications where failing quickly is better than waiting

- In `hbase-site.xml` on the client side, set the following parameters:
  - Set `hbase.client.pause = 1000`
  - Set `hbase.client.retries.number = 3`
  - If you want to ride over splits and region moves, increase `hbase.client.retries.number` substantially (>= 20)
  - Set the RecoverableZookeeper retry count: `zookeeper.recovery.retry = 1` (no retry)
- In `hbase-site.xml` on the server side, set the Zookeeper session timeout for detecting server failures: `zookeeper.session.timeout` ⇐ 30 seconds (20-30 is good).

For applications that can tolerate slightly out of date information

HBase timeline consistency (HBASE-10070) With read replicas enabled, read-only copies of regions (replicas) are distributed over the cluster. One RegionServer services the default or primary replica, which is the only replica that can service writes. Other RegionServers serve the secondary replicas, follow the primary RegionServer, and only see committed updates. The secondary replicas are read-only, but can serve reads immediately while the primary is failing over, cutting read availability blips from seconds to milliseconds. Phoenix supports timeline consistency as of 4.4.0 Tips:

- Deploy HBase 1.0.0 or later.
- Enable timeline consistent replicas on the server side.
- Use one of the following methods to set timeline consistency:
  - Use `ALTER SESSION SET CONSISTENCY = 'TIMELINE'`
  - Set the connection property `Consistency` to `timeline` in the JDBC connect string

More Information

See the Performance section perf.schema for more information about operational and performance schema design options, such as Bloom Filters, Table-configured regionsizes, compression, and blocksizes.

HBase and MapReduce

Apache MapReduce is a software framework used to analyze large amounts of data. It is provided by Apache Hadoop. MapReduce itself is out of the scope of this document. A good place to get started with MapReduce is https://hadoop.apache.org/docs/r2.6.0/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html. MapReduce version 2 (MR2)is now part of YARN.

This chapter discusses specific configuration steps you need to take to use MapReduce on data within HBase. In addition, it discusses other interactions and issues between HBase and MapReduce jobs.

> **mapred and mapreduce**
>
> There are two mapreduce packages in HBase as in MapReduce itself: *org.apache.hadoop.hbase.mapred* and *org.apache.hadoop.hbase.mapreduce*. The former does old-style API and the latter the new mode. The latter has more facility though you can usually find an equivalent in the older package. Pick the package that goes with your MapReduce deploy. When in doubt or starting over, pick *org.apache.hadoop.hbase.mapreduce*. In the notes below, we refer to *o.a.h.h.mapreduce* but replace with *o.a.h.h.mapred* if that is what you are using.

HBase, MapReduce, and the CLASSPATH

By default, MapReduce jobs deployed to a MapReduce cluster do not have access to either the HBase configuration under `$HBASE_CONF_DIR` or the HBase classes.

To give the MapReduce jobs the access they need, you could add _hbase-site.xml_to _$HADOOP*HOME/conf* and add HBase jars to the *$HADOOP_HOME/lib* directory. You would then need to copy these changes across your cluster. Or you could edit *$HADOOP_HOME/conf/hadoop-env.sh* and add hbase dependencies to the `HADOOP_CLASSPATH` variable. Neither of these approaches is recommended because it will pollute your Hadoop install with HBase references. It also requires you restart the Hadoop cluster before Hadoop can use the HBase data.

The recommended approach is to let HBase add its dependency jars and use `HADOOP_CLASSPATH` or `-libjars`.

Since HBase `0.90.x`, HBase adds its dependency JARs to the job configuration itself. The dependencies only need to be available on the local `CLASSPATH` and from here they'll be picked up and bundled into the fat job jar deployed to the MapReduce cluster. A basic trick just passes the full hbase classpath — all hbase and dependent jars as well as configurations — to the mapreduce job runner letting hbase utility pick out from the full-on classpath what it needs adding them to the MapReduce job configuration (See the source at `TableMapReduceUtil#addDependencyJars(org.apache.hadoop.mapreduce.Job)` for how this is done).

The following example runs the bundled HBase RowCounter MapReduce job against a table named `usertable`. It sets into `HADOOP_CLASSPATH` the jars hbase needs to run in an MapReduce context (including configuration files such as hbase-site.xml). Be sure to use the correct version of the HBase JAR for your system; replace the VERSION string in the below command line w/ the version of your local hbase install. The backticks (```) cause the shell to execute the sub-commands, setting the output of `hbase classpath` into `HADOOP_CLASSPATH`. This example assumes you use a BASH-compatible shell.

```
$ HADOOP_CLASSPATH=`${HBASE_HOME}/bin/hbase classpath` \
  ${HADOOP_HOME}/bin/hadoop jar ${HBASE_HOME}/lib/hbase-mapreduce-VERSION.jar \
  org.apache.hadoop.hbase.mapreduce.RowCounter usertable
```

The above command will launch a row counting mapreduce job against the hbase cluster that is pointed to by your local configuration on a cluster that the hadoop configs are pointing to.

The main for the `hbase-mapreduce.jar` is a Driver that lists a few basic mapreduce tasks that ship with hbase. For example, presuming your install is hbase `2.0.0-SNAPSHOT`:

```
$ HADOOP_CLASSPATH=`${HBASE_HOME}/bin/hbase classpath` \
  ${HADOOP_HOME}/bin/hadoop jar ${HBASE_HOME}/lib/hbase-mapreduce-2.0.0-SNAPSHOT.
jar
An example program must be given as the first argument.
Valid program names are:
  CellCounter: Count cells in HBase table.
  WALPlayer: Replay WAL files.
  completebulkload: Complete a bulk data load.
  copytable: Export a table from local cluster to peer cluster.
  export: Write table data to HDFS.
  exportsnapshot: Export the specific snapshot to a given FileSystem.
  import: Import data written by Export.
  importtsv: Import data in TSV format.
  rowcounter: Count rows in HBase table.
```

```
  verifyrep: Compare the data from tables in two different clusters. WARNING: It
doesn't work for incrementColumnValues'd cells since the timestamp is changed aft
er being appended to the log.
```

You can use the above listed shortnames for mapreduce jobs as in the below re-run of the row counter job (again, presuming your install is hbase `2.0.0-SNAPSHOT`):

```
$ HADOOP_CLASSPATH=`${HBASE_HOME}/bin/hbase classpath` \
  ${HADOOP_HOME}/bin/hadoop jar ${HBASE_HOME}/lib/hbase-mapreduce-2.0.0-SNAPSHOT.
jar \
  rowcounter usertable
```

You might find the more selective `hbase mapredcp` tool output of interest; it lists the minimum set of jars needed to run a basic mapreduce job against an hbase install. It does not include configuration. You'll probably need to add these if you want your MapReduce job to find the target cluster. You'll probably have to also add pointers to extra jars once you start to do anything of substance. Just specify the extras by passing the system propery `-Dtmpjars` when you run `hbase mapredcp`.

For jobs that do not package their dependencies or call `TableMapReduceUtil#addDependencyJars`, the following command structure is necessary:

```
$ HADOOP_CLASSPATH=`${HBASE_HOME}/bin/hbase mapredcp`:${HBASE_HOME}/conf hadoop j
ar MyApp.jar MyJobMainClass -libjars $(${HBASE_HOME}/bin/hbase mapredcp | tr ':'
',') ...
```

The example may not work if you are running HBase from its build directory rather than an installed location. You may see an error like the following:

```
java.lang.RuntimeException: java.lang.ClassNotFoundException: org.apache.h
adoop.hbase.mapreduce.RowCounter$RowCounterMapper
```

If this occurs, try modifying the command as follows, so that it uses the HBase JARs from the *target/* directory within the build environment.

```
$ HADOOP_CLASSPATH=${HBASE_BUILD_HOME}/hbase-mapreduce/target/hbase-mapred
uce-VERSION-SNAPSHOT.jar:`${HBASE_BUILD_HOME}/bin/hbase classpath` ${HADOO
P_HOME}/bin/hadoop jar ${HBASE_BUILD_HOME}/hbase-mapreduce/target/hbase-ma
preduce-VERSION-SNAPSHOT.jar rowcounter usertable
```

⚠ **Notice to MapReduce users of HBase between 0.96.1 and 0.98.4**

Some MapReduce jobs that use HBase fail to launch. The symptom is an exception similar to the following:

```
Exception in thread "main" java.lang.IllegalAccessError: class
    com.google.protobuf.ZeroCopyLiteralByteString cannot access its superc
lass
    com.google.protobuf.LiteralByteString
    at java.lang.ClassLoader.defineClass1(Native Method)
    at java.lang.ClassLoader.defineClass(ClassLoader.java:792)
    at java.security.SecureClassLoader.defineClass(SecureClassLoader.java:
142)
    at java.net.URLClassLoader.defineClass(URLClassLoader.java:449)
    at java.net.URLClassLoader.access$100(URLClassLoader.java:71)
    at java.net.URLClassLoader$1.run(URLClassLoader.java:361)
    at java.net.URLClassLoader$1.run(URLClassLoader.java:355)
    at java.security.AccessController.doPrivileged(Native Method)
    at java.net.URLClassLoader.findClass(URLClassLoader.java:354)
    at java.lang.ClassLoader.loadClass(ClassLoader.java:424)
    at java.lang.ClassLoader.loadClass(ClassLoader.java:357)
    at
    org.apache.hadoop.hbase.protobuf.ProtobufUtil.toScan(ProtobufUtil.jav
a:818)
    at
    org.apache.hadoop.hbase.mapreduce.TableMapReduceUtil.convertScanToStri
ng(TableMapReduceUtil.java:433)
    at
    org.apache.hadoop.hbase.mapreduce.TableMapReduceUtil.initTableMapperJo
b(TableMapReduceUtil.java:186)
    at
    org.apache.hadoop.hbase.mapreduce.TableMapReduceUtil.initTableMapperJo
b(TableMapReduceUtil.java:147)
    at
    org.apache.hadoop.hbase.mapreduce.TableMapReduceUtil.initTableMapperJo
b(TableMapReduceUtil.java:270)
    at
    org.apache.hadoop.hbase.mapreduce.TableMapReduceUtil.initTableMapperJo
b(TableMapReduceUtil.java:100)
...
```

This is caused by an optimization introduced in HBASE-9867 that inadvertently introduced a classloader dependency.

This affects both jobs using the `-libjars` option and "fat jar," those which package their runtime dependencies in a nested `lib` folder.

In order to satisfy the new classloader requirements, `hbase-protocol.jar` must be included in Hadoop's classpath. See HBase, MapReduce, and the CLASSPATH for current recommendations for resolving classpath errors. The following is included for historical purposes.

This can be resolved system-wide by including a reference to the `hbase-protocol.jar` in Hadoop's lib directory, via a symlink or by copying the jar into the new location.

This can also be achieved on a per-job launch basis by including it in the `HADOOP_CLASSPATH` environment variable at job submission time. When launching jobs that package their dependencies, all three of the following job launching commands satisfy this requirement:

```
$ HADOOP_CLASSPATH=/path/to/hbase-protocol.jar:/path/to/hbase/conf hadoop
jar MyJob.jar MyJobMainClass
$ HADOOP_CLASSPATH=$(hbase mapredcp):/path/to/hbase/conf hadoop jar MyJob.
jar MyJobMainClass
$ HADOOP_CLASSPATH=$(hbase classpath) hadoop jar MyJob.jar MyJobMainClass
```

For jars that do not package their dependencies, the following command structure is necessary:

```
$ HADOOP_CLASSPATH=$(hbase mapredcp):/etc/hbase/conf hadoop jar MyApp.jar
MyJobMainClass -libjars $(hbase mapredcp | tr ':' ',') ...
```

See also HBASE-10304 for further discussion of this issue.

MapReduce Scan Caching

TableMapReduceUtil now restores the option to set scanner caching (the number of rows which are cached before returning the result to the client) on the Scan object that is passed in. This functionality was lost due to a bug in HBase 0.95 (HBASE-11558), which is fixed for HBase 0.98.5 and 0.96.3. The priority order for choosing the scanner caching is as follows:

1. Caching settings which are set on the scan object.
2. Caching settings which are specified via the configuration option `hbase.client.scanner.caching`, which can either be set manually in *hbase-site.xml* or via the helper method `TableMapReduceUtil.setScannerCaching()`.
3. The default value `HConstants.DEFAULT_HBASE_CLIENT_SCANNER_CACHING`, which is set to `100`.

Optimizing the caching settings is a balance between the time the client waits for a result and the number of sets of results the client needs to receive. If the caching setting is too large, the client could end up waiting for a long time or the request could even time out. If the setting is too small, the scan needs to return results in several pieces. If you think of the scan as a shovel, a bigger cache setting is analogous to a bigger shovel, and a smaller cache setting is equivalent to more shoveling in order to fill the bucket.

The list of priorities mentioned above allows you to set a reasonable default, and override it for specific operations.

See the API documentation for Scan for more details.

Bundled HBase MapReduce Jobs

The HBase JAR also serves as a Driver for some bundled MapReduce jobs. To learn about the bundled MapReduce jobs, run the following command.

```
$ ${HADOOP_HOME}/bin/hadoop jar ${HBASE_HOME}/hbase-mapreduce-VERSION.jar
An example program must be given as the first argument.
Valid program names are:
  copytable: Export a table from local cluster to peer cluster
  completebulkload: Complete a bulk data load.
  export: Write table data to HDFS.
  import: Import data written by Export.
  importtsv: Import data in TSV format.
  rowcounter: Count rows in HBase table
```

Each of the valid program names are bundled MapReduce jobs. To run one of the jobs, model your command after the following example.

```
$ ${HADOOP_HOME}/bin/hadoop jar ${HBASE_HOME}/hbase-mapreduce-VERSION.jar rowcoun
ter myTable
```

HBase as a MapReduce Job Data Source and Data Sink

HBase can be used as a data source, TableInputFormat, and data sink, TableOutputFormat or MultiTableOutputFormat, for MapReduce jobs. Writing MapReduce jobs that read or write HBase, it is advisable to subclass TableMapper and/or TableReducer. See the do-nothing pass-through classes IdentityTableMapper and IdentityTableReducer for basic usage. For a more involved example, see RowCounter or review the `org.apache.hadoop.hbase.mapreduce.TestTableMapReduce` unit test.

If you run MapReduce jobs that use HBase as source or sink, need to specify source and sink table and column names in your configuration.

When you read from HBase, the `TableInputFormat` requests the list of regions from HBase and makes a map, which is either a `map-per-region` or `mapreduce.job.maps` map, whichever is smaller. If your job only has two maps, raise `mapreduce.job.maps` to a number greater than the number of regions. Maps will run on the adjacent TaskTracker/NodeManager if you are running a TaskTracer/NodeManager and RegionServer per node. When writing to HBase, it may make sense to avoid the Reduce step and write back into HBase from within your map. This approach works when your job does not need the sort and collation that MapReduce does on the map-emitted data. On insert, HBase 'sorts' so there is no point double-sorting (and shuffling data around your MapReduce cluster) unless you need to. If you do not need the Reduce, your map might emit counts of records processed for reporting at the end of the job, or set the number of Reduces to zero and use TableOutputFormat. If running the Reduce step makes sense in your case, you should typically use multiple reducers so that load is spread across the HBase cluster.

A new HBase partitioner, the HRegionPartitioner, can run as many reducers the number of existing regions. The HRegionPartitioner is suitable when your table is large and your upload will not greatly alter the number of existing regions upon completion. Otherwise use the default partitioner.

Writing HFiles Directly During Bulk Import

If you are importing into a new table, you can bypass the HBase API and write your content directly to the filesystem, formatted into HBase data files (HFiles). Your import will run faster, perhaps an order of magnitude faster. For more on how this mechanism works, see Bulk Load.

RowCounter Example

The included RowCounter MapReduce job uses `TableInputFormat` and does a count of all rows in the specified table. To run it, use the following command:

```
$ ./bin/hadoop jar hbase-X.X.X.jar
```

This will invoke the HBase MapReduce Driver class. Select `rowcounter` from the choice of jobs offered. This will print rowcounter usage advice to standard output. Specify the

tablename, column to count, and output directory. If you have classpath errors, see HBase, MapReduce, and the CLASSPATH.

Map-Task Splitting

The Default HBase MapReduce Splitter

When TableInputFormat is used to source an HBase table in a MapReduce job, its splitter will make a map task for each region of the table. Thus, if there are 100 regions in the table, there will be 100 map-tasks for the job - regardless of how many column families are selected in the Scan.

Custom Splitters

For those interested in implementing custom splitters, see the method `getSplits` in TableInputFormatBase. That is where the logic for map-task assignment resides.

HBase MapReduce Examples

HBase MapReduce Read Example

The following is an example of using HBase as a MapReduce source in read-only manner. Specifically, there is a Mapper instance but no Reducer, and nothing is being emitted from the Mapper. The job would be defined as follows...

```
Configuration config = HBaseConfiguration.create();
Job job = new Job(config, "ExampleRead");
job.setJarByClass(MyReadJob.class);     // class that contains mapper

Scan scan = new Scan();
scan.setCaching(500);        // 1 is the default in Scan, which will be bad for M
apReduce jobs
scan.setCacheBlocks(false);  // don't set to true for MR jobs
// set other scan attrs
...

TableMapReduceUtil.initTableMapperJob(
  tableName,        // input HBase table name
  scan,             // Scan instance to control CF and attribute selection
  MyMapper.class,   // mapper
  null,             // mapper output key
  null,             // mapper output value
```

```
  job);
job.setOutputFormatClass(NullOutputFormat.class);   // because we aren't emitting
anything from mapper

boolean b = job.waitForCompletion(true);
if (!b) {
  throw new IOException("error with job!");
}
```

...and the mapper instance would extend TableMapper...

```
public static class MyMapper extends TableMapper<Text, Text> {

  public void map(ImmutableBytesWritable row, Result value, Context context) thro
ws InterruptedException, IOException {
    // process data for the row from the Result instance.
   }
}
```

HBase MapReduce Read/Write Example

The following is an example of using HBase both as a source and as a sink with MapReduce. This example will simply copy data from one table to another.

```
Configuration config = HBaseConfiguration.create();
Job job = new Job(config,"ExampleReadWrite");
job.setJarByClass(MyReadWriteJob.class);    // class that contains mapper

Scan scan = new Scan();
scan.setCaching(500);        // 1 is the default in Scan, which will be bad for M
apReduce jobs
scan.setCacheBlocks(false);  // don't set to true for MR jobs
// set other scan attrs

TableMapReduceUtil.initTableMapperJob(
  sourceTable,      // input table
  scan,             // Scan instance to control CF and attribute selection
  MyMapper.class,   // mapper class
  null,             // mapper output key
  null,             // mapper output value
  job);
TableMapReduceUtil.initTableReducerJob(
  targetTable,      // output table
  null,             // reducer class
  job);
job.setNumReduceTasks(0);

boolean b = job.waitForCompletion(true);
if (!b) {
  throw new IOException("error with job!");
```

```
}
```

An explanation is required of what `TableMapReduceUtil` is doing, especially with the reducer. TableOutputFormat is being used as the outputFormat class, and several parameters are being set on the config (e.g., `TableOutputFormat.OUTPUT_TABLE`), as well as setting the reducer output key to `ImmutableBytesWritable` and reducer value to `Writable`. These could be set by the programmer on the job and conf, but `TableMapReduceUtil` tries to make things easier.

The following is the example mapper, which will create a `Put` and matching the input `Result` and emit it. Note: this is what the CopyTable utility does.

```
public static class MyMapper extends TableMapper<ImmutableBytesWritable, Put>  {

  public void map(ImmutableBytesWritable row, Result value, Context context) throws IOException, InterruptedException {
    // this example is just copying the data from the source table...
      context.write(row, resultToPut(row,value));
    }

    private static Put resultToPut(ImmutableBytesWritable key, Result result) throws IOException {
      Put put = new Put(key.get());
      for (Cell cell : result.listCells()) {
        put.add(cell);
      }
      return put;
    }
}
```

There isn't actually a reducer step, so `TableOutputFormat` takes care of sending the `Put` to the target table.

This is just an example, developers could choose not to use `TableOutputFormat` and connect to the target table themselves.

HBase MapReduce Read/Write Example With Multi-Table Output

TODO: example for `MultiTableOutputFormat`.

HBase MapReduce Summary to HBase Example

The following example uses HBase as a MapReduce source and sink with a summarization step. This example will count the number of distinct instances of a value in a table and write those summarized counts in another table.

```
Configuration config = HBaseConfiguration.create();
Job job = new Job(config,"ExampleSummary");
job.setJarByClass(MySummaryJob.class);     // class that contains mapper and redu
cer

Scan scan = new Scan();
scan.setCaching(500);        // 1 is the default in Scan, which will be bad for M
apReduce jobs
scan.setCacheBlocks(false);  // don't set to true for MR jobs
// set other scan attrs

TableMapReduceUtil.initTableMapperJob(
  sourceTable,        // input table
  scan,               // Scan instance to control CF and attribute selection
  MyMapper.class,     // mapper class
  Text.class,         // mapper output key
  IntWritable.class,  // mapper output value
  job);
TableMapReduceUtil.initTableReducerJob(
  targetTable,        // output table
  MyTableReducer.class,    // reducer class
  job);
job.setNumReduceTasks(1);   // at least one, adjust as required

boolean b = job.waitForCompletion(true);
if (!b) {
  throw new IOException("error with job!");
}
```

In this example mapper a column with a String-value is chosen as the value to summarize upon. This value is used as the key to emit from the mapper, and an `IntWritable` represents an instance counter.

```
public static class MyMapper extends TableMapper<Text, IntWritable>  {
  public static final byte[] CF = "cf".getBytes();
  public static final byte[] ATTR1 = "attr1".getBytes();

  private final IntWritable ONE = new IntWritable(1);
  private Text text = new Text();

  public void map(ImmutableBytesWritable row, Result value, Context context) thro
ws IOException, InterruptedException {
    String val = new String(value.getValue(CF, ATTR1));
    text.set(val);     // we can only emit Writables...
```

```
    context.write(text, ONE);
  }
}
```

In the reducer, the "ones" are counted (just like any other MR example that does this), and then emits a `Put`.

```
public static class MyTableReducer extends TableReducer<Text, IntWritable, Immuta
bleBytesWritable>  {
  public static final byte[] CF = "cf".getBytes();
  public static final byte[] COUNT = "count".getBytes();

  public void reduce(Text key, Iterable<IntWritable> values, Context context) thr
ows IOException, InterruptedException {
    int i = 0;
    for (IntWritable val : values) {
      i += val.get();
    }
    Put put = new Put(Bytes.toBytes(key.toString()));
    put.add(CF, COUNT, Bytes.toBytes(i));

    context.write(null, put);
  }
}
```

HBase MapReduce Summary to File Example

This very similar to the summary example above, with exception that this is using HBase as a MapReduce source but HDFS as the sink. The differences are in the job setup and in the reducer. The mapper remains the same.

```
Configuration config = HBaseConfiguration.create();
Job job = new Job(config,"ExampleSummaryToFile");
job.setJarByClass(MySummaryFileJob.class);     // class that contains mapper and
reducer

Scan scan = new Scan();
scan.setCaching(500);        // 1 is the default in Scan, which will be bad for M
apReduce jobs
scan.setCacheBlocks(false);  // don't set to true for MR jobs
// set other scan attrs

TableMapReduceUtil.initTableMapperJob(
  sourceTable,        // input table
  scan,               // Scan instance to control CF and attribute selection
  MyMapper.class,     // mapper class
  Text.class,         // mapper output key
  IntWritable.class,  // mapper output value
  job);
```

```
job.setReducerClass(MyReducer.class);    // reducer class
job.setNumReduceTasks(1);    // at least one, adjust as required
FileOutputFormat.setOutputPath(job, new Path("/tmp/mr/mySummaryFile"));  // adjus
t directories as required

boolean b = job.waitForCompletion(true);
if (!b) {
  throw new IOException("error with job!");
}
```

As stated above, the previous Mapper can run unchanged with this example. As for the Reducer, it is a "generic" Reducer instead of extending TableMapper and emitting Puts.

```
public static class MyReducer extends Reducer<Text, IntWritable, Text, IntWritabl
e>  {

  public void reduce(Text key, Iterable<IntWritable> values, Context context) thr
ows IOException, InterruptedException {
    int i = 0;
    for (IntWritable val : values) {
      i += val.get();
    }
    context.write(key, new IntWritable(i));
  }
}
```

HBase MapReduce Summary to HBase Without Reducer

It is also possible to perform summaries without a reducer - if you use HBase as the reducer.

An HBase target table would need to exist for the job summary. The Table method `incremeⁿtColumnValue` would be used to atomically increment values. From a performance perspective, it might make sense to keep a Map of values with their values to be incremented for each map-task, and make one update per key at during the `cleanup` method of the mapper. However, your mileage may vary depending on the number of rows to be processed and unique keys.

In the end, the summary results are in HBase.

HBase MapReduce Summary to RDBMS

Sometimes it is more appropriate to generate summaries to an RDBMS. For these cases, it is possible to generate summaries directly to an RDBMS via a custom reducer. The `setup`

method can connect to an RDBMS (the connection information can be passed via custom parameters in the context) and the cleanup method can close the connection.

It is critical to understand that number of reducers for the job affects the summarization implementation, and you'll have to design this into your reducer. Specifically, whether it is designed to run as a singleton (one reducer) or multiple reducers. Neither is right or wrong, it depends on your use-case. Recognize that the more reducers that are assigned to the job, the more simultaneous connections to the RDBMS will be created - this will scale, but only to a point.

```
public static class MyRdbmsReducer extends Reducer<Text, IntWritable, Text, IntWr
itable>  {

  private Connection c = null;

  public void setup(Context context) {
    // create DB connection...
  }

  public void reduce(Text key, Iterable<IntWritable> values, Context context) thr
ows IOException, InterruptedException {
    // do summarization
    // in this example the keys are Text, but this is just an example
  }

  public void cleanup(Context context) {
    // close db connection
  }

}
```

In the end, the summary results are written to your RDBMS table/s.

Accessing Other HBase Tables in a MapReduce Job

Although the framework currently allows one HBase table as input to a MapReduce job, other HBase tables can be accessed as lookup tables, etc., in a MapReduce job via creating an Table instance in the setup method of the Mapper.

```
public class MyMapper extends TableMapper<Text, LongWritable> {
  private Table myOtherTable;

  public void setup(Context context) {
    // In here create a Connection to the cluster and save it or use the Connecti
on
```

```
    // from the existing table
    myOtherTable = connection.getTable("myOtherTable");
  }

  public void map(ImmutableBytesWritable row, Result value, Context context) thro
ws IOException, InterruptedException {
    // process Result...
    // use 'myOtherTable' for lookups
  }
```

Speculative Execution

It is generally advisable to turn off speculative execution for MapReduce jobs that use HBase as a source. This can either be done on a per-Job basis through properties, or on the entire cluster. Especially for longer running jobs, speculative execution will create duplicate map-tasks which will double-write your data to HBase; this is probably not what you want.

See Speculative Execution for more information.

Security

Reporting Security Bugs

HBase adheres to the Apache Software Foundation's policy on reported vulnerabilities, available at http://apache.org/security/.

If you wish to send an encrypted report, you can use the GPG details provided for the general ASF security list. This will likely increase the response time to your report.

To protect existing HBase installations from exploitation, please **do not** use JIRA to report security-related bugs. Instead, send your report to the mailing list `private@hbase.apache.org`, which allows anyone to send messages, but restricts who can read them. Someone on that list will contact you to follow up on your report.

Web UI Security

Using Secure HTTP (HTTPS) for the Web UI

A default HBase install uses insecure HTTP connections for Web UIs for the master and region servers. To enable secure HTTP (HTTPS) connections instead, set `hbase.ssl.enabled` to `true` in *hbase-site.xml*(Please prepare SSL certificate and ssl configuration file in advance). This does not change the port used by the Web UI. To change the port for the web UI for a given HBase component, configure that port's setting in hbase-site.xml. These settings are:

- `hbase.master.info.port`
- `hbase.regionserver.info.port`

If you enable secure HTTP, clients should connect to HBase using the `https://` URL. Clients using the `http://` URL will receive an HTTP response of `200`, but will not receive any data. The following exception is logged:

```
javax.net.ssl.SSLException: Unrecognized SSL message, plaintext connectio
n?
```

This is because the same port is used for HTTP and HTTPS.

HBase uses Jetty for the Web UI. Without modifying Jetty itself, it does not seem possible to configure Jetty to redirect one port to another on the same host. See Nick Dimiduk's

> contribution on this Stack Overflow thread for more information. If you know how to fix this without opening a second port for HTTPS, patches are appreciated.

Disable cache in HBase UI

Set the following configuration in hbase-site to set max age to zero and disable cache for the web UI:

```
<property>
  <name>hbase.http.filter.no-store.enable</name>
  <value>true</value>
</property>
```

Using SPNEGO for Kerberos authentication with Web UIs

Kerberos-authentication to HBase Web UIs can be enabled via configuring SPNEGO with the `hbase.security.authentication.ui` property in *hbase-site.xml*. Enabling this authentication requires that HBase is also configured to use Kerberos authentication for RPCs (e.g `hbase.security.authentication` = `kerberos`).

```
<property>
  <name>hbase.security.authentication.ui</name>
  <value>kerberos</value>
  <description>Controls what kind of authentication should be used for the HBase
web UIs.</description>
</property>
<property>
  <name>hbase.security.authentication</name>
  <value>kerberos</value>
  <description>The Kerberos keytab file to use for SPNEGO authentication by the w
eb server.</description>
</property>
```

A number of properties exist to configure SPNEGO authentication for the web server:

```
<property>
  <name>hbase.security.authentication.spnego.kerberos.principal</name>
  <value>HTTP/_HOST@EXAMPLE.COM</value>
  <description>Required for SPNEGO, the Kerberos principal to use for SPNEGO auth
entication by the
  web server. The _HOST keyword will be automatically substituted with the node's
  hostname.</description>
```

```
</property>
<property>
  <name>hbase.security.authentication.spnego.kerberos.keytab</name>
  <value>/etc/security/keytabs/spnego.service.keytab</value>
  <description>Required for SPNEGO, the Kerberos keytab file to use for SPNEGO au
thentication by the
  web server.</description>
</property>
<property>
  <name>hbase.security.authentication.spnego.kerberos.name.rules</name>
  <value></value>
  <description>Optional, Hadoop-style `auth_to_local` rules which will be parsed
and used in the
  handling of Kerberos principals</description>
</property>
<property>
  <name>hbase.security.authentication.signature.secret.file</name>
  <value></value>
  <description>Optional, a file whose contents will be used as a secret to sign t
he HTTP cookies
  as a part of the SPNEGO authentication handshake. If this is not provided, Jav
a's `Random` library
  will be used for the secret.</description>
</property>
```

Defining administrators of the Web UI with SPNEGO

In the previous section, we cover how to enable authentication for the Web UI via SPNEGO. However, some portions of the Web UI could be used to impact the availability and performance of an HBase cluster. As such, it is desirable to ensure that only those with proper authority can interact with these sensitive endpoints.

HBase allows the adminstrators to be defined via a list of usernames or groups in hbase-site.xml

```
<property>
  <name>hbase.security.authentication.spnego.admin.users</name>
  <value></value>
</property>
<property>
  <name>hbase.security.authentication.spnego.admin.groups</name>
  <value></value>
</property>
```

The usernames are those which the Kerberos identity maps to, given the Hadoop `auth_to_local` rules in core-site.xml. The groups here are the Unix groups associated with the

mapped usernames.

Consider the following scenario to describe how the configuration properties operate. Consider three users which are defined in the Kerberos KDC:

- `alice@COMPANY.COM`
- `bob@COMPANY.COM`
- `charlie@COMPANY.COM`

The default Hadoop `auth_to_local` rules map these principals to the "shortname":

- `alice`
- `bob`
- `charlie`

Unix groups membership define that `alice` is a member of the group `admins`. `bob` and `charlie` are not members of the `admins` group.

```
<property>
  <name>hbase.security.authentication.spnego.admin.users</name>
  <value>charlie</value>
</property>
<property>
  <name>hbase.security.authentication.spnego.admin.groups</name>
  <value>admins</value>
</property>
```

Given the above configuration, `alice` is allowed to access sensitive endpoints in the Web UI as she is a member of the `admins` group. `charlie` is also allowed to access sensitive endpoints because he is explicitly listed as an admin in the configuration. `bob` is not allowed to access sensitive endpoints because he is not a member of the `admins` group nor is listed as an explicit admin user via `hbase.security.authentication.spnego.admin.users`, but can still use any non-sensitive endpoints in the Web UI.

If it doesn't go without saying: non-authenticated users cannot access any part of the Web UI.

Using LDAP authentication with Web UIs

LDAP authentication to HBase Web UIs can be enabled via configuring LDAP with the `hbase.security.authentication.ui` property in *hbase-site.xml*. The `hbase.http.filter.initializers` property also needs to have the `AuthenticationFilterInitializer` class.

IMPORTANT: A LDAP server must be configured and running. When TLS is enabled for communication with LDAP server (either via ldaps scheme or 'start TLS' extension), configure the public certificate of the LDAP server in the local truststore. The LDAP authentication mechanism uses HTTP Basic authentication scheme to verify user specified credentials against a configured LDAP (or Active Directory) server. The authentication filter must be configured with the following init parameters:

```
<property>
  <name>hbase.security.authentication.ui</name>
  <value>ldap</value>
  <description>Controls what kind of authentication should be used for the HBase
web UIs.</description>
</property>
<property>
  <name>hbase.http.filter.initializers</name>
  <value>org.apache.hadoop.hbase.http.lib.AuthenticationFilterInitializer</value>
  <description>Comma separated class names corresponding to the Filters that will
be initialized.
  Then, the Filters will be applied to all user facing jsp and servlet web pages.
</description>
</property>
<property>
  <name>hadoop.http.authentication.type</name>
  <value>ldap</value>
  <description>Defines authentication used for the HTTP web-consoles in Hadoop ec
osystem.</description>
</property>
```

A number of properties exist to configure LDAP authentication for the web server:

```
<property>
  <name>hadoop.http.authentication.ldap.binddomain</name>
  <value>EXAMPLE.COM</value>
  <description>The LDAP bind domain value to be used with the LDAP server. This p
roperty is optional
   and useful only in case of Active Directory server (e.g. example.com).</descri
ption>
</property>
<property>
  <name>hadoop.http.authentication.ldap.providerurl</name>
  <value>ldap://ldap-server-host:8920</value>
```

```
  <description>The url of the LDAP server.</description>
</property>
<property>
  <name>hadoop.http.authentication.ldap.enablestarttls</name>
  <value>false</value>
  <description>A boolean value used to define if the LDAP server supports 'StartT
LS' extension.</description>
</property>
<property>
  <name>hadoop.http.authentication.ldap.basedn</name>
  <value>ou=users,dc=example,dc=com</value>
  <description>The base distinguished name (DN) to be used with the LDAP server.
This value is
  appended to the provided user id for authentication purpose. This property is n
ot useful in case
  of Active Directory server.</description>
</property>
```

Defining Administrators of the Web UI with LDAP

In the previous section, we discussed enabling authentication for the Web UI via LDAP. Certain portions of the Web UI can impact the availability and performance of an HBase cluster. To safeguard these sensitive endpoints, it is essential to restrict access to authorized administrators only.

HBase provides a mechanism to define administrators for the Web UI through a list of usernames in the `hbase-site.xml` configuration file.

To specify the administrators, use the following property in `hbase-site.xml`:

```
<property>
  <name>hbase.security.authentication.ldap.admin.users</name>
  <value>admin1,admin2,admin3</value>
</property>
```

The usernames listed in the above property should correspond to the LDAP usernames of the administrators.

Notes

- This feature is supported by only versions of HBase having HBASE-29244
- Ensure that the LDAP server is properly configured and running. See the previous section for details.

- Only users explicitly listed in the `hbase.security.authentication.ldap.admin.users` property will have access to sensitive endpoints.
- Non-administrative users can still access non-sensitive endpoints, provided they are authenticated.

By defining administrators in this way, you can ensure that only authorized personnel can interact with critical Web UI functionalities, thereby enhancing the security and stability of your HBase cluster.

Other UI security-related configuration

While it is a clear anti-pattern for HBase developers, the developers acknowledge that the HBase configuration (including Hadoop configuration files) may contain sensitive information. As such, a user may find that they do not want to expose the HBase service-level configuration to all authenticated users. They may configure HBase to require a user must be an admin to access the service-level configuration via the HBase UI. This configuration is **false** by default (any authenticated user may access the configuration).

Users who wish to change this would set the following in their hbase-site.xml:

```
<property>
  <name>hbase.security.authentication.ui.config.protected</name>
  <value>true</value>
</property>
```

To disable showing stack traces in HBase UI for hiding sensitive information, set the following in hbase-site:

```
<property>
  <name>hbase.ui.show-stack-traces</name>
  <value>false</value>
</property>
```

Secure Client Access to Apache HBase

Newer releases of Apache HBase (>= 0.92) support optional SASL authentication of clients. See also Matteo Bertozzi's article on Understanding User Authentication and

Authorization in Apache HBase.

This describes how to set up Apache HBase and clients for connection to secure HBase resources.

Prerequisites

Hadoop Authentication Configuration

To run HBase RPC with strong authentication, you must set `hbase.security.authentication` to `kerberos`. In this case, you must also set `hadoop.security.authentication` to `kerberos` in core-site.xml. Otherwise, you would be using strong authentication for HBase but not for the underlying HDFS, which would cancel out any benefit.

Kerberos KDC

You need to have a working Kerberos KDC.

Server-side Configuration for Secure Operation

First, refer to security.prerequisites and ensure that your underlying HDFS configuration is secure.

Add the following to the `hbase-site.xml` file on every server machine in the cluster:

```
<property>
  <name>hbase.security.authentication</name>
  <value>kerberos</value>
</property>
<property>
  <name>hbase.security.authorization</name>
  <value>true</value>
</property>
<property>
<name>hbase.coprocessor.region.classes</name>
  <value>org.apache.hadoop.hbase.security.token.TokenProvider</value>
</property>
```

A full shutdown and restart of HBase service is required when deploying these configuration changes.

Client-side Configuration for Secure Operation

First, refer to **Prerequisites** and ensure that your underlying HDFS configuration is secure.

Add the following to the `hbase-site.xml` file on every client:

```
<property>
  <name>hbase.security.authentication</name>
  <value>kerberos</value>
</property>
```

Before 2.2.0 version, the client environment must be logged in to Kerberos from KDC or keytab via the `kinit` command before communication with the HBase cluster will be possible.

Since 2.2.0, client can specify the following configurations in `hbase-site.xml`:

```
<property>
  <name>hbase.client.keytab.file</name>
  <value>/local/path/to/client/keytab</value>
</property>

<property>
  <name>hbase.client.keytab.principal</name>
  <value>foo@EXAMPLE.COM</value>
</property>
```

Then application can automatically do the login and credential renewal jobs without client interference.

It's optional feature, client, who upgrades to 2.2.0, can still keep their login and credential renewal logic already did in older version, as long as keeping `hbase.client.keytab.file` and `hbase.client.keytab.principal` are unset.

Be advised that if the `hbase.security.authentication` in the client- and server-side site files do not match, the client will not be able to communicate with the cluster.

Once HBase is configured for secure RPC it is possible to optionally configure encrypted communication. To do so, add the following to the `hbase-site.xml` file on every client:

```
<property>
```

```
  <name>hbase.rpc.protection</name>
  <value>privacy</value>
</property>
```

This configuration property can also be set on a per-connection basis. Set it in the `Configuration` supplied to `Table`:

```
Configuration conf = HBaseConfiguration.create();
Connection connection = ConnectionFactory.createConnection(conf);
conf.set("hbase.rpc.protection", "privacy");
try (Connection connection = ConnectionFactory.createConnection(conf);
     Table table = connection.getTable(TableName.valueOf(tablename))) {
  .... do your stuff
}
```

Expect a ~10% performance penalty for encrypted communication.

Client-side Configuration for Secure Operation - Thrift Gateway

Add the following to the `hbase-site.xml` file for every Thrift gateway:

```
<property>
  <name>hbase.thrift.keytab.file</name>
  <value>/etc/hbase/conf/hbase.keytab</value>
</property>
<property>
  <name>hbase.thrift.kerberos.principal</name>
  <value>$USER/_HOST@HADOOP.LOCALDOMAIN</value>
  <!-- TODO: This may need to be  HTTP/_HOST@<REALM> and _HOST may not work.
   You may have  to put the concrete full hostname.
   -->
</property>
<!-- Add these if you need to configure a different DNS interface from the defaul
t -->
<property>
  <name>hbase.thrift.dns.interface</name>
  <value>default</value>
</property>
<property>
  <name>hbase.thrift.dns.nameserver</name>
  <value>default</value>
</property>
```

Substitute the appropriate credential and keytab for *$USER* and *$KEYTAB* respectively.

In order to use the Thrift API principal to interact with HBase, it is also necessary to add the `hbase.thrift.kerberos.principal` to the `acl` table. For example, to give the Thrift API principal, `thrift_server`, administrative access, a command such as this one will suffice:

```
grant 'thrift_server', 'RWCA'
```

For more information about ACLs, please see the Access Control Labels (ACLs) section

The Thrift gateway will authenticate with HBase using the supplied credential. No authentication will be performed by the Thrift gateway itself. All client access via the Thrift gateway will use the Thrift gateway's credential and have its privilege.

Configure the Thrift Gateway to Authenticate on Behalf of the Client

Client-side Configuration for Secure Operation - Thrift Gateway describes how to authenticate a Thrift client to HBase using a fixed user. As an alternative, you can configure the Thrift gateway to authenticate to HBase on the client's behalf, and to access HBase using a proxy user. This was implemented in HBASE-11349 for Thrift 1, and HBASE-11474 for Thrift 2.

> If you use framed transport, you cannot yet take advantage of this feature, because SASL does not work with Thrift framed transport at this time.

To enable it, do the following.

1. Be sure Thrift is running in secure mode, by following the procedure described in Client-side Configuration for Secure Operation - Thrift Gateway.
2. Be sure that HBase is configured to allow proxy users, as described in REST Gateway Impersonation Configuration.
3. In *hbase-site.xml* for each cluster node running a Thrift gateway, set the property `hbase.thrift.security.qop` to one of the following three values:
   - `privacy` - authentication, integrity, and confidentiality checking.
   - `integrity` - authentication and integrity checking

- `authentication` - authentication checking only

4. Restart the Thrift gateway processes for the changes to take effect. If a node is running Thrift, the output of the `jps` command will list a `ThriftServer` process. To stop Thrift on a node, run the command `bin/hbase-daemon.sh stop thrift`. To start Thrift on a node, run the command `bin/hbase-daemon.sh start thrift`.

Configure the Thrift Gateway to Use the `doAs` Feature

Configure the Thrift Gateway to Authenticate on Behalf of the Client describes how to configure the Thrift gateway to authenticate to HBase on the client's behalf, and to access HBase using a proxy user. The limitation of this approach is that after the client is initialized with a particular set of credentials, it cannot change these credentials during the session. The `doAs` feature provides a flexible way to impersonate multiple principals using the same client. This feature was implemented in HBASE-12640 for Thrift 1, but is currently not available for Thrift 2.

To enable the `doAs` feature, add the following to the *hbase-site.xml* file for every Thrift gateway:

```
<property>
  <name>hbase.regionserver.thrift.http</name>
  <value>true</value>
</property>
<property>
  <name>hbase.thrift.support.proxyuser</name>
  <value>true</value>
</property>
```

To allow proxy users when using `doAs` impersonation, add the following to the *hbase-site.xml* file for every HBase node:

```
<property>
  <name>hadoop.security.authorization</name>
  <value>true</value>
</property>
<property>
  <name>hadoop.proxyuser.$USER.groups</name>
  <value>$GROUPS</value>
</property>
<property>
  <name>hadoop.proxyuser.$USER.hosts</name>
```

```
  <value>$GROUPS</value>
</property>
```

Take a look at the demo client to get an overall idea of how to use this feature in your client.

Client-side Configuration for Secure Operation - REST Gateway

Add the following to the `hbase-site.xml` file for every REST gateway:

```
<property>
  <name>hbase.rest.keytab.file</name>
  <value>$KEYTAB</value>
</property>
<property>
  <name>hbase.rest.kerberos.principal</name>
  <value>$USER/_HOST@HADOOP.LOCALDOMAIN</value>
</property>
```

Substitute the appropriate credential and keytab for *$USER* and *$KEYTAB* respectively.

The REST gateway will authenticate with HBase using the supplied credential.

In order to use the REST API principal to interact with HBase, it is also necessary to add the `hbase.rest.kerberos.principal` to the `acl` table. For example, to give the REST API principal, `rest_server`, administrative access, a command such as this one will suffice:

```
grant 'rest_server', 'RWCA'
```

For more information about ACLs, please see the Access Control Labels (ACLs) section

HBase REST gateway supports SPNEGO HTTP authentication for client access to the gateway. To enable REST gateway Kerberos authentication for client access, add the following to the `hbase-site.xml` file for every REST gateway.

```
<property>
  <name>hbase.rest.support.proxyuser</name>
  <value>true</value>
</property>
```

```
<property>
  <name>hbase.rest.authentication.type</name>
  <value>kerberos</value>
</property>
<property>
  <name>hbase.rest.authentication.kerberos.principal</name>
  <value>HTTP/_HOST@HADOOP.LOCALDOMAIN</value>
</property>
<property>
  <name>hbase.rest.authentication.kerberos.keytab</name>
  <value>$KEYTAB</value>
</property>
<!-- Add these if you need to configure a different DNS interface from the defaul
t -->
<property>
  <name>hbase.rest.dns.interface</name>
  <value>default</value>
</property>
<property>
  <name>hbase.rest.dns.nameserver</name>
  <value>default</value>
</property>
```

Substitute the keytab for HTTP for *$KEYTAB*.

HBase REST gateway supports different 'hbase.rest.authentication.type': simple, kerberos. You can also implement a custom authentication by implementing Hadoop AuthenticationHandler, then specify the full class name as 'hbase.rest.authentication.type' value. For more information, refer to SPNEGO HTTP authentication.

REST Gateway Impersonation Configuration

By default, the REST gateway doesn't support impersonation. It accesses the HBase on behalf of clients as the user configured as in the previous section. To the HBase server, all requests are from the REST gateway user. The actual users are unknown. You can turn on the impersonation support. With impersonation, the REST gateway user is a proxy user. The HBase server knows the actual/real user of each request. So it can apply proper authorizations.

To turn on REST gateway impersonation, we need to configure HBase servers (masters and region servers) to allow proxy users; configure REST gateway to enable impersonation.

To allow proxy users, add the following to the `hbase-site.xml` file for every HBase server:

```
<property>
  <name>hadoop.security.authorization</name>
  <value>true</value>
</property>
<property>
  <name>hadoop.proxyuser.$USER.groups</name>
  <value>$GROUPS</value>
</property>
<property>
  <name>hadoop.proxyuser.$USER.hosts</name>
  <value>$GROUPS</value>
</property>
```

Substitute the REST gateway proxy user for *$USER*, and the allowed group list for *$GROUPS*.

To enable REST gateway impersonation, add the following to the `hbase-site.xml` file for every REST gateway.

```
<property>
  <name>hbase.rest.authentication.type</name>
  <value>kerberos</value>
</property>
<property>
  <name>hbase.rest.authentication.kerberos.principal</name>
  <value>HTTP/_HOST@HADOOP.LOCALDOMAIN</value>
</property>
<property>
  <name>hbase.rest.authentication.kerberos.keytab</name>
  <value>$KEYTAB</value>
</property>
```

Substitute the keytab for HTTP for *$KEYTAB*.

Simple User Access to Apache HBase

Newer releases of Apache HBase (>= 0.92) support optional SASL authentication of clients. See also Matteo Bertozzi's article on Understanding User Authentication and Authorization in Apache HBase.

This describes how to set up Apache HBase and clients for simple user access to HBase resources.

Simple versus Secure Access

The following section shows how to set up simple user access. Simple user access is not a secure method of operating HBase. This method is used to prevent users from making mistakes. It can be used to mimic the Access Control using on a development system without having to set up Kerberos.

This method is not used to prevent malicious or hacking attempts. To make HBase secure against these types of attacks, you must configure HBase for secure operation. Refer to the section Secure Client Access to Apache HBase and complete all of the steps described there.

Prerequisites

None

Server-side Configuration for Simple User Access Operation

Add the following to the `hbase-site.xml` file on every server machine in the cluster:

```
<property>
  <name>hbase.security.authentication</name>
  <value>simple</value>
</property>
<property>
  <name>hbase.security.authorization</name>
  <value>true</value>
</property>
<property>
  <name>hbase.coprocessor.master.classes</name>
  <value>org.apache.hadoop.hbase.security.access.AccessController</value>
</property>
<property>
  <name>hbase.coprocessor.region.classes</name>
  <value>org.apache.hadoop.hbase.security.access.AccessController</value>
</property>
<property>
  <name>hbase.coprocessor.regionserver.classes</name>
  <value>org.apache.hadoop.hbase.security.access.AccessController</value>
</property>
```

For 0.94, add the following to the `hbase-site.xml` file on every server machine in the cluster:

```
<property>
  <name>hbase.rpc.engine</name>
  <value>org.apache.hadoop.hbase.ipc.SecureRpcEngine</value>
</property>
<property>
  <name>hbase.coprocessor.master.classes</name>
  <value>org.apache.hadoop.hbase.security.access.AccessController</value>
</property>
<property>
  <name>hbase.coprocessor.region.classes</name>
  <value>org.apache.hadoop.hbase.security.access.AccessController</value>
</property>
```

A full shutdown and restart of HBase service is required when deploying these configuration changes.

Client-side Configuration for Simple User Access Operation

Add the following to the `hbase-site.xml` file on every client:

```
<property>
  <name>hbase.security.authentication</name>
  <value>simple</value>
</property>
```

For 0.94, add the following to the `hbase-site.xml` file on every server machine in the cluster:

```
<property>
  <name>hbase.rpc.engine</name>
  <value>org.apache.hadoop.hbase.ipc.SecureRpcEngine</value>
</property>
```

Be advised that if the `hbase.security.authentication` in the client- and server-side site files do not match, the client will not be able to communicate with the cluster.

Client-side Configuration for Simple User Access Operation - Thrift Gateway

The Thrift gateway user will need access. For example, to give the Thrift API user, `thrift_server`, administrative access, a command such as this one will suffice:

```
grant 'thrift_server', 'RWCA'
```

For more information about ACLs, please see the Access Control Labels (ACLs) section

The Thrift gateway will authenticate with HBase using the supplied credential. No authentication will be performed by the Thrift gateway itself. All client access via the Thrift gateway will use the Thrift gateway's credential and have its privilege.

Client-side Configuration for Simple User Access Operation - REST Gateway

The REST gateway will authenticate with HBase using the supplied credential. No authentication will be performed by the REST gateway itself. All client access via the REST gateway will use the REST gateway's credential and have its privilege.

The REST gateway user will need access. For example, to give the REST API user, `rest_server`, administrative access, a command such as this one will suffice:

```
grant 'rest_server', 'RWCA'
```

For more information about ACLs, please see the Access Control Labels (ACLs) section

It should be possible for clients to authenticate with the HBase cluster through the REST gateway in a pass-through manner via SPNEGO HTTP authentication. This is future work.

Transport Level Security (TLS) in HBase RPC communication

Since version `2.6.0` HBase supports TLS encryption in server-client and Master-RegionServer communication. Transport Layer Security (TLS) is a standard cryptographic

protocol designed to provide communications security over a computer network. HBase TLS implementation works exactly how secure websites are accessed via **https** prefix in a web browser: once established all communication on the channel will be securely hidden from malicious access.

The encryption works at the transport level which means it's independent of the configured authentication method. Secure client access mentioned in the previous section requires Kerberos to be configured and used in HBase authentication, while TLS can be configured with any other SASL mechanism or even with simple client access methods, effectively preventing attackers from eavesdropping the communication. No Kerberos KDC or other complicated infrastructure required.

HBase TLS is based on the Netty library therefore it only works with Netty client and server RPC implementations. Netty's powerful SSL implementation is a great foundation for highly secure and performant communication providing the latest and greatest cryptographic solution at all times.

Since Region Servers effectively work as clients from Master's perspective, TLS supports encrypted communication between cluster members too.

> From version 2.6.0 HBase supports the Hadoop CredentialProvider API to avoid storing sensitive information in HBase configuration files. The recommended way of storing keystore / truststore passwords is to use one of the supported credential providers e.g. the local jceks file provider. You can find more information about how to setup credential providers in the Hadoop documentation.
>
> The CLI interface for accessing the Hadoop Credential Shell is also available in HBase CLI. Type `hbase credential` to get help.

Server side configuration

We need to set up Java key store for the server. Key store is the list of private keys that a server can use to configure TLS encryption. See TLS wikipedia page for further details of the protocol. Add the following configuration to `hbase-site.xml` on Master, Region Servers and HBase clients:

```
<property>
  <name>hbase.server.netty.tls.enabled</name>
  <value>true</value>
</property>
```

```
<property>
  <name>hbase.rpc.tls.keystore.location</name>
  <value>/path/to/keystore.jks</value>
</property>
```

Use `hbase.rpc.tls.keystore.password` alias to retrieve key store password from Hadoop credential provider.

> The supported storefile formats are based on the registered security providers and the loader can be autodetected from the file extension. If needed, the file format can be explicitly specified with the `hbase.rpc.tls.keystore.type` property.

Client side configuration

We need to configure trust store for the client. Trust store contains the list of certificates that the client should trust when doing the handshake with the server. Add the following to `hbase-site.xml`.

```
<property>
  <name>hbase.client.netty.tls.enabled</name>
  <value>true</value>
</property>
<property>
  <name>hbase.rpc.tls.truststore.location</name>
  <value>/path/to/truststore.jks</value>
</property>
```

Use `hbase.rpc.tls.truststore.password` alias to retrieve trust store password from Hadoop credential provider.

> The supported storefile formats are based on the registered security providers and the loader can be autodetected from the file extension. If needed, the file format can be explicitly specified with the `hbase.rpc.tls.truststore.type` property.

However, specifying a trust store is not always required. Standard JDK implementations are shipped with a standard list of trusted certificates (the certificates of Certificate Authorities) and if your private key is provided by one of them, you don't need to configure your clients to trust it. Similarly to an internet browser, you don't need to set up the certificates of every single website you're planning to visit. Later in this documentation

we'll walk through the steps of creating self-signed certificates which requires a trust store setup.

You can check the list of public certificate authorities shipped with your JDK implementation:

```
keytool -keystore $JAVA_HOME/jre/lib/security/cacerts -list
```

Password is empty by default.

Creating self-signed certificates

While obtaining globally trusted certificates from Certificate Authorities is convenient, it's perfectly valid to generate your own private/public keypairs and set them up specifically for the HBase cluster. Especially if you don't want to enable public access to the cluster, paying money for a certificate doesn't make sense.

Follow the following steps to generate self-signed certificates.

1. Create SSL key store JKS to store local credentials

   Please note that the alias (-alias) and the distinguished name (-dname) must match the hostname of the machine that is associated with, otherwise hostname verification won't work.

   ```
   keytool -genkeypair -alias $(hostname -f) -keyalg RSA -keysize 2048 -dname "c
   n=$(hostname -f)" -keypass password -keystore keystore.jks -storepass passwor
   d
   ```

   At the end of this operation you'll have as many key store files as many servers you have in your cluster. Each cluster member will have its own key store.

2. Extract the signed public key (certificate) from each key store

   ```
   keytool -exportcert -alias $(hostname -f) -keystore keystore.jks -file $(host
   name -f).cer -rfc
   ```

3. Create SSL trust store JKS containing certificates for the clients

The same truststore (storing all accepted certs) should be shared on participants of the cluster. You need to use different aliases to store multiple certificates in the same truststore. Name of the aliases doesn't matter.

```
keytool -importcert -alias [host1..3] -file [host1..3].cer -keystore truststo
re.jks -storepass password
```

Upgrading existing non-TLS cluster with no downtime

Here are the steps needed to upgrade an already running HBase cluster to TLS without downtime by taking advantage of port unification functionality. There's a property on server side called `hbase.server.netty.tls.supportplaintext` which makes possible to accept TLS and plaintext connections on the same socket port.

1. Create the necessary key stores and trust stores for all server participants as described in the previous section.

2. Enable secure communication on the Master node in *server-only* mode with plaintext support

   ```
   <property>
     <name>hbase.client.netty.tls.enabled</name>
     <value>false</value>
   </property>
   <property>
     <name>hbase.server.netty.tls.enabled</name>
     <value>true</value>
   </property>
   <property>
     <name>hbase.server.netty.tls.supportplaintext</name>
     <value>true</value>
   </property>
   ...keystore / truststore setup ...
   ```

   Restart the Master. Master now accepts both TLS/non-TLS connections and works with non-TLS in client mode.

3. Enable secure communication on the Region Servers in both *server and client* mode with plaintext support

Client mode here will ensure that RegionServer's communication to Master is encrypted.

> ⚠ **Replication**
>
> If you have read replicas enabled in your cluster or replication between two different clusters, you have to break this into two steps. Secure communication has to be enabled on the *server side* first with plaintext support and once all Region Servers are upgraded you can repeat the upgrade by enabling *client side* as well.
>
> You have to prepare all Region Servers for secure communication before upgrading the client side.

```
<property>
  <name>hbase.client.netty.tls.enabled</name>
  <value>true</value>
</property>
<property>
  <name>hbase.server.netty.tls.enabled</name>
  <value>true</value>
</property>
<property>
  <name>hbase.server.netty.tls.supportplaintext</name>
  <value>true</value>
</property>
...keystore / truststore setup ...
```

Restart Region Servers in rolling restart fashion. They send requests with TLS and accept both TLS and non-TLS communication.

4 Enable secure communication on the clients

```
<property>
  <name>hbase.client.netty.tls.enabled</name>
  <value>true</value>
</property>
...truststore setup ...
```

5 Enable client-mode TLS on master and disable plaintext mode

```
<property>
  <name>hbase.client.netty.tls.enabled</name>
  <value>true</value>
</property>
<property>
  <name>hbase.server.netty.tls.enabled</name>
  <value>true</value>
```

```
</property>
<property>
  <name>hbase.server.netty.tls.supportplaintext</name>
  <value>false</value>
</property>
```

Restart Master.

6 Disable plaintext communication on the Region Servers

Disable plaintext communication on the Region Servers by removing `supportplaintext` property. Restart RSs in rolling restart fashion.

> ⚠ Once `hbase.client.netty.tls.enabled` is enabled on the server side, the cluster will only be able to communicate with other clusters which have TLS enabled. For example, this would impact inter-cluster replication.

Enable automatic certificate reloading

Certificates usually expire after some time to improve security. In this case we need to replace them by modifying Keystore / Truststore files and HBase processes have to be restarted. In order to avoid that you can enable automatic file change detection and certificate reloading with the following option. Default: false.

```
<property>
  <name>hbase.rpc.tls.certReload</name>
  <value>true</value>
</property>
```

Additional configuration

Enabled protocols

Comma-separated list of TLS protocol versions to enable. Default is empty.

```
<property>
  <name>hbase.client.netty.tls.enabledProtocols</name>
  <value>TLSv1.2,TLSv1.3</value>
</property>
```

Default protocol

Set the default TLS protocol version to use. Default is TLSv1.2. Use this protocol if enabled protocols is not defined.

```
<property>
  <name>hbase.client.netty.tls.protocol</name>
  <value>TLSv1.2</value>
</property>
```

Enabled cipher suites

List of enabled cipher suites in TLS protocol. Useful when you want to disable certain cipher suites due to recent security concerns. Default value is a mix of CBC and GCM ciphers. Due to performance reasons we prefer CBC ciphers for Java 8 and GCM ciphers for Java 9+.

```
<property>
  <name>hbase.client.netty.tls.ciphersuites</name>
  <value>TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256</value>
</property>
```

Certificate Revocation Checking

There's a built-in mechanism in JDK's TrustManager which automatically checks certificates for revocation. See Managing Server Certificates. Disabled by default.

```
<property>
  <name>hbase.client.netty.tls.clr</name>
  <value>false</value>
</property>
```

Online Certificate Status Protocol

Enables OCSP stapling. Please note that not all `SSLProvider` implementations support OCSP stapling and an exception will be thrown upon. Disabled by default.

```
<property>
  <name>hbase.client.netty.tls.ocsp</name>
  <value>false</value>
```

```
</property>
```

Client handshake timeout

Set the TLS client handshake timeout is milliseconds. Default is 5 seconds.

```
<property>
  <name>hbase.client.netty.tls.handshaketimeout</name>
  <value>5000</value>
</property>
```

Securing Access to HDFS and ZooKeeper

Securing ZooKeeper Data

ZooKeeper has a pluggable authentication mechanism to enable access from clients using different methods. ZooKeeper even allows authenticated and un-authenticated clients at the same time. The access to znodes can be restricted by providing Access Control Lists (ACLs) per znode. An ACL contains two components, the authentication method and the principal. ACLs are NOT enforced hierarchically. See ZooKeeper Programmers Guide for details.

HBase daemons authenticate to ZooKeeper via SASL and kerberos (See ZooKeeper). HBase sets up the znode ACLs so that only the HBase user and the configured hbase superuser (`hbase.superuser`) can access and modify the data. In cases where ZooKeeper is used for service discovery or sharing state with the client, the znodes created by HBase will also allow anyone (regardless of authentication) to read these znodes (clusterId, master address, meta location, etc), but only the HBase user can modify them.

Securing File System (HDFS) Data

All of the data under management is kept under the root directory in the file system (`hbase.rootdir`). Access to the data and WAL files in the filesystem should be restricted so that users cannot bypass the HBase layer, and peek at the underlying data files from the file system. HBase assumes the filesystem used (HDFS or other) enforces permissions hierarchically. If sufficient protection from the file system (both authorization and

authentication) is not provided, HBase level authorization control (ACLs, visibility labels, etc) is meaningless since the user can always access the data from the file system.

HBase enforces the posix-like permissions 700 (`rwx------`) to its root directory. It means that only the HBase user can read or write the files in FS. The default setting can be changed by configuring `hbase.rootdir.perms` in hbase-site.xml. A restart of the active master is needed so that it changes the used permissions. For versions before 1.2.0, you can check whether HBASE-13780 is committed, and if not, you can manually set the permissions for the root directory if needed. Using HDFS, the command would be:

```
sudo -u hdfs hadoop fs -chmod 700 /hbase
```

You should change `/hbase` if you are using a different `hbase.rootdir`.

Securing Access To Your Data

After you have configured secure authentication between HBase client and server processes and gateways, you need to consider the security of your data itself. HBase provides several strategies for securing your data:

- Role-based Access Control (RBAC) controls which users or groups can read and write to a given HBase resource or execute a coprocessor endpoint, using the familiar paradigm of roles.
- Visibility Labels which allow you to label cells and control access to labelled cells, to further restrict who can read or write to certain subsets of your data. Visibility labels are stored as tags. See hbase.tags for more information.
- Transparent encryption of data at rest on the underlying filesystem, both in HFiles and in the WAL. This protects your data at rest from an attacker who has access to the underlying filesystem, without the need to change the implementation of the client. It can also protect against data leakage from improperly disposed disks, which can be important for legal and regulatory compliance.

Server-side configuration, administration, and implementation details of each of these features are discussed below, along with any performance trade-offs. An example security configuration is given at the end, to show these features all used together, as they might be in a real-world scenario.

All aspects of security in HBase are in active development and evolving rapidly. Any strategy you employ for security of your data should be thoroughly tested. In addition, some of these features are still in the experimental stage of development. To take advantage of many of these features, you must be running HBase 0.98+ and using the HFile v3 file format.

Several procedures in this section require you to copy files between cluster nodes. When copying keys, configuration files, or other files containing sensitive strings, use a secure method, such as `ssh`, to avoid leaking sensitive data.

Procedure: Basic Server-Side Configuration

1. Enable HFile v3, by setting `hfile.format.version` to 3 in *hbase-site.xml*. This is the default for HBase 1.0 and newer.

```
<property>
  <name>hfile.format.version</name>
  <value>3</value>
</property>
```

2. Enable SASL and Kerberos authentication for RPC and ZooKeeper, as described in security.prerequisites and ZooKeeper.

Tags

Tags are a feature of HFile v3. A tag is a piece of metadata which is part of a cell, separate from the key, value, and version. Tags are an implementation detail which provides a foundation for other security-related features such as cell-level ACLs and visibility labels. Tags are stored in the HFiles themselves. It is possible that in the future, tags will be used to implement other HBase features. You don't need to know a lot about tags in order to use the security features they enable.

Implementation Details

Every cell can have zero or more tags. Every tag has a type and the actual tag byte array.

Just as row keys, column families, qualifiers and values can be encoded (see data.block.encoding.types), tags can also be encoded as well. You can enable or disable tag encoding at the level of the column family, and it is enabled by default. Use the `HColum`

`nDescriptor#setCompressionTags(boolean compressTags)` method to manage encoding settings on a column family. You also need to enable the DataBlockEncoder for the column family, for encoding of tags to take effect.

You can enable compression of each tag in the WAL, if WAL compression is also enabled, by setting the value of `hbase.regionserver.wal.tags.enablecompression` to `true` in *hbase-site.xml*. Tag compression uses dictionary encoding.

Coprocessors that run server-side on RegionServers can perform get and set operations on cell Tags. Tags are stripped out at the RPC layer before the read response is sent back, so clients do not see these tags. Tag compression is not supported when using WAL encryption.

Access Control Labels (ACLs)

How It Works

ACLs in HBase are based upon a user's membership in or exclusion from groups, and a given group's permissions to access a given resource. ACLs are implemented as a coprocessor called AccessController.

HBase does not maintain a private group mapping, but relies on a ***Hadoop group mapper***, which maps between entities in a directory such as LDAP or Active Directory, and HBase users. Any supported Hadoop group mapper will work. Users are then granted specific permissions (Read, Write, Execute, Create, Admin) against resources (global, namespaces, tables, cells, or endpoints).

> With Kerberos and Access Control enabled, client access to HBase is authenticated and user data is private unless access has been explicitly granted.

HBase has a simpler security model than relational databases, especially in terms of client operations. No distinction is made between an insert (new record) and update (of existing record), for example, as both collapse down into a Put.

Understanding Access Levels

HBase access levels are granted independently of each other and allow for different types of operations at a given scope.

- *Read (R)* - can read data at the given scope
- *Write (W)* - can write data at the given scope
- *Execute (X)* - can execute coprocessor endpoints at the given scope
- *Create (C)* - can create tables or drop tables (even those they did not create) at the given scope
- *Admin (A)* - can perform cluster operations such as balancing the cluster or assigning regions at the given scope

The possible scopes are:

- *Superuser* - superusers can perform any operation available in HBase, to any resource. The user who runs HBase on your cluster is a superuser, as are any principals assigned to the configuration property `hbase.superuser` in *hbase-site.xml* on the HMaster.
- *Global* - permissions granted at *global* scope allow the admin to operate on all tables of the cluster.
- *Namespace* - permissions granted at *namespace* scope apply to all tables within a given namespace.
- *Table* - permissions granted at *table* scope apply to data or metadata within a given table.
- *ColumnFamily* - permissions granted at *ColumnFamily* scope apply to cells within that ColumnFamily.
- *Cell* - permissions granted at *cell* scope apply to that exact cell coordinate (key, value, timestamp). This allows for policy evolution along with data. To change an ACL on a specific cell, write an updated cell with new ACL to the precise coordinates of the original.
  If you have a multi-versioned schema and want to update ACLs on all visible versions, you need to write new cells for all visible versions. The application has complete control over policy evolution.
  The exception to the above rule is `append` and `increment` processing. Appends and increments can carry an ACL in the operation. If one is included in the operation, then it will be applied to the result of the `append` or `increment`. Otherwise, the ACL of the existing cell you are appending to or incrementing is preserved.

The combination of access levels and scopes creates a matrix of possible access levels that can be granted to a user. In a production environment, it is useful to think of access

levels in terms of what is needed to do a specific job. The following list describes appropriate access levels for some common types of HBase users. It is important not to grant more access than is required for a given user to perform their required tasks.

- *Superusers* - In a production system, only the HBase user should have superuser access. In a development environment, an administrator may need superuser access in order to quickly control and manage the cluster. However, this type of administrator should usually be a Global Admin rather than a superuser.
- *Global Admins* - A global admin can perform tasks and access every table in HBase. In a typical production environment, an admin should not have Read or Write permissions to data within tables.
- A global admin with Admin permissions can perform cluster-wide operations on the cluster, such as balancing, assigning or unassigning regions, or calling an explicit major compaction. This is an operations role.
- A global admin with Create permissions can create or drop any table within HBase. This is more of a DBA-type role. In a production environment, it is likely that different users will have only one of Admin and Create permissions.

  > ⚠ In the current implementation, a Global Admin with `Admin` permission can grant himself `Read` and `Write` permissions on a table and gain access to that table's data. For this reason, only grant `Global Admin` permissions to trusted user who actually need them. Also be aware that a `Global Admin` with `Create` permission can perform a `Put` operation on the ACL table, simulating a `grant` or `revoke` and circumventing the authorization check for `Global Admin` permissions. Due to these issues, be cautious with granting `Global Admin` privileges.

- *Namespace Admins* - a namespace admin with `Create` permissions can create or drop tables within that namespace, and take and restore snapshots. A namespace admin with `Admin` permissions can perform operations such as splits or major compactions on tables within that namespace.
- *Table Admins* - A table admin can perform administrative operations only on that table. A table admin with `Create` permissions can create snapshots from that table or restore that table from a snapshot. A table admin with `Admin` permissions can perform operations such as splits or major compactions on that table.
- *Users* - Users can read or write data, or both. Users can also execute coprocessor endpoints, if given `Executable` permissions.

Real-World Example of Access Levels

Job Title	Scope	Permissions	Description
Senior Administrator	Global	Access, Create	Manages the cluster and gives access to Junior Administrators.
Junior Administrator	Global	Create	Creates tables and gives access to Table Administrators.
Table Administrator	Table	Access	Maintains a table from an operations point of view.
Data Analyst	Table	Read	Creates reports from HBase data.
Web Application	Table	Read, Write	Puts data into HBase and uses HBase data to perform operations.

ACL Matrix

For more details on how ACLs map to specific HBase operations and tasks, see appendix acl matrix.

Implementation Details

Cell-level ACLs are implemented using tags (see Tags). In order to use cell-level ACLs, you must be using HFile v3 and HBase 0.98 or newer.

1. Files created by HBase are owned by the operating system user running the HBase process. To interact with HBase files, you should use the API or bulk load facility.
2. HBase does not model "roles" internally in HBase. Instead, group names can be granted permissions. This allows external modeling of roles via group membership. Groups are created and manipulated externally to HBase, via the Hadoop group mapping service.

Server-Side Configuration

1 As a prerequisite, perform the steps in Procedure: Basic Server-Side Configuration.

2 Install and configure the AccessController coprocessor, by setting the following properties in *hbase-site.xml*. These properties take a list of classes.

> If you use the AccessController along with the VisibilityController, the AccessController must come first in the list, because with both components active, the VisibilityController will delegate access control on its system tables to the AccessController. For an example of using both together, see Security Configuration Example.

```
<property>
  <name>hbase.security.authorization</name>
  <value>true</value>
</property>
<property>
  <name>hbase.coprocessor.region.classes</name>
  <value>
    org.apache.hadoop.hbase.security.access.AccessController,
    org.apache.hadoop.hbase.security.token.TokenProvider
  </value>
</property>
<property>
  <name>hbase.coprocessor.master.classes</name>
  <value>org.apache.hadoop.hbase.security.access.AccessController</value>
</property>
<property>
  <name>hbase.coprocessor.regionserver.classes</name>
  <value>org.apache.hadoop.hbase.security.access.AccessController</value>
</property>
<property>
  <name>hbase.security.exec.permission.checks</name>
  <value>true</value>
</property>
```

Optionally, you can enable transport security, by setting `hbase.rpc.protection` to `privacy`. This requires HBase 0.98.4 or newer.

3 Set up the Hadoop group mapper in the Hadoop namenode's *core-site.xml*. This is a Hadoop file, not an HBase file. Customize it to your site's needs. Following is an example.

```
<property>
  <name>hadoop.security.group.mapping</name>
  <value>org.apache.hadoop.security.LdapGroupsMapping</value>
</property>

<property>
  <name>hadoop.security.group.mapping.ldap.url</name>
  <value>ldap://server</value>
</property>
```

```
<property>
  <name>hadoop.security.group.mapping.ldap.bind.user</name>
  <value>Administrator@example-ad.local</value>
</property>

<property>
  <name>hadoop.security.group.mapping.ldap.bind.password</name>
  <value>****</value>
</property>

<property>
  <name>hadoop.security.group.mapping.ldap.base</name>
  <value>dc=example-ad,dc=local</value>
</property>

<property>
  <name>hadoop.security.group.mapping.ldap.search.filter.user</name>
  <value>(&amp;(objectClass=user)(sAMAccountName={0}))</value>
</property>

<property>
  <name>hadoop.security.group.mapping.ldap.search.filter.group</name>
  <value>(objectClass=group)</value>
</property>

<property>
  <name>hadoop.security.group.mapping.ldap.search.attr.member</name>
  <value>member</value>
</property>

<property>
  <name>hadoop.security.group.mapping.ldap.search.attr.group.name</name>
  <value>cn</value>
</property>
```

4 Optionally, enable the early-out evaluation strategy. Prior to HBase 0.98.0, if a user was not granted access to a column family, or at least a column qualifier, an AccessDeniedException would be thrown. HBase 0.98.0 removed this exception in order to allow cell-level exceptional grants. To restore the old behavior in HBase 0.98.0-0.98.6, set `hbase.security.access.early_out` to `true` in *hbase-site.xml*. In HBase 0.98.6, the default has been returned to `true`.

5 Distribute your configuration and restart your cluster for changes to take effect.

6 To test your configuration, log into HBase Shell as a given user and use the `whoami` command to report the groups your user is part of. In this example, the user is reported as being a member of the `services` group.

```
hbase> whoami
service (auth:KERBEROS)
    groups: services
```

Administration

Administration tasks can be performed from HBase Shell or via an API.

> Many of the API examples below are taken from source files *hbase-server/src/test/java/org/apache/hadoop/hbase/security/access/TestAccessController.java* and *hbase-server/src/test/java/org/apache/hadoop/hbase/security/access/SecureTestUtil.java*.
>
> Neither the examples, nor the source files they are taken from, are part of the public HBase API, and are provided for illustration only. Refer to the official API for usage instructions.

1. As a prerequisite, perform the steps in Procedure: Basic Server-Side Configuration..

2. Install and configure the AccessController coprocessor, by setting the following properties in *hbase-site.xml*. These properties take a list of classes.

   > If you use the AccessController along with the VisibilityController, the AccessController must come first in the list, because with both components active, the VisibilityController will delegate access control on its system tables to the AccessController. For an example of using both together, see Security Configuration Example.

   ```
   <property>
     <name>hbase.security.authorization</name>
     <value>true</value>
   </property>
   <property>
     <name>hbase.coprocessor.region.classes</name>
     <value>
       org.apache.hadoop.hbase.security.access.AccessController,
       org.apache.hadoop.hbase.security.token.TokenProvider
     </value>
   </property>
   <property>
     <name>hbase.coprocessor.master.classes</name>
     <value>org.apache.hadoop.hbase.security.access.AccessController</value>
   </property>
   <property>
     <name>hbase.coprocessor.regionserver.classes</name>
     <value>org.apache.hadoop.hbase.security.access.AccessController</value>
   </property>
   <property>
     <name>hbase.security.exec.permission.checks</name>
     <value>true</value>
   ```

```
</property>
```

Optionally, you can enable transport security, by setting `hbase.rpc.protection` to `privacy`. This requires HBase 0.98.4 or newer.

3 Set up the Hadoop group mapper in the Hadoop namenode's *core-site.xml*. This is a Hadoop file, not an HBase file. Customize it to your site's needs. Following is an example.

```
<property>
  <name>hadoop.security.group.mapping</name>
  <value>org.apache.hadoop.security.LdapGroupsMapping</value>
</property>

<property>
  <name>hadoop.security.group.mapping.ldap.url</name>
  <value>ldap://server</value>
</property>

<property>
  <name>hadoop.security.group.mapping.ldap.bind.user</name>
  <value>Administrator@example-ad.local</value>
</property>

<property>
  <name>hadoop.security.group.mapping.ldap.bind.password</name>
  <value>****</value>
</property>

<property>
  <name>hadoop.security.group.mapping.ldap.base</name>
  <value>dc=example-ad,dc=local</value>
</property>

<property>
  <name>hadoop.security.group.mapping.ldap.search.filter.user</name>
  <value>(&amp;(objectClass=user)(sAMAccountName={0}))</value>
</property>

<property>
  <name>hadoop.security.group.mapping.ldap.search.filter.group</name>
  <value>(objectClass=group)</value>
</property>

<property>
  <name>hadoop.security.group.mapping.ldap.search.attr.member</name>
  <value>member</value>
</property>

<property>
  <name>hadoop.security.group.mapping.ldap.search.attr.group.name</name>
```

```
  <value>cn</value>
</property>
```

4. Optionally, enable the early-out evaluation strategy. Prior to HBase 0.98.0, if a user was not granted access to a column family, or at least a column qualifier, an AccessDeniedException would be thrown. HBase 0.98.0 removed this exception in order to allow cell-level exceptional grants. To restore the old behavior in HBase 0.98.0-0.98.6, set `hbase.security.access.early_out` to `true` in *hbase-site.xml*. In HBase 0.98.6, the default has been returned to `true`.

5. Distribute your configuration and restart your cluster for changes to take effect.

6. To test your configuration, log into HBase Shell as a given user and use the `whoami` command to report the groups your user is part of. In this example, the user is reported as being a member of the `services` group.

```
hbase> whoami
service (auth:KERBEROS)
    groups: services
```

API example:

```
public static void verifyAllowed(User user, AccessTestAction action, int count) t
hrows Exception {
  try {
    Object obj = user.runAs(action);
    if (obj != null && obj instanceof List<?>) {
      List<?> results = (List<?>) obj;
      if (results != null && results.isEmpty()) {
        fail("Empty non null results from action for user '" + user.getShortName
() + "'");
      }
      assertEquals(count, results.size());
    }
  } catch (AccessDeniedException ade) {
    fail("Expected action to pass for user '" + user.getShortName() + "' but was
denied");
  }
}
```

Visibility Labels

Visibility labels control can be used to only permit users or principals associated with a given label to read or access cells with that label. For instance, you might label a cell `top-secret`, and only grant access to that label to the `managers` group. Visibility labels are implemented using Tags, which are a feature of HFile v3, and allow you to store metadata on a per-cell basis. A label is a string, and labels can be combined into expressions by using logical operators (&, |, or !), and using parentheses for grouping. HBase does not do any kind of validation of expressions beyond basic well-formedness. Visibility labels have no meaning on their own, and may be used to denote sensitivity level, privilege level, or any other arbitrary semantic meaning.

If a user's labels do not match a cell's label or expression, the user is denied access to the cell.

In HBase 0.98.6 and newer, UTF-8 encoding is supported for visibility labels and expressions. When creating labels using the `addLabels(conf, labels)` method provided by the `org.apache.hadoop.hbase.security.visibility.VisibilityClient` class and passing labels in Authorizations via Scan or Get, labels can contain UTF-8 characters, as well as the logical operators normally used in visibility labels, with normal Java notations, without needing any escaping method. However, when you pass a CellVisibility expression via a Mutation, you must enclose the expression with the `CellVisibility.quote()` method if you use UTF-8 characters or logical operators. See `TestExpressionParser` and the source file *hbase-client/src/test/java/org/apache/hadoop/hbase/client/TestScan.java*.

A user adds visibility expressions to a cell during a Put operation. In the default configuration, the user does not need to have access to a label in order to label cells with it. This behavior is controlled by the configuration option `hbase.security.visibility.mutations.checkauths`. If you set this option to `true`, the labels the user is modifying as part of the mutation must be associated with the user, or the mutation will fail. Whether a user is authorized to read a labelled cell is determined during a Get or Scan, and results which the user is not allowed to read are filtered out. This incurs the same I/O penalty as if the results were returned, but reduces load on the network.

Visibility labels can also be specified during Delete operations. For details about visibility labels and Deletes, see HBASE-10885.

The user's effective label set is built in the RPC context when a request is first received by the RegionServer. The way that users are associated with labels is pluggable. The default plugin passes through labels specified in Authorizations added to the Get or Scan and checks those against the calling user's authenticated labels list. When the client passes labels for which the user is not authenticated, the default plugin drops them. You can pass a subset of user authenticated labels via the `Get#setAuthorizations(Authorizations(String,...))` and `Scan#setAuthorizations(Authorizations(String,...));` methods.

Groups can be granted visibility labels the same way as users. Groups are prefixed with an @ symbol. When checking visibility labels of a user, the server will include the visibility labels of the groups of which the user is a member, together with the user's own labels. When the visibility labels are retrieved using API `VisibilityClient#getAuths` or Shell command `get_auths` for a user, we will return labels added specifically for that user alone, not the group level labels.

Visibility label access checking is performed by the VisibilityController coprocessor. You can use interface `VisibilityLabelService` to provide a custom implementation and/or control the way that visibility labels are stored with cells. See the source file *hbase-server/src/test/java/org/apache/hadoop/hbase/security/visibility/TestVisibilityLabelsWithCustomVisLabService.java* for one example.

Visibility labels can be used in conjunction with ACLs.

> The labels have to be explicitly defined before they can be used in visibility labels. See below for an example of how this can be done.

> There is currently no way to determine which labels have been applied to a cell. See HBASE-12470 for details.

> Visibility labels are not currently applied for superusers.

Examples of Visibility Expressions

Expression	Interpretation
`fulltime`	Allow access to users associated with the fulltime label.

Expression	Interpretation
`!public`	Allow access to users not associated with the pu blic label.
`( secret | topsecret ) & !probationary`	Allow access to users associated with either the secret or topsecret label and not associated with the probationary label.

Server-Side Configuration

1. As a prerequisite, perform the steps in Procedure: Basic Server-Side Configuration..

2. Install and configure the VisibilityController coprocessor by setting the following properties in *hbase-site.xml*. These properties take a list of class names.

```
<property>
  <name>hbase.security.authorization</name>
  <value>true</value>
</property>
<property>
  <name>hbase.coprocessor.region.classes</name>
  <value>org.apache.hadoop.hbase.security.visibility.VisibilityController</v
alue>
</property>
<property>
  <name>hbase.coprocessor.master.classes</name>
  <value>org.apache.hadoop.hbase.security.visibility.VisibilityController</v
alue>
</property>
```

> If you use the AccessController and VisibilityController coprocessors together, the AccessController must come first in the list, because with both components active, the VisibilityController will delegate access control on its system tables to the AccessController.

3. Adjust Configuration

By default, users can label cells with any label, including labels they are not associated with, which means that a user can Put data that he cannot read. For example, a user could label a cell with the (hypothetical) 'topsecret' label even if the user is not associated with that label. If you only want users to be able to label cells with labels they are associated with, set `hbase.security.visibility.mutations.checka`

`uths` to `true`. In that case, the mutation will fail if it makes use of labels the user is not associated with.

4 Distribute your configuration and restart your cluster for changes to take effect.

Administration

Administration tasks can be performed using the HBase Shell or the Java API. For defining the list of visibility labels and associating labels with users, the HBase Shell is probably simpler.

> Many of the Java API examples in this section are taken from the source file *hbase-server/src/test/java/org/apache/hadoop/hbase/security/visibility/TestVisibilityLabels.java*. Refer to that file or the API documentation for more context.
>
> Neither these examples, nor the source file they were taken from, are part of the public HBase API, and are provided for illustration only. Refer to the official API for usage instructions.

1 Define the List of Visibility Labels

HBase Shell:

```
hbase> add_labels [ 'admin', 'service', 'developer', 'test' ]
```

Java API:

```
public static void addLabels() throws Exception {
  PrivilegedExceptionAction<VisibilityLabelsResponse> action = new Privilege
dExceptionAction<VisibilityLabelsResponse>() {
    public VisibilityLabelsResponse run() throws Exception {
      String[] labels = { SECRET, TOPSECRET, CONFIDENTIAL, PUBLIC, PRIVATE,
COPYRIGHT, ACCENT,
          UNICODE_VIS_TAG, UC1, UC2 };
      try {
        VisibilityClient.addLabels(conf, labels);
      } catch (Throwable t) {
        throw new IOException(t);
      }
      return null;
    }
  };
  SUPERUSER.runAs(action);
}
```

2 Associate Labels with Users

HBase Shell:

```
hbase> set_auths 'service', [ 'service' ]
hbase> set_auths 'testuser', [ 'test' ]
hbase> set_auths 'qa', [ 'test', 'developer' ]
hbase> set_auths '@qagroup', [ 'test' ]
```

Java API:

```
public void testSetAndGetUserAuths() throws Throwable {
  final String user = "user1";
  PrivilegedExceptionAction<Void> action = new PrivilegedExceptionAction<Voi
d>() {
    public Void run() throws Exception {
      String[] auths = { SECRET, CONFIDENTIAL };
      try {
        VisibilityClient.setAuths(conf, auths, user);
      } catch (Throwable e) {
      }
      return null;
    }
    ...
```

3 Clear Labels From Users

HBase Shell:

```
hbase> clear_auths 'service', [ 'service' ]
hbase> clear_auths 'testuser', [ 'test' ]
hbase> clear_auths 'qa', [ 'test', 'developer' ]
hbase> clear_auths '@qagroup', [ 'test', 'developer' ]
```

Java API:

```
...
auths = new String[] { SECRET, PUBLIC, CONFIDENTIAL };
VisibilityLabelsResponse response = null;
try {
  response = VisibilityClient.clearAuths(conf, auths, user);
} catch (Throwable e) {
  fail("Should not have failed");
  ...
}
```

4 Apply a Label or Expression to a Cell

The label is only applied when data is written. The label is associated with a given version of the cell.

HBase Shell:

```
hbase> set_visibility 'user', 'admin|service|developer', { COLUMNS => 'i' }
hbase> set_visibility 'user', 'admin|service', { COLUMNS => 'pii' }
hbase> set_visibility 'user', 'test', { COLUMNS => [ 'i', 'pii' ], FILTER =>
"(PrefixFilter ('test'))" }
```

> HBase Shell support for applying labels or permissions to cells is for testing and verification support, and should not be employed for production use because it won't apply the labels to cells that don't exist yet. The correct way to apply cell level labels is to do so in the application code when storing the values.

Java API example:

```
static Table createTableAndWriteDataWithLabels(TableName tableName, Strin
g... labelExps)
    throws Exception {
  Configuration conf = HBaseConfiguration.create();
  Connection connection = ConnectionFactory.createConnection(conf);
  Table table = NULL;
      try {
        table = TEST_UTIL.createTable(tableName, fam);
        int i = 1;
        List<Put> puts = new ArrayList<Put>();
        for (String labelExp : labelExps) {
          Put put = new Put(Bytes.toBytes("row" + i));
          put.add(fam, qual, HConstants.LATEST_TIMESTAMP, value);
          put.setCellVisibility(new CellVisibility(labelExp));
          puts.add(put);
          i++;
        }
        table.put(puts);
      } finally {
        if (table != null) {
          table.flushCommits();
        }
      }
    }
```

Reading Cells with Labels

When you issue a Scan or Get, HBase uses your default set of authorizations to filter out cells that you do not have access to. A superuser can set the default set of authorizations for a given user by using the `set_auths` HBase Shell command or the VisibilityClient.setAuths() method.

You can specify a different authorization during the Scan or Get, by passing the AUTHORIZATIONS option in HBase Shell, or the Scan.setAuthorizations() method if you use the API. This authorization will be combined with your default set as an additional filter. It will further filter your results, rather than giving you additional authorization.

HBase Shell:

```
hbase> get_auths 'myUser'
hbase> scan 'table1', AUTHORIZATIONS => ['private']
```

Java API:

```
...
public Void run() throws Exception {
  String[] auths1 = { SECRET, CONFIDENTIAL };
  GetAuthsResponse authsResponse = null;
  try {
    VisibilityClient.setAuths(conf, auths1, user);
    try {
      authsResponse = VisibilityClient.getAuths(conf, user);
    } catch (Throwable e) {
      fail("Should not have failed");
    }
  } catch (Throwable e) {
  }
  List<String> authsList = new ArrayList<String>();
  for (ByteString authBS : authsResponse.getAuthList()) {
    authsList.add(Bytes.toString(authBS.toByteArray()));
  }
  assertEquals(2, authsList.size());
  assertTrue(authsList.contains(SECRET));
  assertTrue(authsList.contains(CONFIDENTIAL));
  return null;
}
...
```

Implementing Your Own Visibility Label Algorithm

Interpreting the labels authenticated for a given get/scan request is a pluggable algorithm.

You can specify a custom plugin or plugins by using the property `hbase.regionserver.scan.visibility.label.generator.class`. The output for the first `ScanLabelGenerator` will be the input for the next one, until the end of the list.

The default implementation, which was implemented in HBASE-12466, loads two plugins, `FeedUserAuthScanLabelGenerator` and `DefinedSetFilterScanLabelGenerator`. See Reading Cells with Labels.

Replicating Visibility Tags as Strings

As mentioned in the above sections, the interface `VisibilityLabelService` could be used to implement a different way of storing the visibility expressions in the cells. Clusters with replication enabled also must replicate the visibility expressions to the peer cluster. If `DefaultVisibilityLabelServiceImpl` is used as the implementation for `VisibilityLabelService`, all the visibility expression are converted to the corresponding expression based on the ordinals for each visibility label stored in the labels table. During replication, visible cells are also replicated with the ordinal-based expression intact. The peer cluster may not have the same `labels` table with the same ordinal mapping for the visibility labels. In that case, replicating the ordinals makes no sense. It would be better if the replication occurred with the visibility expressions transmitted as strings. To replicate the visibility expression as strings to the peer cluster, create a `RegionServerObserver` configuration which works based on the implementation of the `VisibilityLabelService` interface. The configuration below enables replication of visibility expressions to peer clusters as strings. See HBASE-11639 for more details.

```
<property>
  <name>hbase.security.authorization</name>
  <value>true</value>
</property>
<property>
  <name>hbase.coprocessor.regionserver.classes</name>
  <value>org.apache.hadoop.hbase.security.visibility.VisibilityController$VisibilityReplication</value>
</property>
```

Transparent Encryption of Data At Rest

HBase provides a mechanism for protecting your data at rest, in HFiles and the WAL, which reside within HDFS or another distributed filesystem. A two-tier architecture is used for flexible and non-intrusive key rotation. "Transparent" means that no implementation changes are needed on the client side. When data is written, it is encrypted. When it is read, it is decrypted on demand.

How It Works

The administrator provisions a master key for the cluster, which is stored in a key provider accessible to every trusted HBase process, including the HMaster, RegionServers, and clients (such as HBase Shell) on administrative workstations. The default key provider is integrated with the Java KeyStore API and any key management systems with support for it. Other custom key provider implementations are possible. The key retrieval mechanism is configured in the *hbase-site.xml* configuration file. The master key may be stored on the cluster servers, protected by a secure KeyStore file, or on an external keyserver, or in a hardware security module. This master key is resolved as needed by HBase processes through the configured key provider.

Next, encryption use can be specified in the schema, per column family, by creating or modifying a column descriptor to include two additional attributes: the name of the encryption algorithm to use (currently only "AES" is supported), and optionally, a data key wrapped (encrypted) with the cluster master key. If a data key is not explicitly configured for a ColumnFamily, HBase will create a random data key per HFile. This provides an incremental improvement in security over the alternative. Unless you need to supply an explicit data key, such as in a case where you are generating encrypted HFiles for bulk import with a given data key, only specify the encryption algorithm in the ColumnFamily schema metadata and let HBase create data keys on demand. Per Column Family keys facilitate low impact incremental key rotation and reduce the scope of any external leak of key material. The wrapped data key is stored in the ColumnFamily schema metadata, and in each HFile for the Column Family, encrypted with the cluster master key. After the Column Family is configured for encryption, any new HFiles will be written encrypted. To ensure encryption of all HFiles, trigger a major compaction after enabling this feature.

When the HFile is opened, the data key is extracted from the HFile, decrypted with the cluster master key, and used for decryption of the remainder of the HFile. The HFile will be unreadable if the master key is not available. If a remote user somehow acquires access to

the HFile data because of some lapse in HDFS permissions, or from inappropriately discarded media, it will not be possible to decrypt either the data key or the file data.

It is also possible to encrypt the WAL. Even though WALs are transient, it is necessary to encrypt the WALEdits to avoid circumventing HFile protections for encrypted column families, in the event that the underlying filesystem is compromised. When WAL encryption is enabled, all WALs are encrypted, regardless of whether the relevant HFiles are encrypted.

Enable or disable the feature.

The "Transparent Encryption of Data At Rest" feature is enabled by default, meaning the users can define tables with column families where the HFiles and WAL files will be encrypted by HBase, assuming the feature is properly configured (see Server-Side Configuration).

In some cases (e.g. due to custom security policies), the operator of the HBase cluster might wish to only rely on an encryption at rest mechanism outside of HBase (e.g. those offered by HDFS) and wants to ensure that HBase's encryption at rest system is inactive. Since HBASE-25181 it is possible to explicitly disable HBase's own encryption by setting `hbase.crypto.enabled` to `false`. This configuration is `true` by default. If it is set to `false`, the users won't be able to create any table (column family) with HFile and WAL file encryption and the related create table shell (or API) commands will fail if they try.

Server-Side Configuration

This procedure assumes you are using the default Java keystore implementation. If you are using a custom implementation, check its documentation and adjust accordingly.

1. Create a secret key of appropriate length for AES encryption, using the `keytool` utility.

```
$ keytool -keystore /path/to/hbase/conf/hbase.jks \
  -storetype jceks -storepass **** \
  -genseckey -keyalg AES -keysize 128 \
  -alias <alias>
```

Replace **** with the password for the keystore file and <alias> with the username of the HBase service account, or an arbitrary string. If you use an arbitrary string,

you will need to configure HBase to use it, and that is covered below. Specify a keysize that is appropriate. Do not specify a separate password for the key, but press `Return` when prompted.

2. Set appropriate permissions on the keyfile and distribute it to all the HBase servers.

   The previous command created a file called *hbase.jks* in the HBase *conf/* directory. Set the permissions and ownership on this file such that only the HBase service account user can read the file, and securely distribute the key to all HBase servers.

3. Configure the HBase daemons.

   Set the following properties in *hbase-site.xml* on the region servers, to configure HBase daemons to use a key provider backed by the KeyStore file or retrieving the cluster master key. In the example below, replace **** with the password.

```
<property>
  <name>hbase.crypto.keyprovider</name>
  <value>org.apache.hadoop.hbase.io.crypto.KeyStoreKeyProvider</value>
</property>
<property>
  <name>hbase.crypto.keyprovider.parameters</name>
  <value>jceks:///path/to/hbase/conf/hbase.jks?password=****</value>
</property>
```

   By default, the HBase service account name will be used to resolve the cluster master key. However, you can store it with an arbitrary alias (in the `keytool` command). In that case, set the following property to the alias you used.

```
<property>
  <name>hbase.crypto.master.key.name</name>
  <value>my-alias</value>
</property>
```

   You also need to be sure your HFiles use HFile v3, in order to use transparent encryption. This is the default configuration for HBase 1.0 onward. For previous versions, set the following property in your *hbase-site.xml* file.

```
<property>
  <name>hfile.format.version</name>
  <value>3</value>
```

```
</property>
```

Optionally, you can use a different cipher provider, either a Java Cryptography Encryption (JCE) algorithm provider or a custom HBase cipher implementation.

- JCE:
  - Install a signed JCE provider (supporting `AES/CTR/NoPadding` mode with 128 bit keys)
  - Add it with highest preference to the JCE site configuration file *$JAVA_HOME/lib/security/java.security*.
  - Update `hbase.crypto.algorithm.aes.provider` and `hbase.crypto.algorithm.rng.provider` options in *hbase-site.xml*.
- Custom HBase Cipher:
  - Implement `org.apache.hadoop.hbase.io.crypto.CipherProvider`.
  - Add the implementation to the server classpath.
  - Update `hbase.crypto.cipherprovider` in *hbase-site.xml*.

4 Configure WAL encryption.

Configure WAL encryption in every RegionServer's *hbase-site.xml*, by setting the following properties. You can include these in the HMaster's *hbase-site.xml* as well, but the HMaster does not have a WAL and will not use them.

```
<property>
  <name>hbase.regionserver.hlog.reader.impl</name>
  <value>org.apache.hadoop.hbase.regionserver.wal.SecureProtobufLogReader</v
alue>
</property>
<property>
  <name>hbase.regionserver.hlog.writer.impl</name>
  <value>org.apache.hadoop.hbase.regionserver.wal.SecureProtobufLogWriter</v
alue>
</property>
<property>
  <name>hbase.regionserver.wal.encryption</name>
  <value>true</value>
</property>
```

Starting from 2.6.0, the hbase.regionserver.hlog.reader.impl and hbase.regionserver.hlog.writer.impl configurations are removed, you do not need to specify them any more. Just set hbase.regionserver.wal.encryption to true is enough to enable WAL encryption.

5 (Optional) Configure encryption key hash algorithm.

Since HBASE-25181 it is possible to use custom encryption key hash algorithm instead of the default MD5 algorithm. This hash is needed to verify the secret key during decryption. The MD5 algorithm is considered weak, and can not be used in some (e.g. FIPS compliant) clusters.

The hash is set via the configuration option `hbase.crypto.key.hash.algorithm`. It should be set to a JDK `MessageDigest` algorithm like "MD5", "SHA-384" or "SHA-512". The default is "MD5" for backward compatibility. An example of this configuration parameter on a FIPS-compliant cluster:

```
<property>
  <name>hbase.crypto.key.hash.algorithm</name>
  <value>SHA-384</value>
</property>
```

6 Configure permissions on the *hbase-site.xml* file.

Because the keystore password is stored in the hbase-site.xml, you need to ensure that only the HBase user can read the *hbase-site.xml* file, using file ownership and permissions.

7 Restart your cluster.

Distribute the new configuration file to all nodes and restart your cluster.

Administration

Administrative tasks can be performed in HBase Shell or the Java API.

Java API examples in this section are taken from the source file *hbase-server/src/test/java/org/apache/hadoop/hbase/util/TestHBaseFsckEncryption.java*. .

Neither these examples, nor the source files they are taken from, are part of the public HBase API, and are provided for illustration only. Refer to the official API for usage instructions.

Enable Encryption on a Column Family

To enable encryption on a column family, you can either use HBase Shell or the Java API. After enabling encryption, trigger a major compaction. When the major compaction completes, the compacted new HFiles will be encrypted. However, depending on the compaction settings, it is possible that not all the HFiles will be rewritten during a major compaction and there still might remain some old unencrypted HFiles. Also please note, that the snapshots are immutable. So the snapshots taken before you enabled the encryption will still contain the unencrypted HFiles.

Rotate the Data Key

To rotate the data key, first change the ColumnFamily key in the column descriptor, then trigger a major compaction. Until the compaction completes, the old HFiles will still be readable using the old key. During compaction, the compacted HFiles will be re-encrypted using the new data key. However, depending on the compaction settings, it is possible that not all the HFiles will be rewritten during a major compaction and there still might remain some old HFiles encrypted with the old key. Also please note, that the snapshots are immutable. So the snapshots taken before the changing of the encryption key will still contain the HFiles written using the old key.

Switching Between Using a Random Data Key and Specifying A Key

If you configured a column family to use a specific key and you want to return to the default behavior of using a randomly-generated key for that column family, use the Java API to alter the `HColumnDescriptor` so that no value is sent with the key `ENCRYPTION_KEY`.

Rotate the Master Key

To rotate the master key, first generate and distribute the new key. Then update the KeyStore to contain a new master key, and keep the old master key in the KeyStore using a different alias. Next, configure fallback to the old master key in the *hbase-site.xml* file.

Secure Enable

After hbase-2.x, the default 'hbase.security.authorization' changed. Before hbase-2.x, it defaulted to true, in later HBase versions, the default became false. So to enable hbase authorization, the following propertie must be configured in *hbase-site.xml*. See HBASE-19483;

```
<property>
  <name>hbase.security.authorization</name>
  <value>true</value>
</property>
```

Security Configuration Example

Example Security Settings in *hbase-site.xml*

```
<!-- HFile v3 Support -->
<property>
  <name>hfile.format.version</name>
  <value>3</value>
</property>
<!-- HBase Superuser -->
<property>
  <name>hbase.superuser</name>
  <value>hbase,admin,@superuser-group</value>
</property>
<!-- Coprocessors for ACLs and Visibility Tags -->
<property>
  <name>hbase.security.authorization</name>
  <value>true</value>
</property>
<property>
  <name>hbase.coprocessor.region.classes</name>
  <value>org.apache.hadoop.hbase.security.access.AccessController,
  org.apache.hadoop.hbase.security.visibility.VisibilityController,
  org.apache.hadoop.hbase.security.token.TokenProvider</value>
</property>
<property>
  <name>hbase.coprocessor.master.classes</name>
  <value>org.apache.hadoop.hbase.security.access.AccessController,
  org.apache.hadoop.hbase.security.visibility.VisibilityController</value>
</property>
<property>
  <name>hbase.coprocessor.regionserver.classes</name>
  <value>org.apache.hadoop.hbase.security.access.AccessController</value>
</property>
<!-- Executable ACL for Coprocessor Endpoints -->
<property>
  <name>hbase.security.exec.permission.checks</name>
  <value>true</value>
</property>
<!-- Whether a user needs authorization for a visibility tag to set it on a cell
-->
<property>
  <name>hbase.security.visibility.mutations.checkauth</name>
  <value>false</value>
</property>
<!-- Secure RPC Transport -->
```

```
<property>
  <name>hbase.rpc.protection</name>
  <value>privacy</value>
 </property>
 <!-- Transparent Encryption -->
<property>
  <name>hbase.crypto.keyprovider</name>
  <value>org.apache.hadoop.hbase.io.crypto.KeyStoreKeyProvider</value>
</property>
<property>
  <name>hbase.crypto.keyprovider.parameters</name>
  <value>jceks:///path/to/hbase/conf/hbase.jks?password=***</value>
</property>
<property>
  <name>hbase.crypto.master.key.name</name>
  <value>hbase</value>
</property>
<!-- WAL Encryption -->
<property>
  <name>hbase.regionserver.hlog.reader.impl</name>
  <value>org.apache.hadoop.hbase.regionserver.wal.SecureProtobufLogReader</value>
</property>
<property>
  <name>hbase.regionserver.hlog.writer.impl</name>
  <value>org.apache.hadoop.hbase.regionserver.wal.SecureProtobufLogWriter</value>
</property>
<property>
  <name>hbase.regionserver.wal.encryption</name>
  <value>true</value>
</property>
<!-- For key rotation -->
<property>
  <name>hbase.crypto.master.alternate.key.name</name>
  <value>hbase.old</value>
</property>
```

Starting from 2.6.0, the hbase.regionserver.hlog.reader.impl and hbase.regionserver.hlog.writer.impl configurations are removed, you do not need to specify them any more. Just set hbase.regionserver.wal.encryption to true is enough to enable WAL encryption.

Example Group Mapper in Hadoop *core-site.xml*

Adjust these settings to suit your environment.

```
<property>
  <name>hadoop.security.group.mapping</name>
  <value>org.apache.hadoop.security.LdapGroupsMapping</value>
</property>
<property>
  <name>hadoop.security.group.mapping.ldap.url</name>
  <value>ldap://server</value>
```

```
</property>
<property>
  <name>hadoop.security.group.mapping.ldap.bind.user</name>
  <value>Administrator@example-ad.local</value>
</property>
<property>
  <name>hadoop.security.group.mapping.ldap.bind.password</name>
  <value>****</value> <!-- Replace with the actual password -->
</property>
<property>
  <name>hadoop.security.group.mapping.ldap.base</name>
  <value>dc=example-ad,dc=local</value>
</property>
<property>
  <name>hadoop.security.group.mapping.ldap.search.filter.user</name>
  <value>(&amp;(objectClass=user)(sAMAccountName={0}))</value>
</property>
<property>
  <name>hadoop.security.group.mapping.ldap.search.filter.group</name>
  <value>(objectClass=group)</value>
</property>
<property>
  <name>hadoop.security.group.mapping.ldap.search.attr.member</name>
  <value>member</value>
</property>
<property>
  <name>hadoop.security.group.mapping.ldap.search.attr.group.name</name>
  <value>cn</value>
</property>
```

Architecture

Resources

1. More information about the design and implementation can be found at the jira issue: HBASE-10070

2. HBaseCon 2014 talk: HBase Read High Availability Using Timeline-Consistent Region Replicas also contains some details and slides.

Overview

NoSQL?

HBase is a type of "NoSQL" database. "NoSQL" is a general term meaning that the database isn't an RDBMS which supports SQL as its primary access language, but there are many types of NoSQL databases: BerkeleyDB is an example of a local NoSQL database, whereas HBase is very much a distributed database. Technically speaking, HBase is really more a "Data Store" than "Data Base" because it lacks many of the features you find in an RDBMS, such as typed columns, secondary indexes, triggers, and advanced query languages, etc.

However, HBase has many features which supports both linear and modular scaling. HBase clusters expand by adding RegionServers that are hosted on commodity class servers. If a cluster expands from 10 to 20 RegionServers, for example, it doubles both in terms of storage and as well as processing capacity. An RDBMS can scale well, but only up to a point - specifically, the size of a single database server - and for the best performance requires specialized hardware and storage devices. HBase features of note are:

- Strongly consistent reads/writes: HBase is not an "eventually consistent" DataStore. This makes it very suitable for tasks such as high-speed counter aggregation.
- Automatic sharding: HBase tables are distributed on the cluster via regions, and regions are automatically split and re-distributed as your data grows.
- Automatic RegionServer failover

- Hadoop/HDFS Integration: HBase supports HDFS out of the box as its distributed file system.
- MapReduce: HBase supports massively parallelized processing via MapReduce for using HBase as both source and sink.
- Java Client API: HBase supports an easy to use Java API for programmatic access.
- Thrift/REST API: HBase also supports Thrift and REST for non-Java front-ends.
- Block Cache and Bloom Filters: HBase supports a Block Cache and Bloom Filters for high volume query optimization.
- Operational Management: HBase provides build-in web-pages for operational insight as well as JMX metrics.

When Should I Use HBase?

HBase isn't suitable for every problem.

First, make sure you have enough data. If you have hundreds of millions or billions of rows, then HBase is a good candidate. If you only have a few thousand/million rows, then using a traditional RDBMS might be a better choice due to the fact that all of your data might wind up on a single node (or two) and the rest of the cluster may be sitting idle.

Second, make sure you can live without all the extra features that an RDBMS provides (e.g., typed columns, secondary indexes, transactions, advanced query languages, etc.) An application built against an RDBMS cannot be "ported" to HBase by simply changing a JDBC driver, for example. Consider moving from an RDBMS to HBase as a complete redesign as opposed to a port.

Third, make sure you have enough hardware. Even HDFS doesn't do well with anything less than 5 DataNodes (due to things such as HDFS block replication which has a default of 3), plus a NameNode.

HBase can run quite well stand-alone on a laptop - but this should be considered a development configuration only.

What Is The Difference Between HBase and Hadoop/HDFS?

HDFS is a distributed file system that is well suited for the storage of large files. Its documentation states that it is not, however, a general purpose file system, and does not provide fast individual record lookups in files. HBase, on the other hand, is built on top of HDFS and provides fast record lookups (and updates) for large tables. This can sometimes be a point of conceptual confusion. HBase internally puts your data in indexed "StoreFiles" that exist on HDFS for high-speed lookups. See the Data Model and the rest of this chapter for more information on how HBase achieves its goals.

Catalog Tables

The catalog table `hbase:meta` exists as an HBase table and is filtered out of the HBase shell's `list` command, but is in fact a table just like any other.

hbase:meta

The `hbase:meta` table (previously called `.META.`) keeps a list of all regions in the system, and the location of `hbase:meta` is stored in ZooKeeper.

The `hbase:meta` table structure is as follows:

Key:

- Region key of the format (`[table],[region start key],[region id]`)

Values:

- `info:regioninfo` (serialized RegionInfo instance for this region)
- `info:server` (server:port of the RegionServer containing this region)
- `info:serverstartcode` (start-time of the RegionServer process containing this region)

When a table is in the process of splitting, two other columns will be created, called `info:splitA` and `info:splitB`. These columns represent the two daughter regions. The values

for these columns are also serialized HRegionInfo instances. After the region has been split, eventually this row will be deleted.

> **Note on HRegionInfo**
>
> The empty key is used to denote table start and table end. A region with an empty start key is the first region in a table. If a region has both an empty start and an empty end key, it is the only region in the table

In the (hopefully unlikely) event that programmatic processing of catalog metadata is required, see the RegionInfo.parseFrom utility.

Startup Sequencing

First, the location of `hbase:meta` is looked up in ZooKeeper. Next, `hbase:meta` is updated with server and startcode values.

For information on region-RegionServer assignment, see Region-RegionServer Assignment.

Client

The HBase client finds the RegionServers that are serving the particular row range of interest. It does this by querying the `hbase:meta` table. See hbase:meta for details. After locating the required region(s), the client contacts the RegionServer serving that region, rather than going through the master, and issues the read or write request. This information is cached in the client so that subsequent requests need not go through the lookup process. Should a region be reassigned either by the master load balancer or because a RegionServer has died, the client will requery the catalog tables to determine the new location of the user region.

See Runtime Impact for more information about the impact of the Master on HBase Client communication.

Administrative functions are done via an instance of Admin

Cluster Connections

The API changed in HBase 1.0. For connection configuration information, see Client configuration and dependencies connecting to an HBase cluster.

API as of HBase 1.0.0

It's been cleaned up and users are returned Interfaces to work against rather than particular types. In HBase 1.0, obtain a `Connection` object from `ConnectionFactory` and thereafter, get from it instances of `Table`, `Admin`, and `RegionLocator` on an as-need basis. When done, close the obtained instances. Finally, be sure to cleanup your `Connection` instance before exiting. `Connections` are heavyweight objects but thread-safe so you can create one for your application and keep the instance around. `Table`, `Admin` and `RegionLocator` instances are lightweight. Create as you go and then let go as soon as you are done by closing them. See the Client Package Javadoc Description for example usage of the new HBase 1.0 API.

API before HBase 1.0.0

Instances of `HTable` are the way to interact with an HBase cluster earlier than 1.0.0. *Table instances are not thread-safe*. Only one thread can use an instance of Table at any given time. When creating Table instances, it is advisable to use the same HBaseConfiguration instance. This will ensure sharing of ZooKeeper and socket instances to the RegionServers which is usually what you want. For example, this is preferred:

```
HBaseConfiguration conf = HBaseConfiguration.create();
HTable table1 = new HTable(conf, "myTable");
HTable table2 = new HTable(conf, "myTable");
```

as opposed to this:

```
HBaseConfiguration conf1 = HBaseConfiguration.create();
HTable table1 = new HTable(conf1, "myTable");
HBaseConfiguration conf2 = HBaseConfiguration.create();
HTable table2 = new HTable(conf2, "myTable");
```

For more information about how connections are handled in the HBase client, see ConnectionFactory.

Connection Pooling

For applications which require high-end multithreaded access (e.g., web-servers or application servers that may serve many application threads in a single JVM), you can pre-create a `Connection`, as shown in the following example:

Example 24. Pre-Creating a `Connection`

```
// Create a connection to the cluster.
Configuration conf = HBaseConfiguration.create();
try (Connection connection = ConnectionFactory.createConnection(conf);
     Table table = connection.getTable(TableName.valueOf(tablename))) {
  // use table as needed, the table returned is lightweight
}
```

> **HTablePool is Deprecated**
>
> Previous versions of this guide discussed `HTablePool`, which was deprecated in HBase 0.94, 0.95, and 0.96, and removed in 0.98.1, by HBASE-6580, or `HConnection`, which is deprecated in HBase 1.0 by `Connection`. Please use Connection instead.

WriteBuffer and Batch Methods

In HBase 1.0 and later, HTable is deprecated in favor of Table. `Table` does not use autoflush. To do buffered writes, use the BufferedMutator class.

In HBase 2.0 and later, HTable does not use BufferedMutator to execute the `Put` operation. Refer to HBASE-18500 for more information.

For additional information on write durability, review the ACID semantics page.

For fine-grained control of batching of `Put`s or `Delete`s, see the batch methods on Table.

Asynchronous Client

It is a new API introduced in HBase 2.0 which aims to provide the ability to access HBase asynchronously.

You can obtain an `AsyncConnection` from `ConnectionFactory`, and then get a asynchronous table instance from it to access HBase. When done, close the `AsyncConnection` instance(usually when your program exits).

For the asynchronous table, most methods have the same meaning with the old `Table` interface, expect that the return value is wrapped with a CompletableFuture usually. We do not have any buffer here so there is no close method for asynchronous table, you do not need to close it. And it is thread safe.

There are several differences for scan:

- There is still a `getScanner` method which returns a `ResultScanner`. You can use it in the old way and it works like the old `ClientAsyncPrefetchScanner`.
- There is a `scanAll` method which will return all the results at once. It aims to provide a simpler way for small scans which you want to get the whole results at once usually.
- The Observer Pattern. There is a scan method which accepts a `ScanResultConsumer` as a parameter. It will pass the results to the consumer.

Notice that `AsyncTable` interface is templatized. The template parameter specifies the type of `ScanResultConsumerBase` used by scans, which means the observer style scan APIs are different. The two types of scan consumers are - `ScanResultConsumer` and `AdvancedScanResultConsumer`.

`ScanResultConsumer` needs a separate thread pool which is used to execute the callbacks registered to the returned CompletableFuture. Because the use of separate thread pool frees up RPC threads, callbacks are free to do anything. Use this if the callbacks are not quick, or when in doubt.

`AdvancedScanResultConsumer` executes callbacks inside the framework thread. It is not allowed to do time consuming work in the callbacks else it will likely block the framework threads and cause very bad performance impact. As its name, it is designed for advanced users who want to write high performance code. See `org.apache.hadoop.hbase.client.example.HttpProxyExample` for how to write fully asynchronous code with it.

Asynchronous Admin

You can obtain an `AsyncConnection` from `ConnectionFactory`, and then get a `AsyncAdmin` instance from it to access HBase. Notice that there are two `getAdmin` methods to get a `AsyncAdmin` instance. One method has one extra thread pool parameter which is used to execute callbacks. It is designed for normal users. Another method doesn't need a thread pool and all the callbacks are executed inside the framework thread so it is not allowed to do time consuming works in the callbacks. It is designed for advanced users.

The default `getAdmin` methods will return a `AsyncAdmin` instance which use default configs. If you want to customize some configs, you can use `getAdminBuilder` methods to get a `AsyncAdminBuilder` for creating `AsyncAdmin` instance. Users are free to only set the configs they care about to create a new `AsyncAdmin` instance.

For the `AsyncAdmin` interface, most methods have the same meaning with the old `Admin` interface, expect that the return value is wrapped with a CompletableFuture usually.

For most admin operations, when the returned CompletableFuture is done, it means the admin operation has also been done. But for compact operation, it only means the compact request was sent to HBase and may need some time to finish the compact operation. For `rollWALWriter` method, it only means the rollWALWriter request was sent to the region server and may need some time to finish the `rollWALWriter` operation.

For region name, we only accept `byte[]` as the parameter type and it may be a full region name or a encoded region name. For server name, we only accept `ServerName` as the parameter type. For table name, we only accept `TableName` as the parameter type. For `list*` operations, we only accept `Pattern` as the parameter type if you want to do regex matching.

External Clients

Information on non-Java clients and custom protocols is covered in Apache HBase External APIs

Master Registry (new as of 2.3.0)

Starting from 2.5.0, MasterRegistry is deprecated. It's functionality is completely superseded by the RpcConnectionRegistry. Please see Rpc Connection Registry (new as of 2.5.0) for more details.

Client internally works with a *connection registry* to fetch the metadata needed by connections. This connection registry implementation is responsible for fetching the following metadata.

- Active master address
- Current meta region(s) locations
- Cluster ID (unique to this cluster)

This information is needed as a part of various client operations like connection set up, scans, gets, etc. Traditionally, the connection registry implementation has been based on ZooKeeper as the source of truth and clients fetched the metadata directly from the ZooKeeper quorum. HBase 2.3.0 introduces a new connection registry implementation based on direct communication with the Masters. With this implementation, clients now fetch required metadata via master RPC end points instead of maintaining connections to ZooKeeper. This change was done for the following reasons.

- Reduce load on ZooKeeper since that is critical for cluster operation.
- Holistic client timeout and retry configurations since the new registry brings all the client operations under HBase rpc framework.
- Remove the ZooKeeper client dependency on HBase client library.

This means:

- At least a single active or stand by master is needed for cluster connection setup. Refer to Runtime Impact for more details.
- Master can be in a critical path of read/write operations, especially if the client metadata cache is empty or stale.
- There is higher connection load on the masters that before since the clients talk directly to HMasters instead of ZooKeeper ensemble`

To reduce hot-spotting on a single master, all the masters (active & stand-by) expose the needed service to fetch the connection metadata. This lets the client connect to any master (not just active). Both ZooKeeper-based and Master-based connection registry implementations are available in 2.3+. For 2.x and earlier, the ZooKeeper-based implementation remains the default configuration. For 3.0.0, RpcConnectionRegistry becomes the default configuration, as the alternate to MasterRegistry.

Change the connection registry implementation by updating the value configured for `hbase.client.registry.impl`. To explicitly enable the ZooKeeper-based registry, use

```
<property>
  <name>hbase.client.registry.impl</name>
  <value>org.apache.hadoop.hbase.client.ZKConnectionRegistry</value>
</property>
```

To explicitly enable the Master-based registry, use

```
<property>
  <name>hbase.client.registry.impl</name>
  <value>org.apache.hadoop.hbase.client.MasterRegistry</value>
</property>
```

MasterRegistry RPC hedging

MasterRegistry implements hedging of connection registry RPCs across active and stand-by masters. This lets the client make the same request to multiple servers and which ever responds first is returned back to the client immediately. This improves performance, especially when a subset of servers are under load. The hedging fan out size is configurable, meaning the number of requests that are hedged in a single attempt, using the configuration key *hbase.client.master_registry.hedged.fanout* in the client configuration. It defaults to 2. With this default, the RPCs are tried in batches of 2. The hedging policy is still primitive and does not adapt to any sort of live rpc performance metrics.

Additional Notes

- Clients hedge the requests in a randomized order to avoid hot-spotting a single master.
- Cluster internal connections (masters ↔ regionservers) still use ZooKeeper based connection registry.

- Cluster internal state is still tracked in Zookeeper, hence ZK availability requirements are same as before.
- Inter cluster replication still uses ZooKeeper based connection registry to simplify configuration management.

For more implementation details, please refer to the design doc and HBASE-18095.

Rpc Connection Registry (new as of 2.5.0)

As said in the Master Registry (new as of 2.3.0) section, there are some disadvantages and limitations for MasterRegistry, especially that it puts master in the critical path of read/write operations. In order to address these problems, we introduced a more generic RpcConnectionRegistry.

It is also rpc based, like MasterRegistry, with several differences

1. Region server also implements the necessary rpc service, so you can config any nodes in the cluster as bootstrap nodes, not only masters
2. Support refreshing bootstrap nodes, for spreading loads across the nodes in the cluster, and also remove the dead nodes in bootstrap nodes.

To explicitly enable the rpc-based registry, use

```
<property>
  <name>hbase.client.registry.impl</name>
  <value>org.apache.hadoop.hbase.client.RpcConnectionRegistry</value>
</property>
```

To configure the bootstrap nodes, use

```
<property>
  <name>hbase.client.bootstrap.servers</name>
  <value>server1:16020,server2:16020,server3:16020</value>
</property>
```

If not configured, we will fallback to use master addresses as the bootstrap nodes.

RpcConnectionRegistry is available in 2.5+, and becomes the default client registry implementation in 3.0.0.

RpcConnectionRegistry RPC hedging

Hedged read is still supported, the configuration key is now *hbase.client.bootstrap.hedged.fanout*, and its default value is still 2.

RpcConnectionRegistry bootstrap nodes refreshing

There are basically two reasons for us to refresh the bootstrap nodes

- Periodically. This is for spreading loads across the nodes in the cluster. There are two configurations

  1. *hbase.client.bootstrap.refresh_interval_secs*: the refresh interval in seconds, default 300. A value less than or equal to zero means disable refreshing.
  2. *hbase.client.bootstrap.initial_refresh_delay_secs*: the initial refresh interval in seconds, the default value is 1/10 of *hbase.client.bootstrap.refresh_interval_secs*. The reason why we want to introduce a separated configuration for the delay for first refreshing is that, as end users could configure any nodes in a cluster as the initial bootstrap nodes, it is possible that different end users will configure the same machine which makes the machine over load. So we should have a shorter delay for the initial refresh, to let users quickly switch to the bootstrap nodes we want them to connect to.

- When there is a connection error while requesting the nodes, we will refresh immediately, to remove the dead nodes. To avoid putting too much pressure to the cluster, there is a configuration *hbase.client.bootstrap.min_secs_between_refreshes*, to control the minimum interval between two refreshings. The default value is 60, but notice that, if you change *hbase.client.bootstrap.refresh_interval_secs* to a small value, you need to make sure to also change *hbase.client.bootstrap.min_secs_between_refreshes* to a value smaller than *hbase.client.bootstrap.refresh_interval_secs*, otherwise an IllegalArgumentException will be thrown.

> (Advanced) In case of any issues with the rpc/master based registry, use the following configuration to fallback to the ZooKeeper based connection registry implementation.

```
<property>
  <name>hbase.client.registry.impl</name>
  <value>org.apache.hadoop.hbase.client.ZKConnectionRegistry</value>
</property>
```

Connection URI

Starting from 2.7.0, we add the support for specifying the connection information for a HBase cluster through an URI, which we call a "connection URI". And we've added several methods in *ConnectionFactory* to let you get a connection to the cluster specified by the URI. It looks like:

```
URI uri = new URI("hbase+rpc://server1:16020,server2:16020,server3:16020");
try (Connection conn = ConnectionFactory.createConnection(uri)) {
  ...
}
```

Supported Schemes

Currently there are two schemes supported, *hbase+rpc* for *RpcConnectionRegistry* and *hbase+zk* for *ZKConnectionRegistry*. *MasterRegistry* is deprecated so we do not expose it through connection URI.

For *hbase+rpc*, it looks like

```
hbase+rpc://server1:16020,server2:16020,server3:16020
```

The authority part *server1:16020,server2:16020,server3:16020* specifies the bootstrap nodes and their rpc ports, i.e, the configuration value for *hbase.client.bootstrap.servers* in the past.

For *hbase+zk*, it looks like

```
hbase+zk://zk1:2181,zk2:2181,zk3:2181/hbase
```

The authority part *zk1:2181,zk2:2181,zk3:2181* is the zk quorum, i.e, the configuration value for *hbase.zookeeper.quorum* in the past. The path part */hbase* is the znode parent, i.e, the configuration value for *zookeeper.znode.parent* in the past.

Specify Configuration through URI Queries

To let users fully specify the connection information through a connection URI, we support specifying configuration values through URI Queries. It looks like:

```
hbase+rpc://server1:16020?hbase.client.operation.timeout=10000
```

In this way you can set the operation timeout to 10 seconds. Notice that, the configuration values specified in the connection URI will override the ones in the configuration file.

Implement Your Own Connection Registry

We use *ServiceLoader* to load different connection registry implementations, the entry point is *org.apache.hadoop.hbase.client.ConnectionRegistryURIFactory*. So if you implement your own *ConnectionRegistryURIFactory* which has a different scheme, and register it in the services file, we can load it at runtime.

Connection URI is still a very new feature which has not been used extensively in production, so we do not want to expose the ability to customize *ConnectionRegistryURIFactory* yet as the API may be changed frequently in the beginning.

If you really want to implement your own connection registry, you can use the above way but take your own risk.

Client Request Filters

Get and Scan instances can be optionally configured with filters which are applied on the RegionServer.

Filters can be confusing because there are many different types, and it is best to approach them by understanding the groups of Filter functionality.

Structural

Structural Filters contain other Filters.

FilterList

FilterList represents a list of Filters with a relationship of `FilterList.Operator.MUST_PASS_ALL` or `FilterList.Operator.MUST_PASS_ONE` between the Filters. The following example shows

an 'or' between two Filters (checking for either 'my value' or 'my other value' on the same attribute).

```
FilterList list = new FilterList(FilterList.Operator.MUST_PASS_ONE);
SingleColumnValueFilter filter1 = new SingleColumnValueFilter(
  cf,
  column,
  CompareOperator.EQUAL,
  Bytes.toBytes("my value")
  );
list.add(filter1);
SingleColumnValueFilter filter2 = new SingleColumnValueFilter(
  cf,
  column,
  CompareOperator.EQUAL,
  Bytes.toBytes("my other value")
  );
list.add(filter2);
scan.setFilter(list);
```

Column Value

SingleColumnValueFilter

A SingleColumnValueFilter (see: https://hbase.apache.org/devapidocs/org/apache/hadoop/hbase/filter/SingleColumnValueFilter.html) can be used to test column values for equivalence (`CompareOperaor.EQUAL`), inequality (`CompareOperaor.NOT_EQUAL`), or ranges (e.g., `CompareOperaor.GREATER`). The following is an example of testing equivalence of a column to a String value "my value"...

```
SingleColumnValueFilter filter = new SingleColumnValueFilter(
  cf,
  column,
  CompareOperaor.EQUAL,
  Bytes.toBytes("my value")
  );
scan.setFilter(filter);
```

ColumnValueFilter

Introduced in HBase-2.0.0 version as a complementation of SingleColumnValueFilter, ColumnValueFilter gets matched cell only, while SingleColumnValueFilter gets the entire

row (has other columns and values) to which the matched cell belongs. Parameters of constructor of ColumnValueFilter are the same as SingleColumnValueFilter.

```
ColumnValueFilter filter = new ColumnValueFilter(
  cf,
  column,
  CompareOperaor.EQUAL,
  Bytes.toBytes("my value")
  );
scan.setFilter(filter);
```

Note. For simple query like "equals to a family:qualifier:value", we highly recommend to use the following way instead of using SingleColumnValueFilter or ColumnValueFilter:

```
Scan scan = new Scan();
scan.addColumn(Bytes.toBytes("family"), Bytes.toBytes("qualifier"));
ValueFilter vf = new ValueFilter(CompareOperator.EQUAL,
  new BinaryComparator(Bytes.toBytes("value")));
scan.setFilter(vf);
...
```

This scan will restrict to the specified column 'family:qualifier', avoiding scan of unrelated families and columns, which has better performance, and `ValueFilter` is the condition used to do the value filtering.

But if query is much more complicated beyond this book, then please make your good choice case by case.

Column Value Comparators

There are several Comparator classes in the Filter package that deserve special mention. These Comparators are used in concert with other Filters, such as SingleColumnValueFilter.

RegexStringComparator

RegexStringComparator supports regular expressions for value comparisons.

```
RegexStringComparator comp = new RegexStringComparator("my.");   // any value tha
t starts with 'my'
SingleColumnValueFilter filter = new SingleColumnValueFilter(
```

```
  cf,
  column,
  CompareOperaor.EQUAL,
  comp
  );
scan.setFilter(filter);
```

See the Oracle JavaDoc for supported RegEx patterns in Java.

SubstringComparator

SubstringComparator can be used to determine if a given substring exists in a value. The comparison is case-insensitive.

```
SubstringComparator comp = new SubstringComparator("y val");   // looking for 'my
value'
SingleColumnValueFilter filter = new SingleColumnValueFilter(
  cf,
  column,
  CompareOperaor.EQUAL,
  comp
  );
scan.setFilter(filter);
```

BinaryPrefixComparator

See BinaryPrefixComparator.

BinaryComparator

See BinaryComparator.

BinaryComponentComparator

BinaryComponentComparator can be used to compare specific value at specific location with in the cell value. The comparison can be done for both ascii and binary data.

```
byte[] partialValue = Bytes.toBytes("partial_value");
int partialValueOffset = 0;
Filter partialValueFilter = new ValueFilter(CompareFilter.CompareOp.GREATER,
        new BinaryComponentComparator(partialValue,partialValueOffset));
```

See HBASE-22969 for other use cases and details.

KeyValue Metadata

As HBase stores data internally as KeyValue pairs, KeyValue Metadata Filters evaluate the existence of keys (i.e., ColumnFamily:Column qualifiers) for a row, as opposed to values the previous section.

FamilyFilter

FamilyFilter can be used to filter on the ColumnFamily. It is generally a better idea to select ColumnFamilies in the Scan than to do it with a Filter.

QualifierFilter

QualifierFilter can be used to filter based on Column (aka Qualifier) name.

ColumnPrefixFilter

ColumnPrefixFilter can be used to filter based on the lead portion of Column (aka Qualifier) names.

A ColumnPrefixFilter seeks ahead to the first column matching the prefix in each row and for each involved column family. It can be used to efficiently get a subset of the columns in very wide rows.

Note: The same column qualifier can be used in different column families. This filter returns all matching columns.

Example: Find all columns in a row and family that start with "abc"

```
Table t = ...;
byte[] row = ...;
byte[] family = ...;
byte[] prefix = Bytes.toBytes("abc");
Scan scan = new Scan(row, row); // (optional) limit to one row
scan.addFamily(family); // (optional) limit to one family
Filter f = new ColumnPrefixFilter(prefix);
scan.setFilter(f);
scan.setBatch(10); // set this if there could be many columns returned
ResultScanner rs = t.getScanner(scan);
for (Result r = rs.next(); r != null; r = rs.next()) {
  for (Cell cell : result.listCells()) {
    // each cell represents a column
  }
```

```
}
rs.close();
```

MultipleColumnPrefixFilter

MultipleColumnPrefixFilter behaves like ColumnPrefixFilter but allows specifying multiple prefixes.

Like ColumnPrefixFilter, MultipleColumnPrefixFilter efficiently seeks ahead to the first column matching the lowest prefix and also seeks past ranges of columns between prefixes. It can be used to efficiently get discontinuous sets of columns from very wide rows.

Example: Find all columns in a row and family that start with "abc" or "xyz"

```
Table t = ...;
byte[] row = ...;
byte[] family = ...;
byte[][] prefixes = new byte[][] {Bytes.toBytes("abc"), Bytes.toBytes("xyz")};
Scan scan = new Scan(row, row); // (optional) limit to one row
scan.addFamily(family); // (optional) limit to one family
Filter f = new MultipleColumnPrefixFilter(prefixes);
scan.setFilter(f);
scan.setBatch(10); // set this if there could be many columns returned
ResultScanner rs = t.getScanner(scan);
for (Result r = rs.next(); r != null; r = rs.next()) {
  for (Cell cell : result.listCells()) {
    // each cell represents a column
  }
}
rs.close();
```

ColumnRangeFilter

A ColumnRangeFilter allows efficient intra row scanning.

A ColumnRangeFilter can seek ahead to the first matching column for each involved column family. It can be used to efficiently get a 'slice' of the columns of a very wide row. i.e. you have a million columns in a row but you only want to look at columns bbbb-bbdd.

Note: The same column qualifier can be used in different column families. This filter returns all matching columns.

Example: Find all columns in a row and family between "bbbb" (inclusive) and "bbdd" (inclusive)

```
Table t = ...;
byte[] row = ...;
byte[] family = ...;
byte[] startColumn = Bytes.toBytes("bbbb");
byte[] endColumn = Bytes.toBytes("bbdd");
Scan scan = new Scan(row, row); // (optional) limit to one row
scan.addFamily(family); // (optional) limit to one family
Filter f = new ColumnRangeFilter(startColumn, true, endColumn, true);
scan.setFilter(f);
scan.setBatch(10); // set this if there could be many columns returned
ResultScanner rs = t.getScanner(scan);
for (Result r = rs.next(); r != null; r = rs.next()) {
  for (Cell cell : result.listCells()) {
    // each cell represents a column
  }
}
rs.close();
```

Note: Introduced in HBase 0.92

RowKey

RowFilter

It is generally a better idea to use the startRow/stopRow methods on Scan for row selection, however RowFilter can also be used.

You can supplement a scan (both bounded and unbounded) with RowFilter constructed from BinaryComponentComparator for further filtering out or filtering in rows. See HBASE-22969 for use cases and other details.

Utility

FirstKeyOnlyFilter

This is primarily used for rowcount jobs. See FirstKeyOnlyFilter.

Master

`HMaster` is the implementation of the Master Server. The Master server is responsible for monitoring all RegionServer instances in the cluster, and is the interface for all metadata changes. In a distributed cluster, the Master typically runs on the NameNode. J Mohamed Zahoor goes into some more detail on the Master Architecture in this blog posting, HBase HMaster Architecture.

Startup Behavior

If run in a multi-Master environment, all Masters compete to run the cluster. If the active Master loses its lease in ZooKeeper (or the Master shuts down), then the remaining Masters jostle to take over the Master role.

Runtime Impact

A common dist-list question involves what happens to an HBase cluster when the Master goes down. This information has changed starting 3.0.0.

Up until releases 2.x.y

Because the HBase client talks directly to the RegionServers, the cluster can still function in a "steady state". Additionally, per Catalog Tables, `hbase:meta` exists as an HBase table and is not resident in the Master. However, the Master controls critical functions such as RegionServer failover and completing region splits. So while the cluster can still run for a short time without the Master, the Master should be restarted as soon as possible.

Staring release 3.0.0

As mentioned in section Master Registry (new as of 2.3.0), the default connection registry for clients is now based on master rpc end points. Hence the requirements for masters' uptime are even tighter starting this release.

- At least one active or stand by master is needed for a connection set up, unlike before when all the clients needed was a ZooKeeper ensemble.

- Master is now in critical path for read/write operations. For example, if the meta region bounces off to a different region server, clients need master to fetch the new locations. Earlier this was done by fetching this information directly from ZooKeeper.
- Masters will now have higher connection load than before. So, the server side configuration might need adjustment depending on the load.

Overall, the master uptime requirements, when this feature is enabled, are even higher for the client operations to go through.

Interface

The methods exposed by `HMasterInterface` are primarily metadata-oriented methods:

- Table (createTable, modifyTable, removeTable, enable, disable)
- ColumnFamily (addColumn, modifyColumn, removeColumn)
- Region (move, assign, unassign) For example, when the `Admin` method `disableTable` is invoked, it is serviced by the Master server.

Processes

The Master runs several background threads:

LoadBalancer

Periodically, and when there are no regions in transition, a load balancer will run and move regions around to balance the cluster's load. See Balancer for configuring this property.

See Region-RegionServer Assignment for more information on region assignment.

CatalogJanitor

Periodically checks and cleans up the `hbase:meta` table. See hbase:meta for more information on the meta table.

MasterProcWAL

*MasterProcWAL is replaced in hbase-2.3.0 by an alternate Procedure Store implementation; see **in-master-procedure-store-region**. This section pertains to hbase-2.0.0 through hbase-2.2.x*

HMaster records administrative operations and their running states, such as the handling of a crashed server, table creation, and other DDLs, into a Procedure Store. The Procedure Store WALs are stored under the MasterProcWALs directory. The Master WALs are not like RegionServer WALs. Keeping up the Master WAL allows us to run a state machine that is resilient across Master failures. For example, if a HMaster was in the middle of creating a table encounters an issue and fails, the next active HMaster can take up where the previous left off and carry the operation to completion. Since hbase-2.0.0, a new AssignmentManager (A.K.A AMv2) was introduced and the HMaster handles region assignment operations, server crash processing, balancing, etc., all via AMv2 persisting all state and transitions into MasterProcWALs rather than up into ZooKeeper, as we do in hbase-1.x.

See AMv2 Description for Devs (and Procedure Framework (Pv2): HBASE-12439 for its basis) if you would like to learn more about the new AssignmentManager.

Configurations for MasterProcWAL

Here are the list of configurations that effect MasterProcWAL operation. You should not have to change your defaults.

- `hbase.procedure.store.wal.periodic.roll.msec`
  Description: Frequency of generating a new WAL
  Default: `1h (3600000 in msec)`

- `hbase.procedure.store.wal.roll.threshold`
  Description: Threshold in size before the WAL rolls. Every time the WAL reaches this size or the above period, 1 hour, passes since last log roll, the HMaster will generate a new WAL.
  Default: `32MB (33554432 in byte)`

- `hbase.procedure.store.wal.warn.threshold`
  Description: If the number of WALs goes beyond this threshold, the following message

should appear in the HMaster log with WARN level when rolling.

```
procedure WALs count=xx above the warning threshold 64. check running procedure
s to see if something is stuck.
```

Default: `64`

- `hbase.procedure.store.wal.max.retries.before.roll`

  Description: Max number of retry when syncing slots (records) to its underlying storage, such as HDFS. Every attempt, the following message should appear in the HMaster log.

  ```
  unable to sync slots, retry=xx
  ```

  Default: `3`

- `hbase.procedure.store.wal.sync.failure.roll.max`

  Description: After the above 3 retrials, the log is rolled and the retry count is reset to 0, thereon a new set of retrial starts. This configuration controls the max number of attempts of log rolling upon sync failure. That is, HMaster is allowed to fail to sync 9 times in total. Once it exceeds, the following log should appear in the HMaster log.

  ```
  Sync slots after log roll failed, abort.
  ```

  Default: `3`

RegionServer

`HRegionServer` is the RegionServer implementation. It is responsible for serving and managing regions. In a distributed cluster, a RegionServer runs on a DataNode.

Interface

The methods exposed by `HRegionRegionInterface` contain both data-oriented and region-maintenance methods:

- Data (get, put, delete, next, etc.)
- Region (splitRegion, compactRegion, etc.) For example, when the `Admin` method `majorCompact` is invoked on a table, the client is actually iterating through all regions for the specified table and requesting a major compaction directly to each region.

Processes

The RegionServer runs a variety of background threads:

CompactSplitThread

Checks for splits and handle minor compactions.

MajorCompactionChecker

Checks for major compactions.

MemStoreFlusher

Periodically flushes in-memory writes in the MemStore to StoreFiles.

LogRoller

Periodically checks the RegionServer's WAL.

Coprocessors

Coprocessors were added in 0.92. There is a thorough Blog Overview of CoProcessors posted. Documentation will eventually move to this reference guide, but the blog is the most current information available at this time.

Block Cache

HBase provides two different BlockCache implementations to cache data read from HDFS: the default on-heap `LruBlockCache` and the `BucketCache`, which is (usually) off-heap. This

section discusses benefits and drawbacks of each implementation, how to choose the appropriate option, and configuration options for each.

> **Block Cache Reporting: UI**
>
> See the RegionServer UI for detail on caching deploy. See configurations, sizings, current usage, time-in-the-cache, and even detail on block counts and types.

Cache Choices

`LruBlockCache` is the original implementation, and is entirely within the Java heap. `BucketCache` is optional and mainly intended for keeping block cache data off-heap, although `BucketCache` can also be a file-backed cache. In file-backed we can either use it in the file mode or the mmaped mode. We also have pmem mode where the bucket cache resides on the persistent memory device.

When you enable BucketCache, you are enabling a two tier caching system. We used to describe the tiers as "L1" and "L2" but have deprecated this terminology as of hbase-2.0.0. The "L1" cache referred to an instance of LruBlockCache and "L2" to an off-heap BucketCache. Instead, when BucketCache is enabled, all DATA blocks are kept in the BucketCache tier and meta blocks — INDEX and BLOOM blocks — are on-heap in the `LruBlockCache`. Management of these two tiers and the policy that dictates how blocks move between them is done by `CombinedBlockCache`.

General Cache Configurations

Apart from the cache implementation itself, you can set some general configuration options to control how the cache performs. See CacheConfig. After setting any of these options, restart or rolling restart your cluster for the configuration to take effect. Check logs for errors or unexpected behavior.

See also Prefetch Option for Blockcache, which discusses a new option introduced in HBASE-9857.

LruBlockCache Design

The LruBlockCache is an LRU cache that contains three levels of block priority to allow for scan-resistance and in-memory ColumnFamilies:

- Single access priority: The first time a block is loaded from HDFS it normally has this priority and it will be part of the first group to be considered during evictions. The advantage is that scanned blocks are more likely to get evicted than blocks that are getting more usage.
- Multi access priority: If a block in the previous priority group is accessed again, it upgrades to this priority. It is thus part of the second group considered during evictions.
- In-memory access priority: If the block's family was configured to be "in-memory", it will be part of this priority disregarding the number of times it was accessed. Catalog tables are configured like this. This group is the last one considered during evictions.
  To mark a column family as in-memory, call

  ```
  HColumnDescriptor.setInMemory(true);
  ```

  if creating a table from java, or set `IN_MEMORY ⇒ true` when creating or altering a table in the shell: e.g.

  ```
  hbase(main):003:0> create  't', {NAME => 'f', IN_MEMORY => 'true'}
  ```

  For more information, see the LruBlockCache source

LruBlockCache Usage

Block caching is enabled by default for all the user tables which means that any read operation will load the LRU cache. This might be good for a large number of use cases, but further tunings are usually required in order to achieve better performance. An important concept is the working set size, or WSS, which is: "the amount of memory needed to compute the answer to a problem". For a website, this would be the data that's needed to answer the queries over a short amount of time.

The way to calculate how much memory is available in HBase for caching is:

```
number of region servers * heap size * hfile.block.cache.size * 0.99
```

The default value for the block cache is 0.4 which represents 40% of the available heap. The last value (99%) is the default acceptable loading factor in the LRU cache after which eviction is started. The reason it is included in this equation is that it would be unrealistic to

say that it is possible to use 100% of the available memory since this would make the process blocking from the point where it loads new blocks. Here are some examples:

- One region server with the heap size set to 1 GB and the default block cache size will have 405 MB of block cache available.
- 20 region servers with the heap size set to 8 GB and a default block cache size will have 63.3 GB of block cache.
- 100 region servers with the heap size set to 24 GB and a block cache size of 0.5 will have about 1.16 TB of block cache.

Your data is not the only resident of the block cache. Here are others that you may have to take into account:

- Catalog Tables
  The `hbase:meta` table is forced into the block cache and have the in-memory priority which means that they are harder to evict.

  > The hbase:meta tables can occupy a few MBs depending on the number of regions.

- HFiles Indexes
  An *HFile* is the file format that HBase uses to store data in HDFS. It contains a multi-layered index which allows HBase to seek the data without having to read the whole file. The size of those indexes is a factor of the block size (64KB by default), the size of your keys and the amount of data you are storing. For big data sets it's not unusual to see numbers around 1GB per region server, although not all of it will be in cache because the LRU will evict indexes that aren't used.

- Keys
  The values that are stored are only half the picture, since each value is stored along with its keys (row key, family qualifier, and timestamp). See Try to minimize row and column sizes.

- Bloom Filters
  Just like the HFile indexes, those data structures (when enabled) are stored in the LRU.

Currently the recommended way to measure HFile indexes and bloom filters sizes is to look at the region server web UI and checkout the relevant metrics. For keys, sampling can be done by using the HFile command line tool and look for the average key size metric.

Since HBase 0.98.3, you can view details on BlockCache stats and metrics in a special Block Cache section in the UI. As of HBase 2.4.14, you can estimate HFile indexes and bloom filters vs other DATA blocks using blockCacheCount and blockCacheDataBlockCount in JMX. The formula `(blockCacheCount - blockCacheDataBlockCount) * blockSize` will give you an estimate which can be useful when trying to enable the BucketCache. You should make sure the post-BucketCache config gives enough memory to the on-heap LRU cache to hold at least the same number of non-DATA blocks from pre-BucketCache. Once BucketCache is enabled, the L1 metrics like l1CacheSize, l1CacheCount, and l1CacheEvictionCount can help you further tune the size.

It's generally bad to use block caching when the WSS doesn't fit in memory. This is the case when you have for example 40GB available across all your region servers' block caches but you need to process 1TB of data. One of the reasons is that the churn generated by the evictions will trigger more garbage collections unnecessarily. Here are two use cases:

- Fully random reading pattern: This is a case where you almost never access the same row twice within a short amount of time such that the chance of hitting a cached block is close to 0. Setting block caching on such a table is a waste of memory and CPU cycles, more so that it will generate more garbage to pick up by the JVM. For more information on monitoring GC, see JVM Garbage Collection Logs.
- Mapping a table: In a typical MapReduce job that takes a table in input, every row will be read only once so there's no need to put them into the block cache. The Scan object has the option of turning this off via the setCacheBlocks method (set it to false). You can still keep block caching turned on on this table if you need fast random read access. An example would be counting the number of rows in a table that serves live traffic, caching every block of that table would create massive churn and would surely evict data that's currently in use.

Caching META blocks only (DATA blocks in fscache)

An interesting setup is one where we cache META blocks only and we read DATA blocks in on each access. If the DATA blocks fit inside fscache, this alternative may make sense when access is completely random across a very large dataset. To enable this setup, alter your table and for each column family set `BLOCKCACHE ⇒ 'false'`. You are 'disabling' the BlockCache for this column family only. You can never disable the caching of META blocks. Since HBASE-4683 Always cache index and bloom blocks, we will cache META blocks even if the BlockCache is disabled.

Off-heap Block Cache

How to Enable BucketCache

The usual deployment of BucketCache is via a managing class that sets up two caching tiers: an on-heap cache implemented by LruBlockCache and a second cache implemented with BucketCache. The managing class is CombinedBlockCache by default. The previous link describes the caching 'policy' implemented by CombinedBlockCache. In short, it works by keeping meta blocks — INDEX and BLOOM in the on-heap LruBlockCache tier — and DATA blocks are kept in the BucketCache tier.

- Pre-hbase-2.0.0 versions

  Fetching will always be slower when fetching from BucketCache in pre-hbase-2.0.0, as compared to the native on-heap LruBlockCache. However, latencies tend to be less erratic across time, because there is less garbage collection when you use BucketCache since it is managing BlockCache allocations, not the GC. If the BucketCache is deployed in off-heap mode, this memory is not managed by the GC at all. This is why you'd use BucketCache in pre-2.0.0, so your latencies are less erratic, to mitigate GCs and heap fragmentation, and so you can safely use more memory. See Nick Dimiduk's BlockCache 101 for comparisons running on-heap vs off-heap tests. Also see Comparing BlockCache Deploys which finds that if your dataset fits inside your LruBlockCache deploy, use it otherwise if you are experiencing cache churn (or you want your cache to exist beyond the vagaries of java GC), use BucketCache.

  In pre-2.0.0, one can configure the BucketCache so it receives the `victim` of an LruBlockCache eviction. All Data and index blocks are cached in L1 first. When eviction happens from L1, the blocks (or `victims`) will get moved to L2. Set `cacheDataInL1` via `(HColumnDescriptor.setCacheDataInL1(true)` or in the shell, creating or amending column families setting `CACHE_DATA_IN_L1` to true: e.g.

  ```
  hbase(main):003:0> create 't', {NAME => 't', CONFIGURATION => {CACHE_DATA_IN_L1
  => 'true'}}
  ```

- hbase-2.0.0+ versions

  HBASE-11425 changed the HBase read path so it could hold the read-data off-heap avoiding copying of cached data on to the java heap. See Offheap read-path. In hbase-

2.0.0, off-heap latencies approach those of on-heap cache latencies with the added benefit of NOT provoking GC.

From HBase 2.0.0 onwards, the notions of L1 and L2 have been deprecated. When BucketCache is turned on, the DATA blocks will always go to BucketCache and INDEX/BLOOM blocks go to on heap LRUBlockCache. `cacheDataInL1` support has been removed.

BucketCache Deploy Modes

The BucketCache Block Cache can be deployed *offheap*, *file* or *mmaped* file mode.

You set which via the `hbase.bucketcache.ioengine` setting. Setting it to `offheap` will have BucketCache make its allocations off-heap, and an ioengine setting of `file:PATH_TO_FILE` will direct BucketCache to use file caching (Useful in particular if you have some fast I/O attached to the box such as SSDs). From 2.0.0, it is possible to have more than one file backing the BucketCache. This is very useful especially when the Cache size requirement is high. For multiple backing files, configure ioengine as `files:PATH_TO_FILE1,PATH_TO_FILE2,PATH_TO_FILE3`. BucketCache can be configured to use an mmapped file also. Configure ioengine as `mmap:PATH_TO_FILE` for this.

It is possible to deploy a tiered setup where we bypass the CombinedBlockCache policy and have BucketCache working as a strict L2 cache to the L1 LruBlockCache. For such a setup, set `hbase.bucketcache.combinedcache.enabled` to `false`. In this mode, on eviction from L1, blocks go to L2. When a block is cached, it is cached first in L1. When we go to look for a cached block, we look first in L1 and if none found, then search L2. Let us call this deploy format, *Raw L1+L2*. NOTE: This L1+L2 mode is removed from 2.0.0. When BucketCache is used, it will be strictly the DATA cache and the LruBlockCache will cache INDEX/META blocks.

Other BucketCache configs include: specifying a location to persist cache to across restarts, how many threads to use writing the cache, etc. See the CacheConfig.html class for configuration options and descriptions.

To check it enabled, look for the log line describing cache setup; it will detail how BucketCache has been deployed. Also see the UI. It will detail the cache tiering and their configuration.

BucketCache Example Configuration

This sample provides a configuration for a 4 GB off-heap BucketCache with a 1 GB on-heap cache.

Configuration is performed on the RegionServer.

Setting `hbase.bucketcache.ioengine` and `hbase.bucketcache.size` > 0 enables `CombinedBlockCache`. Let us presume that the RegionServer has been set to run with a 5G heap: i.e. `HBASE_HEAPSIZE=5g`.

1. First, edit the RegionServer's *hbase-env.sh* and set `HBASE_OFFHEAPSIZE` to a value greater than the off-heap size wanted, in this case, 4 GB (expressed as 4G). Let's set it to 5G. That'll be 4G for our off-heap cache and 1G for any other uses of off-heap memory (there are other users of off-heap memory other than BlockCache; e.g. DFSClient in RegionServer can make use of off-heap memory). See Direct Memory Usage In HBase below.

   ```
   HBASE_OFFHEAPSIZE=5G
   ```

2. Next, add the following configuration to the RegionServer's *hbase-site.xml*.

   ```
   <property>
     <name>hbase.bucketcache.ioengine</name>
     <value>offheap</value>
   </property>
   <property>
     <name>hfile.block.cache.size</name>
     <value>0.2</value>
   </property>
   <property>
     <name>hbase.bucketcache.size</name>
     <value>4196</value>
   </property>
   ```

3. Restart or rolling restart your cluster, and check the logs for any issues.

In the above, we set the BucketCache to be 4G. We configured the on-heap LruBlockCache have 20% (0.2) of the RegionServer's heap size (0.2 * 5G = 1G). In other words, you configure the L1 LruBlockCache as you would normally (as if there were no L2 cache present).

HBASE-10641 introduced the ability to configure multiple sizes for the buckets of the BucketCache, in HBase 0.98 and newer. To configurable multiple bucket sizes, configure the new property `hbase.bucketcache.bucket.sizes` to a comma-separated list of block sizes, ordered from smallest to largest, with no spaces. The goal is to optimize the bucket sizes based on your data access patterns. The following example configures buckets of size 4096 and 8192.

```
<property>
  <name>hbase.bucketcache.bucket.sizes</name>
  <value>4096,8192</value>
</property>
```

Direct Memory Usage In HBase

The default maximum direct memory varies by JVM. Traditionally it is 64M or some relation to allocated heap size (-Xmx) or no limit at all (JDK7 apparently). HBase servers use direct memory, in particular short-circuit reading (See Leveraging local data), the hosted DFSClient will allocate direct memory buffers. How much the DFSClient uses is not easy to quantify; it is the number of open HFiles * `hbase.dfs.client.read.shortcircuit.buffer.size` where `hbase.dfs.client.read.shortcircuit.buffer.size` is set to 128k in HBase — see *hbase-default.xml* default configurations. If you do off-heap block caching, you'll be making use of direct memory. The RPCServer uses a ByteBuffer pool. From 2.0.0, these buffers are off-heap ByteBuffers. Starting your JVM, make sure the `-XX:MaxDirectMemorySize` setting in *conf/hbase-env.sh* considers off-heap BlockCache (`hbase.bucketcache.size`), DFSClient usage, RPC side ByteBufferPool max size. This has to be bit higher than sum of off heap BlockCache size and max ByteBufferPool size. Allocating an extra of 1-2 GB for the max direct memory size has worked in tests. Direct memory, which is part of the Java process heap, is separate from the object heap allocated by -Xmx. The value allocated by `MaxDirectMemorySize` must not exceed physical RAM, and is likely to be less than the total available RAM due to other memory requirements and system constraints.

You can see how much memory — on-heap and off-heap/direct — a RegionServer is configured to use and how much it is using at any one time by looking at the *Server Metrics: Memory* tab in the UI. It can also be gotten via JMX. In particular the direct memory currently used by the server can be found on the `java.nio.type=BufferPool,name=direct` bean. Terracotta has a good write up on using off-heap memory in Java. It is for their product BigMemory but a lot of the issues noted apply in general to any attempt at going off-heap. Check it out.

hbase.bucketcache.percentage.in.combinedcache

This is a pre-HBase 1.0 configuration removed because it was confusing. It was a float that you would set to some value between 0.0 and 1.0. Its default was 0.9. If the deploy was using CombinedBlockCache, then the LruBlockCache L1 size was calculated to be `(1 - hbase.bucketcache.percentage.in.combinedcache) * size-of-bucketcache` and the BucketCache size was `hbase.bucketcache.percentage.in.combinedcache * size-of-bucket-cache`. where size-of-bucket-cache itself is EITHER the value of the configuration `hbase.bucketcache.size`

> IF it was specified as Megabytes OR `hbase.bucketcache.size` * `-XX:MaxDirectMemorySize` if `hbase.bucketcache.size` is between 0 and 1.0.
>
> In 1.0, it should be more straight-forward. Onheap LruBlockCache size is set as a fraction of java heap using `hfile.block.cache.size setting` (not the best name) and BucketCache is set as above in absolute Megabytes.

Time Based Priority for BucketCache

HBASE-28463 introduced time based priority for blocks in BucketCache. It allows for defining an age threshold at individual column families' configuration, whereby blocks older than this configured threshold would be targeted first for eviction.

Blocks from column families that don't define the age threshold wouldn't be evaluated by the time based priority, and would only be evicted following the LRU eviction logic.

This feature is mostly useful for use cases where most recent data is more frequently accessed, and therefore should get higher priority in the cache. Configuring Time Based Priority with the "age" of most accessed data would then give a finer control over blocks allocation in the BucketCache than the built-in LRU eviction logic.

Time Based Priority for BucketCache provides three different strategies for defining data age:

- Cell timestamps: Uses the timestamp portion of HBase cells for comparing the data age.
- Custom cell qualifiers: Uses a custom-defined date qualifier for comparing the data age. It uses that value to tier the entire row containing the given qualifier value. This requires that the custom qualifier be a valid Java long timestamp.
- Custom value provider: Allows for defining a pluggable implementation that contains the logic for identifying the date value to be used for comparison. This also provides additional flexibility for different use cases that might have the date stored in other formats or embedded with other data in various portions of a given row.

For use cases where priority is determined by the order of record ingestion in HBase (with the most recent being the most relevant), the built-in cell timestamp offers the most convenient and efficient method for configuring age-based priority. See **Using Cell timestamps for Time Based Priority**.

Some applications may utilize a custom date column to define the priority of table records. In such instances, a custom cell qualifier-based priority is advisable. See **Using Custom**

Cell Qualifiers for Time Based Priority.

Finally, more intricate schemas may incorporate domain-specific logic for defining the age of each record. The custom value provider facilitates the integration of custom code to implement the appropriate parsing of the date value that should be used for the priority comparison. See Using a Custom value provider for Time Based Priority.

With Time Based Priority for BucketCache, blocks age is evaluated when deciding if a block should be cached (i.e. during reads, writes, compaction and prefetch), as well as during the cache freeSpace run (mass eviction), prior to executing the LRU logic.

Because blocks don't hold any specific meta information other than type, it's necessary to group blocks of the same "age group" on separate files, using specialized compaction implementations (see more details in the configuration section below). The time range of all blocks in each file is then appended at the file meta info section, and is used for evaluating the age of blocks that should be considered in the Time Based Priority logic.

Configuring Time Based Priority for BucketCache

Finding the age of each block involves an extra overhead, therefore the feature is disabled by default at a global configuration level.

To enable it, the following configuration should be set on RegionServers' *hbase-site.xml*:

```
<property>
  <name>hbase.regionserver.datatiering.enable</name>
  <value>true</value>
</property>
```

Once enabled globally, it's necessary to define the desired strategy-specific settings at the individual column family level.

Using Cell timestamps for Time Based Priority

This strategy is the most efficient to run, as it uses the timestamp portion of each cell containing the data for comparing the age of blocks. It requires DateTieredCompaction for splitting the blocks into separate files according to blocks' ages.

The example below sets the hot age threshold to one week (in milliseconds) for the column family 'cf1' in table 'orders':

```
hbase(main):003:0> alter 'orders', {NAME => 'cf1',
  CONFIGURATION => {'hbase.hstore.datatiering.type' => 'TIME_RANGE',
    'hbase.hstore.datatiering.hot.age.millis' => '604800000',
    'hbase.hstore.engine.class' => 'org.apache.hadoop.hbase.regionserver.DateTier
edStoreEngine',
    'hbase.hstore.blockingStoreFiles' => '60',
    'hbase.hstore.compaction.min' => '2',
    'hbase.hstore.compaction.max' => '60'
  }
}
```

Date Tiered Compaction specific tunings

In the example above, the properties governing the number of windows and period of each window in the date tiered compaction were not set. With the default settings, the compaction will create initially four windows of six hours, then four windows of one day each, then another four windows of four days each and so on until the minimum timestamp among the selected files is covered. This can create a large number of files, therefore, additional changes to the 'hbase.hstore.blockingStoreFiles', 'hbase.hstore.compaction.min' and 'hbase.hstore.compaction.max' are recommended.

Alternatively, consider adjusting the initial window size to the same as the hot age threshold, and two windows only per tier:

```
hbase(main):003:0> alter 'orders', {NAME => 'cf1',
  CONFIGURATION => {'hbase.hstore.datatiering.type' => 'TIME_RANGE',
    'hbase.hstore.datatiering.hot.age.millis' => '604800000',
    'hbase.hstore.engine.class' => 'org.apache.hadoop.hbase.regionserver.D
ateTieredStoreEngine',
    'hbase.hstore.compaction.date.tiered.base.window.millis' => '60480000
0',
    'hbase.hstore.compaction.date.tiered.windows.per.tier' => '2'
  }
}
```

Using Custom Cell Qualifiers for Time Based Priority

This strategy uses a new compaction implementation designed for Time Based Priority. It extends date tiered compaction, but instead of producing multiple tiers of various time windows, it simply splits files into two groups: the "cold" group, where all blocks are older than the defined threshold age, and the "hot" group, where all blocks are newer than the threshold age.

The example below defines a cell qualifier 'event_date' to be used for comparing the age of blocks within the custom cell qualifier strategy:

```
hbase(main):003:0> alter 'orders', {NAME => 'cf1',
```

```
  CONFIGURATION => {'hbase.hstore.datatiering.type' => 'CUSTOM',
    'TIERING_CELL_QUALIFIER' => 'event_date',
    'hbase.hstore.datatiering.hot.age.millis' => '604800000',
    'hbase.hstore.engine.class' => 'org.apache.hadoop.hbase.regionserver.CustomTi
eredStoreEngine',
    'hbase.hstore.compaction.date.tiered.custom.age.limit.millis' => '604800000'
  }
}
```

> **Time Based Priority x Compaction Age Threshold Configurations**
>
> Note that there are two different configurations for defining the hot age threshold. This is because the Time Based Priority enforcer operates independently of the compaction implementation.

Using a Custom value provider for Time Based Priority

It's also possible to hook in domain-specific logic for defining the data age of each row to be used for comparing blocks priorities. The Custom Time Based Priority framework defines the `CustomTieredCompactor.TieringValueProvider` interface, which can be implemented to provide the specific date value to be used by compaction for grouping the blocks according to the threshold age.

In the following example, the `RowKeyPortionTieringValueProvider` implements the `getTierin gValue` method. This method parses the date from a segment of the row key value, specifically between positions 14 and 29, using the "yyyyMMddHHmmss" format. The parsed date is then returned as a long timestamp, which is then used by custom tiered compaction to group the blocks based on the defined hot age threshold:

```
public class RowKeyPortionTieringValueProvider implements CustomTieredCompactor.T
ieringValueProvider {
   private SimpleDateFormat sdf = new SimpleDateFormat("yyyyMMddHHmmss");
   @Override
   public void init(Configuration configuration) throws Exception {}

   @Override
   public long getTieringValue(Cell cell) {
     byte[] rowArray = new byte[cell.getRowLength()];
     System.arraycopy(cell.getRowArray(), cell.getRowOffset(), rowArray, 0, cell.
getRowLength());
     String datePortion = Bytes.toString(rowArray).substring(14, 29).trim();
     try {
       return sdf.parse(datePortion).getTime();
     } catch (ParseException e) {
       //handle error
     }
     return Long.MAX_VALUE;
   }
```

```
}
```

The Tiering Value Provider above can then be configured for Time Based Priority as follows:

```
hbase(main):003:0> alter 'orders', {NAME => 'cf1',
  CONFIGURATION => {'hbase.hstore.datatiering.type' => 'CUSTOM',
    'hbase.hstore.custom-tiering-value.provider.class' =>
      'org.apache.hbase.client.example.RowKeyPortionTieringValueProvider',
    'hbase.hstore.datatiering.hot.age.millis' => '604800000',
    'hbase.hstore.engine.class' => 'org.apache.hadoop.hbase.regionserver.CustomTi
eredStoreEngine',
    'hbase.hstore.compaction.date.tiered.custom.age.limit.millis' => '604800000'
  }
}
```

> Upon enabling Custom Time Based Priority (either the custom qualifier or custom value provider) in the column family configuration, it is imperative that major compaction be executed twice on the specified tables to ensure the effective application of the newly configured priorities within the bucket cache.

> Time Based Priority was originally implemented with the cell timestamp strategy only. The original design covering cell timestamp based strategy is available here.
>
> The second phase including the two custom strategies mentioned above is detailed in this separate design doc.

Compressed BlockCache

HBASE-11331 introduced lazy BlockCache decompression, more simply referred to as compressed BlockCache. When compressed BlockCache is enabled data and encoded data blocks are cached in the BlockCache in their on-disk format, rather than being decompressed and decrypted before caching.

For a RegionServer hosting more data than can fit into cache, enabling this feature with SNAPPY compression has been shown to result in 50% increase in throughput and 30% improvement in mean latency while, increasing garbage collection by 80% and increasing overall CPU load by 2%. See HBASE-11331 for more details about how performance was measured and achieved. For a RegionServer hosting data that can comfortably fit into cache, or if your workload is sensitive to extra CPU or garbage-collection load, you may receive less benefit.

The compressed BlockCache is disabled by default. To enable it, set `hbase.block.data.cachecompressed` to `true` in *hbase-site.xml* on all RegionServers.

Cache Aware Load Balancer

Depending on the data size and the configured cache size, the cache warm up can take anywhere from a few minutes to a few hours. This becomes even more critical for HBase deployments over cloud storage, where compute is separated from storage. Doing this everytime the region server starts can be a very expensive process. To eliminate this, HBASE-27313 implemented the cache persistence feature where the region servers periodically persist the blocks cached in the bucket cache. This persisted information is then used to resurrect the cache in the event of a region server restart because of normal restart or crash.

HBASE-27999 implements the cache aware load balancer, which adds to the load balancer the ability to consider the cache allocation of each region on region servers when calculating a new assignment plan, using the region/region server cache allocation information reported by region servers to calculate the percentage of HFiles cached for each region on the hosting server. This information is then used by the balancer as a factor when deciding on an optimal, new assignment plan.

The master node captures the caching information from all the region servers and uses this information to decide on new region assignments while ensuring a minimal impact on the current cache allocation. A region is assigned to the region server where it has a better cache ratio as compared to the region server where it is currently hosted.

The CacheAwareLoadBalancer uses two cost elements for deciding the region allocation. These are described below:

1. Cache Cost
   The cache cost is calculated as the percentage of data for a region cached on the region server where it is either currently hosted or was previously hosted. A region may have multiple HFiles, each of different sizes. A HFile is considered to be fully prefetched when all the data blocks in this file are in the cache. The region server hosting this region calculates the ratio of number of HFiles fully cached in the cache to the total number of HFiles in the region. This ratio will vary from 0 (region hosted on this server, but none of its HFiles are cached into the cache) to 1 (region hosted on this server and all the HFiles for this region are cached into the cache).

Every region server maintains this information for all the regions currently hosted there. In addition to that, this cache ratio is also maintained for the regions which were previously hosted on this region server giving historical information about the regions.

2. Skewness Cost

The cache aware balancer will consider cache cost with the skewness cost to decide on the region assignment plan under following conditions:

1. There is an idle server in the cluster. This can happen when an existing server is restarted or a new server is added to the cluster.
2. When the cost of maintaining the balance in the cluster is greater than the minimum threshold defined by the configuration *hbase.master.balancer.stochastic.minCostNeedBalance*.

The CacheAwareLoadBalancer can be enabled in the cluster by setting the following configuration properties in the master master configuration:

```
<property>
  <name>hbase.master.loadbalancer.class</name>
  <value>org.apache.hadoop.hbase.master.balancer.CacheAwareLoadBalancer</value>
</property>
<property>
  <name>hbase.bucketcache.persistent.path</name>
  <value>/path/to/bucketcache_persistent_file</value>
</property>
```

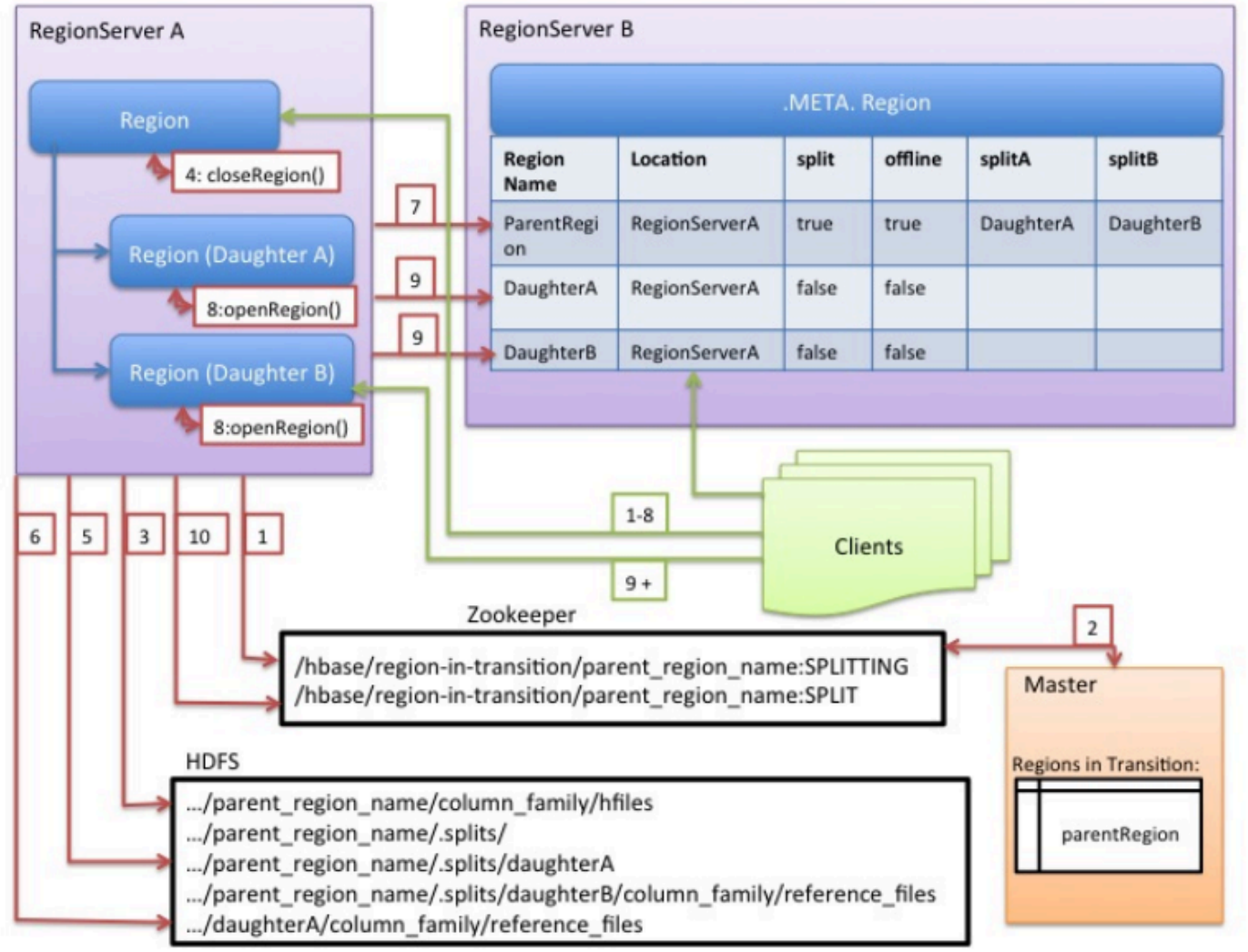
Within HBASE-29168, the CacheAwareLoadBalancer implements region move throttling. This mitigates the impact of "losing" cache factor when balancing mainly due to region skewness, i.e. when new region servers are added to the cluster, a large bulk of cached regions may move to the new servers at once, which can cause noticeable read performance impacts for cache sensitive use cases. The throttling sleep time is determined by the **hbase.master.balancer.move.throttlingMillis** property, and it defaults to 60000 millis. If a region planned to be moved has a cache ratio on the target server above the thershold configurable by the **hbase.master.balancer.stochastic.throttling.cacheRatio** property (80% by default), no throttling will be applied in this region move.

RegionServer Splitting Implementation

As write requests are handled by the region server, they accumulate in an in-memory storage system called the *memstore*. Once the memstore fills, its content are written to disk as additional store files. This event is called a *memstore flush*. As store files accumulate, the RegionServer will compact them into fewer, larger files. After each flush or compaction finishes, the amount of data stored in the region has changed. The RegionServer consults the region split policy to determine if the region has grown too large or should be split for another policy-specific reason. A region split request is enqueued if the policy recommends it.

Logically, the process of splitting a region is simple. We find a suitable point in the keyspace of the region where we should divide the region in half, then split the region's data into two new regions at that point. The details of the process however are not simple. When a split happens, the newly created *daughter regions* do not rewrite all the data into new files immediately. Instead, they create small files similar to symbolic link files, named Reference files, which point to either the top or bottom part of the parent store file according to the split point. The reference file is used just like a regular data file, but only half of the records are considered. The region can only be split if there are no more references to the immutable data files of the parent region. Those reference files are cleaned gradually by compactions, so that the region will stop referring to its parents files, and can be split further.

Although splitting the region is a local decision made by the RegionServer, the split process itself must coordinate with many actors. The RegionServer notifies the Master before and after the split, updates the `.META.` table so that clients can discover the new daughter regions, and rearranges the directory structure and data files in HDFS. Splitting is a multi-task process. To enable rollback in case of an error, the RegionServer keeps an in-memory journal about the execution state. The steps taken by the RegionServer to execute the split are illustrated in the "RegionServer Split Process" schema below. Each step is labeled with its step number. Actions from RegionServers or Master are shown in red, while actions from the clients are shown in green.



1. The RegionServer decides locally to split the region, and prepares the split. **THE SPLIT TRANSACTION IS STARTED.** As a first step, the RegionServer acquires a shared read lock on the table to prevent schema modifications during the splitting process. Then it creates a znode in zookeeper under `/hbase/region-in-transition/region-name`, and sets the znode's state to `SPLITTING`.
2. The Master learns about this znode, since it has a watcher for the parent `region-in-transition` znode.
3. The RegionServer creates a sub-directory named `.splits` under the parent's `region` directory in HDFS.
4. The RegionServer closes the parent region and marks the region as offline in its local data structures. **THE SPLITTING REGION IS NOW OFFLINE.** At this point, client requests coming to the parent region will throw `NotServingRegionException`. The client will retry with some backoff. The closing region is flushed.
5. The RegionServer creates region directories under the `.splits` directory, for daughter regions A and B, and creates necessary data structures. Then it splits the store files, in

the sense that it creates two Reference files per store file in the parent region. Those reference files will point to the parent region's files.

6. The RegionServer creates the actual region directory in HDFS, and moves the reference files for each daughter.
7. The RegionServer sends a `Put` request to the `.META.` table, to set the parent as offline in the `.META.` table and add information about daughter regions. At this point, there won't be individual entries in `.META.` for the daughters. Clients will see that the parent region is split if they scan `.META.`, but won't know about the daughters until they appear in `.META.`. Also, if this `Put` to `.META` succeeds, the parent will be effectively split. If the RegionServer fails before this RPC succeeds, Master and the next Region Server opening the region will clean dirty state about the region split. After the `.META.` update, though, the region split will be rolled-forward by Master.
8. The RegionServer opens daughters A and B in parallel.
9. The RegionServer adds the daughters A and B to `.META.`, together with information that it hosts the regions. **THE SPLIT REGIONS (DAUGHTERS WITH REFERENCES TO PARENT) ARE NOW ONLINE.** After this point, clients can discover the new regions and issue requests to them. Clients cache the `.META.` entries locally, but when they make requests to the RegionServer or `.META.`, their caches will be invalidated, and they will learn about the new regions from `.META.`.
10. The RegionServer updates znode `/hbase/region-in-transition/region-name` in ZooKeeper to state `SPLIT`, so that the master can learn about it. The balancer can freely re-assign the daughter regions to other region servers if necessary. **THE SPLIT TRANSACTION IS NOW FINISHED.**
11. After the split, `.META.` and HDFS will still contain references to the parent region. Those references will be removed when compactions in daughter regions rewrite the data files. Garbage collection tasks in the master periodically check whether the daughter regions still refer to the parent region's files. If not, the parent region will be removed.

Note on split directory behavior (HBASE-26187)

> **Behavior change in newer split implementation**
>
> The numbered split procedure above documents the original, pre-HBASE-26187 implementation, in which daughter regions are first created under a `.splits` sub-directory of the parent region

directory. In newer HBase implementations that include HBASE-26187, this on-disk layout has changed: daughter region directories are created directly under the table directory, rather than under a parent region's `.splits` directory. When reading the numbered procedure for such newer versions, interpret any steps that mention creating or using a `.splits` directory as instead preparing and promoting daughter region files directly under the table directory. This update does not change the high-level split semantics (references, meta updates, and online transition of daughters); it only affects where temporary/initial daughter files are prepared in HDFS.

Write Ahead Log (WAL)

Purpose

The *Write Ahead Log (WAL)* records all changes to data in HBase, to file-based storage. Under normal operations, the WAL is not needed because data changes move from the MemStore to StoreFiles. However, if a RegionServer crashes or becomes unavailable before the MemStore is flushed, the WAL ensures that the changes to the data can be replayed. If writing to the WAL fails, the entire operation to modify the data fails.

HBase uses an implementation of the WAL interface. Usually, there is only one instance of a WAL per RegionServer. An exception is the RegionServer that is carrying *hbase:meta*; the *meta* table gets its own dedicated WAL. The RegionServer records Puts and Deletes to its WAL, before recording them these Mutations MemStore for the affected Store.

The HLog

Prior to 2.0, the interface for WALs in HBase was named `HLog`. In 0.94, HLog was the name of the implementation of the WAL. You will likely find references to the HLog in documentation tailored to these older versions.

The WAL resides in HDFS in the */hbase/WALs/* directory, with subdirectories per RegionServer.

For more general information about the concept of write ahead logs, see the Wikipedia Write-Ahead Log article.

WAL Providers

In HBase, there are a number of WAL implementations (or 'Providers'). Each is known by a short name label (that unfortunately is not always descriptive). You set the provider in *hbase-site.xml* passing the WAL provider short-name as the value on the

hbase.wal.provider property (Set the provider for *hbase:meta* using the *hbase.wal.meta_provider* property, otherwise it uses the same provider configured by *hbase.wal.provider*).

- *asyncfs*: The **default**. New since hbase-2.0.0 (HBASE-15536, HBASE-14790). This *AsyncFSWAL* provider, as it identifies itself in RegionServer logs, is built on a new non-blocking dfsclient implementation. It is currently resident in the hbase codebase but intent is to move it back up into HDFS itself. WALs edits are written concurrently ("fan-out") style to each of the WAL-block replicas on each DataNode rather than in a chained pipeline as the default client does. Latencies should be better. See Apache HBase Improvements and Practices at Xiaomi at slide 14 onward for more detail on implementation.
- *filesystem*: This was the default in hbase-1.x releases. It is built on the blocking *DFSClient* and writes to replicas in classic *DFSCLient* pipeline mode. In logs it identifies as *FSHLog* or *FSHLogProvider*.
- *multiwal*: This provider is made of multiple instances of *asyncfs* or *filesystem*. See the next section for more on *multiwal*.

Look for the lines like the below in the RegionServer log to see which provider is in place (The below shows the default AsyncFSWALProvider):

```
2018-04-02 13:22:37,983 INFO  [regionserver/ve0528:16020] wal.WALFactory: Instant
iating WALProvider of type class org.apache.hadoop.hbase.wal.AsyncFSWALProvider
```

> As the *AsyncFSWAL* hacks into the internal of DFSClient implementation, it will be easily broken by upgrading the hadoop dependencies, even for a simple patch release. So if you do not specify the wal provider explicitly, we will first try to use the *asyncfs*, if failed, we will fall back to use *filesystem*. And notice that this may not always work, so if you still have problem starting HBase due to the problem of starting *AsyncFSWAL*, please specify *filesystem* explicitly in the config file.

> EC support has been added to hadoop-3.x, and it is incompatible with WAL as the EC output stream does not support hflush/hsync. In order to create a non-EC file in an EC directory, we need to use the new builder-based create API for *FileSystem*, but it is only introduced in hadoop-2.9+ and for HBase we still need to support hadoop-2.7.x. So please do not enable EC for the WAL directory until we find a way to deal with it.

MultiWAL

With a single WAL per RegionServer, the RegionServer must write to the WAL serially, because HDFS files must be sequential. This causes the WAL to be a performance bottleneck.

HBase 1.0 introduces support MultiWal in HBASE-5699. MultiWAL allows a RegionServer to write multiple WAL streams in parallel, by using multiple pipelines in the underlying HDFS instance, which increases total throughput during writes. This parallelization is done by partitioning incoming edits by their Region. Thus, the current implementation will not help with increasing the throughput to a single Region.

RegionServers using the original WAL implementation and those using the MultiWAL implementation can each handle recovery of either set of WALs, so a zero-downtime configuration update is possible through a rolling restart.

Configure MultiWAL

To configure MultiWAL for a RegionServer, set the value of the property `hbase.wal.provider` to `multiwal` by pasting in the following XML:

```
<property>
  <name>hbase.wal.provider</name>
  <value>multiwal</value>
</property>
```

Restart the RegionServer for the changes to take effect.

To disable MultiWAL for a RegionServer, unset the property and restart the RegionServer.

WAL Flushing

TODO (describe).

WAL Splitting

A RegionServer serves many regions. All of the regions in a region server share the same active WAL file. Each edit in the WAL file includes information about which region it belongs to. When a region is opened, the edits in the WAL file which belong to that region need to be replayed. Therefore, edits in the WAL file must be grouped by region so that

particular sets can be replayed to regenerate the data in a particular region. The process of grouping the WAL edits by region is called *log splitting*. It is a critical process for recovering data if a region server fails.

Log splitting is done by the HMaster during cluster start-up or by the ServerShutdownHandler as a region server shuts down. So that consistency is guaranteed, affected regions are unavailable until data is restored. All WAL edits need to be recovered and replayed before a given region can become available again. As a result, regions affected by log splitting are unavailable until the process completes.

Procedure: Log Splitting, Step by Step

1. The `/hbase/WALs/HOST,PORT,STARTCODE` directory is renamed

   Renaming the directory is important because a RegionServer may still be up and accepting requests even if the HMaster thinks it is down. If the RegionServer does not respond immediately and does not heartbeat its ZooKeeper session, the HMaster may interpret this as a RegionServer failure. Renaming the logs directory ensures that existing, valid WAL files which are still in use by an active but busy RegionServer are not written to by accident.

   The new directory is named according to the following pattern:

   ```
   /hbase/WALs/HOST,PORT,STARTCODE-splitting
   ```

   An example of such a renamed directory might look like the following:

   ```
   /hbase/WALs/srv.example.com,60020,1254173957298-splitting
   ```

2. Each log file is split, one at a time

   The log splitter reads the log file one edit entry at a time and puts each edit entry into the buffer corresponding to the edit's region. At the same time, the splitter starts several writer threads. Writer threads pick up a corresponding buffer and write the edit entries in the buffer to a temporary recovered edit file. The temporary edit file is stored to disk with the following naming pattern:

   ```
   /hbase/TABLE_NAME/REGION_ID/recovered.edits/.temp
   ```

This file is used to store all the edits in the WAL log for this region. After log splitting completes, the *.temp* file is renamed to the sequence ID of the first log written to the file.

To determine whether all edits have been written, the sequence ID is compared to the sequence of the last edit that was written to the HFile. If the sequence of the last edit is greater than or equal to the sequence ID included in the file name, it is clear that all writes from the edit file have been completed.

3. After log splitting is complete, each affected region is assigned to a RegionServer

   When the region is opened, the *recovered.edits* folder is checked for recovered edits files. If any such files are present, they are replayed by reading the edits and saving them to the MemStore. After all edit files are replayed, the contents of the MemStore are written to disk (HFile) and the edit files are deleted.

Handling of Errors During Log Splitting

If you set the `hbase.hlog.split.skip.errors` option to `true`, errors are treated as follows:

- Any error encountered during splitting will be logged.
- The problematic WAL log will be moved into the *.corrupt* directory under the hbase `root dir`,
- Processing of the WAL will continue

If the `hbase.hlog.split.skip.errors` option is set to `false`, the default, the exception will be propagated and the split will be logged as failed. See HBASE-2958 When hbase.hlog.split.skip.errors is set to false, we fail the split but that's it. We need to do more than just fail split if this flag is set.

How EOFExceptions are treated when splitting a crashed RegionServer's WALs

If an EOFException occurs while splitting logs, the split proceeds even when `hbase.hlog.split.skip.errors` is set to `false`. An EOFException while reading the last log in the set of files to split is likely, because the RegionServer was likely in the process of writing a record at the time of a crash. For background, see HBASE-2643 Figure how to deal with eof splitting logs

Performance Improvements during Log Splitting

WAL log splitting and recovery can be resource intensive and take a long time, depending on the number of RegionServers involved in the crash and the size of the regions. Distributed log splitting was developed to improve performance during log splitting.

Enabling or Disabling Distributed Log Splitting

Distributed log processing is enabled by default since HBase 0.92. The setting is controlled by the `hbase.master.distributed.log.splitting` property, which can be set to `true` or `false`, but defaults to `true`.

WAL splitting based on procedureV2

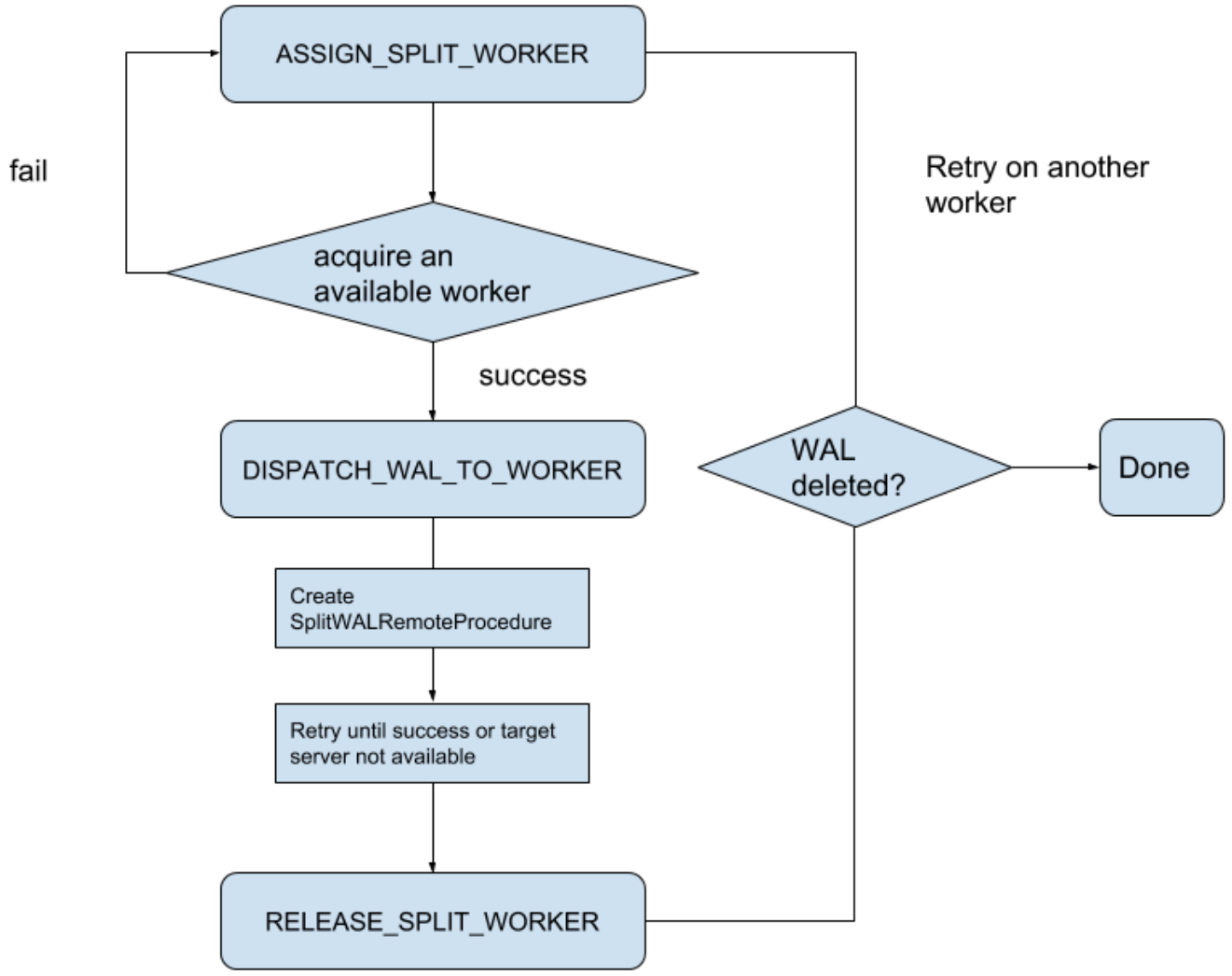
After HBASE-20610, we introduce a new way to do WAL splitting coordination by procedureV2 framework. This can simplify the process of WAL splitting and no need to connect zookeeper any more.

Background

Currently, splitting WAL processes are coordinated by zookeeper. Each region server are trying to grab tasks from zookeeper. And the burden becomes heavier when the number of region server increase.

Implementation on Master side

During ServerCrashProcedure, SplitWALManager will create one SplitWALProcedure for each WAL file which should be split. Then each SplitWALProcedure will spawn a SplitWalRemoteProcedure to send the request to region server. SplitWALProcedure is a StateMachineProcedure and here is the state transfer diagram.



Implementation on Region Server side

Region Server will receive a SplitWALCallable and execute it, which is much more straightforward than before. It will return null if success and return exception if there is any error.

Performance

According to tests on a cluster which has 5 regionserver and 1 master. procedureV2 coordinated WAL splitting has a better performance than ZK coordinated WAL splitting no master when restarting the whole cluster or one region server crashing.

Enable this feature

To enable this feature, first we should ensure our package of HBase already contains these code. If not, please upgrade the package of HBase cluster without any configuration change first. Then change configuration 'hbase.split.wal.zk.coordinated' to false. Rolling upgrade the master with new configuration. Now WAL splitting are handled by our new

implementation. But region server are still trying to grab tasks from zookeeper, we can rolling upgrade the region servers with the new configuration to stop that.

- Steps as follows:
  - Upgrade whole cluster to get the new Implementation.
  - Upgrade Master with new configuration 'hbase.split.wal.zk.coordinated'=false.
  - Upgrade region server to stop grab tasks from zookeeper.

WAL Compression

The content of the WAL can be compressed using LRU Dictionary compression. This can be used to speed up WAL replication to different datanodes. The dictionary can store up to 2^{15} elements; eviction starts after this number is exceeded.

To enable WAL compression, set the `hbase.regionserver.wal.enablecompression` property to `true`. The default value for this property is `false`. By default, WAL tag compression is turned on when WAL compression is enabled. You can turn off WAL tag compression by setting the `hbase.regionserver.wal.tags.enablecompression` property to 'false'.

A possible downside to WAL compression is that we lose more data from the last block in the WAL if it is ill-terminated mid-write. If entries in this last block were added with new dictionary entries but we failed persist the amended dictionary because of an abrupt termination, a read of this last block may not be able to resolve last-written entries.

Durability

It is possible to set *durability* on each Mutation or on a Table basis. Options include:

- *SKIP_WAL*: Do not write Mutations to the WAL (See the next section, Disabling the WAL).
- *ASYNC_WAL*: Write the WAL asynchronously; do not hold-up clients waiting on the sync of their write to the filesystem but return immediately. The edit becomes visible. Meanwhile, in the background, the Mutation will be flushed to the WAL at some time later. This option currently may lose data. See HBASE-16689.
- *SYNC_WAL*: The **default**. Each edit is sync'd to HDFS before we return success to the client.

- *FSYNC_WAL*: Each edit is fsync'd to HDFS and the filesystem before we return success to the client.

Do not confuse the *ASYNC_WAL* option on a Mutation or Table with the *AsyncFSWAL* writer; they are distinct options unfortunately closely named

Custom WAL Directory

HBASE-17437 added support for specifying a WAL directory outside the HBase root directory or even in a different FileSystem since 1.3.3/2.0+. Some FileSystems (such as Amazon S3) don't support append or consistent writes, in such scenario WAL directory needs to be configured in a different FileSystem to avoid loss of writes.

Following configurations are added to accomplish this:

1. `hbase.wal.dir`
   This defines where the root WAL directory is located, could be on a different FileSystem than the root directory. WAL directory can not be set to a subdirectory of the root directory. The default value of this is the root directory if unset.
2. `hbase.rootdir.perms`
   Configures FileSystem permissions to set on the root directory. This is '700' by default.
3. `hbase.wal.dir.perms`
   Configures FileSystem permissions to set on the WAL directory FileSystem. This is '700' by default.

> While migrating to custom WAL dir (outside the HBase root directory or a different FileSystem) existing WAL files must be copied manually to new WAL dir, otherwise it may lead to data loss/inconsistency as HMaster has no information about previous WAL directory.

Disabling the WAL

It is possible to disable the WAL, to improve performance in certain specific situations. However, disabling the WAL puts your data at risk. The only situation where this is recommended is during a bulk load. This is because, in the event of a problem, the bulk load can be re-run with no risk of data loss.

The WAL is disabled by calling the HBase client field `Mutation.writeToWAL(false)`. Use the `Mutation.setDurability(Durability.SKIP_WAL)` and Mutation.getDurability() methods to set and get the field's value. There is no way to disable the WAL for only a specific table.

> ⚠️ If you disable the WAL for anything other than bulk loads, your data is at risk.

Regions

Regions are the basic element of availability and distribution for tables, and are comprised of a Store per Column Family. The hierarchy of objects is as follows:

```
Table                    (HBase table)
    Region               (Regions for the table)
        Store            (Store per ColumnFamily for each Region for the table)
            MemStore     (MemStore for each Store for each Region for the table)
            StoreFile    (StoreFiles for each Store for each Region for the tabl
e)
                Block    (Blocks within a StoreFile within a Store for each Regio
n for the table)
```

For a description of what HBase files look like when written to HDFS, see Browsing HDFS for HBase Objects.

Considerations for Number of Regions

In general, HBase is designed to run with a small (20-200) number of relatively large (5-20Gb) regions per server. The considerations for this are as follows:

Why should I keep my Region count low?

Typically you want to keep your region count low on HBase for numerous reasons. Usually right around 100 regions per RegionServer has yielded the best results. Here are some of the reasons below for keeping region count low:

1. MSLAB (MemStore-local allocation buffer) requires 2MB per MemStore (that's 2MB per family per region). 1000 regions that have 2 families each is 3.9GB of heap used, and it's not even storing data yet. NB: the 2MB value is configurable.

2. If you fill all the regions at somewhat the same rate, the global memory usage makes it that it forces tiny flushes when you have too many regions which in turn generates compactions. Rewriting the same data tens of times is the last thing you want. An example is filling 1000 regions (with one family) equally and let's consider a lower bound for global MemStore usage of 5GB (the region server would have a big heap). Once it reaches 5GB it will force flush the biggest region, at that point they should almost all have about 5MB of data so it would flush that amount. 5MB inserted later, it would flush another region that will now have a bit over 5MB of data, and so on. This is currently the main limiting factor for the number of regions; see **Number of regions per RS - upper bound** for detailed formula.
3. The master as is is allergic to tons of regions, and will take a lot of time assigning them and moving them around in batches. The reason is that it's heavy on ZK usage, and it's not very async at the moment (could really be improved — and has been improved a bunch in 0.96 HBase).
4. In older versions of HBase (pre-HFile v2, 0.90 and previous), tons of regions on a few RS can cause the store file index to rise, increasing heap usage and potentially creating memory pressure or OOME on the RSs

Another issue is the effect of the number of regions on MapReduce jobs; it is typical to have one mapper per HBase region. Thus, hosting only 5 regions per RS may not be enough to get sufficient number of tasks for a MapReduce job, while 1000 regions will generate far too many tasks.

See **Determining region count and size** for configuration guidelines.

Region-RegionServer Assignment

This section describes how Regions are assigned to RegionServers.

Startup

When HBase starts regions are assigned as follows (short version):

1. The Master invokes the `AssignmentManager` upon startup.
2. The `AssignmentManager` looks at the existing region assignments in `hbase:meta`.

3. If the region assignment is still valid (i.e., if the RegionServer is still online) then the assignment is kept.
4. If the assignment is invalid, then the `LoadBalancerFactory` is invoked to assign the region. The load balancer (`StochasticLoadBalancer` by default in HBase 1.0) assign the region to a RegionServer.
5. `hbase:meta` is updated with the RegionServer assignment (if needed) and the RegionServer start codes (start time of the RegionServer process) upon region opening by the RegionServer.

Failover

When a RegionServer fails:

1. The regions immediately become unavailable because the RegionServer is down.
2. The Master will detect that the RegionServer has failed.
3. The region assignments will be considered invalid and will be re-assigned just like the startup sequence.
4. In-flight queries are re-tried, and not lost.
5. Operations are switched to a new RegionServer within the following amount of time:

```
ZooKeeper session timeout + split time + assignment/replay time
```

Region Load Balancing

Regions can be periodically moved by the LoadBalancer.

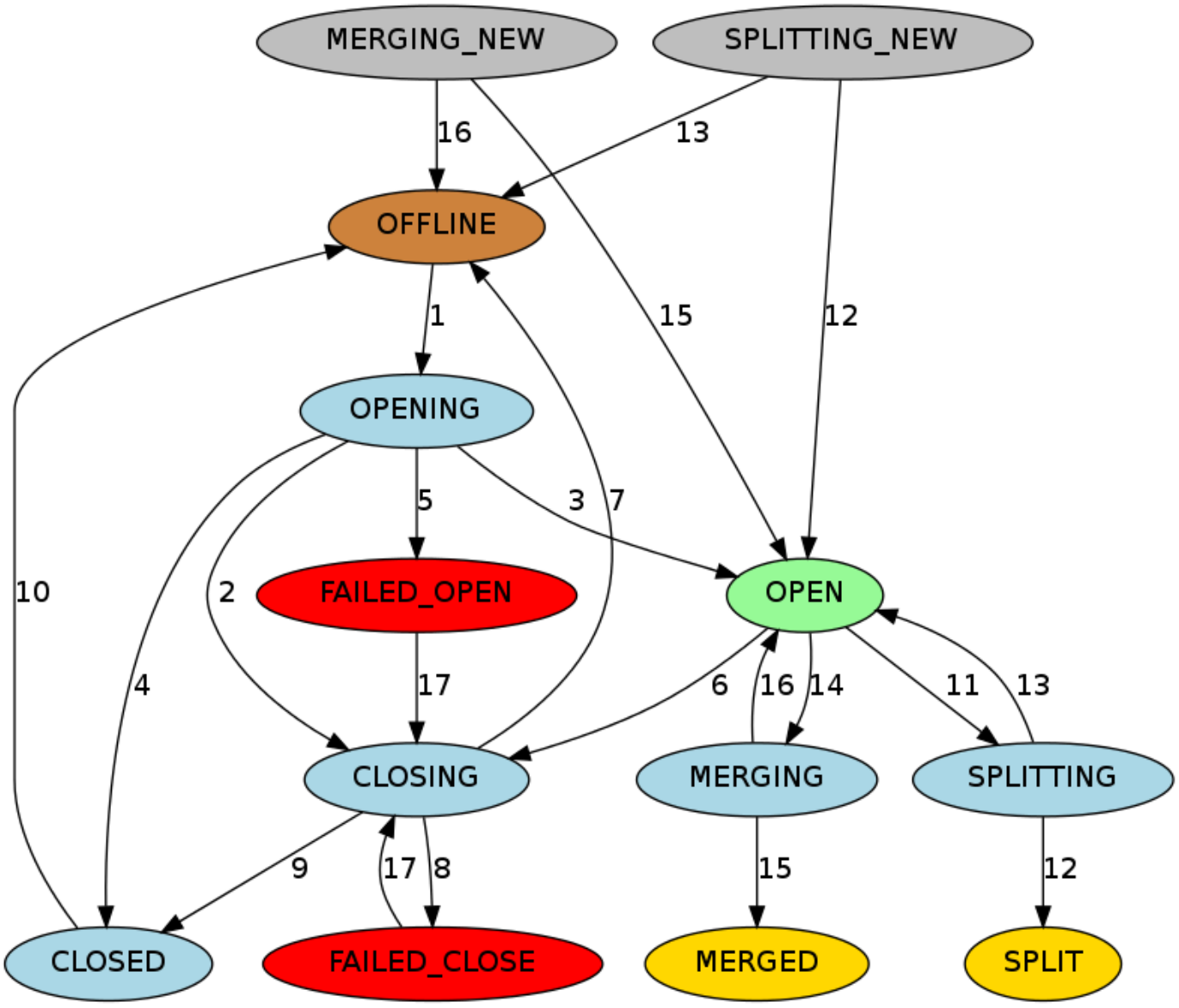
Region State Transition

HBase maintains a state for each region and persists the state in `hbase:meta`. The state of the `hbase:meta` region itself is persisted in ZooKeeper. You can see the states of regions in transition in the Master web UI. Following is the list of possible region states.

Possible Region States:

- `OFFLINE`: the region is offline and not opening
- `OPENING`: the region is in the process of being opened

- `OPEN`: the region is open and the RegionServer has notified the master
- `FAILED_OPEN`: the RegionServer failed to open the region
- `CLOSING`: the region is in the process of being closed
- `CLOSED`: the RegionServer has closed the region and notified the master
- `FAILED_CLOSE`: the RegionServer failed to close the region
- `SPLITTING`: the RegionServer notified the master that the region is splitting
- `SPLIT`: the RegionServer notified the master that the region has finished splitting
- `SPLITTING_NEW`: this region is being created by a split which is in progress
- `MERGING`: the RegionServer notified the master that this region is being merged with another region
- `MERGED`: the RegionServer notified the master that this region has been merged
- `MERGING_NEW`: this region is being created by a merge of two regions



Graph Legend:

- Brown: Offline state, a special state that can be transient (after closed before opening), terminal (regions of disabled tables), or initial (regions of newly created tables)
- Palegreen: Online state that regions can serve requests
- Lightblue: Transient states
- Red: Failure states that need OPS attention
- Gold: Terminal states of regions split/merged
- Grey: Initial states of regions created through split/merge

Transition State Descriptions:

1. The master moves a region from `OFFLINE` to `OPENING` state and tries to assign the region to a RegionServer. The RegionServer may or may not have received the open

region request. The master retries sending the open region request to the RegionServer until the RPC goes through or the master runs out of retries. After the RegionServer receives the open region request, the RegionServer begins opening the region.

2. If the master is running out of retries, the master prevents the RegionServer from opening the region by moving the region to `CLOSING` state and trying to close it, even if the RegionServer is starting to open the region.
3. After the RegionServer opens the region, it continues to try to notify the master until the master moves the region to `OPEN` state and notifies the RegionServer. The region is now open.
4. If the RegionServer cannot open the region, it notifies the master. The master moves the region to `CLOSED` state and tries to open the region on a different RegionServer.
5. If the master cannot open the region on any of a certain number of regions, it moves the region to `FAILED_OPEN` state, and takes no further action until an operator intervenes from the HBase shell, or the server is dead.
6. The master moves a region from `OPEN` to `CLOSING` state. The RegionServer holding the region may or may not have received the close region request. The master retries sending the close request to the server until the RPC goes through or the master runs out of retries.
7. If the RegionServer is not online, or throws `NotServingRegionException`, the master moves the region to `OFFLINE` state and re-assigns it to a different RegionServer.
8. If the RegionServer is online, but not reachable after the master runs out of retries, the master moves the region to `FAILED_CLOSE` state and takes no further action until an operator intervenes from the HBase shell, or the server is dead.
9. If the RegionServer gets the close region request, it closes the region and notifies the master. The master moves the region to `CLOSED` state and re-assigns it to a different RegionServer.
10. Before assigning a region, the master moves the region to `OFFLINE` state automatically if it is in `CLOSED` state.
11. When a RegionServer is about to split a region, it notifies the master. The master moves the region to be split from `OPEN` to `SPLITTING` state and add the two new regions to be created to the RegionServer. These two regions are in `SPLITTING_NEW` state initially.

12. After notifying the master, the RegionServer starts to split the region. Once past the point of no return, the RegionServer notifies the master again so the master can update the `hbase:meta` table. However, the master does not update the region states until it is notified by the server that the split is done. If the split is successful, the splitting region is moved from `SPLITTING` to `SPLIT` state and the two new regions are moved from `SPLITTING_NEW` to `OPEN` state.
13. If the split fails, the splitting region is moved from `SPLITTING` back to `OPEN` state, and the two new regions which were created are moved from `SPLITTING_NEW` to `OFFLINE` state.
14. When a RegionServer is about to merge two regions, it notifies the master first. The master moves the two regions to be merged from `OPEN` to `MERGING` state, and adds the new region which will hold the contents of the merged regions region to the RegionServer. The new region is in `MERGING_NEW` state initially.
15. After notifying the master, the RegionServer starts to merge the two regions. Once past the point of no return, the RegionServer notifies the master again so the master can update the META. However, the master does not update the region states until it is notified by the RegionServer that the merge has completed. If the merge is successful, the two merging regions are moved from `MERGING` to `MERGED` state and the new region is moved from `MERGING_NEW` to `OPEN` state.
16. If the merge fails, the two merging regions are moved from `MERGING` back to `OPEN` state, and the new region which was created to hold the contents of the merged regions is moved from `MERGING_NEW` to `OFFLINE` state.
17. For regions in `FAILED_OPEN` or `FAILED_CLOSE` states, the master tries to close them again when they are reassigned by an operator via HBase Shell.

Region-RegionServer Locality

Over time, Region-RegionServer locality is achieved via HDFS block replication. The HDFS client does the following by default when choosing locations to write replicas:

1. First replica is written to local node
2. Second replica is written to a random node on another rack

3. Third replica is written on the same rack as the second, but on a different node chosen randomly
4. Subsequent replicas are written on random nodes on the cluster. See *Replica Placement: The First Baby Steps* on this page: HDFS Architecture

Thus, HBase eventually achieves locality for a region after a flush or a compaction. In a RegionServer failover situation a RegionServer may be assigned regions with non-local StoreFiles (because none of the replicas are local), however as new data is written in the region, or the table is compacted and StoreFiles are re-written, they will become "local" to the RegionServer.

For more information, see *Replica Placement: The First Baby Steps* on this page: HDFS Architecture and also Lars George's blog on HBase and HDFS locality.

Region Splits

Regions split when they reach a configured threshold. Below we treat the topic in short. For a longer exposition, see Apache HBase Region Splitting and Merging by our Enis Soztutar.

Splits run unaided on the RegionServer; i.e. the Master does not participate. The RegionServer splits a region, offlines the split region and then adds the daughter regions to `hbase:meta`, opens daughters on the parent's hosting RegionServer and then reports the split to the Master. See Managed Splitting for how to manually manage splits (and for why you might do this).

Custom Split Policies

You can override the default split policy using a custom RegionSplitPolicy (HBase 0.94+). Typically a custom split policy should extend HBase's default split policy: IncreasingToUpperBoundRegionSplitPolicy.

The policy can set globally through the HBase configuration or on a per-table basis.

Configuring the Split Policy Globally in *hbase-site.xml*

```
<property>
  <name>hbase.regionserver.region.split.policy</name>
```

```
  <value>org.apache.hadoop.hbase.regionserver.IncreasingToUpperBoundRegionSplitPo
licy</value>
</property>
```

Configuring a Split Policy On a Table Using the Java API

```
HTableDescriptor tableDesc = new HTableDescriptor("test");
tableDesc.setValue(HTableDescriptor.SPLIT_POLICY, ConstantSizeRegionSplitPolicy.c
lass.getName());
tableDesc.addFamily(new HColumnDescriptor(Bytes.toBytes("cf1")));
admin.createTable(tableDesc);
```

Configuring the Split Policy On a Table Using HBase Shell

```
hbase> create 'test', {METADATA => {'SPLIT_POLICY' => 'org.apache.hadoop.hbase.re
gionserver.ConstantSizeRegionSplitPolicy'}},{NAME => 'cf1'}
```

The policy can be set globally through the HBaseConfiguration used or on a per table basis:

```
HTableDescriptor myHtd = ...;
myHtd.setValue(HTableDescriptor.SPLIT_POLICY, MyCustomSplitPolicy.class.getName
());
```

> The `DisabledRegionSplitPolicy` policy blocks manual region splitting.

Manual Region Splitting

It is possible to manually split your table, either at table creation (pre-splitting), or at a later time as an administrative action. You might choose to split your region for one or more of the following reasons. There may be other valid reasons, but the need to manually split your table might also point to problems with your schema design.

Reasons to Manually Split Your Table:

- Your data is sorted by timeseries or another similar algorithm that sorts new data at the end of the table. This means that the Region Server holding the last region is always under load, and the other Region Servers are idle, or mostly idle. See also Monotonically Increasing Row Keys/Timeseries Data.

- You have developed an unexpected hotspot in one region of your table. For instance, an application which tracks web searches might be inundated by a lot of searches for a celebrity in the event of news about that celebrity. See perf.one.region for more discussion about this particular scenario.
- After a big increase in the number of RegionServers in your cluster, to get the load spread out quickly.
- Before a bulk-load which is likely to cause unusual and uneven load across regions.

See Managed Splitting for a discussion about the dangers and possible benefits of managing splitting completely manually.

> The `DisabledRegionSplitPolicy` policy blocks manual region splitting.

Determining Split Points

The goal of splitting your table manually is to improve the chances of balancing the load across the cluster in situations where good rowkey design alone won't get you there. Keeping that in mind, the way you split your regions is very dependent upon the characteristics of your data. It may be that you already know the best way to split your table. If not, the way you split your table depends on what your keys are like.

Alphanumeric Rowkeys

If your rowkeys start with a letter or number, you can split your table at letter or number boundaries. For instance, the following command creates a table with regions that split at each vowel, so the first region has A-D, the second region has E-H, the third region has I-N, the fourth region has O-V, and the fifth region has U-Z.

Using a Custom Algorithm

The RegionSplitter tool is provided with HBase, and uses a *SplitAlgorithm* to determine split points for you. As parameters, you give it the algorithm, desired number of regions, and column families. It includes three split algorithms. The first is the `HexStringSplit` algorithm, which assumes the row keys are hexadecimal strings. The second is the `DecimalStringSplit` algorithm, which assumes the row keys are decimal strings in the range 00000000 to 99999999. The third, `UniformSplit`, assumes the row keys are random byte arrays. You will probably need to develop your own `SplitAlgorithm`, using the provided ones as models.

Online Region Merges

Both Master and RegionServer participate in the event of online region merges. Client sends merge RPC to the master, then the master moves the regions together to the RegionServer where the more heavily loaded region resided. Finally the master sends the merge request to this RegionServer which then runs the merge. Similar to process of region splitting, region merges run as a local transaction on the RegionServer. It offlines the regions and then merges two regions on the file system, atomically delete merging regions from `hbase:meta` and adds the merged region to `hbase:meta`, opens the merged region on the RegionServer and reports the merge to the Master.

An example of region merges in the HBase shell

```
$ hbase> merge_region 'ENCODED_REGIONNAME', 'ENCODED_REGIONNAME'
$ hbase> merge_region 'ENCODED_REGIONNAME', 'ENCODED_REGIONNAME', true
```

It's an asynchronous operation and call returns immediately without waiting merge completed. Passing `true` as the optional third parameter will force a merge. Normally only adjacent regions can be merged. The `force` parameter overrides this behaviour and is for expert use only.

Store

A Store hosts a MemStore and 0 or more StoreFiles (HFiles). A Store corresponds to a column family for a table for a given region.

MemStore

The MemStore holds in-memory modifications to the Store. Modifications are Cells/KeyValues. When a flush is requested, the current MemStore is moved to a snapshot and is cleared. HBase continues to serve edits from the new MemStore and backing snapshot until the flusher reports that the flush succeeded. At this point, the snapshot is discarded. Note that when the flush happens, MemStores that belong to the same region will all be flushed.

MemStore Flush

A MemStore flush can be triggered under any of the conditions listed below. The minimum flush unit is per region, not at individual MemStore level.

1. When a MemStore reaches the size specified by `hbase.hregion.memstore.flush.size`, all MemStores that belong to its region will be flushed out to disk.
2. When the overall MemStore usage reaches the value specified by `hbase.regionserver.global.memstore.upperLimit`, MemStores from various regions will be flushed out to disk to reduce overall MemStore usage in a RegionServer.
   The flush order is based on the descending order of a region's MemStore usage. Regions will have their MemStores flushed until the overall MemStore usage drops to or slightly below `hbase.regionserver.global.memstore.lowerLimit`.
3. When the number of WAL log entries in a given region server's WAL reaches the value specified in `hbase.regionserver.max.logs`, MemStores from various regions will be flushed out to disk to reduce the number of logs in the WAL.
   The flush order is based on time. Regions with the oldest MemStores are flushed first until WAL count drops below `hbase.regionserver.max.logs`.

Scans

- When a client issues a scan against a table, HBase generates `RegionScanner` objects, one per region, to serve the scan request.
- The `RegionScanner` object contains a list of `StoreScanner` objects, one per column family.
- Each `StoreScanner` object further contains a list of `StoreFileScanner` objects, corresponding to each StoreFile and HFile of the corresponding column family, and a list of `KeyValueScanner` objects for the MemStore.
- The two lists are merged into one, which is sorted in ascending order with the scan object for the MemStore at the end of the list.
- When a `StoreFileScanner` object is constructed, it is associated with a `MultiVersionConcurrencyControl` read point, which is the current `memstoreTS`, filtering out any new updates beyond the read point.

StoreFile (HFile)

StoreFiles are where your data lives.

HFile Format

The *HFile* file format is based on the SSTable file described in the BigTable [2006] paper and on Hadoop's TFile (The unit test suite and the compression harness were taken directly from TFile). Schubert Zhang's blog post on HFile: A Block-Indexed File Format to Store Sorted Key-Value Pairs makes for a thorough introduction to HBase's HFile. Matteo Bertozzi has also put up a helpful description, HBase I/O: HFile.

For more information, see the HFile source code. Also see HBase file format with inline blocks (version 2) for information about the HFile v2 format that was included in 0.92.

HFile Tool

To view a textualized version of HFile content, you can use the `hbase hfile` tool. Type the following to see usage:

```
$ ${HBASE_HOME}/bin/hbase hfile
```

For example, to view the content of the file *hdfs://10.81.47.41:9000/hbase/default/TEST/1418428042/DSMP/4759508618286845475*, type the following:

```
$ ${HBASE_HOME}/bin/hbase hfile -v -f hdfs://10.81.47.41:9000/hbase/default/TEST/
1418428042/DSMP/4759508618286845475
```

If you leave off the option -v to see just a summary on the HFile. See usage for other things to do with the `hfile` tool.

> In the output of this tool, you might see 'seqid=0' for certain keys in places such as 'Mid-key'/'firstKey'/'lastKey'. These are 'KeyOnlyKeyValue' type instances - meaning their seqid is irrelevant & we just need the keys of these Key-Value instances.

StoreFile Directory Structure on HDFS

For more information of what StoreFiles look like on HDFS with respect to the directory structure, see Browsing HDFS for HBase Objects.

Blocks

StoreFiles are composed of blocks. The blocksize is configured on a per-ColumnFamily basis.

Compression happens at the block level within StoreFiles. For more information on compression, see **Compression and Data Block Encoding In HBase**.

For more information on blocks, see the HFileBlock source code.

KeyValue

The KeyValue class is the heart of data storage in HBase. KeyValue wraps a byte array and takes offsets and lengths into the passed array which specify where to start interpreting the content as KeyValue.

The KeyValue format inside a byte array is:

- keylength
- valuelength
- key
- value

The Key is further decomposed as:

- rowlength
- row (i.e., the rowkey)
- columnfamilylength
- columnfamily
- columnqualifier
- timestamp
- keytype (e.g., Put, Delete, DeleteColumn, DeleteFamily)

KeyValue instances are *not* split across blocks. For example, if there is an 8 MB KeyValue, even if the block-size is 64kb this KeyValue will be read in as a coherent block. For more information, see the KeyValue source code.

Example

To emphasize the points above, examine what happens with two Puts for two different columns for the same row:

- Put #1: `rowkey=row1, cf:attr1=value1`
- Put #2: `rowkey=row1, cf:attr2=value2`

Even though these are for the same row, a KeyValue is created for each column:

Key portion for Put #1:

- `rowlength ------------→ 4`
- `row ----------------→ row1`
- `columnfamilylength --→ 2`
- `columnfamily --------→ cf`
- `columnqualifier -----→ attr1`
- `timestamp -----------→ server time of Put`
- `keytype -------------→ Put`

Key portion for Put #2:

- `rowlength ------------→ 4`
- `row ----------------→ row1`
- `columnfamilylength --→ 2`
- `columnfamily --------→ cf`
- `columnqualifier -----→ attr2`
- `timestamp -----------→ server time of Put`
- `keytype -------------→ Put`

It is critical to understand that the rowkey, ColumnFamily, and column (aka columnqualifier) are embedded within the KeyValue instance. The longer these identifiers are, the bigger the KeyValue is.

Compaction

Ambiguous Terminology:

- A *StoreFile* is a facade of HFile. In terms of compaction, use of StoreFile seems to have prevailed in the past.
- A *Store* is the same thing as a ColumnFamily. StoreFiles are related to a Store, or ColumnFamily.
- If you want to read more about StoreFiles versus HFiles and Stores versus ColumnFamilies, see HBASE-11316.

When the MemStore reaches a given size (`hbase.hregion.memstore.flush.size`), it flushes its contents to a StoreFile. The number of StoreFiles in a Store increases over time. *Compaction* is an operation which reduces the number of StoreFiles in a Store, by merging them together, in order to increase performance on read operations. Compactions can be resource-intensive to perform, and can either help or hinder performance depending on many factors.

Compactions fall into two categories: minor and major. Minor and major compactions differ in the following ways.

Minor compactions usually select a small number of small, adjacent StoreFiles and rewrite them as a single StoreFile. Minor compactions do not drop (filter out) deletes or expired versions, because of potential side effects. See Compaction and Deletions and Compaction and Versions for information on how deletes and versions are handled in relation to compactions. The end result of a minor compaction is fewer, larger StoreFiles for a given Store.

The end result of a *major compaction* is a single StoreFile per Store. Major compactions also process delete markers and max versions. See Compaction and Deletions and Compaction and Versions for information on how deletes and versions are handled in relation to compactions.

Compaction and Deletions

When an explicit deletion occurs in HBase, the data is not actually deleted. Instead, a *tombstone* marker is written. The tombstone marker prevents the data from being returned with queries. During a major compaction, the data is actually deleted, and the tombstone marker is removed from the StoreFile. If the deletion happens because of an expired TTL,

no tombstone is created. Instead, the expired data is filtered out and is not written back to the compacted StoreFile.

Compaction and Versions

When you create a Column Family, you can specify the maximum number of versions to keep, by specifying `ColumnFamilyDescriptorBuilder.setMaxVersions(int versions)`. The default value is `1`. If more versions than the specified maximum exist, the excess versions are filtered out and not written back to the compacted StoreFile.

> In some situations, older versions can be inadvertently resurrected if a newer version is explicitly deleted. See Major compactions change query results for a more in-depth explanation. This situation is only possible before the compaction finishes.

In theory, major compactions improve performance. However, on a highly loaded system, major compactions can require an inappropriate number of resources and adversely affect performance. In a default configuration, major compactions are scheduled automatically to run once in a 7-day period. This is sometimes inappropriate for systems in production. You can manage major compactions manually. See Managed Compactions.

Compactions do not perform region merges. See Merge for more information on region merging.

Compaction Switch

We can switch on and off the compactions at region servers. Switching off compactions will also interrupt any currently ongoing compactions. It can be done dynamically using the "compaction_switch" command from hbase shell. If done from the command line, this setting will be lost on restart of the server. To persist the changes across region servers modify the configuration hbase.regionserver .compaction.enabled in hbase-site.xml and restart HBase.

Compaction Policy - HBase 0.96.x and newer

Compacting large StoreFiles, or too many StoreFiles at once, can cause more IO load than your cluster is able to handle without causing performance problems. The method by which HBase selects which StoreFiles to include in a compaction (and whether the compaction is a minor or major compaction) is called the *compaction policy*.

Prior to HBase 0.96.x, there was only one compaction policy. That original compaction policy is still available as `RatioBasedCompactionPolicy`. The new compaction default policy,

called `ExploringCompactionPolicy`, was subsequently backported to HBase 0.94 and HBase 0.95, and is the default in HBase 0.96 and newer. It was implemented in HBASE-7842. In short, `ExploringCompactionPolicy` attempts to select the best possible set of StoreFiles to compact with the least amount of work, while the `RatioBasedCompactionPolicy` selects the first set that meets the criteria.

Regardless of the compaction policy used, file selection is controlled by several configurable parameters and happens in a multi-step approach. These parameters will be explained in context, and then will be given in a table which shows their descriptions, defaults, and implications of changing them.

Being Stuck

When the MemStore gets too large, it needs to flush its contents to a StoreFile. However, Stores are configured with a bound on the number StoreFiles, `hbase.hstore.blockingStoreFiles`, and if in excess, the MemStore flush must wait until the StoreFile count is reduced by one or more compactions. If the MemStore is too large and the number of StoreFiles is also too high, the algorithm is said to be "stuck". By default we'll wait on compactions up to `hbase.hstore.blockingWaitTime` milliseconds. If this period expires, we'll flush anyways even though we are in excess of the `hbase.hstore.blockingStoreFiles` count.

Upping the `hbase.hstore.blockingStoreFiles` count will allow flushes to happen but a Store with many StoreFiles in will likely have higher read latencies. Try to figure why Compactions are not keeping up. Is it a write spurt that is bringing about this situation or is a regular occurance and the cluster is under-provisioned for the volume of writes?

The ExploringCompactionPolicy Algorithm

The ExploringCompactionPolicy algorithm considers each possible set of adjacent StoreFiles before choosing the set where compaction will have the most benefit.

One situation where the ExploringCompactionPolicy works especially well is when you are bulk-loading data and the bulk loads create larger StoreFiles than the StoreFiles which are holding data older than the bulk-loaded data. This can "trick" HBase into choosing to perform a major compaction each time a compaction is needed, and cause a lot of extra overhead. With the ExploringCompactionPolicy, major compactions happen much less frequently because minor compactions are more efficient.

In general, ExploringCompactionPolicy is the right choice for most situations, and thus is the default compaction policy. You can also use ExploringCompactionPolicy along with Experimental: Stripe Compactions.

The logic of this policy can be examined in hbase-server/src/main/java/org/apache/hadoop/hbase/regionserver/compactions/ExploringCompactionPolicy.java. The following is a walk-through of the logic of the ExploringCompactionPolicy.

1. Make a list of all existing StoreFiles in the Store. The rest of the algorithm filters this list to come up with the subset of HFiles which will be chosen for compaction.

2. If this was a user-requested compaction, attempt to perform the requested compaction type, regardless of what would normally be chosen. Note that even if the user requests a major compaction, it may not be possible to perform a major compaction. This may be because not all StoreFiles in the Column Family are available to compact or because there are too many Stores in the Column Family.

3. Some StoreFiles are automatically excluded from consideration. These include:

   - StoreFiles that are larger than `hbase.hstore.compaction.max.size`
   - StoreFiles that were created by a bulk-load operation which explicitly excluded compaction. You may decide to exclude StoreFiles resulting from bulk loads, from compaction. To do this, specify the `hbase.mapreduce.hfileoutputformat.compaction.exclude` parameter during the bulk load operation.

4. Iterate through the list from step 1, and make a list of all potential sets of StoreFiles to compact together. A potential set is a grouping of `hbase.hstore.compaction.min` contiguous StoreFiles in the list. For each set, perform some sanity-checking and figure out whether this is the best compaction that could be done:

   - If the number of StoreFiles in this set (not the size of the StoreFiles) is fewer than `hbase.hstore.compaction.min` or more than `hbase.hstore.compaction.max`, take it out of consideration.

   - Compare the size of this set of StoreFiles with the size of the smallest possible compaction that has been found in the list so far. If the size of this set of StoreFiles represents the smallest compaction that could be done, store it to be used as a fall-

back if the algorithm is "stuck" and no StoreFiles would otherwise be chosen. See Being Stuck.

- Do size-based sanity checks against each StoreFile in this set of StoreFiles.
  - If the size of this StoreFile is larger than `hbase.hstore.compaction.max.size`, take it out of consideration.
  - If the size is greater than or equal to `hbase.hstore.compaction.min.size`, sanity-check it against the file-based ratio to see whether it is too large to be considered.

  The sanity-checking is successful if:

  - There is only one StoreFile in this set, or
  - For each StoreFile, its size multiplied by `hbase.hstore.compaction.ratio` (or `hbase.hstore.compaction.ratio.offpeak` if off-peak hours are configured and it is during off-peak hours) is less than the sum of the sizes of the other HFiles in the set.

5. If this set of StoreFiles is still in consideration, compare it to the previously-selected best compaction. If it is better, replace the previously-selected best compaction with this one.

6. When the entire list of potential compactions has been processed, perform the best compaction that was found. If no StoreFiles were selected for compaction, but there are multiple StoreFiles, assume the algorithm is stuck (see Being Stuck) and if so, perform the smallest compaction that was found in step 3.

RatioBasedCompactionPolicy Algorithm

The RatioBasedCompactionPolicy was the only compaction policy prior to HBase 0.96, though ExploringCompactionPolicy has now been backported to HBase 0.94 and 0.95. To use the RatioBasedCompactionPolicy rather than the ExploringCompactionPolicy, set `hbase.hstore.defaultengine.compactionpolicy.class` to `RatioBasedCompactionPolicy` in the *hbase-site.xml* file. To switch back to the ExploringCompactionPolicy, remove the setting from the *hbase-site.xml*.

The following section walks you through the algorithm used to select StoreFiles for compaction in the RatioBasedCompactionPolicy.

1. The first phase is to create a list of all candidates for compaction. A list is created of all StoreFiles not already in the compaction queue, and all StoreFiles newer than the newest file that is currently being compacted. This list of StoreFiles is ordered by the sequence ID. The sequence ID is generated when a Put is appended to the write-ahead log (WAL), and is stored in the metadata of the HFile.
2. Check to see if the algorithm is stuck (see **Being Stuck**, and if so, a major compaction is forced. This is a key area where **The ExploringCompactionPolicy Algorithm** is often a better choice than the RatioBasedCompactionPolicy.
3. If the compaction was user-requested, try to perform the type of compaction that was requested. Note that a major compaction may not be possible if all HFiles are not available for compaction or if too many StoreFiles exist (more than `hbase.hstore.compaction.max`).
4. Some StoreFiles are automatically excluded from consideration. These include:
   - StoreFiles that are larger than `hbase.hstore.compaction.max.size`
   - StoreFiles that were created by a bulk-load operation which explicitly excluded compaction. You may decide to exclude StoreFiles resulting from bulk loads, from compaction. To do this, specify the `hbase.mapreduce.hfileoutputformat.compaction.exclude` parameter during the bulk load operation.
5. The maximum number of StoreFiles allowed in a major compaction is controlled by the `hbase.hstore.compaction.max` parameter. If the list contains more than this number of StoreFiles, a minor compaction is performed even if a major compaction would otherwise have been done. However, a user-requested major compaction still occurs even if there are more than `hbase.hstore.compaction.max` StoreFiles to compact.
6. If the list contains fewer than `hbase.hstore.compaction.min` StoreFiles to compact, a minor compaction is aborted. Note that a major compaction can be performed on a single HFile. Its function is to remove deletes and expired versions, and reset locality on the StoreFile.
7. The value of the `hbase.hstore.compaction.ratio` parameter is multiplied by the sum of StoreFiles smaller than a given file, to determine whether that StoreFile is selected for compaction during a minor compaction. For instance, if hbase.hstore.compaction.ratio is 1.2, FileX is 5MB, FileY is 2MB, and FileZ is 3MB:

```
5 <= 1.2 x (2 + 3)           or           5 <= 6
```

In this scenario, FileX is eligible for minor compaction. If FileX were 7MB, it would not be eligible for minor compaction. This ratio favors smaller StoreFile. You can configure a different ratio for use in off-peak hours, using the parameter `hbase.hstore.compaction.ratio.offpeak`, if you also configure `hbase.offpeak.start.hour` and `hbase.offpeak.end.hour`.

8. If the last major compaction was too long ago and there is more than one StoreFile to be compacted, a major compaction is run, even if it would otherwise have been minor. By default, the maximum time between major compactions is 7 days, plus or minus a 4.8 hour period, and determined randomly within those parameters. Prior to HBase 0.96, the major compaction period was 24 hours. See `hbase.hregion.majorcompaction` in the table below to tune or disable time-based major compactions.

Parameters Used by Compaction Algorithm

This table contains the main configuration parameters for compaction. This list is not exhaustive. To tune these parameters from the defaults, edit the *hbase-default.xml* file. For a full list of all configuration parameters available, see config.files

- `hbase.hstore.compaction.min`

  The minimum number of StoreFiles which must be eligible for compaction before compaction can run. The goal of tuning `hbase.hstore.compaction.min` is to avoid ending up with too many tiny StoreFiles to compact. Setting this value to 2 would cause a minor compaction each time you have two StoreFiles in a Store, and this is probably not appropriate. If you set this value too high, all the other values will need to be adjusted accordingly. For most cases, the default value is appropriate. In previous versions of HBase, the parameter `hbase.hstore.compaction.min` was called `hbase.hstore.compactionThreshold`.

  Default: 3

- `hbase.hstore.compaction.max` The maximum number of StoreFiles which will be selected for a single minor compaction, regardless of the number of eligible StoreFiles. Effectively, the value of `hbase.hstore.compaction.max` controls the length of time it takes a single compaction to complete. Setting it larger means that more StoreFiles are included in a compaction. For most cases, the default value is appropriate.

  Default: 10

- `hbase.hstore.compaction.min.size`

  A StoreFile smaller than this size will always be eligible for minor compaction. StoreFiles this size or larger are evaluated by `hbase.hstore.compaction.ratio` to determine if they are eligible. Because this limit represents the "automatic include" limit for all StoreFiles smaller than this value, this value may need to be reduced in write-heavy environments where many files in the 1-2 MB range are being flushed, because every StoreFile will be targeted for compaction and the resulting StoreFiles may still be under the minimum size and require further compaction. If this parameter is lowered, the ratio check is triggered more quickly. This addressed some issues seen in earlier versions of HBase but changing this parameter is no longer necessary in most situations.
  Default:128 MB

- `hbase.hstore.compaction.max.size`

  A StoreFile larger than this size will be excluded from compaction. The effect of raising `hbase.hstore.compaction.max.size` is fewer, larger StoreFiles that do not get compacted often. If you feel that compaction is happening too often without much benefit, you can try raising this value.
  Default: `Long.MAX_VALUE`

- `hbase.hstore.compaction.ratio`

  For minor compaction, this ratio is used to determine whether a given StoreFile which is larger than `hbase.hstore.compaction.min.size` is eligible for compaction. Its effect is to limit compaction of large StoreFile. The value of `hbase.hstore.compaction.ratio` is expressed as a floating-point decimal.

  - A large ratio, such as 10, will produce a single giant StoreFile. Conversely, a value of .25, will produce behavior similar to the BigTable compaction algorithm, producing four StoreFiles.
  - A moderate value of between 1.0 and 1.4 is recommended. When tuning this value, you are balancing write costs with read costs. Raising the value (to something like 1.4) will have more write costs, because you will compact larger StoreFiles. However, during reads, HBase will need to seek through fewer StoreFiles to accomplish the read. Consider this approach if you cannot take advantage of Bloom Filters.
  - Alternatively, you can lower this value to something like 1.0 to reduce the background cost of writes, and use to limit the number of StoreFiles touched during reads. For

most cases, the default value is appropriate.
Default: `1.2F`

- `hbase.hstore.compaction.ratio.offpeak`
  The compaction ratio used during off-peak compactions, if off-peak hours are also configured (see below). Expressed as a floating-point decimal. This allows for more aggressive (or less aggressive, if you set it lower than `hbase.hstore.compaction.ratio`) compaction during a set time period. Ignored if off-peak is disabled (default). This works the same as `hbase.hstore.compaction.ratio`.
  Default: `5.0F`

- `hbase.offpeak.start.hour`
  The start of off-peak hours, expressed as an integer between 0 and 23, inclusive. Set to -1 to disable off-peak.
  Default: `-1` (disabled)

- `hbase.offpeak.end.hour`
  The end of off-peak hours, expressed as an integer between 0 and 23, inclusive. Set to -1 to disable off-peak.
  Default: `-1` (disabled)

- `hbase.regionserver.thread.compaction.throttle`
  There are two different thread pools for compactions, one for large compactions and the other for small compactions. This helps to keep compaction of lean tables (such as `hbase:meta`) fast. If a compaction is larger than this threshold, it goes into the large compaction pool. In most cases, the default value is appropriate.
  Default: `2 x hbase.hstore.compaction.max x hbase.hregion.memstore.flush.size` (which defaults to `128`)

- `hbase.hregion.majorcompaction`
  Time between major compactions, expressed in milliseconds. Set to 0 to disable time-based automatic major compactions. User-requested and size-based major compactions will still run. This value is multiplied by `hbase.hregion.majorcompaction.jitter` to cause compaction to start at a somewhat-random time during a given window of time.
  Default: 7 days (`604800000` milliseconds)

- `hbase.hregion.majorcompaction.jitter`
  A multiplier applied to hbase.hregion.majorcompaction to cause compaction to occur a given amount of time either side of `hbase.hregion.majorcompaction`. The smaller the number, the closer the compactions will happen to the `hbase.hregion.majorcompaction` interval. Expressed as a floating-point decimal.
  Default: `.50F`

Compaction File Selection

> **Legacy Information**
>
> This section has been preserved for historical reasons and refers to the way compaction worked prior to HBase 0.96.x. You can still use this behavior if you enable RatioBasedCompactionPolicy Algorithm. For information on the way that compactions work in HBase 0.96.x and later, see Compaction.

To understand the core algorithm for StoreFile selection, there is some ASCII-art in the Store source code that will serve as useful reference.

It has been copied below:

```
/* normal skew:
 *
 *         older ----> newer
 *      _
 *     | |   _
 *     | |  | |   _
 *   --|-|- |-|- |-|---_-------_-------  minCompactSize
 *     | |  | |  | |  | |  _  | |
 *     | |  | |  | |  | | | | | |
 *     | |  | |  | |  | | | | | |
 */
```

Important knobs:

- `hbase.hstore.compaction.ratio` Ratio used in compaction file selection algorithm (default 1.2f).
- `hbase.hstore.compaction.min` (in HBase v 0.90 this is called `hbase.hstore.compactionThreshold`) (files) Minimum number of StoreFiles per Store to be selected for a compaction to occur (default 2).
- `hbase.hstore.compaction.max` (files) Maximum number of StoreFiles to compact per minor compaction (default 10).

- `hbase.hstore.compaction.min.size` (bytes) Any StoreFile smaller than this setting with automatically be a candidate for compaction. Defaults to `hbase.hregion.memstore.flush.size` (128 mb).
- `hbase.hstore.compaction.max.size` (.92) (bytes) Any StoreFile larger than this setting with automatically be excluded from compaction (default Long.MAX_VALUE).

The minor compaction StoreFile selection logic is size based, and selects a file for compaction when the `file ⇐ sum(smaller_files) * hbase.hstore.compaction.ratio`.

Minor Compaction File Selection - Example #1 (Basic Example)

This example mirrors an example from the unit test `TestCompactSelection`.

- `hbase.hstore.compaction.ratio` = 1.0f
- `hbase.hstore.compaction.min` = 3 (files)
- `hbase.hstore.compaction.max` = 5 (files)
- `hbase.hstore.compaction.min.size` = 10 (bytes)
- `hbase.hstore.compaction.max.size` = 1000 (bytes)

The following StoreFiles exist: 100, 50, 23, 12, and 12 bytes apiece (oldest to newest). With the above parameters, the files that would be selected for minor compaction are 23, 12, and 12.

Why?

- 100 → No, because sum(50, 23, 12, 12) * 1.0 = 97.
- 50 → No, because sum(23, 12, 12) * 1.0 = 47.
- 23 → Yes, because sum(12, 12) * 1.0 = 24.
- 12 → Yes, because the previous file has been included, and because this does not exceed the max-file limit of 5
- 12 → Yes, because the previous file had been included, and because this does not exceed the max-file limit of 5.

Minor Compaction File Selection - Example #2 (Not Enough Files ToCompact)

This example mirrors an example from the unit test `TestCompactSelection`.

- `hbase.hstore.compaction.ratio` = 1.0f
- `hbase.hstore.compaction.min` = 3 (files)
- `hbase.hstore.compaction.max` = 5 (files)
- `hbase.hstore.compaction.min.size` = 10 (bytes)
- `hbase.hstore.compaction.max.size` = 1000 (bytes)

The following StoreFiles exist: 100, 25, 12, and 12 bytes apiece (oldest to newest). With the above parameters, no compaction will be started.

Why?

- 100 → No, because sum(25, 12, 12) * 1.0 = 47
- 25 → No, because sum(12, 12) * 1.0 = 24
- 12 → No. Candidate because sum(12) * 1.0 = 12, there are only 2 files to compact and that is less than the threshold of 3
- 12 → No. Candidate because the previous StoreFile was, but there are not enough files to compact

Minor Compaction File Selection - Example #3 (Limiting Files To Compact)

This example mirrors an example from the unit test `TestCompactSelection`.

- `hbase.hstore.compaction.ratio` = 1.0f
- `hbase.hstore.compaction.min` = 3 (files)
- `hbase.hstore.compaction.max` = 5 (files)
- `hbase.hstore.compaction.min.size` = 10 (bytes)
- `hbase.hstore.compaction.max.size` = 1000 (bytes)

The following StoreFiles exist: 7, 6, 5, 4, 3, 2, and 1 bytes apiece (oldest to newest). With the above parameters, the files that would be selected for minor compaction are 7, 6, 5, 4, 3.

Why?

- 7 → Yes, because sum(6, 5, 4, 3, 2, 1) * 1.0 = 21. Also, 7 is less than the min-size

- 6 → Yes, because sum(5, 4, 3, 2, 1) * 1.0 = 15. Also, 6 is less than the min-size.
- 5 → Yes, because sum(4, 3, 2, 1) * 1.0 = 10. Also, 5 is less than the min-size.
- 4 → Yes, because sum(3, 2, 1) * 1.0 = 6. Also, 4 is less than the min-size.
- 3 → Yes, because sum(2, 1) * 1.0 = 3. Also, 3 is less than the min-size.
- 2 → No. Candidate because previous file was selected and 2 is less than the min-size, but the max-number of files to compact has been reached.
- 1 → No. Candidate because previous file was selected and 1 is less than the min-size, but max-number of files to compact has been reached.

> **Impact of Key Configuration Options**
>
> This information is now included in the configuration parameter table in Parameters Used by Compaction Algorithm.

Date Tiered Compaction

Date tiered compaction is a date-aware store file compaction strategy that is beneficial for time-range scans for time-series data.

When To Use Date Tiered Compactions

Consider using Date Tiered Compaction for reads for limited time ranges, especially scans of recent data

Don't use it for

- random gets without a limited time range
- frequent deletes and updates
- Frequent out of order data writes creating long tails, especially writes with future timestamps
- frequent bulk loads with heavily overlapping time ranges

Performance Improvements

Performance testing has shown that the performance of time-range scans improve greatly for limited time ranges, especially scans of recent data.

Enabling Date Tiered Compaction

You can enable Date Tiered compaction for a table or a column family, by setting its `hbase.hstore.engine.class` to `org.apache.hadoop.hbase.regionserver.DateTieredStoreEngine`.

You also need to set `hbase.hstore.blockingStoreFiles` to a high number, such as 60, if using all default settings, rather than the default value of 12). Use 1.5~2 x projected file count if changing the parameters, Projected file count = windows per tier x tier count + incoming window min + files older than max age

You also need to set `hbase.hstore.compaction.max` to the same value as `hbase.hstore.blockingStoreFiles` to unblock major compaction.

Procedure: Enable Date Tiered Compaction

1. Run one of following commands in the HBase shell. Replace the table name `orders_table` with the name of your table.

```
alter 'orders_table', CONFIGURATION => {'hbase.hstore.engine.class' => 'org.
apache.hadoop.hbase.regionserver.DateTieredStoreEngine', 'hbase.hstore.block
ingStoreFiles' => '60', 'hbase.hstore.compaction.min'=>'2', 'hbase.hstore.co
mpaction.max'=>'60'}
alter 'orders_table', {NAME => 'blobs_cf', CONFIGURATION => {'hbase.hstore.e
ngine.class' => 'org.apache.hadoop.hbase.regionserver.DateTieredStoreEngin
e', 'hbase.hstore.blockingStoreFiles' => '60', 'hbase.hstore.compaction.mi
n'=>'2', 'hbase.hstore.compaction.max'=>'60'}}
create 'orders_table', 'blobs_cf', CONFIGURATION => {'hbase.hstore.engine.cl
ass' => 'org.apache.hadoop.hbase.regionserver.DateTieredStoreEngine', 'hbas
e.hstore.blockingStoreFiles' => '60', 'hbase.hstore.compaction.min'=>'2', 'h
base.hstore.compaction.max'=>'60'}
```

2. Configure other options if needed. See Configuring Date Tiered Compaction for more information.

Procedure: Disable Date Tiered Compaction

1. Set the `hbase.hstore.engine.class` option to either nil or `org.apache.hadoop.hbase.regionserver.DefaultStoreEngine`. Either option has the same effect. Make sure you set the other options you changed to the original settings too.

```
alter 'orders_table', CONFIGURATION => {'hbase.hstore.engine.class' => 'org.
apache.hadoop.hbase.regionserver.DefaultStoreEngine', 'hbase.hstore.blocking
StoreFiles' => '12', 'hbase.hstore.compaction.min'=>'6', 'hbase.hstore.compa
```

```
ction.max'=>'12'}
```

When you change the store engine either way, a major compaction will likely be performed on most regions. This is not necessary on new tables.

Configuring Date Tiered Compaction

Each of the settings for date tiered compaction should be configured at the table or column family level. If you use HBase shell, the general command pattern is as follows:

```
alter 'orders_table', CONFIGURATION => {'key' => 'value', ..., 'key' => 'value'}}
```

Data Tier Parameters

You can configure your date tiers by changing the settings for the following parameters:

Setting	Notes
`hbase.hstore.compaction.date.tiered.max.storefile.age.millis`	Files with max-timestamp smaller than this will no longer be compacted.Default at Long.MAX_VALUE.
`hbase.hstore.compaction.date.tiered.base.window.millis`	Base window size in milliseconds. Default at 6 hours.
`hbase.hstore.compaction.date.tiered.windows.per.tier`	Number of windows per tier. Default at 4.
`hbase.hstore.compaction.date.tiered.incoming.window.min`	Minimal number of files to compact in the incoming window. Set it to expected number of files in the window to avoid wasteful compaction. Default at 6.
`hbase.hstore.compaction.date.tiered.window.policy.class`	The policy to select store files within the same time window. It doesn't apply to the incoming window. Default at exploring compaction. This is to avoid wasteful compaction.

Compaction Throttler

With tiered compaction all servers in the cluster will promote windows to higher tier at the same time, so using a compaction throttle is recommended: Set `hbase.regionserver.throughput.controller` to `org.apache.hadoop.hbase.regionserver.compactions.PressureAwareCompactionThroughputController`.

For more information about date tiered compaction, please refer to the design specification at https://docs.google.com/document/d/1_AmlNb2N8Us1xICsTeGDLKIqL6T-oHoRLZ323MG_uy8

Experimental: Stripe Compactions

Stripe compactions is an experimental feature added in HBase 0.98 which aims to improve compactions for large regions or non-uniformly distributed row keys. In order to achieve smaller and/or more granular compactions, the StoreFiles within a region are maintained separately for several row-key sub-ranges, or "stripes", of the region. The stripes are transparent to the rest of HBase, so other operations on the HFiles or data work without modification.

Stripe compactions change the HFile layout, creating sub-regions within regions. These sub-regions are easier to compact, and should result in fewer major compactions. This approach alleviates some of the challenges of larger regions.

Stripe compaction is fully compatible with Compaction and works in conjunction with either the ExploringCompactionPolicy or RatioBasedCompactionPolicy. It can be enabled for existing tables, and the table will continue to operate normally if it is disabled later.

When To Use Stripe Compactions

Consider using stripe compaction if you have either of the following:

- Large regions. You can get the positive effects of smaller regions without additional overhead for MemStore and region management overhead.
- Non-uniform keys, such as time dimension in a key. Only the stripes receiving the new keys will need to compact. Old data will not compact as often, if at all

Performance Improvements

Performance testing has shown that the performance of reads improves somewhat, and variability of performance of reads and writes is greatly reduced. An overall long-term performance improvement is seen on large non-uniform-row key regions, such as a hash-prefixed timestamp key. These performance gains are the most dramatic on a table which is already large. It is possible that the performance improvement might extend to region splits.

Enabling Stripe Compaction

You can enable stripe compaction for a table or a column family, by setting its `hbase.hstore.engine.class` to `org.apache.hadoop.hbase.regionserver.StripeStoreEngine`. You also need to set the `hbase.hstore.blockingStoreFiles` to a high number, such as 100 (rather than the default value of 10).

Procedure: Enable Stripe Compaction

1. Run one of following commands in the HBase shell. Replace the table name `orders_table` with the name of your table.

```
alter 'orders_table', CONFIGURATION => {'hbase.hstore.engine.class' => 'org.apache.hadoop.hbase.regionserver.StripeStoreEngine', 'hbase.hstore.blockingStoreFiles' => '100'}
alter 'orders_table', {NAME => 'blobs_cf', CONFIGURATION => {'hbase.hstore.engine.class' => 'org.apache.hadoop.hbase.regionserver.StripeStoreEngine', 'hbase.hstore.blockingStoreFiles' => '100'}}
create 'orders_table', 'blobs_cf', CONFIGURATION => {'hbase.hstore.engine.class' => 'org.apache.hadoop.hbase.regionserver.StripeStoreEngine', 'hbase.hstore.blockingStoreFiles' => '100'}
```

2. Configure other options if needed. See Configuring Stripe Compaction for more information.
3. Enable the table.

Procedure: Disable Stripe Compaction

1. Set the `hbase.hstore.engine.class` option to either nil or `org.apache.hadoop.hbase.regionserver.DefaultStoreEngine`. Either option has the same effect.

```
alter 'orders_table', CONFIGURATION => {'hbase.hstore.engine.class' => 'org.apache.hadoop.hbase.regionserver.DefaultStoreEngine'}
```

2. Enable the table.

When you enable a large table after changing the store engine either way, a major compaction will likely be performed on most regions. This is not necessary on new tables.

Configuring Stripe Compaction

Each of the settings for stripe compaction should be configured at the table or column family level. If you use HBase shell, the general command pattern is as follows:

```
alter 'orders_table', CONFIGURATION => {'key' => 'value', ..., 'key' => 'value'}}
```

Region and stripe sizing

You can configure your stripe sizing based upon your region sizing. By default, your new regions will start with one stripe. On the next compaction after the stripe has grown too large (16 x MemStore flushes size), it is split into two stripes. Stripe splitting continues as the region grows, until the region is large enough to split.

You can improve this pattern for your own data. A good rule is to aim for a stripe size of at least 1 GB, and about 8-12 stripes for uniform row keys. For example, if your regions are 30 GB, 12 × 2.5 GB stripes might be a good starting point.

Stripe Sizing Settings

Setting	Notes
`hbase.store.stripe.initialStripeCount`	The number of stripes to create when stripe compaction is enabled. You can use it as follows: - For relatively uniform row keys, if you know the approximate target number of stripes from the above, you can avoid some splitting overhead by starting with several stripes (2, 5, 10...). If the early data is not representative of overall row key distribution, this will not be as efficient. - For existing tables with a large amount of data, this setting will effectively pre-split your stripes. - For keys such as hash-prefixed sequential keys, with more than one hash prefix per region, pre-splitting may make sense.
`hbase.store.stripe.sizeToSplit`	The maximum size a stripe grows before splitting. Use this in conjunction with `hbase.store.stripe.splitPartCount` to control the target stripe size (`sizeToSplit = splitPartsCount * target stripe size`), according to the above sizing considerations.

Setting	Notes
`hbase.store.stripe.splitPartCount`	The number of new stripes to create when splitting a stripe. The default is 2, which is appropriate for most cases. For non-uniform row keys, you can experiment with increasing the number to 3 or 4, to isolate the arriving updates into narrower slice of the region without additional splits being required.

MemStore Size Settings

By default, the flush creates several files from one MemStore, according to existing stripe boundaries and row keys to flush. This approach minimizes write amplification, but can be undesirable if the MemStore is small and there are many stripes, because the files will be too small.

In this type of situation, you can set `hbase.store.stripe.compaction.flushToL0` to `true`. This will cause a MemStore flush to create a single file instead. When at least `hbase.store.stripe.compaction.minFilesL0` such files (by default, 4) accumulate, they will be compacted into striped files.

Normal Compaction Configuration and Stripe Compaction

All the settings that apply to normal compactions (see Parameters Used by Compaction Algorithm) apply to stripe compactions. The exceptions are the minimum and maximum number of files, which are set to higher values by default because the files in stripes are smaller. To control these for stripe compactions, use `hbase.store.stripe.compaction.minFiles` and `hbase.store.stripe.compaction.maxFiles`, rather than `hbase.hstore.compaction.min` and `hbase.hstore.compaction.max`.

FIFO Compaction

FIFO compaction policy selects only files which have all cells expired. The column family **MUST** have non-default TTL. Essentially, FIFO compactor only collects expired store files.

Because we don't do any real compaction, we do not use CPU and IO (disk and network) and evict hot data from a block cache. As a result, both RW throughput and latency can be improved.

When To Use FIFO Compaction

Consider using FIFO Compaction when your use case is

- Very high volume raw data which has low TTL and which is the source of another data (after additional processing).
- Data which can be kept entirely in a a block cache (RAM/SSD). No need for compaction of a raw data at all.

Do not use FIFO compaction when

- Table/ColumnFamily has MIN_VERSION > 0
- Table/ColumnFamily has TTL = FOREVER (HColumnDescriptor.DEFAULT_TTL)

Enabling FIFO Compaction

For Table:

```
HTableDescriptor desc = new HTableDescriptor(tableName);
desc.setConfiguration(DefaultStoreEngine.DEFAULT_COMPACTION_POLICY_CLASS_KEY,
  FIFOCompactionPolicy.class.getName());
```

For Column Family:

```
HColumnDescriptor desc = new HColumnDescriptor(family);
desc.setConfiguration(DefaultStoreEngine.DEFAULT_COMPACTION_POLICY_CLASS_KEY,
  FIFOCompactionPolicy.class.getName());
```

From HBase Shell:

```
create 'x',{NAME=>'y', TTL=>'30'}, {CONFIGURATION => {'hbase.hstore.defaultengin
e.compactionpolicy.class' => 'org.apache.hadoop.hbase.regionserver.compactions.FI
FOCompactionPolicy', 'hbase.hstore.blockingStoreFiles' => 1000}}
```

Although region splitting is still supported, for optimal performance it should be disabled, either by setting explicitly `DisabledRegionSplitPolicy` or by setting `ConstantSizeRegionSplitPolicy` and very large max region size. You will have to increase to a very large number store's blocking file (`hbase.hstore.blockingStoreFiles`) as well. There is a sanity check on table/column family configuration in case of FIFO compaction and minimum value for number of blocking file is 1000.

Bulk Loading

Overview

HBase includes several methods of loading data into tables. The most straightforward method is to either use the `TableOutputFormat` class from a MapReduce job, or use the normal client APIs; however, these are not always the most efficient methods.

The bulk load feature uses a MapReduce job to output table data in HBase's internal data format, and then directly load the generated StoreFiles into a running cluster. Using bulk load will use less CPU and network resources than loading via the HBase API.

Bulk Load Architecture

The HBase bulk load process consists of two main steps.

Preparing data via a MapReduce job

The first step of a bulk load is to generate HBase data files (StoreFiles) from a MapReduce job using `HFileOutputFormat2`. This output format writes out data in HBase's internal storage format so that they can be later loaded efficiently into the cluster.

In order to function efficiently, `HFileOutputFormat2` must be configured such that each output HFile fits within a single region. In order to do this, jobs whose output will be bulk loaded into HBase use Hadoop's `TotalOrderPartitioner` class to partition the map output into disjoint ranges of the key space, corresponding to the key ranges of the regions in the table.

`HFileOutputFormat2` includes a convenience function, `configureIncrementalLoad()`, which automatically sets up a `TotalOrderPartitioner` based on the current region boundaries of a table.

Completing the data load

After a data import has been prepared, either by using the `importtsv` tool with the “importtsv.bulk.output” option or by some other MapReduce job using the `HFileOutputFormat`, the `completebulkload` tool is used to import the data into the running cluster. This

command line tool iterates through the prepared data files, and for each one determines the region the file belongs to. It then contacts the appropriate RegionServer which adopts the HFile, moving it into its storage directory and making the data available to clients.

If the region boundaries have changed during the course of bulk load preparation, or between the preparation and completion steps, the `completebulkload` utility will automatically split the data files into pieces corresponding to the new boundaries. This process is not optimally efficient, so users should take care to minimize the delay between preparing a bulk load and importing it into the cluster, especially if other clients are simultaneously loading data through other means.

```
$ hadoop jar hbase-mapreduce-VERSION.jar completebulkload [-c /path/to/hbase/conf
ig/hbase-site.xml] /user/todd/myoutput mytable
```

The `-c config-file` option can be used to specify a file containing the appropriate hbase parameters (e.g., hbase-site.xml) if not supplied already on the CLASSPATH (In addition, the CLASSPATH must contain the directory that has the zookeeper configuration file if zookeeper is NOT managed by HBase).

See Also

For more information about the referenced utilities, see ImportTsv and CompleteBulkLoad.

See How-to: Use HBase Bulk Loading, and Why for an old blog post on loading.

Advanced Usage

Although the `importtsv` tool is useful in many cases, advanced users may want to generate data programmatically, or import data from other formats. To get started doing so, dig into `ImportTsv.java` and check the JavaDoc for HFileOutputFormat.

The import step of the bulk load can also be done programmatically. See the `LoadIncrementalHFiles` class for more information.

'Adopting' Stray Data

Should an HBase cluster lose account of regions or files during an outage or error, you can use the `completebulkload` tool to add back the dropped data. HBase operator tooling such as HBCK2 or the reporting added to the Master's UI under the `HBCK Report` (Since HBase 2.0.6/2.1.6/2.2.1) can identify such 'orphan' directories.

Before you begin the 'adoption', ensure the `hbase:meta` table is in a healthy state. Run the `CatalogJanitor` by executing the `catalogjanitor_run` command on the HBase shell. When finished, check the `HBCK Report` page on the Master UI. Work on fixing any inconsistencies, holes, or overlaps found before proceeding. The `hbase:meta` table is the authority on where all data is to be found and must be consistent for the `completebulkload` tool to work properly.

The `completebulkload` tool takes a directory and a `tablename`. The directory has subdirectories named for column families of the targeted `tablename`. In these subdirectories are `hfiles` to load. Given this structure, you can pass errant region directories (and the table name to which the region directory belongs) and the tool will bring the data files back into the fold by moving them under the approprate serving directory. If stray files, then you will need to mock up this structure before invoking the `co mpletebulkload` tool; you may have to look at the file content using the HFile Tool to see what the column family to use is. When the tool completes its run, you will notice that the source errant directory has had its storefiles moved/removed. It is now desiccated since its data has been drained, and the pointed-to directory can be safely removed. It may still have `.regioninfo` files and other subdirectories but they are of no relevance now (There may be content still under the *recovered_edits* directory; a TODO is tooling to replay the content of *recovered_edits* if needed; see Add RecoveredEditsPlayer). If you pass `complet ebulkload` a directory without store files, it will run and note the directory is storefile-free. Just remove such 'empty' directories.

For example, presuming a directory at the top level in HDFS named `eb3352fb5c9c9a05feeb2 caba101e1cc` has data we need to re-add to the HBase `TestTable`:

```
$ ${HBASE_HOME}/bin/hbase --config ~/hbase-conf completebulkload hdfs://server.ex
ample.org:9000/eb3352fb5c9c9a05feeb2caba101e1cc TestTable
```

After it successfully completes, any files that were in `eb3352fb5c9c9a05feeb2caba101e1cc` have been moved under hbase and the `eb3352fb5c9c9a05feeb2caba101e1cc` directory can be deleted (Check content before and after by running `ls -r` on the HDFS directory).

Bulk Loading Replication

HBASE-13153 adds replication support for bulk loaded HFiles, available since HBase 1.3/2.0. This feature is enabled by setting `hbase.replication.bulkload.enabled` to `true` (default is `false`). You also need to copy the source cluster configuration files to the destination cluster.

Additional configurations are required too:

1. `hbase.replication.source.fs.conf.provider`
   This defines the class which loads the source cluster file system client configuration in the destination cluster. This should be configured for all the RS in the destination cluster. Default is `org.apache.hadoop.hbase.replication.regionserver.DefaultSourceFSConfigurationProvider`.
2. `hbase.replication.conf.dir`
   This represents the base directory where the file system client configurations of the source cluster are copied to the destination cluster. This should be configured for all the RS in the destination cluster. Default is `$HBASE_CONF_DIR`.
3. `hbase.replication.cluster.id`
   This configuration is required in the cluster where replication for bulk loaded data is enabled. A source cluster is uniquely identified by the destination cluster using this id. This should be configured for all the RS in the source cluster configuration file for all the RS.

For example: If source cluster FS client configurations are copied to the destination cluster under directory `/home/user/dc1/`, then `hbase.replication.cluster.id` should be configured as `dc1` and `hbase.replication.conf.dir` as `/home/user`.

> `DefaultSourceFSConfigurationProvider` supports only `xml` type files. It loads source cluster FS client configuration only once, so if source cluster FS client configuration files are updated, every peer(s) cluster RS must be restarted to reload the configuration.

HDFS

As HBase runs on HDFS (and each StoreFile is written as a file on HDFS), it is important to have an understanding of the HDFS Architecture especially in terms of how it stores files, handles failovers, and replicates blocks.

See the Hadoop documentation on HDFS Architecture for more information.

NameNode

The NameNode is responsible for maintaining the filesystem metadata. See the above HDFS Architecture link for more information.

DataNode

The DataNodes are responsible for storing HDFS blocks. See the above HDFS Architecture link for more information.

Timeline-consistent High Available Reads

Introduction

HBase, architecturally, always had the strong consistency guarantee from the start. All reads and writes are routed through a single region server, which guarantees that all writes happen in an order, and all reads are seeing the most recent committed data.

However, because of this single homing of the reads to a single location, if the server becomes unavailable, the regions of the table that were hosted in the region server become unavailable for some time. There are three phases in the region recovery process - detection, assignment, and recovery. Of these, the detection is usually the longest and is presently in the order of 20-30 seconds depending on the ZooKeeper session timeout. During this time and before the recovery is complete, the clients will not be able to read the region data.

However, for some use cases, either the data may be read-only, or doing reads against some stale data is acceptable. With timeline-consistent high available reads, HBase can be

used for these kind of latency-sensitive use cases where the application can expect to have a time bound on the read completion.

For achieving high availability for reads, HBase provides a feature called *region replication*. In this model, for each region of a table, there will be multiple replicas that are opened in different RegionServers. By default, the region replication is set to 1, so only a single region replica is deployed and there will not be any changes from the original model. If region replication is set to 2 or more, then the master will assign replicas of the regions of the table. The Load Balancer ensures that the region replicas are not co-hosted in the same region servers and also in the same rack (if possible).

All of the replicas for a single region will have a unique replica*id, starting from 0. The region replica having replica_id==0 is called the primary region, and the others _secondary regions* or secondaries. Only the primary can accept writes from the client, and the primary will always contain the latest changes. Since all writes still have to go through the primary region, the writes are not highly-available (meaning they might block for some time if the region becomes unavailable).

Timeline Consistency

With this feature, HBase introduces a Consistency definition, which can be provided per read operation (get or scan).

```
public enum Consistency {
    STRONG,
    TIMELINE
}
```

`Consistency.STRONG` is the default consistency model provided by HBase. In case the table has region replication = 1, or in a table with region replicas but the reads are done with this consistency, the read is always performed by the primary regions, so that there will not be any change from the previous behaviour, and the client always observes the latest data.

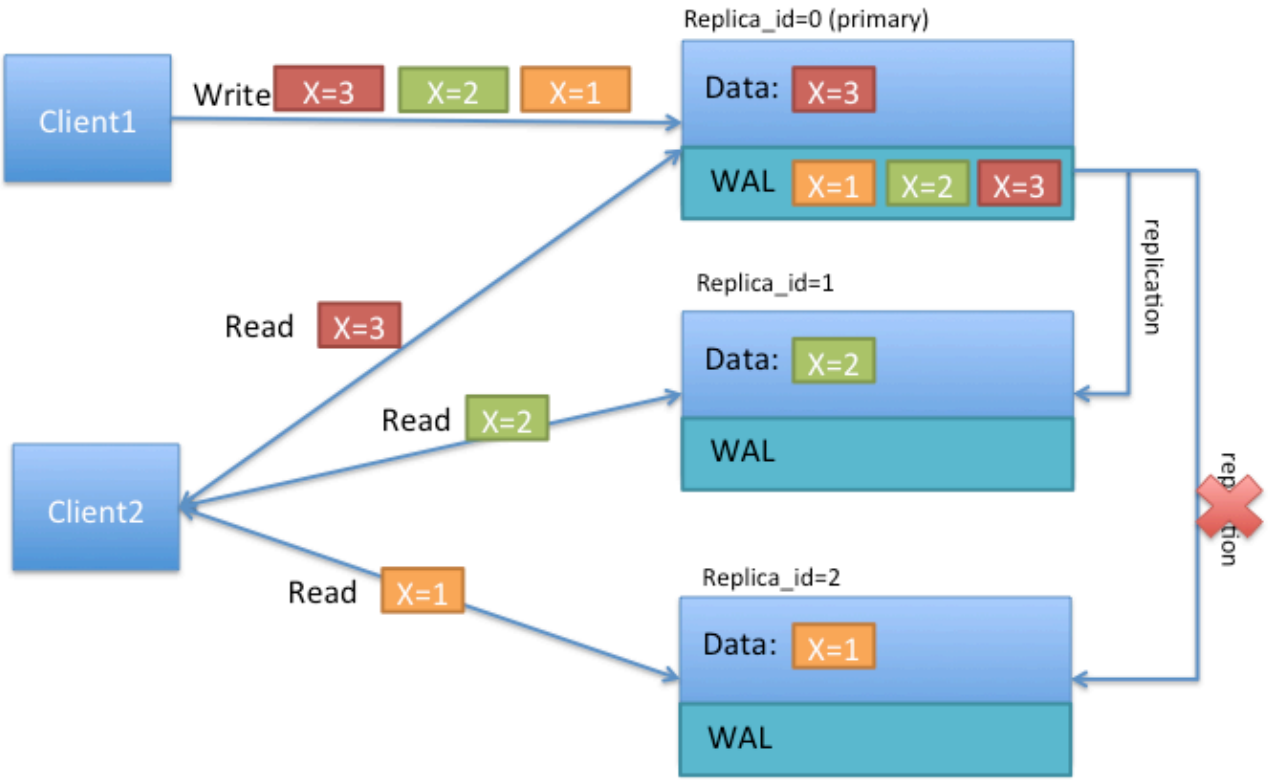
In case a read is performed with `Consistency.TIMELINE`, then the read RPC will be sent to the primary region server first. After a short interval (`hbase.client.primaryCallTimeout.get`, 10ms by default), parallel RPC for secondary region replicas will also be sent if the primary does not respond back. After this, the result is returned from whichever RPC is finished first. If the response came back from the primary region replica, we can always know that

the data is latest. For this Result.isStale() API has been added to inspect the staleness. If the result is from a secondary region, then Result.isStale() will be set to true. The user can then inspect this field to possibly reason about the data.

In terms of semantics, TIMELINE consistency as implemented by HBase differs from pure eventual consistency in these respects:

- Single homed and ordered updates: Region replication or not, on the write side, there is still only 1 defined replica (primary) which can accept writes. This replica is responsible for ordering the edits and preventing conflicts. This guarantees that two different writes are not committed at the same time by different replicas and the data diverges. With this, there is no need to do read-repair or last-timestamp-wins kind of conflict resolution.
- The secondaries also apply the edits in the order that the primary committed them. This way the secondaries will contain a snapshot of the primaries data at any point in time. This is similar to RDBMS replications and even HBase's own multi-datacenter replication, however in a single cluster.
- On the read side, the client can detect whether the read is coming from up-to-date data or is stale data. Also, the client can issue reads with different consistency requirements on a per-operation basis to ensure its own semantic guarantees.
- The client can still observe edits out-of-order, and can go back in time, if it observes reads from one secondary replica first, then another secondary replica. There is no stickiness to region replicas or a transaction-id based guarantee. If required, this can be implemented later though.

Timeline Consistency



To better understand the TIMELINE semantics, let's look at the above diagram. Let's say that there are two clients, and the first one writes x=1 at first, then x=2 and x=3 later. As above, all writes are handled by the primary region replica. The writes are saved in the write ahead log (WAL), and replicated to the other replicas asynchronously. In the above diagram, notice that replica_id=1 received 2 updates, and its data shows that x=2, while the replica_id=2 only received a single update, and its data shows that x=1.

If client1 reads with STRONG consistency, it will only talk with the replica_id=0, and thus is guaranteed to observe the latest value of x=3. In case of a client issuing TIMELINE consistency reads, the RPC will go to all replicas (after primary timeout) and the result from the first response will be returned back. Thus the client can see either 1, 2 or 3 as the value of x. Let's say that the primary region has failed and log replication cannot continue for some time. If the client does multiple reads with TIMELINE consistency, she can observe x=2 first, then x=1, and so on.

Tradeoffs

Having secondary regions hosted for read availability comes with some tradeoffs which should be carefully evaluated per use case. Following are advantages and disadvantages.

Advantages:

- High availability for read-only tables
- High availability for stale reads
- Ability to do very low latency reads with very high percentile (99.9%+) latencies for stale reads

Disadvantages:

- Double / Triple MemStore usage (depending on region replication count) for tables with region replication > 1
- Increased block cache usage
- Extra network traffic for log replication
- Extra backup RPCs for replicas

To serve the region data from multiple replicas, HBase opens the regions in secondary mode in the region servers. The regions opened in secondary mode will share the same data files with the primary region replica, however each secondary region replica will have its own MemStore to keep the unflushed data (only primary region can do flushes). Also to serve reads from secondary regions, the blocks of data files may be also cached in the block caches for the secondary regions.

Where is the code

This feature is delivered in two phases, Phase 1 and 2. The first phase is done in time for HBase-1.0.0 release. Meaning that using HBase-1.0.x, you can use all the features that are marked for Phase 1. Phase 2 is committed in HBase-1.1.0, meaning all HBase versions after 1.1.0 should contain Phase 2 items.

Propagating writes to region replicas

As discussed above writes only go to the primary region replica. For propagating the writes from the primary region replica to the secondaries, there are two different mechanisms. For read-only tables, you do not need to use any of the following methods. Disabling and enabling the table should make the data available in all region replicas. For mutable tables, you have to use **only** one of the following mechanisms: storefile refresher, or async wal replication. The latter is recommended.

StoreFile Refresher

The first mechanism is store file refresher which is introduced in HBase-1.0+. Store file refresher is a thread per region server, which runs periodically, and does a refresh operation for the store files of the primary region for the secondary region replicas. If enabled, the refresher will ensure that the secondary region replicas see the new flushed, compacted or bulk loaded files from the primary region in a timely manner. However, this means that only flushed data can be read back from the secondary region replicas, and after the refresher is run, making the secondaries lag behind the primary for an a longer time.

For turning this feature on, you should configure `hbase.regionserver.storefile.refresh.period` to a non-zero value. See Configuration section below.

Async WAL replication

The second mechanism for propagation of writes to secondaries is done via the “Async WAL Replication” feature. It is only available in HBase-1.1+. This works similarly to HBase's multi-datacenter replication, but instead the data from a region is replicated to the secondary regions. Each secondary replica always receives and observes the writes in the same order that the primary region committed them. In some sense, this design can be thought of as “in-cluster replication”, where instead of replicating to a different datacenter, the data goes to secondary regions to keep secondary region's in-memory state up to date. The data files are shared between the primary region and the other replicas, so that there is no extra storage overhead. However, the secondary regions will have recent non-flushed data in their memstores, which increases the memory overhead. The primary region writes flush, compaction, and bulk load events to its WAL as well, which are also replicated through wal replication to secondaries. When they observe the

flush/compaction or bulk load event, the secondary regions replay the event to pick up the new files and drop the old ones.

Committing writes in the same order as in primary ensures that the secondaries won't diverge from the primary regions data, but since the log replication is asynchronous, the data might still be stale in secondary regions.

Async WAL Replication is **disabled** by default. You can enable this feature by setting `hbase.region.replica.replication.enabled` to `true`.

Before 3.0.0, this feature works as a replication endpoint, the performance and latency characteristics is expected to be similar to inter-cluster replication. And once enabled, it will create a replication peer named `region_replica_replication` as a replication peer when you create a table with region replication > 1 for the first time.

if you want to disable this feature, you need to do two actions in the following order: * Set configuration property `hbase.region.replica.replication.enabled` to false in `hbase-site.xml` (see Configuration section below) * Disable the replication peer named `region_replica_replication` in the cluster using hbase shell or `Admin` class:

```
hbase> disable_peer 'region_replica_replication'
```

In 3.0.0, this feature is re-implemented to decouple with the general replication framework. Now we do not need to create a special replication peer. And during rolling upgrading, we will remove this replication peer automatically if it is present. See HBASE-26233 and the design doc in our git repo for more details.

Async WAL Replication and the `hbase:meta` table is a little more involved and gets its own section below; see Region replication for META table's region

Store File TTL

In both of the write propagation approaches mentioned above, store files of the primary will be opened in secondaries independent of the primary region. So for files that the primary compacted away, the secondaries might still be referring to these files for reading. Both features are using HFileLinks to refer to files, but there is no protection (yet) for guaranteeing that the file will not be deleted prematurely. Thus, as a guard, you should set

the configuration property `hbase.master.hfilecleaner.ttl` to a larger value, such as 1 hour to guarantee that you will not receive IOExceptions for requests going to replicas.

Region replication for META table's region

The general Async WAL Replication does not work for the META table's WAL. The meta table's secondary replicas refresh themselves from the persistent store files every `hbase.regionserver.meta.storefile.refresh.period`, (a non-zero value). Note how the META replication period is distinct from the user-space `hbase.regionserver.storefile.refresh.period` value.

Async WAL Replication for META table as of hbase-2.4.0+

Async WAL replication for META is added as a new feature in 2.4.0. Set `hbase.region.replica.replication.catalog.enabled` to enable async WAL Replication for META region replicas. It is off by default.

Regarding META replicas count, up to hbase-2.4.0, you would set the special property 'hbase.meta.replica.count'. Now you can alter the META table as you would a user-space table (if `hbase.meta.replica.count` is set, it will take precedent over what is set for replica count in the META table updating META replica count to match).

Async WAL Replication for META table as of hbase-3.0.0+

In HBASE-26233 we re-implemented the region replication framework to not rely on the general replication framework, so it can work together with META table as well. The code described in the above section have been removed mostly, but the config `hbase.region.replica.replication.catalog.enabled` is still kept, you could still use it to control whether to enable async wal replication for META table. And the ability to alter META table is also kept.

Load Balancing META table load

hbase-2.4.0 also adds a new client-side `LoadBalance` mode. When enabled client-side, clients will try to read META replicas first before falling back on the primary. Before this, the replica lookup mode — now named `HedgedRead` in hbase-2.4.0 — had clients read the

primary and if no response after a configurable amount of time had elapsed, it would start up reads against the replicas. Starting from hbase-2.4.12(and all higher minor versions), with client-side `LoadBalance` mode, clients load balance META scan requests across all META replica regions, including the primary META region. In case of exceptions such as NotServingRegionException, it will fall back on the primary META region.

The new 'LoadBalance' mode helps alleviate hotspotting on the META table distributing META read load.

To enable the meta replica locator's load balance mode, please set the following configuration at on the **client-side** (only): set 'hbase.locator.meta.replicas.mode' to "LoadBalance". Valid options for this configuration are `None`, `HedgedRead`, and `LoadBalance`. Option parse is case insensitive. The default mode is `None` (which falls through to `HedgedRead`, the current default). Do NOT put this configuration in any hbase server-side's configuration, Master or RegionServer (Master could make decisions based off stale state — to be avoided).

Memory accounting

The secondary region replicas refer to the data files of the primary region replica, but they have their own memstores (in HBase-1.1+) and uses block cache as well. However, one distinction is that the secondary region replicas cannot flush the data when there is memory pressure for their memstores. They can only free up memstore memory when the primary region does a flush and this flush is replicated to the secondary. Since in a region server hosting primary replicas for some regions and secondaries for some others, the secondaries might cause extra flushes to the primary regions in the same host. In extreme situations, there can be no memory left for adding new writes coming from the primary via wal replication. For unblocking this situation (and since secondary cannot flush by itself), the secondary is allowed to do a “store file refresh” by doing a file system list operation to pick up new files from primary, and possibly dropping its memstore. This refresh will only be performed if the memstore size of the biggest secondary region replica is at least `hbase.region.replica.storefile.refresh.memstore.multiplier` (default 4) times bigger than the biggest memstore of a primary replica. One caveat is that if this is performed, the secondary can observe partial row updates across column families (since column families are flushed independently). The default should be good to not do this operation frequently.

You can set this value to a large number to disable this feature if desired, but be warned that it might cause the replication to block forever.

Secondary replica failover

When a secondary region replica first comes online, or fails over, it may have served some edits from its memstore. Since the recovery is handled differently for secondary replicas, the secondary has to ensure that it does not go back in time before it starts serving requests after assignment. For doing that, the secondary waits until it observes a full flush cycle (start flush, commit flush) or a "region open event" replicated from the primary. Until this happens, the secondary region replica will reject all read requests by throwing an IOException with message "The region's reads are disabled". However, the other replicas will probably still be available to read, thus not causing any impact for the rpc with TIMELINE consistency. To facilitate faster recovery, the secondary region will trigger a flush request from the primary when it is opened. The configuration property `hbase.regio n.replica.wait.for.primary.flush` (enabled by default) can be used to disable this feature if needed.

Configuration properties

To use highly available reads, you should set the following properties in `hbase-site.xml` file. There is no specific configuration to enable or disable region replicas. Instead you can change the number of region replicas per table to increase or decrease at the table creation or with alter table. The following configuration is for using async wal replication and using meta replicas of 3.

Server side properties

```
<property>
  <name>hbase.regionserver.storefile.refresh.period</name>
  <value>0</value>
  <description>
    The period (in milliseconds) for refreshing the store files for the secondary
regions. 0 means this feature is disabled. Secondary regions sees new files (from
flushes and compactions) from primary once the secondary region refreshes the lis
t of files in the region (there is no notification mechanism). But too frequent r
```

```
efreshes might cause extra Namenode pressure. If the files cannot be refreshed fo
r longer than HFile TTL (hbase.master.hfilecleaner.ttl) the requests are rejecte
d. Configuring HFile TTL to a larger value is also recommended with this setting.
  </description>
</property>

<property>
  <name>hbase.regionserver.meta.storefile.refresh.period</name>
  <value>300000</value>
  <description>
    The period (in milliseconds) for refreshing the store files for the hbase:met
a tables secondary regions. 0 means this feature is disabled. Secondary regions s
ees new files (from flushes and compactions) from primary once the secondary regi
on refreshes the list of files in the region (there is no notification mechanis
m). But too frequent refreshes might cause extra Namenode pressure. If the files
cannot be refreshed for longer than HFile TTL (hbase.master.hfilecleaner.ttl) the
requests are rejected. Configuring HFile TTL to a larger value is also recommende
d with this setting. This should be a non-zero number if meta replicas are enable
d.
  </description>
</property>

<property>
  <name>hbase.region.replica.replication.enabled</name>
  <value>true</value>
  <description>
    Whether asynchronous WAL replication to the secondary region replicas is enab
led or not. If this is enabled, a replication peer named "region_replica_replicat
ion" will be created which will tail the logs and replicate the mutations to regi
on replicas for tables that have region replication > 1. If this is enabled once,
disabling this replication also requires disabling the replication peer using she
ll or Admin java class. Replication to secondary region replicas works over stand
ard inter-cluster replication.
  </description>
</property>

<property>
  <name>hbase.master.hfilecleaner.ttl</name>
  <value>3600000</value>
  <description>
    The period (in milliseconds) to keep store files in the archive folder before
deleting them from the file system.
  </description>
</property>

<property>
  <name>hbase.region.replica.storefile.refresh.memstore.multiplier</name>
  <value>4</value>
  <description>
    The multiplier for a "store file refresh" operation for the secondary region
replica. If a region server has memory pressure, the secondary region will refres
h it's store files if the memstore size of the biggest secondary replica is bigge
r this many times than the memstore size of the biggest primary replica. Set this
to a very big value to disable this feature (not recommended).
  </description>
</property>
```

```
<property>
  <name>hbase.region.replica.wait.for.primary.flush</name>
  <value>true</value>
  <description>
    Whether to wait for observing a full flush cycle from the primary before star
t serving data in a secondary. Disabling this might cause the secondary region re
plicas to go back in time for reads between region movements.Please note that if
you set per-table property `REGION_MEMSTORE_REPLICATION` to false,`hbase.region.r
eplica.wait.for.primary.flush` will be ignored.
  </description>
</property>
```

One thing to keep in mind also is that, region replica placement policy is only enforced by the `StochasticLoadBalancer` which is the default balancer. If you are using a custom load balancer property in hbase-site.xml (`hbase.master.loadbalancer.class`) replicas of regions might end up being hosted in the same server.

Client side properties

Ensure to set the following for all clients (and servers) that will use region replicas.

```
<property>
  <name>hbase.ipc.client.specificThreadForWriting</name>
  <value>true</value>
  <description>
    Whether to enable interruption of RPC threads at the client side. This is req
uired for region replicas with fallback RPC's to secondary regions.
  </description>
</property>
<property>
  <name>hbase.client.primaryCallTimeout.get</name>
  <value>10000</value>
  <description>
    The timeout (in microseconds), before secondary fallback RPC's are submitted
for get requests with Consistency.TIMELINE to the secondary replicas of the regio
ns. Defaults to 10ms. Setting this lower will increase the number of RPC's, but w
ill lower the p99 latencies.
  </description>
</property>
<property>
  <name>hbase.client.primaryCallTimeout.multiget</name>
  <value>10000</value>
  <description>
    The timeout (in microseconds), before secondary fallback RPC's are submitted
for multi-get requests (Table.get(List<Get>)) with Consistency.TIMELINE to the se
condary replicas of the regions. Defaults to 10ms. Setting this lower will increa
se the number of RPC's, but will lower the p99 latencies.
  </description>
</property>
<property>
```

```
  <name>hbase.client.replicaCallTimeout.scan</name>
  <value>1000000</value>
  <description>
    The timeout (in microseconds), before secondary fallback RPC's are submitted
for scan requests with Consistency.TIMELINE to the secondary replicas of the regi
ons. Defaults to 1 sec. Setting this lower will increase the number of RPC's, but
will lower the p99 latencies.
  </description>
</property>
<property>
  <name>hbase.meta.replicas.use</name>
  <value>true</value>
  <description>
    Whether to use meta table replicas or not. Default is false.
  </description>
</property>
```

Note HBase-1.0.x users should use `hbase.ipc.client.allowsInterrupt` rather than `hbase.ipc.client.specificThreadForWriting`.

User Interface

In the masters user interface, the region replicas of a table are also shown together with the primary regions. You can notice that the replicas of a region will share the same start and end keys and the same region name prefix. The only difference would be the appended replica_id (which is encoded as hex), and the region encoded name will be different. You can also see the replica ids shown explicitly in the UI.

Creating a table with region replication

Region replication is a per-table property. All tables have `REGION_REPLICATION = 1` by default, which means that there is only one replica per region. You can set and change the number of replicas per region of a table by supplying the `REGION_REPLICATION` property in the table descriptor.

There is another per-table property `REGION_MEMSTORE_REPLICATION`.All tables have `REGION_MEMSTORE_REPLICATION = true` by default, which means the new data written to the primary region should be replicated. If you set this to `false`, replicas do not receive memstore updates from the primary RegionServer,they will only receive updates for events like flushes and bulkloads, and will not have access to data which the primary has not yet

flushed. Please note that if you set `REGION_MEMSTORE_REPLICATION` to false, `hbase.region.replica.wait.for.primary.flush` will be ignored.

Shell

```
create 't1', 'f1', {REGION_REPLICATION => 2}

describe 't1'
for i in 1..100
put 't1', "r#{i}", 'f1:c1', i
end
flush 't1'
```

Java

```
HTableDescriptor htd = new HTableDescriptor(TableName.valueOf(“test_table”));
htd.setRegionReplication(2);
...
admin.createTable(htd);
```

You can also use `setRegionReplication()` and alter table to increase, decrease the region replication for a table.

Read API and Usage

Shell

You can do reads in shell using a the Consistency.TIMELINE semantics as follows

```
hbase(main):001:0> get 't1','r6', {CONSISTENCY => "TIMELINE"}
```

You can simulate a region server pausing or becoming unavailable and do a read from the secondary replica:

```
$ kill -STOP <pid or primary region server>

hbase(main):001:0> get 't1','r6', {CONSISTENCY => "TIMELINE"}
```

Using scans is also similar

```
hbase> scan 't1', {CONSISTENCY => 'TIMELINE'}
```

Java

You can set the consistency for Gets and Scans and do requests as follows.

```
Get get = new Get(row);
get.setConsistency(Consistency.TIMELINE);
...
Result result = table.get(get);
```

You can also pass multiple gets:

```
Get get1 = new Get(row);
get1.setConsistency(Consistency.TIMELINE);
...
ArrayList<Get> gets = new ArrayList<Get>();
gets.add(get1);
...
Result[] results = table.get(gets);
```

And Scans:

```
Scan scan = new Scan();
scan.setConsistency(Consistency.TIMELINE);
...
ResultScanner scanner = table.getScanner(scan);
```

You can inspect whether the results are coming from primary region or not by calling the `Result.isStale()` method:

```
Result result = table.get(get);
if (result.isStale()) {
  ...
}
```

Storing Medium-sized Objects (MOB)

Data comes in many sizes, and saving all of your data in HBase, including binary data such as images and documents, is ideal. While HBase can technically handle binary objects with

cells that are larger than 100 KB in size, HBase's normal read and write paths are optimized for values smaller than 100KB in size. When HBase deals with large numbers of objects over this threshold, referred to here as medium objects, or MOBs, performance is degraded due to write amplification caused by splits and compactions. When using MOBs, ideally your objects will be between 100KB and 10MB (see the faq). HBase 2 added special internal handling of MOBs to maintain performance, consistency, and low operational overhead. MOB support is provided by the work done in HBASE-11339. To take advantage of MOB, you need to use HFile version 3. Optionally, configure the MOB file reader's cache settings for each RegionServer (see Configure the MOB Cache), then configure specific columns to hold MOB data. Client code does not need to change to take advantage of HBase MOB support. The feature is transparent to the client.

Configuring Columns for MOB

You can configure columns to support MOB during table creation or alteration, either in HBase Shell or via the Java API. The two relevant properties are the boolean `IS_MOB` and the `MOB_THRESHOLD`, which is the number of bytes at which an object is considered to be a MOB. Only `IS_MOB` is required. If you do not specify the `MOB_THRESHOLD`, the default threshold value of 100 KB is used.

Configure a Column for MOB Using HBase Shell

```
hbase> create 't1', {NAME => 'f1', IS_MOB => true, MOB_THRESHOLD => 102400}
hbase> alter 't1', {NAME => 'f1', IS_MOB => true, MOB_THRESHOLD => 102400}
```

Configure a Column for MOB Using the Java API

```
...
HColumnDescriptor hcd = new HColumnDescriptor("f");
hcd.setMobEnabled(true);
...
hcd.setMobThreshold(102400L);
...
```

Testing MOB

The utility `org.apache.hadoop.hbase.IntegrationTestIngestWithMOB` is provided to assist with testing the MOB feature. The utility is run as follows:

```
$ sudo -u hbase hbase org.apache.hadoop.hbase.IntegrationTestIngestWithMOB \
        -threshold 1024 \
        -minMobDataSize 512 \
        -maxMobDataSize 5120
```

- `threshold` is the threshold at which cells are considered to be MOBs. The default is 1 kB, expressed in bytes.
- `minMobDataSize` is the minimum value for the size of MOB data. The default is 512 B, expressed in bytes.
- `maxMobDataSize` is the maximum value for the size of MOB data. The default is 5 kB, expressed in bytes.

MOB architecture

This section is derived from information found in HBASE-11339, which covered the initial GA implementation of MOB in HBase and HBASE-22749, which improved things by parallelizing MOB maintenance across the RegionServers. For more information see the last version of the design doc created during the initial work, "HBASE-11339 MOB GA design.pdf", and the design doc for the distributed mob compaction feature, "HBASE-22749 MOB distributed compaction.pdf".

Overview

The MOB feature reduces the overall IO load for configured column families by storing values that are larger than the configured threshold outside of the normal regions to avoid splits, merges, and most importantly normal compactions.

When a cell is first written to a region it is stored in the WAL and memstore regardless of value size. When memstores from a column family configured to use MOB are eventually flushed two hfiles are written simultaneously. Cells with a value smaller than the threshold size are written to a normal region hfile. Cells with a value larger than the threshold are written into a special MOB hfile and also have a MOB reference cell written into the normal

region HFile. As the Region Server flushes a MOB enabled memstore and closes a given normal region HFile it appends metadata that lists each of the special MOB hfiles referenced by the cells within.

MOB reference cells have the same key as the cell they are based on. The value of the reference cell is made up of two pieces of metadata: the size of the actual value and the MOB hfile that contains the original cell. In addition to any tags originally written to HBase, the reference cell prepends two additional tags. The first is a marker tag that says the cell is a MOB reference. This can be used later to scan specifically just for reference cells. The second stores the namespace and table at the time the MOB hfile is written out. This tag is used to optimize how the MOB system finds the underlying value in MOB hfiles after a series of HBase snapshot operations (ref HBASE-12332). Note that tags are only available within HBase servers and by default are not sent over RPCs.

All MOB hfiles for a given table are managed within a logical region that does not directly serve requests. When these MOB hfiles are created from a flush or MOB compaction they are placed in a dedicated mob data area under the hbase root directory specific to the namespace, table, mob logical region, and column family. In general that means a path structured like:

```
%HBase Root Dir%/mobdir/data/%namespace%/%table%/%logical region%/%column famil
y%/
```

With default configs, an example table named 'some_table' in the default namespace with a MOB enabled column family named 'foo' this HDFS directory would be

```
/hbase/mobdir/data/default/some_table/372c1b27e3dc0b56c3a031926e5efbe9/foo/
```

These MOB hfiles are maintained by special chores in the HBase Master and across the individual Region Servers. Specifically those chores take care of enforcing TTLs and compacting them. Note that this compaction is primarily a matter of controlling the total number of files in HDFS because our operational assumptions for MOB data is that it will seldom update or delete.

When a given MOB hfile is no longer needed as a result of our compaction process then a chore in the Master will take care of moving it to the archive just like any normal hfile. Because the table's mob region is independent of all the normal regions it can coexist with them in the regular archive storage area:

```
/hbase/archive/data/default/some_table/372c1b27e3dc0b56c3a031926e5efbe9/foo/
```

The same hfile cleaning chores that take care of eventually deleting unneeded archived files from normal regions thus also will take care of these MOB hfiles. As such, if there is a snapshot of a MOB enabled table then the cleaning system will make sure those MOB files stick around in the archive area as long as they are needed by a snapshot or a clone of a snapshot.

MOB compaction

Each time the memstore for a MOB enabled column family performs a flush HBase will write values over the MOB threshold into MOB specific hfiles. When normal region compaction occurs the Region Server rewrites the normal data files while maintaining references to these MOB files without rewriting them. Normal client lookups for MOB values transparently will receive the original values because the Region Server internals take care of using the reference data to then pull the value out of a specific MOB file. This indirection means that building up a large number of MOB hfiles doesn't impact the overall time to retrieve any specific MOB cell. Thus, we need not perform compactions of the MOB hfiles nearly as often as normal hfiles. As a result, HBase saves IO by not rewriting MOB hfiles as a part of the periodic compactions a Region Server does on its own.

However, if deletes and updates of MOB cells are frequent then this indirection will begin to waste space. The only way to stop using the space of a particular MOB hfile is to ensure no cells still hold references to it. To do that we need to ensure we have written the current values into a new MOB hfile. If our backing filesystem has a limitation on the number of files that can be present, as HDFS does, then even if we do not have deletes or updates of MOB cells eventually there will be a sufficient number of MOB hfiles that we will need to coalesce them.

Periodically a chore in the master coordinates having the region servers perform a special major compaction that also handles rewriting new MOB files. Like all compactions the Region Server will create updated hfiles that hold both the cells that are smaller than the MOB threshold and cells that hold references to the newly rewritten MOB file. Because this rewriting has the advantage of looking across all active cells for the region our several small MOB files should end up as a single MOB file per region. The chore defaults to running weekly and can be configured by setting `hbase.mob.compaction.chore.period` to the desired period in seconds.

```
<property>
  <name>hbase.mob.compaction.chore.period</name>
  <value>2592000</value>
  <description>Example of changing the chore period from a week to a month.</desc
ription>
</property>
```

By default, the periodic MOB compaction coordination chore will attempt to keep every region busy doing compactions in parallel in order to maximize the amount of work done on the cluster. If you need to tune the amount of IO this compaction generates on the underlying filesystem, you can control how many concurrent region-level compaction requests are allowed by setting `hbase.mob.major.compaction.region.batch.size` to an integer number greater than zero. If you set the configuration to 0 then you will get the default behavior of attempting to do all regions in parallel.

```
<property>
  <name>hbase.mob.major.compaction.region.batch.size</name>
  <value>1</value>
  <description>Example of switching from "as parallel as possible" to "serially"
</description>
</property>
```

MOB file archiving

Eventually we will have MOB hfiles that are no longer needed. Either clients will overwrite the value or a MOB-rewriting compaction will store a reference to a newer larger MOB hfile. Because any given MOB cell could have originally been written either in the current region or in a parent region that existed at some prior point in time, individual Region Servers do not decide when it is time to archive MOB hfiles. Instead a periodic chore in the Master evaluates MOB hfiles for archiving.

A MOB HFile will be subject to archiving under any of the following conditions:

- Any MOB HFile older than the column family's TTL
- Any MOB HFile older than a "too recent" threshold with no references to it from the regular hfiles for all regions in a column family

To determine if a MOB HFile meets the second criteria the chore extracts metadata from the regular HFiles for each MOB enabled column family for a given table. That metadata

enumerates the complete set of MOB HFiles needed to satisfy the references stored in the normal HFile area.

The period of the cleaner chore can be configured by setting `hbase.master.mob.cleaner.period` to a positive integer number of seconds. It defaults to running daily. You should not need to tune it unless you have a very aggressive TTL or a very high rate of MOB updates with a correspondingly high rate of non-MOB compactions.

MOB Optimization Tasks

Further limiting write amplification

If your MOB workload has few to no updates or deletes then you can opt-in to MOB compactions that optimize for limiting the amount of write amplification. It achieves this by setting a size threshold to ignore MOB files during the compaction process. When a given region goes through MOB compaction it will evaluate the size of the MOB file that currently holds the actual value and skip rewriting the value if that file is over threshold.

The bound of write amplification in this mode can be approximated as "Write Amplification" = $\log_K\left(\frac{M}{S}\right)$ where **K** is the number of files in compaction selection, **M** is the configurable threshold for MOB files size, and **S** is the minmum size of memstore flushes that create MOB files in the first place. For example given 5 files picked up per compaction, a threshold of 1 GB, and a flush size of 10MB the write amplification will be $\log_5\left(\frac{1\,\mathrm{GB}}{10\,\mathrm{MB}}\right) = \log_5(100) \approx 2.86$.

If we are using an underlying filesystem with a limitation on the number of files, such as HDFS, and we know our expected data set size we can choose our maximum file size in order to approach this limit but stay within it in order to minimize write amplification. For example, if we expect to store a petabyte and we have a conservative limitation of a million files in our HDFS instance, then $\frac{1\,\mathrm{PB}}{1\,\mathrm{M}} = 1\,\mathrm{GB}$ gives us a target limitation of a gigabyte per MOB file.

To opt-in to this compaction mode you must set `hbase.mob.compaction.type` to `optimized`. The default MOB size threshold in this mode is set to 1GB. It can be changed by setting `hbase.mob.compactions.max.file.size` to a positive integer number of bytes.

```
<property>
  <name>hbase.mob.compaction.type</name>
```

```
  <value>optimized</value>
  <description>opt-in to write amplification optimized mob compaction.</descripti
on>
</property>
<property>
  <name>hbase.mob.compactions.max.file.size</name>
  <value>10737418240</value>
  <description>Example of tuning the max mob file size to 10GB</description>
</property>
```

Additionally, when operating in this mode the compaction process will seek to avoid writing MOB files that are over the max file threshold. As it is writing out a additional MOB values into a MOB hfile it will check to see if the additional data causes the hfile to be over the max file size. When the hfile of MOB values reaches limit, the MOB hfile is committed to the MOB storage area and a new one is created. The hfile with reference cells will track the complete set of MOB hfiles it needs in its metadata.

> ⚠ **Be mindful of total time to complete compaction of a region**
>
> When using the write amplification optimized compaction mode you need to watch for the maximum time to compact a single region. If it nears an hour you should read through the troubleshooting section below Adjusting the MOB cleaner's tolerance for new hfiles. Failure to make the adjustments discussed there could lead to dataloss.

Configuring the MOB Cache

Because there can be a large number of MOB files at any time, as compared to the number of HFiles, MOB files are not always kept open. The MOB file reader cache is a LRU cache which keeps the most recently used MOB files open. To configure the MOB file reader's cache on each RegionServer, add the following properties to the RegionServer's `hbase-site.xml`, customize the configuration to suit your environment, and restart or rolling restart the RegionServer.

Example MOB Cache Configuration

```
<property>
  <name>hbase.mob.file.cache.size</name>
  <value>1000</value>
  <description>
    Number of opened file handlers to cache.
    A larger value will benefit reads by providing more file handlers per mob
    file cache and would reduce frequent file opening and closing.
    However, if this is set too high, this could lead to a "too many opened file
handers"
    The default value is 1000.
```

```
  </description>
</property>
<property>
  <name>hbase.mob.cache.evict.period</name>
  <value>3600</value>
  <description>
    The amount of time in seconds after which an unused file is evicted from the
    MOB cache. The default value is 3600 seconds.
  </description>
</property>
<property>
  <name>hbase.mob.cache.evict.remain.ratio</name>
  <value>0.5f</value>
  <description>
    A multiplier (between 0.0 and 1.0), which determines how many files remain ca
ched
    after the threshold of files that remains cached after a cache eviction occur
s
    which is triggered by reaching the `hbase.mob.file.cache.size` threshold.
    The default value is 0.5f, which means that half the files (the least-recentl
y-used
    ones) are evicted.
  </description>
</property>
```

Manually Compacting MOB Files

To manually compact MOB files, rather than waiting for the periodic chore to trigger compaction, use the `major_compact` HBase shell commands. These commands require the first argument to be the table name, and take a column family as the second argument. If used with a column family that includes MOB data, then these operator requests will result in the MOB data being compacted.

```
hbase> major_compact 't1'
hbase> major_compact 't2', 'c1'
```

This same request can be made via the `Admin.majorCompact` Java API.

MOB Troubleshooting

Adjusting the MOB cleaner's tolerance for new hfiles

The MOB cleaner chore ignores all MOB hfiles that were created more recently than an hour prior to the start of the chore to ensure we don't miss the reference metadata from

the corresponding regular hfile. Without this safety check it would be possible for the cleaner chore to see a MOB hfile for an in progress flush or compaction and prematurely archive the MOB data. This default buffer should be sufficient for normal use.

You will need to adjust the tolerance if you use write amplification optimized MOB compaction and the combination of your underlying filesystem performance and data shape is such that it could take more than an hour to complete major compaction of a single region. For example, if your MOB data is distributed such that your largest region adds 80GB of MOB data between compactions that include rewriting MOB data and your HDFS cluster is only capable of writing 20MB/s for a single file then when performing the optimized compaction the Region Server will take about a minute to write the first 1GB MOB hfile and then another hour and seven minutes to write the remaining seventy-nine 1GB MOB hfiles before finally committing the new reference hfile at the end of the compaction. Given this example, you would need a larger tolerance window.

You will also need to adjust the tolerance if Region Server flush operations take longer than an hour for the two HDFS move operations needed to commit both the MOB hfile and the normal hfile that references it. Such a delay should not happen with a normally configured and healthy HDFS and HBase.

The cleaner's window for "too recent" is controlled by setting `hbase.mob.min.age.archive` to a positive integer number of milliseconds.

```
<property>
  <name>hbase.mob.min.age.archive</name>
  <value>86400000</value>
  <description>Example of tuning the cleaner to only archive files older than a d
ay.</description>
</property>
```

Retrieving MOB metadata through the HBase Shell

While working on troubleshooting failures in the MOB system you can retrieve some of the internal information through the HBase shell by specifying special attributes on a scan.

```
hbase(main):112:0> scan 'some_table', {STARTROW => '00012-example-row-key', LIMIT
=> 1,
hbase(main):113:1*     CACHE_BLOCKS => false, ATTRIBUTES => { 'hbase.mob.scan.ra
w' => '1',
hbase(main):114:2*     'hbase.mob.scan.ref.only' => '1' } }
```

The MOB internal information is stored as four bytes for the size of the underlying cell value and then a UTF8 string with the name of the MOB HFile that contains the underlying cell value. Note that by default the entirety of this serialized structure will be passed through the HBase shell's binary string converter. That means the bytes that make up the value size will most likely be written as escaped non-printable byte values, e.g. '\x03', unless they happen to correspond to ASCII characters.

Let's look at a specific example:

```
hbase(main):112:0> scan 'some_table', {STARTROW => '00012-example-row-key', LIMIT
=> 1,
hbase(main):113:1*     CACHE_BLOCKS => false, ATTRIBUTES => { 'hbase.mob.scan.ra
w' => '1',
hbase(main):114:2*     'hbase.mob.scan.ref.only' => '1' } }
ROW                        COLUMN+CELL
 00012-example-row-key     column=foo:bar, timestamp=1511179764, value=\x00\x02|
\x94d41d8cd98f00b204
                           e9800998ecf8427e19700118ffd9c244fe69488bbc9f2c77d24a3e6a
1 row(s) in 0.0130 seconds
```

In this case the first four bytes are `\x00\x02|\x94` which corresponds to the bytes `[0x00, 0x02, 0x7C, 0x94]`. (Note that the third byte was printed as the ASCII character '|'.) Decoded as an integer this gives us an underlying value size of 162,964 bytes.

The remaining bytes give us an HFile name, 'd41d8cd98f00b204e9800998ecf8427e19700118ffd9c244fe69488bbc9f2c77d24a3e6a'. This HFile will most likely be stored in the designated MOB storage area for this specific table. However, the file could also be in the archive area if this table is from a restored snapshot. Furthermore, if the table is from a cloned snapshot of a different table then the file could be in either the active or archive area of that source table. As mentioned in the explanation of MOB reference cells above, the Region Server will use a server side tag to optimize looking at the mob and archive area of the correct original table when finding the MOB HFile. Since your scan is client side it can't retrieve that tag and you'll either need to already know the lineage of your table or you'll need to search across all tables.

Assuming you are authenticated as a user with HBase superuser rights, you can search for it:

```
$> hdfs dfs -find /hbase -name \
d41d8cd98f00b204e9800998ecf8427e19700118ffd9c244fe69488bbc9f2c77d24a3e6a
```

```
/hbase/mobdir/data/default/some_table/372c1b27e3dc0b56c3a031926e5efbe9/foo/d41d8c
d98f00b204e9800998ecf8427e19700118ffd9c244fe69488bbc9f2c77d24a3e6a
```

Moving a column family out of MOB

If you want to disable MOB on a column family you must ensure you instruct HBase to migrate the data out of the MOB system prior to turning the feature off. If you fail to do this HBase will return the internal MOB metadata to applications because it will not know that it needs to resolve the actual values.

The following procedure will safely migrate the underlying data without requiring a cluster outage. Clients will see a number of retries when configuration settings are applied and regions are reloaded.

Procedure: Stop MOB maintenance, change MOB threshold, rewrite data via compaction

1. Ensure the MOB compaction chore in the Master is off by setting `hbase.mob.compaction.chore.period` to `0`. Applying this configuration change will require a rolling restart of HBase Masters. That will require at least one fail-over of the active master, which may cause retries for clients doing HBase administrative operations.

2. Ensure no MOB compactions are issued for the table via the HBase shell for the duration of this migration.

3. Change the MOB size threshold

   Use the HBase shell to change the MOB size threshold for the column family you are migrating to a value that is larger than the largest cell present in the column family. E.g. given a table named 'some_table' and a column family named 'foo' we can pick one gigabyte as an arbitrary "bigger than what we store" value:

   ```
   hbase(main):011:0> alter 'some_table', {NAME => 'foo', MOB_THRESHOLD => '1
   000000000'}
   Updating all regions with the new schema...
   9/25 regions updated.
   25/25 regions updated.
   Done.
   0 row(s) in 3.4940 seconds
   ```

Note that if you are still ingesting data you must ensure this threshold is larger than any cell value you might write; MAX_INT would be a safe choice.

4 Perform a major compaction on the table

Specifically you are performing a "normal" compaction and not a MOB compaction.

```
hbase(main):012:0> major_compact 'some_table'
0 row(s) in 0.2600 seconds
```

5 Monitor for the end of the major compaction

Since compaction is handled asynchronously you'll need to use the shell to first see the compaction start and then see it end.

HBase should first say that a "MAJOR" compaction is happening.

```
hbase(main):015:0> @hbase.admin(@formatter).instance_eval do
hbase(main):016:1*   p @admin.get_compaction_state('some_table').to_string
hbase(main):017:2* end
"MAJOR"
```

When the compaction has finished the result should print out "NONE".

```
hbase(main):015:0> @hbase.admin(@formatter).instance_eval do
hbase(main):016:1*   p @admin.get_compaction_state('some_table').to_string
hbase(main):017:2* end
"NONE"
```

6 Run the *mobrefs* utility to ensure there are no MOB cells. Specifically, the tool will launch a Hadoop MapReduce job that will show a job counter of 0 input records when we've successfully rewritten all of the data.

```
$> HADOOP_CLASSPATH=/etc/hbase/conf:$(hbase mapredcp) yarn jar \
/some/path/to/hbase-shaded-mapreduce.jar mobrefs mobrefs-report-output some_
table foo
...
19/12/10 11:38:47 INFO impl.YarnClientImpl: Submitted application applicatio
n_1575695902338_0004
19/12/10 11:38:47 INFO mapreduce.Job: The url to track the job: https://rm-
2.example.com:8090/proxy application_1575695902338_0004/
19/12/10 11:38:47 INFO mapreduce.Job: Running job: job_1575695902338_0004
19/12/10 11:38:57 INFO mapreduce.Job: Job job_1575695902338_0004 running in
uber mode : false
19/12/10 11:38:57 INFO mapreduce.Job:  map 0% reduce 0%
```

```
19/12/10 11:39:07 INFO mapreduce.Job:  map 7% reduce 0%
19/12/10 11:39:17 INFO mapreduce.Job:  map 13% reduce 0%
19/12/10 11:39:19 INFO mapreduce.Job:  map 33% reduce 0%
19/12/10 11:39:21 INFO mapreduce.Job:  map 40% reduce 0%
19/12/10 11:39:22 INFO mapreduce.Job:  map 47% reduce 0%
19/12/10 11:39:23 INFO mapreduce.Job:  map 60% reduce 0%
19/12/10 11:39:24 INFO mapreduce.Job:  map 73% reduce 0%
19/12/10 11:39:27 INFO mapreduce.Job:  map 100% reduce 0%
19/12/10 11:39:35 INFO mapreduce.Job:  map 100% reduce 100%
19/12/10 11:39:35 INFO mapreduce.Job: Job job_1575695902338_0004 completed s
uccessfully
19/12/10 11:39:35 INFO mapreduce.Job: Counters: 54
...
       Map-Reduce Framework
               Map input records=0
...
19/12/09 22:41:28 INFO mapreduce.MobRefReporter: Finished creating report fo
r 'some_table', family='foo'
```

If the data has not successfully been migrated out, this report will show both a non-zero number of input records and a count of mob cells.

```
$> HADOOP_CLASSPATH=/etc/hbase/conf:$(hbase mapredcp) yarn jar \
/some/path/to/hbase-shaded-mapreduce.jar mobrefs mobrefs-report-output some_
table foo
...
19/12/10 11:44:18 INFO impl.YarnClientImpl: Submitted application applicatio
n_1575695902338_0005
19/12/10 11:44:18 INFO mapreduce.Job: The url to track the job: https://busb
ey-2.gce.cloudera.com:8090 proxy/application_1575695902338_0005/
19/12/10 11:44:18 INFO mapreduce.Job: Running job: job_1575695902338_0005
19/12/10 11:44:26 INFO mapreduce.Job: Job job_1575695902338_0005 running in
uber mode : false
19/12/10 11:44:26 INFO mapreduce.Job:  map 0% reduce 0%
19/12/10 11:44:36 INFO mapreduce.Job:  map 7% reduce 0%
19/12/10 11:44:45 INFO mapreduce.Job:  map 13% reduce 0%
19/12/10 11:44:47 INFO mapreduce.Job:  map 27% reduce 0%
19/12/10 11:44:48 INFO mapreduce.Job:  map 33% reduce 0%
19/12/10 11:44:50 INFO mapreduce.Job:  map 40% reduce 0%
19/12/10 11:44:51 INFO mapreduce.Job:  map 53% reduce 0%
19/12/10 11:44:52 INFO mapreduce.Job:  map 73% reduce 0%
19/12/10 11:44:54 INFO mapreduce.Job:  map 100% reduce 0%
19/12/10 11:44:59 INFO mapreduce.Job:  map 100% reduce 100%
19/12/10 11:45:00 INFO mapreduce.Job: Job job_1575695902338_0005 completed s
uccessfully
19/12/10 11:45:00 INFO mapreduce.Job: Counters: 54
...
       Map-Reduce Framework
               Map input records=1
...
       MOB
               NUM_CELLS=1
...
```

```
19/12/10 11:45:00 INFO mapreduce.MobRefReporter: Finished creating report fo
r 'some_table', family='foo'
```

If this happens you should verify that MOB compactions are disabled, verify that you have picked a sufficiently large MOB threshold, and redo the major compaction step.

7 Disable the MOB feature for the column family

When the *mobrefs* report shows that no more data is stored in the MOB system then you can safely alter the column family configuration so that the MOB feature is disabled.

```
hbase(main):017:0> alter 'some_table', {NAME => 'foo', IS_MOB => 'false'}
Updating all regions with the new schema...
8/25 regions updated.
25/25 regions updated.
Done.
0 row(s) in 2.9370 seconds
```

The MOB feature will be disabled on a column family only after altering the column family and performing a major compaction. Before performing the major compaction after altering the column family, the MOB cells will still be present in the MOB storage.

8 After the column family no longer shows the MOB feature enabled, it is safe to start MOB maintenance chores again. You can allow the default to be used for `hbase.mob.compaction.chore.period` by removing it from your configuration files or restore it to whatever custom value you had prior to starting this process.

9 Clean up residual MOB data

Once the MOB feature is disabled for the column family there will be no internal HBase process looking for data in the MOB storage area specific to this column family. There will still be data present there from prior to the compaction process that rewrote the values into HBase's data area. You can check for this residual data directly in HDFS as an HBase superuser.

```
$ hdfs dfs -count /hbase/mobdir/data/default/some_table
           4           54         9063269081 /hbase/mobdir/data/default/some
_table
```

> This data is spurious and may be reclaimed. You should sideline it, verify your application's view of the table, and then delete it.

Data values over than the MOB threshold show up stored in non-MOB hfiles

Bulk load and WAL split-to-HFile don't consider MOB threshold and write data into normal hfile (under /hbase/data directory).

> This won't cause any functional problem, during next compaction such data will be written out to the MOB hfiles.

MOB Upgrade Considerations

Generally, data stored using the MOB feature should transparently continue to work correctly across HBase upgrades.

Upgrading to a version with the "distributed MOB compaction" feature

Prior to the work in HBASE-22749, "Distributed MOB compactions", HBase had the Master coordinate all compaction maintenance of the MOB hfiles. Centralizing management of the MOB data allowed for space optimizations but safely coordinating that management with Region Servers resulted in edge cases that caused data loss (ref HBASE-22075).

Users of the MOB feature upgrading to a version of HBase that includes HBASE-22749 should be aware of the following changes:

- The MOB system no longer allows setting "MOB Compaction Policies"
- The MOB system no longer attempts to group MOB values by the date of the original cell's timestamp according to said compaction policies, daily or otherwise
- The MOB system no longer needs to track individual cell deletes through the use of special files in the MOB storage area with the suffix `_del`. After upgrading you should sideline these files.
- Under default configuration the MOB system should take much less time to perform a compaction of MOB stored values. This is a direct consequence of the fact that HBase

will place a much larger load on the underlying filesystem when doing compactions of MOB stored values; the additional load should be a multiple on the order of magnitude of number of region servers. I.e. for a cluster with three region servers and two masters the default configuration should have HBase put three times the load on HDFS during major compactions that rewrite MOB data when compared to Master handled MOB compaction; it should also be approximately three times as fast.

- When the MOB system detects that a table has hfiles with references to MOB data but the reference hfiles do not yet have the needed file level metadata (i.e. from use of the MOB feature prior to HBASE-22749) then it will refuse to archive *any* MOB hfiles from that table. The normal course of periodic compactions done by Region Servers will update existing hfiles with MOB references, but until a given table has been through the needed compactions operators should expect to see an increased amount of storage used by the MOB feature.
- Performing a compaction with type "MOB" no longer has special handling to compact specifically the MOB hfiles. Instead it will issue a warning and do a compaction of the table. For example using the HBase shell as follows will result in a warning in the Master logs followed by a major compaction of the 'example' table in its entirety or for the 'big' column respectively.

  ```
  hbase> major_compact 'example', nil, 'MOB'
  hbase> major_compact 'example', 'big', 'MOB'
  ```

  The same is true for directly using the Java API for `admin.majorCompact(TableName.valueOf("example"), CompactType.MOB)`.
- Similarly, manually performing a major compaction on a table or region will also handle compacting the MOB stored values for that table or region respectively.

The following configuration setting has been deprecated and replaced:

- `hbase.master.mob.ttl.cleaner.period` has been replaced with `hbase.master.mob.cleaner.period`

The following configuration settings are no longer used:

- `hbase.mob.compaction.mergeable.threshold`
- `hbase.mob.delfile.max.count`

- `hbase.mob.compaction.batch.size`
- `hbase.mob.compactor.class`
- `hbase.mob.compaction.threads.max`

Scan Over Snapshot

In HBase, a scan of a table costs server-side HBase resources reading, formating, and returning data back to the client. Luckily, HBase provides a TableSnapshotScanner and TableSnapshotInputFormat (introduced by HBASE-8369), which can scan HBase-written HFiles directly in the HDFS filesystem completely by-passing hbase. This access mode performs better than going via HBase and can be used with an offline HBase with in-place or exported snapshot HFiles.

To read HFiles directly, the user must have sufficient permissions to access snapshots or in-place hbase HFiles.

TableSnapshotScanner

TableSnapshotScanner provides a means for running a single client-side scan over snapshot files. When using TableSnapshotScanner, we must specify a temporary directory to copy the snapshot files into. The client user should have write permissions to this directory, and the dir should not be a subdirectory of the hbase.rootdir. The scanner deletes the contents of the directory once the scanner is closed.

Use TableSnapshotScanner

```
Path restoreDir = new Path("XX"); // restore dir should not be a subdirectory of
hbase.rootdir
Scan scan = new Scan();
try (TableSnapshotScanner scanner = new TableSnapshotScanner(conf, restoreDir, sn
apshotName, scan)) {
    Result result = scanner.next();
    while (result != null) {
        ...
        result = scanner.next();
    }
}
```

TableSnapshotInputFormat

TableSnapshotInputFormat provides a way to scan over snapshot HFiles in a MapReduce job.

Use TableSnapshotInputFormat

```
Job job = new Job(conf);
Path restoreDir = new Path("XX"); // restore dir should not be a subdirectory of
hbase.rootdir
Scan scan = new Scan();
TableMapReduceUtil.initTableSnapshotMapperJob(snapshotName, scan, MyTableMapper.c
lass, MyMapKeyOutput.class, MyMapOutputValueWritable.class, job, true, restoreDi
r);
```

Permission to access snapshot and data files

Generally, only the HBase owner or the HDFS admin have the permission to access HFiles.

HBASE-18659 uses HDFS ACLs to make HBase granted user have permission to access snapshot files.

HDFS ACLs

HDFS ACLs supports an "access ACL", which defines the rules to enforce during permission checks, and a "default ACL", which defines the ACL entries that new child files or sub-directories receive automatically during creation. Via HDFS ACLs, HBase syncs granted users with read permission to HFiles.

Basic idea

The HBase files are organized in the following ways:

- `{hbase-rootdir}/.tmp/data/{namespace}/{table}`
- `{hbase-rootdir}/data/{namespace}/{table}`
- `{hbase-rootdir}/archive/data/{namespace}/{table}`
- `{hbase-rootdir}/.hbase-snapshot/{snapshotName}`

So the basic idea is to add or remove HDFS ACLs to files of the global/namespace/table directory when grant or revoke permission to global/namespace/table.

See the design doc in HBASE-18659 for more details.

Configuration to use this feature

- Firstly, make sure that HDFS ACLs are enabled and umask is set to 027

```
dfs.namenode.acls.enabled = true
fs.permissions.umask-mode = 027
```

- Add master coprocessor, please make sure the SnapshotScannerHDFSAclController is configured after AccessController

```
hbase.coprocessor.master.classes = "org.apache.hadoop.hbase.security.access.Acc
essController
,org.apache.hadoop.hbase.security.access.SnapshotScannerHDFSAclController"
```

- Enable this feature

```
hbase.acl.sync.to.hdfs.enable=true
```

- Modify table scheme to enable this feature for a specified table, this config is false by default for every table, this means the HBase granted ACLs will not be synced to HDFS

```
alter 't1', CONFIGURATION => {'hbase.acl.sync.to.hdfs.enable' => 'true'}
```

Limitation

There are some limitations for this feature:

- If we enable this feature, some master operations such as grant, revoke, snapshot... (See the design doc for more details) will be slower as we need to sync HDFS ACLs to related hfiles.
- HDFS has a config which limits the max ACL entries num for one directory or file:

```
dfs.namenode.acls.max.entries = 32(default value)
```

The 32 entries include four fixed users for each directory or file: owner, group, other, and mask. For a directory, the four users contain 8 ACL entries(access and default) and for a file, the four users contain 4 ACL entries(access). This means there are 24 ACL entries left for named users or groups.
Based on this limitation, we can only sync up to 12 HBase granted users' ACLs. This means, if a table enables this feature, then the total users with table, namespace of this table, global READ permission should not be greater than 12.

- There are some cases that this coprocessor has not handled or could not handle, so the user HDFS ACLs are not synced normally. It will not make a reference link to another hfile of other tables.

In-memory Compaction

Overview

In-memory Compaction (A.K.A Accordion) is a new feature in hbase-2.0.0. It was first introduced on the Apache HBase Blog at Accordion: HBase Breathes with In-Memory Compaction. Quoting the blog:

> *"Accordion reapplies the LSM principal [Log-Structured-Merge Tree, the design pattern upon which HBase is based] to MemStore, in order to eliminate redundancies and other overhead while the data is still in RAM. Doing so decreases the frequency of flushes to HDFS, thereby reducing the write amplification and the overall disk footprint. With less flushes, the write operations are stalled less frequently as the MemStore overflows, therefore the write performance is improved. Less data on disk also implies less pressure on the block cache, higher hit rates, and eventually better read response times. Finally, having less disk writes also means having less compaction happening in the background, i.e., less cycles are stolen from productive (read and write) work. All in all, the effect of in-memory compaction can be envisioned as a catalyst that enables the system move faster as a whole."*

A developer view is available at Accordion: Developer View of In-Memory Compaction.

In-memory compaction works best when high data churn; overwrites or over-versions can be eliminated while the data is still in memory. If the writes are all uniques, it may drag write throughput (In-memory compaction costs CPU). We suggest you test and compare before deploying to production.

In this section we describe how to enable Accordion and the available configurations.

Enabling

To enable in-memory compactions, set the *IN_MEMORY_COMPACTION* attribute on per column family where you want the behavior. The *IN_MEMORY_COMPACTION* attribute can have one of four values.

- *NONE*: No in-memory compaction.

- *BASIC*: Basic policy enables flushing and keeps a pipeline of flushes until we trip the pipeline maximum threshold and then we flush to disk. No in-memory compaction but can help throughput as data is moved from the profligate, native ConcurrentSkipListMap data-type to more compact (and efficient) data types.
- *EAGER*: This is *BASIC* policy plus in-memory compaction of flushes (much like the on-disk compactions done to hfiles); on compaction we apply on-disk rules eliminating versions, duplicates, ttl'd cells, etc.
- *ADAPTIVE*: Adaptive compaction adapts to the workload. It applies either index compaction or data compaction based on the ratio of duplicate cells in the data. Experimental.

To enable *BASIC* on the *info* column family in the table *radish*, add the attribute to the *info* column family:

```
hbase(main):003:0> alter 'radish', {NAME => 'info', IN_MEMORY_COMPACTION => 'BASI
C'}
Updating all regions with the new schema...
All regions updated.
Done.
Took 1.2413 seconds
hbase(main):004:0> describe 'radish'
Table radish is DISABLED
radish
COLUMN FAMILIES DESCRIPTION
{NAME => 'info', VERSIONS => '1', EVICT_BLOCKS_ON_CLOSE => 'false', NEW_VERSION_B
EHAVIOR => 'false', KEEP_DELETED_CELLS => 'FALSE', CACHE_DATA_ON_WRITE => 'fals
e', DATA_BLOCK_ENCODING => 'NONE', TTL => 'FOREVER', MIN_VERSIONS => '0', REPLICA
TION_SCOPE => '0', BLOOMFILTER => 'ROW', CACHE_INDEX_ON_WRITE => 'false', IN_MEMO
RY => 'false', CACHE_BLOOMS_ON_WRITE => 'false', PREFETCH_BLOCKS_ON_OPEN => 'fals
e', COMPRESSION => 'NONE', BLOCKCACHE => 'true', BLOCKSIZE => '65536', METADATA =
> {
'IN_MEMORY_COMPACTION' => 'BASIC'}}
1 row(s)
Took 0.0239 seconds
```

Note how the IN*MEMORY_COMPACTION attribute shows as part of the _METADATA* map.

There is also a global configuration, *hbase.hregion.compacting.memstore.type* which you can set in your *hbase-site.xml* file. Use it to set the default on creation of a new table (On creation of a column family Store, we look first to the column family configuration looking for the *IN_MEMORY_COMPACTION* setting, and if none, we then consult the *hbase.hregion.compacting.memstore.type* value using its content; default is *NONE*).

By default, new hbase system tables will have *NONE* in-memory compaction set. To specify otherwise, on new table-creation, set *hbase.systemtables.compacting.memstore.type* to *BASIC* | *EAGER* | *ADAPTIVE* (Note, setting this value post-creation of system tables will not have a retroactive effect; you will have to alter your tables to set the in-memory attribute).

When an in-memory flush happens is calculated by dividing the configured region flush size (Set in the table descriptor or read from *hbase.hregion.memstore.flush.size*) by the number of column families and then multiplying by *hbase.memstore.inmemoryflush.threshold.factor*. Default is 0.014.

The number of flushes carried by the pipeline is monitored so as to fit within the bounds of memstore sizing but you can also set a maximum on the number of flushes total by setting *hbase.hregion.compacting.pipeline.segments.limit*. Default is 2.

When a column family Store is created, it says what memstore type is in effect. As of this writing there is the old-school *DefaultMemStore* which fills a *ConcurrentSkipListMap* and then flushes to disk or the new *CompactingMemStore* that is the implementation that provides this new in-memory compactions facility. Here is a log-line from a RegionServer that shows a column family Store named *family* configured to use a *CompactingMemStore*:

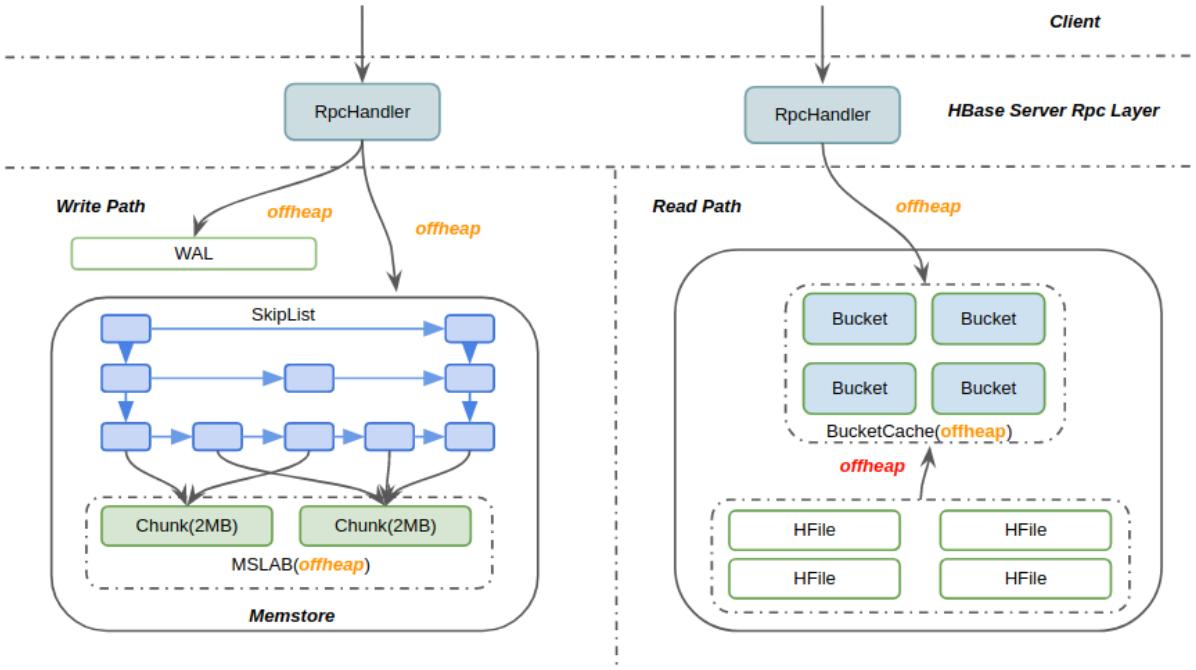
```
Note how the IN_MEMORY_COMPACTION attribute shows as part of the _METADATA_ map.
2018-03-30 11:02:24,466 INFO  [Time-limited test] regionserver.HStore(325): Store
=family,  memstore type=CompactingMemStore, storagePolicy=HOT, verifyBulkLoads=fa
lse, parallelPutCountPrintThreshold=10
```

Enable TRACE-level logging on the CompactingMemStore class (*org.apache.hadoop.hbase.regionserver.CompactingMemStore*) to see detail on its operation.

RegionServer Offheap Read/Write Path

Overview

To help reduce P99/P999 RPC latencies, HBase 2.x has made the read and write path use a pool of offheap buffers. Cells are allocated in offheap memory outside of the purview of the JVM garbage collector with attendent reduction in GC pressure. In the write path, the request packet received from client will be read in on a pre-allocated offheap buffer and retained offheap until those cells are successfully persisted to the WAL and Memstore. The memory data structure in Memstore does not directly store the cell memory, but references the cells encoded in the offheap buffers. Similarly for the read path. We'll try to read the block cache first and if a cache misses, we'll go to the HFile and read the respective block. The workflow from reading blocks to sending cells to client does its best to avoid on-heap memory allocations reducing the amount of work the GC has to do.



For redress for the single mention of onheap in the read-section of the diagram above see Read block from HDFS to offheap directly.

Offheap read-path

In HBase-2.0.0, HBASE-11425 changed the HBase read path so it could hold the read-data off-heap avoiding copying of cached data (BlockCache) on to the java heap (for uncached data, see note under the diagram in the section above). This reduces GC pauses given there is less garbage made and so less to clear. The off-heap read path can have a performance that is similar or better to that of the on-heap LRU cache. This feature is available since HBase 2.0.0. Refer to below blogs for more details and test results on off heaped read path Offheaping the Read Path in Apache HBase: Part 1 of 2 and Offheap Read-Path in Production - The Alibaba story

For an end-to-end off-heaped read-path, all you have to do is enable an off-heap backed Off-heap Block Cache(BC). To do this, configure *hbase.bucketcache.ioengine* to be *offheap* in *hbase-site.xml* (See BucketCache Deploy Modes to learn more about *hbase.bucketcache.ioengine* options). Also specify the total capacity of the BC using `hbase.bucketcache.size`. Please remember to adjust value of 'HBASE*OFFHEAPSIZE' in _hbase-env.sh* (See BucketCache Example Configuration for help sizing and an example enabling). This configuration is for specifying the maximum possible off-heap memory allocation for the RegionServer java process. This should be bigger than the off-heap BC size to accommodate usage by other features making use of off-heap memory such as Server RPC buffer pool and short-circuit reads (See discussion in BucketCache Example Configuration).

Please keep in mind that there is no default for `hbase.bucketcache.ioengine` which means the `BlockCache` is OFF by default (See the "Direct Memory Usage In HBase" info section in BucketCache Example Configuration).

This is all you need to do to enable off-heap read path. Most buffers in HBase are already off-heap. With BC off-heap, the read pipeline will copy data between HDFS and the server socket — caveat hbase.ipc.server.reservoir.initial.max — sending results back to the client.

Tuning the RPC buffer pool

It is possible to tune the ByteBuffer pool on the RPC server side used to accumulate the cell bytes and create result cell blocks to send back to the client side. Use `hbase.ipc.server.reservoir.enabled` to turn this pool ON or OFF. By default this pool is ON and available.

HBase will create off-heap ByteBuffers and pool them them by default. Please make sure not to turn this OFF if you want end-to-end off-heaping in read path.

If this pool is turned off, the server will create temp buffers onheap to accumulate the cell bytes and make a result cell block. This can impact the GC on a highly read loaded server.

> The config keys which start with prefix `hbase.ipc.server.reservoir` are deprecated in hbase-3.x (the internal pool implementation changed). If you are still in hbase-2.2.x or older, then just use the old config keys. Otherwise if in hbase-3.x or hbase-2.3.x+, please use the new config keys (See deprecated and new configs in HBase3.x)

Next thing to tune is the ByteBuffer pool on the RPC server side. The user can tune this pool with respect to how many buffers are in the pool and what should be the size of each ByteBuffer. Use the config `hbase.ipc.server.reservoir.initial.buffer.size` to tune each of the buffer sizes. Default is 64KB for hbase-2.2.x and less, changed to 65KB by default for hbase-2.3.x+ (see HBASE-22532)

When the result size is larger than one 64KB (Default) ByteBuffer size, the server will try to grab more than one ByteBuffer and make a result cell block out of a collection of fixed-sized ByteBuffers. When the pool is running out of buffers, the server will skip the pool and create temporary on-heap buffers.

The maximum number of ByteBuffers in the pool can be tuned using the config `hbase.ipc.server.reservoir.initial.max`. Its default is a factor of region server handlers count (See the config `hbase.regionserver.handler.count`). The math is such that by default we consider 2 MB as the result cell block size per read result and each handler will be handling a read. For 2 MB size, we need 32 buffers each of size 64 KB (See default buffer size in pool). So per handler 32 ByteBuffers(BB). We allocate twice this size as the max BBs count such that one handler can be creating the response and handing it to the RPC Responder thread and then handling a new request creating a new response cell block (using pooled buffers). Even if the responder could not send back the first TCP reply immediately, our count should allow that we should still have enough buffers in our pool without having to make temporary buffers on the heap. Again for smaller sized random row reads, tune this max count. These are lazily created buffers and the count is the max count to be pooled.

If you still see GC issues even after making end-to-end read path off-heap, look for issues in the appropriate buffer pool. Check for the below RegionServer log line at INFO level in

HBase2.x:

```
Pool already reached its max capacity : XXX and no free buffers now. Consider inc
reasing the value for 'hbase.ipc.server.reservoir.initial.max' ?
```

Or the following log message in HBase3.x:

```
Pool already reached its max capacity : XXX and no free buffers now. Consider inc
reasing the value for 'hbase.server.allocator.max.buffer.count' ?
```

The setting for *HBASE_OFFHEAPSIZE* in *hbase-env.sh* should consider this off heap buffer pool on the server side also. We need to config this max off heap size for the RegionServer as a bit higher than the sum of this max pool size and the off heap cache size. The TCP layer will also need to create direct bytebuffers for TCP communication. Also the DFS client will need some off-heap to do its workings especially if short-circuit reads are configured. Allocating an extra 1 - 2 GB for the max direct memory size has worked in tests.

If you are using coprocessors and refer to the Cells in the read results, DO NOT store reference to these Cells out of the scope of the CP hook methods. Some times the CPs want to store info about the cell (Like its row key) for considering in the next CP hook call etc. For such cases, pls clone the required fields of the entire Cell as per the use cases. [See CellUtil#cloneXXX(Cell) APIs]

Read block from HDFS to offheap directly

In HBase-2.x, the RegionServer will read blocks from HDFS to a temporary onheap ByteBuffer and then flush to the BucketCache. Even if the BucketCache is offheap, we will first pull the HDFS read onheap before writing it out to the offheap BucketCache. We can observe much GC pressure when cache hit ratio low (e.g. a cacheHitRatio ~ 60%). HBASE-21879 addresses this issue (Requires hbase-2.3.x/hbase-3.x). It depends on there being a supporting HDFS being in place (hadoop-2.10.x or hadoop-3.3.x) and it may require patching HBase itself (as of this writing); see HBASE-21879 Read HFile's block to ByteBuffer directly instead of to byte for reducing young gc purpose. Appropriately setup, reads from HDFS can be into offheap buffers passed offheap to the offheap BlockCache to cache.

For more details about the design and performance improvement, please see the Design Doc -Read HFile's block to Offheap.

Here we will share some best practice about the performance tuning but first we introduce new (hbase-3.x/hbase-2.3.x) configuration names that go with the new internal pool implementation (`ByteBuffAllocator` vs the old `ByteBufferPool`), some of which mimic now deprecated hbase-2.2.x configurations discussed above in the Tuning the RPC buffer pool. Much of the advice here overlaps that given above in the Tuning the RPC buffer pool since the implementations have similar configurations.

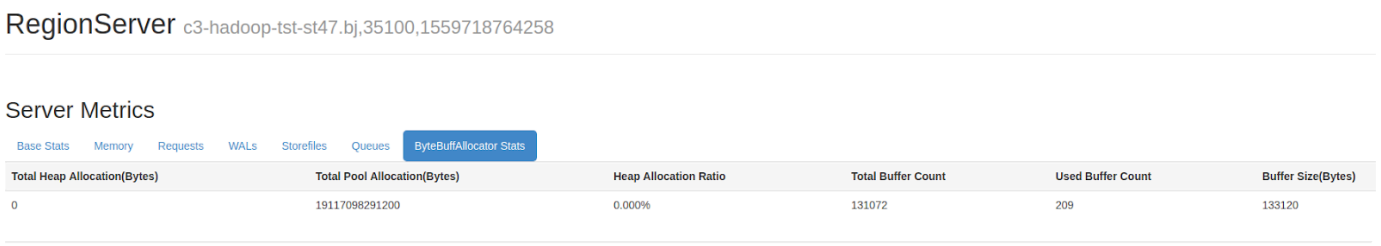
1. `hbase.server.allocator.pool.enabled` is for whether the RegionServer will use the pooled offheap ByteBuffer allocator. Default value is true. In hbase-2.x, the deprecated `hbase.ipc.server.reservoir.enabled` did similar and is mapped to this config until support for the old configuration is removed. This new name will be used in hbase-3.x and hbase-2.3.x+.
2. `hbase.server.allocator.minimal.allocate.size` is the threshold at which we start allocating from the pool. Otherwise the request will be allocated from onheap directly because it would be wasteful allocating small stuff from our pool of fixed-size ByteBuffers. The default minimum is `hbase.server.allocator.buffer.size/6`.
3. `hbase.server.allocator.max.buffer.count`: The `ByteBuffAllocator`, the new pool/reservoir implementation, has fixed-size ByteBuffers. This config is for how many buffers to pool. Its default value is 2MB _ 2 _ hbase.regionserver.handler.count / 65KB (similar to the discussion above in Tuning the RPC buffer pool). If the default `hbase.regionserver.handler.count` is 30, then the default will be 1890.
4. `hbase.server.allocator.buffer.size`: The byte size of each ByteBuffer. The default value is 66560 (65KB), here we choose 65KB instead of 64KB because of HBASE-22532.

The three config keys — `hbase.ipc.server.reservoir.enabled`, `hbase.ipc.server.reservoir.initial.buffer.size` and `hbase.ipc.server.reservoir.initial.max` — introduced in hbase-2.x have been renamed and deprecated in hbase-3.x/hbase-2.3.x. Please use the new config keys instead: `hbase.server.allocator.pool.enabled`, `hbase.server.allocator.buffer.size` and `hbase.server.allocator.max.buffer.count`.

Next, we have some suggestions regards performance.

Please make sure that there are enough pooled DirectByteBuffer in your ByteBuffAllocator.
The ByteBuffAllocator will allocate ByteBuffer from the DirectByteBuffer pool first. If there's no available ByteBuffer in the pool, then we will allocate the ByteBuffers from onheap. By default, we will pre-allocate 4MB for each RPC handler (The handler count is determined by the config: `hbase.regionserver.handler.count`, it has the default value 30) . That's to say, if your `hbase.server.allocator.buffer.size` is 65KB, then your pool will have 2MB _ 2 / 65KB _ 30 = 945 DirectByteBuffer. If you have a large scan and a big cache, you may have a RPC response whose bytes size is greater than 2MB (another 2MB for receiving rpc request), then it will be better to increase the `hbase.server.allocator.max.buffer.count`.

The RegionServer web UI has statistics on ByteBuffAllocator:



If the following condition is met, you may need to increase your max buffer.count:

```
heapAllocationRatio >= hbase.server.allocator.minimal.allocate.size / hbase.serve
r.allocator.buffer.size * 100%
```

Please make sure the buffer size is greater than your block size.
We have the default block size of 64KB, so almost all of the data blocks will be 64KB + a small delta, where the delta is very small, depending on the size of the last Cell. If we set `hbase.server.allocator.buffer.size` =64KB, then each block will be allocated as two ByteBuffers: one 64KB DirectByteBuffer and one HeapByteBuffer for the delta bytes. Ideally, we should let the data block to be allocated as one ByteBuffer; it has a simpler data structure, faster access speed, and less heap usage. Also, if the blocks are a composite of multiple ByteBuffers, to validate the checksum we have to perform a temporary heap copy (see HBASE-21917) whereas if it's a single ByteBuffer we can speed the checksum by calling the hadoop' checksum native lib; it's more faster.

Please also see: HBASE-22483

Don't forget to up your *HBASE_OFFHEAPSIZE* accordingly.

Offheap write-path

In hbase-2.x, HBASE-15179 made the HBase write path work off-heap. By default, the MemStores in HBase have always used MemStore Local Allocation Buffers (MSLABs) to avoid memory fragmentation; an MSLAB creates bigger fixed sized chunks and then the MemStores Cell's data gets copied into these MSLAB chunks. These chunks can be pooled also and from hbase-2.x on, the MSLAB pool is by default ON. Write off-heaping makes use of the MSLAB pool. It creates MSLAB chunks as Direct ByteBuffers and pools them.

`hbase.regionserver.offheap.global.memstore.size` is the configuration key which controls the amount of off-heap data. Its value is the number of megabytes of off-heap memory that should be used by MSLAB (e.g. `25` would result in 25MB of off-heap). Be sure to increase *HBASE_OFFHEAPSIZE* which will set the JVM's MaxDirectMemorySize property (see Tuning the RPC buffer pool for more on *HBASE_OFFHEAPSIZE*). The default value of `hbase.regionserver.offheap.global.memstore.size` is 0 which means MSLAB uses onheap, not offheap, chunks by default.

`hbase.hregion.memstore.mslab.chunksize` controls the size of each off-heap chunk. Default is `2097152` (2MB).

When a Cell is added to a MemStore, the bytes for that Cell are copied into these off-heap buffers (if `hbase.regionserver.offheap.global.memstore.size` is non-zero) and a Cell POJO will refer to this memory area. This can greatly reduce the on-heap occupancy of the MemStores and reduce the total heap utilization for RegionServers in a write-heavy workload. On-heap and off-heap memory utiliazation are tracked at multiple levels to implement low level and high level memory management. The decision to flush a MemStore considers both the on-heap and off-heap usage of that MemStore. At the Region level, we sum the on-heap and off-heap usages and compare them against the region flush size (128MB, by default). Globally, on-heap size occupancy of all memstores are tracked as well as off-heap size. When any of these sizes breache the lower mark (`hbase.regionserver.global.memstore.size.lower.limit`) or the maximum size `hbase.regionserver.global.memstore.size`), all regions are selected for forced flushes.

Backup and Restore

Overview

Backup and restore is a standard operation provided by many databases. An effective backup and restore strategy helps ensure that users can recover data in case of unexpected failures. The HBase backup and restore feature helps ensure that enterprises using HBase as a canonical data repository can recover from catastrophic failures. Another important feature is the ability to restore the database to a particular point-in-time, commonly referred to as a snapshot.

The HBase backup and restore feature provides the ability to create full backups and incremental backups on tables in an HBase cluster. The full backup is the foundation on which incremental backups are applied to build iterative snapshots. Incremental backups can be run on a schedule to capture changes over time, for example by using a Cron task. Incremental backups are more cost-effective than full backups because they only capture the changes since the last backup and they also enable administrators to restore the database to any prior incremental backup. Furthermore, the utilities also enable table-level data backup-and-recovery if you do not want to restore the entire dataset of the backup.

The backup and restore feature supplements the HBase Replication feature. While HBase replication is ideal for creating "hot" copies of the data (where the replicated data is immediately available for query), the backup and restore feature is ideal for creating "cold" copies of data (where a manual step must be taken to restore the system). Previously, users only had the ability to create full backups via the ExportSnapshot functionality. The incremental backup implementation is the novel improvement over the previous "art" provided by ExportSnapshot.

The backup and restore feature uses DistCp to transfer files between clusters . HADOOP-15850 fixes a bug where CopyCommitter#concatFileChunks unconditionally tried to concatenate the files being DistCp'ed to target cluster (though the files are independent) . Without the fix from HADOOP-15850 , the transfer would fail. So the backup and restore feature need hadoop version as below

- 2.7.x
- 2.8.x
- 2.9.2+

- 2.10.0+
- 3.0.4+
- 3.1.2+
- 3.2.0+
- 3.3.0+

Terminology

The backup and restore feature introduces new terminology which can be used to understand how control flows through the system.

- *A backup*: A logical unit of data and metadata which can restore a table to its state at a specific point in time.
- *Full backup*: a type of backup which wholly encapsulates the contents of the table at a point in time.
- *Incremental backup*: a type of backup which contains the changes in a table since a full backup.
- *Backup set*: A user-defined name which references one or more tables over which a backup can be executed.
- *Backup ID*: A unique names which identifies one backup from the rest, e.g. `backupId_1467823988425`

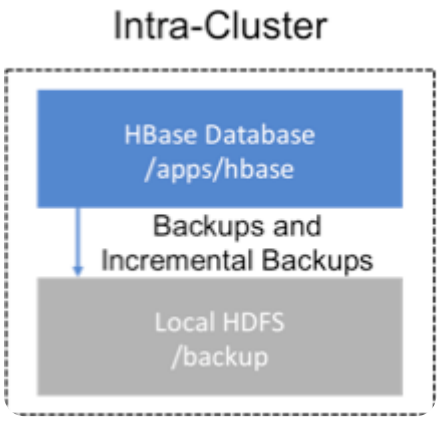
Planning

There are some common strategies which can be used to implement backup and restore in your environment. The following section shows how these strategies are implemented and identifies potential tradeoffs with each.

> ⚠ This backup and restore tools has not been tested on Transparent Data Encryption (TDE) enabled HDFS clusters. This is related to the open issue HBASE-16178.

Backup within a cluster

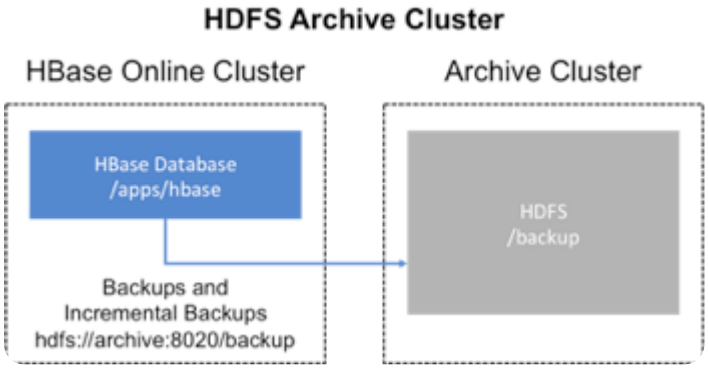
This strategy stores the backups on the same cluster as where the backup was taken. This approach is only appropriate for testing as it does not provide any additional safety on top of what the software itself already provides.



Backup using a dedicated cluster

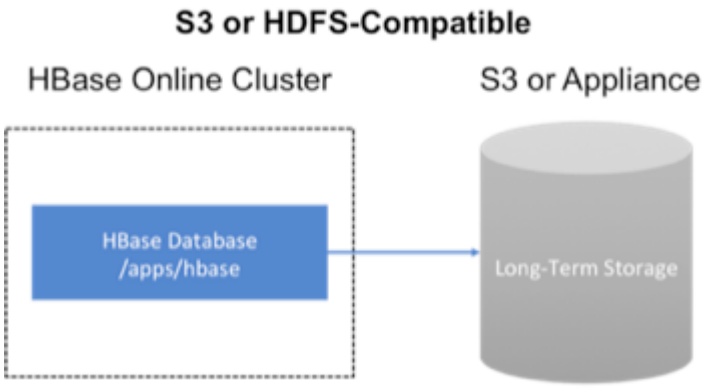
This strategy provides greater fault tolerance and provides a path towards disaster recovery. In this setting, you will store the backup on a separate HDFS cluster by supplying the backup destination cluster's HDFS URL to the backup utility. You should consider backing up to a different physical location, such as a different data center.

Typically, a backup-dedicated HDFS cluster uses a more economical hardware profile to save money.



Backup to the Cloud or a storage vendor appliance

Another approach to safeguarding HBase incremental backups is to store the data on provisioned, secure servers that belong to third-party vendors and that are located off-site. The vendor can be a public cloud provider or a storage vendor who uses a Hadoop-compatible file system, such as S3 and other HDFS-compatible destinations.



The HBase backup utility does not support backup to multiple destinations. A workaround is to manually create copies of the backup files from HDFS or S3.

First-time configuration steps

This section contains the necessary configuration changes that must be made in order to use the backup and restore feature. As this feature makes significant use of YARN's MapReduce framework to parallelize these I/O heavy operations, configuration changes extend outside of just `hbase-site.xml`.

Allow the "hbase" system user in YARN

The YARN *container-executor.cfg* configuration file must have the following property setting: *allowed.system.users=hbase*. No spaces are allowed in entries of this configuration file.

Skipping this step will result in runtime errors when executing the first backup tasks.

Example of a valid container-executor.cfg file for backup and restore:

```
yarn.nodemanager.log-dirs=/var/log/hadoop/mapred
yarn.nodemanager.linux-container-executor.group=yarn
banned.users=hdfs,yarn,mapred,bin
allowed.system.users=hbase
min.user.id=500
```

HBase specific changes

Add the following properties to hbase-site.xml and restart HBase if it is already running.

The ",..." is an ellipsis meant to imply that this is a comma-separated list of values, not literal text which should be added to hbase-site.xml.

```
<property>
  <name>hbase.backup.enable</name>
  <value>true</value>
</property>
<property>
  <name>hbase.master.logcleaner.plugins</name>
  <value>org.apache.hadoop.hbase.backup.master.BackupLogCleaner,...</value>
</property>
<property>
  <name>hbase.procedure.master.classes</name>
  <value>org.apache.hadoop.hbase.backup.master.LogRollMasterProcedureManager,...
</value>
</property>
<property>
  <name>hbase.procedure.regionserver.classes</name>
  <value>org.apache.hadoop.hbase.backup.regionserver.LogRollRegionServerProcedure
Manager,...</value>
</property>
<property>
  <name>hbase.coprocessor.region.classes</name>
  <value>org.apache.hadoop.hbase.backup.BackupObserver,...</value>
</property>
<property>
  <name>hbase.coprocessor.master.classes</name>
  <value>org.apache.hadoop.hbase.backup.BackupMasterObserver,...</value>
</property>
<property>
  <name>hbase.master.hfilecleaner.plugins</name>
  <value>org.apache.hadoop.hbase.backup.BackupHFileCleaner,...</value>
</property>
```

Backup and Restore commands

This covers the command-line utilities that administrators would run to create, restore, and merge backups. Tools to inspect details on specific backup sessions is covered in the next section, Administration of Backup Images.

Run the command `hbase backup help <command>` to access the online help that provides basic information about a command and its options. The below information is captured in this help message for each command.

Creating a Backup Image

> For HBase clusters also using Apache Phoenix: include the SQL system catalog tables in the backup. In the event that you need to restore the HBase backup, access to the system catalog tables enable you to resume Phoenix interoperability with the restored data.

The first step in running the backup and restore utilities is to perform a full backup and to store the data in a separate image from the source. At a minimum, you must do this to get a baseline before you can rely on incremental backups.

Run the following command as HBase superuser:

```
hbase backup create <type> <backup_path>
```

After the command finishes running, the console prints a SUCCESS or FAILURE status message. The SUCCESS message includes a *backup* ID. The backup ID is the Unix time (also known as Epoch time) that the HBase master received the backup request from the client.

> Record the backup ID that appears at the end of a successful backup. In case the source cluster fails and you need to recover the dataset with a restore operation, having the backup ID readily available can save time.

Positional Command-Line Arguments

type

The type of backup to execute: *full* or *incremental*. As a reminder, an *incremental* backup requires a *full* backup to already exist.

backup_path

The *backup_path* argument specifies the full filesystem URI of where to store the backup image. Valid prefixes are *hdfs:*, *webhdfs:*, *s3a:* or other compatible Hadoop File System implementations.

Named Command-Line Arguments

-t <table_name[,table_name]>

A comma-separated list of tables to back up. If no tables are specified, all tables are backed up. No regular-expression or wildcard support is present; all table names must be explicitly listed. See Backup Sets for more information about peforming operations on

collections of tables. Mutually exclusive with the *-s* option; one of these named options are required.

-s <backup_set_name>
Identify tables to backup based on a backup set. See Using Backup Sets for the purpose and usage of backup sets. Mutually exclusive with the *-t* option.

-w <number_workers>
(Optional) Specifies the number of parallel workers to copy data to backup destination. Backups are currently executed by MapReduce jobs so this value corresponds to the number of Mappers that will be spawned by the job.

-b <bandwidth_per_worker>
(Optional) Specifies the bandwidth of each worker in MB per second.

-d
(Optional) Enables "DEBUG" mode which prints additional logging about the backup creation.

-i
(Optional) Ignore checksum verify between source snapshot and exported snapshot. Especially when the source and target file system types are different, we should use -i option to skip checksum-checks.

-q <name>
(Optional) Allows specification of the name of a YARN queue which the MapReduce job to create the backup should be executed in. This option is useful to prevent backup tasks from stealing resources away from other MapReduce jobs of high importance.

Example usage

```
$ hbase backup create full hdfs://host5:9000/data/backup -t SALES2,SALES3 -w 3
```

This command creates a full backup image of two tables, SALES2 and SALES3, in the HDFS instance who NameNode is host5:9000 in the path */data/backup*. The *-w* option specifies that no more than three parallel works complete the operation.

Restoring a Backup Image

Run the following command as an HBase superuser. You can only restore a backup on a running HBase cluster because the data must be redistributed the RegionServers for the operation to complete successfully.

```
hbase restore <backup_path> <backup_id>
```

Positional Command-Line Arguments

backup_path

The *backup_path* argument specifies the full filesystem URI of where to store the backup image. Valid prefixes are *hdfs:*, *webhdfs:*, *s3a:* or other compatible Hadoop File System implementations.

backup_id

The backup ID that uniquely identifies the backup image to be restored.

Named Command-Line Arguments

-t <table_name[,table_name]>

A comma-separated list of tables to restore. See Backup Sets for more information about peforming operations on collections of tables. Mutually exclusive with the *-s* option; one of these named options are required.

-s <backup_set_name>

Identify tables to backup based on a backup set. See Using Backup Sets for the purpose and usage of backup sets. Mutually exclusive with the *-t* option.

-q <name>

(Optional) Allows specification of the name of a YARN queue which the MapReduce job to create the backup should be executed in. This option is useful to prevent backup tasks from stealing resources away from other MapReduce jobs of high importance.

-c

(Optional) Perform a dry-run of the restore. The actions are checked, but not executed.

-m <target_tables>

(Optional) A comma-separated list of tables to restore into. If this option is not provided,

the original table name is used. When this option is provided, there must be an equal number of entries provided in the `-t` option.

-o
(Optional) Overwrites the target table for the restore if the table already exists.

Example of Usage

```
hbase restore /tmp/backup_incremental backupId_1467823988425 -t mytable1,mytable2
```

This command restores two tables of an incremental backup image. In this example: • `/tmp/backup_incremental` is the path to the directory containing the backup image. • `backupId_1467823988425` is the backup ID. • `mytable1` and `mytable2` are the names of tables in the backup image to be restored.

> If the namespace of a table being restored does not exist in the target environment, it will be automatically created during the restore operation. HBASE-25707

Merging Incremental Backup Images

This command can be used to merge two or more incremental backup images into a single incremental backup image. This can be used to consolidate multiple, small incremental backup images into a single larger incremental backup image. This command could be used to merge hourly incremental backups into a daily incremental backup image, or daily incremental backups into a weekly incremental backup.

```
$ hbase backup merge <backup_ids>
```

Positional Command-Line Arguments

backup_ids
A comma-separated list of incremental backup image IDs that are to be combined into a single image.

Named Command-Line Arguments

None.

Example usage

```
$ hbase backup merge backupId_1467823988425,backupId_1467827588425
```

Using Backup Sets

Backup sets can ease the administration of HBase data backups and restores by reducing the amount of repetitive input of table names. You can group tables into a named backup set with the `hbase backup set add` command. You can then use the `-set` option to invoke the name of a backup set in the `hbase backup create` or `hbase restore` rather than list individually every table in the group. You can have multiple backup sets.

> Note the differentiation between the `hbase backup set add` command and the *-set* option. The `hbase backup set add` command must be run before using the `-set` option in a different command because backup sets must be named and defined before using backup sets as a shortcut.

If you run the `hbase backup set add` command and specify a backup set name that does not yet exist on your system, a new set is created. If you run the command with the name of an existing backup set name, then the tables that you specify are added to the set.

In this command, the backup set name is case-sensitive.

> The metadata of backup sets are stored within HBase. If you do not have access to the original HBase cluster with the backup set metadata, then you must specify individual table names to restore the data.

To create a backup set, run the following command as the HBase superuser:

```
$ hbase backup set <subcommand> <backup_set_name> <tables>
```

Backup Set Subcommands

The following list details subcommands of the hbase backup set command.

> You must enter one (and no more than one) of the following subcommands after hbase backup set to complete an operation. Also, the backup set name is case-sensitive in the command-line utility.

add

Adds table[s] to a backup set. Specify a *backup_set_name* value after this argument to create a backup set.

remove

Removes tables from the set. Specify the tables to remove in the tables argument.

list

Lists all backup sets.

describe

Displays a description of a backup set. The information includes whether the set has full or incremental backups, start and end times of the backups, and a list of the tables in the set. This subcommand must precede a valid value for the *backup_set_name* value.

delete

Deletes a backup set. Enter the value for the *backup_set_name* option directly after the `hbase backup set delete` command.

Positional Command-Line Arguments

backup_set_name

Use to assign or invoke a backup set name. The backup set name must contain only printable characters and cannot have any spaces.

tables

List of tables (or a single table) to include in the backup set. Enter the table names as a comma-separated list. If no tables are specified, all tables are included in the set.

> Maintain a log or other record of the case-sensitive backup set names and the corresponding tables in each set on a separate or remote cluster, backup strategy. This information can help you in case of failure on the primary cluster.

Example of Usage

```
$ hbase backup set add Q1Data TEAM3,TEAM_4
```

Depending on the environment, this command results in *one* of the following actions:

- If the `Q1Data` backup set does not exist, a backup set containing tables `TEAM_3` and `TEAM_4` is created.
- If the `Q1Data` backup set exists already, the tables `TEAM_3` and `TEAM_4` are added to the `Q1Data` backup set.

Administration of Backup Images

The `hbase backup` command has several subcommands that help with administering backup images as they accumulate. Most production environments require recurring backups, so it is necessary to have utilities to help manage the data of the backup repository. Some subcommands enable you to find information that can help identify backups that are relevant in a search for particular data. You can also delete backup images.

The following list details each `hbase backup subcommand` that can help administer backups. Run the full command-subcommand line as the HBase superuser.

Managing Backup Progress

You can monitor a running backup in another terminal session by running the *hbase backup progress* command and specifying the backup ID as an argument.

For example, run the following command as hbase superuser to view the progress of a backup

```
$ hbase backup progress <backup_id>
```

Positional Command-Line Arguments

backup_id
Specifies the backup that you want to monitor by seeing the progress information. The backupId is case-sensitive.

Named Command-Line Arguments

None.

Example usage

```
hbase backup progress backupId_1467823988425
```

Managing Backup History

This command displays a log of backup sessions. The information for each session includes backup ID, type (full or incremental), the tables in the backup, status, and start and end time. Specify the number of backup sessions to display with the optional -n argument.

```
$ hbase backup history <backup_id>
```

Positional Command-Line Arguments

backup_id
Specifies the backup that you want to monitor by seeing the progress information. The backupId is case-sensitive.

Named Command-Line Arguments

-n <num_records>
(Optional) The maximum number of backup records (Default: 10).

-p <backup_root_path>
The full filesystem URI of where backup images are stored.

-s <backup_set_name>
The name of the backup set to obtain history for. Mutually exclusive with the *-t* option.

-t <table_name>
The name of table to obtain history for. Mutually exclusive with the *-s* option.

Example usage

```
$ hbase backup history
$ hbase backup history -n 20
$ hbase backup history -t WebIndexRecords
```

Describing a Backup Image

This command can be used to obtain information about a specific backup image.

```
$ hbase backup describe <backup_id>
```

Positional Command-Line Arguments

backup_id The ID of the backup image to describe.

Named Command-Line Arguments

None.

Example usage

```
$ hbase backup describe backupId_1467823988425
```

Deleting Backup Images

The `hbase backup delete` command deletes backup images that are no longer needed.

Syntax

```
$ hbase backup delete -l <backup_id1,backup_id2,...>
$ hbase backup delete -k <days>
```

Named Command-Line Arguments

-l <backup_id1,backup_id2,...>
Comma-separated list of backup IDs to delete.

-k <days>
Deletes all backup images completed more than the specified number of days ago.

> These options are **mutually exclusive**. Only one of `-l` or `-k` may be used at a time.

Example Usage

Delete specific backup images by ID:

```
$ hbase backup delete -l backupId_1467823988425,backupId_1467824989999
```

Delete all backup images older than 30 days:

```
$ hbase backup delete -k 30
```

⚠

- Deleting a backup may affect all following incremental backups (in the same backup root) up to the next full backup. For example, if you take a full backup every 2 weeks and daily incremental backups, running `hbase backup delete -k 7` when the full backup is older than 7 days will effectively remove the data for all subsequent incremental backups. The backup IDs may still be listed, but their data will be gone.
- If the most recent backup is an incremental backup and you delete it, you should run a **full backup** next. Running another incremental backup immediately after may result in missing data in the backup image. (See HBASE-28084)

Backup Repair Command

This command attempts to correct any inconsistencies in persisted backup metadata which exists as the result of software errors or unhandled failure scenarios. While the backup implementation tries to correct all errors on its own, this tool may be necessary in the cases where the system cannot automatically recover on its own.

```
$ hbase backup repair
```

Positional Command-Line Arguments

None.

Named Command-Line Arguments

None.

Example usage

```
$ hbase backup repair
```

Additional Topics

Configuration keys

The backup and restore feature includes both required and optional configuration keys.

Required properties

hbase.backup.enable: Controls whether or not the feature is enabled (Default: `false`). Set this value to `true`.

hbase.master.logcleaner.plugins: A comma-separated list of classes invoked when cleaning logs in the HBase Master. Set this value to `org.apache.hadoop.hbase.backup.master.BackupLogCleaner` or append it to the current value.

hbase.procedure.master.classes: A comma-separated list of classes invoked with the Procedure framework in the Master. Set this value to `org.apache.hadoop.hbase.backup.master.LogRollMasterProcedureManager` or append it to the current value.

hbase.procedure.regionserver.classes: A comma-separated list of classes invoked with the Procedure framework in the RegionServer. Set this value to `org.apache.hadoop.hbase.backup.regionserver.LogRollRegionServerProcedureManager` or append it to the current value.

hbase.coprocessor.region.classes: A comma-separated list of RegionObservers deployed on tables. Set this value to `org.apache.hadoop.hbase.backup.BackupObserver` or append it to the current value.

hbase.coprocessor.master.classes: A comma-separated list of MasterObservers deployed on tables. Set this value to `org.apache.hadoop.hbase.backup.BackupMasterObserver` or append it to the current value.

hbase.master.hfilecleaner.plugins: A comma-separated list of HFileCleaners deployed on the Master. Set this value to `org.apache.hadoop.hbase.backup.BackupHFileCleaner` or append it to the current value.

Optional properties

hbase.backup.system.ttl: The time-to-live in seconds of data in the `hbase:backup` tables (default: forever). This property is only relevant prior to the creation of the `hbase:backup` table. Use the `alter` command in the HBase shell to modify the TTL when this table already exists. See the below section for more details on the impact of this configuration property.

hbase.backup.attempts.max: The number of attempts to perform when taking hbase table snapshots (default: 10).

hbase.backup.attempts.pause.ms: The amount of time to wait between failed snapshot attempts in milliseconds (default: 10000).

hbase.backup.logroll.timeout.millis: The amount of time (in milliseconds) to wait for RegionServers to execute a WAL rolling in the Master's procedure framework (default: 30000).

Best Practices

Formulate a restore strategy and test it.

Before you rely on a backup and restore strategy for your production environment, identify how backups must be performed, and more importantly, how restores must be performed. Test the plan to ensure that it is workable. At a minimum, store backup data from a production cluster on a different cluster or server. To further safeguard the data, use a backup location that is at a different physical location.

If you have a unrecoverable loss of data on your primary production cluster as a result of computer system issues, you may be able to restore the data from a different cluster or server at the same site. However, a disaster that destroys the whole site renders locally stored backups useless. Consider storing the backup data and necessary resources (both computing capacity and operator expertise) to restore the data at a site sufficiently remote from the production site. In the case of a catastrophe at the whole primary site (fire, earthquake, etc.), the remote backup site can be very valuable.

Secure a full backup image first.

As a baseline, you must complete a full backup of HBase data at least once before you can rely on incremental backups. The full backup should be stored outside of the source cluster. To ensure complete dataset recovery, you must run the restore utility with the option to restore baseline full backup. The full backup is the foundation of your dataset. Incremental backup data is applied on top of the full backup during the restore operation to return you to the point in time when backup was last taken.

Define and use backup sets for groups of tables that are logical subsets of the entire dataset.

You can group tables into an object called a backup set. A backup set can save time when you have a particular group of tables that you expect to repeatedly back up or restore.

When you create a backup set, you type table names to include in the group. The backup set includes not only groups of related tables, but also retains the HBase backup metadata. Afterwards, you can invoke the backup set name to indicate what tables apply to the command execution instead of entering all the table names individually.

Document the backup and restore strategy, and ideally log information about each backup.

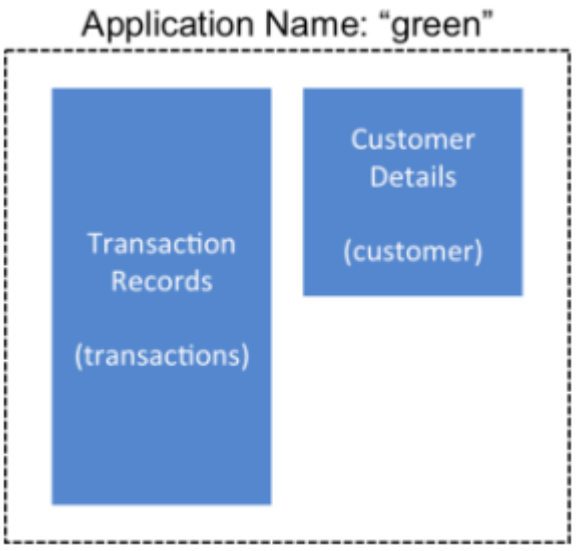
Document the whole process so that the knowledge base can transfer to new administrators after employee turnover. As an extra safety precaution, also log the calendar date, time, and other relevant details about the data of each backup. This metadata can potentially help locate a particular dataset in case of source cluster failure or primary site disaster. Maintain duplicate copies of all documentation: one copy at the production cluster site and another at the backup location or wherever it can be accessed by an administrator remotely from the production cluster.

Scenario: Safeguarding Application Datasets on Amazon S3

This scenario describes how a hypothetical retail business uses backups to safeguard application data and then restore the dataset after failure.

The HBase administration team uses backup sets to store data from a group of tables that have interrelated information for an application called green. In this example, one table contains transaction records and the other contains customer details. The two tables need to be backed up and be recoverable as a group.

The admin team also wants to ensure daily backups occur automatically.



The following is an outline of the steps and examples of commands that are used to backup the data for the *green* application and to recover the data later. All commands are run when logged in as HBase superuser.

- A backup set called *green_set* is created as an alias for both the transactions table and the customer table. The backup set can be used for all operations to avoid typing each table name. The backup set name is case-sensitive and should be formed with only printable characters and without spaces.

```
$ hbase backup set add green_set transactions
$ hbase backup set add green_set customer
```

- The first backup of green_set data must be a full backup. The following command example shows how credentials are passed to Amazon S3 and specifies the file system with the s3a: prefix.

```
$ ACCESS_KEY=ABCDEFGHIJKLMNOPQRST
$ SECRET_KEY=123456789abcdefghijklmnopqrstuvwxyzABCD
$ sudo -u hbase hbase backup create full\
  s3a://$ACCESS_KEY:SECRET_KEY@prodhbasebackups/backups -s green_set
```

- Incremental backups should be run according to a schedule that ensures essential data recovery in the event of a catastrophe. At this retail company, the HBase admin team decides that automated daily backups secures the data sufficiently. The team decides that they can implement this by modifying an existing Cron job that is defined in `/etc/crontab`. Consequently, IT modifies the Cron job by adding the following line:

```
@daily hbase hbase backup create incremental s3a://$ACCESS_KEY:$SECRET_KEY@prodhbasebackups/backups -s green_set
```

- A catastrophic IT incident disables the production cluster that the green application uses. An HBase system administrator of the backup cluster must restore the *green_set* dataset to the point in time closest to the recovery objective.

> If the administrator of the backup HBase cluster has the backup ID with relevant details in accessible records, the following search with the `hdfs dfs -ls` command and manually scanning the backup ID list can be bypassed. Consider continuously maintaining and protecting a detailed log of backup IDs outside the production cluster in your environment.

  The HBase administrator runs the following command on the directory where backups are stored to print the list of successful backup IDs on the console:

```
hdfs dfs -ls -t /prodhbasebackups/backups
```

- The admin scans the list to see which backup was created at a date and time closest to the recovery objective. To do this, the admin converts the calendar timestamp of the recovery point in time to Unix time because backup IDs are uniquely identified with Unix time. The backup IDs are listed in reverse chronological order, meaning the most recent successful backup appears first.

  The admin notices that the following line in the command output corresponds with the *green_set* backup that needs to be restored:

```
/prodhbasebackups/backups/backup_1467823988425`
```

- The admin restores green_set invoking the backup ID and the -overwrite option. The -overwrite option truncates all existing data in the destination and populates the tables with data from the backup dataset. Without this flag, the backup data is appended to the

existing data in the destination. In this case, the admin decides to overwrite the data because it is corrupted.

```
$ sudo -u hbase hbase restore -s green_set \
  s3a://$ACCESS_KEY:$SECRET_KEY@prodhbasebackups/backups backup_1467823988425 \
-overwrite
```

Security of Backup Data

With this feature which makes copying data to remote locations, it's important to take a moment to clearly state the procedural concerns that exist around data security. Like the HBase replication feature, backup and restore provides the constructs to automatically copy data from within a corporate boundary to some system outside of that boundary. It is imperative when storing sensitive data that with backup and restore, much less any feature which extracts data from HBase, the locations to which data is being sent has undergone a security audit to ensure that only authenticated users are allowed to access that data.

For example, with the above example of backing up data to S3, it is of the utmost importance that the proper permissions are assigned to the S3 bucket to ensure that only a minimum set of authorized users are allowed to access this data. Because the data is no longer being accessed via HBase, and its authentication and authorization controls, we must ensure that the filesystem storing that data is providing a comparable level of security. This is a manual step which users **must** implement on their own.

Technical Details of Incremental Backup and Restore

HBase incremental backups enable more efficient capture of HBase table images than previous attempts at serial backup and restore solutions, such as those that only used HBase Export and Import APIs. Incremental backups use Write Ahead Logs (WALs) to capture the data changes since the previous backup was created. A WAL roll (create new WALs) is executed across all RegionServers to track the WALs that need to be in the backup. In addition to WALs, incremental backups also track bulk-loaded HFiles for tables under backup.

Incremental backup gathers all WAL files generated since the last backup from the source cluster, converts them to HFiles in a `.tmp` directory under the `BACKUP_ROOT`, and then

moves these HFiles to their final location under the backup root directory to form the backup image. It also reads bulk load records from the backup system table, forms the paths for the corresponding bulk-loaded HFiles, and copies those files to the backup destination. Bulk-loaded files are preserved (not deleted by cleaner chores) until they've been included in a backup (for each backup root). A process similar to the DistCp (distributed copy) tool is used to move the backup files to the target file system.

When a table restore operation starts, a two-step process is initiated. First, the full backup is restored from the full backup image. Second, all HFiles from incremental backups between the last full backup and the incremental backup being restored (including bulk-loaded HFiles) are bulk loaded into the table using the HBase Bulk Load utility.

You can only restore on a live HBase cluster because the data must be redistributed to complete the restore operation successfully.

A Warning on File System Growth

As a reminder, incremental backups are implemented via retaining the write-ahead logs which HBase primarily uses for data durability. Thus, to ensure that all data needing to be included in a backup is still available in the system, the HBase backup and restore feature retains all write-ahead logs since the last backup until the next incremental backup is executed.

Like HBase Snapshots, this can have an expectedly large impact on the HDFS usage of HBase for high volume tables. Take care in enabling and using the backup and restore feature, specifically with a mind to removing backup sessions when they are not actively being used.

The only automated, upper-bound on retained write-ahead logs for backup and restore is based on the TTL of the `hbase:backup` system table which, as of the time this document is written, is infinite (backup table entries are never automatically deleted). This requires that administrators perform backups on a schedule whose frequency is relative to the amount of available space on HDFS (e.g. less available HDFS space requires more aggressive backup merges and deletions). As a reminder, the TTL can be altered on the `hbase:backup` table using the `alter` command in the HBase shell. Modifying the configuration property `hbase.backup.system.ttl` in hbase-site.xml after the system table exists has no effect.

Capacity Planning

When designing a distributed system deployment, it is critical that some basic mathmatical rigor is executed to ensure sufficient computational capacity is available given the data and software requirements of the system. For this feature, the availability of network capacity is the largest bottleneck when estimating the performance of some implementation of backup and restore. The second most costly function is the speed at which data can be read/written.

Full Backups

To estimate the duration of a full backup, we have to understand the general actions which are invoked:

- Write-ahead log roll on each RegionServer: ones to tens of seconds per RegionServer in parallel. Relative to the load on each RegionServer.
- Take an HBase snapshot of the table(s): tens of seconds. Relative to the number of regions and files that comprise the table.
- Export the snapshot to the destination: see below. Relative to the size of the data and the network bandwidth to the destination.

To approximate how long the final step will take, we have to make some assumptions on hardware. Be aware that these will *not* be accurate for your system — these are numbers that your or your administrator know for your system. Let's say the speed of reading data from HDFS on a single node is capped at 80MB/s (across all Mappers that run on that host), a modern network interface controller (NIC) supports 10Gb/s, the top-of-rack switch can handle 40Gb/s, and the WAN between your clusters is 10Gb/s. This means that you can only ship data to your remote at a speed of 1.25GB/s — meaning that 16 nodes (`1.25 * 1024 / 80 = 16`) participating in the ExportSnapshot should be able to fully saturate the link between clusters. With more nodes in the cluster, we can still saturate the network but at a lesser impact on any one node which helps ensure local SLAs are made. If the size of the snapshot is 10TB, this would full backup would take in the ballpark of 2.5 hours (`10 * 1024 / 1.25 / (60 * 60) = 2.23hrs`)

As a general statement, it is very likely that the WAN bandwidth between your local cluster and the remote storage is the largest bottleneck to the speed of a full backup.

When the concern is restricting the computational impact of backups to a "production system", the above formulas can be reused with the optional command-line arguments to `hbase backup create`: `-b`, `-w`, `-q`. The `-b` option defines the bandwidth at which each worker (Mapper) would write data. The `-w` argument limits the number of workers that would be spawned in the DistCp job. The `-q` allows the user to specify a YARN queue which can limit the specific nodes where the workers will be spawned — this can quarantine the backup workers performing the copy to a set of non-critical nodes. Relating the `-b` and `-w` options to our earlier equations: `-b` would be used to restrict each node from reading data at the full 80MB/s and `-w` is used to limit the job from spawning 16 worker tasks.

Incremental Backup

Like we did for full backups, we have to understand the incremental backup process to approximate its runtime and cost.

- Identify new write-ahead logs since the last full or incremental backup: negligible. Apriori knowledge from the backup system table(s).
- Read, filter, and write "minimized" HFiles equivalent to the WALs: dominated by the speed of writing data. Relative to write speed of HDFS.
- Read bulk load records from the backup system table, form the paths for bulk-loaded HFiles, and copy them to the backup destination.
- DistCp the HFiles to the destination: see above.

For the second step, the dominating cost of this operation would be the re-writing the data (under the assumption that a majority of the data in the WAL is preserved). In this case, we can assume an aggregate write speed of 30MB/s per node. Continuing our 16-node cluster example, this would require approximately 15 minutes to perform this step for 50GB of data (50 * 1024 / 60 / 60 = 14.2). The amount of time to start the DistCp MapReduce job would likely dominate the actual time taken to copy the data (50 / 1.25 = 40 seconds) and can be ignored.

Limitations of the Backup and Restore Utility

Serial backup operations

Backup operations cannot be run concurrently. An operation includes actions like create,

delete, restore, and merge. Only one active backup session is supported. HBASE-16391 will introduce multiple-backup sessions support.

No means to cancel backups

Both backup and restore operations cannot be canceled. (HBASE-15997, HBASE-15998). The workaround to cancel a backup would be to kill the client-side backup command (`control-C`), ensure all relevant MapReduce jobs have exited, and then run the `hbase backup repair` command to ensure the system backup metadata is consistent.

Backups can only be saved to a single location

Copying backup information to multiple locations is an exercise left to the user. HBASE-15476 will introduce the ability to specify multiple-backup destinations intrinsically.

HBase superuser access is required

Only an HBase superuser (e.g. hbase) is allowed to perform backup/restore, can pose a problem for shared HBase installations. Current mitigations would require coordination with system administrators to build and deploy a backup and restore strategy (HBASE-14138).

Backup restoration is an online operation

To perform a restore from a backup, it requires that the HBase cluster is online as a caveat of the current implementation (HBASE-16573).

Some operations may fail and require re-run

The HBase backup feature is primarily client driven. While there is the standard HBase retry logic built into the HBase Connection, persistent errors in executing operations may propagate back to the client (e.g. snapshot failure due to region splits). The backup implementation should be moved from client-side into the ProcedureV2 framework in the future which would provide additional robustness around transient/retryable failures. The `hbase backup repair` command is meant to correct states which the system cannot automatically detect and recover from.

Avoidance of declaration of public API

While the Java API to interact with this feature exists and its implementation is separated from an interface, insufficient rigor has been applied to determine if it is exactly what we intend to ship to users. As such, it is marked as for a `Private` audience with the expectation that, as users begin to try the feature, there will be modifications that would necessitate breaking compatibility (HBASE-17517).

Lack of global metrics for backup and restore

Individual backup and restore operations contain metrics about the amount of work the operation included, but there is no centralized location (e.g. the Master UI) which present information for consumption (HBASE-16565).

Synchronous Replication

Background

The current Cluster Replication in HBase in asynchronous. So if the master cluster crashes, the slave cluster may not have the newest data. If users want strong consistency then they can not switch to the slave cluster.

Design

Please see the design doc on HBASE-19064

Operation and maintenance

Case.1 Setup two synchronous replication clusters

- Add a synchronous peer in both source cluster and peer cluster.

For source cluster:

```
hbase> add_peer  '1', CLUSTER_KEY => 'lg-hadoop-tst-st01.bj:10010,lg-hadoop-tst-s
t02.bj:10010,lg-hadoop-tst-st03.bj:10010:/hbase/test-hbase-slave', REMOTE_WAL_DIR
=>'hdfs://lg-hadoop-tst-st01.bj:20100/hbase/test-hbase-slave/remoteWALs', TABLE_C
FS => {"ycsb-test"=>[]}
```

For peer cluster:

```
hbase> add_peer  '1', CLUSTER_KEY => 'lg-hadoop-tst-st01.bj:10010,lg-hadoop-tst-s
t02.bj:10010,lg-hadoop-tst-st03.bj:10010:/hbase/test-hbase', REMOTE_WAL_DIR=>'hdf
s://lg-hadoop-tst-st01.bj:20100/hbase/test-hbase/remoteWALs', TABLE_CFS => {"ycsb
-test"=>[]}
```

> For synchronous replication, the current implementation require that we have the same peer id for both source and peer cluster. Another thing that need attention is: the peer does not support cluster-level, namespace-level, or cf-level replication, only support table-level replication now.

- Transit the peer cluster to be STANDBY state

```
hbase> transit_peer_sync_replication_state '1', 'STANDBY'
```

- Transit the source cluster to be ACTIVE state

```
hbase> transit_peer_sync_replication_state '1', 'ACTIVE'
```

Now, the synchronous replication has been set up successfully. the HBase client can only request to source cluster, if request to peer cluster, the peer cluster which is STANDBY state now will reject the read/write requests.

Case.2 How to operate when standby cluster crashed

If the standby cluster has been crashed, it will fail to write remote WAL for the active cluster. So we need to transit the source cluster to DOWNGRANDE_ACTIVE state, which means source cluster won't write any remote WAL any more, but the normal replication (asynchronous Replication) can still work fine, it queue the newly written WALs, but the replication block until the peer cluster come back.

```
hbase> transit_peer_sync_replication_state '1', 'DOWNGRADE_ACTIVE'
```

Once the peer cluster come back, we can just transit the source cluster to ACTIVE, to ensure that the replication will be synchronous.

```
hbase> transit_peer_sync_replication_state '1', 'ACTIVE'
```

Case.3 How to operate when active cluster crashed

If the active cluster has been crashed (it may be not reachable now), so let's just transit the standby cluster to DOWNGRADE_ACTIVE state, and after that, we should redirect all the requests from client to the DOWNGRADE_ACTIVE cluster.

```
hbase> transit_peer_sync_replication_state '1', 'DOWNGRADE_ACTIVE'
```

If the crashed cluster come back again, we just need to transit it to STANDBY directly. Otherwise if you transit the cluster to DOWNGRADE_ACTIVE, the original ACTIVE cluster may have redundant data compared to the current ACTIVE cluster. Because we designed

to write source cluster WALs and remote cluster WALs concurrently, so it's possible that the source cluster WALs has more data than the remote cluster, which result in data inconsistency. The procedure of transiting ACTIVE to STANDBY has no problem, because we'll skip to replay the original WALs.

```
hbase> transit_peer_sync_replication_state '1', 'STANDBY'
```

After that, we can promote the DOWNGRADE_ACTIVE cluster to ACTIVE now, to ensure that the replication will be synchronous.

```
hbase> transit_peer_sync_replication_state '1', 'ACTIVE'
```

Apache HBase APIs

This information is not exhaustive, and provides a quick reference in addition to the User API Reference. The examples here are not comprehensive or complete, and should be used for purposes of illustration only.

Apache HBase also works with multiple external APIs. See Apache HBase External APIs for more information.

Examples

Create, modify and delete a Table Using Java

```
package com.example.hbase.admin;

import java.io.IOException;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.hbase.HBaseConfiguration;
import org.apache.hadoop.hbase.HColumnDescriptor;
import org.apache.hadoop.hbase.HConstants;
import org.apache.hadoop.hbase.HTableDescriptor;
import org.apache.hadoop.hbase.TableName;
import org.apache.hadoop.hbase.client.Admin;
import org.apache.hadoop.hbase.client.Connection;
import org.apache.hadoop.hbase.client.ConnectionFactory;
import org.apache.hadoop.hbase.io.compress.Compression.Algorithm;

public class Example {

  private static final String TABLE_NAME = "MY_TABLE_NAME_TOO";
  private static final String CF_DEFAULT = "DEFAULT_COLUMN_FAMILY";

  public static void createOrOverwrite(Admin admin, HTableDescriptor table) throw
s IOException {
    if (admin.tableExists(table.getTableName())) {
      admin.disableTable(table.getTableName());
      admin.deleteTable(table.getTableName());
    }
    admin.createTable(table);
  }

  public static void createSchemaTables(Configuration config) throws IOException
{
    try (Connection connection = ConnectionFactory.createConnection(config);
         Admin admin = connection.getAdmin()) {
```

```
      HTableDescriptor table = new HTableDescriptor(TableName.valueOf(TABLE_NAM
E));
      table.addFamily(new HColumnDescriptor(CF_DEFAULT).setCompressionType(Algori
thm.NONE));

      System.out.print("Creating table. ");
      createOrOverwrite(admin, table);
      System.out.println(" Done.");
    }
  }

  public static void modifySchema (Configuration config) throws IOException {
    try (Connection connection = ConnectionFactory.createConnection(config);
         Admin admin = connection.getAdmin()) {

      TableName tableName = TableName.valueOf(TABLE_NAME);
      if (!admin.tableExists(tableName)) {
        System.out.println("Table does not exist.");
        System.exit(-1);
      }

      HTableDescriptor table = admin.getTableDescriptor(tableName);

      // Update existing table
      HColumnDescriptor newColumn = new HColumnDescriptor("NEWCF");
      newColumn.setCompactionCompressionType(Algorithm.GZ);
      newColumn.setMaxVersions(HConstants.ALL_VERSIONS);
      admin.addColumn(tableName, newColumn);

      // Update existing column family
      HColumnDescriptor existingColumn = new HColumnDescriptor(CF_DEFAULT);
      existingColumn.setCompactionCompressionType(Algorithm.GZ);
      existingColumn.setMaxVersions(HConstants.ALL_VERSIONS);
      table.modifyFamily(existingColumn);
      admin.modifyTable(tableName, table);

      // Disable an existing table
      admin.disableTable(tableName);

      // Delete an existing column family
      admin.deleteColumn(tableName, CF_DEFAULT.getBytes("UTF-8"));

      // Delete a table (Need to be disabled first)
      admin.deleteTable(tableName);
    }
  }

  public static void main(String... args) throws IOException {
    Configuration config = HBaseConfiguration.create();

    //Add any necessary configuration files (hbase-site.xml, core-site.xml)
    config.addResource(new Path(System.getenv("HBASE_CONF_DIR"), "hbase-site.xm
l"));
    config.addResource(new Path(System.getenv("HADOOP_CONF_DIR"), "core-site.xm
l"));
    createSchemaTables(config);
```

```
    modifySchema(config);
  }
}
```

Apache HBase External APIs

For information on using the native HBase APIs, refer to User API Reference and the HBase APIs chapter.

REST

Representational State Transfer (REST) was introduced in 2000 in the doctoral dissertation of Roy Fielding, one of the principal authors of the HTTP specification.

REST itself is out of the scope of this documentation, but in general, REST allows client-server interactions via an API that is tied to the URL itself. This section discusses how to configure and run the REST server included with HBase, which exposes HBase tables, rows, cells, and metadata as URL specified resources. There is also a nice series of blogs on How-to: Use the Apache HBase REST Interface by Jesse Anderson.

Starting and Stopping the REST Server

The included REST server can run as a daemon which starts an embedded Jetty servlet container and deploys the servlet into it. Use one of the following commands to start the REST server in the foreground or background. The port is optional, and defaults to 8080.

```
# Foreground
$ bin/hbase rest start -p <port>

# Background, logging to a file in $HBASE_LOGS_DIR
$ bin/hbase-daemon.sh start rest -p <port>
```

To stop the REST server, use Ctrl-C if you were running it in the foreground, or the following command if you were running it in the background.

```
$ bin/hbase-daemon.sh stop rest
```

Configuring the REST Server and Client

For information about configuring the REST server and client for SSL, as well as `doAs` impersonation for the REST server, see Configure the Thrift Gateway to Authenticate on Behalf of the Client and other portions of the Securing Apache HBase chapter.

Using REST Endpoints

The following examples use the placeholder server http://example.com:8000, and the following commands can all be run using `curl` or `wget` commands. You can request plain text (the default), XML , or JSON output by adding no header for plain text, or the header "Accept: text/xml" for XML, "Accept: application/json" for JSON, or "Accept: application/x-protobuf" to for protocol buffers.

> Unless specified, use `GET` requests for queries, `PUT` or `POST` requests for creation or mutation, and `DELETE` for deletion.

Cluster-Wide Endpoints

Endpoint	HTTP Verb	Description
`/version/cluster`	`GET`	Version of HBase running on thi s cluster
`/version/rest`	`GET`	Version of the HBase REST Ser ver
`/status/cluster`	`GET`	Cluster status
`/`	`GET`	List of all non-system tables

Examples:

```
# Get cluster version
curl -vi -X GET \
  -H "Accept: text/xml" \
  "http://example.com:8000/version/cluster"

# Get REST server version
curl -vi -X GET \
  -H "Accept: text/xml" \
  "http://example.com:8000/version/rest"

# Get cluster status
curl -vi -X GET \
  -H "Accept: text/xml" \
  "http://example.com:8000/status/cluster"

# List all non-system tables
curl -vi -X GET \
  -H "Accept: text/xml" \
  "http://example.com:8000/"
```

Namespace Endpoints

Endpoint	HTTP Verb	Description
`/namespaces`	`GET`	List all namespaces
`/namespaces/namespace`	`GET`	Describe a specific namespace
`/namespaces/namespace`	`POST`	Create a new namespace
`/namespaces/namespace/tables`	`GET`	List all tables in a specific name space
`/namespaces/namespace`	`PUT`	Alter an existing namespace. Currently not used.
`/namespaces/namespace`	`DELETE`	Delete a namespace. The name space must be empty.

Examples:

```
# List all namespaces
curl -vi -X GET \
  -H "Accept: text/xml" \
  "http://example.com:8000/namespaces/"

# Describe a specific namespace
curl -vi -X GET \
  -H "Accept: text/xml" \
  "http://example.com:8000/namespaces/special_ns"

# Create a new namespace
curl -vi -X POST \
  -H "Accept: text/xml" \
  "example.com:8000/namespaces/special_ns"

# List all tables in a specific namespace
curl -vi -X GET \
  -H "Accept: text/xml" \
  "http://example.com:8000/namespaces/special_ns/tables"

# Alter an existing namespace
curl -vi -X PUT \
  -H "Accept: text/xml" \
  "http://example.com:8000/namespaces/special_ns

# Delete a namespace
curl -vi -X DELETE \
  -H "Accept: text/xml" \
  "example.com:8000/namespaces/special_ns"
```

Table Endpoints

Endpoint	HTTP Verb	Description
`/table/exists`	`GET`	Returns if the specified table e xists.
`/table/schema`	`GET`	Describe the schema of the sp ecified table.
`/table/schema`	`POST`	Update an existing table with t he provided schema fragment
`/table/schema`	`PUT`	Create a new table, or replace an existing table's schema
`/table/schema`	`DELETE`	Delete the table. You must use the `/table/schema` endpoint, not just `/table/`.
`/table/regions`	`GET`	List the table regions

Examples:

```
# Check if table exists
curl -vi -X GET \
  -H "Accept: text/xml" \
  "http://example.com:8000/users/exists"

# Get table schema
curl -vi -X GET \
  -H "Accept: text/xml" \
  "http://example.com:8000/users/schema"

# Update table schema
curl -vi -X POST \
  -H "Accept: text/xml" \
  -H "Content-Type: text/xml" \
  -d '<?xml version="1.0" encoding="UTF-8"?><TableSchema name="users"><ColumnSche
ma name="cf" KEEP_DELETED_CELLS="true" /></TableSchema>' \
  "http://example.com:8000/users/schema"

# Create or replace table schema
curl -vi -X PUT \
  -H "Accept: text/xml" \
  -H "Content-Type: text/xml" \
  -d '<?xml version="1.0" encoding="UTF-8"?><TableSchema name="users"><ColumnSche
ma name="cf" /></TableSchema>' \
  "http://example.com:8000/users/schema"

# Delete table
curl -vi -X DELETE \
```

```
  -H "Accept: text/xml" \
  "http://example.com:8000/users/schema"

# List table regions
curl -vi -X GET \
  -H "Accept: text/xml" \
  "http://example.com:8000/users/regions"
```

Endpoints for `Get` Operations

Endpoint	HTTP Verb	Description
`/table/row`	`GET`	Get all columns of a single row. Values are Base-64 encoded. T his requires the "Accept" reque st header with a type that can hold multiple columns (like xml, json or protobuf).
`/table/row/column:qualifie r/timestamp`	`GET`	Get the value of a single colum n. Values are Base-64 encode d.
`/table/row/column:qualifie r`	`GET`	Get the value of a single colum n. Values are Base-64 encode d.
`/table/row/column:qualifie r?e=b64`	`GET`	Get the value of a single colum n using a binary rowkey and co lumn name, encoded in URL-sa fe base64. Returned values are Base-64 encoded.
`/table/row_prefix*/column`	`GET`	Get a combination of rows whic h matches the given row prefix and column family. Returned va lues are Base-64 encoded.
`/table/row_prefix*/column: qualifier`	`GET`	Get a combination of rows whic h matches the given row prefix, column family and qualifier. Ret urned values are Base-64 enco ded.
`/table/multiget?row=row&ro w=row/column:qualifier&row =...`	`GET`	Multi-Get a combination of row s/columns. Values are Base-64 encoded.

Endpoint	HTTP Verb	Description
`/table/multiget?e=b64&row=row&row=row/column:qualifier&row=...`	`GET`	Multi-Get a combination of rows/columns using binary rowkeys and column names, encoded in URL-safe base64. Returned values are Base-64 encoded.
`/table/multiget?row=row&row=row/column:qualifier&filter=url_encoded_filter`	`GET`	Multi-Get a combination of rows/columns with a filter. The filter should be specified according to the Thrift Filter Language and then encoded as `application/x-www-form-urlencoded` MIME format string. This example uses `PrefixFilter('row1')`.
`/table/multiget?row=row&row=row/column:qualifier&row=...&filter_b64=b64_encoded_filter`	`GET`	Multi-Get a combination of rows/columns with a filter. The filter should be specified according to the Thrift Filter Language and then encoded in URL-safe base64. This example uses `PrefixFilter('row1')`.
`/table/row/column:qualifier/?v=number_of_versions`	`GET`	Multi-Get a specified number of versions of a given cell. Values are Base-64 encoded.

Examples:

```
# Get all columns of a single row
curl -vi -X GET \
  -H "Accept: text/xml" \
  "http://example.com:8000/users/row1"

# Get single column with timestamp
curl -vi -X GET \
  -H "Accept: text/xml" \
  "http://example.com:8000/users/row1/cf:a/1458586888395"

# Get single column
curl -vi -X GET \
  -H "Accept: text/xml" \
  "http://example.com:8000/users/row1/cf:a"
```

```
curl -vi -X GET \
  -H "Accept: text/xml" \
  "http://example.com:8000/users/row1/cf:a/"

# Get single column with base64 encoding
curl -vi -X GET \
  -H "Accept: text/xml" \
  "http://example.com:8000/users/cm93MQ/Y2Y6YQ?e=b64"

curl -vi -X GET \
  -H "Accept: text/xml" \
  -H "Encoding: base64" \
  "http://example.com:8000/users/cm93MQ/Y2Y6YQ/"

# Get rows with prefix
curl -vi -X GET \
  -H "Accept: text/xml" \
  "http://example.com:8000/users/row*/cf"

# Multi-get
curl -vi -X GET \
  -H "Accept: text/xml" \
  "http://example.com:8000/users/multiget?row=row1&row=row2/cf:a"

# Multi-get with base64
curl -vi -X GET \
  -H "Accept: text/xml" \
  "http://example.com:8000/users/multiget?e=b64&row=cm93MQ&row=cm93Mg%2FY2Y6YQ"

# Multi-get with filter
curl -vi -X GET \
  -H "Accept: text/xml" \
  "http://example.com:8000/users/multiget?row=row1&row=row2/cf:a&filter=PrefixFil
ter%28%27row1%27%29"

# Multi-get with base64 filter
curl -vi -X GET \
  -H "Accept: text/xml" \
  "http://example.com:8000/users/multiget?row=row1&row=row2/cf:a&filter_b64=UHJlZ
ml4RmlsdGVyKCdyb3cxJyk"

# Get multiple versions
curl -vi -X GET \
  -H "Accept: text/xml" \
  "http://example.com:8000/users/row1/cf:a?v=2"
```

Endpoints for `Delete` Operations

Endpoint	HTTP Verb	Description
`/table/row`	`DELETE`	Delete all columns of a single row.

Endpoint	HTTP Verb	Description
`/table/row/column_family:`	`DELETE`	Delete all columns of a single row and column family.
`/table/row/column:qualifier/timestamp`	`DELETE`	Delete a single column.
`/table/row/column:qualifier`	`DELETE`	Delete a single column.
`/table/row/column:qualifier?e=b64`	`DELETE`	Delete a single column using a binary rowkey and column name, encoded in URL-safe base64.

Examples:

```
# Delete all columns of a row
curl -vi -X DELETE \
  "http://example.com:8000/users/row1"

# Delete all columns of a row and column family
curl -vi -X DELETE \
  "http://example.com:8000/users/row1/cf"

# Delete a single column with timestamp
curl -vi -X DELETE \
  "http://example.com:8000/users/row1/cf:a/1458586888395"

# Delete a single column
curl -vi -X DELETE \
  "http://example.com:8000/users/row1/cf:a"

curl -vi -X DELETE \
  -H "Accept: text/xml" \
  "http://example.com:8000/users/row1/cf:a/"

# Delete with base64 encoding
curl -vi -X DELETE \
  "http://example.com:8000/users/cm93MQ/Y2Y6YQ?e=b64"

curl -vi -X DELETE \
  -H "Encoding: base64" \
  "http://example.com:8000/users/cm93MQ/Y2Y6YQ/"
```

Stateful endpoints for `Scan` Operations

Endpoint	HTTP Verb	Description
`/table/scanner/`	`PUT`	Get a Scanner object. Required by all other Scan operations. Adjust the batch parameter to the number of rows the scan should return in a batch. See the next example for adding filters to your scanner. The scanner endpoint URL is returned as the `Location` in the HTTP response. The other examples in this table assume that the scanner endpoint is `http://example.com:8000/users/scanner/1458690724824375522207`.
`/table/scanner/`	`PUT`	To supply filters to the Scanner object or configure the Scanner in any other way, you can create a text file and add your filter to the file. For example, to return only rows for which keys start with u123 and use a batch size of 100, pass the file to the `-d` argument of the `curl` request (see example below).
`/table/scanner/scanner-id`	`GET`	Get the next batch from the scanner. Cell values are byte-encoded. If the scanner has been exhausted, HTTP status `204` is returned.
`table/scanner/scanner-id`	`DELETE`	Deletes the scanner and frees the resources it used.

For the filter file example, it should contain:

```
<Scanner batch="100">
  <filter>
    {
      "type": "PrefixFilter",
```

```
      "value": "u123"
    }
  </filter>
</Scanner>
```

Examples:

```
# Create a scanner
curl -vi -X PUT \
  -H "Accept: text/xml" \
  -H "Content-Type: text/xml" \
  -d '<Scanner batch="1"/>' \
  "http://example.com:8000/users/scanner/"

# Create a scanner with filter from file
curl -vi -X PUT \
  -H "Accept: text/xml" \
  -H "Content-Type:text/xml" \
  -d @filter.txt \
  "http://example.com:8000/users/scanner/"

# Get next batch from scanner
curl -vi -X GET \
  -H "Accept: text/xml" \
  "http://example.com:8000/users/scanner/145869072824375522207"

# Delete scanner
curl -vi -X DELETE \
  -H "Accept: text/xml" \
  "http://example.com:8000/users/scanner/145869072824375522207"
```

Stateless endpoints for `Scan` Operations

Endpoint	HTTP Verb	Description
`/table/*`	`GET`	Scanning the entire table. The stateless scanner endpoint doe s not require a followup call to r eturn the results.
`/table/*?limit=number_of_r ows`	`GET`	Scanning the first row of the ta ble.
`/table/*?column=column:qua lifier`	`GET`	Scanning a given column of the table.
`/table/*?column=column1:qu alifier1,column2:qualifier 2`	`GET`	Scanning more than one colum n of the table.

Endpoint	HTTP Verb	Description
`/table/*?startrow=row&limit=number_of_rows`	`GET`	Scanning table with start row and limit.
`/table/row_prefix*`	`GET`	Scanning table with row prefix.
`/table/*?reversed=true`	`GET`	Scanning table in reverse.
`/table/*?filter=url_encoded_filter`	`GET`	Scanning with a filter `PrefixFilter('row1')`. The filter should be specified according to the Thrift Filter Language and then encoded as `application/x-www-form-urlencoded` MIME format string.
`/table/*?filter_b64=b64_encoded_filter`	`GET`	Scanning with a filter `PrefixFilter('row1')`. The filter should be specified according to the Thrift Filter Language and then encoded in URL-safe base64.

Examples:

```
# Scan entire table
curl -vi -X GET \
  -H "Accept: text/xml" \
  "http://example.com:8000/users/*"

# Scan with limit
curl -vi -X GET \
  -H "Accept: text/xml" \
  "http://example.com:8000/users/*?limit=1"

# Scan single column
curl -vi -X GET \
  -H "Accept: text/xml" \
  "http://example.com:8000/users/*?column=cf:a"

# Scan multiple columns
curl -vi -X GET \
  -H "Accept: text/xml" \
  "http://example.com:8000/users/*?column=cf:a,cf:b"

# Scan with start row and limit
curl -vi -X GET \
  -H "Accept: text/xml" \
  "http://example.com:8000/users/*?startrow=row1&limit=2"
```

```
# Scan with row prefix
curl -vi -X GET \
  -H "Accept: text/xml" \
  "http://example.com:8000/users/row1*"

# Scan in reverse
curl -vi -X GET \
  -H "Accept: text/xml" \
  "http://example.com:8000/users/*?reversed=true"

# Scan with filter
curl -vi -X GET \
  -H "Accept: text/xml" \
  "http://example.com:8000/users/*?filter=PrefixFilter%28%27row1%27%29"

# Scan with base64 filter
curl -vi -X GET \
  -H "Accept: text/xml" \
  "http://example.com:8000/users/*?filter_b64=UHJlZml4RmlsdGVyKCdyb3cxJyk"
```

The stateful scanner API expects clients to restart scans if there is a REST server failure in the midst. The stateless does not store any state related to scan operation and all the parameters are specified as query parameters.

> The stateless endpoints are optimized for small results, while the stateful scanner API can also be used for large results.

The following are the scan parameters:

- `startrow` - The start row for the scan.
- `endrow` - The end row for the scan.
- `column` - The comma separated list of columns to scan.
- `starttime`, `endtime` - To only retrieve columns within a specific range of version timestamps, both start and end time must be specified.
- `maxversions` - To limit the number of versions of each column to be returned.
- `batchsize` - To limit the maximum number of values returned for each call to next().
- `limit` - The number of rows to return in the scan operation.
- `cacheblocks` - Whether to use the Block Cache in the RegionServer. By default `true`.
- `reversed` - When set to `true`, reverse scan will be executed. By default `false`.

- `filter` - Allows to specify a filter for the scan as an `application/x-www-form-urlencoded` MIME format string.
- `filter_b64` - On versions which include the HBASE-28518 patch, `filter_b64` allows to specify a URL-safe base64 encoded filter for the scan. When both `filter` and `filter_b64` are specified, only `filter_b64` is considered.
- `includeStartRow` - Whether start row should be included in the scan. By default `true`.
- `includeStopRow` - Whether end row (stop row) should be included in the scan. By default `false`.

> `includeStartRow` and `includeStopRow` are only supported on versions that include HBASE-28627.
>
> Versions without this patch will either ignore these parameters or will error out if they are set to a non-default value.

More on start row, end row and limit parameters:

- If start row, end row and limit not specified, then the whole table will be scanned.
- If start row and limit (say N) is specified, then the scan operation will return N rows from the start row specified.
- If only limit parameter is specified, then the scan operation will return N rows from the start of the table.
- If limit and end row are specified, then the scan operation will return N rows from start of table till the end row. If the end row is reached before N rows (say M and M < N), then M rows will be returned to the user.
- If start row, end row and limit (say N) are specified and N < number of rows between start row and end row, then N rows from start row will be returned to the user. If N > (number of rows between start row and end row (say M), then M number of rows will be returned to the user.

Endpoints for `Put` Operations

Endpoint	HTTP Verb	Description
`/table/row_key`	`PUT`	Write a row to a table. The row, column qualifier, and value mus t each be Base-64 encoded. T o encode a string, use the `bas e64` command-line utility. To de code the string, use `base64 - d`. The payload is in the `--dat a` argument, and the `/users/f akerow` value is a placeholder. Insert multiple rows by adding t hem to the `<CellSet>` elemen t. You can also save the data to be inserted to a file and pass it to the `-d` parameter with synt ax like `-d @filename.txt`.

Example:

```
# XML format
curl -vi -X PUT \
  -H "Accept: text/xml" \
  -H "Content-Type: text/xml" \
  -d '<?xml version="1.0" encoding="UTF-8" standalone="yes"?><CellSet><Row key="c
m93NQo="><Cell column="Y2Y6ZQo=">dmFsdWU1Cg==</Cell></Row></CellSet>' \
  "http://example.com:8000/users/fakerow"

# JSON format
curl -vi -X PUT \
  -H "Accept: text/json" \
  -H "Content-Type: text/json" \
  -d '{"Row":[{"key":"cm93NQo=", "Cell": [{"column":"Y2Y6ZQo=", "$":"dmFsdWU1Cg=
="}]}]}'' \
  "example.com:8000/users/fakerow"
```

Endpoints for `Check-And-Put` Operations

Endpoint	HTTP Verb	Description
`/table/row_key/?check=put`	`PUT`	Conditional Put - Change the current version value of a cell: Compare the current or latest version value (`current-version-value`) of a cell with the `check-value`, and if `current-version-value` == `check-value`, write new data (the `new-value`) into the cell as the current or latest version. The row, column qualifier, and value must each be Base-64 encoded. To encode a string, use the `base64` command-line utility. To decode the string, use `base64 -d`. The payload is in the `--data` or `-d` argument, with `the check cell name (column family:column name) and value` always at the end and right after `the new Put cell name (column family:column name) and value` of the same row key. You can also save the data to be inserted to a file and pass it to the `-d` parameter with syntax like `-d @filename.txt`.

Example:

```
# XML format
curl -vi -X PUT \
  -H "Accept: text/xml" \
  -H "Content-Type: text/xml" \
  -d '<?xml version="1.0" encoding="UTF-8" standalone="yes"?><CellSet><Row key="cm93MQ=="><Cell column="Y2ZhOmFsaWFz">T2xkR3V5</Cell><Cell column="Y2ZhOmFsaWFz">TmV3R3V5</Cell></Row></CellSet>' \
  "http://example.com:8000/users/row1/?check=put"

# JSON format
curl -vi -X PUT \
```

```
  -H "Accept: application/json" \
  -H "Content-Type: application/json" \
  -d '{"Row":[{"key":"cm93MQ==","Cell":[{"column":"Y2ZhOmFsaWFz","$":"T2xkR3V5"},
{"column":"Y2ZhOmFsaWFz", "$":"TmV3R3V5"}] }]}' \
  "http://example.com:8000/users/row1/?check=put"
```

Detailed Explanation:

- In the above json-format example:

  1. `{"column":"Y2ZhOmFsaWFz", "$":"TmV3R3V5"}` at the end of `-d` option are `the check cell name and check cell value in Base-64` respectively: `"Y2ZhOmFsaWFz" for "cfa:alias"`, and `"TmV3R3V5" for "NewGuy"`
  2. `{{"column":"Y2ZhOmFsaWFz","$":"T2xkR3V5"}}` are `the new Put cell name and cell value in Base-64` respectively: `"Y2ZhOmFsaWFz" for "cfa:alias"`, and `"T2xkR3V5" for "OldGuy"`
  3. `"cm93MQ=="` is `the Base-64 for "row1"` for the checkAndPut `row key`
  4. `"/?check=put"` after the `"row key" in the request URL` is required for checkAndPut WebHBase operation to work
  5. The `"row key" in the request URL` should be URL-encoded, e.g., `"david%20chen"` and `"row1"` are the URL-encoded formats of row keys `"david chen"` and `"row1"`, respectively

     > Note: "cfa" is the column family name and "alias" are the column (qualifier) name for the non-Base64 encoded cell name.

- Basically, the xml-format example is the same as the json-format example, and will not be explained here in detail.

Endpoints for `Check-And-Delete` Operations

Endpoint	HTTP Verb	Description
`/table/row_key/?check=delete`	`DELETE`	Conditional Deleting a Row: Compare the value of any version of a cell (`any-version-value`) with the `check-value`, and if `any-version-value` == `check-value`, delete the row specified by the `row_key` inside the requesting URL. The row, column qualifier, and value for checking in the payload must each be Base-64 encoded. To encode a string, use the base64 command-line utility. To decode the string, use base64 -d. The payload is in the --data argument. You can also save the data to be checked to a file and pass it to the `-d` parameter with syntax like `-d @filename.txt`.
`/table/row_key/column_family/?check=delete`	`DELETE`	Conditional Deleting a Column Family of a Row: Compare the value of any version of a cell (`any-version-value`) with the `check-value`, and if `any-version-value` == `check-value`, delete the column family of a row specified by the `row_key/column_family` inside the requesting URL. Anything else is the same as those in `Conditional Deleting a Row`.

Endpoint	HTTP Verb	Description
`/table/row_key/column:qualifier/?check=delete`	`DELETE`	Conditional Deleting All Versions of a Column of a Row: Compare the value of any version of a cell (`any-version-value`) with the `check-value`, and if `any-version-value == check-value`, delete the column of a row specified by the `row_key/column:qualifier` inside the requesting URL. The `column:qualifier` in the requesting URL is the `column_family:column_name`. Anything else is the same as those in `Conditional Deleting a Row`.
`/table/row_key/column:qualifier/version_id/?check=delete`	`DELETE`	Conditional Deleting a Single Version of a Column of a Row: Compare the value of any version of a cell (`any-version-value`) with the `check-value`, and if `any-version-value == check-value`, delete the version of a column of a row specified by the `row_key/column:qualifier/version_id` inside the requesting URL. The `column:qualifier` in the requesting URL is the `column_family:column_name`. The `version_id` in the requesting URL is a number, which equals to `the timestamp of the targeted version + 1`. Anything else is the same as those in `Conditional Deleting a Row`.

Examples:

```
# Conditional delete a row (XML)
```

```
curl -vi -X DELETE \
  -H "Accept: text/xml" \
  -H "Content-Type: text/xml" \
  -d '<?xml version="1.0" encoding="UTF-8" standalone="yes"?><CellSet><Row key="c
m93MQ=="><Cell column="Y2ZhOmFsaWFz">TmV3R3V5</Cell></Row></CellSet>' \
  "http://example.com:8000/users/row1/?check=delete"

# Conditional delete a row (JSON)
curl -vi -X DELETE \
  -H "Accept: application/json" \
  -H "Content-Type: application/json" \
  -d '{"Row":[{"key":"cm93MQ==","Cell":[{"column":"Y2ZhOmFsaWFz","$":"TmV3R3V
5"}]}]}' \
  "http://example.com:8000/users/row1/?check=delete"

# Conditional delete a column family (XML)
curl -vi -X DELETE \
  -H "Accept: text/xml" \
  -H "Content-Type: text/xml" \
  -d '<?xml version="1.0" encoding="UTF-8" standalone="yes"?><CellSet><Row key="c
m93MQ=="><Cell column="Y2ZhOmFsaWFz">TmV3R3V5</Cell></Row></CellSet>' \
  "http://example.com:8000/users/row1/cfa/?check=delete"

# Conditional delete a column family (JSON)
curl -vi -X DELETE \
  -H "Accept: application/json" \
  -H "Content-Type: application/json" \
  -d '{"Row":[{"key":"cm93MQ==","Cell":[{"column":"Y2ZhOmFsaWFz","$":"TmV3R3V
5"}]}]}' \
  "http://example.com:8000/users/row1/cfa/?check=delete"

# Conditional delete all versions of a column (XML)
curl -vi -X DELETE \
  -H "Accept: text/xml" \
  -H "Content-Type: text/xml" \
  -d '<?xml version="1.0" encoding="UTF-8" standalone="yes"?><CellSet><Row key="c
m93MQ=="><Cell column="Y2ZhOmFsaWFz">TmV3R3V5</Cell></Row></CellSet>' \
  "http://example.com:8000/users/row1/cfa:alias/?check=delete"

# Conditional delete all versions of a column (JSON)
curl -vi -X DELETE \
  -H "Accept: application/json" \
  -H "Content-Type: application/json" \
  -d '{"Row":[{"key":"cm93MQ==","Cell":[{"column":"Y2ZhOmFsaWFz","$":"TmV3R3V
5"}]}]}' \
  "http://example.com:8000/users/row1/cfa:alias/?check=delete"

# Conditional delete a single version (XML)
curl -vi -X DELETE \
  -H "Accept: text/xml" \
  -H "Content-Type: text/xml" \
  -d '<?xml version="1.0" encoding="UTF-8" standalone="yes"?><CellSet><Row key="c
m93MQ=="><Cell column="Y2ZhOmFsaWFz">TmV3R3V5</Cell></Row></CellSet>' \
  "http://example.com:8000/users/row1/cfa:alias/1519423552160/?check=delete"

# Conditional delete a single version (JSON)
```

```
curl -vi -X DELETE \
  -H "Accept: application/json" \
  -H "Content-Type: application/json" \
  -d '{"Row":[{"key":"cm93MQ==","Cell":[{"column":"Y2ZhOmFsaWFz","$":"TmV3R3V
5"}]}]}' \
  "http://example.com:8000/users/row1/cfa:alias/1519423552160/?check=delete"
```

Detailed Explanation:

- In the above 4 json-format examples:
  1. `{"column":"Y2ZhOmFsaWFz", "$":"TmV3R3V5"}` at the end of `-d` option are `the check cell name and check cell value in Base-64` respectively: `"Y2ZhOmFsaWFz" for "cfa:alias"`, and `"TmV3R3V5" for "NewGuy"`
  2. `"cm93MQ=="` is `the Base-64 for "row1"` for the checkAndDelete `row key`
  3. `"/?check=delete"` at the end of `the request URL` is required for checkAndDelete WebHBase operation to work
  4. `"version_id"` in the `request URL` of the last json-format example should be equivalent to the value of `"the timestamp number + 1"`
  5. The `"row key"`, `"column family"`, `"cell name" or "column family:column name"`, and `"version_id"` in `the request URL` of a checkAndDelete WebHBase operation should be URL-encoded, e.g., `"row1"`, `"cfa"`, `"cfa:alias"` and `"1519423552160"` in the examples are the URL-encoded `"row key"`, `"column family"`, `"column family:column name"`, and `"version_id"`, respectively
- Basically, the 4 xml-format examples are the same as the 4 corresponding json-format examples, and will not be explained here in detail.

Endpoints for `Append` Operations

Endpoint	HTTP Verb	Description
`/table/row_key/?check=append`	`PUT`	Appends the given new value to the end of the current value of the cell. The row, column qualifier, and value must each be Base-64 encoded.

Example:

```
# XML format
curl -vi -X PUT \
  -H "Accept: text/xml" \
  -H "Content-Type: text/xml" \
  -d '<?xml version="1.0" encoding="UTF-8" standalone="yes"?><CellSet><Row key="c
m93NQo="><Cell column="Y2Y6ZQo=">dmFsdWU1Cg==</Cell></Row></CellSet>' \
  "http://example.com:8000/users/row5?check=append"

# JSON format
curl -vi -X PUT \
  -H "Content-type: application/json" \
  -H "Accept: application/json" \
  -d '{"Row":[{"key":"dGVzdHJvdzE=","Cell":[{"column":"YTox","$":"dGVzdHZhbHVlMg
o"},{"column":"YToy","$":"dGVzdHZhbHVlMTIK"}]}]}' \
  "http://localhost:8080/users/testrow1?check=append"
```

Endpoints for `Increment` Operations

Endpoint	HTTP Verb	Description
`/table/row_key/?check=increment`	`PUT`	Increments the current value of the cell. The row, column qualifier, and value must each be Base-64 encoded.

Example:

```
# XML format
curl -vi -X PUT \
  -H "Accept: text/xml" \
  -H "Content-Type: text/xml" \
  -d '<?xml version="1.0" encoding="UTF-8" standalone="yes"?><CellSet><Row key="c
m93NQo="><Cell column="YTox">MQ==</Cell></Row></CellSet>' \
  "http://localhost:8080/users/row5?check=increment"

# JSON format
curl -vi -X PUT \
  -H "Content-type: application/json" \
  -H "Accept: application/json" \
  -d '{"Row":[{"key":"dGVzdHJvdzE=","Cell":[{"column":"YTox","$":"MQ=="},{"colum
n":"YToy","$":"MQ=="}]}]}' \
  "http://localhost:8080/users/testrow1?check=increment"
```

REST XML Schema

```
<schema xmlns="http://www.w3.org/2001/XMLSchema" xmlns:tns="RESTSchema">

  <element name="Version" type="tns:Version"></element>
```

```
  <complexType name="Version">
    <attribute name="REST" type="string"></attribute>
    <attribute name="JVM" type="string"></attribute>
    <attribute name="OS" type="string"></attribute>
    <attribute name="Server" type="string"></attribute>
    <attribute name="Jersey" type="string"></attribute>
    <attribute name="Version" type="string"></attribute>
    <attribute name="Revision" type="string"></attribute>
  </complexType>

  <element name="TableList" type="tns:TableList"></element>

  <complexType name="TableList">
    <sequence>
      <element name="table" type="tns:Table" maxOccurs="unbounded" minOccurs="1">
</element>
    </sequence>
  </complexType>

  <complexType name="Table">
    <sequence>
      <element name="name" type="string"></element>
    </sequence>
  </complexType>

  <element name="TableInfo" type="tns:TableInfo"></element>

  <complexType name="TableInfo">
    <sequence>
      <element name="region" type="tns:TableRegion" maxOccurs="unbounded" minOccu
rs="1"></element>
    </sequence>
    <attribute name="name" type="string"></attribute>
  </complexType>

  <complexType name="TableRegion">
    <attribute name="name" type="string"></attribute>
    <attribute name="id" type="int"></attribute>
    <attribute name="startKey" type="base64Binary"></attribute>
    <attribute name="endKey" type="base64Binary"></attribute>
    <attribute name="location" type="string"></attribute>
  </complexType>

  <element name="TableSchema" type="tns:TableSchema"></element>

  <complexType name="TableSchema">
    <sequence>
      <element name="column" type="tns:ColumnSchema" maxOccurs="unbounded" minOcc
urs="1"></element>
    </sequence>
    <attribute name="name" type="string"></attribute>
    <anyAttribute></anyAttribute>
  </complexType>

  <complexType name="ColumnSchema">
```

```
    <attribute name="name" type="string"></attribute>
    <anyAttribute></anyAttribute>
  </complexType>

  <element name="CellSet" type="tns:CellSet"></element>

  <complexType name="CellSet">
    <sequence>
      <element name="row" type="tns:Row" maxOccurs="unbounded" minOccurs="1"></el
ement>
    </sequence>
  </complexType>

  <element name="Row" type="tns:Row"></element>

  <complexType name="Row">
    <sequence>
      <element name="key" type="base64Binary"></element>
      <element name="cell" type="tns:Cell" maxOccurs="unbounded" minOccurs="1"></
element>
    </sequence>
  </complexType>

  <element name="Cell" type="tns:Cell"></element>

  <complexType name="Cell">
    <sequence>
      <element name="value" maxOccurs="1" minOccurs="1">
        <simpleType><restriction base="base64Binary">
        </simpleType>
      </element>
    </sequence>
    <attribute name="column" type="base64Binary" />
    <attribute name="timestamp" type="int" />
  </complexType>

  <element name="Scanner" type="tns:Scanner"></element>

  <complexType name="Scanner">
    <sequence>
      <element name="column" type="base64Binary" minOccurs="0" maxOccurs="unbound
ed"></element>
    </sequence>
    <attribute name="startRow" type="base64Binary"></attribute>
    <attribute name="endRow" type="base64Binary"></attribute>
    <attribute name="batch" type="int"></attribute>
    <attribute name="startTime" type="int"></attribute>
    <attribute name="endTime" type="int"></attribute>
    <attribute name="filter" type="string"></attribute>
    <attribute name="caching" type="int"></attribute>
    <sequence>
        <element name="labels" type="string" minOccurs="0" maxOccurs="unbounded">
</element>
    </sequence>
    <attribute name="cacheBlocks" type="boolean"></attribute>
    <attribute name="maxVersions" type="int"></attribute>
```

```
  <attribute name="limit" type="int"></attribute>
  <attribute name="includeStartRow" type="boolean"></attribute>
  <attribute name="includeStopRow" type="boolean"></attribute>
</complexType>

<element name="StorageClusterVersion" type="tns:StorageClusterVersion" />

<complexType name="StorageClusterVersion">
  <attribute name="version" type="string"></attribute>
</complexType>

<element name="StorageClusterStatus"
  type="tns:StorageClusterStatus">
</element>

<complexType name="StorageClusterStatus">
  <sequence>
    <element name="liveNode" type="tns:Node"
      maxOccurs="unbounded" minOccurs="0">
    </element>
    <element name="deadNode" type="string" maxOccurs="unbounded"
      minOccurs="0">
    </element>
  </sequence>
  <attribute name="regions" type="int"></attribute>
  <attribute name="requests" type="int"></attribute>
  <attribute name="averageLoad" type="float"></attribute>
</complexType>

<complexType name="Node">
  <sequence>
    <element name="region" type="tns:Region"
 maxOccurs="unbounded" minOccurs="0">
    </element>
  </sequence>
  <attribute name="name" type="string"></attribute>
  <attribute name="startCode" type="int"></attribute>
  <attribute name="requests" type="int"></attribute>
  <attribute name="heapSizeMB" type="int"></attribute>
  <attribute name="maxHeapSizeMB" type="int"></attribute>
</complexType>

<complexType name="Region">
  <attribute name="name" type="base64Binary"></attribute>
  <attribute name="stores" type="int"></attribute>
  <attribute name="storefiles" type="int"></attribute>
  <attribute name="storefileSizeMB" type="int"></attribute>
  <attribute name="memstoreSizeMB" type="int"></attribute>
  <attribute name="storefileIndexSizeKB" type="int"></attribute>
  <attribute name="readRequestsCount" type="int"></attribute>
  <attribute name="cpRequestsCount" type="int"></attribute>
  <attribute name="writeRequestsCount" type="int"></attribute>
  <attribute name="rootIndexSizeKB" type="int"></attribute>
  <attribute name="totalStaticIndexSizeKB" type="int"></attribute>
  <attribute name="totalStaticBloomSizeKB" type="int"></attribute>
  <attribute name="totalCompactingKVs" type="int"></attribute>
```

```
    <attribute name="currentCompactedKVs" type="int"></attribute>
  </complexType>

</schema>
```

REST Protobufs Schema

```
message Version {
  optional string restVersion = 1;
  optional string jvmVersion = 2;
  optional string osVersion = 3;
  optional string serverVersion = 4;
  optional string jerseyVersion = 5;
  optional string version = 6;
  optional string revision = 7;
}

message StorageClusterStatus {
  message Region {
    required bytes name = 1;
    optional int32 stores = 2;
    optional int32 storefiles = 3;
    optional int32 storefileSizeMB = 4;
    optional int32 memStoreSizeMB = 5;
    optional int64 storefileIndexSizeKB = 6;
    optional int64 readRequestsCount = 7;
    optional int64 writeRequestsCount = 8;
    optional int32 rootIndexSizeKB = 9;
    optional int32 totalStaticIndexSizeKB = 10;
    optional int32 totalStaticBloomSizeKB = 11;
    optional int64 totalCompactingKVs = 12;
    optional int64 currentCompactedKVs = 13;
    optional int64 cpRequestsCount = 14;
  }
  message Node {
    required string name = 1;    // name:port
    optional int64 startCode = 2;
    optional int32 requests = 3;
    optional int32 heapSizeMB = 4;
    optional int32 maxHeapSizeMB = 5;
    repeated Region regions = 6;
  }
  // node status
  repeated Node liveNodes = 1;
  repeated string deadNodes = 2;
  // summary statistics
  optional int32 regions = 3;
  optional int32 requests = 4;
  optional double averageLoad = 5;
}

message TableList {
  repeated string name = 1;
```

```
}

message TableInfo {
  required string name = 1;
  message Region {
    required string name = 1;
    optional bytes startKey = 2;
    optional bytes endKey = 3;
    optional int64 id = 4;
    optional string location = 5;
  }
  repeated Region regions = 2;
}

message TableSchema {
  optional string name = 1;
  message Attribute {
    required string name = 1;
    required string value = 2;
  }
  repeated Attribute attrs = 2;
  repeated ColumnSchema columns = 3;
  // optional helpful encodings of commonly used attributes
  optional bool inMemory = 4;
  optional bool readOnly = 5;
}

message ColumnSchema {
  optional string name = 1;
  message Attribute {
    required string name = 1;
    required string value = 2;
  }
  repeated Attribute attrs = 2;
  // optional helpful encodings of commonly used attributes
  optional int32 ttl = 3;
  optional int32 maxVersions = 4;
  optional string compression = 5;
}

message Cell {
  optional bytes row = 1;       // unused if Cell is in a CellSet
  optional bytes column = 2;
  optional int64 timestamp = 3;
  optional bytes data = 4;
}

message CellSet {
  message Row {
    required bytes key = 1;
    repeated Cell values = 2;
  }
  repeated Row rows = 1;
}

message Scanner {
```

```
  optional bytes startRow = 1;
  optional bytes endRow = 2;
  repeated bytes columns = 3;
  optional int32 batch = 4;
  optional int64 startTime = 5;
  optional int64 endTime = 6;
  optional int32 maxVersions = 7;
  optional string filter = 8;
  optional int32 caching = 9;      // specifies REST scanner caching
  repeated string labels = 10;
  optional bool cacheBlocks = 11; // server side block caching hint
  optional int32 limit = 12;
  optional bool includeStartRow = 13;
  optional bool includeStopRow = 14;
}
```

Thrift

Documentation about Thrift has moved to Thrift API and Filter Language.

C/C++ Apache HBase Client

FB's Chip Turner wrote a pure C/C++ client. Check it out.

C++ client implementation. To see HBASE-14850.

Using Java Data Objects (JDO) with HBase

Java Data Objects (JDO) is a standard way to access persistent data in databases, using plain old Java objects (POJO) to represent persistent data.

Dependencies

This code example has the following dependencies:

1. HBase 0.90.x or newer
2. commons-beanutils.jar (https://commons.apache.org/)
3. commons-pool-1.5.5.jar (https://commons.apache.org/)
4. transactional-tableindexed for HBase 0.90 (https://github.com/hbase-trx/hbase-transactional-tableindexed)

Download `hbase-jdo`

Download the code from http://code.google.com/p/hbase-jdo/.

JDO Example

This example uses JDO to create a table and an index, insert a row into a table, get a row, get a column value, perform a query, and do some additional HBase operations.

```
package com.apache.hadoop.hbase.client.jdo.examples;

import java.io.File;
import java.io.FileInputStream;
import java.io.InputStream;
import java.util.Hashtable;

import org.apache.hadoop.fs.Path;
import org.apache.hadoop.hbase.client.tableindexed.IndexedTable;

import com.apache.hadoop.hbase.client.jdo.AbstractHBaseDBO;
import com.apache.hadoop.hbase.client.jdo.HBaseBigFile;
import com.apache.hadoop.hbase.client.jdo.HBaseDBOImpl;
import com.apache.hadoop.hbase.client.jdo.query.DeleteQuery;
import com.apache.hadoop.hbase.client.jdo.query.HBaseOrder;
import com.apache.hadoop.hbase.client.jdo.query.HBaseParam;
import com.apache.hadoop.hbase.client.jdo.query.InsertQuery;
import com.apache.hadoop.hbase.client.jdo.query.QSearch;
import com.apache.hadoop.hbase.client.jdo.query.SelectQuery;
import com.apache.hadoop.hbase.client.jdo.query.UpdateQuery;

/**
 * Hbase JDO Example.
 *
 * dependency library.
 * - commons-beanutils.jar
 * - commons-pool-1.5.5.jar
 * - hbase0.90.0-transactionl.jar
 *
 * you can expand Delete,Select,Update,Insert Query classes.
 *
 */
public class HBaseExample {
  public static void main(String[] args) throws Exception {
    AbstractHBaseDBO dbo = new HBaseDBOImpl();

    //*drop if table is already exist.*
    if(dbo.isTableExist("user")){
     dbo.deleteTable("user");
    }

    //*create table*
    dbo.createTableIfNotExist("user",HBaseOrder.DESC,"account");
    //dbo.createTableIfNotExist("user",HBaseOrder.ASC,"account");
```

```
    //create index.
    String[] cols={"id","name"};
    dbo.addIndexExistingTable("user","account",cols);

    //insert
    InsertQuery insert = dbo.createInsertQuery("user");
    UserBean bean = new UserBean();
    bean.setFamily("account");
    bean.setAge(20);
    bean.setEmail("ncanis@gmail.com");
    bean.setId("ncanis");
    bean.setName("ncanis");
    bean.setPassword("1111");
    insert.insert(bean);

    //select 1 row
    SelectQuery select = dbo.createSelectQuery("user");
    UserBean resultBean = (UserBean)select.select(bean.getRow(),UserBean.class);

    // select column value.
    String value = (String)select.selectColumn(bean.getRow(),"account","id",Strin
g.class);

    // search with option (QSearch has EQUAL, NOT_EQUAL, LIKE)
    // select id,password,name,email from account where id='ncanis' limit startRo
w,20
    HBaseParam param = new HBaseParam();
    param.setPage(bean.getRow(),20);
    param.addColumn("id","password","name","email");
    param.addSearchOption("id","ncanis",QSearch.EQUAL);
    select.search("account", param, UserBean.class);

    // search column value is existing.
    boolean isExist = select.existColumnValue("account","id","ncanis".getBytes
());

    // update password.
    UpdateQuery update = dbo.createUpdateQuery("user");
    Hashtable<String, byte[]> colsTable = new Hashtable<String, byte[]>();
    colsTable.put("password","2222".getBytes());
    update.update(bean.getRow(),"account",colsTable);

    //delete
    DeleteQuery delete = dbo.createDeleteQuery("user");
    delete.deleteRow(resultBean.getRow());

    ////////////////////////////////
    // etc

    // HTable pool with apache commons pool
    // borrow and release. HBasePoolManager(maxActive, minIdle etc..)
    IndexedTable table = dbo.getPool().borrow("user");
    dbo.getPool().release(table);

    // upload bigFile by hadoop directly.
    HBaseBigFile bigFile = new HBaseBigFile();
```

```
    File file = new File("doc/movie.avi");
    FileInputStream fis = new FileInputStream(file);
    Path rootPath = new Path("/files/");
    String filename = "movie.avi";
    bigFile.uploadFile(rootPath,filename,fis,true);

    // receive file stream from hadoop.
    Path p = new Path(rootPath,filename);
    InputStream is = bigFile.path2Stream(p,4096);

  }
}
```

Scala

Setting the Classpath

To use Scala with HBase, your CLASSPATH must include HBase's classpath as well as the Scala JARs required by your code. First, use the following command on a server running the HBase RegionServer process, to get HBase's classpath.

```
$ ps aux |grep regionserver| awk -F 'java.library.path=' {'print $2'} | awk {'pri
nt $1'}

/usr/lib/hadoop/lib/native:/usr/lib/hbase/lib/native/Linux-amd64-64
```

Set the `$CLASSPATH` environment variable to include the path you found in the previous step, plus the path of `scala-library.jar` and each additional Scala-related JAR needed for your project.

```
$ export CLASSPATH=$CLASSPATH:/usr/lib/hadoop/lib/native:/usr/lib/hbase/lib/nativ
e/Linux-amd64-64:/path/to/scala-library.jar
```

Scala SBT File

Your `build.sbt` file needs the following `resolvers` and `libraryDependencies` to work with HBase.

```
resolvers += "Apache HBase" at "https://repository.apache.org/content/repositorie
s/releases"

resolvers += "Thrift" at "https://people.apache.org/~rawson/repo/"
```

```
libraryDependencies ++= Seq(
    "org.apache.hadoop" % "hadoop-core" % "0.20.2",
    "org.apache.hbase" % "hbase" % "0.90.4"
)
```

Example Scala Code

This example lists HBase tables, creates a new table, adds a row to it, and gets the value of the row.

```
import org.apache.hadoop.hbase.{HBaseConfiguration, TableName}
import org.apache.hadoop.hbase.client.{Admin, Connection, ConnectionFactory, Get,
Put}
import org.apache.hadoop.hbase.util.Bytes

val conf = HBaseConfiguration.create()
val connection = ConnectionFactory.createConnection(conf);
val admin = connection.getAdmin();

// list the tables
val listtables = admin.listTables()
listtables.foreach(println)

// let's insert some data in 'mytable' and get the row
val table = connection.getTable(TableName.valueOf("mytable"))

val theput = new Put(Bytes.toBytes("rowkey1"))

theput.addColumn(Bytes.toBytes("ids"),Bytes.toBytes("id1"),Bytes.toBytes("one"))
table.put(theput)

val theget = new Get(Bytes.toBytes("rowkey1"))
val result = table.get(theget)
val value = result.value()
println(Bytes.toString(value))
```

Jython

Setting the Classpath

To use Jython with HBase, your CLASSPATH must include HBase's classpath as well as the Jython JARs required by your code.

Set the path to directory containing the `jython.jar` and each additional Jython-related JAR needed for your project. Then export HBASE_CLASSPATH pointing to the

$JYTHON_HOME env. variable.

```
$ export HBASE_CLASSPATH=/directory/jython.jar
```

Start a Jython shell with HBase and Hadoop JARs in the classpath: $ bin/hbase org.python.util.jython

Jython Code Examples

Example: Table Creation, Population, Get, and Delete with Jython

The following Jython code example checks for table, if it exists, deletes it and then creates it. Then it populates the table with data and fetches the data.

```
import java.lang
from org.apache.hadoop.hbase import HBaseConfiguration, HTableDescriptor, HColumn
Descriptor, TableName
from org.apache.hadoop.hbase.client import Admin, Connection, ConnectionFactory,
Get, Put, Result, Table
from org.apache.hadoop.conf import Configuration

# First get a conf object.  This will read in the configuration
# that is out in your hbase-*.xml files such as location of the
# hbase master node.
conf = HBaseConfiguration.create()
connection = ConnectionFactory.createConnection(conf)
admin = connection.getAdmin()

# Create a table named 'test' that has a column family
# named 'content'.
tableName = TableName.valueOf("test")
table = connection.getTable(tableName)

desc = HTableDescriptor(tableName)
desc.addFamily(HColumnDescriptor("content"))

# Drop and recreate if it exists
if admin.tableExists(tableName):
    admin.disableTable(tableName)
    admin.deleteTable(tableName)

admin.createTable(desc)

# Add content to 'column:' on a row named 'row_x'
row = 'row_x'
put = Put(row)
put.addColumn("content", "qual", "some content")
table.put(put)

# Now fetch the content just added, returns a byte[]
```

```
get = Get(row)

result = table.get(get)
data = java.lang.String(result.getValue("content", "qual"), "UTF8")

print "The fetched row contains the value '%s'" % data
```

Example: Table Scan Using Jython

This example scans a table and returns the results that match a given family qualifier.

```
import java.lang
from org.apache.hadoop.hbase import TableName, HBaseConfiguration
from org.apache.hadoop.hbase.client import Connection, ConnectionFactory, Result,
ResultScanner, Table, Admin
from org.apache.hadoop.conf import Configuration
conf = HBaseConfiguration.create()
connection = ConnectionFactory.createConnection(conf)
admin = connection.getAdmin()
tableName = TableName.valueOf('wiki')
table = connection.getTable(tableName)

cf = "title"
attr = "attr"
scanner = table.getScanner(cf)
while 1:
    result = scanner.next()
    if not result:
       break
    print java.lang.String(result.row), java.lang.String(result.getValue(cf, att
r))
```

Thrift API and Filter Language

You can configure Thrift for secure authentication at the server and client side, by following the procedures in Client-side Configuration for Secure Operation - Thrift Gateway and Configure the Thrift Gateway to Authenticate on Behalf of the Client.

The rest of this chapter discusses the filter language provided by the Thrift API.

Filter Language

Thrift Filter Language was introduced in HBase 0.92. It allows you to perform server-side filtering when accessing HBase over Thrift or in the HBase shell. You can find out more about shell integration by using the `scan help` command in the shell.

You specify a filter as a string, which is parsed on the server to construct the filter.

General Filter String Syntax

A simple filter expression is expressed as a string:

```
"FilterName (argument, argument,... , argument)"
```

Keep the following syntax guidelines in mind.

- Specify the name of the filter followed by the comma-separated argument list in parentheses.
- If the argument represents a string, it should be enclosed in single quotes (`'`).
- Arguments which represent a boolean, an integer, or a comparison operator (such as `<`, `>`, or `!=`), should not be enclosed in quotes
- The filter name must be a single word. All ASCII characters are allowed except for whitespace, single quotes and parentheses.
- The filter's arguments can contain any ASCII character. If single quotes are present in the argument, they must be escaped by an additional preceding single quote.

Compound Filters and Operators

Binary Operators

`AND`

If the `AND` operator is used, the key-value must satisfy both filters.

`OR`

If the `OR` operator is used, the key-value must satisfy at least one of the filters.

Unary Operators

`SKIP`

For a particular row, if any of the key-values fail the filter condition, the entire row is skipped.

`WHILE`

For a particular row, key-values will be emitted until a key-value is reached that fails the filter condition.

Compound Operators

You can combine multiple operators to create a hierarchy of filters, such as the following example:

```
(Filter1 AND Filter2) OR (Filter3 AND Filter4)
```

Order of Evaluation

1. Parentheses have the highest precedence.
2. The unary operators `SKIP` and `WHILE` are next, and have the same precedence.
3. The binary operators follow. `AND` has highest precedence, followed by `OR`.

Precedence Example

```
Filter1 AND Filter2 OR Filter
is evaluated as
(Filter1 AND Filter2) OR Filter3
```

```
Filter1 AND SKIP Filter2 OR Filter3
```

```
is evaluated as
(Filter1 AND (SKIP Filter2)) OR Filter3
```

You can use parentheses to explicitly control the order of evaluation.

Compare Operator

The following compare operators are provided:

1. LESS (`<`)
2. LESS_OR_EQUAL (`<=`)
3. EQUAL (`=`)
4. NOT_EQUAL (`!=`)
5. GREATER_OR_EQUAL (`>=`)
6. GREATER (`>`)
7. NO_OP (no operation)

The client should use the symbols (`<`, `<=`, `=`, `!=`, `>`, `>=`) to express compare operators.

Comparator

A comparator can be any of the following:

1. *BinaryComparator* - This lexicographically compares against the specified byte array using Bytes.compareTo(byte[], byte[])
2. *BinaryPrefixComparator* - This lexicographically compares against a specified byte array. It only compares up to the length of this byte array.
3. *RegexStringComparator* - This compares against the specified byte array using the given regular expression. Only EQUAL and NOT_EQUAL comparisons are valid with this comparator
4. *SubStringComparator* - This tests if the given substring appears in a specified byte array. The comparison is case insensitive. Only EQUAL and NOT_EQUAL comparisons are valid with this comparator

The general syntax of a comparator is: `ComparatorType:ComparatorValue`

The ComparatorType for the various comparators is as follows:

1. *BinaryComparator* - binary
2. *BinaryPrefixComparator* - binaryprefix
3. *RegexStringComparator* - regexstring
4. *SubStringComparator* - substring

The ComparatorValue can be any value.

Example ComparatorValues

1. `binary:abc` will match everything that is lexicographically greater than "abc"
2. `binaryprefix:abc` will match everything whose first 3 characters are lexicographically equal to "abc"
3. `regexstring:ab*yz` will match everything that doesn't begin with "ab" and ends with "yz"
4. `substring:abc123` will match everything that begins with the substring "abc123"

Example PHP Client Program that uses the Filter Language

```
<?
  $_SERVER['PHP_ROOT'] = realpath(dirname(__FILE__).'/..');
  require_once $_SERVER['PHP_ROOT'].'/flib/__flib.php';
  flib_init(FLIB_CONTEXT_SCRIPT);
  require_module('storage/hbase');
  $hbase = new HBase('<server_name_running_thrift_server>', <port on which thrift
server is running>);
  $hbase->open();
  $client = $hbase->getClient();
  $result = $client->scannerOpenWithFilterString('table_name', "(PrefixFilter ('r
ow2') AND (QualifierFilter (>=, 'binary:xyz'))) AND (TimestampsFilter ( 123, 45
6))");
  $to_print = $client->scannerGetList($result,1);
  while ($to_print) {
    print_r($to_print);
    $to_print = $client->scannerGetList($result,1);
  }
  $client->scannerClose($result);
?>
```

Example Filter Strings

- `"PrefixFilter ('Row') AND PageFilter (1) AND FirstKeyOnlyFilter ()"` will return all key-value pairs that match the following conditions:

  1. The row containing the key-value should have prefix *Row*
  2. The key-value must be located in the first row of the table
  3. The key-value pair must be the first key-value in the row

- `"(RowFilter (=, 'binary:Row 1') AND TimeStampsFilter (74689, 89734)) OR ColumnRangeFilter ('abc', true, 'xyz', false))"` will return all key-value pairs that match both the following conditions:
  - The key-value is in a row having row key *Row 1*
  - The key-value must have a timestamp of either 74689 or 89734.
  - Or it must match the following condition:
    - The key-value pair must be in a column that is lexicographically >= abc and < xyz
- `"SKIP ValueFilter (0)"` will skip the entire row if any of the values in the row is not 0

Individual Filter Syntax

KeyOnlyFilter

This filter doesn't take any arguments. It returns only the key component of each key-value.

FirstKeyOnlyFilter

This filter doesn't take any arguments. It returns only the first key-value from each row.

PrefixFilter

This filter takes one argument – a prefix of a row key. It returns only those key-values present in a row that starts with the specified row prefix

ColumnPrefixFilter

This filter takes one argument – a column prefix. It returns only those key-values present in a column that starts with the specified column prefix. The column prefix must be of the form: `"qualifier"`.

MultipleColumnPrefixFilter

This filter takes a list of column prefixes. It returns key-values that are present in a column that starts with any of the specified column prefixes. Each of the column prefixes must be of the form: `"qualifier"`.

ColumnCountGetFilter

This filter takes one argument – a limit. It returns the first limit number of columns in the table.

PageFilter

This filter takes one argument – a page size. It returns page size number of rows from the table.

ColumnPaginationFilter

This filter takes two arguments – a limit and offset. It returns limit number of columns after offset number of columns. It does this for all the rows.

InclusiveStopFilter

This filter takes one argument – a row key on which to stop scanning. It returns all key-values present in rows up to and including the specified row.

TimeStampsFilter

This filter takes a list of timestamps. It returns those key-values whose timestamps matches any of the specified timestamps.

RowFilter

This filter takes a compare operator and a comparator. It compares each row key with the comparator using the compare operator and if the comparison returns true, it returns all the key-values in that row.

Family Filter

This filter takes a compare operator and a comparator. It compares each column family name with the comparator using the compare operator and if the comparison returns true, it returns all the Cells in that column family.

QualifierFilter

This filter takes a compare operator and a comparator. It compares each qualifier name with the comparator using the compare operator and if the comparison returns true, it returns all the key-values in that column.

ValueFilter

This filter takes a compare operator and a comparator. It compares each value with the comparator using the compare operator and if the comparison returns true, it returns that key-value.

DependentColumnFilter

This filter takes two arguments – a family and a qualifier. It tries to locate this column in each row and returns all key-values in that row that have the same timestamp. If the row doesn't contain the specified column – none of the key-values in that row will be returned.

SingleColumnValueFilter

This filter takes a column family, a qualifier, a compare operator and a comparator. If the specified column is not found – all the columns of that row will be emitted. If the column is found and the comparison with the comparator returns true, all the columns of the row will be emitted. If the condition fails, the row will not be emitted.

SingleColumnValueExcludeFilter

This filter takes the same arguments and behaves same as SingleColumnValueFilter – however, if the column is found and the condition passes, all the columns of the row will be emitted except for the tested column value.

ColumnRangeFilter

This filter is used for selecting only those keys with columns that are between minColumn and maxColumn. It also takes two boolean variables to indicate whether to include the minColumn and maxColumn or not.

HBase and Spark

Spark itself is out of scope of this document, please refer to the Spark site for more information on the Spark project and subprojects. This document will focus on 4 main interaction points between Spark and HBase. Those interaction points are:

Basic Spark
The ability to have an HBase Connection at any point in your Spark DAG.

Spark Streaming
The ability to have an HBase Connection at any point in your Spark Streaming application.

Spark Bulk Load
The ability to write directly to HBase HFiles for bulk insertion into HBase

SparkSQL/DataFrames
The ability to write SparkSQL that draws on tables that are represented in HBase.

The following sections will walk through examples of all these interaction points.

Basic Spark

This section discusses Spark HBase integration at the lowest and simplest levels. All the other interaction points are built upon the concepts that will be described here.

At the root of all Spark and HBase integration is the HBaseContext. The HBaseContext takes in HBase configurations and pushes them to the Spark executors. This allows us to have an HBase Connection per Spark Executor in a static location.

For reference, Spark Executors can be on the same nodes as the Region Servers or on different nodes, there is no dependence on co-location. Think of every Spark Executor as a multi-threaded client application. This allows any Spark Tasks running on the executors to access the shared Connection object.

HBaseContext Usage Example

This example shows how HBaseContext can be used to do a `foreachPartition` on a RDD in Scala:

```
val sc = new SparkContext("local", "test")
val config = new HBaseConfiguration()

...

val hbaseContext = new HBaseContext(sc, config)

rdd.hbaseForeachPartition(hbaseContext, (it, conn) => {
 val bufferedMutator = conn.getBufferedMutator(TableName.valueOf("t1"))
 it.foreach((putRecord) => {
. val put = new Put(putRecord._1)
. putRecord._2.foreach((putValue) => put.addColumn(putValue._1, putValue._2, putV
alue._3))
. bufferedMutator.mutate(put)
 })
 bufferedMutator.flush()
 bufferedMutator.close()
})
```

Here is the same example implemented in Java:

```
JavaSparkContext jsc = new JavaSparkContext(sparkConf);

try {
  List<byte[]> list = new ArrayList<>();
  list.add(Bytes.toBytes("1"));
  ...
  list.add(Bytes.toBytes("5"));

  JavaRDD<byte[]> rdd = jsc.parallelize(list);
  Configuration conf = HBaseConfiguration.create();

  JavaHBaseContext hbaseContext = new JavaHBaseContext(jsc, conf);

  hbaseContext.foreachPartition(rdd,
      new VoidFunction<Tuple2<Iterator<byte[]>, Connection>>() {
   public void call(Tuple2<Iterator<byte[]>, Connection> t)
        throws Exception {
    Table table = t._2().getTable(TableName.valueOf(tableName));
    BufferedMutator mutator = t._2().getBufferedMutator(TableName.valueOf(tableNa
me));
    while (t._1().hasNext()) {
      byte[] b = t._1().next();
      Result r = table.get(new Get(b));
      if (r.getExists()) {
       mutator.mutate(new Put(b));
      }
    }

    mutator.flush();
    mutator.close();
    table.close();
   }
```

```
  });
} finally {
  jsc.stop();
}
```

All functionality between Spark and HBase will be supported both in Scala and in Java, with the exception of SparkSQL which will support any language that is supported by Spark. For the remaining of this documentation we will focus on Scala examples.

The examples above illustrate how to do a foreachPartition with a connection. A number of other Spark base functions are supported out of the box:

`bulkPut`

For massively parallel sending of puts to HBase

`bulkDelete`

For massively parallel sending of deletes to HBase

`bulkGet`

For massively parallel sending of gets to HBase to create a new RDD

`mapPartition`

To do a Spark Map function with a Connection object to allow full access to HBase

`hbaseRDD`

To simplify a distributed scan to create a RDD

For examples of all these functionalities, see the hbase-spark integration in the hbase-connectors repository (the hbase-spark connectors live outside hbase core in a related, Apache HBase project maintained, associated repo).

Spark Streaming

Spark Streaming is a micro batching stream processing framework built on top of Spark. HBase and Spark Streaming make great companions in that HBase can help serve the following benefits alongside Spark Streaming.

- A place to grab reference data or profile data on the fly

- A place to store counts or aggregates in a way that supports Spark Streaming's promise of *only once processing*.

The hbase-spark integration with Spark Streaming is similar to its normal Spark integration points, in that the following commands are possible straight off a Spark Streaming DStream.

`bulkPut`

For massively parallel sending of puts to HBase

`bulkDelete`

For massively parallel sending of deletes to HBase

`bulkGet`

For massively parallel sending of gets to HBase to create a new RDD

`mapPartition`

To do a Spark Map function with a Connection object to allow full access to HBase

`hbaseRDD`

To simplify a distributed scan to create a RDD

`bulkPut` Example with DStreams

Below is an example of bulkPut with DStreams. It is very close in feel to the RDD bulk put.

```
val sc = new SparkContext("local", "test")
val config = new HBaseConfiguration()

val hbaseContext = new HBaseContext(sc, config)
val ssc = new StreamingContext(sc, Milliseconds(200))

val rdd1 = ...
val rdd2 = ...

val queue = mutable.Queue[RDD[(Array[Byte], Array[(Array[Byte],
    Array[Byte], Array[Byte])])]]()

queue += rdd1
queue += rdd2

val dStream = ssc.queueStream(queue)

dStream.hbaseBulkPut(
  hbaseContext,
```

```
  TableName.valueOf(tableName),
  (putRecord) => {
   val put = new Put(putRecord._1)
   putRecord._2.foreach((putValue) => put.addColumn(putValue._1, putValue._2, put
Value._3))
   put
  })
```

There are three inputs to the `hbaseBulkPut` function. The hbaseContext that carries the configuration broadcast information link to the HBase Connections in the executor, the table name of the table we are putting data into, and a function that will convert a record in the DStream into an HBase Put object.

Bulk Load

There are two options for bulk loading data into HBase with Spark. There is the basic bulk load functionality that will work for cases where your rows have millions of columns and cases where your columns are not consolidated and partitioned before the map side of the Spark bulk load process.

There is also a thin record bulk load option with Spark. This second option is designed for tables that have less then 10k columns per row. The advantage of this second option is higher throughput and less over-all load on the Spark shuffle operation.

Both implementations work more or less like the MapReduce bulk load process in that a partitioner partitions the rowkeys based on region splits and the row keys are sent to the reducers in order, so that HFiles can be written out directly from the reduce phase.

In Spark terms, the bulk load will be implemented around a Spark `repartitionAndSortWithinPartitions` followed by a Spark `foreachPartition`.

First lets look at an example of using the basic bulk load functionality

Bulk Loading Example

The following example shows bulk loading with Spark.

```
val sc = new SparkContext("local", "test")
val config = new HBaseConfiguration()

val hbaseContext = new HBaseContext(sc, config)
```

```
val stagingFolder = ...
val rdd = sc.parallelize(Array(
      (Bytes.toBytes("1"),
        (Bytes.toBytes(columnFamily1), Bytes.toBytes("a"), Bytes.toBytes("foo
1"))),
      (Bytes.toBytes("3"),
        (Bytes.toBytes(columnFamily1), Bytes.toBytes("b"), Bytes.toBytes("foo2.
b"))), ...

rdd.hbaseBulkLoad(TableName.valueOf(tableName),
  t => {
   val rowKey = t._1
   val family:Array[Byte] = t._2(0)._1
   val qualifier = t._2(0)._2
   val value = t._2(0)._3

   val keyFamilyQualifier= new KeyFamilyQualifier(rowKey, family, qualifier)

   Seq((keyFamilyQualifier, value)).iterator
  },
  stagingFolder.getPath)

val load = new LoadIncrementalHFiles(config)
load.doBulkLoad(new Path(stagingFolder.getPath),
  conn.getAdmin, table, conn.getRegionLocator(TableName.valueOf(tableName)))
```

The `hbaseBulkLoad` function takes three required parameters:

1. The table name of the table we intend to bulk load too
2. A function that will convert a record in the RDD to a tuple key value par. With the tuple key being a KeyFamilyQualifer object and the value being the cell value. The KeyFamilyQualifer object will hold the RowKey, Column Family, and Column Qualifier. The shuffle will partition on the RowKey but will sort by all three values.
3. The temporary path for the HFile to be written out too

Following the Spark bulk load command, use the HBase's LoadIncrementalHFiles object to load the newly created HFiles into HBase.

Additional Parameters for Bulk Loading with Spark

You can set the following attributes with additional parameter options on hbaseBulkLoad.

- Max file size of the HFiles
- A flag to exclude HFiles from compactions

- Column Family settings for compression, bloomType, blockSize, and dataBlockEncoding

Using Additional Parameters

```
val sc = new SparkContext("local", "test")
val config = new HBaseConfiguration()

val hbaseContext = new HBaseContext(sc, config)

val stagingFolder = ...
val rdd = sc.parallelize(Array(
      (Bytes.toBytes("1"),
        (Bytes.toBytes(columnFamily1), Bytes.toBytes("a"), Bytes.toBytes("foo
1"))),
      (Bytes.toBytes("3"),
        (Bytes.toBytes(columnFamily1), Bytes.toBytes("b"), Bytes.toBytes("foo2.
b"))), ...

val familyHBaseWriterOptions = new java.util.HashMap[Array[Byte], FamilyHFileWrit
eOptions]
val f1Options = new FamilyHFileWriteOptions("GZ", "ROW", 128, "PREFIX")

familyHBaseWriterOptions.put(Bytes.toBytes("columnFamily1"), f1Options)

rdd.hbaseBulkLoad(TableName.valueOf(tableName),
  t => {
   val rowKey = t._1
   val family:Array[Byte] = t._2(0)._1
   val qualifier = t._2(0)._2
   val value = t._2(0)._3

   val keyFamilyQualifier= new KeyFamilyQualifier(rowKey, family, qualifier)

   Seq((keyFamilyQualifier, value)).iterator
  },
  stagingFolder.getPath,
  familyHBaseWriterOptions,
  compactionExclude = false,
  HConstants.DEFAULT_MAX_FILE_SIZE)

val load = new LoadIncrementalHFiles(config)
load.doBulkLoad(new Path(stagingFolder.getPath),
  conn.getAdmin, table, conn.getRegionLocator(TableName.valueOf(tableName)))
```

Now lets look at how you would call the thin record bulk load implementation

Using thin record bulk load

```
val sc = new SparkContext("local", "test")
val config = new HBaseConfiguration()
```

```
val hbaseContext = new HBaseContext(sc, config)

val stagingFolder = ...
val rdd = sc.parallelize(Array(
      ("1",
        (Bytes.toBytes(columnFamily1), Bytes.toBytes("a"), Bytes.toBytes("foo
1"))),
      ("3",
        (Bytes.toBytes(columnFamily1), Bytes.toBytes("b"), Bytes.toBytes("foo2.
b"))), ...

rdd.hbaseBulkLoadThinRows(hbaseContext,
      TableName.valueOf(tableName),
      t => {
        val rowKey = t._1

        val familyQualifiersValues = new FamiliesQualifiersValues
        t._2.foreach(f => {
          val family:Array[Byte] = f._1
          val qualifier = f._2
          val value:Array[Byte] = f._3

          familyQualifiersValues +=(family, qualifier, value)
        })
        (new ByteArrayWrapper(Bytes.toBytes(rowKey)), familyQualifiersValues)
      },
      stagingFolder.getPath,
      new java.util.HashMap[Array[Byte], FamilyHFileWriteOptions],
      compactionExclude = false,
      20)

val load = new LoadIncrementalHFiles(config)
load.doBulkLoad(new Path(stagingFolder.getPath),
  conn.getAdmin, table, conn.getRegionLocator(TableName.valueOf(tableName)))
```

Note that the big difference in using bulk load for thin rows is the function returns a tuple with the first value being the row key and the second value being an object of FamiliesQualifiersValues, which will contain all the values for this row for all column families.

SparkSQL/DataFrames

The hbase-spark integration leverages DataSource API (SPARK-3247) introduced in Spark-1.2.0, which bridges the gap between simple HBase KV store and complex relational SQL queries and enables users to perform complex data analytical work on top of HBase using Spark. HBase Dataframe is a standard Spark Dataframe, and is able to interact with any other data sources such as Hive, Orc, Parquet, JSON, etc. The hbase-spark

integration applies critical techniques such as partition pruning, column pruning, predicate pushdown and data locality.

To use the hbase-spark integration connector, users need to define the Catalog for the schema mapping between HBase and Spark tables, prepare the data and populate the HBase table, then load the HBase DataFrame. After that, users can do integrated query and access records in HBase tables with SQL query. The following illustrates the basic procedure.

Define catalog

```
def catalog = s"""{
       |"table":{"namespace":"default", "name":"table1"},
       |"rowkey":"key",
       |"columns":{
         |"col0":{"cf":"rowkey", "col":"key", "type":"string"},
         |"col1":{"cf":"cf1", "col":"col1", "type":"boolean"},
         |"col2":{"cf":"cf2", "col":"col2", "type":"double"},
         |"col3":{"cf":"cf3", "col":"col3", "type":"float"},
         |"col4":{"cf":"cf4", "col":"col4", "type":"int"},
         |"col5":{"cf":"cf5", "col":"col5", "type":"bigint"},
         |"col6":{"cf":"cf6", "col":"col6", "type":"smallint"},
         |"col7":{"cf":"cf7", "col":"col7", "type":"string"},
         |"col8":{"cf":"cf8", "col":"col8", "type":"tinyint"}
       |}
     |}""".stripMargin
```

Catalog defines a mapping between HBase and Spark tables. There are two critical parts of this catalog. One is the rowkey definition and the other is the mapping between table column in Spark and the column family and column qualifier in HBase. The above defines a schema for a HBase table with name as table1, row key as key and a number of columns (col1 `-` col8). Note that the rowkey also has to be defined in details as a column (col0), which has a specific cf (rowkey).

Save the DataFrame

```
case class HBaseRecord(
   col0: String,
   col1: Boolean,
   col2: Double,
   col3: Float,
   col4: Int,
   col5: Long,
   col6: Short,
```

```
   col7: String,
   col8: Byte)

object HBaseRecord
{
   def apply(i: Int, t: String): HBaseRecord = {
      val s = s"""row${"%03d".format(i)}"""
      HBaseRecord(s,
      i % 2 == 0,
      i.toDouble,
      i.toFloat,
      i,
      i.toLong,
      i.toShort,
      s"String$i: $t",
      i.toByte)
  }
}

val data = (0 to 255).map { i =>  HBaseRecord(i, "extra")}

sc.parallelize(data).toDF.write.options(
 Map(HBaseTableCatalog.tableCatalog -> catalog, HBaseTableCatalog.newTable ->
"5"))
 .format("org.apache.hadoop.hbase.spark ")
 .save()
```

`data` prepared by the user is a local Scala collection which has 256 HBaseRecord objects. `sc.parallelize(data)` function distributes `data` to form an RDD. `toDF` returns a DataFrame. `write` function returns a DataFrameWriter used to write the DataFrame to external storage systems (e.g. HBase here). Given a DataFrame with specified schema `catalog`, `save` function will create an HBase table with 5 regions and save the DataFrame inside.

Load the DataFrame

```
def withCatalog(cat: String): DataFrame = {
  sqlContext
  .read
  .options(Map(HBaseTableCatalog.tableCatalog->cat))
  .format("org.apache.hadoop.hbase.spark")
  .load()
}
val df = withCatalog(catalog)
```

In 'withCatalog' function, sqlContext is a variable of SQLContext, which is the entry point for working with structured data (rows and columns) in Spark. `read` returns a

DataFrameReader that can be used to read data in as a DataFrame. `option` function adds input options for the underlying data source to the DataFrameReader, and `format` function specifies the input data source format for the DataFrameReader. The `load()` function loads input in as a DataFrame. The date frame `df` returned by `withCatalog` function could be used to access HBase table, such as 4.4 and 4.5.

Language Integrated Query

```
val s = df.filter(($"col0" <= "row050" && $"col0" > "row040") ||
  $"col0" === "row005" ||
  $"col0" <= "row005")
  .select("col0", "col1", "col4")
s.show
```

DataFrame can do various operations, such as join, sort, select, filter, orderBy and so on. `df.filter` above filters rows using the given SQL expression. `select` selects a set of columns: `col0`, `col1` and `col4`.

SQL Query

```
df.registerTempTable("table1")
sqlContext.sql("select count(col1) from table1").show
```

`registerTempTable` registers `df` DataFrame as a temporary table using the table name `table1`. The lifetime of this temporary table is tied to the SQLContext that was used to create `df`. `sqlContext.sql` function allows the user to execute SQL queries.

Others

Query with different timestamps

In HBaseSparkConf, four parameters related to timestamp can be set. They are TIMESTAMP, MIN_TIMESTAMP, MAX_TIMESTAMP and MAX_VERSIONS respectively. Users can query records with different timestamps or time ranges with MIN_TIMESTAMP and MAX_TIMESTAMP. In the meantime, use concrete value instead of tsSpecified and oldMs in the examples below.

The example below shows how to load df DataFrame with different timestamps. tsSpecified is specified by the user. HBaseTableCatalog defines the HBase and Relation relation schema. writeCatalog defines catalog for the schema mapping.

```
val df = sqlContext.read
      .options(Map(HBaseTableCatalog.tableCatalog -> writeCatalog, HBaseSparkCon
f.TIMESTAMP -> tsSpecified.toString))
      .format("org.apache.hadoop.hbase.spark")
      .load()
```

The example below shows how to load df DataFrame with different time ranges. oldMs is specified by the user.

```
val df = sqlContext.read
      .options(Map(HBaseTableCatalog.tableCatalog -> writeCatalog, HBaseSparkCon
f.MIN_TIMESTAMP -> "0",
        HBaseSparkConf.MAX_TIMESTAMP -> oldMs.toString))
      .format("org.apache.hadoop.hbase.spark")
      .load()
```

After loading df DataFrame, users can query data.

```
df.registerTempTable("table")
sqlContext.sql("select count(col1) from table").show
```

Native Avro support

The hbase-spark integration connector supports different data formats like Avro, JSON, etc. The use case below shows how spark supports Avro. Users can persist the Avro record into HBase directly. Internally, the Avro schema is converted to a native Spark Catalyst data type automatically. Note that both key-value parts in an HBase table can be defined in Avro format.

1. Define catalog for the schema mapping:

```
def catalog = s"""{
                    |"table":{"namespace":"default", "name":"Avrotable"},
                      |"rowkey":"key",
                      |"columns":{
                      |"col0":{"cf":"rowkey", "col":"key", "type":"string"},
                      |"col1":{"cf":"cf1", "col":"col1", "type":"binary"}
                      |}
                      |}""".stripMargin
```

`catalog` is a schema for a HBase table named `Avrotable`. row key as key and one column col1. The rowkey also has to be defined in details as a column (col0), which has a specific cf (rowkey).

2. Prepare the Data:

```
object AvroHBaseRecord {
  val schemaString =
    s"""{"namespace": "example.avro",
        |   "type": "record",      "name": "User",
        |    "fields": [
        |        {"name": "name", "type": "string"},
        |        {"name": "favorite_number",  "type": ["int", "null"]},
        |        {"name": "favorite_color", "type": ["string", "null"]},
        |        {"name": "favorite_array", "type": {"type": "array", "item
s": "string"}},
        |        {"name": "favorite_map", "type": {"type": "map", "values":
"int"}}
        |      ]    }""".stripMargin

  val avroSchema: Schema = {
    val p = new Schema.Parser
    p.parse(schemaString)
  }

  def apply(i: Int): AvroHBaseRecord = {
    val user = new GenericData.Record(avroSchema);
    user.put("name", s"name${"%03d".format(i)}")
    user.put("favorite_number", i)
    user.put("favorite_color", s"color${"%03d".format(i)}")
    val favoriteArray = new GenericData.Array[String](2, avroSchema.getField
("favorite_array").schema())
    favoriteArray.add(s"number${i}")
    favoriteArray.add(s"number${i+1}")
    user.put("favorite_array", favoriteArray)
    import collection.JavaConverters._
    val favoriteMap = Map[String, Int](("key1" -> i), ("key2" -> (i+1))).asJa
va
    user.put("favorite_map", favoriteMap)
    val avroByte = AvroSedes.serialize(user, avroSchema)
    AvroHBaseRecord(s"name${"%03d".format(i)}", avroByte)
  }
}

val data = (0 to 255).map { i =>
    AvroHBaseRecord(i)
}
```

`schemaString` is defined first, then it is parsed to get `avroSchema`. `avroSchema` is used to generate `AvroHBaseRecord`. `data` prepared by users is a local Scala collection which

has 256 `AvroHBaseRecord` objects.

3. Save DataFrame:

```
sc.parallelize(data).toDF.write.options(
    Map(HBaseTableCatalog.tableCatalog -> catalog, HBaseTableCatalog.newTable
-> "5"))
    .format("org.apache.spark.sql.execution.datasources.hbase")
    .save()
```

Given a data frame with specified schema `catalog`, above will create an HBase table with 5 regions and save the data frame inside.

4. Load the DataFrame

```
def avroCatalog = s"""{
            |"table":{"namespace":"default", "name":"avrotable"},
            |"rowkey":"key",
            |"columns":{
              |"col0":{"cf":"rowkey", "col":"key", "type":"string"},
              |"col1":{"cf":"cf1", "col":"col1", "avro":"avroSchema"}
            |}
          |}""".stripMargin

def withCatalog(cat: String): DataFrame = {
    sqlContext
        .read
        .options(Map("avroSchema" -> AvroHBaseRecord.schemaString, HBaseTable
Catalog.tableCatalog -> avroCatalog))
        .format("org.apache.spark.sql.execution.datasources.hbase")
        .load()
}
val df = withCatalog(catalog)
```

In `withCatalog` function, `read` returns a DataFrameReader that can be used to read data in as a DataFrame. The `option` function adds input options for the underlying data source to the DataFrameReader. There are two options: one is to set `avroSchema` as `AvroHBaseRecord.schemaString`, and one is to set `HBaseTableCatalog.tableCatalog` as `avroCatalog`. The `load()` function loads input in as a DataFrame. The date frame `df` returned by `withCatalog` function could be used to access the HBase table.

5. SQL Query

```
df.registerTempTable("avrotable")
```

```
val c = sqlContext.sql("select count(1) from avrotable").
```

After loading df DataFrame, users can query data. registerTempTable registers df DataFrame as a temporary table using the table name avrotable. `sqlContext.sql` function allows the user to execute SQL queries.

Apache HBase Coprocessors

HBase Coprocessors are modeled after Google BigTable's coprocessor implementation (http://research.google.com/people/jeff/SOCC2010-keynote-slides.pdf pages 41-42.). Efforts are ongoing to bridge gaps between HBase's implementation and BigTable's architecture. For more information see HBASE-4047.

The information in this chapter is primarily sourced and heavily reused from the following resources:

1. Mingjie Lai's blog post Coprocessor Introduction.
2. Gaurav Bhardwaj's blog post The How To Of HBase Coprocessors.

> ⚠ **Use Coprocessors At Your Own Risk**
>
> Coprocessors are an advanced feature of HBase and are intended to be used by system developers only. Because coprocessor code runs directly on the RegionServer and has direct access to your data, they introduce the risk of data corruption, man-in-the-middle attacks, or other malicious data access. Currently, there is no mechanism to prevent data corruption by coprocessors, though work is underway on HBASE-4047.
>
> In addition, there is no resource isolation, so a well-intentioned but misbehaving coprocessor can severely degrade cluster performance and stability.

Coprocessor Overview

In HBase, you fetch data using a `Get` or `Scan`, whereas in an RDBMS you use a SQL query. In order to fetch only the relevant data, you filter it using a HBase Filter , whereas in an RDBMS you use a `WHERE` predicate.

After fetching the data, you perform computations on it. This paradigm works well for "small data" with a few thousand rows and several columns. However, when you scale to billions of rows and millions of columns, moving large amounts of data across your network will create bottlenecks at the network layer, and the client needs to be powerful enough and have enough memory to handle the large amounts of data and the computations. In addition, the client code can grow large and complex.

In this scenario, coprocessors might make sense. You can put the business computation code into a coprocessor which runs on the RegionServer, in the same location as the data,

and returns the result to the client.

This is only one scenario where using coprocessors can provide benefit. Following are some analogies which may help to explain some of the benefits of coprocessors.

Coprocessor Analogies

Triggers and Stored Procedure

An Observer coprocessor is similar to a trigger in a RDBMS in that it executes your code either before or after a specific event (such as a `Get` or `Put`) occurs. An endpoint coprocessor is similar to a stored procedure in a RDBMS because it allows you to perform custom computations on the data on the RegionServer itself, rather than on the client.

MapReduce

MapReduce operates on the principle of moving the computation to the location of the data. Coprocessors operate on the same principal.

AOP

If you are familiar with Aspect Oriented Programming (AOP), you can think of a coprocessor as applying advice by intercepting a request and then running some custom code, before passing the request on to its final destination (or even changing the destination).

Coprocessor Implementation Overview

1. Your class should implement one of the Coprocessor interfaces - Coprocessor, RegionObserver, CoprocessorService - to name a few.
2. Load the coprocessor, either statically (from the configuration) or dynamically, using HBase Shell. For more details see Loading Coprocessors.
3. Call the coprocessor from your client-side code. HBase handles the coprocessor transparently.

The framework API is provided in the coprocessor package.

Types of Coprocessors

Observer Coprocessors

Observer coprocessors are triggered either before or after a specific event occurs. Observers that happen before an event use methods that start with a `pre` prefix, such as `prePut`. Observers that happen just after an event override methods that start with a `post` prefix, such as `postPut`.

Use Cases for Observer Coprocessors

Security

Before performing a `Get` or `Put` operation, you can check for permission using `preGet` or `prePut` methods.

Referential Integrity

HBase does not directly support the RDBMS concept of refential integrity, also known as foreign keys. You can use a coprocessor to enforce such integrity. For instance, if you have a business rule that every insert to the `users` table must be followed by a corresponding entry in the `user_daily_attendance` table, you could implement a coprocessor to use the `prePut` method on `user` to insert a record into `user_daily_attendance`.

Secondary Indexes

You can use a coprocessor to maintain secondary indexes. For more information, see SecondaryIndexing.

Types of Observer Coprocessor

RegionObserver

A RegionObserver coprocessor allows you to observe events on a region, such as `Get` and `Put` operations. See RegionObserver.

RegionServerObserver

A RegionServerObserver allows you to observe events related to the RegionServer's operation, such as starting, stopping, or performing merges, commits, or rollbacks. See RegionServerObserver.

MasterObserver

A MasterObserver allows you to observe events related to the HBase Master, such as table creation, deletion, or schema modification. See MasterObserver.

WalObserver

A WalObserver allows you to observe events related to writes to the Write-Ahead Log (WAL). See WALObserver.

Examples provides working examples of observer coprocessors.

Endpoint Coprocessor

Endpoint processors allow you to perform computation at the location of the data. See Coprocessor Analogy. An example is the need to calculate a running average or summation for an entire table which spans hundreds of regions.

In contrast to observer coprocessors, where your code is run transparently, endpoint coprocessors must be explicitly invoked using the CoprocessorService() method available in AsyncTable.

> ⚠ On using coprocessorService method with sync client
>
> The coprocessorService method in Table has been deprecated.
>
> In HBASE-21512 we reimplement the sync client based on the async client. The coprocessorService method defined in `Table` interface directly references a method from protobuf's `BlockingInterface`, which means we need to use a separate thread pool to execute the method so we avoid blocking the async client(We want to avoid blocking calls in our async implementation).
>
> Since coprocessor is an advanced feature, we believe it is OK for coprocessor users to instead switch over to use `AsyncTable`. There is a lightweight toAsyncConnection method to get an `AsyncConnection` from `Connection` if needed.

Starting with HBase 0.96, endpoint coprocessors are implemented using Google Protocol Buffers (protobuf). For more details on protobuf, see Google's Protocol Buffer Guide. Endpoints Coprocessor written in version 0.94 are not compatible with version 0.96 or later. See HBASE-5448). To upgrade your HBase cluster from 0.94 or earlier to 0.96 or later, you need to reimplement your coprocessor.

In HBase 2.x, we made use of a shaded version of protobuf 3.x, but kept the protobuf for coprocessors on 2.5.0. In HBase 3.0.0, we removed all dependencies on non-shaded

protobuf so you need to reimplement your coprocessor to make use of the shaded protobuf version provided in hbase-thirdparty. Please see the **protobuf** section for more details.

Coprocessor Endpoints should make no use of HBase internals and only avail of public APIs; ideally a CPEP should depend on Interfaces and data structures only. This is not always possible but beware that doing so makes the Endpoint brittle, liable to breakage as HBase internals evolve. HBase internal APIs annotated as private or evolving do not have to respect semantic versioning rules or general java rules on deprecation before removal. While generated protobuf files are absent the hbase audience annotations — they are created by the protobuf protoc tool which knows nothing of how HBase works — they should be consided `@InterfaceAudience.Private` so are liable to change.

Examples provides working examples of endpoint coprocessors.

Loading Coprocessors

To make your coprocessor available to HBase, it must be *loaded*, either statically (through the HBase configuration) or dynamically (using HBase Shell or the Java API).

Static Loading

Follow these steps to statically load your coprocessor. Keep in mind that you must restart HBase to unload a coprocessor that has been loaded statically.

1. Define the Coprocessor in *hbase-site.xml*, with a `<property>` element with a `<name>` and a `<value>` sub-element. The `<name>` should be one of the following:

   - `hbase.coprocessor.region.classes` for RegionObservers and Endpoints.
   - `hbase.coprocessor.wal.classes` for WALObservers.
   - `hbase.coprocessor.master.classes` for MasterObservers.

   `<value>` must contain the fully-qualified class name of your coprocessor's implementation class.

For example to load a Coprocessor (implemented in class SumEndPoint.java) you have to create following entry in RegionServer's 'hbase-site.xml' file (generally located under 'conf' directory):

```
<property>
    <name>hbase.coprocessor.region.classes</name>
    <value>org.myname.hbase.coprocessor.endpoint.SumEndPoint</value>
</property>
```

If multiple classes are specified for loading, the class names must be comma-separated. The framework attempts to load all the configured classes using the default class loader. Therefore, the jar file must reside on the server-side HBase classpath.

Coprocessors which are loaded in this way will be active on all regions of all tables. These are also called system Coprocessor. The first listed Coprocessors will be assigned the priority `Coprocessor.Priority.SYSTEM`. Each subsequent coprocessor in the list will have its priority value incremented by one (which reduces its priority, because priorities have the natural sort order of Integers).

These priority values can be manually overriden in hbase-site.xml. This can be useful if you want to guarantee that a coprocessor will execute after another. For example, in the following configuration `SumEndPoint` would be guaranteed to go last, except in the case of a tie with another coprocessor:

```
<property>
    <name>hbase.coprocessor.region.classes</name>
    <value>org.myname.hbase.coprocessor.endpoint.SumEndPoint|2147483647</va
lue>
</property>
```

When calling out to registered observers, the framework executes their callbacks methods in the sorted order of their priority. Ties are broken arbitrarily.

2. Put your code on HBase's classpath. One easy way to do this is to drop the jar (containing you code and all the dependencies) into the `lib/` directory in the HBase installation.

3. Restart HBase.

Static Unloading

1. Delete the coprocessor's `<property>` element, including sub-elements, from `hbase-site.xml`.
2. Restart HBase.
3. Optionally, remove the coprocessor's JAR file from the classpath or HBase's `lib/` directory.

Dynamic Loading

You can also load a coprocessor dynamically, without restarting HBase. This may seem preferable to static loading, but dynamically loaded coprocessors are loaded on a per-table basis, and are only available to the table for which they were loaded. For this reason, dynamically loaded tables are sometimes called **Table Coprocessor**.

In addition, dynamically loading a coprocessor acts as a schema change on the table, and the table must be taken offline to load the coprocessor.

There are three ways to dynamically load Coprocessor.

> **Assumptions**
>
> The below mentioned instructions makes the following assumptions:
>
> - A JAR called `coprocessor.jar` contains the Coprocessor implementation along with all of its dependencies.
> - The JAR is available in HDFS in some location like `hdfs://NAMENODE:PORT/user/HADOOP_USER/coprocessor.jar`.

Using HBase Shell

1. Load the Coprocessor, using a command like the following:

```
hbase alter 'users', METHOD => 'table_att', 'Coprocessor'=>'hdfs://NAMENODE:PORT/user/HADOOP_USER/coprocessor.jar|org.myname.hbase.Coprocessor.RegionObserverExample|1073741823|arg1=1,arg2=2'
```

   The Coprocessor framework will try to read the class information from the coprocessor table attribute value. The value contains four pieces of information which are separated by the pipe (`|`) character.

- File path: The jar file containing the Coprocessor implementation must be in a location where all region servers can read it. You could copy the file onto the local disk on each region server, but it is recommended to store it in HDFS. HBASE-14548 allows a directory containing the jars or some wildcards to be specified, such as: `hdfs://NAMENODE:PORT/user/HADOOP_USER/` or `hdfs://NAMENODE:PORT/user/HADOOP_USER/*.jar`. Please note that if a directory is specified, all jar files(.jar) in the directory are added. It does not search for files in sub-directories. Do not use a wildcard if you would like to specify a directory. This enhancement applies to the usage via the JAVA API as well.
- Class name: The full class name of the Coprocessor.
- Priority: An integer. The framework will determine the execution sequence of all configured observers registered at the same hook using priorities. This field can be left blank. In that case the framework will assign a default priority value.
- Arguments (Optional): This field is passed to the Coprocessor implementation. This is optional.

2. Verify that the coprocessor loaded:

```
hbase(main):04:0> describe 'users'
```

The coprocessor should be listed in the `TABLE_ATTRIBUTES`.

Using the Java API (all HBase versions)

The following Java code shows how to use the `setValue()` method of `HTableDescriptor` to load a coprocessor on the `users` table.

```
TableName tableName = TableName.valueOf("users");
String path = "hdfs://<namenode>:<port>/user/<hadoop-user>/coprocessor.jar";
Configuration conf = HBaseConfiguration.create();
Connection connection = ConnectionFactory.createConnection(conf);
Admin admin = connection.getAdmin();
HTableDescriptor hTableDescriptor = new HTableDescriptor(tableName);
HColumnDescriptor columnFamily1 = new HColumnDescriptor("personalDet");
columnFamily1.setMaxVersions(3);
hTableDescriptor.addFamily(columnFamily1);
HColumnDescriptor columnFamily2 = new HColumnDescriptor("salaryDet");
columnFamily2.setMaxVersions(3);
hTableDescriptor.addFamily(columnFamily2);
hTableDescriptor.setValue("COPROCESSOR$1", path + "|"
+ RegionObserverExample.class.getCanonicalName() + "|"
```

```
+ Coprocessor.PRIORITY_USER);
admin.modifyTable(tableName, hTableDescriptor);
```

Using the Java API (HBase 0.96+ only)

In HBase 0.96 and newer, the `addCoprocessor()` method of `HTableDescriptor` provides an easier way to load a coprocessor dynamically.

```
TableName tableName = TableName.valueOf("users");
Path path = new Path("hdfs://<namenode>:<port>/user/<hadoop-user>/coprocessor.ja
r");
Configuration conf = HBaseConfiguration.create();
Connection connection = ConnectionFactory.createConnection(conf);
Admin admin = connection.getAdmin();
HTableDescriptor hTableDescriptor = new HTableDescriptor(tableName);
HColumnDescriptor columnFamily1 = new HColumnDescriptor("personalDet");
columnFamily1.setMaxVersions(3);
hTableDescriptor.addFamily(columnFamily1);
HColumnDescriptor columnFamily2 = new HColumnDescriptor("salaryDet");
columnFamily2.setMaxVersions(3);
hTableDescriptor.addFamily(columnFamily2);
hTableDescriptor.addCoprocessor(RegionObserverExample.class.getCanonicalName(), p
ath,
Coprocessor.PRIORITY_USER, null);
admin.modifyTable(tableName, hTableDescriptor);
```

> ⚠ There is no guarantee that the framework will load a given Coprocessor successfully. For example, the shell command neither guarantees a jar file exists at a particular location nor verifies whether the given class is actually contained in the jar file.

Dynamic Unloading

Using HBase Shell

1. Alter the table to remove the coprocessor with `table_att_unset`.

```
hbase> alter 'users', METHOD => 'table_att_unset', NAME => 'coprocessor$1'
```

2. Alter the table to remove the coprocessor with `table_remove_coprocessor` introduced in HBASE-26524 by specifying an explicit classname

```
hbase> alter 'users', METHOD => 'table_remove_coprocessor', CLASSNAME => \
    'org.myname.hbase.Coprocessor.RegionObserverExample'
```

Using the Java API

Reload the table definition without setting the value of the coprocessor either by using `setValue()` or `addCoprocessor()` methods. This will remove any coprocessor attached to the table.

```
TableName tableName = TableName.valueOf("users");
String path = "hdfs://<namenode>:<port>/user/<hadoop-user>/coprocessor.jar";
Configuration conf = HBaseConfiguration.create();
Connection connection = ConnectionFactory.createConnection(conf);
Admin admin = connection.getAdmin();
HTableDescriptor hTableDescriptor = new HTableDescriptor(tableName);
HColumnDescriptor columnFamily1 = new HColumnDescriptor("personalDet");
columnFamily1.setMaxVersions(3);
hTableDescriptor.addFamily(columnFamily1);
HColumnDescriptor columnFamily2 = new HColumnDescriptor("salaryDet");
columnFamily2.setMaxVersions(3);
hTableDescriptor.addFamily(columnFamily2);
admin.modifyTable(tableName, hTableDescriptor);
```

In HBase 0.96 and newer, you can instead use the `removeCoprocessor()` method of the `HTableDescriptor` class.

Examples

HBase ships examples for Observer Coprocessor.

A more detailed example is given below.

These examples assume a table called `users`, which has two column families `personalDet` and `salaryDet`, containing personal and salary details. Below is the graphical representation of the `users` table.

Users Table

	personalDet			salaryDet		
rowkey	name	lastname	dob	gross	net	allowances
admin	Admin	Admin				
cdickens	Charles	Dickens	02/07/1812	10000	8000	2000

	personalDet			salaryDet		
jverne	Jules	Verne	02/08/1828	12000	9000	3000

Observer Example

The following Observer coprocessor prevents the details of the user `admin` from being returned in a `Get` or `Scan` of the `users` table.

1. Write a class that implements the RegionCoprocessor, RegionObserver class.
2. Override the `preGetOp()` method (the `preGet()` method is deprecated) to check whether the client has queried for the rowkey with value `admin`. If so, return an empty result. Otherwise, process the request as normal.
3. Put your code and dependencies in a JAR file.
4. Place the JAR in HDFS where HBase can locate it.
5. Load the Coprocessor.
6. Write a simple program to test it.

Following are the implementation of the above steps:

```
public class RegionObserverExample implements RegionCoprocessor, RegionObserver {

    private static final byte[] ADMIN = Bytes.toBytes("admin");
    private static final byte[] COLUMN_FAMILY = Bytes.toBytes("details");
    private static final byte[] COLUMN = Bytes.toBytes("Admin_det");
    private static final byte[] VALUE = Bytes.toBytes("You can't see Admin detail
s");

    @Override
    public Optional<RegionObserver> getRegionObserver() {
      return Optional.of(this);
    }

    @Override
    public void preGetOp(final ObserverContext<RegionCoprocessorEnvironment> e, f
inal Get get, final List<Cell> results)
    throws IOException {

        if (Bytes.equals(get.getRow(),ADMIN)) {
            Cell c = CellUtil.createCell(get.getRow(),COLUMN_FAMILY, COLUMN,
            System.currentTimeMillis(), (byte)4, VALUE);
            results.add(c);
```

```
            e.bypass();
        }
    }
}
```

Overriding the `preGetOp()` will only work for `Get` operations. You also need to override the `preScannerOpen()` method to filter the `admin` row from scan results.

```
@Override
public RegionScanner preScannerOpen(final ObserverContext<RegionCoprocessorEnviro
nment> e, final Scan scan,
final RegionScanner s) throws IOException {

    Filter filter = new RowFilter(CompareOp.NOT_EQUAL, new BinaryComparator(ADMI
N));
    scan.setFilter(filter);
    return s;
}
```

This method works but there is a *side effect*. If the client has used a filter in its scan, that filter will be replaced by this filter. Instead, you can explicitly remove any `admin` results from the scan:

```
@Override
public boolean postScannerNext(final ObserverContext<RegionCoprocessorEnvironment
> e, final InternalScanner s,
final List<Result> results, final int limit, final boolean hasMore) throws IOExce
ption {
        Result result = null;
    Iterator<Result> iterator = results.iterator();
    while (iterator.hasNext()) {
    result = iterator.next();
        if (Bytes.equals(result.getRow(), ROWKEY)) {
            iterator.remove();
            break;
        }
    }
    return hasMore;
}
```

Endpoint Example

Still using the `users` table, this example implements a coprocessor to calculate the sum of all employee salaries, using an endpoint coprocessor.

1. Create a '.proto' file defining your service.

```
option java_package = "org.myname.hbase.coprocessor.autogenerated";
option java_outer_classname = "Sum";
option java_generic_services = true;
option java_generate_equals_and_hash = true;
option optimize_for = SPEED;
message SumRequest {
    required string family = 1;
    required string column = 2;
}

message SumResponse {
required int64 sum = 1 [default = 0];
}

service SumService {
rpc getSum(SumRequest)
    returns (SumResponse);
}
```

2. Execute the `protoc` command to generate the Java code from the above .proto' file.

```
$ mkdir src
$ protoc --java_out=src ./sum.proto
```

This will generate a class call `Sum.java`.

3. Write a class that extends the generated service class, implement the `Coprocessor` and `CoprocessorService` classes, and override the service method.

> ⚠ If you load a coprocessor from `hbase-site.xml` and then load the same coprocessor again using HBase Shell, it will be loaded a second time. The same class will exist twice, and the second instance will have a higher ID (and thus a lower priority). The effect is that the duplicate coprocessor is effectively ignored.

```
public class SumEndPoint extends Sum.SumService implements Coprocessor, Copro
cessorService {

    private RegionCoprocessorEnvironment env;

    @Override
    public Service getService() {
        return this;
    }

    @Override
    public void start(CoprocessorEnvironment env) throws IOException {
        if (env instanceof RegionCoprocessorEnvironment) {
```

```
            this.env = (RegionCoprocessorEnvironment)env;
        } else {
            throw new CoprocessorException("Must be loaded on a table regio
n!");
        }
    }

    @Override
    public void stop(CoprocessorEnvironment env) throws IOException {
        // do nothing
    }

    @Override
    public void getSum(RpcController controller, Sum.SumRequest request, RpcC
allback<Sum.SumResponse> done) {
        Scan scan = new Scan();
        scan.addFamily(Bytes.toBytes(request.getFamily()));
        scan.addColumn(Bytes.toBytes(request.getFamily()), Bytes.toBytes(requ
est.getColumn()));

        Sum.SumResponse response = null;
        InternalScanner scanner = null;

        try {
            scanner = env.getRegion().getScanner(scan);
            List<Cell> results = new ArrayList<>();
            boolean hasMore = false;
            long sum = 0L;

            do {
                hasMore = scanner.next(results);
                for (Cell cell : results) {
                    sum = sum + Bytes.toLong(CellUtil.cloneValue(cell));
                }
                results.clear();
            } while (hasMore);

            response = Sum.SumResponse.newBuilder().setSum(sum).build();
        } catch (IOException ioe) {
            ResponseConverter.setControllerException(controller, ioe);
        } finally {
            if (scanner != null) {
                try {
                    scanner.close();
                } catch (IOException ignored) {}
            }
        }

        done.run(response);
    }
}
```

```
Configuration conf = HBaseConfiguration.create();
Connection connection = ConnectionFactory.createConnection(conf);
```

```
TableName tableName = TableName.valueOf("users");
Table table = connection.getTable(tableName);

final Sum.SumRequest request = Sum.SumRequest.newBuilder().setFamily("salaryD
et").setColumn("gross").build();
try {
    Map<byte[], Long> results = table.coprocessorService(
        Sum.SumService.class,
        null,  /* start key */
        null,  /* end   key */
        new Batch.Call<Sum.SumService, Long>() {
            @Override
            public Long call(Sum.SumService aggregate) throws IOException {
                BlockingRpcCallback<Sum.SumResponse> rpcCallback = new Blocki
ngRpcCallback<>();
                aggregate.getSum(null, request, rpcCallback);
                Sum.SumResponse response = rpcCallback.get();

                return response.hasSum() ? response.getSum() : 0L;
            }
        }
    );

    for (Long sum : results.values()) {
        System.out.println("Sum = " + sum);
    }
} catch (ServiceException e) {
    e.printStackTrace();
} catch (Throwable e) {
    e.printStackTrace();
}
```

4. Load the Coprocessor.

5. Write a client code to call the Coprocessor.

Guidelines For Deploying A Coprocessor

Bundling Coprocessors

You can bundle all classes for a coprocessor into a single JAR on the RegionServer's classpath, for easy deployment. Otherwise, place all dependencies on the RegionServer's classpath so that they can be loaded during RegionServer start-up. The classpath for a RegionServer is set in the RegionServer's `hbase-env.sh` file.

Automating Deployment

You can use a tool such as Puppet, Chef, or Ansible to ship the JAR for the coprocessor to the required location on your RegionServers' filesystems and restart each RegionServer, to

automate coprocessor deployment. Details for such set-ups are out of scope of this document.

Updating a Coprocessor

Deploying a new version of a given coprocessor is not as simple as disabling it, replacing the JAR, and re-enabling the coprocessor. This is because you cannot reload a class in a JVM unless you delete all the current references to it. Since the current JVM has reference to the existing coprocessor, you must restart the JVM, by restarting the RegionServer, in order to replace it. This behavior is not expected to change.

Coprocessor Logging

The Coprocessor framework does not provide an API for logging beyond standard Java logging.

Coprocessor Configuration

If you do not want to load coprocessors from the HBase Shell, you can add their configuration properties to `hbase-site.xml`. In Using HBase Shell, two arguments are set: `arg1=1,arg2=2`. These could have been added to `hbase-site.xml` as follows:

```
<property>
  <name>arg1</name>
  <value>1</value>
</property>
<property>
  <name>arg2</name>
  <value>2</value>
</property>
```

Then you can read the configuration using code like the following:

```
Configuration conf = HBaseConfiguration.create();
Connection connection = ConnectionFactory.createConnection(conf);
TableName tableName = TableName.valueOf("users");
Table table = connection.getTable(tableName);

Get get = new Get(Bytes.toBytes("admin"));
Result result = table.get(get);
for (Cell c : result.rawCells()) {
    System.out.println(Bytes.toString(CellUtil.cloneRow(c))
        + "==> " + Bytes.toString(CellUtil.cloneFamily(c))
        + "{" + Bytes.toString(CellUtil.cloneQualifier(c))
        + ":" + Bytes.toLong(CellUtil.cloneValue(c)) + "}");
}
Scan scan = new Scan();
ResultScanner scanner = table.getScanner(scan);
```

```
for (Result res : scanner) {
    for (Cell c : res.rawCells()) {
        System.out.println(Bytes.toString(CellUtil.cloneRow(c))
        + " ==> " + Bytes.toString(CellUtil.cloneFamily(c))
        + " {" + Bytes.toString(CellUtil.cloneQualifier(c))
        + ":" + Bytes.toLong(CellUtil.cloneValue(c))
        + "}");
    }
}
```

Restricting Coprocessor Usage

Restricting arbitrary user coprocessors can be a big concern in multitenant environments. HBase provides a continuum of options for ensuring only expected coprocessors are running:

- `hbase.coprocessor.enabled`: Enables or disables all coprocessors. This will limit the functionality of HBase, as disabling all coprocessors will disable some security providers. An example coproccessor so affected is `org.apache.hadoop.hbase.security.access.AccessController`.
  - `hbase.coprocessor.user.enabled`: Enables or disables loading coprocessors on tables (i.e. user coprocessors).
  - One can statically load coprocessors, and optionally tune their priorities, via the following tunables in `hbase-site.xml`:
    - `hbase.coprocessor.regionserver.classes`: A comma-separated list of coprocessors that are loaded by region servers
    - `hbase.coprocessor.region.classes`: A comma-separated list of RegionObserver and Endpoint coprocessors
    - `hbase.coprocessor.user.region.classes`: A comma-separated list of coprocessors that are loaded by all regions
    - `hbase.coprocessor.master.classes`: A comma-separated list of coprocessors that are loaded by the master (MasterObserver coprocessors)
    - `hbase.coprocessor.wal.classes`: A comma-separated list of WALObserver coprocessors to load

- `hbase.coprocessor.abortonerror`: Whether to abort the daemon which has loaded the coprocessor if the coprocessor should error other than `IOError`. If this is set to false and an access controller coprocessor should have a fatal error the coprocessor will be circumvented, as such in secure installations this is advised to be `true`; however, one may override this on a per-table basis for user coprocessors, to ensure they do not abort their running region server and are instead unloaded on error.
- `hbase.coprocessor.region.whitelist.paths`: A comma separated list available for those loading `org.apache.hadoop.hbase.security.access.CoprocessorWhitelistMasterObserver` whereby one can use the following options to white-list paths from which coprocessors may be loaded.
  - Coprocessors on the classpath are implicitly white-listed
  - `*` to wildcard all coprocessor paths
  - An entire filesystem (e.g. `hdfs://my-cluster/`)
  - A wildcard path to be evaluated by FilenameUtils.wildcardMatch
  - Note: Path can specify scheme or not (e.g. `file:///usr/hbase/lib/coprocessors` or for all filesystems `/usr/hbase/lib/coprocessors`)

Apache HBase Performance Tuning

Operating System

Memory

RAM, RAM, RAM. Don't starve HBase.

64-bit

Use a 64-bit platform (and 64-bit JVM).

Swapping

Watch out for swapping. Set `swappiness` to 0.

CPU

Make sure you have set up your Hadoop to use native, hardware checksumming. See hadoop.native.lib.

Network

Perhaps the most important factor in avoiding network issues degrading Hadoop and HBase performance is the switching hardware that is used, decisions made early in the scope of the project can cause major problems when you double or triple the size of your cluster (or more).

Important items to consider:

- Switching capacity of the device
- Number of systems connected
- Uplink capacity

Single Switch

The single most important factor in this configuration is that the switching capacity of the hardware is capable of handling the traffic which can be generated by all systems connected to the switch. Some lower priced commodity hardware can have a slower switching capacity than could be utilized by a full switch.

Multiple Switches

Multiple switches are a potential pitfall in the architecture. The most common configuration of lower priced hardware is a simple 1Gbps uplink from one switch to another. This often overlooked pinch point can easily become a bottleneck for cluster communication. Especially with MapReduce jobs that are both reading and writing a lot of data the communication across this uplink could be saturated.

Mitigation of this issue is fairly simple and can be accomplished in multiple ways:

- Use appropriate hardware for the scale of the cluster which you're attempting to build.
- Use larger single switch configurations i.e. single 48 port as opposed to 2× 24 port
- Configure port trunking for uplinks to utilize multiple interfaces to increase cross switch bandwidth.

Multiple Racks

Multiple rack configurations carry the same potential issues as multiple switches, and can suffer performance degradation from two main areas:

- Poor switch capacity performance
- Insufficient uplink to another rack

If the switches in your rack have appropriate switching capacity to handle all the hosts at full speed, the next most likely issue will be caused by homing more of your cluster across racks. The easiest way to avoid issues when spanning multiple racks is to use port trunking to create a bonded uplink to other racks. The downside of this method however, is in the overhead of ports that could potentially be used. An example of this is, creating an 8Gbps port channel from rack A to rack B, using 8 of your 24 ports to communicate between racks gives you a poor ROI, using too few however can mean you're not getting the most out of your cluster.

Using 10Gbe links between racks will greatly increase performance, and assuming your switches support a 10Gbe uplink or allow for an expansion card will allow you to save your ports for machines as opposed to uplinks.

Network Interfaces

Are all the network interfaces functioning correctly? Are you sure? See the Troubleshooting Case Study in Case Study #1 (Performance Issue On A Single Node).

Network Consistency and Partition Tolerance

The CAP Theorem states that a distributed system can maintain two out of the following three characteristics:

- *C*onsistency — all nodes see the same data.
- *A*vailability — every request receives a response about whether it succeeded or failed.
- *P*artition tolerance — the system continues to operate even if some of its components become unavailable to the others.

HBase favors consistency and partition tolerance, where a decision has to be made. Coda Hale explains why partition tolerance is so important, in http://codahale.com/you-cant-sacrifice-partition-tolerance/.

Robert Yokota used an automated testing framework called Jepson to test HBase's partition tolerance in the face of network partitions, using techniques modeled after Aphyr's Call Me Maybe series. The results, available as a blog post and an addendum, show that HBase performs correctly.

Java

The Garbage Collector and Apache HBase

Long GC pauses

In his presentation, Avoiding Full GCs with MemStore-Local Allocation Buffers, Todd Lipcon describes two cases of stop-the-world garbage collections common in HBase, especially during loading; CMS failure modes and old generation heap fragmentation brought.

To address the first, start the CMS earlier than default by adding `-XX:CMSInitiatingOccupancyFraction` and setting it down from defaults. Start at 60 or 70 percent (The lower you bring down the threshold, the more GCing is done, the more CPU used). To address the second fragmentation issue, Todd added an experimental facility, (MSLAB), that must be explicitly enabled in Apache HBase 0.90.x (It's defaulted to be *on* in Apache 0.92.x HBase). Set `hbase.hregion.memstore.mslab.enabled` to true in your `Configuration`. See the cited slides for background and detail. The latest JVMs do better regards fragmentation so make sure you are running a recent release. Read down in the message, **Identifying concurrent mode failures caused by fragmentation**. Be aware that when enabled, each MemStore instance will occupy at least an MSLAB instance of memory. If you have thousands of regions or lots of regions each with many column families, this allocation of MSLAB may be responsible for a good portion of your heap allocation and in an extreme case cause you to OOME. Disable MSLAB in this case, or lower the amount of memory it uses or float less regions per server.

If you have a write-heavy workload, check out **HBASE-8163 MemStoreChunkPool: An improvement for JAVA GC when using MSLAB**. It describes configurations to lower the amount of young GC during write-heavy loadings. If you do not have HBASE-8163 installed, and you are trying to improve your young GC times, one trick to consider — courtesy of our Liang Xie — is to set the GC config `-XX:PretenureSizeThreshold` in *hbase-env.sh* to be just smaller than the size of `hbase.hregion.memstore.mslab.chunksize` so MSLAB allocations happen in the tenured space directly rather than first in the young gen. You'd do this because these MSLAB allocations are going to likely make it to the old gen anyways and rather than pay the price of a copies between s0 and s1 in eden space followed by the copy up from young to old gen after the MSLABs have achieved sufficient tenure, save a bit of YGC churn and allocate in the old gen directly.

Other sources of long GCs can be the JVM itself logging. See **Eliminating Large JVM GC Pauses Caused by Background IO Traffic**

For more information about GC logs, see **JVM Garbage Collection Logs**.

Consider also enabling the off-heap Block Cache. This has been shown to mitigate GC pause times. See **Block Cache**

HBase Configurations

See Recommended Configurations.

Improving the 99th Percentile

Try hedged_reads.

Managing Compactions

For larger systems, managing compactions and splits may be something you want to consider.

`hbase.regionserver.handler.count`

See hbase.regionserver.handler.count.

`hfile.block.cache.size`

See hfile.block.cache.size. A memory setting for the RegionServer process.

Prefetch Option for Blockcache

HBASE-9857 adds a new option to prefetch HFile contents when opening the BlockCache, if a Column family or RegionServer property is set. This option is available for HBase 0.98.3 and later. The purpose is to warm the BlockCache as rapidly as possible after the cache is opened, using in-memory table data, and not counting the prefetching as cache misses. This is great for fast reads, but is not a good idea if the data to be preloaded will not fit into the BlockCache. It is useful for tuning the IO impact of prefetching versus the time before all data blocks are in cache.

To enable prefetching on a given column family, you can use HBase Shell or use the API.

Enable Prefetch Using HBase Shell

```
hbase> create 'MyTable', { NAME => 'myCF', PREFETCH_BLOCKS_ON_OPEN => 'true' }
```

Enable Prefetch Using the API

```
// ...
HTableDescriptor tableDesc = new HTableDescriptor("myTable");
HColumnDescriptor cfDesc = new HColumnDescriptor("myCF");
cfDesc.setPrefetchBlocksOnOpen(true);
tableDesc.addFamily(cfDesc);
// ...
```

See the API documentation for CacheConfig.

To see prefetch in operation, enable TRACE level logging on `org.apache.hadoop.hbase.io.hfile.HFileReaderImpl` in hbase-2.0+ or on `org.apache.hadoop.hbase.io.hfile.HFileReaderV2` in earlier versions, hbase-1.x, of HBase.

`hbase.regionserver.global.memstore.size`

See hbase.regionserver.global.memstore.size. This memory setting is often adjusted for the RegionServer process depending on needs.

`hbase.regionserver.global.memstore.size.lower.limit`

See hbase.regionserver.global.memstore.size.lower.limit. This memory setting is often adjusted for the RegionServer process depending on needs.

`hbase.hstore.blockingStoreFiles`

See hbase.hstore.blockingStoreFiles. If there is blocking in the RegionServer logs, increasing this can help.

`hbase.hregion.memstore.block.multiplier`

See hbase.hregion.memstore.block.multiplier. If there is enough RAM, increasing this can help.

`hbase.regionserver.checksum.verify`

Have HBase write the checksum into the datablock and save having to do the checksum seek whenever you read.

See hbase.regionserver.checksum.verify, hbase.hstore.bytes.per.checksum and hbase.hstore.checksum.algorithm. For more information see the release note on HBASE-

5074 support checksums in HBase block cache.

Tuning `callQueue` Options

HBASE-11355 introduces several callQueue tuning mechanisms which can increase performance. See the JIRA for some benchmarking information.

To increase the number of callqueues, set `hbase.ipc.server.num.callqueue` to a value greater than `1`. To split the callqueue into separate read and write queues, set `hbase.ipc.server.callqueue.read.ratio` to a value between `0` and `1`. This factor weights the queues toward writes (if below .5) or reads (if above .5). Another way to say this is that the factor determines what percentage of the split queues are used for reads. The following examples illustrate some of the possibilities. Note that you always have at least one write queue, no matter what setting you use.

- The default value of `0` does not split the queue.
- A value of `.3` uses 30% of the queues for reading and 70% for writing. Given a value of `10` for `hbase.ipc.server.num.callqueue`, 3 queues would be used for reads and 7 for writes.
- A value of `.5` uses the same number of read queues and write queues. Given a value of `10` for `hbase.ipc.server.num.callqueue`, 5 queues would be used for reads and 5 for writes.
- A value of `.6` uses 60% of the queues for reading and 40% for writing. Given a value of `10` for `hbase.ipc.server.num.callqueue`, 6 queues would be used for reads and 4 for writes.
- A value of `1.0` uses one queue to process write requests, and all other queues process read requests. A value higher than `1.0` has the same effect as a value of `1.0`. Given a value of `10` for `hbase.ipc.server.num.callqueue`, 9 queues would be used for reads and 1 for writes.

You can also split the read queues so that separate queues are used for short reads (from Get operations) and long reads (from Scan operations), by setting the `hbase.ipc.server.callqueue.scan.ratio` option. This option is a factor between 0 and 1, which determine the ratio of read queues used for Gets and Scans. More queues are used for Gets if the value

is below `.5` and more are used for scans if the value is above `.5`. No matter what setting you use, at least one read queue is used for Get operations.

- A value of `0` does not split the read queue.
- A value of `.3` uses 70% of the read queues for Gets and 30% for Scans. Given a value of `20` for `hbase.ipc.server.num.callqueue` and a value of `.5` for `hbase.ipc.server.callqueue.read.ratio`, 10 queues would be used for reads, out of those 10, 7 would be used for Gets and 3 for Scans.
- A value of `.5` uses half the read queues for Gets and half for Scans. Given a value of `20` for `hbase.ipc.server.num.callqueue` and a value of `.5` for `hbase.ipc.server.callqueue.read.ratio`, 10 queues would be used for reads, out of those 10, 5 would be used for Gets and 5 for Scans.
- A value of `.7` uses 30% of the read queues for Gets and 70% for Scans. Given a value of `20` for `hbase.ipc.server.num.callqueue` and a value of `.5` for `hbase.ipc.server.callqueue.read.ratio`, 10 queues would be used for reads, out of those 10, 3 would be used for Gets and 7 for Scans.
- A value of `1.0` uses all but one of the read queues for Scans. Given a value of `20` for `hbase.ipc.server.num.callqueue` and a value of `.5` for `hbase.ipc.server.callqueue.read.ratio`, 10 queues would be used for reads, out of those 10, 1 would be used for Gets and 9 for Scans.

You can use the new option `hbase.ipc.server.callqueue.handler.factor` to programmatically tune the number of queues:

- A value of `0` uses a single shared queue between all the handlers.
- A value of `1` uses a separate queue for each handler.
- A value between `0` and `1` tunes the number of queues against the number of handlers. For instance, a value of `.5` shares one queue between each two handlers.
  Having more queues, such as in a situation where you have one queue per handler, reduces contention when adding a task to a queue or selecting it from a queue. The trade-off is that if you have some queues with long-running tasks, a handler may end up waiting to execute from that queue rather than processing another queue which has waiting tasks.

For these values to take effect on a given RegionServer, the RegionServer must be restarted. These parameters are intended for testing purposes and should be used carefully.

ZooKeeper

See ZooKeeper for information on configuring ZooKeeper, and see the part about having a dedicated disk.

Schema Design

Number of Column Families

See On the number of column families.

Key and Attribute Lengths

See Try to minimize row and column sizes. See also However... for compression caveats.

Table RegionSize

The regionsize can be set on a per-table basis via `setMaxFileSize` on TableDescriptorBuilder in the event where certain tables require different regionsizes than the configured default regionsize.

See Determining region count and size for more information.

Bloom Filters

A Bloom filter, named for its creator, Burton Howard Bloom, is a data structure which is designed to predict whether a given element is a member of a set of data. A positive result from a Bloom filter is not always accurate, but a negative result is guaranteed to be accurate. Bloom filters are designed to be "accurate enough" for sets of data which are so large that conventional hashing mechanisms would be impractical. For more information about Bloom filters in general, refer to http://en.wikipedia.org/wiki/Bloom_filter.

In terms of HBase, Bloom filters provide a lightweight in-memory structure to reduce the number of disk reads for a given Get operation (Bloom filters do not work with Scans) to only the StoreFiles likely to contain the desired Row. The potential performance gain increases with the number of parallel reads.

The Bloom filters themselves are stored in the metadata of each HFile and never need to be updated. When an HFile is opened because a region is deployed to a RegionServer, the Bloom filter is loaded into memory.

HBase includes some tuning mechanisms for folding the Bloom filter to reduce the size and keep the false positive rate within a desired range.

Bloom filters were introduced in HBASE-1200. Since HBase 0.96, row-based Bloom filters are enabled by default. (HBASE-8450)

For more information on Bloom filters in relation to HBase, see Bloom Filters for more information, or the following Quora discussion: How are bloom filters used in HBase?.

When To Use Bloom Filters

Since HBase 0.96, row-based Bloom filters are enabled by default. You may choose to disable them or to change some tables to use row+column Bloom filters, depending on the characteristics of your data and how it is loaded into HBase.

To determine whether Bloom filters could have a positive impact, check the value of `blockCacheHitRatio` in the RegionServer metrics. If Bloom filters are enabled, the value of `blockCacheHitRatio` should increase, because the Bloom filter is filtering out blocks that are definitely not needed.

You can choose to enable Bloom filters for a row or for a row+column combination. If you generally scan entire rows, the row+column combination will not provide any benefit. A row-based Bloom filter can operate on a row+column Get, but not the other way around. However, if you have a large number of column-level Puts, such that a row may be present in every StoreFile, a row-based filter will always return a positive result and provide no benefit. Unless you have one column per row, row+column Bloom filters require more space, in order to store more keys. Bloom filters work best when the size of each data entry is at least a few kilobytes in size.

Overhead will be reduced when your data is stored in a few larger StoreFiles, to avoid extra disk IO during low-level scans to find a specific row.

Bloom filters need to be rebuilt upon deletion, so may not be appropriate in environments with a large number of deletions.

Enabling Bloom Filters

Bloom filters are enabled on a Column Family. You can do this by using the setBloomFilterType method of HColumnDescriptor or using the HBase API. Valid values are `NONE`, `ROW` (default), or `ROWCOL`. See When To Use Bloom Filters for more information on `ROW` versus `ROWCOL`. See also the API documentation for ColumnFamilyDescriptorBuilder.

The following example creates a table and enables a ROWCOL Bloom filter on the `colfam1` column family.

```
hbase> create 'mytable',{NAME => 'colfam1', BLOOMFILTER => 'ROWCOL'}
```

Configuring Server-Wide Behavior of Bloom Filters

You can configure the following settings in the *hbase-site.xml*.

Parameter	Default	Description
io.storefile.bloom.enabled	yes	Set to no to kill bloom filters server-wide if something goes wrong
io.storefile.bloom.error.rate	.01	The average false positive rate for bloom filters. Folding is used to maintain the false positive rate. Expressed as a decimal representation of a percentage.
io.storefile.bloom.max.fold	7	The guaranteed maximum fold rate. Changing this setting should not be necessary and is not recommended.
io.storefile.bloom.max.keys	128000000	For default (single-block) Bloom filters, this specifies the maximum number of keys.
io.storefile.delete.family.bloom.enabled	true	Master switch to enable Delete Family Bloom filters and store them in the StoreFile.

Parameter	Default	Description
io.storefile.bloom.block.size	131072	Target Bloom block size. Bloom filter blocks of approximately t his size are interleaved with dat a blocks.
hfile.block.bloom.cacheonwrite	false	Enables cache-on-write for inlin e blocks of a compound Bloom filter.

ColumnFamily BlockSize

The blocksize can be configured for each ColumnFamily in a table, and defaults to 64k. Larger cell values require larger blocksizes. There is an inverse relationship between blocksize and the resulting StoreFile indexes (i.e., if the blocksize is doubled then the resulting indexes should be roughly halved).

See ColumnFamilyDescriptorBuilder and Store for more information.

In-Memory ColumnFamilies

ColumnFamilies can optionally be defined as in-memory. Data is still persisted to disk, just like any other ColumnFamily. In-memory blocks have the highest priority in the Block Cache, but it is not a guarantee that the entire table will be in memory.

See ColumnFamilyDescriptorBuilder for more information.

Compression

Production systems should use compression with their ColumnFamily definitions. See Compression and Data Block Encoding In HBase for more information.

However...

Compression deflates data *on disk*. When it's in-memory (e.g., in the MemStore) or on the wire (e.g., transferring between RegionServer and Client) it's inflated. So while using ColumnFamily compression is a best practice, but it's not going to completely eliminate the impact of over-sized Keys, over-sized ColumnFamily names, or over-sized Column names.

See Try to minimize row and column sizes on for schema design tips, and KeyValue for more information on HBase stores data internally.

HBase General Patterns

Constants

When people get started with HBase they have a tendency to write code that looks like this:

```
Get get = new Get(rowkey);
Result r = table.get(get);
byte[] b = r.getValue(Bytes.toBytes("cf"), Bytes.toBytes("attr"));  // returns cu
rrent version of value
```

But especially when inside loops (and MapReduce jobs), converting the columnFamily and column-names to byte-arrays repeatedly is surprisingly expensive. It's better to use constants for the byte-arrays, like this:

```
public static final byte[] CF = "cf".getBytes();
public static final byte[] ATTR = "attr".getBytes();
...
Get get = new Get(rowkey);
Result r = table.get(get);
byte[] b = r.getValue(CF, ATTR);  // returns current version of value
```

Writing to HBase

Batch Loading

Use the bulk load tool if you can. See Bulk Loading. Otherwise, pay attention to the below.

Table Creation: Pre-Creating Regions

Tables in HBase are initially created with one region by default. For bulk imports, this means that all clients will write to the same region until it is large enough to split and become distributed across the cluster. A useful pattern to speed up the bulk import process is to pre-create empty regions. Be somewhat conservative in this, because too-many regions can actually degrade performance.

There are two different approaches to pre-creating splits using the HBase API. The first approach is to rely on the default `Admin` strategy (which is implemented in `Bytes.split`)...

```
byte[] startKey = ...;      // your lowest key
byte[] endKey = ...;        // your highest key
int numberOfRegions = ...;  // # of regions to create
admin.createTable(table, startKey, endKey, numberOfRegions);
```

And the other approach, using the HBase API, is to define the splits yourself...

```
byte[][] splits = ...;   // create your own splits
admin.createTable(table, splits);
```

You can achieve a similar effect using the HBase Shell to create tables by specifying split options.

```
# create table with specific split points
hbase>create 't1','f1',SPLITS => ['\x10\x00', '\x20\x00', '\x30\x00', '\x40\x00']

# create table with four regions based on random bytes keys
hbase>create 't2','f1', { NUMREGIONS => 4 , SPLITALGO => 'UniformSplit' }

# create table with five regions based on hex keys
create 't3','f1', { NUMREGIONS => 5, SPLITALGO => 'HexStringSplit' }
```

See Relationship Between RowKeys and Region Splits for issues related to understanding your keyspace and pre-creating regions. See manual region splitting decisions for discussion on manually pre-splitting regions. See Pre-splitting tables with the HBase Shell for more details of using the HBase Shell to pre-split tables.

Table Creation: Deferred Log Flush

The default behavior for Puts using the Write Ahead Log (WAL) is that `WAL` edits will be written immediately. If deferred log flush is used, WAL edits are kept in memory until the flush period. The benefit is aggregated and asynchronous `WAL`- writes, but the potential downside is that if the RegionServer goes down the yet-to-be-flushed edits are lost. This is safer, however, than not using WAL at all with Puts.

Deferred log flush can be configured on tables via TableDescriptorBuilder. The default value of `hbase.regionserver.optionallogflushinterval` is 1000ms.

HBase Client: Turn off WAL on Puts

A frequent request is to disable the WAL to increase performance of Puts. This is only appropriate for bulk loads, as it puts your data at risk by removing the protection of the WAL in the event of a region server crash. Bulk loads can be re-run in the event of a crash, with little risk of data loss.

> ⚠ If you disable the WAL for anything other than bulk loads, your data is at risk.

In general, it is best to use WAL for Puts, and where loading throughput is a concern to use bulk loading techniques instead. For normal Puts, you are not likely to see a performance improvement which would outweigh the risk. To disable the WAL, see Disabling the WAL.

HBase Client: Group Puts by RegionServer

In addition to using the writeBuffer, grouping `Put`s by RegionServer can reduce the number of client RPC calls per writeBuffer flush. There is a utility `HTableUtil` currently on MASTER that does this, but you can either copy that or implement your own version for those still on 0.90.x or earlier.

MapReduce: Skip The Reducer

When writing a lot of data to an HBase table from a MR job (e.g., with TableOutputFormat), and specifically where Puts are being emitted from the Mapper, skip the Reducer step. When a Reducer step is used, all of the output (Puts) from the Mapper will get spooled to disk, then sorted/shuffled to other Reducers that will most likely be off-node. It's far more efficient to just write directly to HBase.

For summary jobs where HBase is used as a source and a sink, then writes will be coming from the Reducer step (e.g., summarize values then write out result). This is a different processing problem than from the above case.

Anti-Pattern: One Hot Region

If all your data is being written to one region at a time, then re-read the section on processing timeseries data.

Also, if you are pre-splitting regions and all your data is *still* winding up in a single region even though your keys aren't monotonically increasing, confirm that your keyspace

actually works with the split strategy. There are a variety of reasons that regions may appear "well split" but won't work with your data. As the HBase client communicates directly with the RegionServers, this can be obtained via RegionLocator.getRegionLocation.

See Table Creation: Pre-Creating Regions, as well as HBase Configurations

Reading from HBase

The mailing list can help if you are having performance issues.

Scan Caching

If HBase is used as an input source for a MapReduce job, for example, make sure that the input Scan instance to the MapReduce job has `setCaching` set to something greater than the default (which is 1). Using the default value means that the map-task will make call back to the region-server for every record processed. Setting this value to 500, for example, will transfer 500 rows at a time to the client to be processed. There is a cost/benefit to have the cache value be large because it costs more in memory for both client and RegionServer, so bigger isn't always better.

Scan Caching in MapReduce Jobs

Scan settings in MapReduce jobs deserve special attention. Timeouts can result (e.g., UnknownScannerException) in Map tasks if it takes longer to process a batch of records before the client goes back to the RegionServer for the next set of data. This problem can occur because there is non-trivial processing occurring per row. If you process rows quickly, set caching higher. If you process rows more slowly (e.g., lots of transformations per row, writes), then set caching lower.

Timeouts can also happen in a non-MapReduce use case (i.e., single threaded HBase client doing a Scan), but the processing that is often performed in MapReduce jobs tends to exacerbate this issue.

Scan Attribute Selection

Whenever a Scan is used to process large numbers of rows (and especially when used as a MapReduce source), be aware of which attributes are selected. If `scan.addFamily` is

called then *all* of the attributes in the specified ColumnFamily will be returned to the client. If only a small number of the available attributes are to be processed, then only those attributes should be specified in the input scan because attribute over-selection is a non-trivial performance penalty over large datasets.

Avoid scan seeks

When columns are selected explicitly with `scan.addColumn`, HBase will schedule seek operations to seek between the selected columns. When rows have few columns and each column has only a few versions this can be inefficient. A seek operation is generally slower if does not seek at least past 5-10 columns/versions or 512-1024 bytes.

In order to opportunistically look ahead a few columns/versions to see if the next column/version can be found that way before a seek operation is scheduled, a new attribute `Scan.HINT_LOOKAHEAD` can be set on the Scan object. The following code instructs the RegionServer to attempt two iterations of next before a seek is scheduled:

```
Scan scan = new Scan();
scan.addColumn(...);
scan.setAttribute(Scan.HINT_LOOKAHEAD, Bytes.toBytes(2));
table.getScanner(scan);
```

MapReduce - Input Splits

For MapReduce jobs that use HBase tables as a source, if there a pattern where the "slow" map tasks seem to have the same Input Split (i.e., the RegionServer serving the data), see the Troubleshooting Case Study in Case Study #1 (Performance Issue On A Single Node).

Close ResultScanners

This isn't so much about improving performance but rather *avoiding* performance problems. If you forget to close ResultScanners you can cause problems on the RegionServers. Always have ResultScanner processing enclosed in try/catch blocks.

```
Scan scan = new Scan();
// set attrs...
ResultScanner rs = table.getScanner(scan);
try {
  for (Result r = rs.next(); r != null; r = rs.next()) {
    // process result...
  }
```

```
} finally {
  rs.close();  // always close the ResultScanner!
}
table.close();
```

Block Cache

Scan instances can be set to use the block cache in the RegionServer via the `setCacheBlocks` method. For input Scans to MapReduce jobs, this should be `false`. For frequently accessed rows, it is advisable to use the block cache.

Cache more data by moving your Block Cache off-heap. See Off-heap Block Cache

Optimal Loading of Row Keys

When performing a table scan where only the row keys are needed (no families, qualifiers, values or timestamps), add a FilterList with a `MUST_PASS_ALL` operator to the scanner using `setFilter`. The filter list should include both a FirstKeyOnlyFilter and a KeyOnlyFilter. Using this filter combination will result in a worst case scenario of a RegionServer reading a single value from disk and minimal network traffic to the client for a single row.

Concurrency: Monitor Data Spread

When performing a high number of concurrent reads, monitor the data spread of the target tables. If the target table(s) have too few regions then the reads could likely be served from too few nodes.

See Table Creation: Pre-Creating Regions, as well as HBase Configurations

Bloom Filters

Enabling Bloom Filters can save your having to go to disk and can help improve read latencies.

Bloom filters were developed over in HBase-1200 Add bloomfilters. For description of the development process — why static blooms rather than dynamic — and for an overview of the unique properties that pertain to blooms in HBase, as well as possible future directions, see the *Development Process* section of the document BloomFilters in HBase attached to HBASE-1200. The bloom filters described here are actually version two of blooms in

HBase. In versions up to 0.19.x, HBase had a dynamic bloom option based on work done by the European Commission One-Lab Project 034819. The core of the HBase bloom work was later pulled up into Hadoop to implement org.apache.hadoop.io.BloomMapFile. Version 1 of HBase blooms never worked that well. Version 2 is a rewrite from scratch though again it starts with the one-lab work.

See also Bloom Filters.

Bloom StoreFile footprint

Bloom filters add an entry to the `StoreFile` general `FileInfo` data structure and then two extra entries to the `StoreFile` metadata section.

BloomFilter in the `StoreFile` `FileInfo` data structure

`FileInfo` has a `BLOOM_FILTER_TYPE` entry which is set to `NONE`, `ROW` or `ROWCOL.`

BloomFilter entries in `StoreFile` metadata

`BLOOM_FILTER_META` holds Bloom Size, Hash Function used, etc. It's small in size and is cached on `StoreFile.Reader` load.
`BLOOM_FILTER_DATA` is the actual bloomfilter data. Obtained on-demand. Stored in the LRU cache, if it is enabled (It's enabled by default).

Bloom Filter Configuration

`io.storefile.bloom.enabled` global kill switch

`io.storefile.bloom.enabled` in `Configuration` serves as the kill switch in case something goes wrong. Default = `true`.

`io.storefile.bloom.error.rate`

`io.storefile.bloom.error.rate` = average false positive rate. Default = 1%. Decrease rate by ½ (e.g. to .5%) == +1 bit per bloom entry.

`io.storefile.bloom.max.fold`

`io.storefile.bloom.max.fold` = guaranteed minimum fold rate. Most people should leave this alone. Default = 7, or can collapse to at least 1/128th of original size. See the *Development Process* section of the document BloomFilters in HBase for more on what this option means.

Hedged Reads

Hedged reads are a feature of HDFS, introduced in Hadoop 2.4.0 with HDFS-5776. Normally, a single thread is spawned for each read request. However, if hedged reads are enabled, the client waits some configurable amount of time, and if the read does not return, the client spawns a second read request, against a different block replica of the same data. Whichever read returns first is used, and the other read request is discarded.

Hedged reads are "...very good at eliminating outlier datanodes, which in turn makes them very good choice for latency sensitive setups. But, if you are looking for maximizing throughput, hedged reads tend to create load amplification as things get slower in general. In short, the thing to watch out for is the non-graceful performance degradation when you are running close a certain throughput threshold." (Quote from Ashu Pachauri in HBASE-17083).

Other concerns to keep in mind while running with hedged reads enabled include:

- They may lead to network congestion. See HBASE-17083
- Make sure you set the thread pool large enough so as blocking on the pool does not become a bottleneck (Again see HBASE-17083)

(From Yu Li up in HBASE-17083)

Because an HBase RegionServer is a HDFS client, you can enable hedged reads in HBase, by adding the following properties to the RegionServer's hbase-site.xml and tuning the values to suit your environment.

Configuration for Hedged Reads

- `dfs.client.hedged.read.threadpool.size` - the number of threads dedicated to servicing hedged reads. If this is set to 0 (the default), hedged reads are disabled.
- `dfs.client.hedged.read.threshold.millis` - the number of milliseconds to wait before spawning a second read thread.

Hedged Reads Configuration Example

```
<property>
  <name>dfs.client.hedged.read.threadpool.size</name>
  <value>20</value>  <!-- 20 threads -->
</property>
```

```
<property>
  <name>dfs.client.hedged.read.threshold.millis</name>
  <value>10</value>  <!-- 10 milliseconds -->
</property>
```

Use the following metrics to tune the settings for hedged reads on your cluster. See HBase Metrics for more information.

Metrics for Hedged Reads

- hedgedReadOps - the number of times hedged read threads have been triggered. This could indicate that read requests are often slow, or that hedged reads are triggered too quickly.
- hedgeReadOpsWin - the number of times the hedged read thread was faster than the original thread. This could indicate that a given RegionServer is having trouble servicing requests.
- hedgedReadOpsInCurThread - the number of times hedged read was rejected from executor and needed to fallback to be executed in current thread. This could indicate that current hedged read thread pool size is not appropriate.

Deleting from HBase

Using HBase Tables as Queues

HBase tables are sometimes used as queues. In this case, special care must be taken to regularly perform major compactions on tables used in this manner. As is documented in Data Model, marking rows as deleted creates additional StoreFiles which then need to be processed on reads. Tombstones only get cleaned up with major compactions.

See also Compaction and Admin.majorCompact.

Delete RPC Behavior

Be aware that `Table.delete(Delete)` doesn't use the writeBuffer. It will execute an RegionServer RPC with each invocation. For a large number of deletes, consider `Table.delete(List)`.

See hbase.client.Delete

HDFS

Because HBase runs on HDFS it is important to understand how it works and how it affects HBase.

Current Issues With Low-Latency Reads

The original use-case for HDFS was batch processing. As such, there low-latency reads were historically not a priority. With the increased adoption of Apache HBase this is changing, and several improvements are already in development. See the Umbrella Jira Ticket for HDFS Improvements for HBase.

Leveraging local data

Since Hadoop 1.0.0 (also 0.22.1, 0.23.1, CDH3u3 and HDP 1.0) via HDFS-2246, it is possible for the DFSClient to take a "short circuit" and read directly from the disk instead of going through the DataNode when the data is local. What this means for HBase is that the RegionServers can read directly off their machine's disks instead of having to open a socket to talk to the DataNode, the former being generally much faster. See JD's Performance Talk. Also see HBase, mail # dev - read short circuit thread for more discussion around short circuit reads.

To enable "short circuit" reads, it will depend on your version of Hadoop. The original shortcircuit read patch was much improved upon in Hadoop 2 in HDFS-347. See http://blog.cloudera.com/blog/2013/08/how-improved-short-circuit-local-reads-bring-better-performance-and-security-to-hadoop/ for details on the difference between the old and new implementations. See Hadoop shortcircuit reads configuration page for how to enable the latter, better version of shortcircuit. For example, here is a minimal config. enabling short-circuit reads added to *hbase-site.xml*:

```
<property>
  <name>dfs.client.read.shortcircuit</name>
  <value>true</value>
  <description>
    This configuration parameter turns on short-circuit local reads.
  </description>
</property>
<property>
  <name>dfs.domain.socket.path</name>
  <value>/home/stack/sockets/short_circuit_read_socket_PORT</value>
  <description>
    Optional.  This is a path to a UNIX domain socket that will be used for
```

```
    communication between the DataNode and local HDFS clients.
    If the string "_PORT" is present in this path, it will be replaced by the
    TCP port of the DataNode.
  </description>
</property>
```

Be careful about permissions for the directory that hosts the shared domain socket; dfsclient will complain if open to other than the hbase user.

If you are running on an old Hadoop, one that is without HDFS-347 but that has HDFS-2246, you must set two configurations. First, the hdfs-site.xml needs to be amended. Set the property `dfs.block.local-path-access.user` to be the *only* user that can use the shortcut. This has to be the user that started HBase. Then in hbase-site.xml, set `dfs.clien t.read.shortcircuit` to be `true`

Services — at least the HBase RegionServers — will need to be restarted in order to pick up the new configurations.

> **dfs.client.read.shortcircuit.buffer.size**
>
> The default for this value is too high when running on a highly trafficked HBase. In HBase, if this value has not been set, we set it down from the default of 1M to 128k (Since HBase 0.98.0 and 0.96.1). See HBASE-8143 HBase on Hadoop 2 with local short circuit reads (ssr) causes OOM. The Hadoop DFSClient in HBase will allocate a direct byte buffer of this size for *each* block it has open; given HBase keeps its HDFS files open all the time, this can add up quickly.

Performance Comparisons of HBase vs. HDFS

A fairly common question on the dist-list is why HBase isn't as performant as HDFS files in a batch context (e.g., as a MapReduce source or sink). The short answer is that HBase is doing a lot more than HDFS (e.g., reading the KeyValues, returning the most current row or specified timestamps, etc.), and as such HBase is 4-5 times slower than HDFS in this processing context. There is room for improvement and this gap will, over time, be reduced, but HDFS will always be faster in this use-case.

Amazon EC2

Performance questions are common on Amazon EC2 environments because it is a shared environment. You will not see the same throughput as a dedicated server. In terms of

running tests on EC2, run them several times for the same reason (i.e., it's a shared environment and you don't know what else is happening on the server).

If you are running on EC2 and post performance questions on the dist-list, please state this fact up-front that because EC2 issues are practically a separate class of performance issues.

Collocating HBase and MapReduce

It is often recommended to have different clusters for HBase and MapReduce. A better qualification of this is: don't collocate an HBase that serves live requests with a heavy MR workload. OLTP and OLAP-optimized systems have conflicting requirements and one will lose to the other, usually the former. For example, short latency-sensitive disk reads will have to wait in line behind longer reads that are trying to squeeze out as much throughput as possible. MR jobs that write to HBase will also generate flushes and compactions, which will in turn invalidate blocks in the Block Cache.

If you need to process the data from your live HBase cluster in MR, you can ship the deltas with CopyTable or use replication to get the new data in real time on the OLAP cluster. In the worst case, if you really need to collocate both, set MR to use less Map and Reduce slots than you'd normally configure, possibly just one.

When HBase is used for OLAP operations, it's preferable to set it up in a hardened way like configuring the ZooKeeper session timeout higher and giving more memory to the MemStores (the argument being that the Block Cache won't be used much since the workloads are usually long scans).

Case Studies

For Performance and Troubleshooting Case Studies, see Apache HBase Case Studies.

Profiler Servlet

Background

HBASE-21926 introduced a new servlet that supports integrated, on-demand profiling via the Async Profiler project.

Prerequisites

Go to the Async Profiler Home Page, download a release appropriate for your platform, and install on every cluster host. If running a Linux kernel v4.6 or later, be sure to set proc variables as per the Basic Usage section. Not doing so will result in flame graphs that contain no content.

Set `ASYNC_PROFILER_HOME` in the environment (put it in hbase-env.sh) to the root directory of the async-profiler install location, or pass it on the HBase daemon's command line as a system property as `-Dasync.profiler.home=/path/to/async-profiler`.

Usage

Once the prerequisites are satisfied, access to async-profiler is available by way of the HBase UI or direct interaction with the infoserver.

Examples:

- To collect 30 second CPU profile of current process (returns FlameGraph svg) `curl http://localhost:16030/prof`
- To collect 1 minute CPU profile of current process and output in tree format (html) `curl http://localhost:16030/prof?output=tree&duration=60`
- To collect 30 second heap allocation profile of current process (returns FlameGraph svg) `curl http://localhost:16030/prof?event=alloc`
- To collect lock contention profile of current process (returns FlameGraph svg) `curl http://localhost:16030/prof?event=lock`

The following event types are supported by async-profiler. Use the 'event' parameter to specify. Default is 'cpu'. Not all operating systems will support all types.

Perf events:

- cpu
- page-faults
- context-switches
- cycles
- instructions
- cache-references
- cache-misses
- branches
- branch-misses
- bus-cycles
- L1-dcache-load-misses
- LLC-load-misses
- dTLB-load-misses

Java events:

- alloc
- lock

The following output formats are supported. Use the 'output' parameter to specify. Default is 'flamegraph'.

Output formats:

- summary: A dump of basic profiling statistics.
- traces: Call traces.
- flat: Flat profile (top N hot methods).
- collapsed: Collapsed call traces in the format used by FlameGraph script. This is a collection of call stacks, where each line is a semicolon separated list of frames

followed by a counter.

- svg: FlameGraph in SVG format.
- tree: Call tree in HTML format.
- jfr: Call traces in Java Flight Recorder format.

The 'duration' parameter specifies how long to collect trace data before generating output, specified in seconds. The default is 10 seconds.

UI

In the UI, there is a new entry 'Profiler' in the top menu that will run the default action, which is to profile the CPU usage of the local process for thirty seconds and then produce FlameGraph SVG output.

Notes

The query parameter `pid` can be used to specify the process id of a specific process to be profiled. If this parameter is missing the local process in which the infoserver is embedded will be profiled. Profile targets that are not JVMs might work but is not specifically supported. There are security implications. Access to the infoserver should be appropriately restricted.

Troubleshooting and Debugging Apache HBase

General Guidelines

Always start with the master log (TODO: Which lines?). Normally it's just printing the same lines over and over again. If not, then there's an issue. Google should return some hits for those exceptions you're seeing.

An error rarely comes alone in Apache HBase, usually when something gets screwed up what will follow may be hundreds of exceptions and stack traces coming from all over the place. The best way to approach this type of problem is to walk the log up to where it all began, for example one trick with RegionServers is that they will print some metrics when aborting so grepping for *Dump* should get you around the start of the problem.

RegionServer suicides are 'normal', as this is what they do when something goes wrong. For example, if ulimit and max transfer threads (the two most important initial settings, see [ulimit] and `dfs.datanode.max.transfer.threads`) aren't changed, it will make it impossible at some point for DataNodes to create new threads that from the HBase point of view is seen as if HDFS was gone. Think about what would happen if your MySQL database was suddenly unable to access files on your local file system, well it's the same with HBase and HDFS. Another very common reason to see RegionServers committing seppuku is when they enter prolonged garbage collection pauses that last longer than the default ZooKeeper session timeout. For more information on GC pauses, see the 3 part blog post by Todd Lipcon and Long GC pauses above.

Logs

The key process logs are as follows... (replace `<user>` with the user that started the service, and `<hostname>` for the machine name)

NameNode: *$HADOOP_HOME/logs/hadoop-<user>-namenode-<hostname>.log*

DataNode: *$HADOOP_HOME/logs/hadoop-<user>-datanode-<hostname>.log*

JobTracker: *$HADOOP_HOME/logs/hadoop-<user>-jobtracker-<hostname>.log*

TaskTracker: *$HADOOP_HOME/logs/hadoop-<user>-tasktracker-<hostname>.log*

HMaster: *$HBASE_HOME/logs/hbase-<user>-master-<hostname>.log*

RegionServer: *$HBASE_HOME/logs/hbase-<user>-regionserver-<hostname>.log*

ZooKeeper: *TODO*

Log Locations

For stand-alone deployments the logs are obviously going to be on a single machine, however this is a development configuration only. Production deployments need to run on a cluster.

NameNode

The NameNode log is on the NameNode server. The HBase Master is typically run on the NameNode server, and well as ZooKeeper.

For smaller clusters the JobTracker/ResourceManager is typically run on the NameNode server as well.

DataNode

Each DataNode server will have a DataNode log for HDFS, as well as a RegionServer log for HBase.

Additionally, each DataNode server will also have a TaskTracker/NodeManager log for MapReduce task execution.

Log Levels

Enabling RPC-level logging

Enabling the RPC-level logging on a RegionServer can often give insight on timings at the server. Once enabled, the amount of log spewed is voluminous. It is not recommended that you leave this logging on for more than short bursts of time. To enable RPC-level logging,

browse to the RegionServer UI and click on *Log Level*. Set the log level to `TRACE` for the package `org.apache.hadoop.hbase.ipc`, then tail the RegionServers log. Analyze.

To disable, set the logging level back to `INFO` level.

The same log settings also work on Master and for the client.

Handling Multiple SLF4J Bindings to Prevent Overridden Log Configurations

If you have a local installation of Hadoop (e.g., installed via Homebrew on macOS), you may encounter a "multiple SLF4J bindings" warning when starting HBase in standalone mode using `bin/start-hbase.sh`. For example, you may see the following when starting HBase on your machine:

```
SLF4J: Class path contains multiple SLF4J bindings.
SLF4J: Found binding in [jar:file:/opt/homebrew/Cellar/hadoop/3.4.1/libexec/shar
e/hadoop/common/lib/slf4j-reload4j-1.7.36.jar!/org/slf4j/impl/StaticLoggerBinder.
class]
SLF4J: Found binding in [jar:file:$HOME/.m2/repository/org/apache/logging/log4j/l
og4j-slf4j-impl/2.25.3/log4j-slf4j-impl-2.25.3.jar!/org/slf4j/impl/StaticLoggerBi
nder.class]
SLF4J: See http://www.slf4j.org/codes.html#multiple_bindings for an explanation.
SLF4J: Actual binding is of type [org.slf4j.impl.Reload4jLoggerFactory]
```

This is happening because the `bin/hbase` script automatically detects local Hadoop installations and adds their classpath to HBase. If your Hadoop installation uses the legacy Reload4j (or Log4j 1.2) binder, SLF4J may prioritize it over the HBase Log4j2 binder. When this happens, your `conf/log4j2.properties` file and `HBASE_ROOT_LOGGER` environment variable are completely ignored.

To get around this issue, you can set `HBASE_DISABLE_HADOOP_CLASSPATH_LOOKUP` to `true`. You can modify this directly in `conf/hbase-env.sh` by uncommenting the following line:

```
export HBASE_DISABLE_HADOOP_CLASSPATH_LOOKUP="true"
```

Note: Setting `HBASE_DISABLE_HADOOP_CLASSPATH_LOOKUP` does not fix the issue when HBase is in distributed mode. There may be a different warning instead.

JVM Garbage Collection Logs

> All example Garbage Collection logs in this section are based on Java 8 output. The introduction of Unified Logging in Java 9 and newer will result in very different looking logs.

HBase is memory intensive, and using the default GC you can see long pauses in all threads including the *Juliet Pause* aka "GC of Death". To help debug this or confirm this is happening GC logging can be turned on in the Java virtual machine.

To enable, in *hbase-env.sh*, uncomment one of the below lines :

```
# This enables basic gc logging to the .out file.
# export SERVER_GC_OPTS="-verbose:gc -XX:+PrintGCDetails -XX:+PrintGCDateStamps"

# This enables basic gc logging to its own file.
# export SERVER_GC_OPTS="-verbose:gc -XX:+PrintGCDetails -XX:+PrintGCDateStamps -
Xloggc:<FILE-PATH>"

# This enables basic GC logging to its own file with automatic log rolling. Only
applies to jdk 1.6.0_34+ and 1.7.0_2+.
# export SERVER_GC_OPTS="-verbose:gc -XX:+PrintGCDetails -XX:+PrintGCDateStamps -
Xloggc:<FILE-PATH> -XX:+UseGCLogFileRotation -XX:NumberOfGCLogFiles=1 -XX:GCLogFi
leSize=512M"

# If <FILE-PATH> is not replaced, the log file(.gc) would be generated in the HBA
SE_LOG_DIR.
```

At this point you should see logs like so:

```
64898.952: [GC [1 CMS-initial-mark: 2811538K(3055704K)] 2812179K(3061272K), 0.000
7360 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
64898.953: [CMS-concurrent-mark-start]
64898.971: [GC 64898.971: [ParNew: 5567K->576K(5568K), 0.0101110 secs] 2817105K->
2812715K(3061272K), 0.0102200 secs] [Times: user=0.07 sys=0.00, real=0.01 secs]
```

In this section, the first line indicates a 0.0007360 second pause for the CMS to initially mark. This pauses the entire VM, all threads for that period of time.

The third line indicates a "minor GC", which pauses the VM for 0.0101110 seconds - aka 10 milliseconds. It has reduced the "ParNew" from about 5.5m to 576k. Later on in this cycle we see:

```
64901.445: [CMS-concurrent-mark: 1.542/2.492 secs] [Times: user=10.49 sys=0.33, r
eal=2.49 secs]
```

```
64901.445: [CMS-concurrent-preclean-start]
64901.453: [GC 64901.453: [ParNew: 5505K->573K(5568K), 0.0062440 secs] 2868746K->
2864292K(3061272K), 0.0063360 secs] [Times: user=0.05 sys=0.00, real=0.01 secs]
64901.476: [GC 64901.476: [ParNew: 5563K->575K(5568K), 0.0072510 secs] 2869283K->
2864837K(3061272K), 0.0073320 secs] [Times: user=0.05 sys=0.01, real=0.01 secs]
64901.500: [GC 64901.500: [ParNew: 5517K->573K(5568K), 0.0120390 secs] 2869780K->
2865267K(3061272K), 0.0121150 secs] [Times: user=0.09 sys=0.00, real=0.01 secs]
64901.529: [GC 64901.529: [ParNew: 5507K->569K(5568K), 0.0086240 secs] 2870200K->
2865742K(3061272K), 0.0087180 secs] [Times: user=0.05 sys=0.00, real=0.01 secs]
64901.554: [GC 64901.555: [ParNew: 5516K->575K(5568K), 0.0107130 secs] 2870689K->
2866291K(3061272K), 0.0107820 secs] [Times: user=0.06 sys=0.00, real=0.01 secs]
64901.578: [CMS-concurrent-preclean: 0.070/0.133 secs] [Times: user=0.48 sys=0.0
1, real=0.14 secs]
64901.578: [CMS-concurrent-abortable-preclean-start]
64901.584: [GC 64901.584: [ParNew: 5504K->571K(5568K), 0.0087270 secs] 2871220K->
2866830K(3061272K), 0.0088220 secs] [Times: user=0.05 sys=0.00, real=0.01 secs]
64901.609: [GC 64901.609: [ParNew: 5512K->569K(5568K), 0.0063370 secs] 2871771K->
2867322K(3061272K), 0.0064230 secs] [Times: user=0.06 sys=0.00, real=0.01 secs]
64901.615: [CMS-concurrent-abortable-preclean: 0.007/0.037 secs] [Times: user=0.1
3 sys=0.00, real=0.03 secs]
64901.616: [GC[YG occupancy: 645 K (5568 K)]64901.616: [Rescan (parallel) , 0.002
0210 secs]64901.618: [weak refs processing, 0.0027950 secs] [1 CMS-remark: 286675
3K(3055704K)] 2867399K(3061272K), 0.0049380 secs] [Times: user=0.00 sys=0.01, rea
l=0.01 secs]
64901.621: [CMS-concurrent-sweep-start]
```

The first line indicates that the CMS concurrent mark (finding garbage) has taken 2.4 seconds. But this is a *concurrent* 2.4 seconds, Java has not been paused at any point in time.

There are a few more minor GCs, then there is a pause at the 2nd last line:

```
64901.616: [GC[YG occupancy: 645 K (5568 K)]64901.616: [Rescan (parallel) , 0.002
0210 secs]64901.618: [weak refs processing, 0.0027950 secs] [1 CMS-remark: 286675
3K(3055704K)] 2867399K(3061272K), 0.0049380 secs] [Times: user=0.00 sys=0.01, rea
l=0.01 secs]
```

The pause here is 0.0049380 seconds (aka 4.9 milliseconds) to 'remark' the heap.

At this point the sweep starts, and you can watch the heap size go down:

```
64901.637: [GC 64901.637: [ParNew: 5501K->569K(5568K), 0.0097350 secs] 2871958K->
2867441K(3061272K), 0.0098370 secs] [Times: user=0.05 sys=0.00, real=0.01 secs]
...  lines removed ...
64904.936: [GC 64904.936: [ParNew: 5532K->568K(5568K), 0.0070720 secs] 1365024K->
1360689K(3061272K), 0.0071930 secs] [Times: user=0.05 sys=0.00, real=0.01 secs]
64904.953: [CMS-concurrent-sweep: 2.030/3.332 secs] [Times: user=9.57 sys=0.26, r
eal=3.33 secs]
```

At this point, the CMS sweep took 3.332 seconds, and heap went from about ~ 2.8 GB to 1.3 GB (approximate).

The key points here is to keep all these pauses low. CMS pauses are always low, but if your ParNew starts growing, you can see minor GC pauses approach 100ms, exceed 100ms and hit as high at 400ms.

This can be due to the size of the ParNew, which should be relatively small. If your ParNew is very large after running HBase for a while, in one example a ParNew was about 150MB, then you might have to constrain the size of ParNew (The larger it is, the longer the collections take but if it's too small, objects are promoted to old gen too quickly). In the below we constrain new gen size to 64m.

Add the below line in *hbase-env.sh*:

```
export SERVER_GC_OPTS="$SERVER_GC_OPTS -XX:NewSize=64m -XX:MaxNewSize=64m"
```

Similarly, to enable GC logging for client processes, uncomment one of the below lines in *hbase-env.sh*:

```
# This enables basic gc logging to the .out file.
# export CLIENT_GC_OPTS="-verbose:gc -XX:+PrintGCDetails -XX:+PrintGCDateStamps"

# This enables basic gc logging to its own file.
# export CLIENT_GC_OPTS="-verbose:gc -XX:+PrintGCDetails -XX:+PrintGCDateStamps -
Xloggc:<FILE-PATH>"

# This enables basic GC logging to its own file with automatic log rolling. Only
applies to jdk 1.6.0_34+ and 1.7.0_2+.
# export CLIENT_GC_OPTS="-verbose:gc -XX:+PrintGCDetails -XX:+PrintGCDateStamps -
Xloggc:<FILE-PATH> -XX:+UseGCLogFileRotation -XX:NumberOfGCLogFiles=1 -XX:GCLogFi
leSize=512M"

# If <FILE-PATH> is not replaced, the log file(.gc) would be generated in the HBA
SE_LOG_DIR .
```

For more information on GC pauses, see the 3 part blog post by Todd Lipcon and Long GC pauses above.

Resources

Mailing Lists

Ask a question on the Apache HBase mailing lists. The 'dev' mailing list is aimed at the community of developers actually building Apache HBase and for features currently under development, and 'user' is generally used for questions on released versions of Apache HBase. Before going to the mailing list, make sure your question has not already been answered by searching the mailing list archives first. For those who prefer to communicate in Chinese, they can use the 'user-zh' mailing list instead of the 'user' list. Take some time crafting your question. See Getting Answers for ideas on crafting good questions. A quality question that includes all context and exhibits evidence the author has tried to find answers in the manual and out on lists is more likely to get a prompt response.

Slack

#hbase on https://the-asf.slack.com/

IRC

(You will probably get a more prompt response on the Slack channel)

#hbase on irc.freenode.net

JIRA

JIRA is also really helpful when looking for Hadoop/HBase-specific issues.

Tools

Builtin Tools

Master Web Interface

The Master starts a web-interface on port 16010 by default.

The Master web UI lists created tables and their definition (e.g., ColumnFamilies, blocksize, etc.). Additionally, the available RegionServers in the cluster are listed along

with selected high-level metrics (requests, number of regions, usedHeap, maxHeap). The Master web UI allows navigation to each RegionServer's web UI.

RegionServer Web Interface

RegionServers starts a web-interface on port 16030 by default.

The RegionServer web UI lists online regions and their start/end keys, as well as point-in-time RegionServer metrics (requests, regions, storeFileIndexSize, compactionQueueSize, etc.).

See HBase Metrics for more information in metric definitions.

zkcli

`zkcli` is a very useful tool for investigating ZooKeeper-related issues. To invoke:

```
./hbase zkcli -server host:port <cmd> <args>
```

The commands (and arguments) are:

```
  connect host:port
  get path [watch]
  ls path [watch]
  set path data [version]
  delquota [-n|-b] path
  quit
  printwatches on|off
  create [-s] [-e] path data acl
  stat path [watch]
  close
  ls2 path [watch]
  history
  listquota path
  setAcl path acl
  getAcl path
  sync path
  redo cmdno
  addauth scheme auth
  delete path [version]
  setquota -n|-b val path
```

Maintenance Mode

If the cluster has gotten stuck in some state and the standard techniques aren't making progress, it is possible to restart the cluster in "maintenance mode." This mode features

drastically reduced capabilities and surface area, making it easier to enact very low-level changes such as repairing/recovering the `hbase:meta` table.

To enter maintenance mode, set `hbase.master.maintenance_mode` to `true` either in your `hbase-site.xml` or via system propery when starting the master process (`-D...=true`). Entering and exiting this mode requires a service restart, however the typical use will be when HBase Master is already facing startup difficulties.

When maintenance mode is enabled, the master will host all system tables - ensure that it has enough memory to do so. RegionServers will not be assigned any regions from user-space tables; in fact, they will go completely unused while in maintenance mode. Additionally, the master will not load any coprocessors, will not run any normalization or merge/split operations, and will not enforce quotas.

External Tools

tail

`tail` is the command line tool that lets you look at the end of a file. Add the `-f` option and it will refresh when new data is available. It's useful when you are wondering what's happening, for example, when a cluster is taking a long time to shutdown or startup as you can just fire a new terminal and tail the master log (and maybe a few RegionServers).

top

`top` is probably one of the most important tools when first trying to see what's running on a machine and how the resources are consumed. Here's an example from production system:

```
top - 14:46:59 up 39 days, 11:55,  1 user,  load average: 3.75, 3.57, 3.84
Tasks: 309 total,   1 running, 308 sleeping,   0 stopped,   0 zombie
Cpu(s):  4.5%us,  1.6%sy,  0.0%ni, 91.7%id,  1.4%wa,  0.1%hi,  0.6%si,  0.0%st
Mem:  24414432k total, 24296956k used,   117476k free,     7196k buffers
Swap: 16008732k total,  14348k used, 15994384k free, 11106908k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM  TIME+  COMMAND
15558 hadoop    18  -2 3292m 2.4g 3556 S   79 10.4   6523:52 java
13268 hadoop    18  -2 8967m 8.2g 4104 S   21 35.1   5170:30 java
 8895 hadoop    18  -2 1581m 497m 3420 S   11  2.1   4002:32 java
...
```

Here we can see that the system load average during the last five minutes is 3.75, which very roughly means that on average 3.75 threads were waiting for CPU time during these 5 minutes. In general, the *perfect* utilization equals to the number of cores, under that number the machine is under utilized and over that the machine is over utilized. This is an important concept, see this article to understand it more: http://www.linuxjournal.com/article/9001.

Apart from load, we can see that the system is using almost all its available RAM but most of it is used for the OS cache (which is good). The swap only has a few KBs in it and this is wanted, high numbers would indicate swapping activity which is the nemesis of performance of Java systems. Another way to detect swapping is when the load average goes through the roof (although this could also be caused by things like a dying disk, among others).

The list of processes isn't super useful by default, all we know is that 3 java processes are using about 111% of the CPUs. To know which is which, simply type `c` and each line will be expanded. Typing `1` will give you the detail of how each CPU is used instead of the average for all of them like shown here.

jps

`jps` is shipped with every JDK and gives the java process ids for the current user (if root, then it gives the ids for all users). Example:

```
hadoop@sv4borg12:~$ jps
1322 TaskTracker
17789 HRegionServer
27862 Child
1158 DataNode
25115 HQuorumPeer
2950 Jps
19750 ThriftServer
18776 jmx
```

In order, we see a:

- Hadoop TaskTracker, manages the local Childs
- HBase RegionServer, serves regions
- Child, its MapReduce task, cannot tell which type exactly
- Hadoop TaskTracker, manages the local Childs

- Hadoop DataNode, serves blocks
- HQuorumPeer, a ZooKeeper ensemble member
- Jps, well... it's the current process
- ThriftServer, it's a special one will be running only if thrift was started
- jmx, this is a local process that's part of our monitoring platform (poorly named maybe). You probably don't have that.

You can then do stuff like checking out the full command line that started the process:

```
hadoop@sv4borg12:~$ ps aux | grep HRegionServer
hadoop   17789  155 35.2 9067824 8604364 ?     S<l  Mar04 9855:48 /usr/java/jdk1.
6.0_14/bin/java -Xmx8000m -XX:+DoEscapeAnalysis -XX:+AggressiveOpts -XX:+UseConcM
arkSweepGC -XX:NewSize=64m -XX:MaxNewSize=64m -XX:CMSInitiatingOccupancyFraction=
88 -verbose:gc -XX:+PrintGCDetails -XX:+PrintGCTimeStamps -Xloggc:/export1/hadoo
p/logs/gc-hbase.log -Dcom.sun.management.jmxremote.port=10102 -Dcom.sun.managemen
t.jmxremote.authenticate=true -Dcom.sun.management.jmxremote.ssl=false -Dcom.sun.
management.jmxremote.password.file=/home/hadoop/hbase/conf/jmxremote.password -Dc
om.sun.management.jmxremote -Dhbase.log.dir=/export1/hadoop/logs -Dhbase.log.file
=hbase-hadoop-regionserver-sv4borg12.log -Dhbase.home.dir=/home/hadoop/hbase -Dhb
ase.id.str=hadoop -Dhbase.root.logger=INFO,DRFA -Djava.library.path=/home/hadoop/
hbase/lib/native/Linux-amd64-64 -classpath /home/hadoop/hbase/bin/../conf:[many j
ars]:/home/hadoop/hadoop/conf org.apache.hadoop.hbase.regionserver.HRegionServer
start
```

jstack

`jstack` is one of the most important tools when trying to figure out what a java process is doing apart from looking at the logs. It has to be used in conjunction with jps in order to give it a process id. It shows a list of threads, each one has a name, and they appear in the order that they were created (so the top ones are the most recent threads). Here are a few example:

The main thread of a RegionServer waiting for something to do from the master:

```
"regionserver60020" prio=10 tid=0x0000000040ab4000 nid=0x45cf waiting on conditio
n [0x00007f16b6a96000..0x00007f16b6a96a70]
java.lang.Thread.State: TIMED_WAITING (parking)
    at sun.misc.Unsafe.park(Native Method)
        - parking to wait for  <0x00007f16cd5c2f30> (a java.util.concurrent.lock
s.AbstractQueuedSynchronizer$ConditionObject)
    at java.util.concurrent.locks.LockSupport.parkNanos(LockSupport.java:198)
    at java.util.concurrent.locks.AbstractQueuedSynchronizer$ConditionObject.awai
tNanos(AbstractQueuedSynchronizer.java:1963)
    at java.util.concurrent.LinkedBlockingQueue.poll(LinkedBlockingQueue.java:39
5)
```

```
    at org.apache.hadoop.hbase.regionserver.HRegionServer.run(HRegionServer.java:
647)
    at java.lang.Thread.run(Thread.java:619)
```

The MemStore flusher thread that is currently flushing to a file:

```
"regionserver60020.cacheFlusher" daemon prio=10 tid=0x0000000040f4e000 nid=0x45eb
in Object.wait() [0x00007f16b5b86000..0x00007f16b5b87af0]
java.lang.Thread.State: WAITING (on object monitor)
    at java.lang.Object.wait(Native Method)
    at java.lang.Object.wait(Object.java:485)
    at org.apache.hadoop.ipc.Client.call(Client.java:803)
        - locked <0x00007f16cb14b3a8> (a org.apache.hadoop.ipc.Client$Call)
    at org.apache.hadoop.ipc.RPC$Invoker.invoke(RPC.java:221)
    at $Proxy1.complete(Unknown Source)
    at sun.reflect.GeneratedMethodAccessor38.invoke(Unknown Source)
    at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorIm
pl.java:25)
    at java.lang.reflect.Method.invoke(Method.java:597)
    at org.apache.hadoop.io.retry.RetryInvocationHandler.invokeMethod(RetryInvoca
tionHandler.java:82)
    at org.apache.hadoop.io.retry.RetryInvocationHandler.invoke(RetryInvocationHa
ndler.java:59)
    at $Proxy1.complete(Unknown Source)
    at org.apache.hadoop.hdfs.DFSClient$DFSOutputStream.closeInternal(DFSClient.j
ava:3390)
        - locked <0x00007f16cb14b470> (a org.apache.hadoop.hdfs.DFSClient$DFSOutp
utStream)
    at org.apache.hadoop.hdfs.DFSClient$DFSOutputStream.close(DFSClient.java:330
4)
    at org.apache.hadoop.fs.FSDataOutputStream$PositionCache.close(FSDataOutputSt
ream.java:61)
    at org.apache.hadoop.fs.FSDataOutputStream.close(FSDataOutputStream.java:86)
    at org.apache.hadoop.hbase.io.hfile.HFile$Writer.close(HFile.java:650)
    at org.apache.hadoop.hbase.regionserver.StoreFile$Writer.close(StoreFile.jav
a:853)
    at org.apache.hadoop.hbase.regionserver.Store.internalFlushCache(Store.java:4
67)
        - locked <0x00007f16d00e6f08> (a java.lang.Object)
    at org.apache.hadoop.hbase.regionserver.Store.flushCache(Store.java:427)
    at org.apache.hadoop.hbase.regionserver.Store.access$100(Store.java:80)
    at org.apache.hadoop.hbase.regionserver.Store$StoreFlusherImpl.flushCache(Sto
re.java:1359)
    at org.apache.hadoop.hbase.regionserver.HRegion.internalFlushcache(HRegion.ja
va:907)
    at org.apache.hadoop.hbase.regionserver.HRegion.internalFlushcache(HRegion.ja
va:834)
    at org.apache.hadoop.hbase.regionserver.HRegion.flushcache(HRegion.java:786)
    at org.apache.hadoop.hbase.regionserver.MemStoreFlusher.flushRegion(MemStoreF
lusher.java:250)
    at org.apache.hadoop.hbase.regionserver.MemStoreFlusher.flushRegion(MemStoreF
lusher.java:224)
```

```
    at org.apache.hadoop.hbase.regionserver.MemStoreFlusher.run(MemStoreFlusher.j
ava:146)
```

A handler thread that's waiting for stuff to do (like put, delete, scan, etc.):

```
"IPC Server handler 16 on 60020" daemon prio=10 tid=0x00007f16b011d800 nid=0x4a5e
waiting on condition [0x00007f16afefd000..0x00007f16afefd9f0]
   java.lang.Thread.State: WAITING (parking)
          at sun.misc.Unsafe.park(Native Method)
              - parking to wait for  <0x00007f16cd3f8dd8> (a java.util.concurren
t.locks.AbstractQueuedSynchronizer$ConditionObject)
          at java.util.concurrent.locks.LockSupport.park(LockSupport.java:158)
          at java.util.concurrent.locks.AbstractQueuedSynchronizer$ConditionObjec
t.await(AbstractQueuedSynchronizer.java:1925)
          at java.util.concurrent.LinkedBlockingQueue.take(LinkedBlockingQueue.ja
va:358)
          at org.apache.hadoop.hbase.ipc.HBaseServer$Handler.run(HBaseServer.jav
a:1013)
```

And one that's busy doing an increment of a counter (it's in the phase where it's trying to create a scanner in order to read the last value):

```
"IPC Server handler 66 on 60020" daemon prio=10 tid=0x00007f16b006e800 nid=0x4a90
runnable [0x00007f16acb77000..0x00007f16acb77cf0]
   java.lang.Thread.State: RUNNABLE
          at org.apache.hadoop.hbase.regionserver.KeyValueHeap.<init>(KeyValueHea
p.java:56)
          at org.apache.hadoop.hbase.regionserver.StoreScanner.<init>(StoreScanne
r.java:79)
          at org.apache.hadoop.hbase.regionserver.Store.getScanner(Store.java:120
2)
          at org.apache.hadoop.hbase.regionserver.HRegion$RegionScanner.<init>(HR
egion.java:2209)
          at org.apache.hadoop.hbase.regionserver.HRegion.instantiateInternalScan
ner(HRegion.java:1063)
          at org.apache.hadoop.hbase.regionserver.HRegion.getScanner(HRegion.jav
a:1055)
          at org.apache.hadoop.hbase.regionserver.HRegion.getScanner(HRegion.jav
a:1039)
          at org.apache.hadoop.hbase.regionserver.HRegion.getLastIncrement(HRegio
n.java:2875)
          at org.apache.hadoop.hbase.regionserver.HRegion.incrementColumnValue(HR
egion.java:2978)
          at org.apache.hadoop.hbase.regionserver.HRegionServer.incrementColumnVa
lue(HRegionServer.java:2433)
          at sun.reflect.GeneratedMethodAccessor20.invoke(Unknown Source)
          at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAcce
ssorImpl.java:25)
          at java.lang.reflect.Method.invoke(Method.java:597)
          at org.apache.hadoop.hbase.ipc.HBaseRPC$Server.call(HBaseRPC.java:560)
```

```
          at org.apache.hadoop.hbase.ipc.HBaseServer$Handler.run(HBaseServer.jav
a:1027)
```

A thread that receives data from HDFS:

```
"IPC Client (47) connection to sv4borg9/10.4.24.40:9000 from hadoop" daemon prio=
10 tid=0x00007f16a02d0000 nid=0x4fa3 runnable [0x00007f16b517d000..0x00007f16b517
dbf0]
   java.lang.Thread.State: RUNNABLE
          at sun.nio.ch.EPollArrayWrapper.epollWait(Native Method)
          at sun.nio.ch.EPollArrayWrapper.poll(EPollArrayWrapper.java:215)
          at sun.nio.ch.EPollSelectorImpl.doSelect(EPollSelectorImpl.java:65)
          at sun.nio.ch.SelectorImpl.lockAndDoSelect(SelectorImpl.java:69)
              - locked <0x00007f17d5b68c00> (a sun.nio.ch.Util$1)
              - locked <0x00007f17d5b68be8> (a java.util.Collections$Unmodifiable
Set)
              - locked <0x00007f1877959b50> (a sun.nio.ch.EPollSelectorImpl)
          at sun.nio.ch.SelectorImpl.select(SelectorImpl.java:80)
          at org.apache.hadoop.net.SocketIOWithTimeout$SelectorPool.select(Socket
IOWithTimeout.java:332)
          at org.apache.hadoop.net.SocketIOWithTimeout.doIO(SocketIOWithTimeout.j
ava:157)
          at org.apache.hadoop.net.SocketInputStream.read(SocketInputStream.java:
155)
          at org.apache.hadoop.net.SocketInputStream.read(SocketInputStream.java:
128)
          at java.io.FilterInputStream.read(FilterInputStream.java:116)
          at org.apache.hadoop.ipc.Client$Connection$PingInputStream.read(Client.
java:304)
          at java.io.BufferedInputStream.fill(BufferedInputStream.java:218)
          at java.io.BufferedInputStream.read(BufferedInputStream.java:237)
              - locked <0x00007f1808539178> (a java.io.BufferedInputStream)
          at java.io.DataInputStream.readInt(DataInputStream.java:370)
          at org.apache.hadoop.ipc.Client$Connection.receiveResponse(Client.java:
569)
          at org.apache.hadoop.ipc.Client$Connection.run(Client.java:477)
```

And here is a master trying to recover a lease after a RegionServer died:

```
"LeaseChecker" daemon prio=10 tid=0x00000000407ef800 nid=0x76cd waiting on condit
ion [0x00007f6d0eae2000..0x00007f6d0eae2a70]
--
   java.lang.Thread.State: WAITING (on object monitor)
          at java.lang.Object.wait(Native Method)
          at java.lang.Object.wait(Object.java:485)
          at org.apache.hadoop.ipc.Client.call(Client.java:726)
          - locked <0x00007f6d1cd28f80> (a org.apache.hadoop.ipc.Client$Call)
          at org.apache.hadoop.ipc.RPC$Invoker.invoke(RPC.java:220)
          at $Proxy1.recoverBlock(Unknown Source)
          at org.apache.hadoop.hdfs.DFSClient$DFSOutputStream.processDatanodeErro
r(DFSClient.java:2636)
```

```
          at org.apache.hadoop.hdfs.DFSClient$DFSOutputStream.<init>(DFSClient.ja
va:2832)
          at org.apache.hadoop.hdfs.DFSClient.append(DFSClient.java:529)
          at org.apache.hadoop.hdfs.DistributedFileSystem.append(DistributedFileS
ystem.java:186)
          at org.apache.hadoop.fs.FileSystem.append(FileSystem.java:530)
          at org.apache.hadoop.hbase.util.FSUtils.recoverFileLease(FSUtils.java:6
19)
          at org.apache.hadoop.hbase.regionserver.wal.HLog.splitLog(HLog.java:132
2)
          at org.apache.hadoop.hbase.regionserver.wal.HLog.splitLog(HLog.java:121
0)
          at org.apache.hadoop.hbase.master.HMaster.splitLogAfterStartup(HMaster.
java:648)
          at org.apache.hadoop.hbase.master.HMaster.joinCluster(HMaster.java:572)
          at org.apache.hadoop.hbase.master.HMaster.run(HMaster.java:503)
```

OpenTSDB

OpenTSDB is an excellent alternative to Ganglia as it uses Apache HBase to store all the time series and doesn't have to downsample. Monitoring your own HBase cluster that hosts OpenTSDB is a good exercise.

Here's an example of a cluster that's suffering from hundreds of compactions launched almost all around the same time, which severely affects the IO performance: (TODO: insert graph plotting compactionQueueSize)

It's a good practice to build dashboards with all the important graphs per machine and per cluster so that debugging issues can be done with a single quick look. For example, at StumbleUpon there's one dashboard per cluster with the most important metrics from both the OS and Apache HBase. You can then go down at the machine level and get even more detailed metrics.

clusterssh+top

clusterssh+top, it's like a poor man's monitoring system and it can be quite useful when you have only a few machines as it's very easy to setup. Starting clusterssh will give you one terminal per machine and another terminal in which whatever you type will be retyped in every window. This means that you can type `top` once and it will start it for all of your machines at the same time giving you full view of the current state of your cluster. You can also tail all the logs at the same time, edit files, etc.

Client

For more information on the HBase client, see client.

ScannerTimeoutException or UnknownScannerException

This is thrown if the time between RPC calls from the client to RegionServer exceeds the scan timeout. For example, if `Scan.setCaching` is set to 500, then there will be an RPC call to fetch the next batch of rows every 500 `.next()` calls on the ResultScanner because data is being transferred in blocks of 500 rows to the client. Reducing the setCaching value may be an option, but setting this value too low makes for inefficient processing on numbers of rows.

See Scan Caching.

Performance Differences in Thrift and Java APIs

Poor performance, or even `ScannerTimeoutExceptions`, can occur if `Scan.setCaching` is too high, as discussed in ScannerTimeoutException or UnknownScannerException. If the Thrift client uses the wrong caching settings for a given workload, performance can suffer compared to the Java API. To set caching for a given scan in the Thrift client, use the `scannerGetList(scannerId, numRows)` method, where `numRows` is an integer representing the number of rows to cache. In one case, it was found that reducing the cache for Thrift scans from 1000 to 100 increased performance to near parity with the Java API given the same queries.

See also Jesse Andersen's blog post about using Scans with Thrift.

`LeaseException` when calling `Scanner.next`

In some situations clients that fetch data from a RegionServer get a LeaseException instead of the usual ScannerTimeoutException or UnknownScannerException. Usually the source of the exception is `org.apache.hadoop.hbase.regionserver.Leases.removeLease(Leases.java:230)` (line number may vary). It tends to happen in the context of a slow/freezing `RegionServer#next` call. It can be prevented by having `hbase.rpc.timeout` > `hbase.client.scanner.timeout.period`. Harsh J investigated the issue as part of the mailing list thread HBase, mail # user - Lease does not exist exceptions

Shell or client application throws lots of scary exceptions during normal operation

Since 0.20.0 the default log level for `org.apache.hadoop.hbase.*` is DEBUG.

On your clients, edit *$HBASE_HOME/conf/log4j.properties* and change this: `log4j.logger.org.apache.hadoop.hbase=DEBUG` to this: `log4j.logger.org.apache.hadoop.hbase=INFO`, or even `log4j.logger.org.apache.hadoop.hbase=WARN`.

Long Client Pauses With Compression

This is a fairly frequent question on the Apache HBase dist-list. The scenario is that a client is typically inserting a lot of data into a relatively un-optimized HBase cluster. Compression can exacerbate the pauses, although it is not the source of the problem.

See Table Creation: Pre-Creating Regions on the pattern for pre-creating regions and confirm that the table isn't starting with a single region.

See HBase Configurations for cluster configuration, particularly `hbase.hstore.blockingStoreFiles`, `hbase.hregion.memstore.block.multiplier`, `MAX_FILESIZE` (region size), and `MEMSTORE_FLUSHSIZE.`

A slightly longer explanation of why pauses can happen is as follows: Puts are sometimes blocked on the MemStores which are blocked by the flusher thread which is blocked because there are too many files to compact because the compactor is given too many small files to compact and has to compact the same data repeatedly. This situation can occur even with minor compactions. Compounding this situation, Apache HBase doesn't compress data in memory. Thus, the 64MB that lives in the MemStore could become a 6MB file after compression - which results in a smaller StoreFile. The upside is that more data is packed into the same region, but performance is achieved by being able to write larger files - which is why HBase waits until the flushsize before writing a new StoreFile. And smaller StoreFiles become targets for compaction. Without compression the files are much bigger and don't need as much compaction, however this is at the expense of I/O.

Secure Client Connect ([Caused by GSSException: No valid credentials provided...])

You may encounter the following error:

```
Secure Client Connect ([Caused by GSSException: No valid credentials provided
        (Mechanism level: Request is a replay (34) V PROCESS_TGS)])
```

This issue is caused by bugs in the MIT Kerberos replay_cache component, #1201 and #5924. These bugs caused the old version of krb5-server to erroneously block subsequent requests sent from a Principal. This caused krb5-server to block the connections sent from one Client (one HTable instance with multi-threading connection instances for each RegionServer); Messages, such as `Request is a replay (34)`, are logged in the client log You can ignore the messages, because HTable will retry 5 * 10 (50) times for each failed connection by default. HTable will throw IOException if any connection to the RegionServer fails after the retries, so that the user client code for HTable instance can handle it further. NOTE: `HTable` is deprecated in HBase 1.0, in favor of `Table`.

Alternatively, update krb5-server to a version which solves these issues, such as krb5-server-1.10.3. See JIRA HBASE-10379 for more details.

ZooKeeper Client Connection Errors

Errors like this...

```
11/07/05 11:26:41 WARN zookeeper.ClientCnxn: Session 0x0 for server null,
 unexpected error, closing socket connection and attempting reconnect
 java.net.ConnectException: Connection refused: no further information
        at sun.nio.ch.SocketChannelImpl.checkConnect(Native Method)
        at sun.nio.ch.SocketChannelImpl.finishConnect(Unknown Source)
        at org.apache.zookeeper.ClientCnxn$SendThread.run(ClientCnxn.java:1078)
 11/07/05 11:26:43 INFO zookeeper.ClientCnxn: Opening socket connection to
 server localhost/127.0.0.1:2181
 11/07/05 11:26:44 WARN zookeeper.ClientCnxn: Session 0x0 for server null,
 unexpected error, closing socket connection and attempting reconnect
 java.net.ConnectException: Connection refused: no further information
        at sun.nio.ch.SocketChannelImpl.checkConnect(Native Method)
        at sun.nio.ch.SocketChannelImpl.finishConnect(Unknown Source)
        at org.apache.zookeeper.ClientCnxn$SendThread.run(ClientCnxn.java:1078)
 11/07/05 11:26:45 INFO zookeeper.ClientCnxn: Opening socket connection to
 server localhost/127.0.0.1:2181
```

...are either due to ZooKeeper being down, or unreachable due to network issues.

The utility **zkcli** may help investigate ZooKeeper issues.

Client running out of memory though heap size seems to be stable (but the off-heap/direct heap keeps growing)

You are likely running into the issue that is described and worked through in the mail thread HBase, mail # user - Suspected memory leak and continued over in HBase, mail # dev - FeedbackRe: Suspected memory leak. A workaround is passing your client-side JVM a reasonable value for `-XX:MaxDirectMemorySize`. By default, the `MaxDirectMemorySize` is equal to your `-Xmx` max heapsize setting (if `-Xmx` is set). Try setting it to something smaller (for example, one user had success setting it to `1g` when they had a client-side heap of `12g`). If you set it too small, it will bring on `FullGCs` so keep it a bit hefty. You want to make this setting client-side only especially if you are running the new experimental server-side off-heap cache since this feature depends on being able to use big direct buffers (You may have to keep separate client-side and server-side config dirs).

Secure Client Cannot Connect ([Caused by GSSException: No valid credentials provided(Mechanism level: Failed to find any Kerberos tgt)])

There can be several causes that produce this symptom.

First, check that you have a valid Kerberos ticket. One is required in order to set up communication with a secure Apache HBase cluster. Examine the ticket currently in the credential cache, if any, by running the `klist` command line utility. If no ticket is listed, you must obtain a ticket by running the `kinit` command with either a keytab specified, or by interactively entering a password for the desired principal.

Then, consult the Java Security Guide troubleshooting section. The most common problem addressed there is resolved by setting `javax.security.auth.useSubjectCredsOnly` system property value to `false`.

Because of a change in the format in which MIT Kerberos writes its credentials cache, there is a bug in the Oracle JDK 6 Update 26 and earlier that causes Java to be unable to read the Kerberos credentials cache created by versions of MIT Kerberos 1.8.1 or higher. If you have this problematic combination of components in your environment, to work around this problem, first log in with `kinit` and then immediately refresh the credential cache with `kinit -R`. The refresh will rewrite the credential cache without the problematic formatting.

Prior to JDK 1.4, the JCE was an unbundled product, and as such, the JCA and JCE were regularly referred to as separate, distinct components. As JCE is now bundled in the JDK 7.0, the distinction is becoming less apparent. Since the JCE uses the same architecture as the JCA, the JCE should be more properly thought of as a part of the JCA.

You may need to install the Java Cryptography Extension, or JCE because of JDK 1.5 or earlier version. Insure the JCE jars are on the classpath on both server and client systems.

You may also need to download the unlimited strength JCE policy files. Uncompress and extract the downloaded file, and install the policy jars into *<java-home>/lib/security*.

Trouble shooting master registry issues

- For connectivity issues, usually an exception like "MasterRegistryFetchException: Exception making rpc to masters..." is logged in the client logs. The logging includes the list of master end points that were attempted by the client. The bottom part of the stack trace should include the underlying reason. If you suspect connectivity issues (ConnectionRefused?), make sure the master end points are accessible from client.
- If there is a suspicion of higher load on the masters due to hedging of RPCs, it can be controlled by either reducing the hedging fan out (via *hbase.rpc.hedged.fanout*) or by restricting the set of masters that clients can access for the master registry purposes (via *hbase.masters*).

Refer to Master Registry (new as of 2.3.0) and Client configuration and dependencies connecting to an HBase cluster for more details.

MapReduce

You Think You're On The Cluster, But You're Actually Local

This following stacktrace happened using `ImportTsv`, but things like this can happen on any job with a mis-configuration.

```
    WARN mapred.LocalJobRunner: job_local_0001
java.lang.IllegalArgumentException: Can't read partitions file
       at org.apache.hadoop.hbase.mapreduce.hadoopbackport.TotalOrderPartitioner.
setConf(TotalOrderPartitioner.java:111)
       at org.apache.hadoop.util.ReflectionUtils.setConf(ReflectionUtils.java:62)
       at org.apache.hadoop.util.ReflectionUtils.newInstance(ReflectionUtils.jav
a:117)
```

```
        at org.apache.hadoop.mapred.MapTask$NewOutputCollector.<init>(MapTask.jav
a:560)
        at org.apache.hadoop.mapred.MapTask.runNewMapper(MapTask.java:639)
        at org.apache.hadoop.mapred.MapTask.run(MapTask.java:323)
        at org.apache.hadoop.mapred.LocalJobRunner$Job.run(LocalJobRunner.java:21
0)
Caused by: java.io.FileNotFoundException: File _partition.lst does not exist.
        at org.apache.hadoop.fs.RawLocalFileSystem.getFileStatus(RawLocalFileSyste
m.java:383)
        at org.apache.hadoop.fs.FilterFileSystem.getFileStatus(FilterFileSystem.ja
va:251)
        at org.apache.hadoop.fs.FileSystem.getLength(FileSystem.java:776)
        at org.apache.hadoop.io.SequenceFile$Reader.<init>(SequenceFile.java:1424)
        at org.apache.hadoop.io.SequenceFile$Reader.<init>(SequenceFile.java:1419)
        at org.apache.hadoop.hbase.mapreduce.hadoopbackport.TotalOrderPartitioner.
readPartitions(TotalOrderPartitioner.java:296)
```

...see the critical portion of the stack? It's...

```
at org.apache.hadoop.mapred.LocalJobRunner$Job.run(LocalJobRunner.java:210)
```

LocalJobRunner means the job is running locally, not on the cluster.

To solve this problem, you should run your MR job with your `HADOOP_CLASSPATH` set to include the HBase dependencies. The "hbase classpath" utility can be used to do this easily. For example (substitute VERSION with your HBase version):

```
HADOOP_CLASSPATH=`hbase classpath` hadoop jar $HBASE_HOME/hbase-mapreduce-VERSIO
N.jar rowcounter usertable
```

See HBase, MapReduce, and the CLASSPATH for more information on HBase MapReduce jobs and classpaths.

Launching a job, you get java.lang.IllegalAccessError: com/google/protobuf/HBaseZeroCopyByteString or class com.google.protobuf.ZeroCopyLiteralByteString cannot access its superclass com.google.protobuf.LiteralByteString

See HBASE-10304 Running an hbase job jar: IllegalAccessError: class com.google.protobuf.ZeroCopyLiteralByteString cannot access its superclass com.google.protobuf.LiteralByteString and HBASE-11118 non environment variable solution for "IllegalAccessError: class com.google.protobuf.ZeroCopyLiteralByteString

cannot access its superclass com.google.protobuf.LiteralByteString". The issue can also show up when trying to run spark jobs. See HBASE-10877 HBase non-retriable exception list should be expanded.

NameNode

For more information on the NameNode, see HDFS.

HDFS Utilization of Tables and Regions

To determine how much space HBase is using on HDFS use the `hadoop` shell commands from the NameNode. For example...

```
hadoop fs -dus /hbase/
```

...returns the summarized disk utilization for all HBase objects.

```
hadoop fs -dus /hbase/myTable
```

...returns the summarized disk utilization for the HBase table 'myTable'.

```
hadoop fs -du /hbase/myTable
```

...returns a list of the regions under the HBase table 'myTable' and their disk utilization.

For more information on HDFS shell commands, see the HDFS FileSystem Shell documentation.

Browsing HDFS for HBase Objects

Sometimes it will be necessary to explore the HBase objects that exist on HDFS. These objects could include the WALs (Write Ahead Logs), tables, regions, StoreFiles, etc. The easiest way to do this is with the NameNode web application that runs on port 50070. The NameNode web application will provide links to the all the DataNodes in the cluster so that they can be browsed seamlessly.

The HDFS directory structure of HBase tables in the cluster is...

```
/hbase
    /data
        /<Namespace>                    (Namespaces in the cluster)
            /<Table>                    (Tables in the cluster)
                /<Region>               (Regions for the table)
                    /<ColumnFamily>     (ColumnFamilies for the Region for the ta
ble)
                        /<StoreFile>    (StoreFiles for the ColumnFamily for the
Regions for the table)
```

The HDFS directory structure of HBase WAL is..

```
/hbase
    /WALs
        /<RegionServer>    (RegionServers)
            /<WAL>         (WAL files for the RegionServer)
```

See the HDFS User Guide for other non-shell diagnostic utilities like `fsck`.

Zero size WALs with data in them

Problem: when getting a listing of all the files in a RegionServer's *WALs* directory, one file has a size of 0 but it contains data.

Answer: It's an HDFS quirk. A file that's currently being written to will appear to have a size of 0 but once it's closed it will show its true size

Use Cases

Two common use-cases for querying HDFS for HBase objects is research the degree of uncompaction of a table. If there are a large number of StoreFiles for each ColumnFamily it could indicate the need for a major compaction. Additionally, after a major compaction if the resulting StoreFile is "small" it could indicate the need for a reduction of ColumnFamilies for the table.

Unexpected Filesystem Growth

If you see an unexpected spike in filesystem usage by HBase, two possible culprits are snapshots and WALs.

Snapshots

When you create a snapshot, HBase retains everything it needs to recreate the table's state at that time of the snapshot. This includes deleted cells or expired versions. For this

reason, your snapshot usage pattern should be well-planned, and you should prune snapshots that you no longer need. Snapshots are stored in `/hbase/.hbase-snapshot`, and archives needed to restore snapshots are stored in `/hbase/archive/<tablename>/<region>/<column_family>/`.

> ⚠ **Do not** manage snapshots or archives manually via HDFS. HBase provides APIs and HBase Shell commands for managing them. For more information, see ops.snapshots.

WAL

Write-ahead logs (WALs) are stored in subdirectories of the HBase root directory, typically `/hbase/`, depending on their status. Already-processed WALs are stored in `/hbase/oldWALs/` and corrupt WALs are stored in `/hbase/.corrupt/` for examination. If the size of one of these subdirectories is growing, examine the HBase server logs to find the root cause for why WALs are not being processed correctly.

If you use replication and `/hbase/oldWALs/` is using more space than you expect, remember that WALs are saved when replication is disabled, as long as there are peers.

Do not manage WALs manually via HDFS.

Network

Network Spikes

If you are seeing periodic network spikes you might want to check the `compactionQueues` to see if major compactions are happening.

See Managed Compactions for more information on managing compactions.

Loopback IP

HBase expects the loopback IP Address to be 127.0.0.1.

Network Interfaces

Are all the network interfaces functioning correctly? Are you sure? See the Troubleshooting Case Study in Case Studies.

RegionServer

For more information on the RegionServers, see RegionServer.

Startup Errors

Master Starts, But RegionServers Do Not

The Master believes the RegionServers have the IP of 127.0.0.1 - which is localhost and resolves to the master's own localhost.

The RegionServers are erroneously informing the Master that their IP addresses are 127.0.0.1.

Modify */etc/hosts* on the region servers, from...

```
# Do not remove the following line, or various programs
# that require network functionality will fail.
127.0.0.1               fully.qualified.regionservername regionservername  localh
ost.localdomain localhost
::1             localhost6.localdomain6 localhost6
```

... to (removing the master node's name from localhost)...

```
# Do not remove the following line, or various programs
# that require network functionality will fail.
127.0.0.1               localhost.localdomain localhost
::1             localhost6.localdomain6 localhost6
```

Compression Link Errors

Since compression algorithms such as LZO need to be installed and configured on each cluster this is a frequent source of startup error. If you see messages like this...

```
11/02/20 01:32:15 ERROR lzo.GPLNativeCodeLoader: Could not load native gpl librar
y
java.lang.UnsatisfiedLinkError: no gplcompression in java.library.path
        at java.lang.ClassLoader.loadLibrary(ClassLoader.java:1734)
        at java.lang.Runtime.loadLibrary0(Runtime.java:823)
        at java.lang.System.loadLibrary(System.java:1028)
```

... then there is a path issue with the compression libraries. See the Configuration section on LZO compression configuration.

RegionServer aborts due to lack of hsync for filesystem

In order to provide data durability for writes to the cluster HBase relies on the ability to durably save state in a write ahead log. When using a version of Apache Hadoop Common's filesystem API that supports checking on the availability of needed calls, HBase will proactively abort the cluster if it finds it can't operate safely.

For RegionServer roles, the failure will show up in logs like this:

```
2018-04-05 11:36:22,785 ERROR [regionserver/192.168.1.123:16020] wal.AsyncFSWALPr
ovider: The RegionServer async write ahead log provider relies on the ability to
call hflush and hsync for proper operation during component failures, but the cur
rent FileSystem does not support doing so. Please check the config value of 'hbas
e.wal.dir' and ensure it points to a FileSystem mount that has suitable capabilit
ies for output streams.
2018-04-05 11:36:22,799 ERROR [regionserver/192.168.1.123:16020] regionserver.HRe
gionServer: ***** ABORTING region server 192.168.1.123,16020,1522946074234: Unhan
dled: cannot get log writer *****
java.io.IOException: cannot get log writer
        at org.apache.hadoop.hbase.wal.AsyncFSWALProvider.createAsyncWriter(Async
FSWALProvider.java:112)
        at org.apache.hadoop.hbase.regionserver.wal.AsyncFSWAL.createWriterInstan
ce(AsyncFSWAL.java:612)
        at org.apache.hadoop.hbase.regionserver.wal.AsyncFSWAL.createWriterInstan
ce(AsyncFSWAL.java:124)
        at org.apache.hadoop.hbase.regionserver.wal.AbstractFSWAL.rollWriter(Abst
ractFSWAL.java:759)
        at org.apache.hadoop.hbase.regionserver.wal.AbstractFSWAL.rollWriter(Abst
ractFSWAL.java:489)
        at org.apache.hadoop.hbase.regionserver.wal.AsyncFSWAL.<init>(AsyncFSWAL.
java:251)
        at org.apache.hadoop.hbase.wal.AsyncFSWALProvider.createWAL(AsyncFSWALPro
vider.java:69)
        at org.apache.hadoop.hbase.wal.AsyncFSWALProvider.createWAL(AsyncFSWALPro
vider.java:44)
        at org.apache.hadoop.hbase.wal.AbstractFSWALProvider.getWAL(AbstractFSWAL
Provider.java:138)
        at org.apache.hadoop.hbase.wal.AbstractFSWALProvider.getWAL(AbstractFSWAL
Provider.java:57)
        at org.apache.hadoop.hbase.wal.WALFactory.getWAL(WALFactory.java:252)
        at org.apache.hadoop.hbase.regionserver.HRegionServer.getWAL(HRegionServe
r.java:2105)
        at org.apache.hadoop.hbase.regionserver.HRegionServer.buildServerLoad(HRe
gionServer.java:1326)
        at org.apache.hadoop.hbase.regionserver.HRegionServer.tryRegionServerRepo
rt(HRegionServer.java:1191)
        at org.apache.hadoop.hbase.regionserver.HRegionServer.run(HRegionServer.j
ava:1007)
        at java.lang.Thread.run(Thread.java:745)
Caused by: org.apache.hadoop.hbase.util.CommonFSUtils$StreamLacksCapabilityExcept
ion: hflush and hsync
        at org.apache.hadoop.hbase.io.asyncfs.AsyncFSOutputHelper.createOutput(As
yncFSOutputHelper.java:69)
```

```
        at org.apache.hadoop.hbase.regionserver.wal.AsyncProtobufLogWriter.initOu
tput(AsyncProtobufLogWriter.java:168)
        at org.apache.hadoop.hbase.regionserver.wal.AbstractProtobufLogWriter.ini
t(AbstractProtobufLogWriter.java:167)
        at org.apache.hadoop.hbase.wal.AsyncFSWALProvider.createAsyncWriter(Async
FSWALProvider.java:99)
        ... 15 more
```

If you are attempting to run in standalone mode and see this error, please walk back through the section Quick Start - Standalone HBase and ensure you have included **all** the given configuration settings.

RegionServer aborts due to can not initialize access to HDFS

We will try to use *AsyncFSWAL* for HBase-2.x as it has better performance while consuming less resources. But the problem for *AsyncFSWAL* is that it hacks into the internal of the DFSClient implementation, so it will easily be broken when upgrading hadoop, even for a simple patch release.

If you do not specify the wal provider, we will try to fall back to the old *FSHLog* if we fail to initialize *AsyncFSWAL*, but it may not always work. The failure will show up in logs like this:

```
18/07/02 18:51:06 WARN concurrent.DefaultPromise: An exception was
thrown by org.apache.hadoop.hbase.io.asyncfs.FanOutOneBlockAsyncDFSOutputHelper$1
3.operationComplete()
java.lang.Error: Couldn't properly initialize access to HDFS
internals. Please update your WAL Provider to not make use of the
'asyncfs' provider. See HBASE-16110 for more information.
     at org.apache.hadoop.hbase.io.asyncfs.FanOutOneBlockAsyncDFSOutputSaslHelpe
r.<clinit>(FanOutOneBlockAsyncDFSOutputSaslHelper.java:268)
     at org.apache.hadoop.hbase.io.asyncfs.FanOutOneBlockAsyncDFSOutputHelper.ini
tialize(FanOutOneBlockAsyncDFSOutputHelper.java:661)
     at org.apache.hadoop.hbase.io.asyncfs.FanOutOneBlockAsyncDFSOutputHelper.acc
ess$300(FanOutOneBlockAsyncDFSOutputHelper.java:118)
     at org.apache.hadoop.hbase.io.asyncfs.FanOutOneBlockAsyncDFSOutputHelper$13.
operationComplete(FanOutOneBlockAsyncDFSOutputHelper.java:720)
     at org.apache.hadoop.hbase.io.asyncfs.FanOutOneBlockAsyncDFSOutputHelper$13.
operationComplete(FanOutOneBlockAsyncDFSOutputHelper.java:715)
     at org.apache.hbase.thirdparty.io.netty.util.concurrent.DefaultPromise.notif
yListener0(DefaultPromise.java:507)
     at org.apache.hbase.thirdparty.io.netty.util.concurrent.DefaultPromise.notif
yListeners0(DefaultPromise.java:500)
     at org.apache.hbase.thirdparty.io.netty.util.concurrent.DefaultPromise.notif
yListenersNow(DefaultPromise.java:479)
     at org.apache.hbase.thirdparty.io.netty.util.concurrent.DefaultPromise.notif
yListeners(DefaultPromise.java:420)
     at org.apache.hbase.thirdparty.io.netty.util.concurrent.DefaultPromise.trySu
ccess(DefaultPromise.java:104)
```

```
     at org.apache.hbase.thirdparty.io.netty.channel.DefaultChannelPromise.trySuc
cess(DefaultChannelPromise.java:82)
     at org.apache.hbase.thirdparty.io.netty.channel.epoll.AbstractEpollChannel$A
bstractEpollUnsafe.fulfillConnectPromise(AbstractEpollChannel.java:638)
     at org.apache.hbase.thirdparty.io.netty.channel.epoll.AbstractEpollChannel$A
bstractEpollUnsafe.finishConnect(AbstractEpollChannel.java:676)
     at org.apache.hbase.thirdparty.io.netty.channel.epoll.AbstractEpollChannel$A
bstractEpollUnsafe.epollOutReady(AbstractEpollChannel.java:552)
     at org.apache.hbase.thirdparty.io.netty.channel.epoll.EpollEventLoop.process
Ready(EpollEventLoop.java:394)
     at org.apache.hbase.thirdparty.io.netty.channel.epoll.EpollEventLoop.run(Epo
llEventLoop.java:304)
     at org.apache.hbase.thirdparty.io.netty.util.concurrent.SingleThreadEventExe
cutor$5.run(SingleThreadEventExecutor.java:858)
     at org.apache.hbase.thirdparty.io.netty.util.concurrent.DefaultThreadFactory
$DefaultRunnableDecorator.run(DefaultThreadFactory.java:138)
     at java.lang.Thread.run(Thread.java:748)
 Caused by: java.lang.NoSuchMethodException:
org.apache.hadoop.hdfs.DFSClient.decryptEncryptedDataEncryptionKey(org.apache.had
oop.fs.FileEncryptionInfo)
     at java.lang.Class.getDeclaredMethod(Class.java:2130)
     at org.apache.hadoop.hbase.io.asyncfs.FanOutOneBlockAsyncDFSOutputSaslHelpe
r.createTransparentCryptoHelper(FanOutOneBlockAsyncDFSOutputSaslHelper.java:232)
     at org.apache.hadoop.hbase.io.asyncfs.FanOutOneBlockAsyncDFSOutputSaslHelpe
r.<clinit>(FanOutOneBlockAsyncDFSOutputSaslHelper.java:262)
     ... 18 more
```

If you hit this error, please specify *FSHLog*, i.e, *filesystem*, explicitly in your config file.

```
<property>
  <name>hbase.wal.provider</name>
  <value>filesystem</value>
</property>
```

And do not forget to send an email to the `user@hbase.apache.org` or `dev@hbase.apache.org` to report the failure and also your hadoop version, we will try to fix the problem ASAP in the next release.

Runtime Errors

RegionServer Hanging

Are you running an old JVM (`<` 1.6.0*u21?)? When you look at a thread dump, does it look like threads are BLOCKED but no one holds the lock all are blocked on? See* ***HBASE 3622 Deadlock in HBaseServer (JVM bug?)****. Adding* `-XX:+UseMembar` *to the HBase* `HBASE_OPTS` *in _conf/hbase-env.sh* may fix it.

java.io.IOException...(Too many open files)

If you see log messages like this...

```
2010-09-13 01:24:17,336 WARN org.apache.hadoop.hdfs.server.datanode.DataNode:
Disk-related IOException in BlockReceiver constructor. Cause is java.io.IOExcepti
on: Too many open files
        at java.io.UnixFileSystem.createFileExclusively(Native Method)
        at java.io.File.createNewFile(File.java:883)
```

... see the Getting Started section on ulimit and nproc configuration.

xceiverCount 258 exceeds the limit of concurrent xcievers 256

This typically shows up in the DataNode logs.

TODO: add link. See the Getting Started section on xceivers configuration.

System instability, and the presence of "java.lang.OutOfMemoryError: unable to createnew native thread in exceptions" HDFS DataNode logs or that of any system daemon

See the Getting Started section on ulimit and nproc configuration. The default on recent Linux distributions is 1024 - which is far too low for HBase.

DFS instability and/or RegionServer lease timeouts

If you see warning messages like this...

```
2009-02-24 10:01:33,516 WARN org.apache.hadoop.hbase.util.Sleeper: We slept xxx m
s, ten times longer than scheduled: 10000
2009-02-24 10:01:33,516 WARN org.apache.hadoop.hbase.util.Sleeper: We slept xxx m
s, ten times longer than scheduled: 15000
2009-02-24 10:01:36,472 WARN org.apache.hadoop.hbase.regionserver.HRegionServer:
unable to report to master for xxx milliseconds - retrying
```

... or see full GC compactions then you may be experiencing full GC's.

"No live nodes contain current block" and/or YouAreDeadException

These errors can happen either when running out of OS file handles or in periods of severe network problems where the nodes are unreachable.

See the Getting Started section on ulimit and nproc configuration and check your network.

ZooKeeper SessionExpired events

Master or RegionServers shutting down with messages like those in the logs:

```
WARN org.apache.zookeeper.ClientCnxn: Exception
closing session 0x278bd16a96000f to sun.nio.ch.SelectionKeyImpl@355811ec
java.io.IOException: TIMED OUT
       at org.apache.zookeeper.ClientCnxn$SendThread.run(ClientCnxn.java:906)
WARN org.apache.hadoop.hbase.util.Sleeper: We slept 79410ms, ten times longer tha
n scheduled: 5000
INFO org.apache.zookeeper.ClientCnxn: Attempting connection to server hostname/I
P:PORT
INFO org.apache.zookeeper.ClientCnxn: Priming connection to java.nio.channels.Soc
ketChannel[connected local=/IP:PORT remote=hostname/IP:PORT]
INFO org.apache.zookeeper.ClientCnxn: Server connection successful
WARN org.apache.zookeeper.ClientCnxn: Exception closing session 0x278bd16a96000d
to sun.nio.ch.SelectionKeyImpl@3544d65e
java.io.IOException: Session Expired
       at org.apache.zookeeper.ClientCnxn$SendThread.readConnectResult(ClientCnx
n.java:589)
       at org.apache.zookeeper.ClientCnxn$SendThread.doIO(ClientCnxn.java:709)
       at org.apache.zookeeper.ClientCnxn$SendThread.run(ClientCnxn.java:945)
ERROR org.apache.hadoop.hbase.regionserver.HRegionServer: ZooKeeper session expir
ed
```

The JVM is doing a long running garbage collecting which is pausing every threads (aka "stop the world"). Since the RegionServer's local ZooKeeper client cannot send heartbeats, the session times out. By design, we shut down any node that isn't able to contact the ZooKeeper ensemble after getting a timeout so that it stops serving data that may already be assigned elsewhere.

- Make sure you give plenty of RAM (in *hbase-env.sh*), the default of 1GB won't be able to sustain long running imports.
- Make sure you don't swap, the JVM never behaves well under swapping.
- Make sure you are not CPU starving the RegionServer thread. For example, if you are running a MapReduce job using 6 CPU-intensive tasks on a machine with 4 cores, you are probably starving the RegionServer enough to create longer garbage collection pauses.
- Increase the ZooKeeper session timeout

If you wish to increase the session timeout, add the following to your *hbase-site.xml* to increase the timeout from the default of 60 seconds to 120 seconds.

```
<property>
```

```
  <name>zookeeper.session.timeout</name>
  <value>120000</value>
</property>
<property>
  <name>hbase.zookeeper.property.tickTime</name>
  <value>6000</value>
</property>
```

Be aware that setting a higher timeout means that the regions served by a failed RegionServer will take at least that amount of time to be transferred to another RegionServer. For a production system serving live requests, we would instead recommend setting it lower than 1 minute and over-provision your cluster in order the lower the memory load on each machines (hence having less garbage to collect per machine).

If this is happening during an upload which only happens once (like initially loading all your data into HBase), consider bulk loading.

See ZooKeeper, The Cluster Canary for other general information about ZooKeeper troubleshooting.

NotServingRegionException

This exception is "normal" when found in the RegionServer logs at DEBUG level. This exception is returned back to the client and then the client goes back to `hbase:meta` to find the new location of the moved region.

However, if the NotServingRegionException is logged ERROR, then the client ran out of retries and something probably wrong.

Logs flooded with '2011-01-10 12:40:48,407 INFO org.apache.hadoop.io.compress.CodecPool: Gotbrand-new compressor' messages

We are not using the native versions of compression libraries. See HBASE-1900 Put back native support when hadoop 0.21 is released. Copy the native libs from hadoop under HBase lib dir or symlink them into place and the message should go away.

Server handler X on 60020 caught: java.nio.channels.ClosedChannelException

If you see this type of message it means that the region server was trying to read/send data from/to a client but it already went away. Typical causes for this are if the client was killed (you see a storm of messages like this when a MapReduce job is killed or fails) or if

the client receives a SocketTimeoutException. It's harmless, but you should consider digging in a bit more if you aren't doing something to trigger them.

Snapshot Errors Due to Reverse DNS

Several operations within HBase, including snapshots, rely on properly configured reverse DNS. Some environments, such as Amazon EC2, have trouble with reverse DNS. If you see errors like the following on your RegionServers, check your reverse DNS configuration:

```
2013-05-01 00:04:56,356 DEBUG org.apache.hadoop.hbase.procedure.Subprocedure: Sub
procedure 'backup1'
coordinator notified of 'acquire', waiting on 'reached' or 'abort' from coordinat
or.
```

In general, the hostname reported by the RegionServer needs to be the same as the hostname the Master is trying to reach. You can see a hostname mismatch by looking for the following type of message in the RegionServer's logs at start-up.

```
2013-05-01 00:03:00,614 INFO org.apache.hadoop.hbase.regionserver.HRegionServer:
Master passed us hostname
to use. Was=myhost-1234, Now=ip-10-55-88-99.ec2.internal
```

Shutdown Errors

Master

For more information on the Master, see master.

Startup Errors

Master says that you need to run the HBase migrations script

Upon running that, the HBase migrations script says no files in root directory.

HBase expects the root directory to either not exist, or to have already been initialized by HBase running a previous time. If you create a new directory for HBase using Hadoop DFS, this error will occur. Make sure the HBase root directory does not currently exist or has been initialized by a previous run of HBase. Sure fire solution is to just use Hadoop dfs to delete the HBase root and let HBase create and initialize the directory itself.

Packet len6080218 is out of range!

If you have many regions on your cluster and you see an error like that reported above in this sections title in your logs, see HBASE-4246 Cluster with too many regions cannot withstand some master failover scenarios.

Master fails to become active due to lack of hsync for filesystem

HBase's internal framework for cluster operations requires the ability to durably save state in a write ahead log. When using a version of Apache Hadoop Common's filesystem API that supports checking on the availability of needed calls, HBase will proactively abort the cluster if it finds it can't operate safely.

For Master roles, the failure will show up in logs like this:

```
2018-04-05 11:18:44,653 ERROR [Thread-21] master.HMaster: Failed to become active
master
java.lang.IllegalStateException: The procedure WAL relies on the ability to hsync
for proper operation during component failures, but the underlying filesystem doe
s not support doing so. Please check the config value of 'hbase.procedure.store.w
al.use.hsync' to set the desired level of robustness and ensure the config value
of 'hbase.wal.dir' points to a FileSystem mount that can provide it.
        at org.apache.hadoop.hbase.procedure2.store.wal.WALProcedureStore.rollWri
ter(WALProcedureStore.java:1034)
        at org.apache.hadoop.hbase.procedure2.store.wal.WALProcedureStore.recover
Lease(WALProcedureStore.java:374)
        at org.apache.hadoop.hbase.procedure2.ProcedureExecutor.start(ProcedureEx
ecutor.java:530)
        at org.apache.hadoop.hbase.master.HMaster.startProcedureExecutor(HMaster.
java:1267)
        at org.apache.hadoop.hbase.master.HMaster.startServiceThreads(HMaster.jav
a:1173)
        at org.apache.hadoop.hbase.master.HMaster.finishActiveMasterInitializatio
n(HMaster.java:881)
        at org.apache.hadoop.hbase.master.HMaster.startActiveMasterManager(HMaste
r.java:2048)
        at org.apache.hadoop.hbase.master.HMaster.lambda$run$0(HMaster.java:568)
        at java.lang.Thread.run(Thread.java:745)
```

If you are attempting to run in standalone mode and see this error, please walk back through the section Quick Start - Standalone HBase and ensure you have included **all** the given configuration settings.

Shutdown Errors

ZooKeeper

Startup Errors

Could not find my address: xyz in list of ZooKeeper quorum servers

A ZooKeeper server wasn't able to start, throws that error. xyz is the name of your server.

This is a name lookup problem. HBase tries to start a ZooKeeper server on some machine but that machine isn't able to find itself in the `hbase.zookeeper.quorum` configuration.

Use the hostname presented in the error message instead of the value you used. If you have a DNS server, you can set `hbase.zookeeper.dns.interface` and `hbase.zookeeper.dns.nameserver` in *hbase-site.xml* to make sure it resolves to the correct FQDN.

ZooKeeper, The Cluster Canary

ZooKeeper is the cluster's "canary in the mineshaft". It'll be the first to notice issues if any so making sure its happy is the short-cut to a humming cluster.

See the ZooKeeper Operating Environment Troubleshooting page. It has suggestions and tools for checking disk and networking performance; i.e. the operating environment your ZooKeeper and HBase are running in.

Additionally, the utility **zkcli** may help investigate ZooKeeper issues.

Amazon EC2

ZooKeeper does not seem to work on Amazon EC2

HBase does not start when deployed as Amazon EC2 instances. Exceptions like the below appear in the Master and/or RegionServer logs:

```
  2009-10-19 11:52:27,030 INFO org.apache.zookeeper.ClientCnxn: Attempting
  connection to server ec2-174-129-15-236.compute-1.amazonaws.com/10.244.9.171:21
81
  2009-10-19 11:52:27,032 WARN org.apache.zookeeper.ClientCnxn: Exception
  closing session 0x0 to sun.nio.ch.SelectionKeyImpl@656dc861
  java.net.ConnectException: Connection refused
```

Security group policy is blocking the ZooKeeper port on a public address. Use the internal EC2 host names when configuring the ZooKeeper quorum peer list.

Instability on Amazon EC2

Questions on HBase and Amazon EC2 come up frequently on the HBase dist-list.

Remote Java Connection into EC2 Cluster Not Working

See Andrew's answer here, up on the user list: Remote Java client connection into EC2 instance.

HBase and Hadoop version issues

...cannot communicate with client version...

If you see something like the following in your logs ... 2012-09-24 10:20:52,168 FATAL org.apache.hadoop.hbase.master.HMaster: Unhandled exception. Starting shutdown. org.apache.hadoop.ipc.RemoteException: Server IPC version 7 cannot communicate with client version 4 ... ...are you trying to talk to an Hadoop 2.0.x from an HBase that has an Hadoop 1.0.x client? Use the HBase built against Hadoop 2.0 or rebuild your HBase passing the -Dhadoop.profile=2.0 attribute to Maven (See Building against various Hadoop versions for more).

HBase and HDFS

General configuration guidance for Apache HDFS is out of the scope of this guide. Refer to the documentation available at https://hadoop.apache.org/ for extensive information about configuring HDFS. This section deals with HDFS in terms of HBase.

In most cases, HBase stores its data in Apache HDFS. This includes the HFiles containing the data, as well as the write-ahead logs (WALs) which store data before it is written to the HFiles and protect against RegionServer crashes. HDFS provides reliability and protection to data in HBase because it is distributed. To operate with the most efficiency, HBase needs data to be available locally. Therefore, it is a good practice to run an HDFS DataNode on each RegionServer.

Important Information and Guidelines for HBase and HDFS

HBase is a client of HDFS.
HBase is an HDFS client, using the HDFS `DFSClient` class, and references to this class appear in HBase logs with other HDFS client log messages.

Configuration is necessary in multiple places.
Some HDFS configurations relating to HBase need to be done at the HDFS (server) side. Others must be done within HBase (at the client side). Other settings need to be set at both the server and client side.

Write errors which affect HBase may be logged in the HDFS logs rather than HBase logs.
When writing, HDFS pipelines communications from one DataNode to another. HBase communicates to both the HDFS NameNode and DataNode, using the HDFS client classes. Communication problems between DataNodes are logged in the HDFS logs, not the HBase logs.

HBase communicates with HDFS using two different ports.
HBase communicates with DataNodes using the `ipc.Client` interface and the `DataNode` class. References to these will appear in HBase logs. Each of these communication channels use a different port (50010 and 50020 by default). The ports are configured in the HDFS configuration, via the `dfs.datanode.address` and `dfs.datanode.ipc.address` parameters.

Errors may be logged in HBase, HDFS, or both.
When troubleshooting HDFS issues in HBase, check logs in both places for errors.

HDFS takes a while to mark a node as dead. You can configure HDFS to avoid using stale DataNodes.
By default, HDFS does not mark a node as dead until it is unreachable for 630 seconds. In Hadoop 1.1 and Hadoop 2.x, this can be alleviated by enabling checks for stale DataNodes, though this check is disabled by default. You can enable the check for reads and writes separately, via `dfs.namenode.avoid.read.stale.datanode` and `dfs.namenode.avoid.write.stale.datanode settings`. A stale DataNode is one that has not been reachable for `dfs.namenode.stale.datanode.interval` (default is 30 seconds). Stale datanodes are avoided, and marked as the last possible target for a read or write operation. For configuration details, see the HDFS documentation.

Settings for HDFS retries and timeouts are important to HBase.
You can configure settings for various retries and timeouts. Always refer to the HDFS documentation for current recommendations and defaults. Some of the settings important to HBase are listed here. Defaults are current as of Hadoop 2.3. Check the Hadoop documentation for the most current values and recommendations.

The HBase Balancer and HDFS Balancer are incompatible
The HDFS balancer attempts to spread HDFS blocks evenly among DataNodes. HBase relies on compactions to restore locality after a region split or failure. These two types of balancing do not work well together.

In the past, the generally accepted advice was to turn off the HDFS load balancer and rely on the HBase balancer, since the HDFS balancer would degrade locality. This advice is still valid if your HDFS version is lower than 2.7.1.

HDFS-6133 provides the ability to exclude favored-nodes (pinned) blocks from the HDFS load balancer, by setting the `dfs.datanode.block-pinning.enabled` property to `true` in the HDFS service configuration.

HBase can be enabled to use the HDFS favored-nodes feature by switching the HBase balancer class (conf: `hbase.master.loadbalancer.class`) to `org.apache.hadoop.hbase.favored.FavoredNodeLoadBalancer` which is documented here.

> HDFS-6133 is available in HDFS 2.7.0 and higher, but HBase does not support running on HDFS 2.7.0, so you must be using HDFS 2.7.1 or higher to use this feature with HBase.

Connection Timeouts

Connection timeouts occur between the client (HBASE) and the HDFS DataNode. They may occur when establishing a connection, attempting to read, or attempting to write. The two settings below are used in combination, and affect connections between the DFSClient and the DataNode, the ipc.cClient and the DataNode, and communication between two DataNodes.

`dfs.client.socket-timeout` (default: 60000)
The amount of time before a client connection times out when establishing a connection or reading. The value is expressed in milliseconds, so the default is 60 seconds.

`dfs.datanode.socket.write.timeout` (default: 480000)
The amount of time before a write operation times out. The default is 8 minutes, expressed as milliseconds.

Typical Error Logs

The following types of errors are often seen in the logs.

`INFO HDFS.DFSClient: Failed to connect to /xxx50010, add to deadNodes and continue java.net.SocketTimeoutException: 60000 millis timeout while waiting for channel to be ready for connect. ch : java.nio.channels.SocketChannel[connection-pending remote=/region-server-1:50010]`:: All DataNodes for a block are dead, and recovery is not possible. Here is the sequence of events that leads to this error:

`INFO org.apache.hadoop.HDFS.DFSClient: Exception in createBlockOutputStream java.net.SocketTimeoutException: 69000 millis timeout while waiting for channel to be ready for connect. ch : java.nio.channels.SocketChannel[connection-pending remote=/ xxx:50010]`:: This type of error indicates a write issue. In this case, the master wants to split the log. It does not have a local DataNodes so it tries to connect to a remote DataNode, but the DataNode is dead.

Running unit or integration tests

Runtime exceptions from MiniDFSCluster when running tests

If you see something like the following

```
...
java.lang.NullPointerException: null
at org.apache.hadoop.hdfs.MiniDFSCluster.startDataNodes
at org.apache.hadoop.hdfs.MiniDFSCluster.<init>
at org.apache.hadoop.hbase.MiniHBaseCluster.<init>
at org.apache.hadoop.hbase.HBaseTestingUtility.startMiniDFSCluster
at org.apache.hadoop.hbase.HBaseTestingUtility.startMiniCluster
...
```

or

```
...
```

```
java.io.IOException: Shutting down
at org.apache.hadoop.hbase.MiniHBaseCluster.init
at org.apache.hadoop.hbase.MiniHBaseCluster.<init>
at org.apache.hadoop.hbase.MiniHBaseCluster.<init>
at org.apache.hadoop.hbase.HBaseTestingUtility.startMiniHBaseCluster
at org.apache.hadoop.hbase.HBaseTestingUtility.startMiniCluster
...
```

... then try issuing the command umask 022 before launching tests. This is a workaround for HDFS-2556

Case Studies

For Performance and Troubleshooting Case Studies, see Apache HBase Case Studies.

Cryptographic Features

sun.security.pkcs11.wrapper.PKCS11Exception: CKR_ARGUMENTS_BAD

This problem manifests as exceptions ultimately caused by:

```
Caused by: sun.security.pkcs11.wrapper.PKCS11Exception: CKR_ARGUMENTS_BAD
  at sun.security.pkcs11.wrapper.PKCS11.C_DecryptUpdate(Native Method)
  at sun.security.pkcs11.P11Cipher.implDoFinal(P11Cipher.java:795)
```

This problem appears to affect some versions of OpenJDK 7 shipped by some Linux vendors. NSS is configured as the default provider. If the host has an x86_64 architecture, depending on if the vendor packages contain the defect, the NSS provider will not function correctly.

To work around this problem, find the JRE home directory and edit the file *lib/security/java.security*. Edit the file to comment out the line:

```
security.provider.1=sun.security.pkcs11.SunPKCS11 ${java.home}/lib/security/nss.c
fg
```

Then renumber the remaining providers accordingly.

Operating System Specific Issues

Page Allocation Failure

> This issue is known to affect CentOS 6.2 and possibly CentOS 6.5. It may also affect some versions of Red Hat Enterprise Linux, according to https://bugzilla.redhat.com/show_bug.cgi?id=770545.

Some users have reported seeing the following error:

```
kernel: java: page allocation failure. order:4, mode:0x20
```

Raising the value of `min_free_kbytes` was reported to fix this problem. This parameter is set to a percentage of the amount of RAM on your system, and is described in more detail at https://docs.kernel.org/admin-guide/sysctl/vm.html#min-free-kbytes.

To find the current value on your system, run the following command:

```
[user@host]# cat /proc/sys/vm/min_free_kbytes
```

Next, raise the value. Try doubling, then quadrupling the value. Note that setting the value too low or too high could have detrimental effects on your system. Consult your operating system vendor for specific recommendations.

Use the following command to modify the value of `min_free_kbytes`, substituting *VALUE* with your intended value:

```
[user@host]# echo <value> > /proc/sys/vm/min_free_kbytes
```

JDK Issues

NoSuchMethodError: java.util.concurrent.ConcurrentHashMap.keySet

If you see this in your logs:

```
Caused by: java.lang.NoSuchMethodError: java.util.concurrent.ConcurrentHashMap.ke
ySet()Ljava/util/concurrent/ConcurrentHashMap$KeySetView;
  at org.apache.hadoop.hbase.master.ServerManager.findServerWithSameHostnamePortW
ithLock(ServerManager.java:393)
  at org.apache.hadoop.hbase.master.ServerManager.checkAndRecordNewServer(ServerM
anager.java:307)
  at org.apache.hadoop.hbase.master.ServerManager.regionServerStartup(ServerManag
er.java:244)
  at org.apache.hadoop.hbase.master.MasterRpcServices.regionServerStartup(MasterR
pcServices.java:304)
  at org.apache.hadoop.hbase.protobuf.generated.RegionServerStatusProtos$RegionSe
rverStatusService$2.callBlockingMethod(RegionServerStatusProtos.java:7910)
  at org.apache.hadoop.hbase.ipc.RpcServer.call(RpcServer.java:2020)
  ... 4 more
```

then check if you compiled with jdk8 and tried to run it on jdk7. If so, this won't work. Run on jdk8 or recompile with jdk7. See HBASE-10607 JDK8 NoSuchMethodError involving ConcurrentHashMap.keySet if running on JRE 7.

Full gc caused by mslab when using G1

The default size of chunk used by mslab is 2MB, when using G1, if heapRegionSize equals 4MB, these chunks are allocated as humongous objects, exclusively allocating one region, then the remaining 2MB become memory fragment.

Lots of memory fragment may lead to full gc even if the percent of used heap not high enough.

The G1HeapRegionSize calculated by initial_heap_size and max_heap_size, here are some cases for better understand:

- xmx=10G → region size 2M
- xms=10G, xmx=10G → region size 4M
- xmx=20G → region size 4M
- xms=20G, xmx=20G → region size 8M
- xmx=30G → region size 4M
- xmx=32G → region size 8M

You can avoid this problem by reducing the chunk size a bit to 2047KB as below.

```
hbase.hregion.memstore.mslab.chunksize 2096128
```

Case Studies

Overview

This chapter will describe a variety of performance and troubleshooting case studies that can provide a useful blueprint on diagnosing Apache HBase cluster issues.

For more information on Performance and Troubleshooting, see Apache HBase Performance Tuning and Troubleshooting and Debugging Apache HBase.

Schema Design

See the schema design case studies here: Schema Design Case Studies

Performance/Troubleshooting

Case Study #1 (Performance Issue On A Single Node)

Scenario

Following a scheduled reboot, one data node began exhibiting unusual behavior. Routine MapReduce jobs run against HBase tables which regularly completed in five or six minutes began taking 30 or 40 minutes to finish. These jobs were consistently found to be waiting on map and reduce tasks assigned to the troubled data node (e.g., the slow map tasks all had the same Input Split). The situation came to a head during a distributed copy, when the copy was severely prolonged by the lagging node.

Hardware

Datanodes:

- Two 12-core processors
- Six Enterprise SATA disks
- 24GB of RAM
- Two bonded gigabit NICs

Network:

- 10 Gigabit top-of-rack switches
- 20 Gigabit bonded interconnects between racks.

Hypotheses

HBase "Hot Spot" Region

We hypothesized that we were experiencing a familiar point of pain: a "hot spot" region in an HBase table, where uneven key-space distribution can funnel a huge number of requests to a single HBase region, bombarding the RegionServer process and cause slow response time. Examination of the HBase Master status page showed that the number of HBase requests to the troubled node was almost zero. Further, examination of the HBase logs showed that there were no region splits, compactions, or other region transitions in progress. This effectively ruled out a "hot spot" as the root cause of the observed slowness.

HBase Region With Non-Local Data

Our next hypothesis was that one of the MapReduce tasks was requesting data from HBase that was not local to the DataNode, thus forcing HDFS to request data blocks from other servers over the network. Examination of the DataNode logs showed that there were very few blocks being requested over the network, indicating that the HBase region was correctly assigned, and that the majority of the necessary data was located on the node. This ruled out the possibility of non-local data causing a slowdown.

Excessive I/O Wait Due To Swapping Or An Over-Worked Or Failing Hard Disk

After concluding that the Hadoop and HBase were not likely to be the culprits, we moved on to troubleshooting the DataNode's hardware. Java, by design, will periodically scan its entire memory space to do garbage collection. If system memory is heavily overcommitted, the Linux kernel may enter a vicious cycle, using up all of its resources swapping Java heap back and forth from disk to RAM as Java tries to run garbage collection. Further, a failing hard disk will often retry reads and/or writes many times before giving up and returning an error. This can manifest as high iowait, as running processes wait for reads and writes to complete. Finally, a disk nearing the upper edge of its performance envelope will begin to cause iowait as it informs the kernel that it cannot accept any more data, and the kernel queues incoming data into the dirty write pool in memory. However, using `vmstat(1)` and `free(1)`, we could see that no swap was being used, and the amount of disk IO was only a few kilobytes per second.

Slowness Due To High Processor Usage

Next, we checked to see whether the system was performing slowly simply due to very high computational load. `top(1)` showed that the system load was higher than normal, but `vmstat(1)` and `mpstat(1)` showed that the amount of processor being used for actual computation was low.

Network Saturation (The Winner)

Since neither the disks nor the processors were being utilized heavily, we moved on to the performance of the network interfaces. The DataNode had two gigabit ethernet adapters, bonded to form an active-standby interface. `ifconfig(8)` showed some unusual anomalies, namely interface errors, overruns, framing errors. While not unheard of, these kinds of errors are exceedingly rare on modern hardware which is operating as it should:

```
$ /sbin/ifconfig bond0
bond0  Link encap:Ethernet  HWaddr 00:00:00:00:00:00
inet addr:10.x.x.x  Bcast:10.x.x.255  Mask:255.255.255.0
UP BROADCAST RUNNING MASTER MULTICAST  MTU:1500  Metric:1
RX packets:2990700159 errors:12 dropped:0 overruns:1 frame:6          <--- Look H
ere! Errors!
TX packets:3443518196 errors:0 dropped:0 overruns:0 carrier:0
collisions:0 txqueuelen:0
RX bytes:2416328868676 (2.4 TB)  TX bytes:3464991094001 (3.4 TB)
```

These errors immediately lead us to suspect that one or more of the ethernet interfaces might have negotiated the wrong line speed. This was confirmed both by running an ICMP ping from an external host and observing round-trip-time in excess of 700ms, and by running `ethtool(8)` on the members of the bond interface and discovering that the active interface was operating at 100Mbs/, full duplex.

```
$ sudo ethtool eth0
Settings for eth0:
Supported ports: [ TP ]
Supported link modes:   10baseT/Half 10baseT/Full
                        100baseT/Half 100baseT/Full
                        1000baseT/Full
Supports auto-negotiation: Yes
Advertised link modes:  10baseT/Half 10baseT/Full
                        100baseT/Half 100baseT/Full
                        1000baseT/Full
Advertised pause frame use: No
Advertised auto-negotiation: Yes
Link partner advertised link modes:  Not reported
Link partner advertised pause frame use: No
Link partner advertised auto-negotiation: No
```

```
Speed: 100Mb/s                                <--- Look Here!  Should say 10
00Mb/s!
Duplex: Full
Port: Twisted Pair
PHYAD: 1
Transceiver: internal
Auto-negotiation: on
MDI-X: Unknown
Supports Wake-on: umbg
Wake-on: g
Current message level: 0x00000003 (3)
Link detected: yes
```

In normal operation, the ICMP ping round trip time should be around 20ms, and the interface speed and duplex should read, "1000MB/s", and, "Full", respectively.

Resolution

After determining that the active ethernet adapter was at the incorrect speed, we used the `ifenslave(8)` command to make the standby interface the active interface, which yielded an immediate improvement in MapReduce performance, and a 10 times improvement in network throughput:

On the next trip to the datacenter, we determined that the line speed issue was ultimately caused by a bad network cable, which was replaced.

Case Study #2 (Performance Research 2012)

Investigation results of a self-described "we're not sure what's wrong, but it seems slow" problem. http://gbif.blogspot.com/2012/03/hbase-performance-evaluation-continued.html

Case Study #3 (Performance Research 2010)

Investigation results of general cluster performance from 2010. Although this research is on an older version of the codebase, this writeup is still very useful in terms of approach. https://web.archive.org/web/20180503124332/http://hstack.org/hbase-performance-testing/

Case Study #4 (max.transfer.threads Config)

Case study of configuring `max.transfer.threads` (previously known as `xcievers`) and diagnosing errors from misconfigurations. http://www.larsgeorge.com/2012/03/hadoop-

hbase-and-xceivers.html

See also `dfs.datanode.max.transfer.threads`.

Operational Management

The subject of operations is related to the topics of Troubleshooting and Debugging Apache HBase, Apache HBase Performance Tuning, and Apache HBase Configuration but is a distinct topic in itself.

HBase Tools and Utilities

HBase provides several tools for administration, analysis, and debugging of your cluster. The entry-point to most of these tools is the *bin/hbase* command, though some tools are available in the *dev-support/* directory.

To see usage instructions for *bin/hbase* command, run it with no arguments, or with the `-h` argument. These are the usage instructions for HBase 0.98.x. Some commands, such as `version`, `pe`, `ltt`, `clean`, are not available in previous versions.

```
$ bin/hbase
Usage: hbase [<options>] <command> [<args>]
Options:
  --config DIR     Configuration direction to use. Default: ./conf
  --hosts HOSTS    Override the list in 'regionservers' file
  --auth-as-server Authenticate to ZooKeeper using servers configuration

Commands:
Some commands take arguments. Pass no args or -h for usage.
  shell           Run the HBase shell
  hbck            Run the HBase 'fsck' tool. Defaults read-only hbck1.
                  Pass '-j /path/to/HBCK2.jar' to run hbase-2.x HBCK2.
  snapshot        Tool for managing snapshots
  wal             Write-ahead-log analyzer
  hfile           Store file analyzer
  zkcli           Run the ZooKeeper shell
  master          Run an HBase HMaster node
  regionserver    Run an HBase HRegionServer node
  zookeeper       Run a ZooKeeper server
  rest            Run an HBase REST server
  thrift          Run the HBase Thrift server
  thrift2         Run the HBase Thrift2 server
  clean           Run the HBase clean up script
  jshell          Run a jshell with HBase on the classpath
  classpath       Dump hbase CLASSPATH
  mapredcp        Dump CLASSPATH entries required by mapreduce
  pe              Run PerformanceEvaluation
  ltt             Run LoadTestTool
  canary          Run the Canary tool
  version         Print the version
```

```
  backup           Backup tables for recovery
  restore          Restore tables from existing backup image
  regionsplitter   Run RegionSplitter tool
  rowcounter       Run RowCounter tool
  cellcounter      Run CellCounter tool
  CLASSNAME        Run the class named CLASSNAME
```

Some of the tools and utilities below are Java classes which are passed directly to the *bin/hbase* command, as referred to in the last line of the usage instructions. Others, such as `hbase shell` (The Apache HBase Shell), `hbase upgrade` (Upgrading), and `hbase thrift` (Thrift API and Filter Language), are documented elsewhere in this guide.

Canary

The Canary tool can help users "canary-test" the HBase cluster status. The default "region mode" fetches a row from every column-family of every regions. In "regionserver mode", the Canary tool will fetch a row from a random region on each of the cluster's RegionServers. In "zookeeper mode", the Canary will read the root znode on each member of the zookeeper ensemble.

To see usage, pass the `-help` parameter (if you pass no parameters, the Canary tool starts executing in the default region "mode" fetching a row from every region in the cluster).

```
2018-10-16 13:11:27,037 INFO  [main] tool.Canary: Execution thread count=16
Usage: canary [OPTIONS] [<TABLE1> [<TABLE2]...] | [<REGIONSERVER1> [<REGIONSERVER
2]..]
Where [OPTIONS] are:
 -h,-help        show this help and exit.
 -regionserver   set 'regionserver mode'; gets row from random region on server
 -allRegions     get from ALL regions when 'regionserver mode', not just random o
ne.
 -zookeeper      set 'zookeeper mode'; grab zookeeper.znode.parent on each ensemb
le member
 -daemon         continuous check at defined intervals.
 -interval <N>   interval between checks in seconds
 -e              consider table/regionserver argument as regular expression
 -f <B>          exit on first error; default=true
 -failureAsError treat read/write failure as error
 -t <N>          timeout for canary-test run; default=600000ms
 -writeSniffing  enable write sniffing
 -writeTable     the table used for write sniffing; default=hbase:canary
 -writeTableTimeout <N>  timeout for writeTable; default=600000ms
 -readTableTimeouts <tableName>=<read timeout>,<tableName>=<read timeout>,...
            comma-separated list of table read timeouts (no spaces);
            logs 'ERROR' if takes longer. default=600000ms
 -permittedZookeeperFailures <N>  Ignore first N failures attempting to
```

```
            connect to individual zookeeper nodes in ensemble

 -D<configProperty>=<value> to assign or override configuration params
 -Dhbase.canary.read.raw.enabled=<true/false> Set to enable/disable raw scan; def
ault=false

Canary runs in one of three modes: region (default), regionserver, or zookeeper.
To sniff/probe all regions, pass no arguments.
To sniff/probe all regions of a table, pass tablename.
To sniff/probe regionservers, pass -regionserver, etc.
See https://hbase.apache.org/docs/operational-management/tools#canary for Canary
documentation.
```

The `Sink` class is instantiated using the `hbase.canary.sink.class` configuration property.

This tool will return non zero error codes to user for collaborating with other monitoring tools, such as Nagios. The error code definitions are:

```
private static final int USAGE_EXIT_CODE = 1;
private static final int INIT_ERROR_EXIT_CODE = 2;
private static final int TIMEOUT_ERROR_EXIT_CODE = 3;
private static final int ERROR_EXIT_CODE = 4;
private static final int FAILURE_EXIT_CODE = 5;
```

Here are some examples based on the following given case: given two Table objects called test-01 and test-02 each with two column family cf1 and cf2 respectively, deployed on 3 RegionServers. See the following table.

RegionServer	test-01	test-02
rs1	r1	r2
rs2	r2	
rs3	r2	r1

Following are some example outputs based on the previous given case.

Canary test for every column family (store) of every region of every table

```
$ ${HBASE_HOME}/bin/hbase canary
```

```
3/12/09 03:26:32 INFO tool.Canary: read from region test-01,,1386230156732.0e3c7d
77ffb6361ea1b996ac1042ca9a. column family cf1 in 2ms
13/12/09 03:26:32 INFO tool.Canary: read from region test-01,,1386230156732.0e3c7
d77ffb6361ea1b996ac1042ca9a. column family cf2 in 2ms
13/12/09 03:26:32 INFO tool.Canary: read from region test-01,0004883,138623015673
2.87b55e03dfeade00f441125159f8ca87. column family cf1 in 4ms
13/12/09 03:26:32 INFO tool.Canary: read from region test-01,0004883,138623015673
2.87b55e03dfeade00f441125159f8ca87. column family cf2 in 1ms
...
13/12/09 03:26:32 INFO tool.Canary: read from region test-02,,1386559511167.aa295
1a86289281beee480f107bb36ee. column family cf1 in 5ms
13/12/09 03:26:32 INFO tool.Canary: read from region test-02,,1386559511167.aa295
1a86289281beee480f107bb36ee. column family cf2 in 3ms
13/12/09 03:26:32 INFO tool.Canary: read from region test-02,0004883,138655951116
7.cbda32d5e2e276520712d84eaaa29d84. column family cf1 in 31ms
13/12/09 03:26:32 INFO tool.Canary: read from region test-02,0004883,138655951116
7.cbda32d5e2e276520712d84eaaa29d84. column family cf2 in 8ms
```

So you can see, table test-01 has two regions and two column families, so the Canary tool in the default "region mode" will pick 4 small piece of data from 4 (2 region * 2 store) different stores. This is a default behavior.

Canary test for every column family (store) of every region of a specific table(s)

You can also test one or more specific tables by passing table names.

```
$ ${HBASE_HOME}/bin/hbase canary test-01 test-02
```

Canary test with RegionServer granularity

In "regionserver mode", the Canary tool will pick one small piece of data from each RegionServer (You can also pass one or more RegionServer names as arguments to the canary-test when in "regionserver mode").

```
$ ${HBASE_HOME}/bin/hbase canary -regionserver

13/12/09 06:05:17 INFO tool.Canary: Read from table:test-01 on region server:rs2
in 72ms
13/12/09 06:05:17 INFO tool.Canary: Read from table:test-02 on region server:rs3
in 34ms
13/12/09 06:05:17 INFO tool.Canary: Read from table:test-01 on region server:rs1
in 56ms
```

Canary test with regular expression pattern

You can pass regexes for table names when in "region mode" or for servernames when in "regionserver mode". The below will test both table test-01 and test-02.

```
$ ${HBASE_HOME}/bin/hbase canary -e test-0[1-2]
```

Run canary test as a "daemon"

Run repeatedly with an interval defined via the option `-interval` (default value is 60 seconds). This daemon will stop itself and return non-zero error code if any error occur. To have the daemon keep running across errors, pass the -f flag with its value set to false (see usage above).

```
$ ${HBASE_HOME}/bin/hbase canary -daemon
```

To run repeatedly with 5 second intervals and not stop on errors, do the following.

```
$ ${HBASE_HOME}/bin/hbase canary -daemon -interval 5 -f false
```

Force timeout if canary test stuck

In some cases the request is stuck and no response is sent back to the client. This can happen with dead RegionServers which the master has not yet noticed. Because of this we provide a timeout option to kill the canary test and return a non-zero error code. The below sets the timeout value to 60 seconds (the default value is 600 seconds).

```
$ ${HBASE_HOME}/bin/hbase canary -t 60000
```

Enable write sniffing in canary

By default, the canary tool only checks read operations. To enable the write sniffing, you can run the canary with the `-writeSniffing` option set. When write sniffing is enabled, the canary tool will create an hbase table and make sure the regions of the table are distributed to all region servers. In each sniffing period, the canary will try to put data to these regions to check the write availability of each region server.

```
$ ${HBASE_HOME}/bin/hbase canary -writeSniffing
```

The default write table is `hbase:canary` and can be specified with the option `-writeTable`.

```
$ ${HBASE_HOME}/bin/hbase canary -writeSniffing -writeTable ns:canary
```

The default value size of each put is 10 bytes. You can set it via the config key: `hbase.canary.write.value.size`.

Treat read / write failure as error

By default, the canary tool only logs read failures — due to e.g. RetriesExhaustedException, etc. — and will return the 'normal' exit code. To treat read/write failure as errors, you can run canary with the `-treatFailureAsError` option. When enabled, read/write failures will result in an error exit code.

```
$ ${HBASE_HOME}/bin/hbase canary -treatFailureAsError
```

Running Canary in a Kerberos-enabled Cluster

To run the Canary in a Kerberos-enabled cluster, configure the following two properties in *hbase-site.xml*:

- `hbase.client.keytab.file`
- `hbase.client.kerberos.principal`

Kerberos credentials are refreshed every 30 seconds when Canary runs in daemon mode.

To configure the DNS interface for the client, configure the following optional properties in *hbase-site.xml*.

- `hbase.client.dns.interface`
- `hbase.client.dns.nameserver`

Example Canary in a Kerberos-Enabled Cluster

This example shows each of the properties with valid values.

```
<property>
  <name>hbase.client.kerberos.principal</name>
  <value>hbase/_HOST@YOUR-REALM.COM</value>
</property>
<property>
  <name>hbase.client.keytab.file</name>
  <value>/etc/hbase/conf/keytab.krb5</value>
</property>

<property>
  <name>hbase.client.dns.interface</name>
  <value>default</value>
</property>
<property>
  <name>hbase.client.dns.nameserver</name>
  <value>default</value>
</property>
```

RegionSplitter

```
usage: bin/hbase regionsplitter <TABLE> <SPLITALGORITHM>
SPLITALGORITHM is the java class name of a class implementing
                      SplitAlgorithm, or one of the special strings
                      HexStringSplit or DecimalStringSplit or
                      UniformSplit, which are built-in split algorithms.
                      HexStringSplit treats keys as hexadecimal ASCII, and
                      DecimalStringSplit treats keys as decimal ASCII, and
                      UniformSplit treats keys as arbitrary bytes.
 -c <region count>         Create a new table with a pre-split number of
                           regions
 -D <property=value>       Override HBase Configuration Settings
 -f <family:family:...>    Column Families to create with new table.
                           Required with -c
    --firstrow <arg>       First Row in Table for Split Algorithm
 -h                        Print this usage help
    --lastrow <arg>        Last Row in Table for Split Algorithm
 -o <count>                Max outstanding splits that have unfinished
                           major compactions
 -r                        Perform a rolling split of an existing region
    --risky                Skip verification steps to complete
                           quickly. STRONGLY DISCOURAGED for production
                           systems.
```

For additional detail, see Manual Region Splitting.

Health Checker

You can configure HBase to run a script periodically and if it fails N times (configurable), have the server exit. See *HBASE-7351 Periodic health check script* for configurations and detail.

Driver

Several frequently-accessed utilities are provided as `Driver` classes, and executed by the *bin/hbase* command. These utilities represent MapReduce jobs which run on your cluster. They are run in the following way, replacing *UtilityName* with the utility you want to run. This command assumes you have set the environment variable `HBASE_HOME` to the directory where HBase is unpacked on your server.

```
${HBASE_HOME}/bin/hbase org.apache.hadoop.hbase.mapreduce.UtilityName
```

The following utilities are available:

`LoadIncrementalHFiles`
Complete a bulk data load.

`CopyTable`
Export a table from the local cluster to a peer cluster.

`Export`
Write table data to HDFS.

`Import`
Import data written by a previous `Export` operation.

`ImportTsv`
Import data in TSV format.

`RowCounter`
Count rows in an HBase table.

`CellCounter`

Count cells in an HBase table.

`replication.VerifyReplication`

Compare the data from tables in two different clusters. WARNING: It doesn't work for incrementColumnValues'd cells since the timestamp is changed. Note that this command is in a different package than the others.

Each command except `RowCounter` and `CellCounter` accept a single `--help` argument to print usage instructions.

HBase `hbck`

The `hbck` tool that shipped with hbase-1.x has been made read-only in hbase-2.x. It is not able to repair hbase-2.x clusters as hbase internals have changed. Nor should its assessments in read-only mode be trusted as it does not understand hbase-2.x operation.

A new tool, HBase `HBCK2`, described in the next section, replaces `hbck`.

HBase `HBCK2`

`HBCK2` is the successor to HBase `HBCK`, the hbase-1.x fix tool (A.K.A `hbck1`). Use it in place of `hbck1` making repairs against hbase-2.x installs.

`HBCK2` does not ship as part of hbase. It can be found as a subproject of the companion hbase-operator-tools repository at Apache HBase HBCK2 Tool. `HBCK2` was moved out of hbase so it could evolve at a cadence apart from that of hbase core.

See the HBCK2 Home Page for how `HBCK2` differs from `hbck1`, and for how to build and use it.

Once built, you can run `HBCK2` as follows:

```
$ hbase hbck -j /path/to/HBCK2.jar
```

This will generate `HBCK2` usage describing commands and options.

HFile Tool

See HFile Tool.

WAL Tools

For bulk replaying WAL files or *recovered.edits* files, see WALPlayer. For reading/verifying individual files, read on.

WALPrettyPrinter

The `WALPrettyPrinter` is a tool with configurable options to print the contents of a WAL or a *recovered.edits* file. You can invoke it via the HBase cli with the 'wal' command.

```
 $ ./bin/hbase wal hdfs://example.org:9000/hbase/WALs/example.org,60020,128351629
3161/10.10.21.10%3A60020.1283973724012
```

> **WAL Printing in older versions of HBase**
>
> Prior to version 2.0, the `WALPrettyPrinter` was called the `HLogPrettyPrinter`, after an internal name for HBase's write ahead log. In those versions, you can print the contents of a WAL using the same configuration as above, but with the 'hlog' command.
>
> ```
>  $ ./bin/hbase hlog hdfs://example.org:9000/hbase/.logs/example.org,60020,
> 1283516293161/10.10.21.10%3A60020.1283973724012
> ```

Compression Tool

See compression.test.

CopyTable

CopyTable is a utility that can copy part or of all of a table, either to the same cluster or another cluster. The target table must first exist. The usage is as follows:

```
$ ./bin/hbase org.apache.hadoop.hbase.mapreduce.CopyTable --help
/bin/hbase org.apache.hadoop.hbase.mapreduce.CopyTable --help
```

```
Usage: CopyTable [general options] [--starttime=X] [--endtime=Y] [--new.name=NEW]
[--peer.adr=ADR] <tablename>

Options:
 rs.class     hbase.regionserver.class of the peer cluster,
              specify if different from current cluster
 rs.impl      hbase.regionserver.impl of the peer cluster,
 startrow     the start row
 stoprow      the stop row
 starttime    beginning of the time range (unixtime in millis)
              without endtime means from starttime to forever
 endtime      end of the time range.  Ignored if no starttime specified.
 versions     number of cell versions to copy
 new.name     new table's name
 peer.uri     The URI of the peer cluster
 peer.adr     Address of the peer cluster given in the format
              hbase.zookeeer.quorum:hbase.zookeeper.client.port:zookeeper.znode.p
arent
              Do not take effect if peer.uri is specified
              Deprecated, please use peer.uri instead
 families     comma-separated list of families to copy
              To copy from cf1 to cf2, give sourceCfName:destCfName.
              To keep the same name, just give "cfName"
 all.cells    also copy delete markers and deleted cells

Args:
 tablename    Name of the table to copy

Examples:
 To copy 'TestTable' to a cluster that uses replication for a 1 hour window:
 $ bin/hbase org.apache.hadoop.hbase.mapreduce.CopyTable --starttime=126587519428
9 --endtime=1265878794289 --peer.adr=server1,server2,server3:2181:/hbase --famili
es=myOldCf:myNewCf,cf2,cf3 TestTable

For performance consider the following general options:
  It is recommended that you set the following to >=100. A higher value uses more
memory but
  decreases the round trip time to the server and may increase performance.
    -Dhbase.client.scanner.caching=100
  The following should always be set to false, to prevent writing data twice, whi
ch may produce
  inaccurate results.
    -Dmapred.map.tasks.speculative.execution=false
```

Starting from 3.0.0, we introduce a `peer.uri` option so the `peer.adr` option is deprecated. Please use connection URI for specifying HBase clusters. For all previous versions, you should still use the `peer.adr` option.

> **Scanner Caching**
>
> Caching for the input Scan is configured via `hbase.client.scanner.caching` in the job configuration.

Versions

By default, CopyTable utility only copies the latest version of row cells unless `--versions=n` is explicitly specified in the command.

Data Load

CopyTable does not perform a diff, it copies all Cells in between the specified startrow/stoprow starttime/endtime range. This means that already existing cells with same values will still be copied.

See Jonathan Hsieh's Online HBase Backups with CopyTable blog post for more on `CopyTable`.

HashTable/SyncTable

HashTable/SyncTable is a two steps tool for synchronizing table data, where each of the steps are implemented as MapReduce jobs. Similarly to CopyTable, it can be used for partial or entire table data syncing, under same or remote cluster. However, it performs the sync in a more efficient way than CopyTable. Instead of copying all cells in specified row key/time period range, HashTable (the first step) creates hashed indexes for batch of cells on source table and output those as results. On the next stage, SyncTable scans the source table and now calculates hash indexes for table cells, compares these hashes with the outputs of HashTable, then it just scans (and compares) cells for diverging hashes, only updating mismatching cells. This results in less network traffic/data transfers, which can be impacting when syncing large tables on remote clusters.

Step 1, HashTable

First, run HashTable on the source table cluster (this is the table whose state will be copied to its counterpart).

Usage:

```
$ ./bin/hbase org.apache.hadoop.hbase.mapreduce.HashTable --help
Usage: HashTable [options] <tablename> <outputpath>

Options:
 batchsize         the target amount of bytes to hash in each batch
                   rows are added to the batch until this size is reached
                   (defaults to 8000 bytes)
```

```
 numhashfiles      the number of hash files to create
                   if set to fewer than number of regions then
                   the job will create this number of reducers
                   (defaults to 1/100 of regions — at least 1)
 startrow          the start row
 stoprow           the stop row
 starttime         beginning of the time range (unixtime in millis)
                   without endtime means from starttime to forever
 endtime           end of the time range.  Ignored if no starttime specified.
 scanbatch         scanner batch size to support intra row scans
 versions          number of cell versions to include
 families          comma-separated list of families to include
 ignoreTimestamps  if true, ignores cell timestamps

Args:
 tablename     Name of the table to hash
 outputpath    Filesystem path to put the output data

Examples:
 To hash 'TestTable' in 32kB batches for a 1 hour window into 50 files:
 $ bin/hbase org.apache.hadoop.hbase.mapreduce.HashTable --batchsize=32000 --numh
ashfiles=50 --starttime=1265875194289 --endtime=1265878794289 --families=cf2,cf3
TestTable /hashes/testTable
```

The **batchsize** property defines how much cell data for a given region will be hashed together in a single hash value. Sizing this properly has a direct impact on the sync efficiency, as it may lead to less scans executed by mapper tasks of SyncTable (the next step in the process). The rule of thumb is that, the smaller the number of cells out of sync (lower probability of finding a diff), larger batch size values can be determined.

Step 2, SyncTable

Once HashTable has completed on source cluster, SyncTable can be ran on target cluster. Just like replication and other synchronization jobs, it requires that all RegionServers/DataNodes on source cluster be accessible by NodeManagers on the target cluster (where SyncTable job tasks will be running).

Usage:

```
$ ./bin/hbase org.apache.hadoop.hbase.mapreduce.SyncTable --help
Usage: SyncTable [options] <sourcehashdir> <sourcetable> <targettable>

Options:
 sourceuri        Cluster connection uri of the source table
                  (defaults to cluster in classpath's config)
 sourcezkcluster  ZK cluster key of the source table
                  (defaults to cluster in classpath's config)
                  Do not take effect if sourceuri is specifie
```

```
                  Deprecated, please use sourceuri instead
 targeturi        Cluster connection uri of the target table
                  (defaults to cluster in classpath's config)
 targetzkcluster  ZK cluster key of the target table
                  (defaults to cluster in classpath's config)
                  Do not take effect if targeturi is specified
                  Deprecated, please use targeturi instead
 dryrun           if true, output counters but no writes
                  (defaults to false)
 doDeletes        if false, does not perform deletes
                  (defaults to true)
 doPuts           if false, does not perform puts
                  (defaults to true)
 ignoreTimestamps if true, ignores cells timestamps while comparing
                  cell values. Any missing cell on target then gets
                  added with current time as timestamp
                  (defaults to false)

Args:
 sourcehashdir    path to HashTable output dir for source table
                  (see org.apache.hadoop.hbase.mapreduce.HashTable)
 sourcetable      Name of the source table to sync from
 targettable      Name of the target table to sync to

Examples:
 For a dry run SyncTable of tableA from a remote source cluster
 to a local target cluster:
 $ bin/hbase org.apache.hadoop.hbase.mapreduce.SyncTable --dryrun=true --sourcezk
cluster=zk1.example.com,zk2.example.com,zk3.example.com:2181:/hbase hdfs://nn:900
0/hashes/tableA tableA tableA
```

Starting from 3.0.0, we introduce `sourceuri` and `targeturi` options so `sourcezkcluster` and `targetzkcluster` are deprecated. Please use connection URI for specifying HBase clusters. For all previous versions, you should still use `sourcezkcluster` and `targetzkcluster`.

Cell comparison takes ROW/FAMILY/QUALIFIER/TIMESTAMP/VALUE into account for equality. When syncing at the target, missing cells will be added with original timestamp value from source. That may cause unexpected results after SyncTable completes, for example, if missing cells on target have a delete marker with a timestamp T2 (say, a bulk delete performed by mistake), but source cells timestamps have an older value T1, then those cells would still be unavailable at target because of the newer delete marker timestamp. Since cell timestamps might not be relevant to all use cases, *ignoreTimestamps* option adds the flexibility to avoid using cells timestamp in the comparison. When using *ignoreTimestamps* set to true, this option must be specified for both HashTable and SyncTable steps.

The **dryrun** option is useful when a read only, diff report is wanted, as it will produce only COUNTERS indicating the differences, but will not perform any actual changes. It can be used as an alternative to VerifyReplication tool.

By default, SyncTable will cause target table to become an exact copy of source table (at least, for the specified startrow/stoprow or/and starttime/endtime).

Setting doDeletes to false modifies default behaviour to not delete target cells that are missing on source. Similarly, setting doPuts to false modifies default behaviour to not add missing cells on target. Setting both doDeletes and doPuts to false would give same effect as setting dryrun to true.

Additional info on doDeletes/doPuts

"doDeletes/doPuts" were only added by HBASE-20305, so these may not be available on all released versions. For major 1.x versions, minimum minor release including it is **1.4.10**. For major 2.x versions, minimum minor release including it is **2.1.5**.

Additional info on ignoreTimestamps

"ignoreTimestamps" was only added by HBASE-24302, so it may not be available on all released versions. For major 1.x versions, minimum minor release including it is **1.4.14**. For major 2.x versions, minimum minor release including it is **2.2.5**.

Set doDeletes to false on Two-Way Replication scenarios

On Two-Way Replication or other scenarios where both source and target clusters can have data ingested, it's advisable to always set doDeletes option to false, as any additional cell inserted on SyncTable target cluster and not yet replicated to source would be deleted, and potentially lost permanently.

Set sourcezkcluster to the actual source cluster ZK quorum

Although not required, if sourcezkcluster is not set, SyncTable will connect to local HBase cluster for both source and target, which does not give any meaningful result.

Remote Clusters on different Kerberos Realms

Often, remote clusters may be deployed on different Kerberos Realms. HBASE-20586 added SyncTable support for cross realm authentication, allowing a SyncTable process running on target cluster to connect to source cluster and read both HashTable output files and the given HBase table when performing the required comparisons.

Export

Export is a utility that will dump the contents of table to HDFS in a sequence file. The Export can be run via a Coprocessor Endpoint or MapReduce. Invoke via:

mapreduce-based Export

```
$ bin/hbase org.apache.hadoop.hbase.mapreduce.Export TABLENAME OUTPUTDIR [VERSION
S [STARTTIME [ENDTIME]]]
```

endpoint-based Export

> Make sure the Export coprocessor is enabled by adding `org.apache.hadoop.hbase.coprocessor.Export` to `hbase.coprocessor.region.classes`.

```
$ bin/hbase org.apache.hadoop.hbase.coprocessor.Export TABLENAME OUTPUTDIR [VERSI
ONS [STARTTIME [ENDTIME]]]
```

The outputdir is a HDFS directory that does not exist prior to the export. When done, the exported files will be owned by the user invoking the export command.

The Comparison of Endpoint-based Export And Mapreduce-based Export

	Endpoint-based Export	Mapreduce-based Export
HBase version requirement	2.0+	0.2.1+
Maven dependency	hbase-endpoint	hbase-mapreduce (2.0+), hbase-server(prior to 2.0)
Requirement before dump	mount the endpoint.Export on the target table	deploy the MapReduce framework
Read latency	low, directly read the data from region	normal, traditional RPC scan
Read Scalability	depend on number of regions	depend on number of mappers (see TableInputFormatBase#getSplits)
Timeout	operation timeout. configured by hbase.client.operation.timeout	scan timeout. configured by hbase.client.scanner.timeout.period

	Endpoint-based Export	Mapreduce-based Export
Permission requirement	READ, EXECUTE	READ
Fault tolerance	no	depend on MapReduce

> To see usage instructions, run the command with no options. Available options include specifying column families and applying filters during the export.

By default, the `Export` tool only exports the newest version of a given cell, regardless of the number of versions stored. To export more than one version, replace *<versions>* with the desired number of versions.

For mapreduce based Export, if you want to export cell tags then set the following config property `hbase.client.rpc.codec` to `org.apache.hadoop.hbase.codec.KeyValueCodecWithTags`

Note: caching for the input Scan is configured via `hbase.client.scanner.caching` in the job configuration.

Import

Import is a utility that will load data that has been exported back into HBase. Invoke via:

```
$ bin/hbase -Dhbase.import.version=0.94 org.apache.hadoop.hbase.mapreduce.Import
<tablename> <inputdir>
```

> To see usage instructions, run the command with no options.

To import 0.94 exported files in a 0.96 cluster or onwards, you need to set system property "hbase.import.version" when running the import command as below:

```
$ bin/hbase -Dhbase.import.version=0.94 org.apache.hadoop.hbase.mapreduce.Import
TABLENAME INPUTDIR
```

If you want to import cell tags then set the following config property `hbase.client.rpc.codec` to `org.apache.hadoop.hbase.codec.KeyValueCodecWithTags`

ImportTsv

ImportTsv is a utility that will load data in TSV format into HBase. It has two distinct usages: loading data from TSV format in HDFS into HBase via Puts, and preparing StoreFiles to be loaded via the `completebulkload`.

To load data via Puts (i.e., non-bulk loading):

```
$ bin/hbase org.apache.hadoop.hbase.mapreduce.ImportTsv -Dimporttsv.columns=a,b,c
<tablename> <hdfs-inputdir>
```

To generate StoreFiles for bulk-loading:

```
$ bin/hbase org.apache.hadoop.hbase.mapreduce.ImportTsv -Dimporttsv.columns=a,b,c
-Dimporttsv.bulk.output=hdfs://storefile-outputdir <tablename> <hdfs-data-inputdi
r>
```

These generated StoreFiles can be loaded into HBase via completebulkload.

ImportTsv Options

Running `ImportTsv` with no arguments prints brief usage information:

```
Usage: importtsv -Dimporttsv.columns=a,b,c TABLENAME INPUTDIR

Imports the given input directory of TSV data into the specified table.

The column names of the TSV data must be specified using the -Dimporttsv.columns
option. This option takes the form of comma-separated column names, where each
column name is either a simple column family, or a columnfamily:qualifier. The sp
ecial
column name HBASE_ROW_KEY is used to designate that this column should be used
as the row key for each imported record. You must specify exactly one column
to be the row key, and you must specify a column name for every column that exist
s in the
input data.

By default importtsv will load data directly into HBase. To instead generate
HFiles of data to prepare for a bulk data load, pass the option:
  -Dimporttsv.bulk.output=/path/for/output
  Note: the target table will be created with default column family descriptors i
f it does not already exist.

Other options that may be specified with -D include:
  -Dimporttsv.skip.bad.lines=false - fail if encountering an invalid line
```

```
  '-Dimporttsv.separator=|' - eg separate on pipes instead of tabs
  -Dimporttsv.timestamp=currentTimeAsLong - use the specified timestamp for the i
mport
  -Dimporttsv.mapper.class=my.Mapper - A user-defined Mapper to use instead of or
g.apache.hadoop.hbase.mapreduce.TsvImporterMapper
```

ImportTsv Example

For example, assume that we are loading data into a table called 'datatsv' with a ColumnFamily called 'd' with two columns "c1" and "c2".

Assume that an input file exists as follows:

```
row1    c1  c2
row2    c1  c2
row3    c1  c2
row4    c1  c2
row5    c1  c2
row6    c1  c2
row7    c1  c2
row8    c1  c2
row9    c1  c2
row10   c1  c2
```

For ImportTsv to use this input file, the command line needs to look like this:

```
 HADOOP_CLASSPATH=`${HBASE_HOME}/bin/hbase classpath` ${HADOOP_HOME}/bin/hadoop j
ar ${HBASE_HOME}/hbase-mapreduce-VERSION.jar importtsv -Dimporttsv.columns=HBASE_
ROW_KEY,d:c1,d:c2 -Dimporttsv.bulk.output=hdfs://storefileoutput datatsv hdfs://i
nputfile
```

... and in this example the first column is the rowkey, which is why the HBASE_ROW_KEY is used. The second and third columns in the file will be imported as "d:c1" and "d:c2", respectively.

ImportTsv Warning

If you have preparing a lot of data for bulk loading, make sure the target HBase table is pre-split appropriately.

See Also

For more information about bulk-loading HFiles into HBase, see arch.bulk.load

CompleteBulkLoad

The `completebulkload` utility will move generated StoreFiles into an HBase table. This utility is often used in conjunction with output from importtsv.

There are two ways to invoke this utility, with explicit classname and via the driver:

Explicit Classname

```
$ bin/hbase org.apache.hadoop.hbase.tool.LoadIncrementalHFiles hdfs://storefileou
tput TABLENAME
```

Driver

```
HADOOP_CLASSPATH=`${HBASE_HOME}/bin/hbase classpath` ${HADOOP_HOME}/bin/hadoop ja
r ${HBASE_HOME}/hbase-mapreduce-VERSION.jar completebulkload hdfs://storefileoutp
ut TABLENAME
```

CompleteBulkLoad Warning

Data generated via MapReduce is often created with file permissions that are not compatible with the running HBase process. Assuming you're running HDFS with permissions enabled, those permissions will need to be updated before you run CompleteBulkLoad.

For more information about bulk-loading HFiles into HBase, see arch.bulk.load.

WALPlayer

WALPlayer is a utility to replay WAL files into HBase.

The WAL can be replayed for a set of tables or all tables, and a timerange can be provided (in milliseconds). The WAL is filtered to this set of tables. The output can optionally be mapped to another set of tables.

WALPlayer can also generate HFiles for later bulk importing, in that case only a single table and no mapping can be specified.

Finally, you can use WALPlayer to replay the content of a Regions `recovered.edits` directory (the files under `recovered.edits` directory have the same format as WAL files).

> **WALPrettyPrinter**
>
> To read or verify single WAL files or *recovered.edits* files, since they share the WAL format, see WAL Tools.

Invoke via:

```
$ bin/hbase org.apache.hadoop.hbase.mapreduce.WALPlayer [options] <WAL inputdir>
[<tables> <tableMappings>]>
```

For example:

```
$ bin/hbase org.apache.hadoop.hbase.mapreduce.WALPlayer /backuplogdir oldTable1,o
ldTable2 newTable1,newTable2
```

WALPlayer, by default, runs as a mapreduce job. To NOT run WALPlayer as a mapreduce job on your cluster, force it to run all in the local process by adding the flags `-Dmapreduce.jobtracker.address=local` on the command line.

WALPlayer Options

Running `WALPlayer` with no arguments prints brief usage information:

```
Usage: WALPlayer [options] <WAL inputdir> [<tables> <tableMappings>]
 <WAL inputdir>   directory of WALs to replay.
 <tables>         comma separated list of tables. If no tables specified,
                  all are imported (even hbase:meta if present).
 <tableMappings>  WAL entries can be mapped to a new set of tables by passing
                  <tableMappings>, a comma separated list of target tables.
                  If specified, each table in <tables> must have a mapping.
To generate HFiles to bulk load instead of loading HBase directly, pass:
 -Dwal.bulk.output=/path/for/output
 Only one table can be specified, and no mapping allowed!
To specify a time range, pass:
 -Dwal.start.time=[date|ms]
 -Dwal.end.time=[date|ms]
 The start and the end date of timerange (inclusive). The dates can be
 expressed in milliseconds-since-epoch or yyyy-MM-dd'T'HH:mm:ss.SS format.
 E.g. 1234567890120 or 2009-02-13T23:32:30.12
Other options:
 -Dmapreduce.job.name=jobName
```

```
 Use the specified mapreduce job name for the wal player
 -Dwal.input.separator=' '
 Change WAL filename separator (WAL dir names use default ','.)
For performance also consider the following options:
  -Dmapreduce.map.speculative=false
  -Dmapreduce.reduce.speculative=false
```

RowCounter

RowCounter is a mapreduce job to count all the rows of a table. This is a good utility to use as a sanity check to ensure that HBase can read all the blocks of a table if there are any concerns of metadata inconsistency. It will run the mapreduce all in a single process but it will run faster if you have a MapReduce cluster in place for it to exploit. It is possible to limit the time range of data to be scanned by using the `--starttime=[starttime]` and `--endtime=[endtime]` flags. The scanned data can be limited based on keys using the `--range=[startKey],[endKey][;[startKey],[endKey]...]` option.

```
$ bin/hbase rowcounter [options] <tablename> [--starttime=<start> --endtime=<end>] [--range=[startKey],[endKey][;[startKey],[endKey]...]] [<column1> <column2>...]
```

RowCounter only counts one version per cell.

For performance consider to use `-Dhbase.client.scanner.caching=100` and `-Dmapreduce.map.speculative=false` options.

CellCounter

HBase ships another diagnostic mapreduce job called CellCounter. Like RowCounter, it gathers more fine-grained statistics about your table. The statistics gathered by CellCounter are more fine-grained and include:

- Total number of rows in the table.
- Total number of CFs across all rows.
- Total qualifiers across all rows.
- Total occurrence of each CF.

- Total occurrence of each qualifier.
- Total number of versions of each qualifier.

The program allows you to limit the scope of the run. Provide a row regex or prefix to limit the rows to analyze. Specify a time range to scan the table by using the `--starttime=<starttime>` and `--endtime=<endtime>` flags.

Use `hbase.mapreduce.scan.column.family` to specify scanning a single column family.

```
$ bin/hbase cellcounter TABLENAME OUTPUT_DIR [reportSeparator] [regex or prefix]
[--starttime=STARTTIME --endtime=ENDTIME]
```

Note: just like RowCounter, caching for the input Scan is configured via `hbase.client.scanner.caching` in the job configuration.

mlockall

It is possible to optionally pin your servers in physical memory making them less likely to be swapped out in oversubscribed environments by having the servers call **mlockall** on startup. See **HBASE-4391 Add ability to start RS as root and call mlockall** for how to build the optional library and have it run on startup.

Offline Compaction Tool

CompactionTool provides a way of running compactions (either minor or major) as an independent process from the RegionServer. It reuses same internal implementation classes executed by RegionServer compaction feature. However, since this runs on a complete separate independent java process, it releases RegionServers from the overhead involved in rewrite a set of hfiles, which can be critical for latency sensitive use cases.

Usage:

```
$ ./bin/hbase org.apache.hadoop.hbase.regionserver.CompactionTool

Usage: java org.apache.hadoop.hbase.regionserver.CompactionTool \
  [-compactOnce] [-major] [-mapred] [-D<property=value>]* files...
```

```
Options:
 mapred         Use MapReduce to run compaction.
 compactOnce    Execute just one compaction step. (default: while needed)
 major          Trigger major compaction.

Note: -D properties will be applied to the conf used.
For example:
 To stop delete of compacted file, pass -Dhbase.compactiontool.delete=false
 To set tmp dir, pass -Dhbase.tmp.dir=ALTERNATE_DIR

Examples:
 To compact the full 'TestTable' using MapReduce:
 $ hbase org.apache.hadoop.hbase.regionserver.CompactionTool -mapred hdfs://hbas
e/data/default/TestTable

 To compact column family 'x' of the table 'TestTable' region 'abc':
 $ hbase org.apache.hadoop.hbase.regionserver.CompactionTool hdfs://hbase/data/de
fault/TestTable/abc/x
```

As shown by usage options above, **CompactionTool** can run as a standalone client or a mapreduce job. When running as mapreduce job, each family dir is handled as an input split, and is processed by a separate map task.

The **compactionOnce** parameter controls how many compaction cycles will be performed until **CompactionTool** program decides to finish its work. If omitted, it will assume it should keep running compactions on each specified family as determined by the given compaction policy configured. For more info on compaction policy, see compaction.

If a major compaction is desired, **major** flag can be specified. If omitted, **CompactionTool** will assume minor compaction is wanted by default.

It also allows for configuration overrides with `-D` flag. In the usage section above, for example, `-Dhbase.compactiontool.delete=false` option will instruct compaction engine to not delete original files from temp folder.

Files targeted for compaction must be specified as parent hdfs dirs. It allows for multiple dirs definition, as long as each for these dirs are either a **family**, a **region**, or a **table** dir. If a table or region dir is passed, the program will recursively iterate through related sub-folders, effectively running compaction for each family found below the table/region level.

Since these dirs are nested under **hbase** hdfs directory tree, **CompactionTool** requires hbase super user permissions in order to have access to required hfiles.

> Running in MapReduce mode

MapReduce mode offers the ability to process each family dir in parallel, as a separate map task. Generally, it would make sense to run in this mode when specifying one or more table dirs as targets for compactions. The caveat, though, is that if number of families to be compacted become too large, the related mapreduce job may have indirect impacts on **RegionServers** performance . Since **NodeManagers** are normally co-located with RegionServers, such large jobs could compete for IO/Bandwidth resources with the **RegionServers**.

MajorCompaction completely disabled on RegionServers due performance impacts

Major compactions can be a costly operation (see compaction), and can indeed impact performance on RegionServers, leading operators to completely disable it for critical low latency application. **CompactionTool** could be used as an alternative in such scenarios, although, additional custom application logic would need to be implemented, such as deciding scheduling and selection of tables/regions/families target for a given compaction run.

For additional details about CompactionTool, see also **CompactionTool**.

`hbase clean`

The `hbase clean` command cleans HBase data from ZooKeeper, HDFS, or both. It is appropriate to use for testing. Run it with no options for usage instructions. The `hbase clean` command was introduced in HBase 0.98.

```
$ bin/hbase clean
Usage: hbase clean (--cleanZk|--cleanHdfs|--cleanAll)
Options:
    --cleanZk   cleans hbase related data from zookeeper.
    --cleanHdfs cleans hbase related data from hdfs.
    --cleanAll  cleans hbase related data from both zookeeper and hdfs.
```

`hbase pe`

The `hbase pe` command runs the PerformanceEvaluation tool, which is used for testing.

The PerformanceEvaluation tool accepts many different options and commands. For usage instructions, run the command with no options.

The PerformanceEvaluation tool has received many updates in recent HBase releases, including support for namespaces, support for tags, cell-level ACLs and visibility labels,

multiget support for RPC calls, increased sampling sizes, an option to randomly sleep during testing, and ability to "warm up" the cluster before testing starts.

`hbase ltt`

The `hbase ltt` command runs the LoadTestTool utility, which is used for testing.

You must specify either `-init_only` or at least one of `-write`, `-update`, or `-read`. For general usage instructions, pass the `-h` option.

The LoadTestTool has received many updates in recent HBase releases, including support for namespaces, support for tags, cell-level ACLS and visibility labels, testing security-related features, ability to specify the number of regions per server, tests for multi-get RPC calls, and tests relating to replication.

Pre-Upgrade validator

Pre-Upgrade validator tool can be used to check the cluster for known incompatibilities before upgrading from HBase 1 to HBase 2.

```
$ bin/hbase pre-upgrade command ...
```

Coprocessor validation

HBase supports co-processors for a long time, but the co-processor API can be changed between major releases. Co-processor validator tries to determine whether the old co-processors are still compatible with the actual HBase version.

```
$ bin/hbase pre-upgrade validate-cp [-jar ...] [-class ... | -table ... | -confi
g]
Options:
 -e            Treat warnings as errors.
 -jar <arg>    Jar file/directory of the coprocessor.
 -table <arg>  Table coprocessor(s) to check.
 -class <arg>  Coprocessor class(es) to check.
 -config         Scan jar for observers.
```

The co-processor classes can be explicitly declared by `-class` option, or they can be obtained from HBase configuration by `-config` option. Table level co-processors can be also checked by `-table` option. The tool searches for co-processors on its classpath, but it can be extended by the `-jar` option. It is possible to test multiple classes with multiple `-class`, multiple tables with multiple `-table` options as well as adding multiple jars to the classpath with multiple `-jar` options.

The tool can report errors and warnings. Errors mean that HBase won't be able to load the coprocessor, because it is incompatible with the current version of HBase. Warnings mean that the co-processors can be loaded, but they won't work as expected. If `-e` option is given, then the tool will also fail for warnings.

Please note that this tool cannot validate every aspect of jar files, it just does some static checks.

For example:

```
$ bin/hbase pre-upgrade validate-cp -jar my-coprocessor.jar -class MyMasterObserv
er -class MyRegionObserver
```

It validates `MyMasterObserver` and `MyRegionObserver` classes which are located in `my-coproc essor.jar`.

```
$ bin/hbase pre-upgrade validate-cp -table .*
```

It validates every table level co-processors where the table name matches to `.*` regular expression.

DataBlockEncoding validation

HBase 2.0 removed `PREFIX_TREE` Data Block Encoding from column families. For further information please check ***prefix-tree* encoding removed**. To verify that none of the column families are using incompatible Data Block Encodings in the cluster run the following command.

```
$ bin/hbase pre-upgrade validate-dbe
```

This check validates all column families and print out any incompatibilities. For example:

```
2018-07-13 09:58:32,028 WARN  [main] tool.DataBlockEncodingValidator: Incompatibl
e DataBlockEncoding for table: t, cf: f, encoding: PREFIX_TREE
```

Which means that Data Block Encoding of table `t`, column family `f` is incompatible. To fix, use `alter` command in HBase shell:

```
alter 't', { NAME => 'f', DATA_BLOCK_ENCODING => 'FAST_DIFF' }
```

Please also validate HFiles, which is described in the next section.

HFile Content validation

Even though Data Block Encoding is changed from `PREFIX_TREE` it is still possible to have HFiles that contain data encoded that way. To verify that HFiles are readable with HBase 2 please use *HFile content validator*.

```
$ bin/hbase pre-upgrade validate-hfile
```

The tool will log the corrupt HFiles and details about the root cause. If the problem is about PREFIX_TREE encoding it is necessary to change encodings before upgrading to HBase 2.

The following log message shows an example of incorrect HFiles.

```
2018-06-05 16:20:46,976 WARN  [hfilevalidator-pool1-t3] hbck.HFileCorruptionCheck
er: Found corrupt HFile hdfs://example.com:9000/hbase/data/default/t/72ea7f7d625e
e30f959897d1a3e2c350/prefix/7e6b3d73263c4851bf2b8590a9b3791e
org.apache.hadoop.hbase.io.hfile.CorruptHFileException: Problem reading HFile Tra
iler from file hdfs://example.com:9000/hbase/data/default/t/72ea7f7d625ee30f95989
7d1a3e2c350/prefix/7e6b3d73263c4851bf2b8590a9b3791e
    ...
Caused by: java.io.IOException: Invalid data block encoding type in file info: PR
EFIX_TREE
    ...
Caused by: java.lang.IllegalArgumentException: No enum constant org.apache.hadoo
p.hbase.io.encoding.DataBlockEncoding.PREFIX_TREE
    ...
2018-06-05 16:20:47,322 INFO  [main] tool.HFileContentValidator: Corrupted file:
hdfs://example.com:9000/hbase/data/default/t/72ea7f7d625ee30f959897d1a3e2c350/pre
fix/7e6b3d73263c4851bf2b8590a9b3791e
```

```
2018-06-05 16:20:47,383 INFO  [main] tool.HFileContentValidator: Corrupted file:
hdfs://example.com:9000/hbase/archive/data/default/t/56be41796340b757eb7fff1eb5e2
a905/f/29c641ae91c34fc3bee881f45436b6d1
```

Fixing PREFIX_TREE errors

It's possible to get `PREFIX_TREE` errors after changing Data Block Encoding to a supported one. It can happen because there are some HFiles which still encoded with `PREFIX_TREE` or there are still some snapshots.

For fixing HFiles, please run a major compaction on the table (it was `default:t` according to the log message):

```
major_compact 't'
```

HFiles can be referenced from snapshots, too. It's the case when the HFile is located under `archive/data`. The first step is to determine which snapshot references that HFile (the name of the file was `29c641ae91c34fc3bee881f45436b6d1` according to the logs):

```
for snapshot in $(hbase snapshotinfo -list-snapshots 2> /dev/null | tail -n -1 |
cut -f 1 -d \|);
do
  echo "checking snapshot named '${snapshot}'";
  hbase snapshotinfo -snapshot "${snapshot}" -files 2> /dev/null | grep 29c641ae9
1c34fc3bee881f45436b6d1;
done
```

The output of this shell script is:

```
checking snapshot named 't_snap'
   1.0 K t/56be41796340b757eb7fff1eb5e2a905/f/29c641ae91c34fc3bee881f45436b6d1 (a
rchive)
```

Which means `t_snap` snapshot references the incompatible HFile. If the snapshot is still needed, then it has to be recreated with HBase shell:

```
# creating a new namespace for the cleanup process
create_namespace 'pre_upgrade_cleanup'

# creating a new snapshot
clone_snapshot 't_snap', 'pre_upgrade_cleanup:t'
```

```
alter 'pre_upgrade_cleanup:t', { NAME => 'f', DATA_BLOCK_ENCODING => 'FAST_DIFF'
}
major_compact 'pre_upgrade_cleanup:t'

# removing the invalid snapshot
delete_snapshot 't_snap'

# creating a new snapshot
snapshot 'pre_upgrade_cleanup:t', 't_snap'

# removing temporary table
disable 'pre_upgrade_cleanup:t'
drop 'pre_upgrade_cleanup:t'
drop_namespace 'pre_upgrade_cleanup'
```

For further information, please refer to HBASE-20649.

Data Block Encoding Tool

Tests various compression algorithms with different data block encoder for key compression on an existing HFile. Useful for testing, debugging and benchmarking.

You must specify `-f` which is the full path of the HFile.

The result shows both the performance (MB/s) of compression/decompression and encoding/decoding, and the data savings on the HFile.

```
$ bin/hbase org.apache.hadoop.hbase.regionserver.DataBlockEncodingTool
Usages: hbase org.apache.hadoop.hbase.regionserver.DataBlockEncodingTool
Options:
        -f HFile to analyse (REQUIRED)
        -n Maximum number of key/value pairs to process in a single benchmark ru
n.
        -b Whether to run a benchmark to measure read throughput.
        -c If this is specified, no correctness testing will be done.
        -a What kind of compression algorithm use for test. Default value: GZ.
        -t Number of times to run each benchmark. Default value: 12.
        -omit Number of first runs of every benchmark to omit from statistics. De
fault value: 2.
```

HBase Conf Tool

HBase Conf tool can be used to print out the current value of a configuration. It can be used by passing the configuration key on the command-line.

```
$ bin/hbase org.apache.hadoop.hbase.util.HBaseConfTool <configuration_key>
```

Node Management

Node Decommission

You can stop an individual RegionServer by running the following script in the HBase directory on the particular node:

```
$ ./bin/hbase-daemon.sh stop regionserver
```

The RegionServer will first close all regions and then shut itself down. On shutdown, the RegionServer's ephemeral node in ZooKeeper will expire. The master will notice the RegionServer gone and will treat it as a 'crashed' server; it will reassign the nodes the RegionServer was carrying.

Disable the Load Balancer before Decommissioning a node

If the load balancer runs while a node is shutting down, then there could be contention between the Load Balancer and the Master's recovery of the just decommissioned RegionServer. Avoid any problems by disabling the balancer first. See the "Load Balancer" info section below.

Kill Node Tool

In hbase-2.0, in the bin directory, we added a script named *considerAsDead.sh* that can be used to kill a regionserver. Hardware issues could be detected by specialized monitoring tools before the zookeeper timeout has expired. *considerAsDead.sh* is a simple function to mark a RegionServer as dead. It deletes all the znodes of the server, starting the recovery process. Plug in the script into your monitoring/fault detection tools to initiate faster failover. Be careful how you use this disruptive tool. Copy the script if you need to make use of it in a version of hbase previous to hbase-2.0.

A downside to the above stop of a RegionServer is that regions could be offline for a good period of time. Regions are closed in order. If many regions on the server, the first region to close may not be back online until all regions close and after the master notices the RegionServer's znode gone. A node can be asked to gradually shed its load and then shutdown itself using the *graceful_stop.sh* script. Here is its usage:

```
$ ./bin/graceful_stop.sh
```

```
Usage: graceful_stop.sh [--config <conf-dir>] [-e] [--restart [--reload]] [--thri
ft] [--rest] [-n |--noack] [--maxthreads <number of threads>] [--movetimeout <tim
eout in seconds>] [-nob |--nobalancer] [-d |--designatedfile <file path>] [-x |--
excludefile <file path>] <hostname>
 thrift         If we should stop/start thrift before/after the hbase stop/start
 rest           If we should stop/start rest before/after the hbase stop/start
 restart        If we should restart after graceful stop
 reload         Move offloaded regions back on to the restarted server
 n|noack        Enable noAck mode in RegionMover. This is a best effort mode for
moving regions
 maxthreads xx  Limit the number of threads used by the region mover. Default val
ue is 1.
 movetimeout xx Timeout for moving regions. If regions are not moved by the timeo
ut value,exit with error. Default value is INT_MAX.
 hostname       Hostname of server we are to stop
 e|failfast     Set -e so exit immediately if any command exits with non-zero sta
tus
 nob|nobalancer Do not manage balancer states. This is only used as optimization
in rolling_restart.sh to avoid multiple calls to hbase shell
 d|designatedfile xx Designated file with <hostname:port> per line as unload targ
ets
 x|excludefile xx Exclude file should have <hostname:port> per line. We do not un
load regions to hostnames given in exclude file
```

To decommission a loaded RegionServer, run the following: $ ./bin/graceful_stop.sh HOSTNAME where `HOSTNAME` is the host carrying the RegionServer you would decommission.

> **On HOSTNAME**
>
> The `HOSTNAME` passed to *graceful_stop.sh* must match the hostname that hbase is using to identify RegionServers. HBase uses fully-qualified domain names usually. Check the list of RegionServers in the master UI for how HBase is referring to servers. Whatever HBase is using, this is what you should pass the *graceful_stop.sh* decommission script. If you pass IPs, the script is not yet smart enough to make a hostname (or FQDN) of it and so it will fail when it checks if server is currently running; the graceful unloading of regions will not run.

The *graceful_stop.sh* script will move the regions off the decommissioned RegionServer one at a time to minimize region churn. It will verify the region deployed in the new location before it will moves the next region and so on until the decommissioned server is carrying zero regions. At this point, the *graceful_stop.sh* tells the RegionServer `stop`. The master will at this point notice the RegionServer gone but all regions will have already been redeployed and because the RegionServer went down cleanly, there will be no WAL logs to split.

> **Load Balancer**

It is assumed that the Region Load Balancer is disabled while the `graceful_stop` script runs (otherwise the balancer and the decommission script will end up fighting over region deployments). Use the shell to disable the balancer:

```
hbase(main):001:0> balance_switch false
true
0 row(s) in 0.3590 seconds
```

This turns the balancer OFF. To reenable, do:

```
hbase(main):001:0> balance_switch true
false
0 row(s) in 0.3590 seconds
```

The `graceful_stop` will check the balancer and if enabled, will turn it off before it goes to work. If it exits prematurely because of error, it will not have reset the balancer. Hence, it is better to manage the balancer apart from `graceful_stop` reenabling it after you are done w/ graceful_stop.

Decommissioning several Regions Servers concurrently

If you have a large cluster, you may want to decommission more than one machine at a time by gracefully stopping multiple RegionServers concurrently. To gracefully drain multiple regionservers at the same time, RegionServers can be put into a "draining" state. This is done by marking a RegionServer as a draining node by creating an entry in ZooKeeper under the *hbase_root/draining* znode. This znode has format `name,port,startcode` just like the regionserver entries under *hbase_root/rs* znode.

Without this facility, decommissioning multiple nodes may be non-optimal because regions that are being drained from one region server may be moved to other regionservers that are also draining. Marking RegionServers to be in the draining state prevents this from happening. See this blog post for more details.

Bad or Failing Disk

It is good having dfs.datanode.failed.volumes.tolerated set if you have a decent number of disks per machine for the case where a disk plain dies. But usually disks do the "John Wayne" — i.e. take a while to go down spewing errors in *dmesg* — or for some reason, run much slower than their companions. In this case you want to decommission the disk. You have two options. You can decommission the datanode or, less disruptive in that only the bad disks data will be rereplicated, can stop the datanode, unmount the bad volume (You can't umount a volume while the datanode is using it), and then restart the datanode

(presuming you have set dfs.datanode.failed.volumes.tolerated > 0). The regionserver will throw some errors in its logs as it recalibrates where to get its data from — it will likely roll its WAL log too — but in general but for some latency spikes, it should keep on chugging.

> **Short Circuit Reads**
>
> If you are doing short-circuit reads, you will have to move the regions off the regionserver before you stop the datanode; when short-circuiting reading, though chmod'd so regionserver cannot have access, because it already has the files open, it will be able to keep reading the file blocks from the bad disk even though the datanode is down. Move the regions back after you restart the datanode.

Rolling Restart

Some cluster configuration changes require either the entire cluster, or the RegionServers, to be restarted in order to pick up the changes. In addition, rolling restarts are supported for upgrading to a minor or maintenance release, and to a major release if at all possible. See the release notes for release you want to upgrade to, to find out about limitations to the ability to perform a rolling upgrade.

There are multiple ways to restart your cluster nodes, depending on your situation. These methods are detailed below.

Using the `rolling-restart.sh` Script

HBase ships with a script, *bin/rolling-restart.sh*, that allows you to perform rolling restarts on the entire cluster, the master only, or the RegionServers only. The script is provided as a template for your own script, and is not explicitly tested. It requires password-less SSH login to be configured and assumes that you have deployed using a tarball. The script requires you to set some environment variables before running it. Examine the script and modify it to suit your needs.

rolling-restart.sh General Usage

```
$ ./bin/rolling-restart.sh --help
Usage: rolling-restart.sh [--config <hbase-confdir>] [--rs-only] [--master-only]
[--graceful] [--maxthreads xx]
```

Rolling Restart on RegionServers Only

To perform a rolling restart on the RegionServers only, use the `--rs-only` option. This

might be necessary if you need to reboot the individual RegionServer or if you make a configuration change that only affects RegionServers and not the other HBase processes.

Rolling Restart on Masters Only

To perform a rolling restart on the active and backup Masters, use the `--master-only` option. You might use this if you know that your configuration change only affects the Master and not the RegionServers, or if you need to restart the server where the active Master is running.

Graceful Restart

If you specify the `--graceful` option, RegionServers are restarted using the *bin/graceful_stop.sh* script, which moves regions off a RegionServer before restarting it. This is safer, but can delay the restart.

Limiting the Number of Threads

To limit the rolling restart to using only a specific number of threads, use the `--maxthreads` option.

Manual Rolling Restart

To retain more control over the process, you may wish to manually do a rolling restart across your cluster. This uses the `graceful-stop.sh` command decommission. In this method, you can restart each RegionServer individually and then move its old regions back into place, retaining locality. If you also need to restart the Master, you need to do it separately, and restart the Master before restarting the RegionServers using this method. The following is an example of such a command. You may need to tailor it to your environment. This script does a rolling restart of RegionServers only. It disables the load balancer before moving the regions.

```
$ for i in `cat conf/regionservers|sort`; do ./bin/graceful_stop.sh --restart --r
eload --debug $i; done &> /tmp/log.txt &;
```

Monitor the output of the */tmp/log.txt* file to follow the progress of the script.

Logic for Crafting Your Own Rolling Restart Script

Use the following guidelines if you want to create your own rolling restart script.

1. Extract the new release, verify its configuration, and synchronize it to all nodes of your cluster using `rsync`, `scp`, or another secure synchronization mechanism.

2. Restart the master first. You may need to modify these commands if your new HBase directory is different from the old one, such as for an upgrade.

```
$ ./bin/hbase-daemon.sh stop master; ./bin/hbase-daemon.sh start master
```

3. Gracefully restart each RegionServer, using a script such as the following, from the Master.

```
$ for i in `cat conf/regionservers|sort`; do ./bin/graceful_stop.sh --restart
--reload --debug $i; done &> /tmp/log.txt &
```

   If you are running Thrift or REST servers, pass the --thrift or --rest options. For other available options, run the `bin/graceful-stop.sh --help` command.

   It is important to drain HBase regions slowly when restarting multiple RegionServers. Otherwise, multiple regions go offline simultaneously and must be reassigned to other nodes, which may also go offline soon. This can negatively affect performance. You can inject delays into the script above, for instance, by adding a Shell command such as `sleep`. To wait for 5 minutes between each RegionServer restart, modify the above script to the following:

```
$ for i in `cat conf/regionservers|sort`; do ./bin/graceful_stop.sh --restart
--reload --debug $i & sleep 5m; done &> /tmp/log.txt &
```

4. Restart the Master again, to clear out the dead servers list and re-enable the load balancer.

Adding a New Node

Adding a new regionserver in HBase is essentially free, you simply start it like this: `$ ./bin/hbase-daemon.sh start regionserver` and it will register itself with the master. Ideally you also started a DataNode on the same machine so that the RS can eventually start to have local files. If you rely on ssh to start your daemons, don't forget to add the new hostname in *conf/regionservers* on the master.

At this point the region server isn't serving data because no regions have moved to it yet. If the balancer is enabled, it will start moving regions to the new RS. On a small/medium cluster this can have a very adverse effect on latency as a lot of regions will be offline at

the same time. It is thus recommended to disable the balancer the same way it's done when decommissioning a node and move the regions manually (or even better, using a script that moves them one by one).

The moved regions will all have 0% locality and won't have any blocks in cache so the region server will have to use the network to serve requests. Apart from resulting in higher latency, it may also be able to use all of your network card's capacity. For practical purposes, consider that a standard 1GigE NIC won't be able to read much more than *100MB/s*. In this case, or if you are in a OLAP environment and require having locality, then it is recommended to major compact the moved regions.

Metrics & Monitoring

HBase Metrics

HBase emits metrics which adhere to the Hadoop Metrics API. Starting with HBase 0.95 [5], HBase is configured to emit a default set of metrics with a default sampling period of every 10 seconds. You can use HBase metrics in conjunction with Ganglia. You can also filter which metrics are emitted and extend the metrics framework to capture custom metrics appropriate for your environment.

Metric Setup

For HBase 0.95 and newer, HBase ships with a default metrics configuration, or ***sink***. This includes a wide variety of individual metrics, and emits them every 10 seconds by default. To configure metrics for a given region server, edit the *conf/hadoop-metrics2-hbase.properties* file. Restart the region server for the changes to take effect.

To change the sampling rate for the default sink, edit the line beginning with `*.period`. To filter which metrics are emitted or to extend the metrics framework, see https://hadoop.apache.org/docs/current/api/org/apache/hadoop/metrics2/package-summary.html

> **HBase Metrics and Ganglia**
>
> By default, HBase emits a large number of metrics per region server. Ganglia may have difficulty processing all these metrics. Consider increasing the capacity of the Ganglia server or reducing the number of metrics emitted by HBase. See Metrics Filtering.

Disabling Metrics

To disable metrics for a region server, edit the *conf/hadoop-metrics2-hbase.properties* file and comment out any uncommented lines. Restart the region server for the changes to take effect.

Enabling Metrics Servlets

HBase exposes the metrics in many formats such as JSON, prometheus-format through different servlets (`/jmx`, `/metrics`, `/prometheus`). Any of these servlets can be enabled or disabled by the configuration property `hbase.http.metrics.servlets`. The value for the property should be a comma separated list of the servlet aliases which are `{jmx, metrics, prometheus}`. `/jmx`, `/metrics`, `/prometheus` are enabled by default. To get metrics using these servlets access the URL `http://SERVER_HOSTNAME:SERVER_WEB_UI_PORT/endpoint`. Where endpoint is one of `/jmx`, `/metrics`, or `/prometheus`. Eg. `http://my.rs.xyz.com:16030/prometheus`

Prometheus servlets

HBase exposes the metrics in prometheus friendly format through a servlet, `/prometheus`. Currently `/prometheus` exposes all the available metrics.

Discovering Available Metrics

Rather than listing each metric which HBase emits by default, you can browse through the available metrics, either as a JSON output or via JMX. Different metrics are exposed for the Master process and each region server process.

Procedure: Access a JSON Output of Available Metrics

1. After starting HBase, access the region server's web UI, at http://REGIONSERVER_HOSTNAME:16030 by default.
2. Click the **Metrics Dump** link near the top. The metrics for the region server are presented as a dump of the JMX bean in JSON format. This will dump out all metrics names and their values. To include metrics descriptions in the listing — this can be useful when you are exploring what is available — add a query string of `?descriptio`

`n=true` so your URL becomes http://REGIONSERVER_HOSTNAME:16030/jmx?description=true. Not all beans and attributes have descriptions.

3. To view metrics for the Master, connect to the Master's web UI instead (defaults to http://localhost:16010) and click its **Metrics Dump** link. To include metrics descriptions in the listing — this can be useful when you are exploring what is available — add a query string of `?description=true` so your URL becomes http://REGIONSERVER_HOSTNAME:16010/jmx?description=true. Not all beans and attributes have descriptions.

You can use many different tools to view JMX content by browsing MBeans. This procedure uses `jvisualvm`, which is an application usually available in the JDK.

Procedure: Browse the JMX Output of Available Metrics

1. Start HBase, if it is not already running.
2. Run the command `jvisualvm` command on a host with a GUI display. You can launch it from the command line or another method appropriate for your operating system.
3. Be sure the **VisualVM-MBeans** plugin is installed. Browse to **Tools** → **Plugins**. Click **Installed** and check whether the plugin is listed. If not, click **Available Plugins**, select it, and click Install. When finished, click Close.
4. To view details for a given HBase process, double-click the process in the **Local** sub-tree in the left-hand panel. A detailed view opens in the right-hand panel. Click the **MBeans** tab which appears as a tab in the top of the right-hand panel.
5. To access the HBase metrics, navigate to the appropriate sub-bean: .* Master: .* RegionServer:
6. The name of each metric and its current value is displayed in the **Attributes** tab. For a view which includes more details, including the description of each attribute, click the **Metadata** tab.

Units of Measure for Metrics

Different metrics are expressed in different units, as appropriate. Often, the unit of measure is in the name (as in the metric `shippedKBs`). Otherwise, use the following guidelines. When

in doubt, you may need to examine the source for a given metric.

- Metrics that refer to a point in time are usually expressed as a timestamp.
- Metrics that refer to an age (such as `ageOfLastShippedOp`) are usually expressed in milliseconds.
- Metrics that refer to memory sizes are in bytes.
- Sizes of queues (such as `sizeOfLogQueue`) are expressed as the number of items in the queue. Determine the size by multiplying by the block size (default is 64 MB in HDFS).
- Metrics that refer to things like the number of a given type of operations (such as `logEditsRead`) are expressed as an integer.

Most Important Master Metrics

Note: Counts are usually over the last metrics reporting interval.

hbase.master.numRegionServers
Number of live regionservers

hbase.master.numDeadRegionServers
Number of dead regionservers

hbase.master.ritCount
The number of regions in transition

hbase.master.ritCountOverThreshold
The number of regions that have been in transition longer than a threshold time (default: 60 seconds)

hbase.master.ritOldestAge
The age of the longest region in transition, in milliseconds

Most Important RegionServer Metrics

Note: Counts are usually over the last metrics reporting interval.

hbase.regionserver.regionCount
The number of regions hosted by the regionserver

hbase.regionserver.storeFileCount

The number of store files on disk currently managed by the regionserver

hbase.regionserver.storeFileSize

Aggregate size of the store files on disk

hbase.regionserver.hlogFileCount

The number of write ahead logs not yet archived

hbase.regionserver.totalRequestCount

The total number of requests received

hbase.regionserver.readRequestCount

The number of read requests received

hbase.regionserver.writeRequestCount

The number of write requests received

hbase.regionserver.numOpenConnections

The number of open connections at the RPC layer

hbase.regionserver.numActiveHandler

The number of RPC handlers actively servicing requests

hbase.regionserver.numCallsInGeneralQueue

The number of currently enqueued user requests

hbase.regionserver.numCallsInReplicationQueue

The number of currently enqueued operations received from replication

hbase.regionserver.numCallsInPriorityQueue

The number of currently enqueued priority (internal housekeeping) requests

hbase.regionserver.flushQueueLength

Current depth of the memstore flush queue. If increasing, we are falling behind with clearing memstores out to HDFS.

hbase.regionserver.updatesBlockedTime

Number of milliseconds updates have been blocked so the memstore can be flushed

hbase.regionserver.compactionQueueLength

Current depth of the compaction request queue. If increasing, we are falling behind with storefile compaction.

hbase.regionserver.blockCacheHitCount

The number of block cache hits

hbase.regionserver.blockCacheMissCount

The number of block cache misses

hbase.regionserver.blockCacheExpressHitPercent

The percent of the time that requests with the cache turned on hit the cache

hbase.regionserver.percentFilesLocal

Percent of store file data that can be read from the local DataNode, 0-100

hbase.regionserver.<op>_<measure>

Operation latencies, where <op> is one of Append, Delete, Mutate, Get, Replay, Increment; and where <measure> is one of min, max, mean, median, 75th_percentile, 95th_percentile, 99th_percentile

hbase.regionserver.slow<op>Count

The number of operations we thought were slow, where <op> is one of the list above

hbase.regionserver.GcTimeMillis

Time spent in garbage collection, in milliseconds

hbase.regionserver.GcTimeMillisParNew

Time spent in garbage collection of the young generation, in milliseconds

hbase.regionserver.GcTimeMillisConcurrentMarkSweep

Time spent in garbage collection of the old generation, in milliseconds

hbase.regionserver.authenticationSuccesses

Number of client connections where authentication succeeded

hbase.regionserver.authenticationFailures

Number of client connection authentication failures

hbase.regionserver.mutationsWithoutWALCount

Count of writes submitted with a flag indicating they should bypass the write ahead log

Meta Table Load Metrics

HBase meta table metrics collection feature is available in HBase 1.4+ but it is disabled by default, as it can affect the performance of the cluster. When it is enabled, it helps to monitor client access patterns by collecting the following statistics:

- number of get, put and delete operations on the `hbase:meta` table
- number of get, put and delete operations made by the top-N clients
- number of operations related to each table
- number of operations related to the top-N regions

  When to use the feature

  This feature can help to identify hot spots in the meta table by showing the regions or tables where the meta info is modified (e.g. by create, drop, split or move tables) or retrieved most frequently. It can also help to find misbehaving client applications by showing which clients are using the meta table most heavily, which can for example suggest the lack of meta table buffering or the lack of re-using open client connections in the client application.

> ⚠ Possible side-effects of enabling this feature
>
> Having large number of clients and regions in the cluster can cause the registration and tracking of a large amount of metrics, which can increase the memory and CPU footprint of the HBase region server handling the `hbase:meta` table. It can also cause the significant increase of the JMX dump size, which can affect the monitoring or log aggregation system you use beside HBase. It is recommended to turn on this feature only during debugging.

Where to find the metrics in JMX

Each metric attribute name will start with the 'MetaTable_' prefix. For all the metrics you will see five different JMX attributes: count, mean rate, 1 minute rate, 5 minute rate and 15 minute rate. You will find these metrics in JMX under the following MBean: `Hadoop → HBase → RegionServer → Coprocessor.Region.CP_org.apache.hadoop.hbase.coprocessor.MetaTableMetrics`.

Examples: some Meta Table metrics you can see in your JMX dump

```
{
```

```
  "MetaTable_get_request_count": 77309,
  "MetaTable_put_request_mean_rate": 0.06339092997186495,
  "MetaTable_table_MyTestTable_request_15min_rate": 1.1020599841623246,
  "MetaTable_client_/172.30.65.42_lossy_request_count": 1786
  "MetaTable_client_/172.30.65.45_put_request_5min_rate": 0.6189810954855728,
  "MetaTable_region_1561131112259.c66e4308d492936179352c80432ccfe0._lossy_request
_count": 38342,
  "MetaTable_region_1561131043640.5bdffe4b9e7e334172065c853cf0caa6._lossy_request
_1min_rate": 0.04925099917433935,
}
```

Configuration

To turn on this feature, you have to enable a custom coprocessor by adding the following section to hbase-site.xml. This coprocessor will run on all the HBase RegionServers, but will be active (i.e. consume memory / CPU) only on the server, where the `hbase:meta` table is located. It will produce JMX metrics which can be downloaded from the web UI of the given RegionServer or by a simple REST call. These metrics will not be present in the JMX dump of the other RegionServers.

Enabling the Meta Table Metrics feature

```
<property>
    <name>hbase.coprocessor.region.classes</name>
    <value>org.apache.hadoop.hbase.coprocessor.MetaTableMetrics</value>
</property>
```

How the top-N metrics are calculated?

The 'top-N' type of metrics will be counted using the Lossy Counting Algorithm (as defined in Motwani, R; Manku, G.S (2002). "Approximate frequency counts over data streams"), which is designed to identify elements in a data stream whose frequency count exceed a user-given threshold. The frequency computed by this algorithm is not always accurate but has an error threshold that can be specified by the user as a configuration parameter. The run time space required by the algorithm is inversely proportional to the specified error threshold, hence larger the error parameter, the smaller the footprint and the less accurate are the metrics.

You can specify the error rate of the algorithm as a floating-point value between 0 and 1 (exclusive), it's default value is 0.02. Having the error rate set to `E` and having `N` as the total number of meta table operations, then (assuming the uniform distribution of the activity of low frequency elements) at most `7 / E` meters will be kept and each kept element will have a frequency higher than `E * N`.

An example: Let's assume we are interested in the HBase clients that are most active in accessing the meta table. When there was 1,000,000 operations on the meta table so far and the error rate parameter is set to 0.02, then we can assume that only at most 350 client IP address related counters will be present in JMX and each of these clients accessed the meta table at least 20,000 times.

```
<property>
    <name>hbase.util.default.lossycounting.errorrate</name>
    <value>0.02</value>
</property>
```

HBase Monitoring

Overview

The following metrics are arguably the most important to monitor for each RegionServer for "macro monitoring", preferably with a system like OpenTSDB. If your cluster is having performance issues it's likely that you'll see something unusual with this group.

HBase

- See rs metrics

OS

- IO Wait
- User CPU

Java

- GC

Slow Query Log

The HBase slow query log consists of parseable JSON structures describing the properties of those client operations (Gets, Puts, Deletes, etc.) that either took too long to run, or produced too much output. The thresholds for "too long to run" and "too much output" are configurable, as described below. The output is produced inline in the main region server logs so that it is easy to discover further details from context with other logged events. It is also prepended with identifying tags `(responseTooSlow)`, `(responseTooLarge)`, `(operationTooSlow)`, and `(operationTooLarge)` in order to enable easy filtering with grep, in case the user desires to see only slow queries.

Configuration

There are four configuration knobs that can be used to adjust the thresholds for when queries are logged. Two of these knobs control the size and time thresholds for all queries. Because Scans can often be larger and slower than other types of queries, there are two additional knobs which can control size and time thresholds for Scans specifically.

- `hbase.ipc.warn.response.time` Maximum number of milliseconds that a query can be run without being logged. Defaults to 10000, or 10 seconds. Can be set to -1 to disable logging by time.
- `hbase.ipc.warn.response.size` Maximum byte size of response that a query can return without being logged. Defaults to 100 megabytes. Can be set to -1 to disable logging by size.
- `hbase.ipc.warn.response.time.scan` Maximum number of milliseconds that a Scan can be run without being logged. Defaults to the `hbase.ipc.warn.response.time` value. Can be set to -1 to disable logging by time.
- `hbase.ipc.warn.response.size.scan` Maximum byte size of response that a Scan can return without being logged. Defaults to the `hbase.ipc.warn.response.size` value. Can be set to -1 to disable logging by size.

Metrics

The slow query log exposes to metrics to JMX.

- `hadoop.regionserver_rpc_slowResponse` a global metric reflecting the durations of all responses that triggered logging.
- `hadoop.regionserver_rpc_methodName.aboveOneSec` A metric reflecting the durations of all responses that lasted for more than one second.

Output

The output is tagged with operation e.g. `(operationTooSlow)` if the call was a client operation, such as a Put, Get, or Delete, which we expose detailed fingerprint information for. If not, it is tagged `(responseTooSlow)` and still produces parseable JSON output, but with less verbose information solely regarding its duration and size in the RPC itself. `TooLarge` is substituted for `TooSlow` if the response size triggered the logging, with `TooLarge` appearing even in the case that both size and duration triggered logging.

Example

```
2011-09-08 10:01:25,824 WARN org.apache.hadoop.ipc.HBaseServer: (operationTooSlo
w): {"tables":{"riley2":{"puts":[{"totalColumns":11,"families":{"actions":[{"time
stamp":1315501284459,"qualifier":"0","vlen":9667580},{"timestamp":131550128445
9,"qualifier":"1","vlen":10122412},{"timestamp":1315501284459,"qualifier":"2","vl
en":11104617},{"timestamp":1315501284459,"qualifier":"3","vlen":13430635}]},"ro
w":"cfcd208495d565ef66e7dff9f98764da:0"}],"families":["actions"]}},"processingtim
ems":956,"client":"10.47.34.63:33623","starttimems":1315501284456,"queuetimems":
0,"totalPuts":1,"class":"HRegionServer","responsesize":0,"method":"multiPut"}
```

Note that everything inside the "tables" structure is output produced by MultiPut's fingerprint, while the rest of the information is RPC-specific, such as processing time and client IP/port. Other client operations follow the same pattern and the same general structure, with necessary differences due to the nature of the individual operations. In the case that the call is not a client operation, that detailed fingerprint information will be completely absent.

This particular example, for example, would indicate that the likely cause of slowness is simply a very large (on the order of 100MB) multiput, as we can tell by the "vlen," or value length, fields of each put in the multiPut.

Get Slow Response Log from shell

When an individual RPC exceeds a configurable time bound we log a complaint by way of the logging subsystem

e.g.

```
2019-10-02 10:10:22,195 WARN [,queue=15,port=60020] ipc.RpcServer - (responseTooS
low):
{"call":"Scan(org.apache.hadoop.hbase.protobuf.generated.ClientProtos$ScanReques
t)",
"starttimems":1567203007549,
"responsesize":6819737,
"method":"Scan",
"param":"region { type: REGION_NAME value: \"t1,\\000\\000\\215\\f)o\\\\\\024\\30
2\\220\\000\\000\\000\\000\\000\\001\\000\\000\\000\\000\\000\\006\\000\\000\\000
\\000\\000\\005\\000\\000<TRUNCATED>",
"processingtimems":28646,
"client":"10.253.196.215:41116",
"queuetimems":22453,
"class":"HRegionServer"}
```

Unfortunately often the request parameters are truncated as per above Example. The truncation is unfortunate because it eliminates much of the utility of the warnings. For example, the region name, the start and end keys, and the filter hierarchy are all important clues for debugging performance problems caused by moderate to low selectivity queries or queries made at a high rate.

HBASE-22978 introduces maintaining an in-memory ring buffer of requests that were judged to be too slow in addition to the responseTooSlow logging. The in-memory representation can be complete. There is some chance a high rate of requests will cause information on other interesting requests to be overwritten before it can be read. This is an acceptable trade off.

In order to enable the in-memory ring buffer at RegionServers, we need to enable config:

```
hbase.regionserver.slowlog.buffer.enabled
```

One more config determines the size of the ring buffer:

```
hbase.regionserver.slowlog.ringbuffer.size
```

Check the config section for the detailed description.

This config would be disabled by default. Turn it on and these shell commands would provide expected results from the ring-buffers.

shell commands to retrieve slowlog responses from RegionServers:

```
Retrieve latest SlowLog Responses maintained by each or specific RegionServers.
Specify '*' to include all RS otherwise array of server names for specific
RS. A server name is the host, port plus startcode of a RegionServer.
e.g.: host187.example.com,60020,1289493121758 (find servername in
master ui or when you do detailed status in shell)

Provide optional filter parameters as Hash.
Default Limit of each server for providing no of slow log records is 10. User can
specify
more limit by 'LIMIT' param in case more than 10 records should be retrieved.

Examples:

  hbase> get_slowlog_responses '*'                                    => get slowlog
responses from all RS
```

```
  hbase> get_slowlog_responses '*', {'LIMIT' => 50}                => get slowlog
responses from all RS
                                                                      with 50 rec
ords limit (default limit: 10)
  hbase> get_slowlog_responses ['SERVER_NAME1', 'SERVER_NAME2']    => get slowlog
responses from SERVER_NAME1,
                                                                      SERVER_NAME
2
  hbase> get_slowlog_responses '*', {'REGION_NAME' => 'hbase:meta,,1'}
                                                                   => get slowlog
responses only related to meta
                                                                      region
  hbase> get_slowlog_responses '*', {'TABLE_NAME' => 't1'}         => get slowlog
responses only related to t1 table
  hbase> get_slowlog_responses '*', {'CLIENT_IP' => '192.162.1.40:60225', 'LIMIT'
=> 100}
                                                                   => get slowlog
responses with given client
                                                                      IP address
and get 100 records limit
                                                                      (default li
mit: 10)
  hbase> get_slowlog_responses '*', {'REGION_NAME' => 'hbase:meta,,1', 'TABLE_NAM
E' => 't1'}
                                                                   => get slowlog
responses with given region name
                                                                      or table na
me
  hbase> get_slowlog_responses '*', {'USER' => 'user_name', 'CLIENT_IP' => '192.1
62.1.40:60225'}
                                                                   => get slowlog
responses that match either
                                                                      provided cl
ient IP address or user name
```

All of above queries with filters have default OR operation applied i.e. all records with any of the provided filters applied will be returned. However, we can also apply AND operator i.e. all records that match all (not any) of the provided filters should be returned.

```
  hbase> get_slowlog_responses '*', {'REGION_NAME' => 'hbase:meta,,1', 'TABLE_NAM
E' => 't1', 'FILTER_BY_OP' => 'AND'}
                                                                   => get slowlog
responses with given region name
                                                                      and table n
ame, both should match

  hbase> get_slowlog_responses '*', {'REGION_NAME' => 'hbase:meta,,1', 'TABLE_NAM
E' => 't1', 'FILTER_BY_OP' => 'OR'}
                                                                   => get slowlog
responses with given region name
                                                                      or table na
me, any one can match
```

```
  hbase> get_slowlog_responses '*', {'TABLE_NAME' => 't1', 'CLIENT_IP' => '192.16
3.41.53:52781', 'FILTER_BY_OP' => 'AND'}
                                                                => get slowlog
responses with given region name
                                                                   and client
IP address, both should match
```

Since OR is the default filter operator, without providing 'FILTER_BY_OP', query will have same result as providing 'FILTER_BY_OP' ⇒ 'OR'.

Sometimes output can be long pretty printed json for user to scroll in a single screen and hence user might prefer redirecting output of get_slowlog_responses to a file.

Example:

```
echo "get_slowlog_responses '*'" | hbase shell > xyz.out 2>&1
```

Similar to slow RPC logs, client can also retrieve large RPC logs. Sometimes, slow logs important to debug perf issues turn out to be larger in size.

```
  hbase> get_largelog_responses '*'                                 => get largel
og responses from all RS
  hbase> get_largelog_responses '*', {'LIMIT' => 50}                => get largel
og responses from all RS
                                                                       with 50 re
cords limit (default limit: 10)
  hbase> get_largelog_responses ['SERVER_NAME1', 'SERVER_NAME2']    => get largel
og responses from SERVER_NAME1,
                                                                       SERVER_NAM
E2
  hbase> get_largelog_responses '*', {'REGION_NAME' => 'hbase:meta,,1'}
                                                                    => get largel
og responses only related to meta
                                                                       region
  hbase> get_largelog_responses '*', {'TABLE_NAME' => 't1'}         => get largel
og responses only related to t1 table
  hbase> get_largelog_responses '*', {'CLIENT_IP' => '192.162.1.40:60225', 'LIMI
T' => 100}
                                                                    => get largel
og responses with given client
                                                                       IP address
and get 100 records limit
                                                                       (default l
imit: 10)
  hbase> get_largelog_responses '*', {'REGION_NAME' => 'hbase:meta,,1', 'TABLE_NA
ME' => 't1'}
                                                                    => get largel
og responses with given region name
```

```
                                                                  or table n
ame
  hbase> get_largelog_responses '*', {'USER' => 'user_name', 'CLIENT_IP' => '192.
162.1.40:60225'}
                                                                => get largel
og responses that match either
                                                                  provided c
lient IP address or user name

  hbase> get_largelog_responses '*', {'REGION_NAME' => 'hbase:meta,,1', 'TABLE_NA
ME' => 't1', 'FILTER_BY_OP' => 'AND'}
                                                               => get largelo
g responses with given region name
                                                                  and table n
ame, both should match

  hbase> get_largelog_responses '*', {'REGION_NAME' => 'hbase:meta,,1', 'TABLE_NA
ME' => 't1', 'FILTER_BY_OP' => 'OR'}
                                                               => get largelo
g responses with given region name
                                                                  or table na
me, any one can match

  hbase> get_largelog_responses '*', {'TABLE_NAME' => 't1', 'CLIENT_IP' => '192.1
63.41.53:52781', 'FILTER_BY_OP' => 'AND'}
                                                               => get largelo
g responses with given region name
                                                                  and client
IP address, both should match
```

shell command to clear slow/largelog responses from RegionServer:

```
Clears SlowLog Responses maintained by each or specific RegionServers.
Specify array of server names for specific RS. A server name is
the host, port plus startcode of a RegionServer.
e.g.: host187.example.com,60020,1289493121758 (find servername in
master ui or when you do detailed status in shell)

Examples:

  hbase> clear_slowlog_responses                                  => clears sl
owlog responses from all RS
  hbase> clear_slowlog_responses ['SERVER_NAME1', 'SERVER_NAME2']    => clears sl
owlog responses from SERVER_NAME1,
                                                                    SERVER_NA
ME2
```

Get Slow/Large Response Logs from System table hbase:slowlog

The above section provides details about Admin APIs:

- get_slowlog_responses

- get_largelog_responses
- clear_slowlog_responses

All of the above APIs access online in-memory ring buffers from individual RegionServers and accumulate logs from ring buffers to display to end user. However, since the logs are stored in memory, after RegionServer is restarted, all the objects held in memory of that RegionServer will be cleaned up and previous logs are lost. What if we want to persist all these logs forever? What if we want to store them in such a manner that operator can get all historical records with some filters? e.g get me all large/slow RPC logs that are triggered by user1 and are related to region: cluster_test,cccccccc,1589635796466.aa45e1571d533f5ed0bb31cdccaaf9cf. ?

If we have a system table that stores such logs in increasing (not so strictly though) order of time, it can definitely help operators debug some historical events (scan, get, put, compaction, flush etc) with detailed inputs.

Config which enabled system table to be created and store all log events is `hbase.regionserver.slowlog.systable.enabled`.

The default value for this config is `false`. If provided `true` (Note: `hbase.regionserver.slowlog.buffer.enabled` should also be `true`), a cron job running in every RegionServer will persist the slow/large logs into table hbase:slowlog. By default cron job runs every 10 min. Duration can be configured with key: `hbase.slowlog.systable.chore.duration`. By default, RegionServer will store upto 1000(config key: `hbase.regionserver.slowlog.systable.queue.size`) slow/large logs in an internal queue and the chore will retrieve these logs from the queue and perform batch insertion in hbase:slowlog.

hbase:slowlog has single ColumnFamily: `info` `info` contains multiple qualifiers which are the same attributes present as part of `get_slowlog_responses` API response.

- info:call_details
- info:client_address
- info:method_name
- info:param
- info:processing_time
- info:queue_time

- info:region_name
- info:response_size
- info:server_class
- info:start_time
- info:type
- info:username

And example of 2 rows from hbase:slowlog scan result:

```
 \x024\xC1\x03\xE9\x04\xF5@                                column=info:call_det
ails, timestamp=2020-05-16T14:58:14.211Z, value=Scan(org.apache.hadoop.hbase.shad
ed.protobuf.generated.ClientProtos$ScanRequest)
 \x024\xC1\x03\xE9\x04\xF5@                                column=info:client_a
ddress, timestamp=2020-05-16T14:58:14.211Z, value=172.20.10.2:57347
 \x024\xC1\x03\xE9\x04\xF5@                                column=info:method_n
ame, timestamp=2020-05-16T14:58:14.211Z, value=Scan
 \x024\xC1\x03\xE9\x04\xF5@                                column=info:param, t
imestamp=2020-05-16T14:58:14.211Z, value=region { type: REGION_NAME value: "hbas
e:meta,,1" } scan { column { family: "info" } attribute { name: "_isolationle
                                                           vel_" value: "\x5C00
0" } start_row: "cluster_test,33333333,99999999999999" stop_row: "cluster_test,,"
time_range { from: 0 to: 9223372036854775807 } max_versions: 1 cache_blocks
                                                           : true max_result_si
ze: 2097152 reversed: true caching: 10 include_stop_row: true readType: PREAD } n
umber_of_rows: 10 close_scanner: false client_handles_partials: true client_
                                                           handles_heartbeats:
true track_scan_metrics: false
 \x024\xC1\x03\xE9\x04\xF5@                                column=info:processi
ng_time, timestamp=2020-05-16T14:58:14.211Z, value=18
 \x024\xC1\x03\xE9\x04\xF5@                                column=info:queue_ti
me, timestamp=2020-05-16T14:58:14.211Z, value=0
 \x024\xC1\x03\xE9\x04\xF5@                                column=info:region_n
ame, timestamp=2020-05-16T14:58:14.211Z, value=hbase:meta,,1
 \x024\xC1\x03\xE9\x04\xF5@                                column=info:response
_size, timestamp=2020-05-16T14:58:14.211Z, value=1575
 \x024\xC1\x03\xE9\x04\xF5@                                column=info:server_c
lass, timestamp=2020-05-16T14:58:14.211Z, value=HRegionServer
 \x024\xC1\x03\xE9\x04\xF5@                                column=info:start_ti
me, timestamp=2020-05-16T14:58:14.211Z, value=1589640743732
 \x024\xC1\x03\xE9\x04\xF5@                                column=info:type, ti
mestamp=2020-05-16T14:58:14.211Z, value=ALL
 \x024\xC1\x03\xE9\x04\xF5@                                column=info:usernam
e, timestamp=2020-05-16T14:58:14.211Z, value=user2
 \x024\xC1\x06X\x81\xF6\xEC                                column=info:call_det
ails, timestamp=2020-05-16T14:59:58.764Z, value=Scan(org.apache.hadoop.hbase.shad
ed.protobuf.generated.ClientProtos$ScanRequest)
 \x024\xC1\x06X\x81\xF6\xEC                                column=info:client_a
ddress, timestamp=2020-05-16T14:59:58.764Z, value=172.20.10.2:57348
```

```
 \x024\xC1\x06X\x81\xF6\xEC                                    column=info:method_n
ame, timestamp=2020-05-16T14:59:58.764Z, value=Scan
 \x024\xC1\x06X\x81\xF6\xEC                                    column=info:param, t
imestamp=2020-05-16T14:59:58.764Z, value=region { type: REGION_NAME value: "clust
er_test,cccccccc,1589635796466.aa45e1571d533f5ed0bb31cdccaaf9cf." } scan { a
                                                              ttribute { name: "_i
solationlevel_" value: "\x5C000" } start_row: "cccccccc" time_range { from: 0 to:
9223372036854775807 } max_versions: 1 cache_blocks: true max_result_size: 2
                                                              097152 caching: 2147
483647 include_stop_row: false } number_of_rows: 2147483647 close_scanner: false
client_handles_partials: true client_handles_heartbeats: true track_scan_met
                                                              rics: false
 \x024\xC1\x06X\x81\xF6\xEC                                    column=info:processi
ng_time, timestamp=2020-05-16T14:59:58.764Z, value=24
 \x024\xC1\x06X\x81\xF6\xEC                                    column=info:queue_ti
me, timestamp=2020-05-16T14:59:58.764Z, value=0
 \x024\xC1\x06X\x81\xF6\xEC                                    column=info:region_n
ame, timestamp=2020-05-16T14:59:58.764Z, value=cluster_test,cccccccc,158963579646
6.aa45e1571d533f5ed0bb31cdccaaf9cf.
 \x024\xC1\x06X\x81\xF6\xEC                                    column=info:response
_size, timestamp=2020-05-16T14:59:58.764Z, value=211227
 \x024\xC1\x06X\x81\xF6\xEC                                    column=info:server_c
lass, timestamp=2020-05-16T14:59:58.764Z, value=HRegionServer
 \x024\xC1\x06X\x81\xF6\xEC                                    column=info:start_ti
me, timestamp=2020-05-16T14:59:58.764Z, value=1589640743932
 \x024\xC1\x06X\x81\xF6\xEC                                    column=info:type, ti
mestamp=2020-05-16T14:59:58.764Z, value=ALL
 \x024\xC1\x06X\x81\xF6\xEC                                    column=info:usernam
e, timestamp=2020-05-16T14:59:58.764Z, value=user1
```

Operator can use ColumnValueFilter to filter records based on region_name, username, client_address etc.

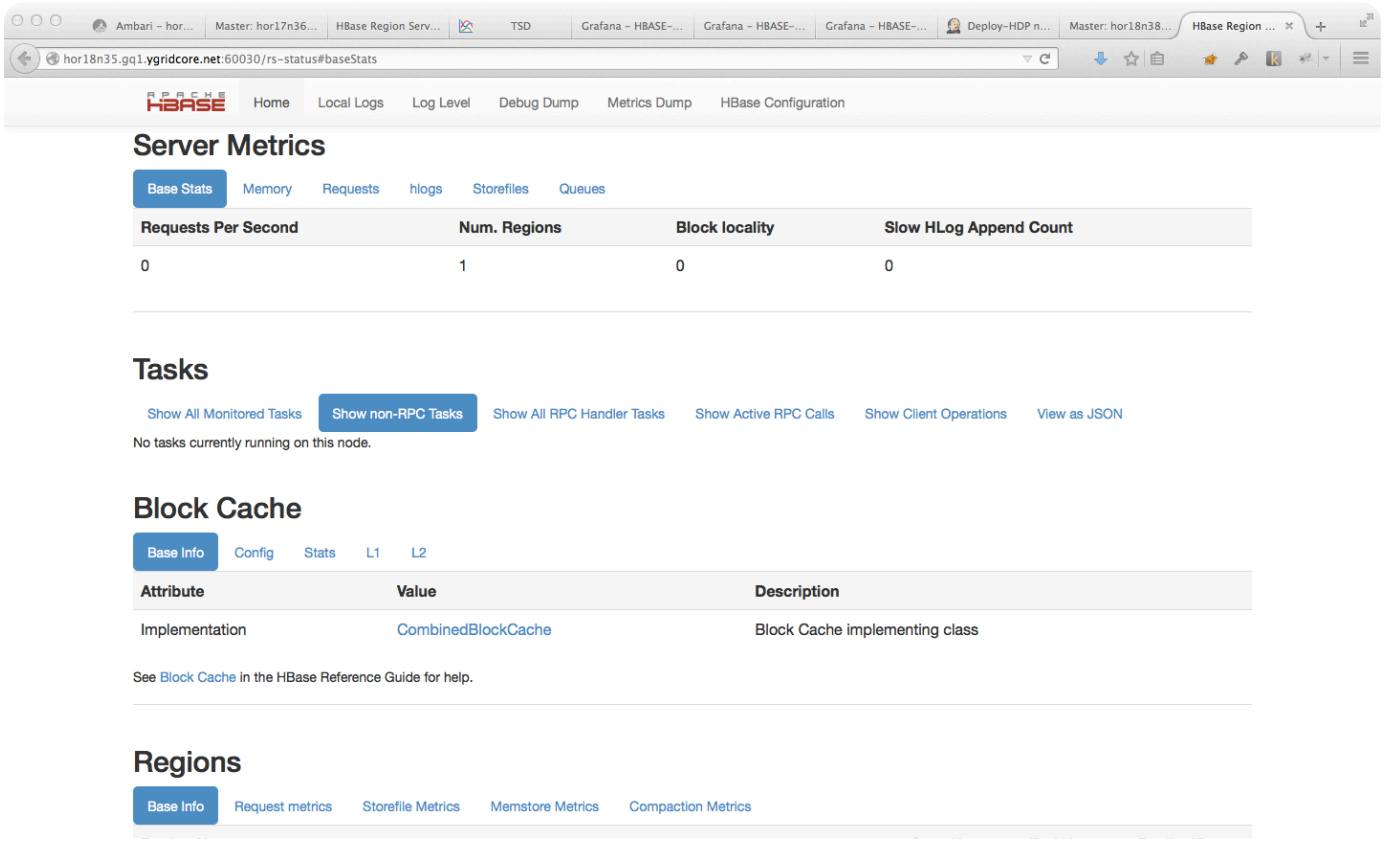
Time range based queries will also be very useful. Example:

```
scan 'hbase:slowlog', { TIMERANGE => [1589621394000, 1589637999999] }
```

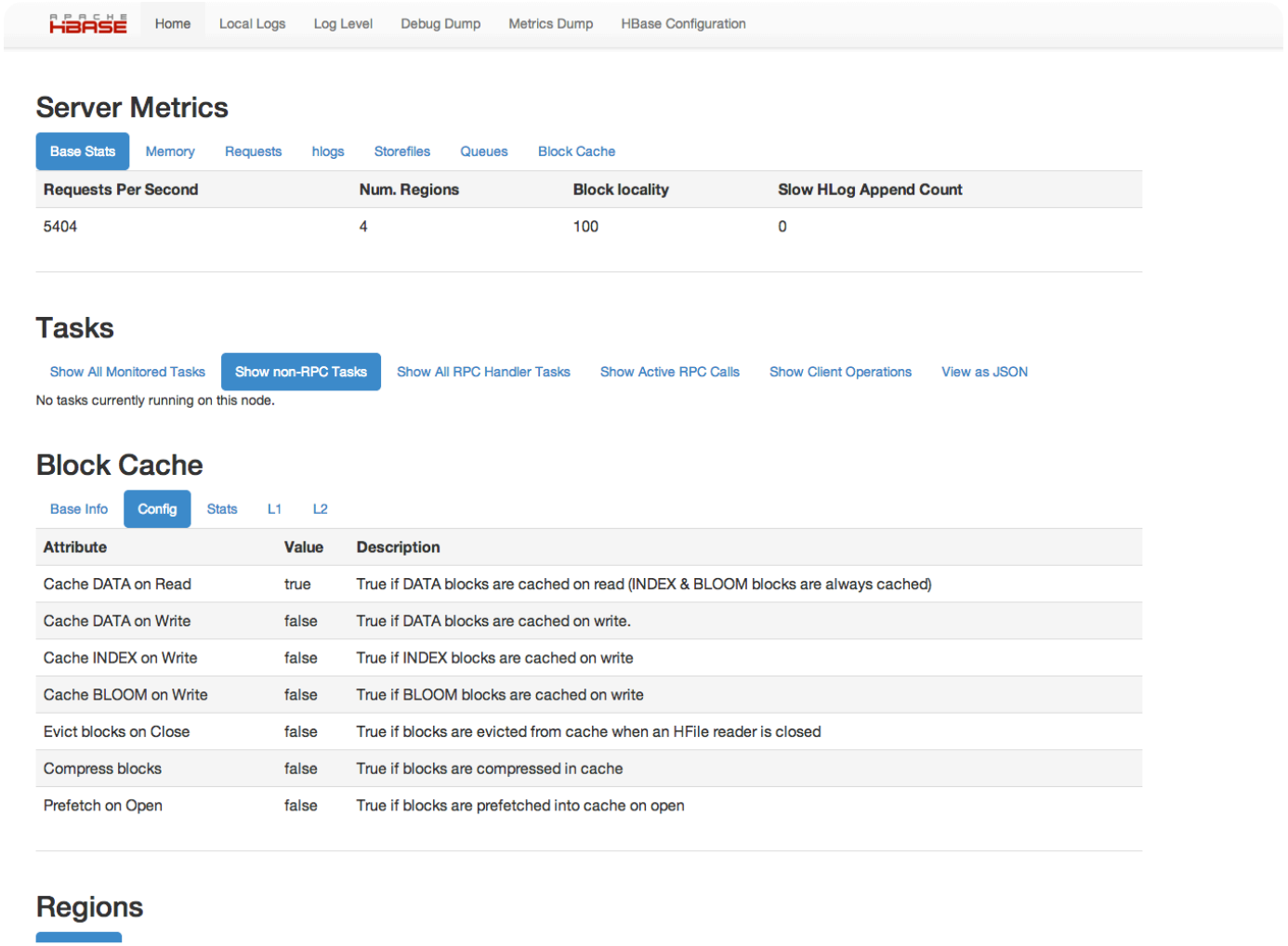
Block Cache Monitoring

Starting with HBase 0.98, the HBase Web UI includes the ability to monitor and report on the performance of the block cache. To view the block cache reports, see the Block Cache section of the region server UI. Following are a few examples of the reporting capabilities.

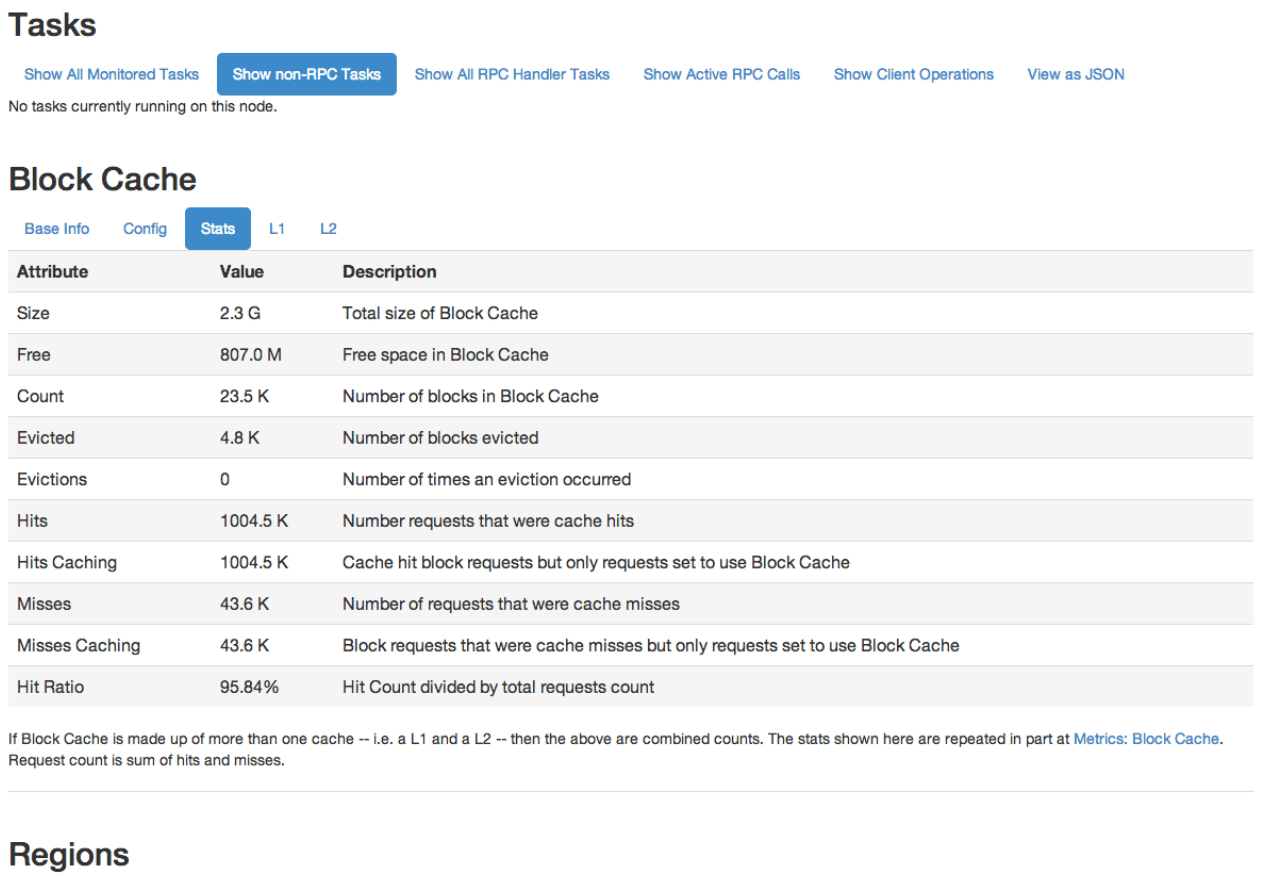
Basic Info shows the cache implementation.



Config shows all cache configuration options.



Stats shows statistics about the performance of the cache.



Tasks

Show All Monitored Tasks | Show non-RPC Tasks | Show All RPC Handler Tasks | Show Active RPC Calls | Show Client Operations | View as JSON

No tasks currently running on this node.

Block Cache

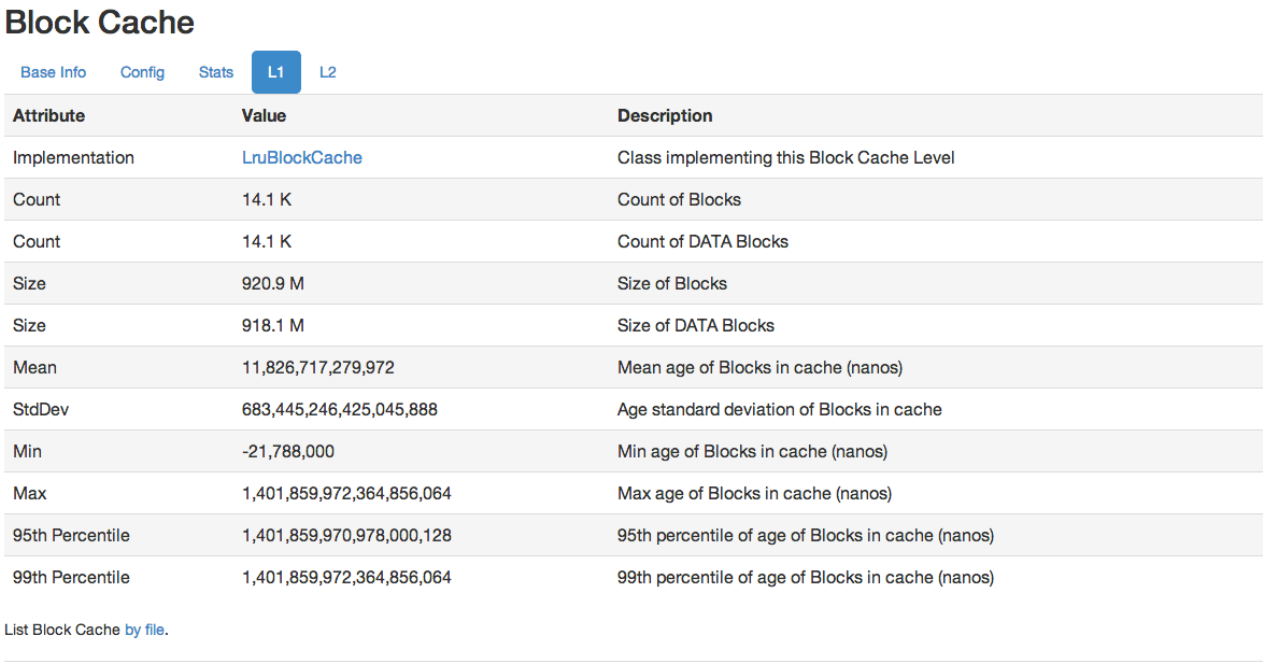
Base Info | Config | Stats | L1 | L2

Attribute	Value	Description
Size	2.3 G	Total size of Block Cache
Free	807.0 M	Free space in Block Cache
Count	23.5 K	Number of blocks in Block Cache
Evicted	4.8 K	Number of blocks evicted
Evictions	0	Number of times an eviction occurred
Hits	1004.5 K	Number requests that were cache hits
Hits Caching	1004.5 K	Cache hit block requests but only requests set to use Block Cache
Misses	43.6 K	Number of requests that were cache misses
Misses Caching	43.6 K	Block requests that were cache misses but only requests set to use Block Cache
Hit Ratio	95.84%	Hit Count divided by total requests count

If Block Cache is made up of more than one cache -- i.e. a L1 and a L2 -- then the above are combined counts. The stats shown here are repeated in part at Metrics: Block Cache. Request count is sum of hits and misses.

Regions

L1 and L2 show information about the L1 and L2 caches.



Block Cache

Base Info | Config | Stats | L1 | L2

Attribute	Value	Description
Implementation	LruBlockCache	Class implementing this Block Cache Level
Count	14.1 K	Count of Blocks
Count	14.1 K	Count of DATA Blocks
Size	920.9 M	Size of Blocks
Size	918.1 M	Size of DATA Blocks
Mean	11,826,717,279,972	Mean age of Blocks in cache (nanos)
StdDev	683,445,246,425,045,888	Age standard deviation of Blocks in cache
Min	-21,788,000	Min age of Blocks in cache (nanos)
Max	1,401,859,972,364,856,064	Max age of Blocks in cache (nanos)
95th Percentile	1,401,859,970,978,000,128	95th percentile of age of Blocks in cache (nanos)
99th Percentile	1,401,859,972,364,856,064	99th percentile of age of Blocks in cache (nanos)

List Block Cache by file.

This is not an exhaustive list of all the screens and reports available. Have a look in the Web UI.

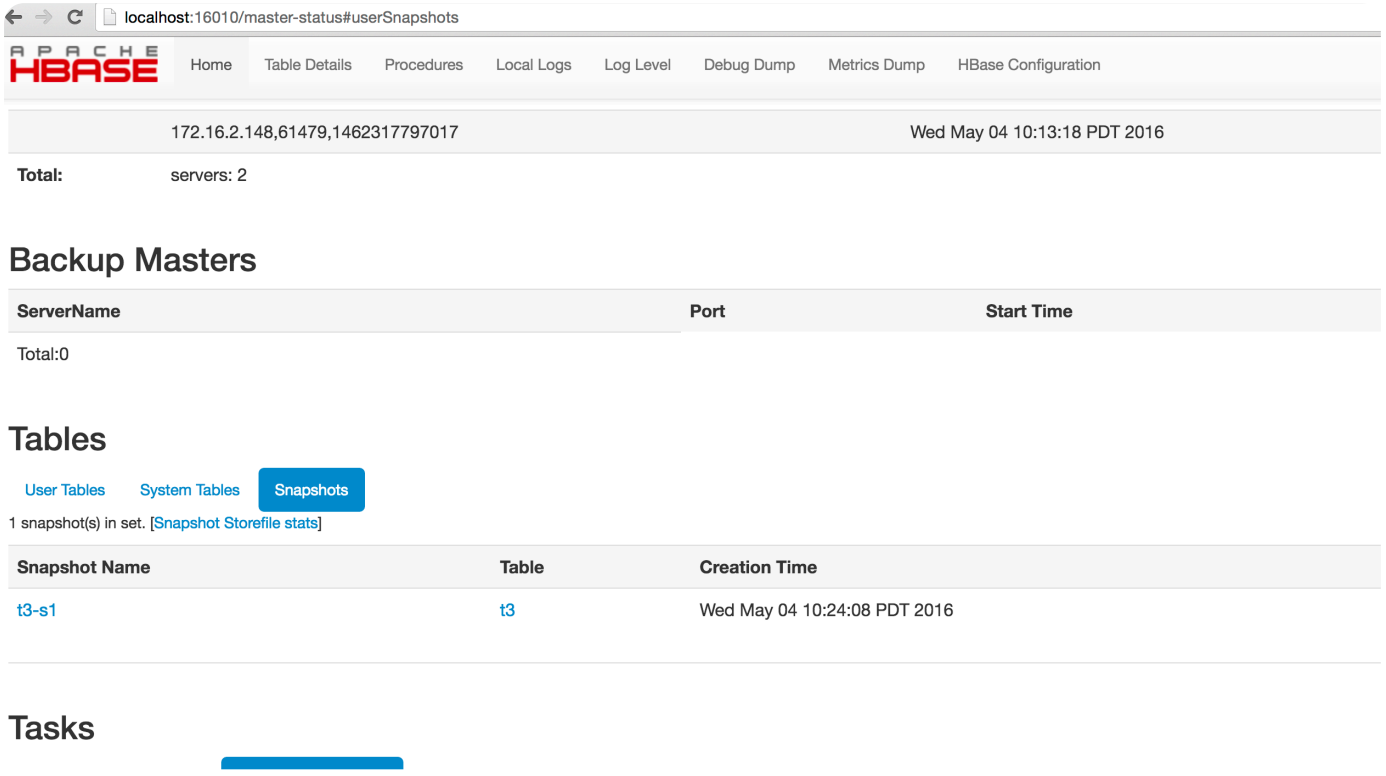
Snapshot Space Usage Monitoring

Starting with HBase 0.95, Snapshot usage information on individual snapshots was shown in the HBase Master Web UI. This was further enhanced starting with HBase 1.3 to show the total Storefile size of the Snapshot Set. The following metrics are shown in the Master Web UI with HBase 1.3 and later.

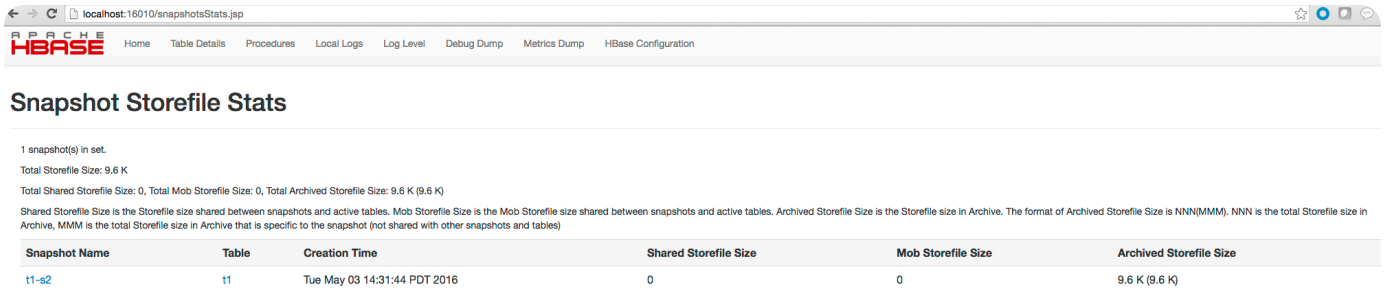
- Shared Storefile Size is the Storefile size shared between snapshots and active tables.
- Mob Storefile Size is the Mob Storefile size shared between snapshots and active tables.
- Archived Storefile Size is the Storefile size in Archive.

The format of Archived Storefile Size is NNN(MMM). NNN is the total Storefile size in Archive, MMM is the total Storefile size in Archive that is specific to the snapshot (not shared with other snapshots and tables).

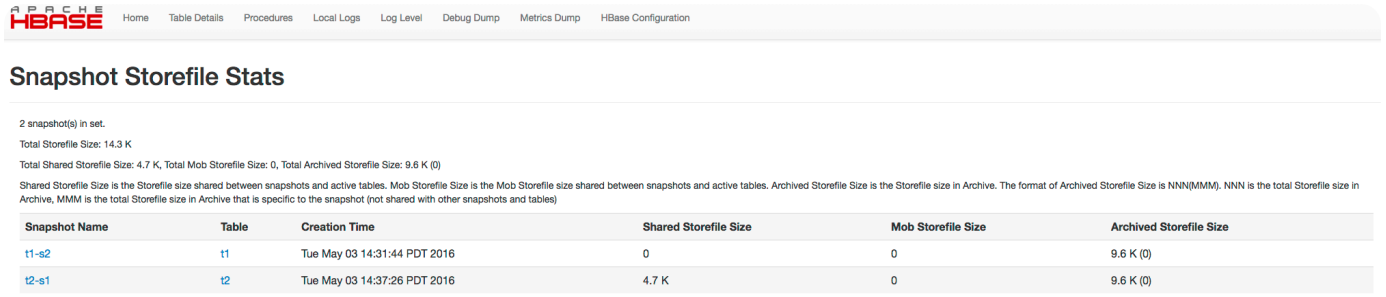
Master Snapshot Overview



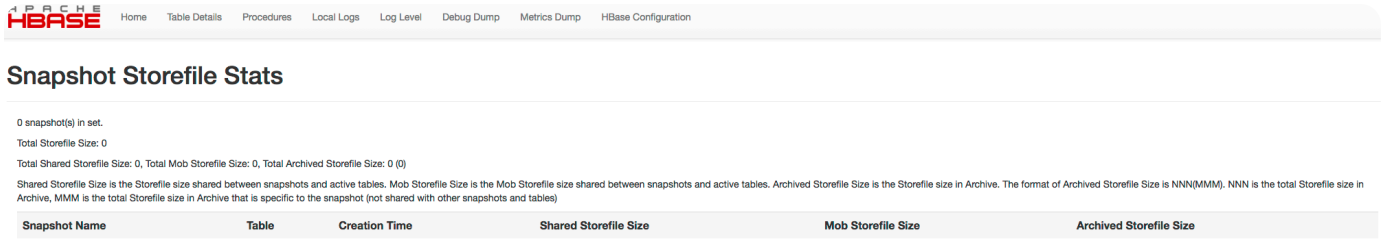
Snapshot Storefile Stats Example 1



Snapshot Storefile Stats Example 2



Empty Snapshot Storfile Stats Example



Cluster Replication

HBase provides a cluster replication mechanism which allows you to keep one cluster's state synchronized with that of another cluster, using the write-ahead log (WAL) of the source cluster to propagate the changes. Some use cases for cluster replication include:

- Backup and disaster recovery
- Data aggregation
- Geographic data distribution
- Online data ingestion combined with offline data analytics

Replication is enabled at the granularity of the column family. Before enabling replication for a column family, create the table and all column families to be replicated, on the destination cluster.

Replication is asynchronous as we send WAL to another cluster in background, which means that when you want to do recovery through replication, you could loss some data. To address this problem, we have introduced a new feature called synchronous replication. As the mechanism is a bit different so we use a separated section to describe it. Please see Synchronous Replication.

At present, there is compatibility problem if Replication and WAL Compression are used together. If you need to use Replication, it is recommended to set the `hbase.regionserver.wal.enablecompression` property to `false`. See (HBASE-26849) for details.

Replication Overview

Cluster replication uses a source-push methodology. An HBase cluster can be a source (also called master or active, meaning that it is the originator of new data), a destination (also called slave or passive, meaning that it receives data via replication), or can fulfill both roles at once. Replication is asynchronous, and the goal of replication is eventual consistency. When the source receives an edit to a column family with replication enabled, that edit is propagated to all destination clusters using the WAL for that for that column family on the RegionServer managing the relevant region.

When data is replicated from one cluster to another, the original source of the data is tracked via a cluster ID which is part of the metadata. In HBase 0.96 and newer (HBASE-7709), all clusters which have already consumed the data are also tracked. This prevents replication loops.

The WALs for each region server must be kept in HDFS as long as they are needed to replicate data to any slave cluster. Each region server reads from the oldest log it needs to replicate and keeps track of its progress processing WALs inside ZooKeeper to simplify failure recovery. The position marker which indicates a slave cluster's progress, as well as the queue of WALs to process, may be different for every slave cluster.

The clusters participating in replication can be of different sizes. The master cluster relies on randomization to attempt to balance the stream of replication on the slave clusters. It is expected that the slave cluster has storage capacity to hold the replicated data, as well as any data it is responsible for ingesting. If a slave cluster does run out of room, or is

inaccessible for other reasons, it throws an error and the master retains the WAL and retries the replication at intervals.

> **Consistency Across Replicated Clusters**
>
> How your application builds on top of the HBase API matters when replication is in play. HBase's replication system provides at-least-once delivery of client edits for an enabled column family to each configured destination cluster. In the event of failure to reach a given destination, the replication system will retry sending edits in a way that might repeat a given message. HBase provides two ways of replication, one is the original replication and the other is serial replication. In the previous way of replication, there is not a guaranteed order of delivery for client edits. In the event of a RegionServer failing, recovery of the replication queue happens independent of recovery of the individual regions that server was previously handling. This means that it is possible for the not-yet-replicated edits to be serviced by a RegionServer that is currently slower to replicate than the one that handles edits from after the failure.
>
> The combination of these two properties (at-least-once delivery and the lack of message ordering) means that some destination clusters may end up in a different state if your application makes use of operations that are not idempotent, e.g. Increments.
>
> To solve the problem, HBase now supports serial replication, which sends edits to destination cluster as the order of requests from client. See Serial Replication.

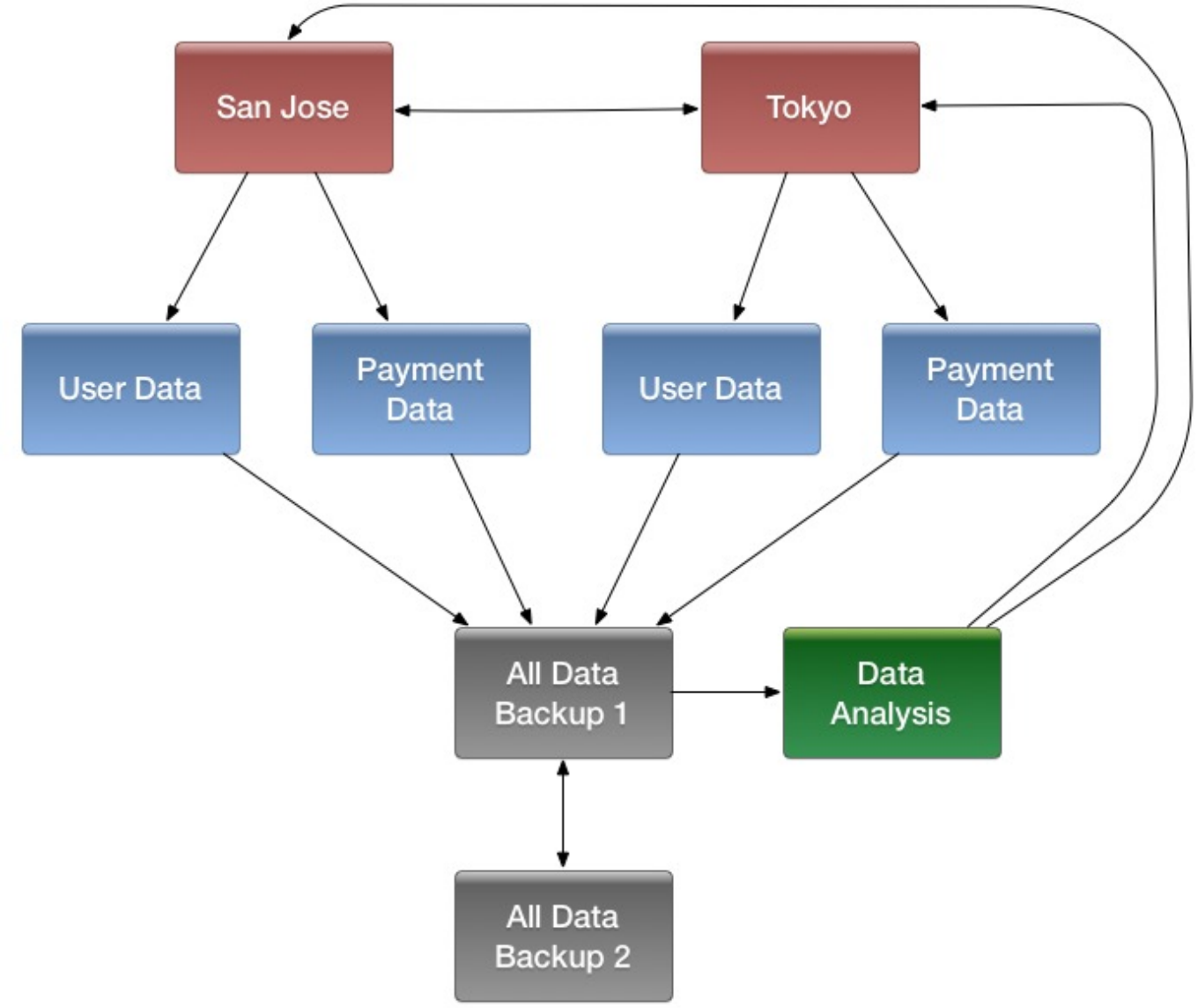
> **Terminology Changes**
>
> Previously, terms such as **master-master**, **master-slave**, and **cyclical** were used to describe replication relationships in HBase. These terms added confusion, and have been abandoned in favor of discussions about cluster topologies appropriate for different scenarios.

Cluster Topologies

- A central source cluster might propagate changes out to multiple destination clusters, for failover or due to geographic distribution.
- A source cluster might push changes to a destination cluster, which might also push its own changes back to the original cluster.
- Many different low-latency clusters might push changes to one centralized cluster for backup or resource-intensive data analytics jobs. The processed data might then be replicated back to the low-latency clusters.

Multiple levels of replication may be chained together to suit your organization's needs. The following diagram shows a hypothetical scenario. Use the arrows to follow the data paths.

Example of a Complex Cluster Replication Configuration



HBase replication borrows many concepts from the **statement-based replication** design used by MySQL. Instead of SQL statements, entire WALEdits (consisting of multiple cell inserts coming from Put and Delete operations on the clients) are replicated in order to maintain atomicity.

Managing and Configuring Cluster Replication

Cluster Configuration Overview

1. Configure and start the source and destination clusters. Create tables with the same names and column families on both the source and destination clusters, so that the destination cluster knows where to store data it will receive.
2. All hosts in the source and destination clusters should be reachable to each other.
3. If both clusters use the same ZooKeeper cluster, you must use a different `zookeeper.znode.parent`, because they cannot write in the same folder.

4. On the source cluster, in HBase Shell, add the destination cluster as a peer, using the `add_peer` command.
5. On the source cluster, in HBase Shell, enable the table replication, using the `enable_table_replication` command.
6. Check the logs to see if replication is taking place. If so, you will see messages like the following, coming from the ReplicationSource.

```
LOG.info("Replicating "+clusterId + " -> " + peerClusterId);
```

Serial Replication Configuration

See Serial Replication

Cluster Management Commands

add_peer `<ID>` `<CLUSTER_KEY>`

Adds a replication relationship between two clusters.

- ID — a unique string, which must not contain a hyphen.
- CLUSTER_KEY: composed using the following template, with appropriate place-holders: `hbase.zookeeper.quorum:hbase.zookeeper.property.clientPort:zookeeper.znode.parent`. This value can be found on the Master UI info page.
- STATE(optional): ENABLED or DISABLED, default value is ENABLED

list_peers

list all replication relationships known by this cluster

enable_peer `<ID>`

Enable a previously-disabled replication relationship

disable_peer `<ID>`

Disable a replication relationship. HBase will no longer send edits to that peer cluster, but it still keeps track of all the new WALs that it will need to replicate if and when it is re-enabled. WALs are retained when enabling or disabling replication as long as peers exist.

remove_peer `<ID>`

Disable and remove a replication relationship. HBase will no longer send edits to that peer cluster or keep track of WALs.

enable_table_replication `<TABLE_NAME>`

Enable the table replication switch for all its column families. If the table is not found in the destination cluster then it will create one with the same name and column families.

disable_table_replication `<TABLE_NAME>`

Disable the table replication switch for all its column families.

peer_modification_switch `<enable_or_disable>`, `<drain_procedures>`

Enabled/Disable peer modification operations, such as adding/removing replication peers. The second parameter means whether you want to wait until all existing peer modification procedures to finish before returning when disabling peer modification.

peer_modification_enabled

Check whether peer modification is enabled.

Migrate Across Different Replication Peer Storages

Starting from 2.6.0, we introduce a file system based `ReplicationPeerStorage`, which stores the replication peer state with files on HFile file system, instead of znodes on ZooKeeper. And we have also implemented a tool to copy replication peer state across different replication peer storages.

```
./bin/hbase copyreppeers <SRC_REPLICATION_PEER_STORAGE> <DST_REPLICATION_PEER_STO
RAGE>
```

To support doing the migrate online, we introduce a shell command called `peer_modificati on_switch`.

```
hbase> peer_modification_switch false, true
```

Use the above command can disable peer modification operations. The second `true` means you want to wait until all the existing replication peer modification procedures to finish before returning. After disabling the peer modification, it is safe for you to copy replication peer state with the above tool, and then update all the *hbase-site.xml* files in the cluster to specify the new replication peer storage, and finally trigger an online configuration update to load the new replication peer storage.

Serial Replication

Note: this feature is introduced in HBase 2.1

Function of serial replication

Serial replication supports to push logs to the destination cluster in the same order as logs reach to the source cluster.

Why need serial replication?

In replication of HBase, we push mutations to destination cluster by reading WAL in each region server. We have a queue for WAL files so we can read them in order of creation time. However, when region-move or RS failure occurs in source cluster, the hlog entries that are not pushed before region-move or RS-failure will be pushed by original RS(for region move) or another RS which takes over the remained hlog of dead RS(for RS failure), and the new entries for the same region(s) will be pushed by the RS which now serves the region(s), but they push the hlog entries of a same region concurrently without coordination.

This treatment can possibly lead to data inconsistency between source and destination clusters:

1. there are put and then delete written to source cluster.
2. due to region-move / RS-failure, they are pushed by different replication-source threads to peer cluster.
3. if delete is pushed to peer cluster before put, and flush and major-compact occurs in peer cluster before put is pushed to peer cluster, the delete is collected and the put remains in peer cluster, but in source cluster the put is masked by the delete, hence data inconsistency between source and destination clusters.

Serial replication configuration

Set the serial flag to true for a repliation peer. And the default serial flag is false.

- Add a new replication peer which serial flag is true

```
hbase> add_peer '1', CLUSTER_KEY => "server1.cie.com:2181:/hbase", SERIAL => tr
ue
```

- Set a replication peer's serial flag to false

```
hbase> set_peer_serial '1', false
```

- Set a replication peer's serial flag to true

```
hbase> set_peer_serial '1', true
```

The serial replication feature had been done firstly in HBASE-9465 and then reverted and redone in HBASE-20046. You can find more details in these issues.

Verifying Replicated Data

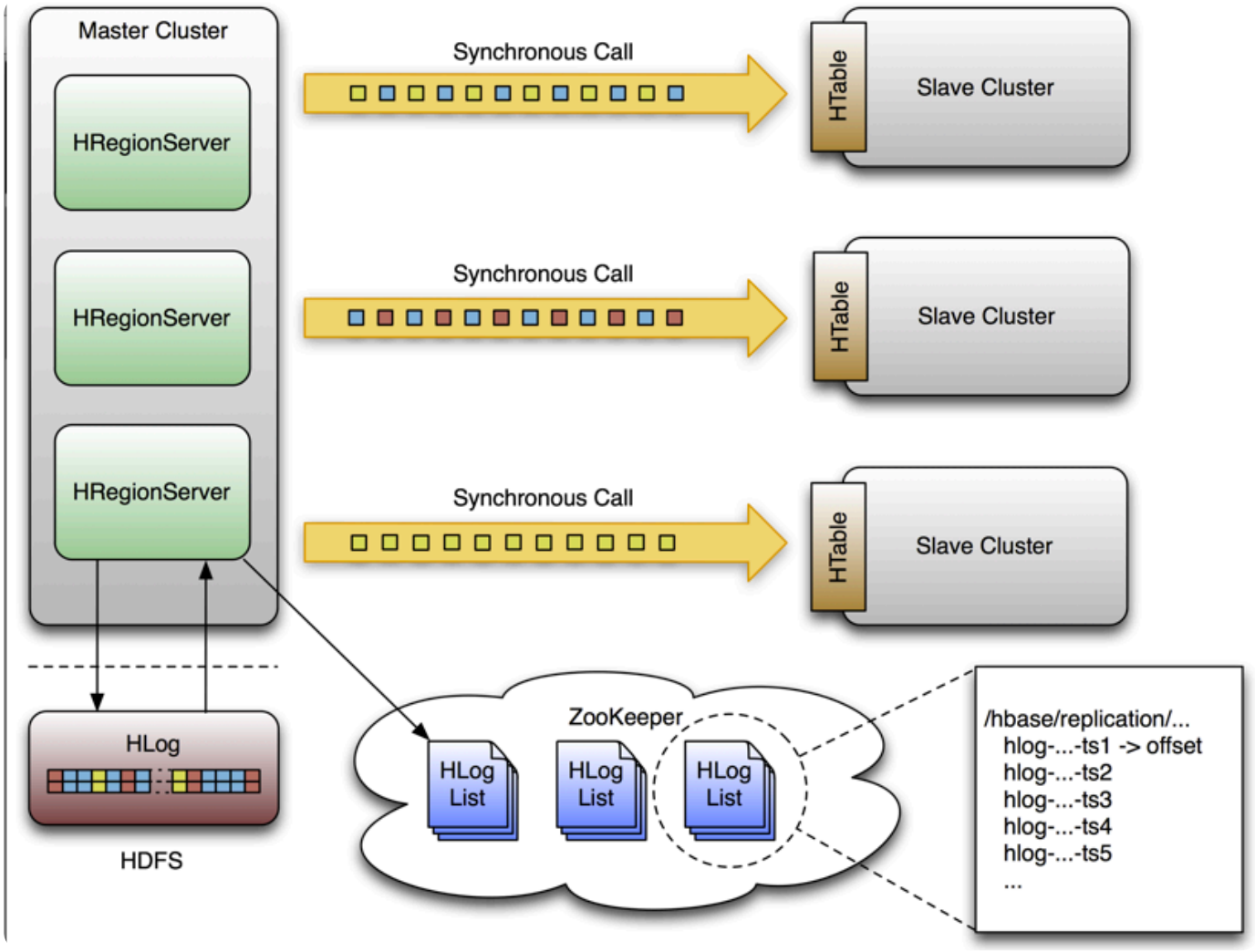
The `VerifyReplication` MapReduce job, which is included in HBase, performs a systematic comparison of replicated data between two different clusters. Run the VerifyReplication job on the master cluster, supplying it with the peer ID and table name to use for validation. You can limit the verification further by specifying a time range or specific families. The job's short name is `verifyrep`. To run the job, use a command like the following:

```
$ HADOOP_CLASSPATH=`${HBASE_HOME}/bin/hbase classpath` "${HADOOP_HOME}/bin/hadoo
p" jar "${HBASE_HOME}/hbase-mapreduce-VERSION.jar" verifyrep --starttime=<timesta
mp> --endtime=<timestamp> --families=<myFam> <ID> <tableName>
```

- The `VerifyReplication` command prints out `GOODROWS` and `BADROWS` counters to indicate rows that did and did not replicate correctly.

Detailed Information About Cluster Replication

Replication Architecture Overview



Life of a WAL Edit

A single WAL edit goes through several steps in order to be replicated to a slave cluster.

1. An HBase client uses a Put or Delete operation to manipulate data in HBase.
2. The region server writes the request to the WAL in a way allows it to be replayed if it is not written successfully.
3. If the changed cell corresponds to a column family that is scoped for replication, the edit is added to the queue for replication.
4. In a separate thread, the edit is read from the log, as part of a batch process. Only the KeyValues that are eligible for replication are kept. Replicable KeyValues are part of a column family whose schema is scoped GLOBAL, are not part of a catalog such as `hbase:meta`, did not originate from the target slave cluster, and have not already been consumed by the target slave cluster.
5. The edit is tagged with the master's UUID and added to a buffer. When the buffer is filled, or the reader reaches the end of the file, the buffer is sent to a random region server on the slave cluster.

6. The region server reads the edits sequentially and separates them into buffers, one buffer per table. After all edits are read, each buffer is flushed using Table, HBase's normal client. The master's UUID and the UUIDs of slaves which have already consumed the data are preserved in the edits they are applied, in order to prevent replication loops.
7. In the master, the offset for the WAL that is currently being replicated is registered in ZooKeeper.
8. The first three steps, where the edit is inserted, are identical.
9. Again in a separate thread, the region server reads, filters, and edits the log edits in the same way as above. The slave region server does not answer the RPC call.
10. The master sleeps and tries again a configurable number of times.
11. If the slave region server is still not available, the master selects a new subset of region server to replicate to, and tries again to send the buffer of edits.
12. Meanwhile, the WALs are rolled and stored in a queue in ZooKeeper. Logs that are **archived** by their region server, by moving them from the region server's log directory to a central log directory, will update their paths in the in-memory queue of the replicating thread.
13. When the slave cluster is finally available, the buffer is applied in the same way as during normal processing. The master region server will then replicate the backlog of logs that accumulated during the outage.

Spreading Queue Failover Load

When replication is active, a subset of region servers in the source cluster is responsible for shipping edits to the sink. This responsibility must be failed over like all other region server functions should a process or node crash. The following configuration settings are recommended for maintaining an even distribution of replication activity over the remaining live servers in the source cluster:

- Set `replication.source.maxretriesmultiplier` to `300`.
- Set `replication.source.sleepforretries` to `1` (1 second). This value, combined with the value of `replication.source.maxretriesmultiplier`, causes the retry cycle to last about 5 minutes.
- Set `replication.sleep.before.failover` to `30000` (30 seconds) in the source cluster site configuration.

Preserving Tags During Replication

By default, the codec used for replication between clusters strips tags, such as cell-level ACLs, from cells. To prevent the tags from being stripped, you can use a different codec which does not strip them. Configure `hbase.replication.rpc.codec` to use `org.apache.hadoop.hbase.codec.KeyValueCodecWithTags`, on both the source and sink RegionServers involved in the replication. This option was introduced in HBASE-10322.

Replication Internals

Replication State Storage

In HBASE-15867, we abstract two interfaces for storing replication state, `ReplicationPeerStorage` and `ReplicationQueueStorage`. The former one is for storing the replication peer related states, and the latter one is for storing the replication queue related states. HBASE-15867 is only half done, as although we have abstract these two interfaces, we still only have zookeeper based implementations.

And in HBASE-27110, we have implemented a file system based replication peer storage, to store replication peer state on file system. Of course you can still use the zookeeper based replication peer storage.

And in HBASE-27109, we have changed the replication queue storage from zookeeper based to hbase table based. See the below `Replication Queue State` in hbase:replication table section for more details.

Replication State in ZooKeeper

By default, the state is contained in the base node */hbase/replication*. Usually this nodes contains two child nodes, the peers znode is for storing replication peer state, and the rs znodes is for storing replication queue state. And if you choose the file system based replication peer storage, you will not see the peers znode. And starting from 3.0.0, we have moved the replication queue state to `hbase:replication` table (see below), so you will not see the rs znode.

The `Peers` Znode

The `peers` znode is stored in */hbase/replication/peers* by default. It consists of a list of all peer replication clusters, along with the status of each of them. The value of each peer is its cluster key, which is provided in the HBase Shell. The cluster key contains a list of ZooKeeper nodes in the cluster's quorum, the client port for the ZooKeeper quorum, and the base znode for HBase in HDFS on that cluster. Starting from 3.0.0, you can also

specify connection URI as a cluster key. See Connection URI for more details about connection URI.

The `RS` Znode

The `rs` znode contains a list of WAL logs which need to be replicated. This list is divided into a set of queues organized by region server and the peer cluster the region server is shipping the logs to. The rs znode has one child znode for each region server in the cluster. The child znode name is the region server's hostname, client port, and start code. This list includes both live and dead region servers.

The hbase:replication Table

After 3.0.0, the `Queue` has been stored in the hbase:replication table, where the row key is `<PeerId>-<ServerName>[/<SourceServerName>]`, the WAL group will be the qualifier, and the serialized ReplicationGroupOffset will be the value. The ReplicationGroupOffset includes the wal file of the corresponding queue (`<PeerId>-<ServerName>[/<SourceServerName>]`) and its offset. Because we track replication offset per queue instead of per file, we only need to store one replication offset per queue.

Other implementations for `ReplicationPeerStorage`

Starting from 2.6.0, we introduce a file system based `ReplicationPeerStorage`, which stores the replication peer state with files on HFile file system, instead of znodes on ZooKeeper. The layout is almost the same with znodes on zookeeper, the main difference is that, the HFile file system may not support atomic rename, so we use two files to store the state and when reading we will read them both and compare the timestamp to find out the newer one. So typically, you will see two peer config files. And for enable/disable state, we just touch a disabled file if the peer is disabled, and remove the file when enabling the peer.

Choosing Region Servers to Replicate To

When a master cluster region server initiates a replication source to a slave cluster, it first connects to the slave's ZooKeeper ensemble using the provided cluster key . It then scans the *rs/* directory to discover all the available sinks (region servers that are accepting incoming streams of edits to replicate) and randomly chooses a subset of them using a configured ratio which has a default value of 10%. For example, if a slave cluster has 150 machines, 15 will be chosen as potential recipient for edits that this master cluster region server sends. Because this selection is performed by each master region server, the probability that all slave region servers are used is very high, and this method works for

clusters of any size. For example, a master cluster of 10 machines replicating to a slave cluster of 5 machines with a ratio of 10% causes the master cluster region servers to choose one machine each at random.

A ZooKeeper watcher is placed on the `$zookeeper.znode.parent/rs` node of the slave cluster by each of the master cluster's region servers. This watch is used to monitor changes in the composition of the slave cluster. When nodes are removed from the slave cluster, or if nodes go down or come back up, the master cluster's region servers will respond by selecting a new pool of slave region servers to replicate to.

Keeping Track of Logs(based on ZooKeeper)

Each master cluster region server has its own znode in the replication znodes hierarchy. It contains one znode per peer cluster (if 5 slave clusters, 5 znodes are created), and each of these contain a queue of WALs to process. Each of these queues will track the WALs created by that region server, but they can differ in size. For example, if one slave cluster becomes unavailable for some time, the WALs should not be deleted, so they need to stay in the queue while the others are processed. See rs.failover.details for an example.

When a source is instantiated, it contains the current WAL that the region server is writing to. During log rolling, the new file is added to the queue of each slave cluster's znode just before it is made available. This ensures that all the sources are aware that a new log exists before the region server is able to append edits into it, but this operations is now more expensive. The queue items are discarded when the replication thread cannot read more entries from a file (because it reached the end of the last block) and there are other files in the queue. This means that if a source is up to date and replicates from the log that the region server writes to, reading up to the "end" of the current file will not delete the item in the queue.

A log can be archived if it is no longer used or if the number of logs exceeds `hbase.region server.maxlogs` because the insertion rate is faster than regions are flushed. When a log is archived, the source threads are notified that the path for that log changed. If a particular source has already finished with an archived log, it will just ignore the message. If the log is in the queue, the path will be updated in memory. If the log is currently being replicated, the change will be done atomically so that the reader doesn't attempt to open the file when has already been moved. Because moving a file is a NameNode operation , if the reader is currently reading the log, it won't generate any exception.

Keeping Track of Logs(based on hbase table)

After 3.0.0, for table based implementation, we have server name in row key, which means we will have lots of rows for a given peer.

For a normal replication queue, the WAL files belong to the region server that is still alive, all the WAL files are kept in memory, so we do not need to get the WAL files from replication queue storage. And for a recovered replication queue, we could get the WAL files of the dead region server by listing the old WAL directory on HDFS. So theoretically, we do not need to store every WAL file in replication queue storage. And what's more, we store the created time(usually) in the WAL file name, so for all the WAL files in a WAL group, we can sort them(actually we will sort them in the current replication framework), which means we only need to store one replication offset per queue. When starting a recovered replication queue, we will skip all the files before this offset, and start replicating from this offset.

For ReplicationLogCleaner, all the files before this offset can be deleted, otherwise not.

Reading, Filtering and Sending Edits

By default, a source attempts to read from a WAL and ship log entries to a sink as quickly as possible. Speed is limited by the filtering of log entries Only KeyValues that are scoped GLOBAL and that do not belong to catalog tables will be retained. Speed is also limited by total size of the list of edits to replicate per slave, which is limited to 64 MB by default. With this configuration, a master cluster region server with three slaves would use at most 192 MB to store data to replicate. This does not account for the data which was filtered but not garbage collected.

Once the maximum size of edits has been buffered or the reader reaches the end of the WAL, the source thread stops reading and chooses at random a sink to replicate to (from the list that was generated by keeping only a subset of slave region servers). It directly issues a RPC to the chosen region server and waits for the method to return. If the RPC was successful, the source determines whether the current file has been emptied or it contains more data which needs to be read. If the file has been emptied, the source deletes the znode in the queue. Otherwise, it registers the new offset in the log's znode. If the RPC threw an exception, the source will retry 10 times before trying to find a different sink.

Cleaning Logs

If replication is not enabled, the master's log-cleaning thread deletes old logs using a configured TTL. This TTL-based method does not work well with replication, because archived logs which have exceeded their TTL may still be in a queue. The default behavior is augmented so that if a log is past its TTL, the cleaning thread looks up every queue until it finds the log, while caching queues it has found. If the log is not found in any queues, the log will be deleted. The next time the cleaning process needs to look for a log, it starts by using its cached list.

> WALs are saved when replication is enabled or disabled as long as peers exist.

Region Server Failover

When no region servers are failing, keeping track of the logs in ZooKeeper adds no value. Unfortunately, region servers do fail, and since ZooKeeper is highly available, it is useful for managing the transfer of the queues in the event of a failure. Each of the master cluster region servers keeps a watcher on every other region server, in order to be notified when one dies (just as the master does). When a failure happens, they all race to create a znode called `lock` inside the dead region server's znode that contains its queues. The region server that creates it successfully then transfers all the queues to its own znode, one at a time since ZooKeeper does not support renaming queues. After queues are all transferred, they are deleted from the old location. The znodes that were recovered are renamed with the ID of the slave cluster appended with the name of the dead server.

Next, the master cluster region server creates one new source thread per copied queue, and each of the source threads follows the read/filter/ship pattern. The main difference is that those queues will never receive new data, since they do not belong to their new region server. When the reader hits the end of the last log, the queue's znode is deleted and the master cluster region server closes that replication source.

And starting from 2.5.0, the failover logic has been moved to SCP, where we add a SERVER_CRASH_CLAIM_REPLICATION_QUEUES step in SCP to claim the replication queues for a dead server. And starting from 3.0.0, where we changed the replication queue storage from zookeeper to table, the update to the replication queue storage is async, so we also need an extra step to add the missing replication queues before claiming.

The replication queue claiming (based on ZooKeeper)

Given a master cluster with 3 region servers replicating to a single slave with id `2`, the following hierarchy represents what the znodes layout could be at some point in time. The region servers' znodes all contain a `peers` znode which contains a single queue. The znode names in the queues represent the actual file names on HDFS in the form `address,port.timestamp`.

```
/hbase/replication/rs/
  1.1.1.1,60020,123456780/
    2/
      1.1.1.1,60020.1234  (Contains a position)
      1.1.1.1,60020.1265
  1.1.1.2,60020,123456790/
    2/
      1.1.1.2,60020.1214  (Contains a position)
      1.1.1.2,60020.1248
      1.1.1.2,60020.1312
  1.1.1.3,60020,    123456630/
    2/
      1.1.1.3,60020.1280  (Contains a position)
```

Assume that 1.1.1.2 loses its ZooKeeper session. The survivors will race to create a lock, and, arbitrarily, 1.1.1.3 wins. It will then start transferring all the queues to its local peers znode by appending the name of the dead server. Right before 1.1.1.3 is able to clean up the old znodes, the layout will look like the following:

```
/hbase/replication/rs/
  1.1.1.1,60020,123456780/
    2/
      1.1.1.1,60020.1234  (Contains a position)
      1.1.1.1,60020.1265
  1.1.1.2,60020,123456790/
    lock
    2/
      1.1.1.2,60020.1214  (Contains a position)
      1.1.1.2,60020.1248
      1.1.1.2,60020.1312
  1.1.1.3,60020,123456630/
    2/
      1.1.1.3,60020.1280  (Contains a position)

    2-1.1.1.2,60020,123456790/
      1.1.1.2,60020.1214  (Contains a position)
      1.1.1.2,60020.1248
      1.1.1.2,60020.1312
```

Some time later, but before 1.1.1.3 is able to finish replicating the last WAL from 1.1.1.2, it dies too. Some new logs were also created in the normal queues. The last region server will then try to lock 1.1.1.3's znode and will begin transferring all the queues. The new layout will be:

```
/hbase/replication/rs/
  1.1.1.1,60020,123456780/
    2/
      1.1.1.1,60020.1378  (Contains a position)

    2-1.1.1.3,60020,123456630/
      1.1.1.3,60020.1325  (Contains a position)
      1.1.1.3,60020.1401

    2-1.1.1.2,60020,123456790-1.1.1.3,60020,123456630/
      1.1.1.2,60020.1312  (Contains a position)
  1.1.1.3,60020,123456630/
    lock
    2/
      1.1.1.3,60020.1325  (Contains a position)
      1.1.1.3,60020.1401

    2-1.1.1.2,60020,123456790/
      1.1.1.2,60020.1312  (Contains a position)
```

The replication queue claiming(based on hbase table)

Given a master cluster with 3 region servers replicating to a single slave with id `2`, the following info represents what the storage layout of queue in the hbase:replication at some point in time. Row key is `<PeerId>-<ServerName>[/<SourceServerName>]`, and value is WAL && Offset.

```
  <PeerId>-<ServerName>[/<SourceServerName>]                        WAL && Offset
  2-1.1.1.1,60020,123456780                              1.1.1.1,60020.1234  (Conta
ins a position)
  2-1.1.1.2,60020,123456790                              1.1.1.2,60020.1214  (Conta
ins a position)
  2-1.1.1.3,60020,123456630                              1.1.1.3,60020.1280  (Conta
ins a position)
```

Assume that 1.1.1.2 failed. The survivors will claim queue of that, and, arbitrarily, 1.1.1.3 wins. It will claim all the queue of 1.1.1.2, including removing the row of a replication queue, and inserting a new row(where we change the server name to the region server which claims the queue). Finally, the layout will look like the following:

```
  <PeerId>-<ServerName>[/<SourceServerName>]                      WAL && Offset
  2-1.1.1.1,60020,123456780                              1.1.1.1,60020.1234  (Conta
ins a position)
  2-1.1.1.3,60020,123456630                              1.1.1.3,60020.1280  (Conta
ins a position)
  2-1.1.1.3,60020,123456630 1.1.1.2,60020,123456790    1.1.1.2,60020.1214  (Conta
ins a position)
```

Replication Metrics

The following metrics are exposed at the global region server level and at the peer level:

`source.sizeOfLogQueue`

number of WALs to process (excludes the one which is being processed) at the Replication source

`source.shippedOps`

number of mutations shipped

`source.logEditsRead`

number of mutations read from WALs at the replication source

`source.ageOfLastShippedOp`

age of last batch that was shipped by the replication source

`source.completedLogs`

The number of write-ahead-log files that have completed their acknowledged sending to the peer associated with this source. Increments to this metric are a part of normal operation of HBase replication.

`source.completedRecoverQueues`

The number of recovery queues this source has completed sending to the associated peer. Increments to this metric are a part of normal recovery of HBase replication in the face of failed Region Servers.

`source.uncleanlyClosedLogs`

The number of write-ahead-log files the replication system considered completed after reaching the end of readable entries in the face of an uncleanly closed file.

`source.ignoredUncleanlyClosedLogContentsInBytes`

When a write-ahead-log file is not closed cleanly, there will likely be some entry that has been partially serialized. This metric contains the number of bytes of such entries the HBase replication system believes were remaining at the end of files skipped in the face of an uncleanly closed file. Those bytes should either be in different file or represent a client write that was not acknowledged.

`source.restartedLogReading`

The number of times the HBase replication system detected that it failed to correctly parse a cleanly closed write-ahead-log file. In this circumstance, the system replays the entire log from the beginning, ensuring that no edits fail to be acknowledged by the associated peer. Increments to this metric indicate that the HBase replication system is having difficulty correctly handling failures in the underlying distributed storage system. No dataloss should occur, but you should check Region Server log files for details of the failures.

`source.repeatedLogFileBytes`

When the HBase replication system determines that it needs to replay a given write-ahead-log file, this metric is incremented by the number of bytes the replication system believes had already been acknowledged by the associated peer prior to starting over.

`source.closedLogsWithUnknownFileLength`

Incremented when the HBase replication system believes it is at the end of a write-ahead-log file but it can not determine the length of that file in the underlying distributed storage system. Could indicate dataloss since the replication system is unable to determine if the end of readable entries lines up with the expected end of the file. You should check Region Server log files for details of the failures.

Replication Configuration Options

Option	Description	Default
zookeeper.znode.parent	The name of the base ZooKeeper znode used for HBase	/hbase
zookeeper.znode.replication	The name of the base znode used for replication	replication
zookeeper.znode.replication.peers	The name of the peer znode	peers

Option	Description	Default
zookeeper.znode.replication.peers.state	The name of peer-state znode	peer-state
zookeeper.znode.replication.rs	The name of the rs znode	rs
replication.sleep.before.failover	How many milliseconds a worker should sleep before attempting to replicate a dead region server's WAL queues.	
replication.executor.workers	The number of region servers a given region server should attempt to failover simultaneously.	1
hbase.replication.peer.storage.impl	The replication peer storage implementation	zookeeper
hbase.replication.peers.directory	The directory for storing replication peer state, when filesystem replication peer storage is specified	peers
hbase.replication.queue.table.name	The table for storing replication queue state	hbase:replication
hbase.replication.queue.storage.impl	The replication queue storage implementation	TableReplicationQueueStorage

Monitoring Replication Status

You can use the HBase Shell command `status 'replication'` to monitor the replication status on your cluster. The command has three variations:

- `status 'replication'` — prints the status of each source and its sinks, sorted by hostname.
- `status 'replication', 'source'` — prints the status for each replication source, sorted by hostname.
- `status 'replication', 'sink'` — prints the status for each replication sink, sorted by hostname.

Understanding the output

The command output will vary according to the state of replication. For example right after a restart and if destination peer is not reachable, no replication source threads would be running, so no metrics would get displayed:

```
hbase01.home:
SOURCE: PeerID=1
Normal Queue: 1
No Reader/Shipper threads runnning yet.
SINK: TimeStampStarted=1591985197350, Waiting for OPs...
```

Under normal circumstances, a healthy, active-active replication deployment would show the following:

```
    hbase01.home:
      SOURCE: PeerID=1
         Normal Queue: 1
           AgeOfLastShippedOp=0, TimeStampOfLastShippedOp=Fri Jun 12 18:49:23 BST
2020, SizeOfLogQueue=1, EditsReadFromLogQueue=1, OpsShippedToTarget=1, TimeStampO
fNextToReplicate=Fri Jun 12 18:49:23 BST 2020, Replication Lag=0
      SINK: TimeStampStarted=1591983663458, AgeOfLastAppliedOp=0, TimeStampsOfLas
tAppliedOp=Fri Jun 12 18:57:18 BST 2020
```

The definition for each of these metrics is detailed below:

Type	Metric Name	Description
Source	AgeOfLastShippedOp	How long last successfully shipped edit took to effectively get replicated on target.
Source	TimeStampOfLastShippedOp	The actual date of last successful edit shipment.
Source	SizeOfLogQueue	Number of wal files on this given queue.
Source	EditsReadFromLogQueue	How many edits have been read from this given queue since this source thread started.
Source	OpsShippedToTarget	How many edits have been shipped to target since this source thread started.

Type	Metric Name	Description
Source	TimeStampOfNextToReplicate	Date of the current edit been at tempted to replicate.
Source	Replication Lag	The elapsed time (in millis), sinc e the last edit to replicate was r ead by this source thread and effectively replicated to target
Sink	TimeStampStarted	Date (in millis) of when this Sink thread started.
Sink	AgeOfLastAppliedOp	How long it took to apply the la st successful shipped edit.
Sink	TimeStampsOfLastAppliedOp	Date of last successful applied edit.

Growing values for `Source.TimeStampsOfLastAppliedOp` and/or `Source.Replication Lag` would indicate replication delays. If those numbers keep going up, while `Source.TimeStamp OfLastShippedOp`, `Source.EditsReadFromLogQueue`, `Source.OpsShippedToTarget` or `Source.Time StampOfNextToReplicate` do not change at all, then replication flow is failing to progress, and there might be problems within clusters communication. This could also happen if replication is manually paused (via hbase shell `disable_peer` command, for example), but data keeps getting ingested in the source cluster tables.

Replication Observability Framework

The core idea is to create `replication marker rows` periodically and insert them into WAL. These marker rows will help track the replication delays/bugs back to the `originating regi on server, WAL and timestamp of occurrence`. This tracker rows' WAL entries are interleaved with the regular table WAL entries and have a very high chance of running into the same replication delays/bugs that the user tables are seeing. Details as follows:

REPLICATION.WALEVENTTRACKER table

Create a new table called `REPLICATION.WALEVENTTRACKER` table and persist all the WAL events (like `ACTIVE`, `ROLLING`, `ROLLED`) to this table.
The properties of this table are: Replication is set to 0, Block Cache is Disabled, Max versions is 1, TTL is 1 year.

This table has single ColumnFamily: `info`

`info` contains multiple qualifiers:

- `info:region_server_name`
- `info:wal_name`
- `info:timestamp`
- `info:wal_state`
- `info:wal_length`

Whenever we roll a WAL (`old-wal-name` → `new-wal-name`), it will create 3 rows in this table.

`<region_server_name>, <old-wal-name>, <current timestamp>, <ROLLING>, <length of old-wal-name>`

`<region_server_name>, <old-wal-name>, <current timestamp>, <ROLLED>, <length of old-wal-name>`

`<region_server_name>, <new-wal-name>, <current timestamp>, <ACTIVE>, 0`

Configuration

To enable persisting WAL events, there is a configuration property: `hbase.regionserver.wal.event.tracker.enabled` (defaults to false)

REPLICATION.SINK_TRACKER table

Create a new table called `REPLICATION.SINK_TRACKER`.

The properties of this table are: Replication is set to 0, Block Cache is Disabled, Max versions is 1, TTL is 1 year.

This table has single ColumnFamily: `info`

`info` contains multiple qualifiers:

- `info:region_server_name`
- `info:wal_name`
- `info:timestamp`
- `info:offset`

Configuration

To create the above table, there is a configuration property: `hbase.regionserver.replication.sink.tracker.enabled` (defaults to false)

ReplicationMarker Chore

We introduced a new chore called `ReplicationMarkerChore` which will create the marker rows periodically into active WAL. The marker rows has the following metadata: `region_server_name, wal_name, timestamp and offset within WAL`. These markers are replicated (with special handling) and they are persisted into a sink side table `REPLICATION.SINK_TRACKER`.

Configuration:

`ReplicationMarkerChore` is enabled with configuration property: `hbase.regionserver.replication.marker.enabled` (defaults to false) and the period at which it creates marker rows is controlled by `hbase.regionserver.replication.marker.chore.duration` (defaults to 30 seconds). Sink cluster can choose to process these marker rows and persist to `REPLICATION.SINK_TRACKER` table or it can ignore these rows. This behavior is controlled by configuration property `hbase.regionserver.replication.sink.tracker.enabled` (defaults to false). If set to false, it will ignore the marker rows.

How to enable end-to-end feature ?

To use this whole feature, we will need to enable the above configuration properties in 2 phases/releases.
In first phase/release, set the following configuration properties to `true`:

- `hbase.regionserver.wal.event.tracker.enabled`: This will just persist all the WAL events to REPLICATION.WALEVENTTRACKER table.
- `hbase.regionserver.replication.sink.tracker.enabled`: This will create REPLICATION.SINK_TRACKER table and will process special marker rows coming from source cluster.

In second phase/release, set the following configuration property to `true`:

- `hbase.regionserver.replication.marker.enabled`: This will create marker rows periodically and sink cluster will persist these marker rows in `REPLICATION.SINK_TRACKER` table.

Running Multiple Workloads on a Single Cluster

HBase provides the following mechanisms for managing the performance of a cluster handling multiple workloads:

- Quotas
- Request Queues
- Multiple-Typed Queues

Quotas

HBASE-11598 introduces RPC quotas, which allow you to throttle requests based on the following limits:

1. The number or size of requests(read, write, or read+write) in a given timeframe
2. The number of tables allowed in a namespace

These limits can be enforced for a specified user, table, or namespace.

Enabling Quotas

Quotas are disabled by default. To enable the feature, set the `hbase.quota.enabled` property to `true` in *hbase-site.xml* file for all cluster nodes.

General Quota Syntax

1. THROTTLE_TYPE can be expressed as READ, WRITE, or the default type(read + write).
2. Timeframes can be expressed in the following units: `sec`, `min`, `hour`, `day`
3. Request sizes can be expressed in the following units: `B` (bytes), `K` (kilobytes), `M` (megabytes), `G` (gigabytes), `T` (terabytes), `P` (petabytes)
4. Numbers of requests are expressed as an integer followed by the string `req`
5. Limits relating to time are expressed as req/time or size/time. For instance `10req/day` or `100P/hour`.
6. Numbers of tables or regions are expressed as integers.

Setting Request Quotas

You can set quota rules ahead of time, or you can change the throttle at runtime. The change will propagate after the quota refresh period has expired. This expiration period defaults to 5 minutes. To change it, modify the `hbase.quota.refresh.period` property in `hbase-site.xml`. This property is expressed in milliseconds and defaults to `300000`.

```
# Limit user u1 to 10 requests per second
hbase> set_quota TYPE => THROTTLE, USER => 'u1', LIMIT => '10req/sec'

# Limit user u1 to 10 read requests per second
hbase> set_quota TYPE => THROTTLE, THROTTLE_TYPE => READ, USER => 'u1', LIMIT =>
'10req/sec'

# Limit user u1 to 10 M per day everywhere
hbase> set_quota TYPE => THROTTLE, USER => 'u1', LIMIT => '10M/day'

# Limit user u1 to 10 M write size per sec
hbase> set_quota TYPE => THROTTLE, THROTTLE_TYPE => WRITE, USER => 'u1', LIMIT =>
'10M/sec'

# Limit user u1 to 5k per minute on table t2
hbase> set_quota TYPE => THROTTLE, USER => 'u1', TABLE => 't2', LIMIT => '5K/min'

# Limit user u1 to 10 read requests per sec on table t2
hbase> set_quota TYPE => THROTTLE, THROTTLE_TYPE => READ, USER => 'u1', TABLE =>
't2', LIMIT => '10req/sec'

# Remove an existing limit from user u1 on namespace ns2
hbase> set_quota TYPE => THROTTLE, USER => 'u1', NAMESPACE => 'ns2', LIMIT => NON
E

# Limit all users to 10 requests per hour on namespace ns1
hbase> set_quota TYPE => THROTTLE, NAMESPACE => 'ns1', LIMIT => '10req/hour'

# Limit all users to 10 T per hour on table t1
hbase> set_quota TYPE => THROTTLE, TABLE => 't1', LIMIT => '10T/hour'

# Remove all existing limits from user u1
hbase> set_quota TYPE => THROTTLE, USER => 'u1', LIMIT => NONE

# List all quotas for user u1 in namespace ns2
hbase> list_quotas USER => 'u1, NAMESPACE => 'ns2'

# List all quotas for namespace ns2
hbase> list_quotas NAMESPACE => 'ns2'

# List all quotas for table t1
hbase> list_quotas TABLE => 't1'

# list all quotas
hbase> list_quotas
```

You can also place a global limit and exclude a user or a table from the limit by applying the `GLOBAL_BYPASS` property.

```
hbase> set_quota NAMESPACE => 'ns1', LIMIT => '100req/min'               # a per-
namespace request limit
hbase> set_quota USER => 'u1', GLOBAL_BYPASS => true                     # user u
1 is not affected by the limit
```

Enabling, Disabling, and Checking RPC Throttling

HBase provides shell commands to control RPC throttling at runtime. When throttling is disabled, HBase will not apply any request throttling. This can be useful in production environments that require temporary unthrottled operation.

The following HBase shell commands are available:

```
# Enable RPC throttling
hbase> enable_rpc_throttle

# Disable RPC throttling
hbase> disable_rpc_throttle

# Check whether RPC throttling is enabled
hbase> rpc_throttle_enabled
```

`enable_rpc_throttle` and `disable_rpc_throttle` return the previous RPC throttling state as a boolean value. `rpc_throttle_enabled` returns the current state.

> If no quotas are configured, RPC throttling is not applied, and enabling or disabling throttling will always return false.

Setting Namespace Quotas

You can specify the maximum number of tables or regions allowed in a given namespace, either when you create the namespace or by altering an existing namespace, by setting the `hbase.namespace.quota.maxtables property` on the namespace.

Limiting Tables Per Namespace

```
# Create a namespace with a max of 5 tables
```

```
hbase> create_namespace 'ns1', {'hbase.namespace.quota.maxtables'=>'5'}

# Alter an existing namespace to have a max of 8 tables
hbase> alter_namespace 'ns2', {METHOD => 'set', 'hbase.namespace.quota.maxtable
s'=>'8'}

# Show quota information for a namespace
hbase> describe_namespace 'ns2'

# Alter an existing namespace to remove a quota
hbase> alter_namespace 'ns2', {METHOD => 'unset', NAME=>'hbase.namespace.quota.ma
xtables'}
```

Limiting Regions Per Namespace

```
# Create a namespace with a max of 10 regions
hbase> create_namespace 'ns1', {'hbase.namespace.quota.maxregions'=>'10'}

# Show quota information for a namespace
hbase> describe_namespace 'ns1'

# Alter an existing namespace to have a max of 20 tables
hbase> alter_namespace 'ns2', {METHOD => 'set', 'hbase.namespace.quota.maxregion
s'=>'20'}

# Alter an existing namespace to remove a quota
hbase> alter_namespace 'ns2', {METHOD => 'unset', NAME=> 'hbase.namespace.quota.m
axregions'}
```

Request Queues

If no throttling policy is configured, when the RegionServer receives multiple requests, they are now placed into a queue waiting for a free execution slot (HBASE-6721). The simplest queue is a FIFO queue, where each request waits for all previous requests in the queue to finish before running. Fast or interactive queries can get stuck behind large requests.

If you are able to guess how long a request will take, you can reorder requests by pushing the long requests to the end of the queue and allowing short requests to preempt them. Eventually, you must still execute the large requests and prioritize the new requests behind them. The short requests will be newer, so the result is not terrible, but still suboptimal compared to a mechanism which allows large requests to be split into multiple smaller ones.

HBASE-10993 introduces such a system for deprioritizing long-running scanners. There are two types of queues, `fifo` and `deadline`. To configure the type of queue used, configure the `hbase.ipc.server.callqueue.type` property in `hbase-site.xml`. There is no way to estimate how long each request may take, so de-prioritization only affects scans, and is based on the number of "next" calls a scan request has made. An assumption is made that when you are doing a full table scan, your job is not likely to be interactive, so if there are concurrent requests, you can delay long-running scans up to a limit tunable by setting the `hbase.ipc.server.queue.max.call.delay` property. The slope of the delay is calculated by a simple square root of `(numNextCall * weight)` where the weight is configurable by setting the `hbase.ipc.server.scan.vtime.weight` property.

Multiple-Typed Queues

You can also prioritize or deprioritize different kinds of requests by configuring a specified number of dedicated handlers and queues. You can segregate the scan requests in a single queue with a single handler, and all the other available queues can service short `Get` requests.

You can adjust the IPC queues and handlers based on the type of workload, using static tuning options. This approach is an interim first step that will eventually allow you to change the settings at runtime, and to dynamically adjust values based on the load.

Multiple Queues

To avoid contention and separate different kinds of requests, configure the `hbase.ipc.server.callqueue.handler.factor` property, which allows you to increase the number of queues and control how many handlers can share the same queue., allows admins to increase the number of queues and decide how many handlers share the same queue.

Using more queues reduces contention when adding a task to a queue or selecting it from a queue. You can even configure one queue per handler. The trade-off is that if some queues contain long-running tasks, a handler may need to wait to execute from that queue rather than stealing from another queue which has waiting tasks.

Read and Write Queues

With multiple queues, you can now divide read and write requests, giving more priority (more queues) to one or the other type. Use the `hbase.ipc.server.callqueue.read.ratio` property to choose to serve more reads or more writes.

Get and Scan Queues

Similar to the read/write split, you can split gets and scans by tuning the `hbase.ipc.server.callqueue.scan.ratio` property to give more priority to gets or to scans. A scan ratio of `0.1` will give more queue/handlers to the incoming gets, which means that more gets can be processed at the same time and that fewer scans can be executed at the same time. A value of `0.9` will give more queue/handlers to scans, so the number of scans executed will increase and the number of gets will decrease.

Space Quotas

HBASE-16961 introduces a new type of quotas for HBase to leverage: filesystem quotas. These "space" quotas limit the amount of space on the filesystem that HBase namespaces and tables can consume. If a user, malicious or ignorant, has the ability to write data into HBase, with enough time, that user can effectively crash HBase (or worse HDFS) by consuming all available space. When there is no filesystem space available, HBase crashes because it can no longer create/sync data to the write-ahead log.

This feature allows a for a limit to be set on the size of a table or namespace. When a space quota is set on a namespace, the quota's limit applies to the sum of usage of all tables in that namespace. When a table with a quota exists in a namespace with a quota, the table quota takes priority over the namespace quota. This allows for a scenario where a large limit can be placed on a collection of tables, but a single table in that collection can have a fine-grained limit set.

The existing `set_quota` and `list_quota` HBase shell commands can be used to interact with space quotas. Space quotas are quotas with a `TYPE` of `SPACE` and have `LIMIT` and `POLICY` attributes. The `LIMIT` is a string that refers to the amount of space on the filesystem that the quota subject (e.g. the table or namespace) may consume. For example, valid values of `LIMIT` are `'10G'`, `'2T'`, or `'256M'`. The `POLICY` refers to the action that HBase

will take when the quota subject's usage exceeds the `LIMIT`. The following are valid `POLICY` values.

- `NO_INSERTS` - No new data may be written (e.g. `Put`, `Increment`, `Append`).
- `NO_WRITES` - Same as `NO_INSERTS` but `Deletes` are also disallowed.
- `NO_WRITES_COMPACTIONS` - Same as `NO_WRITES` but compactions are also disallowed.
  - This policy only prevents user-submitted compactions. System can still run compactions.
- `DISABLE` - The table(s) are disabled, preventing all read/write access.

Setting simple space quotas

```
# Sets a quota on the table 't1' with a limit of 1GB, disallowing Puts/Increments/Appends when the table exceeds 1GB
hbase> set_quota TYPE => SPACE, TABLE => 't1', LIMIT => '1G', POLICY => NO_INSERTS

# Sets a quota on the namespace 'ns1' with a limit of 50TB, disallowing Puts/Increments/Appends/Deletes
hbase> set_quota TYPE => SPACE, NAMESPACE => 'ns1', LIMIT => '50T', POLICY => NO_WRITES

# Sets a quota on the table 't3' with a limit of 2TB, disallowing any writes and compactions when the table exceeds 2TB.
hbase> set_quota TYPE => SPACE, TABLE => 't3', LIMIT => '2T', POLICY => NO_WRITES_COMPACTIONS

# Sets a quota on the table 't2' with a limit of 50GB, disabling the table when it exceeds 50GB
hbase> set_quota TYPE => SPACE, TABLE => 't2', LIMIT => '50G', POLICY => DISABLE
```

Consider the following scenario to set up quotas on a namespace, overriding the quota on tables in that namespace

Table and Namespace space quotas

```
hbase> create_namespace 'ns1'
hbase> create 'ns1:t1'
hbase> create 'ns1:t2'
hbase> create 'ns1:t3'
hbase> set_quota TYPE => SPACE, NAMESPACE => 'ns1', LIMIT => '100T', POLICY => NO_INSERTS
```

```
hbase> set_quota TYPE => SPACE, TABLE => 'ns1:t2', LIMIT => '200G', POLICY => NO_
WRITES
hbase> set_quota TYPE => SPACE, TABLE => 'ns1:t3', LIMIT => '20T', POLICY => NO_W
RITES
```

In the above scenario, the tables in the namespace `ns1` will not be allowed to consume more than 100TB of space on the filesystem among each other. The table 'ns1:t2' is only allowed to be 200GB in size, and will disallow all writes when the usage exceeds this limit. The table 'ns1:t3' is allowed to grow to 20TB in size and also will disallow all writes then the usage exceeds this limit. Because there is no table quota on 'ns1:t1', this table can grow up to 100TB, but only if 'ns1:t2' and 'ns1:t3' have a usage of zero bytes. Practically, it's limit is 100TB less the current usage of 'ns1:t2' and 'ns1:t3'.

Disabling Automatic Space Quota Deletion

By default, if a table or namespace is deleted that has a space quota, the quota itself is also deleted. In some cases, it may be desirable for the space quota to not be automatically deleted. In these cases, the user may configure the system to not delete any space quota automatically via hbase-site.xml.

```
<property>
  <name>hbase.quota.remove.on.table.delete</name>
  <value>false</value>
</property>
```

The value is set to `true` by default.

HBase Snapshots with Space Quotas

One common area of unintended-filesystem-use with HBase is via HBase snapshots. Because snapshots exist outside of the management of HBase tables, it is not uncommon for administrators to suddenly realize that hundreds of gigabytes or terabytes of space is being used by HBase snapshots which were forgotten and never removed.

HBASE-17748 is the umbrella JIRA issue which expands on the original space quota functionality to also include HBase snapshots. While this is a confusing subject, the implementation attempts to present this support in as reasonable and simple of a manner as possible for administrators. This feature does not make any changes to administrator

interaction with space quotas, only in the internal computation of table/namespace usage. Table and namespace usage will automatically incorporate the size taken by a snapshot per the rules defined below.

As a review, let's cover a snapshot's lifecycle: a snapshot is metadata which points to a list of HFiles on the filesystem. This is why creating a snapshot is a very cheap operation; no HBase table data is actually copied to perform a snapshot. Cloning a snapshot into a new table or restoring a table is a cheap operation for the same reason; the new table references the files which already exist on the filesystem without a copy. To include snapshots in space quotas, we need to define which table "owns" a file when a snapshot references the file ("owns" refers to encompassing the filesystem usage of that file).

Consider a snapshot which was made against a table. When the snapshot refers to a file and the table no longer refers to that file, the "originating" table "owns" that file. When multiple snapshots refer to the same file and no table refers to that file, the snapshot with the lowest-sorting name (lexicographically) is chosen and the table which that snapshot was created from "owns" that file. HFiles are not "double-counted" hen a table and one or more snapshots refer to that HFile.

When a table is "rematerialized" (via `clone_snapshot` or `restore_snapshot`), a similar problem of file ownership arises. In this case, while the rematerialized table references a file which a snapshot also references, the table does not "own" the file. The table from which the snapshot was created still "owns" that file. When the rematerialized table is compacted or the snapshot is deleted, the rematerialized table will uniquely refer to a new file and "own" the usage of that file. Similarly, when a table is duplicated via a snapshot and `restore_snapshot`, the new table will not consume any quota size until the original table stops referring to the files, either due to a compaction on the original table, a compaction on the new table, or the original table being deleted.

One new HBase shell command was added to inspect the computed sizes of each snapshot in an HBase instance.

```
hbase> list_snapshot_sizes
SNAPSHOT                                            SIZE
 t1.s1                                              1159108
```

Backup & Snapshots

HBase Backup

There are two broad strategies for performing HBase backups: backing up with a full cluster shutdown, and backing up on a live cluster. Each approach has pros and cons.

For additional information, see HBase Backup Options over on the Sematext Blog.

Full Shutdown Backup

Some environments can tolerate a periodic full shutdown of their HBase cluster, for example if it is being used a back-end analytic capacity and not serving front-end web-pages. The benefits are that the NameNode/Master are RegionServers are down, so there is no chance of missing any in-flight changes to either StoreFiles or metadata. The obvious con is that the cluster is down. The steps include:

Stop HBase

Distcp

Distcp could be used to either copy the contents of the HBase directory in HDFS to either the same cluster in another directory, or to a different cluster.
Note: Distcp works in this situation because the cluster is down and there are no in-flight edits to files. Distcp-ing of files in the HBase directory is not generally recommended on a live cluster.

Restore (if needed)

The backup of the hbase directory from HDFS is copied onto the 'real' hbase directory via distcp. The act of copying these files creates new HDFS metadata, which is why a restore of the NameNode edits from the time of the HBase backup isn't required for this kind of restore, because it's a restore (via distcp) of a specific HDFS directory (i.e., the HBase part) not the entire HDFS file-system.

Live Cluster Backup - Replication

This approach assumes that there is a second cluster. See the HBase page on replication for more information.

Live Cluster Backup - CopyTable

The copytable utility could either be used to copy data from one table to another on the same cluster, or to copy data to another table on another cluster.

Since the cluster is up, there is a risk that edits could be missed in the copy process.

Live Cluster Backup - Export

The export approach dumps the content of a table to HDFS on the same cluster. To restore the data, the import utility would be used.

Since the cluster is up, there is a risk that edits could be missed in the export process. If you want to know more about HBase back-up and restore see the page on Backup and Restore.

HBase Snapshots

HBase Snapshots allow you to take a copy of a table (both contents and metadata)with a very small performance impact. A Snapshot is an immutable collection of table metadata and a list of HFiles that comprised the table at the time the Snapshot was taken. A "clone" of a snapshot creates a new table from that snapshot, and a "restore" of a snapshot returns the contents of a table to what it was when the snapshot was created. The "clone" and "restore" operations do not require any data to be copied, as the underlying HFiles (the files which contain the data for an HBase table) are not modified with either action. Simiarly, exporting a snapshot to another cluster has little impact on RegionServers of the local cluster.

Prior to version 0.94.6, the only way to backup or to clone a table is to use CopyTable/ExportTable, or to copy all the hfiles in HDFS after disabling the table. The disadvantages of these methods are that you can degrade region server performance (Copy/Export Table) or you need to disable the table, that means no reads or writes; and this is usually unacceptable.

Configuration

To turn on the snapshot support just set the `hbase.snapshot.enabled` property to true. (Snapshots are enabled by default in 0.95+ and off by default in 0.94.6+)

```
<property>
  <name>hbase.snapshot.enabled</name>
  <value>true</value>
</property>
```

Take a Snapshot

You can take a snapshot of a table regardless of whether it is enabled or disabled. The snapshot operation doesn't involve any data copying.

```
$ ./bin/hbase shell
hbase> snapshot 'myTable', 'myTableSnapshot-122112'
```

Take a Snapshot Without Flushing

The default behavior is to perform a flush of data in memory before the snapshot is taken. This means that data in memory is included in the snapshot. In most cases, this is the desired behavior. However, if your set-up can tolerate data in memory being excluded from the snapshot, you can use the `SKIP_FLUSH` option of the `snapshot` command to disable and flushing while taking the snapshot.

```
hbase> snapshot 'mytable', 'snapshot123', {SKIP_FLUSH => true}
```

> ⚠ There is no way to determine or predict whether a very concurrent insert or update will be included in a given snapshot, whether flushing is enabled or disabled. A snapshot is only a representation of a table during a window of time. The amount of time the snapshot operation will take to reach each Region Server may vary from a few seconds to a minute, depending on the resource load and speed of the hardware or network, among other factors. There is also no way to know whether a given insert or update is in memory or has been flushed.

Take a Snapshot With TTL

Snapshots have a lifecycle that is independent from the table from which they are created. Although data in a table may be stored with TTL the data files containing them become frozen by the snapshot. Space consumed by expired cells will not be reclaimed by normal table housekeeping like compaction. While this is expected it can be inconvenient at scale. When many snapshots are under management and the data in various tables is expired by TTL some notion of optional TTL (and optional default TTL) for snapshots could be useful.

```
hbase> snapshot 'mytable', 'snapshot1234', {TTL => 86400}
```

The above command creates snapshot `snapshot1234` with TTL of 86400 sec (24 hours) and hence, the snapshot is supposed to be cleaned up after 24 hours

Default Snapshot TTL:

- User specified default TTL with config `hbase.master.snapshot.ttl`
- FOREVER if `hbase.master.snapshot.ttl` is not set

While creating a snapshot, if TTL in seconds is not explicitly specified, the above logic will be followed to determine the TTL. If no configs are changed, the default behavior is that all snapshots will be retained forever (until manual deletion). If a different default TTL behavior is desired, `hbase.master.snapshot.ttl` can be set to a default TTL in seconds. Any snapshot created without an explicit TTL will take this new value.

> If `hbase.master.snapshot.ttl` is set, a snapshot with an explicit {TTL ⇒ 0} or {TTL ⇒ -1} will also take this value. In this case, a TTL < -1 (such as {TTL ⇒ -2}) should be used to indicate FOREVER.

To summarize concisely,

1. Snapshot with TTL value < -1 will stay forever regardless of any server side config changes (until deleted manually by user).
2. Snapshot with TTL value > 0 will be deleted automatically soon after TTL expires.
3. Snapshot created without specifying TTL will always have TTL value represented by config `hbase.master.snapshot.ttl`. Default value of this config is 0, which represents: keep the snapshot forever (until deleted manually by user).
4. From client side, TTL value 0 or -1 should never be explicitly provided because they will be treated same as snapshot without TTL (same as above point 3) and hence will use TTL as per value represented by config `hbase.master.snapshot.ttl`.

Take a snapshot with custom MAX_FILESIZE

Optionally, snapshots can be created with a custom max file size configuration that will be used by cloned tables, instead of the global `hbase.hregion.max.filesize` configuration property. This is mostly useful when exporting snapshots between different clusters. If the HBase cluster where the snapshot is originally taken has a much larger value set for `hbase.hregion.max.filesize` than one or more clusters where the snapshot is being exported to, a storm of region splits may occur when restoring the snapshot on destination clusters.

Specifying `MAX_FILESIZE` on properties passed to `snapshot` command will save informed value into the table's `MAX_FILESIZE` decriptor at snapshot creation time. If the table already defines `MAX_FILESIZE` descriptor, this property would be ignored and have no effect.

```
snapshot 'table01', 'snap01', {MAX_FILESIZE => 21474836480}
```

Enable/Disable Snapshot Auto Cleanup on running cluster:

By default, snapshot auto cleanup based on TTL would be enabled for any new cluster. At any point in time, if snapshot cleanup is supposed to be stopped due to some snapshot restore activity or any other reason, it is advisable to disable it using shell command:

```
hbase> snapshot_cleanup_switch false
```

We can re-enable it using:

```
hbase> snapshot_cleanup_switch true
```

The shell command with switch false would disable snapshot auto cleanup activity based on TTL and return the previous state of the activity(true: running already, false: disabled already)

A sample output for above commands:

```
Previous snapshot cleanup state : true
Took 0.0069 seconds
=> "true"
```

We can query whether snapshot auto cleanup is enabled for cluster using:

```
hbase> snapshot_cleanup_enabled
```

The command would return output in true/false.

Listing Snapshots

List all snapshots taken (by printing the names and relative information).

```
$ ./bin/hbase shell
hbase> list_snapshots
```

Deleting Snapshots

You can remove a snapshot, and the files retained for that snapshot will be removed if no longer needed.

```
$ ./bin/hbase shell
hbase> delete_snapshot 'myTableSnapshot-122112'
```

Clone a table from snapshot

From a snapshot you can create a new table (clone operation) with the same data that you had when the snapshot was taken. The clone operation, doesn't involve data copies, and a change to the cloned table doesn't impact the snapshot or the original table.

```
$ ./bin/hbase shell
hbase> clone_snapshot 'myTableSnapshot-122112', 'myNewTestTable'
```

Restore a snapshot

The restore operation requires the table to be disabled, and the table will be restored to the state at the time when the snapshot was taken, changing both data and schema if required.

```
$ ./bin/hbase shell
hbase> disable 'myTable'
hbase> restore_snapshot 'myTableSnapshot-122112'
```

> Since Replication works at log level and snapshots at file-system level, after a restore, the replicas will be in a different state from the master. If you want to use restore, you need to stop replication and redo the bootstrap.

In case of partial data-loss due to misbehaving client, instead of a full restore that requires the table to be disabled, you can clone the table from the snapshot and use a Map-Reduce job to copy the data that you need, from the clone to the main one.

Snapshots operations and ACLs

If you are using security with the AccessController Coprocessor (See hbase.accesscontrol.configuration), only a global administrator can take, clone, or restore a snapshot, and these actions do not capture the ACL rights. This means that restoring a table preserves the ACL rights of the existing table, while cloning a table creates a new table that has no ACL rights until the administrator adds them.

Export to another cluster

The ExportSnapshot tool copies all the data related to a snapshot (hfiles, logs, snapshot metadata) to another cluster. The tool executes a Map-Reduce job, similar to distcp, to copy files between the two clusters, and since it works at file-system level the hbase cluster does not have to be online.

To copy a snapshot called MySnapshot to an HBase cluster srv2 (hdfs:///srv2:8082/hbase) using 16 mappers:

```
$ bin/hbase org.apache.hadoop.hbase.snapshot.ExportSnapshot -snapshot MySnapshot
-copy-to hdfs://srv2:8082/hbase -mappers 16
```

Limiting Bandwidth Consumption

You can limit the bandwidth consumption when exporting a snapshot, by specifying the `-bandwidth` parameter, which expects an integer representing megabytes per second. The following example limits the above example to 200 MB/sec.

```
$ bin/hbase org.apache.hadoop.hbase.snapshot.ExportSnapshot -snapshot MySnapshot
-copy-to hdfs://srv2:8082/hbase -mappers 16 -bandwidth 200
```

Storing Snapshots in an Amazon S3 Bucket

You can store and retrieve snapshots from Amazon S3, using the following procedure.

> You can also store snapshots in Microsoft Azure Blob Storage. See Storing Snapshots in Microsoft Azure Blob Storage.

Prerequisites

- You must be using HBase 1.0 or higher and Hadoop 2.6.1 or higher, which is the first configuration that uses the Amazon AWS SDK.
- You must use the `s3a://` protocol to connect to Amazon S3. The older `s3n://` and `s3://` protocols have various limitations and do not use the Amazon AWS SDK.
- The `s3a://` URI must be configured and available on the server where you run the commands to export and restore the snapshot.

After you have fulfilled the prerequisites, take the snapshot like you normally would. Afterward, you can export it using the `org.apache.hadoop.hbase.snapshot.ExportSnapshot` command like the one below, substituting your own `s3a://` path in the `copy-from` or `copy-to` directive and substituting or modifying other options as required:

```
$ hbase org.apache.hadoop.hbase.snapshot.ExportSnapshot \
    -snapshot MySnapshot \
    -copy-from hdfs://srv2:8082/hbase \
    -copy-to s3a://<bucket>/<namespace>/hbase \
    -chuser MyUser \
    -chgroup MyGroup \
    -chmod 700 \
    -mappers 16
```

```
$ hbase org.apache.hadoop.hbase.snapshot.ExportSnapshot \
    -snapshot MySnapshot
    -copy-from s3a://<bucket>/<namespace>/hbase \
    -copy-to hdfs://srv2:8082/hbase \
    -chuser MyUser \
    -chgroup MyGroup \
    -chmod 700 \
    -mappers 16
```

You can also use the `org.apache.hadoop.hbase.snapshot.SnapshotInfo` utility with the `s3a://` path by including the `-remote-dir` option.

```
$ hbase org.apache.hadoop.hbase.snapshot.SnapshotInfo \
    -remote-dir s3a://<bucket>/<namespace>/hbase \
    -list-snapshots
```

Storing Snapshots in Microsoft Azure Blob Storage

You can store snapshots in Microsoft Azure Blog Storage using the same techniques as in Storing Snapshots in an Amazon S3 Bucket.

Prerequisites

- You must be using HBase 1.2 or higher with Hadoop 2.7.1 or higher. No version of HBase supports Hadoop 2.7.0.
- Your hosts must be configured to be aware of the Azure blob storage filesystem. See https://hadoop.apache.org/docs/r2.7.1/hadoop-azure/index.html.

After you meet the prerequisites, follow the instructions in Storing Snapshots in an Amazon S3 Bucket, replacingthe protocol specifier with `wasb://` or `wasbs://`.

Storing Snapshots in Aliyun Object Storage Service

You can store snapshots in Aliyun Object Storage Service(Aliyun OSS) using the same techniques as in Storing Snapshots in an Amazon S3 Bucket.

Prerequisites

- You must be using HBase 1.2 or higher with Hadoop 2.9.1 or higher.
- Your hosts must be configured to be aware of the Aliyun oss filesystem. See https://hadoop.apache.org/docs/stable/hadoop-aliyun/tools/hadoop-aliyun/index.html.

After you meet the prerequisites, follow the instructions in Storing Snapshots in an Amazon S3 Bucket, replacing the protocol specifier with `oss://`.

Table Rename

In versions 0.90.x of hbase and earlier, we had a simple script that would rename the hdfs table directory and then do an edit of the hbase:meta table replacing all mentions of the old table name with the new. The script was called `./bin/rename_table.rb`. The script was

deprecated and removed mostly because it was unmaintained and the operation performed by the script was brutal.

As of hbase 0.94.x, you can use the snapshot facility renaming a table. Here is how you would do it using the hbase shell:

```
hbase shell> disable 'tableName'
hbase shell> snapshot 'tableName', 'tableSnapshot'
hbase shell> clone_snapshot 'tableSnapshot', 'newTableName'
hbase shell> delete_snapshot 'tableSnapshot'
hbase shell> drop 'tableName'
```

or in code it would be as follows:

```
void rename(Admin admin, String oldTableName, TableName newTableName) {
  String snapshotName = randomName();
  admin.disableTable(oldTableName);
  admin.snapshot(snapshotName, oldTableName);
  admin.cloneSnapshot(snapshotName, newTableName);
  admin.deleteSnapshot(snapshotName);
  admin.deleteTable(oldTableName);
}
```

Region & Capacity Management

Region Management

Major Compaction

Major compactions can be requested via the HBase shell or Admin.majorCompact.

Note: major compactions do NOT do region merges. See compaction for more information about compactions.

Merge

Merge is a utility that can merge adjoining regions in the same table (see org.apache.hadoop.hbase.util.Merge).

```
$ bin/hbase org.apache.hadoop.hbase.util.Merge <tablename> <region1> <region2>
```

If you feel you have too many regions and want to consolidate them, Merge is the utility you need. Merge must run be done when the cluster is down. See the O'Reilly HBase Book for an example of usage.

You will need to pass 3 parameters to this application. The first one is the table name. The second one is the fully qualified name of the first region to merge, like "table_name,\x0A,1342956111995.7cef47f192318ba7ccc75b1bbf27a82b.". The third one is the fully qualified name for the second region to merge.

Additionally, there is a Ruby script attached to HBASE-1621 for region merging.

Capacity Planning and Region Sizing

There are several considerations when planning the capacity for an HBase cluster and performing the initial configuration. Start with a solid understanding of how HBase handles data internally.

Node count and hardware/VM configuration

Physical data size

Physical data size on disk is distinct from logical size of your data and is affected by the following:

- Increased by HBase overhead
- See keyvalue and keysize. At least 24 bytes per key-value (cell), can be more. Small keys/values means more relative overhead.
- KeyValue instances are aggregated into blocks, which are indexed. Indexes also have to be stored. Blocksize is configurable on a per-ColumnFamily basis. See regions.arch.
- Decreased by compression and data block encoding, depending on data. You might want to test what compression and encoding (if any) make sense for your data.
- Increased by size of region server [wal]/docs/architecture/regionserver#write-ahead-log-wal (usually fixed and negligible - less than half of RS memory size, per RS).
- Increased by HDFS replication - usually x3.

Aside from the disk space necessary to store the data, one RS may not be able to serve arbitrarily large amounts of data due to some practical limits on region count and size (see

ops.capacity.regions).

Read/Write throughput

Number of nodes can also be driven by required throughput for reads and/or writes. The throughput one can get per node depends a lot on data (esp. key/value sizes) and request patterns, as well as node and system configuration. Planning should be done for peak load if it is likely that the load would be the main driver of the increase of the node count. PerformanceEvaluation and ycsb tools can be used to test single node or a test cluster.

For write, usually 5-15Mb/s per RS can be expected, since every region server has only one active WAL. There's no good estimate for reads, as it depends vastly on data, requests, and cache hit rate. perf.casestudy might be helpful.

JVM GC limitations

RS cannot currently utilize very large heap due to cost of GC. There's also no good way of running multiple RS-es per server (other than running several VMs per machine). Thus, ~20-24Gb or less memory dedicated to one RS is recommended. GC tuning is required for large heap sizes. See gcpause, trouble.log.gc and elsewhere (TODO: where?)

Determining region count and size

Generally less regions makes for a smoother running cluster (you can always manually split the big regions later (if necessary) to spread the data, or request load, over the cluster); 20-200 regions per RS is a reasonable range. The number of regions cannot be configured directly (unless you go for fully disable.splitting); adjust the region size to achieve the target region size given table size.

When configuring regions for multiple tables, note that most region settings can be set on a per-table basis via TableDescriptorBuilder, as well as shell commands. These settings will override the ones in `hbase-site.xml`. That is useful if your tables have different workloads/use cases.

Also note that in the discussion of region sizes here, *HDFS replication factor is not (and should not be) taken into account, whereas other factors* ***ops.capacity.nodes.datasize*** *should be.* So, if your data is compressed and replicated 3 ways by HDFS, "9 Gb region" means 9 Gb of compressed data. HDFS replication factor only affects your disk usage and is invisible to most HBase code.

Viewing the Current Number of Regions

You can view the current number of regions for a given table using the HMaster UI. In the **Tables** section, the number of online regions for each table is listed in the **Online Regions** column. This total only includes the in-memory state and does not include disabled or offline regions.

Number of regions per RS - upper bound

In production scenarios, where you have a lot of data, you are normally concerned with the maximum number of regions you can have per server. too many regions has technical discussion on the subject. Basically, the maximum number of regions is mostly determined by memstore memory usage. Each region has its own memstores; these grow up to a configurable size; usually in 128-256 MB range, see hbase.hregion.memstore.flush.size. One memstore exists per column family (so there's only one per region if there's one CF in the table). The RS dedicates some fraction of total memory to its memstores (see hbase.regionserver.global.memstore.size). If this memory is exceeded (too much memstore usage), it can cause undesirable consequences such as unresponsive server or compaction storms. A good starting point for the number of regions per RS (assuming one table) is:

```
((RS memory) * (total memstore fraction)) / ((memstore size)*(# column families))
```

This formula is pseudo-code. Here are two formulas using the actual tunable parameters, first for HBase 0.98+ and second for HBase 0.94.x.

HBase 0.98.x

```
((RS Xmx) * hbase.regionserver.global.memstore.size) / (hbase.hregion.memstore.fl
ush.size * (# column families))
```

HBase 0.94.x

```
((RS Xmx) * hbase.regionserver.global.memstore.upperLimit) / (hbase.hregion.memst
ore.flush.size * (# column families))+
```

If a given RegionServer has 16 GB of RAM, with default settings, the formula works out to 16384*0.4/128 ~ 51 regions per RS is a starting point. The formula can be extended to

multiple tables; if they all have the same configuration, just use the total number of families.

This number can be adjusted; the formula above assumes all your regions are filled at approximately the same rate. If only a fraction of your regions are going to be actively written to, you can divide the result by that fraction to get a larger region count. Then, even if all regions are written to, all region memstores are not filled evenly, and eventually jitter appears even if they are (due to limited number of concurrent flushes). Thus, one can have as many as 2-3 times more regions than the starting point; however, increased numbers carry increased risk.

For write-heavy workload, memstore fraction can be increased in configuration at the expense of block cache; this will also allow one to have more regions.

Number of regions per RS - lower bound

HBase scales by having regions across many servers. Thus if you have 2 regions for 16GB data, on a 20 node machine your data will be concentrated on just a few machines - nearly the entire cluster will be idle. This really can't be stressed enough, since a common problem is loading 200MB data into HBase and then wondering why your awesome 10 node cluster isn't doing anything.

On the other hand, if you have a very large amount of data, you may also want to go for a larger number of regions to avoid having regions that are too large.

Maximum region size

For large tables in production scenarios, maximum region size is mostly limited by compactions - very large compactions, esp. major, can degrade cluster performance. Currently, the recommended maximum region size is 10-20Gb, and 5-10Gb is optimal. For older 0.90.x codebase, the upper-bound of regionsize is about 4Gb, with a default of 256Mb.

The size at which the region is split into two is generally configured via **hbase.hregion.max.filesize**; for details, see **arch.region.splits**.

If you cannot estimate the size of your tables well, when starting off, it's probably best to stick to the default region size, perhaps going smaller for hot tables (or manually split hot regions to spread the load over the cluster), or go with larger region sizes if your cell sizes tend to be largish (100k and up).

In HBase 0.98, experimental stripe compactions feature was added that would allow for larger regions, especially for log data. See ops.stripe.

Total data size per region server

According to above numbers for region size and number of regions per region server, in an optimistic estimate 10 GB x 100 regions per RS will give up to 1TB served per region server, which is in line with some of the reported multi-PB use cases. However, it is important to think about the data vs cache size ratio at the RS level. With 1TB of data per server and 10 GB block cache, only 1% of the data will be cached, which may barely cover all block indices.

Initial configuration and tuning

First, see important configurations. Note that some configurations, more than others, depend on specific scenarios. Pay special attention to:

- hbase.regionserver.handler.count - request handler thread count, vital for high-throughput workloads.
- config.wals - the blocking number of WAL files depends on your memstore configuration and should be set accordingly to prevent potential blocking when doing high volume of writes.

Then, there are some considerations when setting up your cluster and tables.

Compactions

Depending on read/write volume and latency requirements, optimal compaction settings may be different. See compaction for some details.

When provisioning for large data sizes, however, it's good to keep in mind that compactions can affect write throughput. Thus, for write-intensive workloads, you may opt for less frequent compactions and more store files per regions. Minimum number of files for compactions (`hbase.hstore.compaction.min`) can be set to higher value; hbase.hstore.blockingStoreFiles should also be increased, as more files might accumulate in such case. You may also consider manually managing compactions: managed.compactions

Pre-splitting the table

Based on the target number of the regions per RS (see ops.capacity.regions.count) and number of RSes, one can pre-split the table at creation time. This would both avoid some costly splitting as the table starts to fill up, and ensure that the table starts out already distributed across many servers.

If the table is expected to grow large enough to justify that, at least one region per RS should be created. It is not recommended to split immediately into the full target number of regions (e.g. 50 * number of RSes), but a low intermediate value can be chosen. For multiple tables, it is recommended to be conservative with presplitting (e.g. pre-split 1 region per RS at most), especially if you don't know how much each table will grow. If you split too much, you may end up with too many regions, with some tables having too many small regions.

For pre-splitting howto, see manual region splitting decisions and precreate.regions.

RegionServer Grouping

RegionServer Grouping (A.K.A `rsgroup`) is an advanced feature for partitioning regionservers into distinctive groups for strict isolation. It should only be used by users who are sophisticated enough to understand the full implications and have a sufficient background in managing HBase clusters. It was developed by Yahoo! and they run it at scale on their large grid cluster. See HBase at Yahoo! Scale.

RSGroups can be defined and managed with both admin methods and shell commands. A server can be added to a group with hostname and port pair and tables can be moved to this group so that only regionservers in the same rsgroup can host the regions of the table. The group for a table is stored in its TableDescriptor, the property name is `hbase.rsgroup.name`. You can also set this property on a namespace, so it will cause all the tables under this namespace to be placed into this group. RegionServers and tables can only belong to one rsgroup at a time. By default, all tables and regionservers belong to the `default` rsgroup. System tables can also be put into a rsgroup using the regular APIs. A custom balancer implementation tracks assignments per rsgroup and makes sure to move regions to the relevant regionservers in that rsgroup. The rsgroup information is stored in a regular HBase table, and a zookeeper-based read-only cache is used at cluster bootstrap time.

To enable, add the following to your hbase-site.xml and restart your Master:

```
<property>
  <name>hbase.balancer.rsgroup.enabled</name>
  <value>true</value>
</property>
```

Then use the admin/shell *rsgroup* methods/commands to create and manipulate RegionServer groups: e.g. to add a rsgroup and then add a server to it. To see the list of rsgroup commands available in the hbase shell type:

```
hbase(main):008:0> help 'rsgroup'
Took 0.5610 seconds
```

High level, you create a rsgroup that is other than the `default` group using *add_rsgroup* command. You then add servers and tables to this group with the *move_servers_rsgroup* and *move_tables_rsgroup* commands. If necessary, run a balance for the group if tables are slow to migrate to the groups dedicated server with the *balance_rsgroup* command (Usually this is not needed). To monitor effect of the commands, see the `Tables` tab toward the end of the Master UI home page. If you click on a table, you can see what servers it is deployed across. You should see here a reflection of the grouping done with your shell commands. View the master log if issues.

Here is example using a few of the rsgroup commands. To add a group, do as follows:

```
hbase(main):008:0> add_rsgroup 'my_group'
Took 0.5610 seconds
```

RegionServer Groups must be Enabled

If you have not enabled the rsgroup feature and you call any of the rsgroup admin methods or shell commands the call will fail with a `DoNotRetryIOException` with a detail message that says the rsgroup feature is disabled.

Add a server (specified by hostname + port) to the just-made group using the *move_servers_rsgroup* command as follows:

```
hbase(main):010:0> move_servers_rsgroup 'my_group',['k.att.net:51129']
```

Hostname and Port vs ServerName

> The rsgroup feature refers to servers in a cluster with hostname and port only. It does not make use of the HBase ServerName type identifying RegionServers; i.e. hostname + port + starttime to distinguish RegionServer instances. The rsgroup feature keeps working across RegionServer restarts so the starttime of ServerName — and hence the ServerName type — is not appropriate. Administration

Servers come and go over the lifetime of a Cluster. Currently, you must manually align the servers referenced in rsgroups with the actual state of nodes in the running cluster. What we mean by this is that if you decommission a server, then you must update rsgroups as part of your server decommission process removing references. Notice that, by calling `clearDeadServers` manually will also remove the dead servers from any rsgroups, but the problem is that we will lost track of the dead servers after master restarts, which means you still need to update the rsgroup by your own.

Please use `Admin.removeServersFromRSGroup` or shell command *remove_servers_rsgroup* to remove decommission servers from rsgroup.

The `default` group is not like other rsgroups in that it is dynamic. Its server list mirrors the current state of the cluster; i.e. if you shutdown a server that was part of the `default` rsgroup, and then do a *get_rsgroup* `default` to list its content in the shell, the server will no longer be listed. For non-default groups, though a mode may be offline, it will persist in the non-default group's list of servers. But if you move the offline server from the non-default rsgroup to default, it will not show in the `default` list. It will just be dropped.

Best Practice

The authors of the rsgroup feature, the Yahoo! HBase Engineering team, have been running it on their grid for a good while now and have come up with a few best practices informed by their experience.

Isolate System Tables

Either have a system rsgroup where all the system tables are or just leave the system tables in `default` rsgroup and have all user-space tables are in non-default rsgroups.

Dead Nodes

Yahoo! Have found it useful at their scale to keep a special rsgroup of dead or questionable nodes; this is one means of keeping them out of the running until repair.

Be careful replacing dead nodes in an rsgroup. Ensure there are enough live nodes before you start moving out the dead. Move in good live nodes first if you have to.

Troubleshooting

Viewing the Master log will give you insight on rsgroup operation.

If it appears stuck, restart the Master process.

Remove RegionServer Grouping

Simply disable RegionServer Grouping feature is easy, just remove the 'hbase.balancer.rsgroup.enabled' from hbase-site.xml or explicitly set it to false in hbase-site.xml.

```
<property>
  <name>hbase.balancer.rsgroup.enabled</name>
  <value>false</value>
</property>
```

But if you change the 'hbase.balancer.rsgroup.enabled' to true, the old rsgroup configs will take effect again. So if you want to completely remove the RegionServer Grouping feature from a cluster, so that if the feature is re-enabled in the future, the old meta data will not affect the functioning of the cluster, there are more steps to do.

- Move all tables in non-default rsgroups to `default` regionserver group

```
#Reassigning table t1 from non default group - hbase shell
hbase(main):005:0> move_tables_rsgroup 'default',['t1']
```

- Move all regionservers in non-default rsgroups to `default` regionserver group

```
#Reassigning all the servers in the non-default rsgroup to default - hbase shel
l
hbase(main):008:0> move_servers_rsgroup 'default',['rs1.xxx.com:16206','rs2.xx
x.com:16202','rs3.xxx.com:16204']
```

- Remove all non-default rsgroups. `default` rsgroup created implicitly doesn't have to be removed

```
#removing non default rsgroup - hbase shell
hbase(main):009:0> remove_rsgroup 'group2'
```

- Remove the changes made in `hbase-site.xml` and restart the cluster
- Drop the table `hbase:rsgroup` from `hbase`

```
#Through hbase shell drop table hbase:rsgroup
hbase(main):001:0> disable 'hbase:rsgroup'
0 row(s) in 2.6270 seconds

hbase(main):002:0> drop 'hbase:rsgroup'
0 row(s) in 1.2730 seconds
```

- Remove znode `rsgroup` from the cluster ZooKeeper using zkCli.sh

```
#From ZK remove the node /hbase/rsgroup through zkCli.sh
rmr /hbase/rsgroup
```

ACL

To enable ACL, add the following to your hbase-site.xml and restart your Master:

```
<property>
  <name>hbase.security.authorization</name>
  <value>true</value>
<property>
```

Migrating From Old Implementation

The coprocessor `org.apache.hadoop.hbase.rsgroup.RSGroupAdminEndpoint` is deprected, but for compatible, if you want the pre 3.0.0 hbase client/shell to communicate with the new hbase cluster, you still need to add this coprocessor to master.

The `hbase.rsgroup.grouploadbalancer.class` config has been deprecated, as now the top level load balancer will always be `RSGroupBasedLoadBalaner`, and the `hbase.master.loadbalancer.class` config is for configuring the balancer within a group. This also means you should not set `hbase.master.loadbalancer.class` to `RSGroupBasedLoadBalaner` any more even if rsgroup feature is enabled.

And we have done some special changes for compatibility. First, if coprocessor `org.apache.hadoop.hbase.rsgroup.RSGroupAdminEndpoint` is specified, the `hbase.balancer.rsgroup.enabled` flag will be set to true automatically to enable rs group feature. Second, we will load `hbase.rsgroup.grouploadbalancer.class` prior to `hbase.master.loadbalancer.class`. And last, if you do not set `hbase.rsgroup.grouploadbalancer.class` but only set `hbase.master.loadbalancer.class` to `RSGroupBasedLoadBalancer`, we will load the default load balancer to avoid infinite nesting. This means you do not need to change anything when upgrading if you have already enabled rs group feature.

The main difference comparing to the old implementation is that, now the rsgroup for a table is stored in `TableDescriptor`, instead of in `RSGroupInfo`, so the `getTables` method of `RSGroupInfo` has been deprecated. And if you use the `Admin` methods to get the `RSGroupInfo`, its `getTables` method will always return empty. This is because that in the old implementation, this method is a bit broken as you can set rsgroup on namespace and make all the tables under this namespace into this group but you can not get these tables through `RSGroupInfo.getTables`. Now you should use the two new methods `listTablesInRSGroup` and `getConfiguredNamespacesAndTablesInRSGroup` in `Admin` to get tables and namespaces in a rsgroup.

Of course the behavior for the old RSGroupAdminEndpoint is not changed, we will fill the tables field of the RSGroupInfo before returning, to make it compatible with old hbase client/shell.

When upgrading, the migration between the RSGroupInfo and TableDescriptor will be done automatically. It will take sometime, but it is fine to restart master in the middle, the migration will continue after restart. And during the migration, the rs group feature will still work and in most cases the region will not be misplaced(since this is only a one time job and will not last too long so we have not test it very seriously to make sure the region will not be misplaced always, so we use the word 'in most cases'). The implementation is a bit tricky, you can see the code in `RSGroupInfoManagerImpl.migrate` if interested.

Region Normalizer

The Region Normalizer tries to make Regions all in a table about the same in size. It does this by first calculating total table size and average size per region. It splits any region that is larger than twice this size. Any region that is much smaller is merged into an adjacent

region. The Normalizer runs on a regular schedule, which is configurable. It can also be disabled entirely via a runtime "switch". It can be run manually via the shell or Admin API call. Even if normally disabled, it is good to run manually after the cluster has been running a while or say after a burst of activity such as a large delete.

The Normalizer works well for bringing a table's region boundaries into alignment with the reality of data distribution after an initial effort at pre-splitting a table. It is also a nice compliment to the data TTL feature when the schema includes timestamp in the rowkey, as it will automatically merge away regions whose contents have expired.

(The bulk of the below detail was copied wholesale from the blog by Romil Choksi at HBase Region Normalizer).

The Region Normalizer is feature available since HBase-1.2. It runs a set of pre-calculated merge/split actions to resize regions that are either too large or too small compared to the average region size for a given table. Region Normalizer when invoked computes a normalization 'plan' for all of the tables in HBase. System tables (such as hbase:meta, hbase:namespace, Phoenix system tables etc) and user tables with normalization disabled are ignored while computing the plan. For normalization enabled tables, normalization plan is carried out in parallel across multiple tables.

Normalizer can be enabled or disabled globally for the entire cluster using the ‘normalizer_switch' command in the HBase shell. Normalization can also be controlled on a per table basis, which is disabled by default when a table is created. Normalization for a table can be enabled or disabled by setting the NORMALIZATION_ENABLED table attribute to true or false.

To check normalizer status and enable/disable normalizer

```
hbase(main):001:0> normalizer_enabled
true
0 row(s) in 0.4870 seconds

hbase(main):002:0> normalizer_switch false
true
0 row(s) in 0.0640 seconds

hbase(main):003:0> normalizer_enabled
false
0 row(s) in 0.0120 seconds

hbase(main):004:0> normalizer_switch true
false
```

```
0 row(s) in 0.0200 seconds

hbase(main):005:0> normalizer_enabled
true
0 row(s) in 0.0090 seconds
```

When enabled, Normalizer is invoked in the background every 5 mins (by default), which can be configured using `hbase.normalization.period` in `hbase-site.xml`. Normalizer can also be invoked manually/programmatically at will using HBase shell's `normalize` command. HBase by default uses `SimpleRegionNormalizer`, but users can design their own normalizer as long as they implement the RegionNormalizer Interface. Details about the logic used by `SimpleRegionNormalizer` to compute its normalization plan can be found here.

The below example shows a normalization plan being computed for an user table, and merge action being taken as a result of the normalization plan computed by SimpleRegionNormalizer.

Consider an user table with some pre-split regions having 3 equally large regions (about 100K rows) and 1 relatively small region (about 25K rows). Following is the snippet from an hbase meta table scan showing each of the pre-split regions for the user table.

```
table_p8ddpd6q5z,,1469494305548.68b9892220865cb6048 column=info:regioninfo, times
tamp=1469494306375, value={ENCODED => 68b9892220865cb604809c950d1adf48, NAME =>
'table_p8ddpd6q5z,,1469494305548.68b989222 09c950d1adf48.   0865cb604809c950d1adf
48.', STARTKEY => '', ENDKEY => '1'}
....
table_p8ddpd6q5z,1,1469494317178.867b77333bdc75a028 column=info:regioninfo, times
tamp=1469494317848, value={ENCODED => 867b77333bdc75a028bb4c5e4b235f48, NAME =>
'table_p8ddpd6q5z,1,1469494317178.867b7733 bb4c5e4b235f48.  3bdc75a028bb4c5e4b235
f48.', STARTKEY => '1', ENDKEY => '3'}
....
table_p8ddpd6q5z,3,1469494328323.98f019a753425e7977 column=info:regioninfo, times
tamp=1469494328486, value={ENCODED => 98f019a753425e7977ab8636e32deeeb, NAME =>
'table_p8ddpd6q5z,3,1469494328323.98f019a7 ab8636e32deeeb.  53425e7977ab8636e32de
eeb.', STARTKEY => '3', ENDKEY => '7'}
....
table_p8ddpd6q5z,7,1469494339662.94c64e748979ecbb16 column=info:regioninfo, times
tamp=1469494339859, value={ENCODED => 94c64e748979ecbb166f6cc6550e25c6, NAME =>
'table_p8ddpd6q5z,7,1469494339662.94c64e74 6f6cc6550e25c6.   8979ecbb166f6cc6550e
25c6.', STARTKEY => '7', ENDKEY => '8'}
....
table_p8ddpd6q5z,8,1469494339662.6d2b3f5fd1595ab8e7 column=info:regioninfo, times
tamp=1469494339859, value={ENCODED => 6d2b3f5fd1595ab8e7c031876057b1ee, NAME =>
'table_p8ddpd6q5z,8,1469494339662.6d2b3f5f c031876057b1ee.   d1595ab8e7c031876057
b1ee.', STARTKEY => '8', ENDKEY => ''}
```

Invoking the normalizer using 'normalize' int the HBase shell, the below log snippet from HMaster log shows the normalization plan computed as per the logic defined for SimpleRegionNormalizer. Since the total region size (in MB) for the adjacent smallest regions in the table is less than the average region size, the normalizer computes a plan to merge these two regions.

```
2016-07-26 07:08:26,928 DEBUG [B.fifo.QRpcServer.handler=20,queue=2,port=20000] m
aster.HMaster: Skipping normalization for table: hbase:namespace, as it's either
system table or doesn't have auto
normalization turned on
2016-07-26 07:08:26,928 DEBUG [B.fifo.QRpcServer.handler=20,queue=2,port=20000] m
aster.HMaster: Skipping normalization for table: hbase:backup, as it's either sys
tem table or doesn't have auto normalization turned on
2016-07-26 07:08:26,928 DEBUG [B.fifo.QRpcServer.handler=20,queue=2,port=20000] m
aster.HMaster: Skipping normalization for table: hbase:meta, as it's either syste
m table or doesn't have auto normalization turned on
2016-07-26 07:08:26,928 DEBUG [B.fifo.QRpcServer.handler=20,queue=2,port=20000] m
aster.HMaster: Skipping normalization for table: table_h2osxu3wat, as it's either
system table or doesn't have autonormalization turned on
2016-07-26 07:08:26,928 DEBUG [B.fifo.QRpcServer.handler=20,queue=2,port=20000] n
ormalizer.SimpleRegionNormalizer: Computing normalization plan for table: table_p
8ddpd6q5z, number of regions: 5
2016-07-26 07:08:26,929 DEBUG [B.fifo.QRpcServer.handler=20,queue=2,port=20000] n
ormalizer.SimpleRegionNormalizer: Table table_p8ddpd6q5z, total aggregated region
s size: 12
2016-07-26 07:08:26,929 DEBUG [B.fifo.QRpcServer.handler=20,queue=2,port=20000] n
ormalizer.SimpleRegionNormalizer: Table table_p8ddpd6q5z, average region size: 2.
4
2016-07-26 07:08:26,929 INFO  [B.fifo.QRpcServer.handler=20,queue=2,port=20000] n
ormalizer.SimpleRegionNormalizer: Table table_p8ddpd6q5z, small region size: 0 pl
us its neighbor size: 0, less thanthe avg size 2.4, merging them
2016-07-26 07:08:26,971 INFO  [B.fifo.QRpcServer.handler=20,queue=2,port=20000] n
ormalizer.MergeNormalizationPlan: Executing merging normalization plan: MergeNorm
alizationPlan{firstRegion={ENCODED=> d51df2c58e9b525206b1325fd925a971, NAME => 't
able_p8ddpd6q5z,,1469514755237.d51df2c58e9b525206b1325fd925a971.', STARTKEY =>
'', ENDKEY => '1'}, secondRegion={ENCODED => e69c6b25c7b9562d078d9ad3994f5330, NA
ME => 'table_p8ddpd6q5z,1,1469514767669.e69c6b25c7b9562d078d9ad3994f5330.',
STARTKEY => '1', ENDKEY => '3'}}
```

Region normalizer as per it's computed plan, merged the region with start key as '' and end key as '1', with another region having start key as '1' and end key as '3'. Now, that these regions have been merged we see a single new region with start key as '' and end key as '3'

```
table_p8ddpd6q5z,,1469516907210.e06c9b83c4a252b130e column=info:mergeA, timestamp
=1469516907431,
value=PBUF\x08\xA5\xD9\x9E\xAF\xE2*\x12\x1B\x0A\x07default\x12\x10table_p8ddpd6q5
z\x1A\x00"\x011(\x000\x00 ea74d246741ba.   8\x00
```

```
table_p8ddpd6q5z,,1469516907210.e06c9b83c4a252b130e column=info:mergeB, timestamp
=1469516907431,
value=PBUF\x08\xB5\xBA\x9F\xAF\xE2*\x12\x1B\x0A\x07default\x12\x10table_p8ddpd6q5
z\x1A\x011"\x013(\x000\x0 ea74d246741ba.   08\x00
table_p8ddpd6q5z,,1469516907210.e06c9b83c4a252b130e column=info:regioninfo, times
tamp=1469516907431, value={ENCODED => e06c9b83c4a252b130eea74d246741ba, NAME =>
'table_p8ddpd6q5z,,1469516907210.e06c9b83c ea74d246741ba.   4a252b130eea74d246741
ba.', STARTKEY => '', ENDKEY => '3'}
....
table_p8ddpd6q5z,3,1469514778736.bf024670a847c0adff column=info:regioninfo, times
tamp=1469514779417, value={ENCODED => bf024670a847c0adffb74b2e13408b32, NAME =>
'table_p8ddpd6q5z,3,1469514778736.bf024670 b74b2e13408b32.  a847c0adffb74b2e13408
b32.' STARTKEY => '3', ENDKEY => '7'}
....
table_p8ddpd6q5z,7,1469514790152.7c5a67bc755e649db2 column=info:regioninfo, times
tamp=1469514790312, value={ENCODED => 7c5a67bc755e649db22f49af6270f1e1, NAME =>
'table_p8ddpd6q5z,7,1469514790152.7c5a67bc 2f49af6270f1e1.  755e649db22f49af6270f
1e1.', STARTKEY => '7', ENDKEY => '8'}
....
table_p8ddpd6q5z,8,1469514790152.58e7503cda69f98f47 column=info:regioninfo, times
tamp=1469514790312, value={ENCODED => 58e7503cda69f98f4755178e74288c3a, NAME =>
'table_p8ddpd6q5z,8,1469514790152.58e7503c 55178e74288c3a.  da69f98f4755178e74288
c3a.', STARTKEY => '8', ENDKEY => ''}
```

A similar example can be seen for an user table with 3 smaller regions and 1 relatively large region. For this example, we have an user table with 1 large region containing 100K rows, and 3 relatively smaller regions with about 33K rows each. As seen from the normalization plan, since the larger region is more than twice the average region size it ends being split into two regions – one with start key as ‘1' and end key as ‘154717' and the other region with start key as '154717' and end key as ‘3'

```
2016-07-26 07:39:45,636 DEBUG [B.fifo.QRpcServer.handler=7,queue=1,port=20000] ma
ster.HMaster: Skipping normalization for table: hbase:backup, as it's either syst
em table or doesn't have auto normalization turned on
2016-07-26 07:39:45,636 DEBUG [B.fifo.QRpcServer.handler=7,queue=1,port=20000] no
rmalizer.SimpleRegionNormalizer: Computing normalization plan for table: table_p8
ddpd6q5z, number of regions: 4
2016-07-26 07:39:45,636 DEBUG [B.fifo.QRpcServer.handler=7,queue=1,port=20000] no
rmalizer.SimpleRegionNormalizer: Table table_p8ddpd6q5z, total aggregated regions
size: 12
2016-07-26 07:39:45,636 DEBUG [B.fifo.QRpcServer.handler=7,queue=1,port=20000] no
rmalizer.SimpleRegionNormalizer: Table table_p8ddpd6q5z, average region size: 3.0
2016-07-26 07:39:45,636 DEBUG [B.fifo.QRpcServer.handler=7,queue=1,port=20000] no
rmalizer.SimpleRegionNormalizer: No normalization needed, regions look good for t
able: table_p8ddpd6q5z
2016-07-26 07:39:45,636 DEBUG [B.fifo.QRpcServer.handler=7,queue=1,port=20000] no
rmalizer.SimpleRegionNormalizer: Computing normalization plan for table: table_h2
osxu3wat, number of regions: 5
2016-07-26 07:39:45,636 DEBUG [B.fifo.QRpcServer.handler=7,queue=1,port=20000] no
rmalizer.SimpleRegionNormalizer: Table table_h2osxu3wat, total aggregated regions
size: 7
```

```
2016-07-26 07:39:45,636 DEBUG [B.fifo.QRpcServer.handler=7,queue=1,port=20000] no
rmalizer.SimpleRegionNormalizer: Table table_h2osxu3wat, average region size: 1.4
2016-07-26 07:39:45,636 INFO  [B.fifo.QRpcServer.handler=7,queue=1,port=20000] no
rmalizer.SimpleRegionNormalizer: Table table_h2osxu3wat, large region table_h2osx
u3wat,1,1469515926544.27f2fdbb2b6612ea163eb6b40753c3db. has size 4, more than twi
ce avg size, splitting
2016-07-26 07:39:45,640 INFO [B.fifo.QRpcServer.handler=7,queue=1,port=20000] nor
malizer.SplitNormalizationPlan: Executing splitting normalization plan: SplitNorm
alizationPlan{regionInfo={ENCODED => 27f2fdbb2b6612ea163eb6b40753c3db, NAME => 't
able_h2osxu3wat,1,1469515926544.27f2fdbb2b6612ea163eb6b40753c3db.', STARTKEY =>
'1', ENDKEY => '3'}, splitPoint=null}
2016-07-26 07:39:45,656 DEBUG [B.fifo.QRpcServer.handler=7,queue=1,port=20000] ma
ster.HMaster: Skipping normalization for table: hbase:namespace, as it's either s
ystem table or doesn't have auto normalization turned on
2016-07-26 07:39:45,656 DEBUG [B.fifo.QRpcServer.handler=7,queue=1,port=20000] ma
ster.HMaster: Skipping normalization for table: hbase:meta, as it's either system
table or doesn't
have auto normalization turned on ..............
2016-07-26 07:39:46,246 DEBUG [AM.ZK.Worker-pool2-t278] master.RegionStates: Onli
ned 54de97dae764b864504704c1c8d3674a on hbase-test-rc-5.openstacklocal,16020,1469
419333913 {ENCODED => 54de97dae764b864504704c1c8d3674a, NAME => 'table_h2osxu3wa
t,1,1469518785661.54de97dae764b864504704c1c8d3674a.', STARTKEY => '1', ENDKEY =>
'154717'}
2016-07-26 07:39:46,246 INFO  [AM.ZK.Worker-pool2-t278] master.RegionStates: Tran
sition {d6b5625df331cfec84dce4f1122c567f state=SPLITTING_NEW, ts=1469518786246, s
erver=hbase-test-rc-5.openstacklocal,16020,1469419333913} to {d6b5625df331cfec84d
ce4f1122c567f state=OPEN, ts=1469518786246,
server=hbase-test-rc-5.openstacklocal,16020,1469419333913}
2016-07-26 07:39:46,246 DEBUG [AM.ZK.Worker-pool2-t278] master.RegionStates: Onli
ned d6b5625df331cfec84dce4f1122c567f on hbase-test-rc-5.openstacklocal,16020,1469
419333913 {ENCODED => d6b5625df331cfec84dce4f1122c567f, NAME => 'table_h2osxu3wa
t,154717,1469518785661.d6b5625df331cfec84dce4f1122c567f.', STARTKEY => '154717',
ENDKEY => '3'}
```

Auto Region Reopen

We can leak store reader references if a coprocessor or core function somehow opens a scanner, or wraps one, and then does not take care to call close on the scanner or the wrapped instance. Leaked store files can not be removed even after it is invalidated via compaction. A reasonable mitigation for a reader reference leak would be a fast reopen of the region on the same server. This will release all resources, like the refcount, leases, etc. The clients should gracefully ride over this like any other region in transition. By default this auto reopen of region feature would be disabled. To enabled it, please provide high ref count value for config `hbase.regions.recovery.store.file.ref.count`.

Please refer to config descriptions for `hbase.master.regions.recovery.check.interval` and `hbase.regions.recovery.store.file.ref.count`.

Building and Developing

This chapter contains information and guidelines for building and releasing HBase code and documentation. Being familiar with these guidelines will help the HBase committers to use your contributions more easily.

Getting Involved

Apache HBase gets better only when people contribute! If you are looking to contribute to Apache HBase, look for issues in JIRA tagged with the label 'beginner'. These are issues HBase contributors have deemed worthy but not of immediate priority and a good way to ramp on HBase internals. See What label is used for issues that are good on ramps for new contributors? from the dev mailing list for background.

Before you get started submitting code to HBase, please refer to Developer Guidelines.

As Apache HBase is an Apache Software Foundation project, see The Apache Software Foundation for more information about how the ASF functions.

Mailing Lists

Sign up for the dev-list and the user-list. See the mailing lists page. Posing questions - and helping to answer other people's questions - is encouraged! There are varying levels of experience on both lists so patience and politeness are encouraged (and please stay on topic.)

Slack

The Apache HBase project uses the #hbase channel on the official ASF Slack Workspace for real-time questions and discussion. All committers of any Apache projects can join the channel directly, for others, please mail dev@hbase.apache.org to request an invite.

Internet Relay Chat (IRC)

(NOTE: Our IRC channel seems to have been deprecated in favor of the above Slack channel)

For real-time questions and discussions, use the `#hbase` IRC channel on the FreeNode IRC network. FreeNode offers a web-based client, but most people prefer a native client, and several clients are available for each operating system.

Jira

Check for existing issues in Jira. If it's either a new feature request, enhancement, or a bug, file a ticket.

We track multiple types of work in JIRA:

- Bug: Something is broken in HBase itself.
- Test: A test is needed, or a test is broken.
- New feature: You have an idea for new functionality. It's often best to bring these up on the mailing lists first, and then write up a design specification that you add to the feature request JIRA.
- Improvement: A feature exists, but could be tweaked or augmented. It's often best to bring these up on the mailing lists first and have a discussion, then summarize or link to the discussion if others seem interested in the improvement.
- Wish: This is like a new feature, but for something you may not have the background to flesh out yourself.

Bugs and tests have the highest priority and should be actionable.

Guidelines for reporting effective issues

- *Search for duplicates*: Your issue may have already been reported. Have a look, realizing that someone else might have worded the summary differently.

  Also search the mailing lists, which may have information about your problem and how to work around it. Don't file an issue for something that has already been discussed and

resolved on a mailing list, unless you strongly disagree with the resolution *and* are willing to help take the issue forward.

- *Discuss in public*: Use the mailing lists to discuss what you've discovered and see if there is something you've missed. Avoid using back channels, so that you benefit from the experience and expertise of the project as a whole.
- *Don't file on behalf of others*: You might not have all the context, and you don't have as much motivation to see it through as the person who is actually experiencing the bug. It's more helpful in the long term to encourage others to file their own issues. Point them to this material and offer to help out the first time or two.
- *Write a good summary*: A good summary includes information about the problem, the impact on the user or developer, and the area of the code.
  - Good: `Address new license dependencies from hadoop3-alpha4`
  - Room for improvement: `Canary is broken` If you write a bad title, someone else will rewrite it for you. This is time they could have spent working on the issue instead.
- *Give context in the description*: It can be good to think of this in multiple parts:
  - What happens or doesn't happen?
  - How does it impact you?
  - How can someone else reproduce it?
  - What would "fixed" look like?
    You don't need to know the answers for all of these, but give as much information as you can. If you can provide technical information, such as a Git commit SHA that you think might have caused the issue or a build failure on builds.apache.org where you think the issue first showed up, share that info.
- **Fill in all relevant fields**: These fields help us filter, categorize, and find things.
- **One bug, one issue, one patch**: To help with back-porting, don't split issues or fixes among multiple bugs.
- **Add value if you can**: Filing issues is great, even if you don't know how to fix them. But providing as much information as possible, being willing to triage and answer questions, and being willing to test potential fixes is even better! We want to fix your issue as quickly as you want it to be fixed.
- **Don't be upset if we don't fix it**: Time and resources are finite. In some cases, we may not be able to (or might choose not to) fix an issue, especially if it is an edge case or

there is a workaround. Even if it doesn't get fixed, the JIRA is a public record of it, and will help others out if they run into a similar issue in the future.

Working on an issue

To check for existing issues which you can tackle as a beginner, search for issues in JIRA tagged with the label 'beginner'.

JIRA Priorites:

- **Blocker**: Should only be used if the issue WILL cause data loss or cluster instability reliably.
- **Critical**: The issue described can cause data loss or cluster instability in some cases.
- **Major**: Important but not tragic issues, like updates to the client API that will add a lot of much-needed functionality or significant bugs that need to be fixed but that don't cause data loss.
- **Minor**: Useful enhancements and annoying but not damaging bugs.
- **Trivial**: Useful enhancements but generally cosmetic.

Code Blocks in Jira Comments:

A commonly used macro in Jira is `{code}`. Everything inside the tags is preformatted, as in this example.

```
{code}
code snippet
{code}
```

Apache HBase Repositories

Apache HBase consists of multiple repositories which are hosted on Apache GitBox. These are the following:

- hbase - main Apache HBase repository
- hbase-connectors - connectors to Apache Kafka and Apache Spark
- hbase-operator-tools - operability and supportability tools, such as HBase HBCK2

- hbase-site - hbase.apache.org website
- hbase-thirdparty - relocated versions of popular third-party libraries

IDEs

Eclipse

Code Formatting

Under the *dev-support/* folder, you will find *hbase_eclipse_formatter.xml*. We encourage you to have this formatter in place in eclipse when editing HBase code.

Go to `Preferences->Java->Code Style->Formatter->Import` to load the xml file. Go to `Preferences->Java->Editor->Save Actions`, and make sure 'Format source code' and 'Format edited lines' is selected.

In addition to the automatic formatting, make sure you follow the style guidelines explained in Code Formatting Conventions.

Eclipse Git Plugin

If you cloned the project via git, download and install the Git plugin (EGit). Attach to your local git repo (via the Git Repositories window) and you'll be able to see file revision history, generate patches, etc.

HBase Project Setup in Eclipse using `m2eclipse`

The easiest way is to use the `m2eclipse` plugin for Eclipse. Eclipse Indigo or newer includes +m2eclipse+, or you can download it from http://www.eclipse.org/m2e/. It provides Maven integration for Eclipse, and even lets you use the direct Maven commands from within Eclipse to compile and test your project.

To import the project, click and select the HBase root directory. `m2eclipse` locates all the hbase modules for you.

If you install `m2eclipse` and import HBase in your workspace, do the following to fix your eclipse Build Path.

- Remove *target* folder
- Add *target/generated-sources/java* folder.
- Remove from your Build Path the exclusions on the *src/main/resources* and *src/test/resources* to avoid error message in the console, such as the following:

```
Failed to execute goal
org.apache.maven.plugins:maven-antrun-plugin:1.6:run (default) on project hbas
e:
'An Ant BuildException has occurred: Replace: source file .../target/classes/hb
ase-default.xml
doesn't exist
```

  This will also reduce the eclipse build cycles and make your life easier when developing.

HBase Project Setup in Eclipse Using the Command Line

Instead of using `m2eclipse`, you can generate the Eclipse files from the command line.

- First, run the following command, which builds HBase. You only need to do this once.

```
mvn clean install -DskipTests
```

- Close Eclipse, and execute the following command from the terminal, in your local HBase project directory, to generate new *.project* and *.classpath* files.

```
mvn eclipse:eclipse
```

- Reopen Eclipse and import the *.project* file in the HBase directory to a workspace.

Maven Classpath Variable

The `$M2_REPO` classpath variable needs to be set up for the project. This needs to be set to your local Maven repository, which is usually *~/.m2/repository*

If this classpath variable is not configured, you will see compile errors in Eclipse like this:

```
Description        Resource        Path        Location        Type
The project cannot be built until build path errors are resolved        hbase
Unknown        Java Problem
```

```
Unbound classpath variable: 'M2_REPO/asm/asm/3.1/asm-3.1.jar' in project 'hbase'
hbase                Build path        Build Path Problem
Unbound classpath variable: 'M2_REPO/com/google/guava/guava/r09/guava-r09.jar' in
project 'hbase'        hbase                Build path        Build Path Problem
Unbound classpath variable: 'M2_REPO/com/google/protobuf/protobuf-java/2.3.0/prot
obuf-java-2.3.0.jar' in project 'hbase'        hbase                Build path
Build Path Problem Unbound classpath variable:
```

Eclipse Known Issues

Eclipse will currently complain about *Bytes.java*. It is not possible to turn these errors off.

```
Description        Resource        Path        Location        Type
Access restriction: The method arrayBaseOffset(Class) from the type Unsafe is not
accessible due to restriction on required library /System/Library/Java/JavaVirtua
lMachines/1.6.0.jdk/Contents/Classes/classes.jar        Bytes.java        /hbase/
src/main/java/org/apache/hadoop/hbase/util        line 1061        Java Problem
Access restriction: The method arrayIndexScale(Class) from the type Unsafe is not
accessible due to restriction on required library /System/Library/Java/JavaVirtua
lMachines/1.6.0.jdk/Contents/Classes/classes.jar        Bytes.java        /hbase/
src/main/java/org/apache/hadoop/hbase/util        line 1064        Java Problem
Access restriction: The method getLong(Object, long) from the type Unsafe is not
accessible due to restriction on required library /System/Library/Java/JavaVirtua
lMachines/1.6.0.jdk/Contents/Classes/classes.jar        Bytes.java        /hbase/
src/main/java/org/apache/hadoop/hbase/util        line 1111        Java Problem
```

Eclipse - More Information

For additional information on setting up Eclipse for HBase development on Windows, see Michael Morello's blog on the topic.

IntelliJ IDEA

A functional development environment can be setup around an IntelliJ IDEA installation that has the plugins necessary for building Java projects with Maven.

- Use either File > New > "Project from Existing Sources..." or "Project From Version Control.."
- Depending on your version of IntelliJ, you may need to choose Maven as the "project" or "model" type.

The following plugins are recommended:

- Maven, bundled. This allows IntelliJ to resolve dependencies and recognize the project structure.
- EditorConfig, bundled. This will apply project whitespace settings found in the `.editorconfig` file available on branches with HBASE-23234 or later.
- Checkstyle-IDEA. Configure this against the configuration file found under `hbase-checkstyle/src/main/resources/hbase/checkstyle.xml` (If the Intellij checkstyle plugin complains parsing the volunteered hbase `checkstyle.xml`, make sure the plugin's `version` popup menu matches the hbase checkstyle version. Find the current checkstyle version as a property in `pom.xml`. This plugin will highlight style errors in the IDE, so you can fix them before they get flagged during the pre-commit process.
- Protobuf Support. HBase uses Protocol Buffers in a number of places where serialization is required. This plugin is helpful when editing these object definitions.
- MDX. HBase uses MDX (just extended markdown) for building it's project documentation. This plugin is helpful when editing this book.

Other IDEs

If you'd have another environment with which you'd like to develop on HBase, please consider documenting your setup process here.

Building Apache HBase

Basic Compile

HBase is compiled using Maven. You must use at least Maven 3.0.4. To check your Maven version, run the command `mvn -version`.

JDK Version Requirements

HBase has Java version compiler requirements that vary by release branch. At compilation time, HBase has the same version requirements as it does for runtime. See Java for a complete support matrix of Java version by HBase version.

Maven Build Commands

All commands are executed from the local HBase project directory.

Package

The simplest command to compile HBase from its java source code is to use the `package` target, which builds JARs with the compiled files.

```
mvn package -DskipTests
```

Or, to clean up before compiling:

```
mvn clean package -DskipTests
```

With Eclipse set up as explained above in Eclipse, you can also use the **Build** command in Eclipse. To create the full installable HBase package takes a little bit more work, so read on.

Compile

The `compile` target does not create the JARs with the compiled files.

```
mvn compile
```

```
mvn clean compile
```

Install

To install the JARs in your *~/.m2/* directory, use the `install` target.

```
mvn install
```

```
mvn clean install
```

```
mvn clean install -DskipTests
```

Building HBase 2.x on Apple Silicon

Building a non-master branch requires protoc 2.5.0 binary which is not available for Apple Silicon. HBASE-27741 added a workaround to the build to fall back to osx-x86_64 version of protoc automatically by `apple-silicon-workaround` Maven profile. The intention is that this change will permit the build to proceed with the x86 version of `protoc`, making use of the Rosetta instruction translation service built into the OS. If you'd like to provide and make use of your own aarch_64 `protoc`, you can disable this profile on the command line by adding `-P'!apple-silicon-workaround'`, or through configuration in your `settings.xml`.

You can use the following commands to build protoc on your Apple Silicon machine.

```
curl -sSL https://github.com/protocolbuffers/protobuf/releases/download/v2.5.0/pr
otobuf-2.5.0.tar.gz | tar zx -
cd protobuf-2.5.0
curl -L -O https://gist.githubusercontent.com/liusheng/64aee1b27de037f8b9ccf1873b
82c413/raw/118c2fce733a9a62a03281753572a45b6efb8639/protobuf-2.5.0-arm64.patch
patch -p1 < protobuf-2.5.0-arm64.patch
./configure --disable-shared
make
mvn install:install-file -DgroupId=com.google.protobuf -DartifactId=protoc -Dvers
ion=2.5.0 -Dclassifier=osx-aarch_64 -Dpackaging=exe -Dfile=src/protoc
```

Running all or individual Unit Tests

See the Running tests section in Unit Tests

Building against various Hadoop versions

HBase supports building against Apache Hadoop versions: 2.y and 3.y (early release artifacts). Exactly which version of Hadoop is used by default varies by release branch. See the section Hadoop for the complete breakdown of supported Hadoop version by HBase release.

The mechanism for selecting a Hadoop version at build time is identical across all releases. Which version of Hadoop is default varies. We manage Hadoop major version selection by way of Maven profiles. Due to the peculiarities of Maven profile mutual exclusion, the profile that builds against a particular Hadoop version is activated by setting a property, *not* the usual profile activation. Hadoop version profile activation is summarized by the following table.

Hadoop Profile Activation by HBase Release

	Hadoop2 Activation	Hadoop3 Activation
HBase 1.3+	*active by default*	`-Dhadoop.profile=3.0`
HBase 3.0+	*not supported*	*active by default*

> ⚠ Please note that where a profile is active by default, `hadoop.profile` must NOT be provided.

Once the Hadoop major version profile is activated, the exact Hadoop version can be specified by overriding the appropriate property value. For Hadoop2 versions, the property name is `hadoop-two.version`. With Hadoop3 versions, the property name is `hadoop-three.version`.

Example 1: Building HBase 1.7 against Hadoop 2.10.0

For example, to build HBase 1.7 against Hadoop 2.10.0, the profile is set for Hadoop2 by default, so only `hadoop-two.version` must be specified:

```
git checkout branch-1
mvn -Dhadoop-two.version=2.10.0 ...
```

Example 2: Building HBase 2.3 or 2.4 against Hadoop 3.4.0-SNAPSHOT

This is how a developer might check the compatibility of HBase 2.3 or 2.4 against an unreleased Hadoop version (currently 3.4). Both the Hadoop3 profile and version must be specified:

```
git checkout branch-2.4
mvn -Dhadoop.profile=3.0 -Dhadoop-three.version=3.4.0-SNAPSHOT ...
```

Example 3: Building HBase 3.0 against Hadoop 3.4.0-SNAPSHOT

The same developer might want also to check the development version of HBase (currently 3.0) against the development version of Hadoop (currently 3.4). In this case, the Hadoop3 profile is active by default, so only `hadoop-three.version` must be specified:

```
git checkout master
mvn -Dhadoop-three.version=3.4.0-SNAPSHOT ...
```

Building with JDK11 and Hadoop3

HBase manages JDK-specific build settings using Maven profiles. The profile appropriate to the JDK in use is automatically activated. Building and running on JDK8 supports both Hadoop2 and Hadoop3. For JDK11, only Hadoop3 is supported. Thus, the Hadoop3 profile must be active when building on JDK11, and the artifacts used when running HBase on JDK11 must be compiled against Hadoop3. Furthermore, the JDK11 profile requires a minimum Hadoop version of 3.2.0. This value is specified by the JDK11 profile, but it can be overridden using the `hadoop-three.version` property as normal. For details on Hadoop profile activation by HBase branch, see Building against various Hadoop versions. See Java for a complete support matrix of Java version by HBase version.

Example 1: Building HBase 2.3 or 2.4 with JDK11

To build HBase 2.3 or 2.4 with JDK11, the Hadoop3 profile must be activated explicitly.

```
git checkout branch-2.4
JAVA_HOME=/usr/lib/jvm/java-11 mvn -Dhadoop.profile=3.0 ...
```

Example 2: Building HBase 3.0 with JDK11

For HBase 3.0, the Hadoop3 profile is active by default, no additional properties need be specified.

```
git checkout master
JAVA_HOME=/usr/lib/jvm/java-11 mvn ...
```

Building and testing in an IDE with JDK11 and Hadoop3

Continuing the discussion from the earlier section, building and testing with JDK11 and Hadoop3 within an IDE may require additional configuration. Specifically, make sure the JVM version used by the IDE is a JDK11, the active JDK Maven profile is for JDK11, and the Maven profile for JDK8 is NOT active. Likewise, ensure the Hadoop3 Maven profile is active and the Hadoop2 Maven profile is NOT active.

Build Protobuf

You may need to change the protobuf definitions that reside in the *hbase-protocol* module or other modules.

Previous to hbase-2.0.0, protobuf definition files were sprinkled across all hbase modules but now all to do with protobuf must reside in the hbase-protocol module; we are trying to contain our protobuf use so we can freely change versions without upsetting any downstream project use of protobuf.

The protobuf files are located in *hbase-protocol/src/main/protobuf*. For the change to be effective, you will need to regenerate the classes.

```
mvn package -pl hbase-protocol -am
```

Similarly, protobuf definitions for internal use are located in the *hbase-protocol-shaded* module.

```
mvn package -pl hbase-protocol-shaded -am
```

Typically, protobuf code generation is done using the native `protoc` binary. In our build we use a maven plugin for convenience; however, the plugin may not be able to retrieve appropriate binaries for all platforms. If you find yourself on a platform where protoc fails, you will have to compile protoc from source, and run it independent of our maven build. You can disable the inline code generation by specifying `-Dprotoc.skip` in your maven arguments, allowing your build to proceed further.

> If you need to manually generate your protobuf files, you should not use `clean` in subsequent maven calls, as that will delete the newly generated files.

Read the *hbase-protocol/README.txt* for more details

Build Thrift

You may need to change the thrift definitions that reside in the *hbase-thrift* module or other modules.

The thrift files are located in *hbase-thrift/src/main/resources*. For the change to be effective, you will need to regenerate the classes. You can use maven profile `compile-thrift` to do this.

```
mvn compile -Pcompile-thrift
```

You may also want to define `thrift.path` for the thrift binary, using the following command:

```
mvn compile -Pcompile-thrift -Dthrift.path=/opt/local/bin/thrift
```

Build a Tarball

You can build a tarball without going through the release process described in Releasing Apache HBase, by running the following command:

```
mvn -DskipTests clean install && mvn -DskipTests package assembly:single
```

The distribution tarball is built in `hbase-assembly/target/hbase-<version>-bin.tar.gz`.

You can install or deploy the tarball by having the assembly:single goal before install or deploy in the maven command:

```
mvn -DskipTests package assembly:single install
```

```
mvn -DskipTests package assembly:single deploy
```

Build Gotchas

Maven Site failure

If you see `Unable to find resource 'VM_global_library.vm'`, ignore it. It's not an error. It is officially ugly though.

Build On Linux Aarch64

HBase runs on both Windows and UNIX-like systems, and it should run on any platform that runs a supported version of Java. This should include JVMs on x86_64 and aarch64. The documentation below describes how to build hbase on aarch64 platform.

Set Environment Variables

Manually install Java and Maven on aarch64 servers if they are not installed, and set environment variables. For example:

```
export JAVA_HOME=/usr/lib/jvm/java-8-openjdk-arm64
export MAVEN_HOME=/opt/maven
export PATH=${MAVEN_HOME}/bin:${JAVA_HOME}/bin:${PATH}
```

Use Protobuf Supported On Aarch64

Now HBase uses protobuf of two versions. Version '3.11.4' of protobuf that hbase uses internally and version '2.5.0' as external usage. Package protoc-2.5.0 does not work on aarch64 platform, we should add maven profile '-Paarch64' when building. It downloads protoc-2.5.0 package from maven repository which we made on aarch64 platform locally.

```
mvn clean install -Paarch64 -DskipTests
```

> Protobuf is released with aarch64 protoc since version '3.5.0', and we are planning to upgrade protobuf later, then we don't have to add the profile '-Paarch64' anymore.

Releasing Apache HBase

> **Building against HBase 1.x**
>
> See old refguides for how to build HBase 1.x. The below is for building hbase2.

Making a Release Candidate

Only committers can make releases of hbase artifacts.

Before You Begin

Check to be sure recent builds have been passing for the branch from where you are going to take your release. You should also have tried recent branch tips out on a cluster under load, perhaps by running the `hbase-it` integration test suite for a few hours to 'burn in' the near-candidate bits.

You will need a published signing key added to the hbase KEYS file. (For how to add a KEY, see *Step 1.* in How To Release, the Hadoop version of this document).

Next make sure JIRA is properly primed, that all issues targeted against the prospective release have been resolved and are present in git on the particular branch. If any outstanding issues, move them out of the release by adjusting the fix version to remove this pending release as a target. Any JIRA with a fix version that matches the release candidate target release will be included in the generated *CHANGES.md/RELEASENOTES.md* files that ship with the release so make sure JIRA is correct before you begin.

After doing the above, you can move to the manufacture of an RC.

Building an RC is involved so we've scripted it. The script builds in a Docker container to ensure we have a consistent environment building. It will ask you for passwords for apache and for your gpg signing key so it can sign and commit on your behalf. The passwords are passed to gpg-agent in the container and purged along with the container when the build is done.

The script will:

- Set version to the release version
- Updates RELEASENOTES.md and CHANGES.md
- Tag the RC
- Set version to next SNAPSHOT version.
- Builds, signs, and hashes all artifacts.
- Generates the api compatibility report
- Pushes release tgzs to the dev dir in a apache dist.
- Pushes to repository.apache.org staging.
- Creates vote email template.

The *dev-support/create-release/do-release-docker.sh* Release Candidate (RC) Generating script is maintained in the master branch but can generate RCs for any 2.x+ branch (The script does not work against branch-1). Check out and update the master branch when making RCs. See *dev-support/create-release/README.txt* for how to configure your environment and run the script.

dev-support/create-release/do-release-docker.sh supercedes the previous *dev-support/make_rc.sh* script. It is more comprehensive automating all steps, rather than a portion, building a RC.

Release Candidate Procedure

Here we outline the steps involved generating a Release Candidate, the steps automated by the *dev-support/create-release/do-release-docker.sh* script described in the previous section. Running these steps manually tends to be error-prone so is not recommended. The below is informational only.

The process below makes use of various tools, mainly *git* and *maven*.

Specifying the Heap Space for Maven

You may run into OutOfMemoryErrors building, particularly building the site and documentation. Up the heap for Maven by setting the `MAVEN_OPTS` variable. You can prefix the variable to the Maven command, as in the following example:

```
MAVEN_OPTS="-Xmx4g -XX:MaxPermSize=256m" mvn package
```

You could also set this in an environment variable or alias in your shell.

1 **Example *~/.m2/settings.xml* File**

Publishing to maven requires you sign the artifacts you want to upload. For the build to sign them for you, you a properly configured *settings.xml* in your local repository under *.m2*, such as the following.

```
<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0
                      http://maven.apache.org/xsd/settings-1.0.0.xsd">
  <servers>
    <!- To publish a snapshot of some part of Maven -->
    <server>
      <id>apache.snapshots.https</id>
      <username>YOUR_APACHE_ID
      </username>
      <password>YOUR_APACHE_PASSWORD
      </password>
    </server>
    <!-- To publish a website using Maven -->
    <!-- To stage a release of some part of Maven -->
    <server>
```

```
      <id>apache.releases.https</id>
      <username>YOUR_APACHE_ID
      </username>
      <password>YOUR_APACHE_PASSWORD
      </password>
    </server>
  </servers>
  <profiles>
    <profile>
      <id>apache-release</id>
      <properties>
    <gpg.keyname>YOUR_KEYNAME</gpg.keyname>
    <!--Keyname is something like this ... 00A5F21E... do `gpg ——list-keys`
to find it-->
    <gpg.passphrase>YOUR_KEY_PASSWORD
    </gpg.passphrase>
      </properties>
    </profile>
  </profiles>
</settings>
```

2 **Update the *CHANGES.md* and *RELEASENOTES.md* files and the POM files.**

Update *CHANGES.md* with the changes since the last release. Be careful with where you put headings and license. Respect the instructions and warning you find in current *CHANGES.md* and *RELEASENOTES.md* since these two files are processed by tooling that is looking for particular string sequences. See HBASE-21399 for description on how to make use of yetus generating additions to *CHANGES.md* and *RELEASENOTES.md* (RECOMMENDED!). Adding JIRA fixes, make sure the URL to the JIRA points to the proper location which lists fixes for this release.

Next, adjust the version in all the POM files appropriately. If you are making a release candidate, you must remove the `-SNAPSHOT` label from all versions in all pom.xml files. If you are running this receipe to publish a snapshot, you must keep the `-SNAPSHOT` suffix on the hbase version. The Versions Maven Plugin can be of use here. To set a version in all the many poms of the hbase multi-module project, use a command like the following:

```
$ mvn clean org.codehaus.mojo:versions-maven-plugin:2.5:set -DnewVersion=2.
1.0-SNAPSHOT
```

Make sure all versions in poms are changed! Checkin the *CHANGES.md*, *RELEASENOTES.md*, and any maven version changes.

3 **Update the documentation.**

Update the documentation under *hbase-website/app/page/_docs/docs/_mdx/(multi-page)*. This usually involves copying the latest from master branch and making version-particular adjustments to suit this release candidate version. Commit your changes.

4 **Clean the checkout dir**

```
$ mvn clean
$ git clean -f -x -d
```

5 **Run Apache-Rat**

Check licenses are good

```
$ mvn apache-rat:check
```

If the above fails, check the rat log.

```
$ grep 'Rat check' patchprocess/mvn_apache_rat.log
```

6 **Create a release tag.**

Presuming you have run basic tests, the rat check, passes and all is looking good, now is the time to tag the release candidate (You always remove the tag if you need to redo). To tag, do what follows substituting in the version appropriate to your build. All tags should be signed tags; i.e. pass the *-s* option (See Signing Your Work for how to set up your git environment for signing).

```
$ git tag -s 2.0.0-alpha4-RC0 -m "Tagging the 2.0.0-alpha4 first Releae Cand
idate (Candidates start at zero)"
```

Or, if you are making a release, tags should have a *rel/* prefix to ensure they are preserved in the Apache repo as in:

```
+$ git tag -s rel/2.0.0-alpha4 -m "Tagging the 2.0.0-alpha4 Release"
```

Push the (specific) tag (only) so others have access.

```
$ git push origin 2.0.0-alpha4-RC0
```

For how to delete tags, see How to Delete a Tag. Covers deleting tags that have not yet been pushed to the remote Apache repo as well as delete of tags pushed to Apache.

7 **Build the source tarball.**

Now, build the source tarball. Lets presume we are building the source tarball for the tag *2.0.0-alpha4-RC0* into */tmp/hbase-2.0.0-alpha4-RC0/* (This step requires that the mvn and git clean steps described above have just been done).

```
$ git archive --format=tar.gz --output="/tmp/hbase-2.0.0-alpha4-RC0/hbase-2.
0.0-alpha4-src.tar.gz" --prefix="hbase-2.0.0-alpha4/" $git_tag
```

Above we generate the hbase-2.0.0-alpha4-src.tar.gz tarball into the */tmp/hbase-2.0.0-alpha4-RC0* build output directory (We don't want the *RC0* in the name or prefix. These bits are currently a release candidate but if the VOTE passes, they will become the release so we do not taint the artifact names with *RCX*).

8 **Build the binary tarball.**

Next, build the binary tarball. Add the `-Prelease` profile when building. It runs the license apache-rat check among other rules that help ensure all is wholesome. Do it in two steps.

First install into the local repository

```
$ mvn clean install -DskipTests -Prelease
```

Next, generate documentation and assemble the tarball. Be warned, this next step can take a good while, a couple of hours generating site documentation.

```
$ mvn install -DskipTests site assembly:single -Prelease
```

Otherwise, the build complains that hbase modules are not in the maven repository when you try to do it all in one step, especially on a fresh repository. It seems that you need the install goal in both steps.

Extract the generated tarball — you'll find it under *hbase-assembly/target* and check it out. Look at the documentation, see if it runs, etc. If good, copy the tarball beside the source tarball in the build output directory.

9 **Deploy to the Maven Repository.**

Next, deploy HBase to the Apache Maven repository. Add the apache-release `profile when running the` mvn deploy` command. This profile comes from the Apache parent pom referenced by our pom files. It does signing of your artifacts published to Maven, as long as the *settings.xml* is configured correctly, as described in Example ~/.m2/settings.xml File. This step depends on the local repository having been populate by the just-previous bin tarball build.

```
$ mvn deploy -DskipTests -Papache-release -Prelease
```

This command copies all artifacts up to a temporary staging Apache mvn repository in an 'open' state. More work needs to be done on these maven artifacts to make them generally available.

We do not release HBase tarball to the Apache Maven repository. To avoid deploying the tarball, do not include the `assembly:single` goal in your `mvn deploy` command. Check the deployed artifacts as described in the next section.

> **make_rc.sh**
>
> If you ran the old *dev-support/make_rc.sh* script, this is as far as it takes you. To finish the release, take up the script from here on out.

10 **Make the Release Candidate available.**

The artifacts are in the maven repository in the staging area in the 'open' state. While in this 'open' state you can check out what you've published to make sure all is good. To do this, log in to Apache's Nexus at repository.apache.org using your Apache ID. Find your artifacts in the staging repository. Click on 'Staging Repositories' and look for a new one ending in "hbase" with a status of 'Open', select it. Use the tree view to expand the list of repository contents and inspect if the artifacts you expect are present. Check the POMs. As long as the staging repo is open you can re-upload if something is missing or built incorrectly.

If something is seriously wrong and you would like to back out the upload, you can use the 'Drop' button to drop and delete the staging repository. Sometimes the upload fails in the middle. This is another reason you might have to 'Drop' the upload from the staging repository.

If it checks out, close the repo using the 'Close' button. The repository must be closed before a public URL to it becomes available. It may take a few minutes for the repository to close. Once complete you'll see a public URL to the repository in the Nexus UI. You may also receive an email with the URL. Provide the URL to the temporary staging repository in the email that announces the release candidate. (Folks will need to add this repo URL to their local poms or to their local *settings.xml* file to pull the published release candidate artifacts.)

When the release vote concludes successfully, return here and click the 'Release' button to release the artifacts to central. The release process will automatically drop and delete the staging repository.

> **hbase-downstreamer**
>
> See the **hbase-downstreamer** test for a simple example of a project that is downstream of HBase an depends on it. Check it out and run its simple test to make sure maven artifacts are properly deployed to the maven repository. Be sure to edit the pom to point to the proper staging repository. Make sure you are pulling from the repository when tests run and that you are not getting from your local repository, by either passing the `-U` flag or deleting your local repo content and check maven is pulling from remote out of the staging repository.

See **Publishing Maven Artifacts** for some pointers on this maven staging process.

If the HBase version ends in `-SNAPSHOT`, the artifacts go elsewhere. They are put into the Apache snapshots repository directly and are immediately available. Making a SNAPSHOT release, this is what you want to happen.

At this stage, you have two tarballs in your 'build output directory' and a set of artifacts in a staging area of the maven repository, in the 'closed' state. Next sign, fingerprint and then 'stage' your release candiate build output directory via svnpubsub by committing your directory to **The dev distribution directory** (See comments on **HBASE-10554 Please delete old releases from mirroring system** but in essence it is an svn checkout of **dev/hbase** — releases are at **release/hbase**). In the *version directory* run the following commands:

```
$ for i in *.tar.gz; do echo $i; gpg --print-md MD5 $i > $i.md5 ; done
$ for i in *.tar.gz; do echo $i; gpg --print-md SHA512 $i > $i.sha ; done
$ for i in *.tar.gz; do echo $i; gpg --armor --output $i.asc --detach-sig $i
; done
$ cd ..
# Presuming our 'build output directory' is named 0.96.0RC0, copy it to the
svn checkout of the dist dev dir
# in this case named hbase.dist.dev.svn
$ cd /Users/stack/checkouts/hbase.dist.dev.svn
$ svn info
Path: .
Working Copy Root Path: /Users/stack/checkouts/hbase.dist.dev.svn
URL: https://dist.apache.org/repos/dist/dev/hbase
Repository Root: https://dist.apache.org/repos/dist
Repository UUID: 0d268c88-bc11-4956-87df-91683dc98e59
Revision: 15087
Node Kind: directory
Schedule: normal
Last Changed Author: ndimiduk
Last Changed Rev: 15045
Last Changed Date: 2016-08-28 11:13:36 -0700 (Sun, 28 Aug 2016)
$ mv 0.96.0RC0 /Users/stack/checkouts/hbase.dist.dev.svn
$ svn add 0.96.0RC0
$ svn commit ...
```

Ensure it actually gets published by checking https://dist.apache.org/repos/dist/dev/hbase/.

Announce the release candidate on the mailing list and call a vote.

Publishing a SNAPSHOT to maven

Make sure your *settings.xml* is set up properly (see Example ~/.m2/settings.xml File). Make sure the hbase version includes `-SNAPSHOT` as a suffix. Following is an example of publishing SNAPSHOTS of a release that had an hbase version of 0.96.0 in its poms.

```
$ mvn clean install -DskipTests  javadoc:aggregate site assembly:single -Prelease
$ mvn -DskipTests  deploy -Papache-release
```

The *make_rc.sh* script mentioned above (see Making a Release Candidate) can help you publish `SNAPSHOTS`. Make sure your `hbase.version` has a `-SNAPSHOT` suffix before running the script. It will put a snapshot up into the apache snapshot repository for you.

Voting on Release Candidates

Everyone is encouraged to try and vote on HBase release candidates. Only the votes of PMC members are binding. PMC members, please read this WIP doc on policy voting for a release candidate, Release Policy.

> *"Before casting +1 binding votes, individuals are required to download the signed source code package onto their own hardware, compile it as provided, and test the resulting executable on their own platform, along with also validating cryptographic signatures and verifying that the package meets the requirements of the ASF policy on releases."*

Regards the latter, run `mvn apache-rat:check` to verify all files are suitably licensed. See HBase, mail # dev - On recent discussion clarifying ASF release policy for how we arrived at this process.

To help with the release verification, please follow the guideline below and vote based on the your verification.

Baseline Verifications for Voting Release Candidates

Although contributors have their own checklist for verifications, the following items are usually used for voting on release candidates.

- CHANGES.md if any
- RELEASENOTES.md (release notes) if any
- Generated API compatibility report
  - For what should be compatible please refer the versioning guideline, especially for items with marked as high severity
- Use `hbase-vote.sh` to perform sanity checks for checksum, signatures, files are licensed, built from source, and unit tests.
  - `hbase-vote.sh` shell script is available under `dev-support` directory of HBase source. Following are the usage details.

```
./dev-support/hbase-vote.sh -h
hbase-vote. A script for standard vote which verifies the following items
1. Checksum of sources and binaries
2. Signature of sources and binaries
3. Rat check
4. Built from source
5. Unit tests

Usage: hbase-vote.sh -s | --source <url> [-k | --key <signature>] [-f | --keys-
file-url <url>] [-o | --output-dir </path/to/use>] [-P runSmallTests] [-D prope
rty[=value]]
      hbase-vote.sh -h | --help

  -h | --help                   Show this screen.
  -s | --source '<url>'         A URL pointing to the release candidate sources
and binaries
                                e.g. https://dist.apache.org/repos/dist/dev/hba
se/hbase-<version>RC0/
  -k | --key '<signature>'      A signature of the public key, e.g. 9AD2AE49
  -f | --keys-file-url '<url>'   the URL of the key file, default is
                                https://downloads.apache.org/hbase/KEYS
  -o | --output-dir '</path>'   directory which has the stdout and stderr of ea
ch verification target
  -P |                          list of maven profiles to activate for test UT/
IT, i.e. <-P runSmallTests> Defaults to runAllTests
  -D |                          list of maven properties to set for the mvn inv
ocations, i.e. <-D hadoop.profile=3.0> Defaults to unset
```

- If you see any unit test failures, please call out the solo test result and whether it's part of flaky (nightly) tests dashboard, e.g. dashboard of master branch (please change the test branch accordingly).

Additional Verifications for Voting Release Candidates

Other than the common verifications, contributors may call out additional concerns, e.g. for a specific feature by running end to end tests on a distributed environment. This is optional and always encouraged.

- Start a distributed HBase cluster and call out the test result of specific workload on cluster. e.g.
  - Run basic table operations, e.g. `create/put/get/scan/flush/list/disable/drop`
  - Run built-in tests, e.g. `LoadTestTool` (LTT) and `IntegrationTestBigLinkedList` (ITBLL)

Announcing Releases

Once an RC has passed successfully and the needed artifacts have been staged for disribution, you'll need to let everyone know about our shiny new release. It's not a requirement, but to make things easier for release managers we have a template you can start with. Be sure you replace _version_ and other markers with the relevant version numbers. You should manually verify all links before sending.

```
The HBase team is happy to announce the immediate availability of HBase _version
_.

Apache HBase™ is an open-source, distributed, versioned, non-relational database.
Apache HBase gives you low latency random access to billions of rows with
millions of columns atop non-specialized hardware. To learn more about HBase,
see https://hbase.apache.org/.

HBase _version_ is the _nth_ minor release in the HBase _major_.x line, which aim
s to
improve the stability and reliability of HBase. This release includes roughly
XXX resolved issues not covered by previous _major_.x releases.

Notable new features include:
- List text descriptions of features that fit on one line
- Including if JDK or Hadoop support versions changes
- If the "stable" pointer changes, call that out
- For those with obvious JIRA IDs, include them (HBASE-YYYYY)

The full list of issues can be found in the included CHANGES.md and RELEASENOTES.
md,
or via our issue tracker:

    https://s.apache.org/hbase-_version_-jira

To download please follow the links and instructions on our website:

    https://hbase.apache.org/downloads.html


Question, comments, and problems are always welcome at: dev@hbase.apache.org.

Thanks to all who contributed and made this release possible.

Cheers,
The HBase Dev Team
```

You should sent this message to the following lists: dev@hbase.apache.org, user@hbase.apache.org, announce@apache.org. If you'd like a spot check before sending, feel free to ask via jira or the dev list.

Generating the HBase Reference Guide

The manual is marked up using MDX (just extended markdown). Then we render markdown into HTML content by using Fumadocs. To build run `mvn site` from the root or `hbase-website` directory. See appendix contributing to documentation for more information on building the documentation.

Updating hbase.apache.org

Contributing to hbase.apache.org

See appendix contributing to documentation for more information on contributing to the documentation or website.

Publishing hbase.apache.org

See Publishing the HBase Website and Documentation for instructions on publishing the website and documentation.

Tests

Developers, at a minimum, should familiarize themselves with the unit test detail; unit tests in HBase have a character not usually seen in other projects.

This information is about unit tests for HBase itself. For developing unit tests for your HBase applications, see Unit Testing HBase Applications.

Apache HBase Modules

As of 0.96, Apache HBase is split into multiple modules. This creates "interesting" rules for how and where tests are written. If you are writing code for `hbase-server`, see Unit Tests for how to write your tests. These tests can spin up a minicluster and will need to be categorized. For any other module, for example `hbase-common`, the tests must be strict unit

tests and just test the class under test - no use of the HBaseTestingUtility or minicluster is allowed (or even possible given the dependency tree).

Starting from 3.0.0, HBaseTestingUtility is renamed to HBaseTestingUtil and marked as IA.Private. Of course the API is still the same.

Testing the HBase Shell

The HBase shell and its tests are predominantly written in jruby.

In order to make these tests run as a part of the standard build, there are a few JUnit test classes that take care of loading the jruby implemented tests and running them. The tests were split into separate classes to accomodate class level timeouts (see Unit Tests for specifics). You can run all of these tests from the top level with:

```
mvn clean test -Dtest=Test*Shell
```

If you have previously done a `mvn install`, then you can instruct maven to run only the tests in the hbase-shell module with:

```
mvn clean test -pl hbase-shell
```

Alternatively, you may limit the shell tests that run using the system variable `shell.test`. This value should specify the ruby literal equivalent of a particular test case by name. For example, the tests that cover the shell commands for altering tables are contained in the test case `AdminAlterTableTest` and you can run them with:

```
mvn clean test -pl hbase-shell -Dshell.test=/AdminAlterTableTest/
```

You may also use a Ruby Regular Expression literal (in the `/pattern/` style) to select a set of test cases. You can run all of the HBase admin related tests, including both the normal administration and the security administration, with the command:

```
mvn clean test -pl hbase-shell -Dshell.test=/.*Admin.*Test/
```

In the event of a test failure, you can see details by examining the XML version of the surefire report results

```
vim hbase-shell/target/surefire-reports/TEST-org.apache.hadoop.hbase.client.TestShell.xml
```

Running Tests in other Modules

If the module you are developing in has no other dependencies on other HBase modules, then you can cd into that module and just run:

```
mvn test
```

which will just run the tests IN THAT MODULE. If there are other dependencies on other modules, then you will have run the command from the ROOT HBASE DIRECTORY. This will run the tests in the other modules, unless you specify to skip the tests in that module. For instance, to skip the tests in the hbase-server module, you would run:

```
mvn clean test -PskipServerTests
```

from the top level directory to run all the tests in modules other than hbase-server. Note that you can specify to skip tests in multiple modules as well as just for a single module. For example, to skip the tests in `hbase-server` and `hbase-common`, you would run:

```
mvn clean test -PskipServerTests -PskipCommonTests
```

Also, keep in mind that if you are running tests in the `hbase-server` module you will need to apply the maven profiles discussed in Running tests to get the tests to run properly.

Unit Tests

Apache HBase unit tests must carry a Category annotation and as of `hbase-2.0.0`, must be stamped with the HBase `ClassRule`. Here is an example of what a Test Class looks like with a Category and ClassRule included:

```
...
@Category(SmallTests.class)
public class TestHRegionInfo {
  @ClassRule
  public static final HBaseClassTestRule CLASS_RULE =
```

```
      HBaseClassTestRule.forClass(TestHRegionInfo.class);

  @Test
  public void testCreateHRegionInfoName() throws Exception {
    // ...
  }
}
```

Here the Test Class is `TestHRegionInfo`. The `CLASS_RULE` has the same form in every test class only the `.class` you pass is that of the local test; i.e. in the TestTimeout Test Class, you'd pass `TestTimeout.class` to the `CLASS_RULE` instead of the `TestHRegionInfo.class` we have above. The `CLASS_RULE` is where we'll enforce timeouts (currently set at a hard-limit of thirteen! minutes for all tests — 780 seconds) and other cross-unit test facility. The test is in the `SmallTest` Category.

Categories can be arbitrary and provided as a list but each test MUST carry one from the following list of sizings: `small`, `medium`, `large`, and `integration`. The test sizing is designated using the JUnit categories: `SmallTests`, `MediumTests`, `LargeTests`, `IntegrationTests`. JUnit Categories are denoted using java annotations (a special unit test looks for the presence of the @Category annotation in all unit tess and will fail if it finds a test suite missing a sizing marking).

The first three categories, `small`, `medium`, and `large`, are for test cases which run when you type `$ mvn test`. In other words, these three categorizations are for HBase unit tests. The `integration` category is not for unit tests, but for integration tests. These are normally run when you invoke `$ mvn verify`. Integration tests are described in Integration Tests.

Keep reading to figure which annotation of the set `small`, `medium`, and `large` to put on your new HBase test case.

Categorizing Tests

Small Tests:

Small test cases are executed in separate JVM and each test suite/test class should run in 15 seconds or less; i.e. a junit test fixture, a java object made up of test methods, should finish in under 15 seconds, no matter how many or how few test methods it has. These test cases should not use a minicluster as a minicluster starts many services, most unrelated to what is being tested.

Medium Tests:

Medium test cases are executed in separate JVM and individual test suites or test classes or in junit parlance, test fixture, should run in 50 seconds or less. These test cases can use a mini cluster. Since we start up a JVM per test fixture (and often a cluster too), be sure to make the startup pay by writing test fixtures that do a lot of testing running tens of seconds perhaps combining test rather than spin up a jvm (and cluster) per test method; this practice will help w/ overall test times.

Large Tests:

Large test cases are everything else. They are typically large-scale tests, regression tests for specific bugs, timeout tests, or performance tests. No large test suite can take longer than thirteen minutes. It will be killed as timed out. Cast your test as an Integration Test if it needs to run longer.

Integration Tests:

Integration tests are system level tests. See Integration Tests for more info. If you invoke `$ mvn test` on integration tests, there is no timeout for the test.

Running tests

The state of tests on the hbase branches varies. Some branches keep good test hygiene and all tests pass reliably with perhaps an unlucky sporadic flakey test failure. On other branches, the case may be less so with frequent flakies and even broken tests in need of attention that fail 100% of the time. Try and figure the state of tests on the branch you are currently interested in; the current state of nightly apache jenkins builds is a good place to start. Tests on master branch are generally not in the best of condition as releases are less frequent off master. This can make it hard landing patches especially given our dictum that patches land on master branch first.

The full test suite can take from 5-6 hours on an anemic VM with 4 CPUs and minimal parallelism to 50 minutes or less on a linux machine with dozens of CPUs and plenty of RAM.

When you go to run the full test suite, make sure you up the test runner user nproc (`ulimit -u` — make sure it > 6000 or more if more parallelism) and the number of open files (`ulimit -n` — make sure it > 10240 or more) limits on your system. Errors because the test run

hits limits are often only opaquely related to the constraint. You can see the current user settings by running `ulimit -a`.

Default: small and medium category tests

Running `mvn test` will execute all small tests in a single JVM (no fork) and then medium tests in a forked, separate JVM for each test instance (For definition of 'small' test and so on, see Unit Tests). Medium tests are NOT executed if there is an error in a small test. Large tests are NOT executed.

Running all tests

Running `mvn test -P runAllTests` will execute small tests in a single JVM, then medium and large tests in a forked, separate JVM for each test. Medium and large tests are NOT executed if there is an error in a small test.

Running a single test or all tests in a package

To run an individual test, e.g. `MyTest`, rum `mvn test -Dtest=MyTest` You can also pass multiple, individual tests as a comma-delimited list:

```
mvn test  -Dtest=MyTest1,MyTest2,MyTest3
```

You can also pass a package, which will run all tests under the package:

```
mvn test '-Dtest=org.apache.hadoop.hbase.client.*'
```

When `-Dtest` is specified, the `localTests` profile will be used. Each junit test is executed in a separate JVM (A fork per test class). There is no parallelization when tests are running in this mode. You will see a new message at the end of the -report: `"[INFO] Tests are skipped"`. It's harmless. However, you need to make sure the sum of `Tests run:` in the `Results:` section of test reports matching the number of tests you specified because no error will be reported when a non-existent test case is specified.

Other test invocation permutations

Running `mvn test -P runSmallTests` will execute "small" tests only, using a single JVM.

Running `mvn test -P runMediumTests` will execute "medium" tests only, launching a new JVM for each test-class.

Running `mvn test -P runLargeTests` will execute "large" tests only, launching a new JVM for each test-class.

For convenience, you can run `mvn test -P runDevTests` to execute both small and medium tests, using a single JVM.

Running tests faster

By default, `$ mvn test -P runAllTests` runs all tests using a quarter of the CPUs available on machine hosting the test run (see `surefire.firstPartForkCount` and `surefire.secondPartForkCount` in the top-level hbase `pom.xml` which default to 0.25C, or 1/4 of CPU count). Up these counts to get the build to run faster. You can also have hbase modules run their tests in parrallel when the dependency graph allows by passing `--threads=N` when you invoke maven, where `N` is the amount of parallelism wanted. maven, where `N` is the amount of *module* parallelism wanted.

For example, allowing that you want to use all cores on a machine to run tests, you could start up the maven test run with:

```
$ x="1.0C";  mvn -Dsurefire.firstPartForkCount=$x -Dsurefire.secondPartForkCount=
$x test -PrunAllTests
```

If a 32 core machine, you should see periods during which 32 forked jvms appear in your process listing each running unit tests. Your milage may vary. Dependent on hardware, overcommittment of CPU and/or memory can bring the test suite crashing down, usually complaining with a spew of test system exits and incomplete test report xml files. Start gently, with the default fork and move up gradually.

Adding the `--threads=N`, maven will run N maven modules in parallel (when module inter-dependencies allow). Be aware, if you have set the forkcount to `1.0C`, and the `--threads` count to '2', the number of concurrent test runners can approach 2 * CPU, a count likely to overcommit the host machine (with attendant test exits failures).

You will need ~2.2GB of memory per forked JVM plus the memory used by maven itself (3-4G).

RAM Disk

To increase the speed, you can as well use a ramdisk. 2-3G should be sufficient. Be sure to delete the files between each test run. The typical way to configure a ramdisk on Linux is:

```
$ sudo mkdir /ram2G
sudo mount -t tmpfs -o size=2048M tmpfs /ram2G
```

You can then use it to run all HBase tests on 2.0 with the command:

```
mvn test -PrunAllTests -Dtest.build.data.basedirectory=/ram2G
```

hbasetests.sh

It's also possible to use the script `hbasetests.sh`. This script runs the medium and large tests in parallel with two maven instances, and provides a single report. This script does not use the hbase version of surefire so no parallelization is being done other than the two maven instances the script sets up. It must be executed from the directory which contains the *pom.xml*.

For example running `./dev-support/hbasetests.sh` will execute small and medium tests. Running `./dev-support/hbasetests.sh runAllTests` will execute all tests. Running `./dev-support/hbasetests.sh replayFailed` will rerun the failed tests a second time, in a separate jvm and without parallelisation.

Test Timeouts

The HBase unit test sizing Categorization timeouts are not strictly enforced.

Any test that runs longer than ten minutes will be timedout/killed.

As of hbase-2.0.0, we have purged all per-test-method timeouts: i.e.

```
...
  @Test(timeout=30000)
  public void testCreateHRegionInfoName() throws Exception {
    // ...
  }
```

They are discouraged and don't make much sense given we are timing base of how long the whole Test Fixture/Class/Suite takes and that the variance in how long a test method takes varies wildly dependent upon context (loaded Apache Infrastructure versus developer machine with nothing else running on it).

Test Resource Checker

A custom Maven SureFire plugin listener checks a number of resources before and after each HBase unit test runs and logs its findings at the end of the test output files which can be found in *target/surefire-reports* per Maven module (Tests write test reports named for the test class into this directory. Check the **-out.txt* files). The resources counted are the number of threads, the number of file descriptors, etc. If the number has increased, it adds a *LEAK?* comment in the logs. As you can have an HBase instance running in the background, some threads can be deleted/created without any specific action in the test. However, if the test does not work as expected, or if the test should not impact these resources, it's worth checking these log lines `...hbase.ResourceChecker(157): before...` and `...hbase.ResourceChecker(157): after...`. For example:

```
2012-09-26 09:22:15,315 INFO [pool-1-thread-1]
hbase.ResourceChecker(157): after:
regionserver.TestColumnSeeking#testReseeking Thread=65 (was 65),
OpenFileDescriptor=107 (was 107), MaxFileDescriptor=10240 (was 10240),
ConnectionCount=1 (was 1)
```

Writing Tests

General rules

- As much as possible, tests should be written as category small tests.
- All tests must be written to support parallel execution on the same machine, hence they should not use shared resources as fixed ports or fixed file names.
- Tests should not overlog. More than 100 lines/second makes the logs complex to read and use i/o that are hence not available for the other tests.
- Tests can be written with `HBaseTestingUtility`. This class offers helper functions to create a temp directory and do the cleanup, or to start a cluster.

Categories and execution time

- All tests must be categorized, if not they could be skipped.
- All tests should be written to be as fast as possible.
- See Unit Tests for test case categories and corresponding timeouts. This should ensure a good parallelization for people using it, and ease the analysis when the test fails.

Sleeps in tests

Whenever possible, tests should not use `Thread.sleep`, but rather waiting for the real event they need. This is faster and clearer for the reader. Tests should not do a `Thread.sleep` without testing an ending condition. This allows understanding what the test is waiting for. Moreover, the test will work whatever the machine performance is. Sleep should be minimal to be as fast as possible. Waiting for a variable should be done in a 40ms sleep loop. Waiting for a socket operation should be done in a 200 ms sleep loop.

Tests using a cluster

Tests using a HRegion do not have to start a cluster: A region can use the local file system. Start/stopping a cluster cost around 10 seconds. They should not be started per test method but per test class. Started cluster must be shutdown using `HBaseTestingUtility#shutdownMiniCluster`, which cleans the directories. As most as possible, tests should use the default settings for the cluster. When they don't, they should document it. This will allow to share the cluster later.

Tests Skeleton Code

Here is a test skeleton code with Categorization and a Category-based timeout rule to copy and paste and use as basis for test contribution.

```
/**
 * Describe what this testcase tests. Talk about resources initialized in @Before
Class (before
 * any test is run) and before each test is run, etc.
 */
// Specify the category as explained in Unit Tests section.
@Category(SmallTests.class)
public class TestExample {
  // Replace the TestExample.class in the below with the name of your test fixtur
e class.
  private static final Log LOG = LogFactory.getLog(TestExample.class);
```

```
  // Handy test rule that allows you subsequently get the name of the current met
hod. See
  // down in 'testExampleFoo()' where we use it to log current test's name.
  @Rule public TestName testName = new TestName();

  // The below rule does two things. It decides the timeout based on the category
  // (small/medium/large) of the testcase. This @Rule requires that the full test
case runs
  // within this timeout irrespective of individual test methods' times. The seco
nd
  // feature is we'll dump in the log when the test is done a count of threads st
ill
  // running.
  @Rule public static TestRule timeout = CategoryBasedTimeout.builder().
    withTimeout(this.getClass()).withLookingForStuckThread(true).build();

  @Before
  public void setUp() throws Exception {
  }

  @After
  public void tearDown() throws Exception {
  }

  @Test
  public void testExampleFoo() {
    LOG.info("Running test " + testName.getMethodName());
  }
}
```

Integration Tests

HBase integration/system tests are tests that are beyond HBase unit tests. They are generally long-lasting, sizeable (the test can be asked to 1M rows or 1B rows), targetable (they can take configuration that will point them at the ready-made cluster they are to run against; integration tests do not include cluster start/stop code), and verifying success, integration tests rely on public APIs only; they do not attempt to examine server internals asserting success/fail. Integration tests are what you would run when you need to more elaborate proofing of a release candidate beyond what unit tests can do. They are not generally run on the Apache Continuous Integration build server, however, some sites opt to run integration tests as a part of their continuous testing on an actual cluster.

Integration tests currently live under the *src/test* directory in the hbase-it submodule and will match the regex: *IntegrationTest.java*. All integration tests are also annotated with `@Category(IntegrationTests.class)`.

Integration tests can be run in two modes: using a mini cluster, or against an actual distributed cluster. Maven failsafe is used to run the tests using the mini cluster. IntegrationTestsDriver class is used for executing the tests against a distributed cluster. Integration tests SHOULD NOT assume that they are running against a mini cluster, and SHOULD NOT use private API's to access cluster state. To interact with the distributed or mini cluster uniformly, `IntegrationTestingUtility`, and `HBaseCluster` classes, and public client API's can be used.

On a distributed cluster, integration tests that use ChaosMonkey or otherwise manipulate services thru cluster manager (e.g. restart regionservers) use SSH to do it. To run these, test process should be able to run commands on remote end, so ssh should be configured accordingly (for example, if HBase runs under hbase user in your cluster, you can set up passwordless ssh for that user and run the test also under it). To facilitate that, `hbase.it.clustermanager.ssh.user`, `hbase.it.clustermanager.ssh.opts` and `hbase.it.clustermanager.ssh.cmd` configuration settings can be used. "User" is the remote user that cluster manager should use to perform ssh commands. "Opts" contains additional options that are passed to SSH (for example, "-i /tmp/my-key"). Finally, if you have some custom environment setup, "cmd" is the override format for the entire tunnel (ssh) command. The default string is {`/usr/bin/ssh %1$s %2$s%3$s%4$s "%5$s"`} and is a good starting point. This is a standard Java format string with 5 arguments that is used to execute the remote command. The argument 1 (%1$s) is SSH options set the via opts setting or via environment variable, 2 is SSH user name, 3 is "@" if username is set or "" otherwise, 4 is the target host name, and 5 is the logical command to execute (that may include single quotes, so don't use them). For example, if you run the tests under non-hbase user and want to ssh as that user and change to hbase on remote machine, you can use:

```
/usr/bin/ssh %1$s %2$s%3$s%4$s "su hbase - -c \"%5$s\""
```

That way, to kill RS (for example) integration tests may run:

```
{/usr/bin/ssh some-hostname "su hbase - -c \"ps aux | ... | kill ...\""}
```

The command is logged in the test logs, so you can verify it is correct for your environment.

To disable the running of Integration Tests, pass the following profile on the command line `-PskipIntegrationTests`. For example,

```
$ mvn clean install test -Dtest=TestZooKeeper  -PskipIntegrationTests
```

Running integration tests against mini cluster

HBase 0.92 added a `verify` maven target. Invoking it, for example by doing `mvn verify`, will run all the phases up to and including the verify phase via the maven failsafe plugin, running all the above mentioned HBase unit tests as well as tests that are in the HBase integration test group. After you have completed `mvn install -DskipTests` You can run just the integration tests by invoking:

```
cd hbase-it
mvn verify
```

If you just want to run the integration tests in top-level, you need to run two commands. First:

```
mvn failsafe:integration-test
```

This actually runs ALL the integration tests.

> This command will always output `BUILD SUCCESS` even if there are test failures.

At this point, you could grep the output by hand looking for failed tests. However, maven will do this for us; just use:

```
mvn failsafe:verify
```

The above command basically looks at all the test results (so don't remove the 'target' directory) for test failures and reports the results.

Running a subset of Integration tests

This is very similar to how you specify running a subset of unit tests (see above), but use the property `it.test` instead of `test`. To just run `IntegrationTestClassXYZ.java`, use:

```
mvn failsafe:integration-test -Dit.test=IntegrationTestClassXYZ -DfailIfNoTests=f
alse
```

The next thing you might want to do is run groups of integration tests, say all integration tests that are named IntegrationTestClassX*.java:

```
mvn failsafe:integration-test -Dit.test=*ClassX* -DfailIfNoTests=false
```

This runs everything that is an integration test that matches **ClassX**. This means anything matching: "***/IntegrationTest*ClassX**". You can also run multiple groups of integration tests using comma-delimited lists (similar to unit tests). Using a list of matches still supports full regex matching for each of the groups. This would look something like:

```
mvn failsafe:integration-test -Dit.test=*ClassX*,*ClassY -DfailIfNoTests=false
```

Running integration tests against distributed cluster

If you have an already-setup HBase cluster, you can launch the integration tests by invoking the class `IntegrationTestsDriver`. You may have to run test-compile first. The configuration will be picked by the bin/hbase script.

```
mvn test-compile
```

Then launch the tests with:

```
bin/hbase [--config config_dir] org.apache.hadoop.hbase.IntegrationTestsDriver
```

Pass `-h` to get usage on this sweet tool. Running the IntegrationTestsDriver without any argument will launch tests found under `hbase-it/src/test`, having `@Category(IntegrationTests.class)` annotation, and a name starting with `IntegrationTests`. See the usage, by passing -h, to see how to filter test classes. You can pass a regex which is checked against the full class name; so, part of class name can be used. IntegrationTestsDriver uses Junit to run the tests. Currently there is no support for running integration tests against a distributed cluster using maven (see HBASE-6201).

The tests interact with the distributed cluster by using the methods in the `DistributedHBaseCluster` (implementing `HBaseCluster`) class, which in turn uses a pluggable `ClusterManager`. Concrete implementations provide actual functionality for carrying out deployment-specific and environment-dependent tasks (SSH, etc). The default `ClusterManager` is `HBase`

`ClusterManager`, which uses SSH to remotely execute start/stop/kill/signal commands, and assumes some posix commands (ps, etc). Also assumes the user running the test has enough "power" to start/stop servers on the remote machines. By default, it picks up `HBASE_SSH_OPTS`, `HBASE_HOME`, `HBASE_CONF_DIR` from the env, and uses `bin/hbase-daemon.sh` to carry out the actions. Currently tarball deployments, deployments which uses *hbase-daemons.sh*, and Apache Ambari deployments are supported. */etc/init.d/* scripts are not supported for now, but it can be easily added. For other deployment options, a ClusterManager can be implemented and plugged in.

Some integration tests define a *main* method as entry point, and can be run on its' own, rather than using the test driver. For example, the *itbll* test can be run as follows:

```
bin/hbase org.apache.hadoop.hbase.test.IntegrationTestBigLinkedList loop 2 1 1000
00 /temp 1 1000 50 1 0
```

> The *hbase* script assumes all integration tests with exposed *main* methods to be run against a distributed cluster will follow the **IntegrationTest** regex naming pattern mentioned above, in order to proper set test dependencies into the classpath.

Destructive integration / system tests (ChaosMonkey)

HBase 0.96 introduced a tool named `ChaosMonkey`, modeled after same-named tool by Netflix's Chaos Monkey tool. ChaosMonkey simulates real-world faults in a running cluster by killing or disconnecting random servers, or injecting other failures into the environment. You can use ChaosMonkey as a stand-alone tool to run a policy while other tests are running. In some environments, ChaosMonkey is always running, in order to constantly check that high availability and fault tolerance are working as expected.

ChaosMonkey defines **Actions** and **Policies**.

Actions:

Actions are predefined sequences of events, such as the following:

- Restart active master (sleep 5 sec)
- Restart random regionserver (sleep 5 sec)
- Restart random regionserver (sleep 60 sec)
- Restart META regionserver (sleep 5 sec)

- Restart ROOT regionserver (sleep 5 sec)
- Batch restart of 50% of regionservers (sleep 5 sec)
- Rolling restart of 100% of regionservers (sleep 5 sec)

Policies:

A policy is a strategy for executing one or more actions. The default policy executes a random action every minute based on predefined action weights. A given policy will be executed until ChaosMonkey is interrupted.

Most ChaosMonkey actions are configured to have reasonable defaults, so you can run ChaosMonkey against an existing cluster without any additional configuration. The following example runs ChaosMonkey with the default configuration:

```
$ bin/hbase org.apache.hadoop.hbase.chaos.util.ChaosMonkeyRunner

12/11/19 23:21:57 INFO util.ChaosMonkey: Using ChaosMonkey Policy: class org.apac
he.hadoop.hbase.util.ChaosMonkey$PeriodicRandomActionPolicy, period:60000
12/11/19 23:21:57 INFO util.ChaosMonkey: Sleeping for 26953 to add jitter
12/11/19 23:22:24 INFO util.ChaosMonkey: Performing action: Restart active master
12/11/19 23:22:24 INFO util.ChaosMonkey: Killing master:master.example.com,60000,
1353367210440
12/11/19 23:22:24 INFO hbase.HBaseCluster: Aborting Master: master.example.com,60
000,1353367210440
12/11/19 23:22:24 INFO hbase.ClusterManager: Executing remote command: ps aux | g
rep master | grep -v grep | tr -s ' ' | cut -d ' ' -f2 | xargs kill -s SIGKILL ,
hostname:master.example.com
12/11/19 23:22:25 INFO hbase.ClusterManager: Executed remote command, exit code:0
, output:
12/11/19 23:22:25 INFO hbase.HBaseCluster: Waiting service:master to stop: maste
r.example.com,60000,1353367210440
12/11/19 23:22:25 INFO hbase.ClusterManager: Executing remote command: ps aux | g
rep master | grep -v grep | tr -s ' ' | cut -d ' ' -f2 , hostname:master.example.
com
12/11/19 23:22:25 INFO hbase.ClusterManager: Executed remote command, exit code:0
, output:
12/11/19 23:22:25 INFO util.ChaosMonkey: Killed master server:master.example.com,
60000,1353367210440
12/11/19 23:22:25 INFO util.ChaosMonkey: Sleeping for:5000
12/11/19 23:22:30 INFO util.ChaosMonkey: Starting master:master.example.com
12/11/19 23:22:30 INFO hbase.HBaseCluster: Starting Master on: master.example.com
12/11/19 23:22:30 INFO hbase.ClusterManager: Executing remote command: /homes/eni
s/code/hbase-0.94/bin/../bin/hbase-daemon.sh --config /homes/enis/code/hbase-0.9
4/bin/../conf start master , hostname:master.example.com
12/11/19 23:22:31 INFO hbase.ClusterManager: Executed remote command, exit code:0
, output:starting master, logging to /homes/enis/code/hbase-0.94/bin/../logs/hbas
e-enis-master-master.example.com.out
....
12/11/19 23:22:33 INFO util.ChaosMonkey: Started master: master.example.com,6000
0,1353367210440
```

```
12/11/19 23:22:33 INFO util.ChaosMonkey: Sleeping for:51321
12/11/19 23:23:24 INFO util.ChaosMonkey: Performing action: Restart random region
server
12/11/19 23:23:24 INFO util.ChaosMonkey: Killing region server:rs3.example.com,60
020,1353367027826
12/11/19 23:23:24 INFO hbase.HBaseCluster: Aborting RS: rs3.example.com,60020,135
3367027826
12/11/19 23:23:24 INFO hbase.ClusterManager: Executing remote command: ps aux | g
rep regionserver | grep -v grep | tr -s ' ' | cut -d ' ' -f2 | xargs kill -s SIGK
ILL , hostname:rs3.example.com
12/11/19 23:23:25 INFO hbase.ClusterManager: Executed remote command, exit code:0
, output:
12/11/19 23:23:25 INFO hbase.HBaseCluster: Waiting service:regionserver to stop:
rs3.example.com,60020,1353367027826
12/11/19 23:23:25 INFO hbase.ClusterManager: Executing remote command: ps aux | g
rep regionserver | grep -v grep | tr -s ' ' | cut -d ' ' -f2 , hostname:rs3.examp
le.com
12/11/19 23:23:25 INFO hbase.ClusterManager: Executed remote command, exit code:0
, output:
12/11/19 23:23:25 INFO util.ChaosMonkey: Killed region server:rs3.example.com,600
20,1353367027826. Reported num of rs:6
12/11/19 23:23:25 INFO util.ChaosMonkey: Sleeping for:60000
12/11/19 23:24:25 INFO util.ChaosMonkey: Starting region server:rs3.example.com
12/11/19 23:24:25 INFO hbase.HBaseCluster: Starting RS on: rs3.example.com
12/11/19 23:24:25 INFO hbase.ClusterManager: Executing remote command: /homes/eni
s/code/hbase-0.94/bin/../bin/hbase-daemon.sh --config /homes/enis/code/hbase-0.9
4/bin/../conf start regionserver , hostname:rs3.example.com
12/11/19 23:24:26 INFO hbase.ClusterManager: Executed remote command, exit code:0
, output:starting regionserver, logging to /homes/enis/code/hbase-0.94/bin/../log
s/hbase-enis-regionserver-rs3.example.com.out

12/11/19 23:24:27 INFO util.ChaosMonkey: Started region server:rs3.example.com,60
020,1353367027826. Reported num of rs:6
```

The output indicates that ChaosMonkey started the default `PeriodicRandomActionPolicy` policy, which is configured with all the available actions. It chose to run `RestartActiveMaster` and `RestartRandomRs` actions.

ChaosMonkey without SSH

Chaos monkey can be run without SSH using the Chaos service and ZNode cluster manager. HBase ships with many cluster managers, available in the `hbase-it/src/test/java/org/apache/hadoop/hbase/` directory.

Set the following property in hbase configuration to switch to `ZNodeClusterManager`:

```
<property>
  <name>hbase.it.clustermanager.class</name>
```

```
  <value>org.apache.hadoop.hbase.ZNodeClusterManager</value>
</property>
```

Start chaos agent on all hosts where you want to test chaos scenarios.

```
$ bin/hbase org.apache.hadoop.hbase.chaos.ChaosService -c start
```

Start chaos monkey runner from any one host, preferrably an edgenode. An example log while running chaos monkey with default policy `PeriodicRandomActionPolicy` is as shown below:

```
$ bin/hbase org.apache.hadoop.hbase.chaos.util.ChaosMonkeyRunner

INFO  [main] hbase.HBaseCommonTestingUtility: Instantiating org.apache.hadoop.hba
se.ZNodeClusterManager
INFO  [ReadOnlyZKClient-host1.example.com:2181,host2.example.com:2181,host3.examp
le.com:2181@0x003d43fe] zookeeper.ZooKeeper: Initiating client connection, connec
tString=host1.example.com:2181,host2.example.com:2181,host3.example.com:2181 sess
ionTimeout=90000 watcher=org.apache.hadoop.hbase.zookeeper.ReadOnlyZKClient$$Lamb
da$19/2106254492@1a39cf8
INFO  [ReadOnlyZKClient-host1.example.com:2181,host2.example.com:2181,host3.examp
le.com:2181@0x003d43fe] zookeeper.ClientCnxnSocket: jute.maxbuffer value is 41943
04 Bytes
INFO  [ReadOnlyZKClient-host1.example.com:2181,host2.example.com:2181,host3.examp
le.com:2181@0x003d43fe] zookeeper.ClientCnxn: zookeeper.request.timeout value is
0. feature enabled=
INFO  [ReadOnlyZKClient-host1.example.com:2181,host2.example.com:2181,host3.examp
le.com:2181@0x003d43fe-SendThread(host2.example.com:2181)] zookeeper.ClientCnxn:
Opening socket connection to server host2.example.com/10.20.30.40:2181. Will not
attempt to authenticate using SASL (unknown error)
INFO  [ReadOnlyZKClient-host1.example.com:2181,host2.example.com:2181,host3.examp
le.com:2181@0x003d43fe-SendThread(host2.example.com:2181)] zookeeper.ClientCnxn:
Socket connection established, initiating session, client: /10.20.30.40:35164, se
rver: host2.example.com/10.20.30.40:2181
INFO  [ReadOnlyZKClient-host1.example.com:2181,host2.example.com:2181,host3.examp
le.com:2181@0x003d43fe-SendThread(host2.example.com:2181)] zookeeper.ClientCnxn:
Session establishment complete on server host2.example.com/10.20.30.40:2181, sess
ionid = 0x101de9204670877, negotiated timeout = 60000
INFO  [main] policies.Policy: Using ChaosMonkey Policy class org.apache.hadoop.hb
ase.chaos.policies.PeriodicRandomActionPolicy, period=60000 ms
 [ChaosMonkey-2] policies.Policy: Sleeping for 93741 ms to add jitter
INFO  [ChaosMonkey-0] policies.Policy: Sleeping for 9752 ms to add jitter
INFO  [ChaosMonkey-1] policies.Policy: Sleeping for 65562 ms to add jitter
INFO  [ChaosMonkey-3] policies.Policy: Sleeping for 38777 ms to add jitter
INFO  [ChaosMonkey-0] actions.CompactRandomRegionOfTableAction: Performing actio
n: Compact random region of table usertable, major=false
INFO  [ChaosMonkey-0] policies.Policy: Sleeping for 59532 ms
INFO  [ChaosMonkey-3] client.ConnectionImplementation: Getting master connection
state from TTL Cache
```

```
INFO  [ChaosMonkey-3] client.ConnectionImplementation: Getting master state using
rpc call
INFO  [ChaosMonkey-3] actions.DumpClusterStatusAction: Cluster status
Master: host1.example.com,16000,1678339058222
Number of backup masters: 0
Number of live region servers: 3
  host1.example.com,16020,1678794551244
  host2.example.com,16020,1678341258970
  host3.example.com,16020,1678347834336
Number of dead region servers: 0
Number of unknown region servers: 0
Average load: 123.6666666666666
Number of requests: 118645157
Number of regions: 2654
Number of regions in transition: 0
INFO  [ChaosMonkey-3] policies.Policy: Sleeping for 89614 ms
```

For info on more customisations we can see help for the `ChaosMonkeyRunner`. For example we can pass the table name on which the chaos operations to be performed etc. Below is the output of the help command, listing all the supported options.

```
$ bin/hbase org.apache.hadoop.hbase.chaos.util.ChaosMonkeyRunner --help

usage: hbase org.apache.hadoop.hbase.chaos.util.ChaosMonkeyRunner <options>
Options:
 -c <arg>             Name of extra configurations file to find on CLASSPATH
 -m,--monkey <arg>    Which chaos monkey to run
 -monkeyProps <arg>   The properties file for specifying chaos monkey properties.
 -tableName <arg>     Table name in the test to run chaos monkey against
 -familyName <arg>    Family name in the test to run chaos monkey against
```

For example, running the following will start `ServerKillingMonkeyFactory` that chooses among actions to rolling batch restart RS, graceful rolling restart RS one at a time, restart active master, force balancer run etc.

```
$ bin/hbase org.apache.hadoop.hbase.chaos.util.ChaosMonkeyRunner -m org.apache.ha
doop.hbase.chaos.factories.ServerKillingMonkeyFactory
```

Available Policies

HBase ships with several ChaosMonkey policies, available in the `hbase/hbase-it/src/test/java/org/apache/hadoop/hbase/chaos/policies/` directory.

Configuring Individual ChaosMonkey Actions

ChaosMonkey integration tests can be configured per test run. Create a Java properties file in the HBase CLASSPATH and pass it to ChaosMonkey using the `-monkeyProps` configuration flag. Configurable properties, along with their default values if applicable, are listed in the `org.apache.hadoop.hbase.chaos.factories.MonkeyConstants` class. For properties that have defaults, you can override them by including them in your properties file.

The following example uses a properties file called `monkey.properties`.

```
$ bin/hbase org.apache.hadoop.hbase.IntegrationTestIngest -m slowDeterministic -m
onkeyProps monkey.properties
```

The above command will start the integration tests and chaos monkey. It will look for the properties file *monkey.properties* on the HBase CLASSPATH; e.g. inside the HBASE *conf* dir.

Here is an example chaos monkey file:

Example ChaosMonkey Properties File

```
sdm.action1.period=120000
sdm.action2.period=40000
move.regions.sleep.time=80000
move.regions.max.time=1000000
move.regions.sleep.time=80000
batch.restart.rs.ratio=0.4f
```

Periods/time are expressed in milliseconds.

HBase 1.0.2 and newer adds the ability to restart HBase's underlying ZooKeeper quorum or HDFS nodes. To use these actions, you need to configure some new properties, which have no reasonable defaults because they are deployment-specific, in your ChaosMonkey properties file, which may be `hbase-site.xml` or a different properties file.

```
<property>
  <name>hbase.it.clustermanager.hadoop.home</name>
  <value>$HADOOP_HOME</value>
</property>
<property>
  <name>hbase.it.clustermanager.zookeeper.home</name>
```

```
  <value>$ZOOKEEPER_HOME</value>
</property>
<property>
  <name>hbase.it.clustermanager.hbase.user</name>
  <value>hbase</value>
</property>
<property>
  <name>hbase.it.clustermanager.hadoop.hdfs.user</name>
  <value>hdfs</value>
</property>
<property>
  <name>hbase.it.clustermanager.zookeeper.user</name>
  <value>zookeeper</value>
</property>
```

Customizing Destructive ChaosMonkey Actions

The session above shows how to setup custom configurations for the *slowDeterministic* monkey policy. This is a policy that pre-defines a set of destructive actions of varying gravity for a running cluster. These actions are grouped into three categories: *light weight*, *mid weight* and *heavy weight*. Although it's possible to define some properties for the different actions (such as timeouts, frequency, etc), the actions themselves are not configurable.

For certain deployments, it may be interesting to define its own test strategy, either less or more aggressive than the pre-defined set of actions provided by *slowDeterministic*. For such cases, the *configurableSlowDeterministic* policy can be used. It allows for a customizable set of *heavy weight* actions to be defined in the *monkey.properties* properties file:

```
batch.restart.rs.ratio=0.3f
heavy.actions=RestartRandomRsAction(500000);MoveRandomRegionOfTableAction(36000
0,$table_name);SplitAllRegionOfTableAction($table_name)
```

The above properties file definition instructs chaos monkey to perform a RegionServer crash every 8 minutes, a random region move every 6 minutes, and at least one split of all table regions.

To run this policy, just specify *configurableSlowDeterministic* as the monkey policy to run, together with a property file containing the *heavy.actions* property definition:

```
$ bin/hbase org.apache.hadoop.hbase.IntegrationTestIngest -m configurableSlowDete
```

```
rministic -monkeyProps monkey.properties
```

When specifying monkey actions, make sure to define all required constructor parameters. For actions that require a table name parameter, the *$table_name* placeholder can be specified, and it will automatically resort to the table created by the integration test run.

If *heavy.actions* property is omitted in the properties file, *configurableSlowDeterministic* will just run as the *slowDeterministic* policy (it will execute all the heavy weight actions defined by *slowDeterministic* policy).

Developer Guidelines

Branches

We use Git for source code management and latest development happens on `master` branch. There are branches for past major/minor/maintenance releases and important features and bug fixes are often back-ported to them.

Policy for Fix Version in JIRA

To determine if a given fix is in a given release purely from the release numbers following rules are defined:

Fix version of X.Y.Z ⇒ fixed in all releases X.Y.Z' (where Z' = Z).
Fix version of X.Y.0 ⇒ fixed in all releases X.Y'.* (where Y' = Y).
Fix version of X.0.0 ⇒ fixed in all releases X'.*.* (where X' = X).

By this policy, fix version of 1.3.0 implies 1.4.0, but 1.3.2 does not imply 1.4.0 as we could not tell purely from the numbers which release came first.

Code Standards

Interface Classifications

Interfaces are classified both by audience and by stability level. These labels appear at the head of a class. The conventions followed by HBase are inherited by its parent project,

Hadoop.

The following interface classifications are commonly used:

InterfaceAudience

`@InterfaceAudience.Public`

APIs for users and HBase applications. These APIs will be deprecated through major versions of HBase.

`@InterfaceAudience.Private`

APIs for HBase internals developers. No guarantees on compatibility or availability in future versions. Private interfaces do not need an `@InterfaceStability` classification.

`@InterfaceAudience.LimitedPrivate(HBaseInterfaceAudience.COPROC)`

APIs for HBase coprocessor writers.

No `@InterfaceAudience` Classification:

Packages without an `@InterfaceAudience` label are considered private. Mark your new packages if publicly accessible.

> **Excluding Non-Public Interfaces from API Documentation**
>
> Only interfaces classified `@InterfaceAudience.Public` should be included in API documentation (Javadoc). Committers must add new package excludes `ExcludePackageNames` section of the *pom.xml* for new packages which do not contain public classes.

@InterfaceStability

`@InterfaceStability` is important for packages marked `@InterfaceAudience.Public`.

`@InterfaceStability.Stable`

Public packages marked as stable cannot be changed without a deprecation path or a very good reason.

`@InterfaceStability.Unstable`

Public packages marked as unstable can be changed without a deprecation path.

`@InterfaceStability.Evolving`

Public packages marked as evolving may be changed, but it is discouraged.

No `@InterfaceStability` Label: Public classes with no `@InterfaceStability` label are discouraged, and should be considered implicitly unstable.

If you are unclear about how to mark packages, ask on the development list.

Code Formatting Conventions

Please adhere to the following guidelines so that your patches can be reviewed more quickly. These guidelines have been developed based upon common feedback on patches from new contributors.

See the Code Conventions for the Java Programming Language for more information on coding conventions in Java. See Eclipse Code Formatting to setup Eclipse to check for some of these guidelines automatically.

Space Invaders

Do not use extra spaces around brackets. Use the second style, rather than the first.

```
if ( foo.equals( bar ) ) {     // don't do this
```

```
if (foo.equals(bar)) {
```

```
foo = barArray[ i ];     // don't do this
```

```
foo = barArray[i];
```

Auto Generated Code

Auto-generated code in Eclipse often uses bad variable names such as `arg0`. Use more informative variable names. Use code like the second example here.

```
public void readFields(DataInput arg0) throws IOException {    // don't do this
  foo = arg0.readUTF();                                        // don't do this
```

```
public void readFields(DataInput di) throws IOException {
  foo = di.readUTF();
```

Long Lines

Keep lines less than 100 characters. You can configure your IDE to do this automatically.

```
Bar bar = foo.veryLongMethodWithManyArguments(argument1, argument2, argument3, ar
gument4, argument5, argument6, argument7, argument8, argument9);  // don't do thi
s
```

```
Bar bar = foo.veryLongMethodWithManyArguments(
 argument1, argument2, argument3,argument4, argument5, argument6, argument7, argu
ment8, argument9);
```

Trailing Spaces

Be sure there is a line break after the end of your code, and avoid lines with nothing but whitespace. This makes diffs more meaningful. You can configure your IDE to help with this.

```
Bar bar = foo.getBar();     <--- imagine there is an extra space(s) after the sem
icolon.
```

API Documentation (Javadoc)

Don't forget Javadoc!

Javadoc warnings are checked during precommit. If the precommit tool gives you a '-1', please fix the javadoc issue. Your patch won't be committed if it adds such warnings.

Also, no `@author` tags - that's a rule.

Findbugs

`Findbugs` is used to detect common bugs pattern. It is checked during the precommit build. If errors are found, please fix them. You can run findbugs locally with `mvn findbugs:findbugs`, which will generate the `findbugs` files locally. Sometimes, you may have to write code smarter than `findbugs`. You can annotate your code to tell `findbugs` you know what you're doing, by annotating your class with the following annotation:

```
@edu.umd.cs.findbugs.annotations.SuppressWarnings(
value="HE_EQUALS_USE_HASHCODE",
justification="I know what I'm doing")
```

It is important to use the Apache-licensed version of the annotations. That generally means using annotations in the `edu.umd.cs.findbugs.annotations` package so that we can rely on the cleanroom reimplementation rather than annotations in the `javax.annotations` package.

Javadoc - Useless Defaults

Don't just leave javadoc tags the way IDE generates them, or fill redundant information in them.

```
  /**
   * @param table                              <---- don't leave them empty!
   * @param region An HRegion object.          <---- don't fill redundant informa
tion!
   * @return Foo Object foo just created.      <---- Not useful information
   * @throws SomeException                     <---- Not useful. Function declara
tions already tell that!
   * @throws BarException when something went wrong  <---- really?
   */
  public Foo createFoo(Bar bar);
```

Either add something descriptive to the tags, or just remove them. The preference is to add something descriptive and useful.

One Thing At A Time, Folks

If you submit a patch for one thing, don't do auto-reformatting or unrelated reformatting of code on a completely different area of code.

Likewise, don't add unrelated cleanup or refactorings outside the scope of your Jira.

Ambiguous Unit Tests

Make sure that you're clear about what you are testing in your unit tests and why.

Garbage-Collection Conserving Guidelines

The following guidelines were borrowed from http://engineering.linkedin.com/performance/linkedin-feed-faster-less-jvm-garbage. Keep them in mind to keep preventable garbage collection to a minimum. Have a look at the blog post for some great examples of how to refactor your code according to these guidelines.

- Be careful with Iterators

- Estimate the size of a collection when initializing
- Defer expression evaluation
- Compile the regex patterns in advance
- Cache it if you can
- String Interns are useful but dangerous

Invariants

We don't have many but what we have we list below. All are subject to challenge of course but until then, please hold to the rules of the road.

No permanent state in ZooKeeper

ZooKeeper state should transient (treat it like memory). If ZooKeeper state is deleted, hbase should be able to recover and essentially be in the same state.

- .Exceptions: There are currently a few exceptions that we need to fix around whether a table is enabled or disabled.
- Replication data is currently stored only in ZooKeeper. Deleting ZooKeeper data related to replication may cause replication to be disabled. Do not delete the replication tree, */hbase/replication/*.

> ⚠ Replication may be disrupted and data loss may occur if you delete the replication tree (*/hbase/replication/*) from ZooKeeper. Follow progress on this issue at HBASE-10295.

Running In-Situ

If you are developing Apache HBase, frequently it is useful to test your changes against a more-real cluster than what you find in unit tests. In this case, HBase can be run directly from the source in local-mode. All you need to do is run:

```
${HBASE_HOME}/bin/start-hbase.sh
```

This will spin up a full local-cluster, just as if you had packaged up HBase and installed it on your machine.

Keep in mind that you will need to have installed HBase into your local maven repository for the in-situ cluster to work properly. That is, you will need to run:

```
mvn clean install -DskipTests
```

to ensure that maven can find the correct classpath and dependencies. Generally, the above command is just a good thing to try running first, if maven is acting oddly.

Adding Metrics

After adding a new feature a developer might want to add metrics. HBase exposes metrics using the Hadoop Metrics 2 system, so adding a new metric involves exposing that metric to the hadoop system. Unfortunately the API of metrics2 changed from hadoop 1 to hadoop 2. In order to get around this a set of interfaces and implementations have to be loaded at runtime. To get an in-depth look at the reasoning and structure of these classes you can read the blog post located here. To add a metric to an existing MBean follow the short guide below:

Add Metric name and Function to Hadoop Compat Interface.

Inside of the source interface the corresponds to where the metrics are generated (eg MetricsMasterSource for things coming from HMaster) create new static strings for metric name and description. Then add a new method that will be called to add new reading.

Add the Implementation to Both Hadoop 1 and Hadoop 2 Compat modules.

Inside of the implementation of the source (eg. MetricsMasterSourceImpl in the above example) create a new histogram, counter, gauge, or stat in the init method. Then in the method that was added to the interface wire up the parameter passed in to the histogram.

Now add tests that make sure the data is correctly exported to the metrics 2 system. For this the MetricsAssertHelper is provided.

Git Best Practices

Avoid git merges.
Use `git pull --rebase` or `git fetch` followed by `git rebase`.

Do not use `git push --force`.
If the push does not work, fix the problem or ask for help.

Please contribute to this document if you think of other Git best practices.

`rebase_all_git_branches.sh`

The *dev-support/rebase_all_git_branches.sh* script is provided to help keep your Git repository clean. Use the `-h` parameter to get usage instructions. The script automatically refreshes your tracking branches, attempts an automatic rebase of each local branch against its remote branch, and gives you the option to delete any branch which represents a closed `HBASE-` JIRA. The script has one optional configuration option, the location of your Git directory. You can set a default by editing the script. Otherwise, you can pass the git directory manually by using the `-d` parameter, followed by an absolute or relative directory name, or even '.' for the current working directory. The script checks the directory for sub-directory called *.git/*, before proceeding.

Submitting Patches

If you are new to submitting patches to open source or new to submitting patches to Apache, start by reading the On Contributing Patches page from Apache Commons Project. It provides a nice overview that applies equally to the Apache HBase Project.

Make sure you review Code Formatting Conventions for code style. If your patch was generated incorrectly or your code does not adhere to the code formatting guidelines, you may be asked to redo some work.

HBase enforces code style via a maven plugin. After you've written up your changes, apply the formatter before committing.

```
$ mvn spotless:apply
```

When your commit is ready, present it to the community as a GitHub Pull Request.

Few general guidelines

- Always patch against the master branch first, even if you want to patch in another branch. HBase committers always apply patches first to the master branch, and backport as necessary. For complex patches, you may be asked to perform the backport(s) yourself.
- Submit one single PR for a single fix. If necessary, squash local commits to merge local commits into a single one first. See this Stack Overflow question for more information about squashing commits.
- Please understand that not every patch may get committed, and that feedback will likely be provided on the patch.

Unit Tests

Always add and/or update relevant unit tests when making the changes. Make sure that new/changed unit tests pass locally before submitting the patch because it is faster than waiting for presubmit result which runs full test suite. This will save your own time and effort. Use Mockito to make mocks which are very useful for testing failure scenarios by injecting appropriate failures.

If you are creating a new unit test class, notice how other unit test classes have classification/sizing annotations before class name and a static methods for setup/teardown of testing environment. Be sure to include annotations in any new unit test files. See Tests for more information on tests.

Integration Tests

Significant new features should provide an integration test in addition to unit tests, suitable for exercising the new feature at different points in its configuration space.

ReviewBoard

Patches larger than one screen, or patches that will be tricky to review, should go through ReviewBoard.

Procedure: Use ReviewBoard

1. Register for an account if you don't already have one. It does not use the credentials from issues.apache.org. Log in.
2. Click **New Review Request**.
3. Choose the `hbase-git` repository. Click Choose File to select the diff and optionally a parent diff. Click **Create Review Request**.
4. Fill in the fields as required. At the minimum, fill in the **Summary** and choose `hbase` as the **Review Group**. If you fill in the **Bugs** field, the review board links back to the relevant JIRA. The more fields you fill in, the better. Click **Publish** to make your review request public. An email will be sent to everyone in the `hbase` group, to review the patch.
5. Back in your JIRA, click , and paste in the URL of your ReviewBoard request. This attaches the ReviewBoard to the JIRA, for easy access.
6. To cancel the request, click .

For more information on how to use ReviewBoard, see the ReviewBoard documentation.

GitHub

Submitting GitHub pull requests is another accepted form of contributing patches. Refer to GitHub documentation for details on how to create pull requests.

> This section is incomplete and needs to be updated. Refer to HBASE-23557

GitHub Tooling

Browser bookmarks

Following is a useful javascript based browser bookmark that redirects from GitHub pull requests to the corresponding jira work item. This redirects based on the HBase jira ID mentioned in the issue title for the PR. Add the following javascript snippet as a browser bookmark to the tool bar. Clicking on it while you are on an HBase GitHub PR page redirects you to the corresponding jira item.

```
location.href =
  "https://issues.apache.org/jira/browse/" +
```

```
  document.getElementsByClassName("js-issue-title")[0].innerHTML.match(/HBASE-\d
+/)[0];
```

Guide for HBase Committers

Becoming a committer

Committers are responsible for reviewing and integrating code changes, testing and voting on release candidates, weighing in on design discussions, as well as other types of project contributions. The PMC votes to make a contributor a committer based on an assessment of their contributions to the project. It is expected that committers demonstrate a sustained history of high-quality contributions to the project and community involvement.

Contributions can be made in many ways. There is no single path to becoming a committer, nor any expected timeline. Submitting features, improvements, and bug fixes is the most common avenue, but other methods are both recognized and encouraged (and may be even more important to the health of HBase as a project and a community). A non-exhaustive list of potential contributions (in no particular order):

- Update the documentation for new changes, best practices, recipes, and other improvements.
- Keep the website up to date.
- Perform testing and report the results. For instance, scale testing and testing non-standard configurations is always appreciated.
- Maintain the shared Jenkins testing environment and other testing infrastructure.
- Vote on release candidates after performing validation, even if non-binding. A non-binding vote is a vote by a non-committer.
- Provide input for discussion threads on the link:/mail-lists.html[mailing lists] (which usually have `[DISCUSS]` in the subject line).
- Answer questions questions on the user or developer mailing lists and on Slack.
- Make sure the HBase community is a welcoming one and that we adhere to our link:/coc.html[Code of conduct]. Alert the PMC if you have concerns.
- Review other people's work (both code and non-code) and provide public feedback.
- Report bugs that are found, or file new feature requests.
- Triage issues and keep JIRA organized. This includes closing stale issues, labeling new issues, updating metadata, and other tasks as needed.

- Mentor new contributors of all sorts.
- Give talks and write blogs about HBase. Add these to the link:/[News] section of the website.
- Provide UX feedback about HBase, the web UI, the CLI, APIs, and the website.
- Write demo applications and scripts.
- Help attract and retain a diverse community.
- Interact with other projects in ways that benefit HBase and those other projects.

Not every individual is able to do all (or even any) of the items on this list. If you think of other ways to contribute, go for it (and add them to the list). A pleasant demeanor and willingness to contribute are all you need to make a positive impact on the HBase project. Invitations to become a committer are the result of steady interaction with the community over the long term, which builds trust and recognition.

New committers

New committers are encouraged to first read Apache's generic committer documentation:

- Apache New Committer Guide
- Apache Committer FAQ

Review

HBase committers should, as often as possible, attempt to review patches submitted by others. Ideally every submitted patch will get reviewed by a committer *within a few days*. If a committer reviews a patch they have not authored, and believe it to be of sufficient quality, then they can commit the patch. Otherwise the patch should be cancelled with a clear explanation for why it was rejected.

The list of submitted patches is in the HBase Review Queue, which is ordered by time of last modification. Committers should scan the list from top to bottom, looking for patches that they feel qualified to review and possibly commit. If you see a patch you think someone else is better qualified to review, you can mention them by username in the JIRA.

For non-trivial changes, it is required that another committer review your patches before commit. **Self-commits of non-trivial patches are not allowed.** Use the **Submit Patch** button in JIRA, just like other contributors, and then wait for a `+1` response from another committer before committing.

Reject

Patches which do not adhere to the guidelines in HowToContribute and to the code review checklist should be rejected. Committers should always be polite to contributors and try to instruct and encourage them to contribute better patches. If a committer wishes to improve an unacceptable patch, then it should first be rejected, and a new patch should be attached by the committer for further review.

Commit

Committers commit patches to the Apache HBase GIT repository.

> ⚠ **Before you commit!!!!**
>
> Make sure your local configuration is correct, especially your identity and email. Examine the output of the `$ git config --list` command and be sure it is correct. See Set Up Git if you need pointers.

When you commit a patch:

1. Include the Jira issue ID in the commit message along with a short description of the change. Try to add something more than just the Jira title so that someone looking at `git log` output doesn't have to go to Jira to discern what the change is about. Be sure to get the issue ID right, because this causes Jira to link to the change in Git (use the issue's "All" tab to see these automatic links).

2. Commit the patch to a new branch based off `master` or the other intended branch. It's a good idea to include the JIRA ID in the name of this branch. Check out the relevant target branch where you want to commit, and make sure your local branch has all remote changes, by doing a `git pull --rebase` or another similar command. Next, cherry-pick the change into each relevant branch (such as master), and push the changes to the remote branch using a command such as `git push <remote-server> <remote-branch>`.

   > ⚠ If you do not have all remote changes, the push will fail. If the push fails for any reason, fix the problem or ask for help. Do not do a `git push --force`.

   Before you can commit a patch, you need to determine how the patch was created. The instructions and preferences around the way to create patches have changed, and there will be a transition period.

Determine How a Patch Was Created

- If the first few lines of the patch look like the headers of an email, with a From, Date, and Subject, it was created using `git format-patch`. This is the preferred way, because you can reuse the submitter's commit message. If the commit message is not appropriate, you can still use the commit, then run `git commit --amend` and reword as appropriate.
- If the first line of the patch looks similar to the following, it was created using +git diff+ without `--no-prefix`. This is acceptable too. Notice the `a` and `b` in front of the file names. This is the indication that the patch was not created with `--no-prefix`.

```
diff --git a/src/main/asciidoc/_chapters/developer.adoc b/src/main/asciidoc/_chapters/developer.adoc
```

- If the first line of the patch looks similar to the following (without the `a` and `b`), the patch was created with `git diff --no-prefix` and you need to add `-p0` to the `git apply` command below.

```
diff --git src/main/asciidoc/_chapters/developer.adoc src/main/asciidoc/_chapters/developer.adoc
```

Example of committing a Patch

One thing you will notice with these examples is that there are a lot of `git pull` commands. The only command that actually writes anything to the remote repository is `git push`, and you need to make absolutely sure you have the correct versions of everything and don't have any conflicts before pushing. The extra `git pull` commands are usually redundant, but better safe than sorry.

The first example shows how to apply a patch that was generated with +git format-patch+ and apply it to the `master` and `branch-1` branches.

The directive to use `git format-patch` rather than `git diff`, and not to use `--no-prefix`, is a new one. See the second example for how to apply a patch created with `git diff`, and educate the person who created the patch.

```
$ git checkout -b HBASE-XXXX
$ git am ~/Downloads/HBASE-XXXX-v2.patch --signoff  # If you are committing s
omeone else's patch.
$ git checkout master
$ git pull --rebase
$ git cherry-pick <sha-from-commit>
# Resolve conflicts if necessary or ask the submitter to do it
$ git pull --rebase          # Better safe than sorry
$ git push origin master

# Backport to branch-1
$ git checkout branch-1
$ git pull --rebase
$ git cherry-pick <sha-from-commit>
# Resolve conflicts if necessary
$ git pull --rebase          # Better safe than sorry
$ git push origin branch-1
$ git branch -D HBASE-XXXX
```

This example shows how to commit a patch that was created using `git diff` without `--no-prefix`. If the patch was created with `--no-prefix`, add `-p0` to the `git apply` command.

```
$ git apply ~/Downloads/HBASE-XXXX-v2.patch
$ git commit -m "HBASE-XXXX Really Good Code Fix (Joe Schmo)" --author=<contr
ibutor> -a  # This and next command is needed for patches created with 'git d
iff'
$ git commit --amend --signoff
$ git checkout master
$ git pull --rebase
$ git cherry-pick <sha-from-commit>
# Resolve conflicts if necessary or ask the submitter to do it
$ git pull --rebase          # Better safe than sorry
$ git push origin master

# Backport to branch-1
$ git checkout branch-1
$ git pull --rebase
$ git cherry-pick <sha-from-commit>
# Resolve conflicts if necessary or ask the submitter to do it
$ git pull --rebase           # Better safe than sorry
$ git push origin branch-1
$ git branch -D HBASE-XXXX
```

3. Resolve the issue as fixed, thanking the contributor. Always set the "Fix Version" at this point, but only set a single fix version for each branch where the change was committed, the earliest release in that branch in which the change will appear.

Commit Message Format

The commit message should contain the JIRA ID and a description of what the patch does. The preferred commit message format is:

```
<jira-id> <jira-title> (<contributor-name-if-not-commit-author>)
```

```
HBASE-12345 Fix All The Things (jane@example.com)
```

If the contributor used `git format-patch` to generate the patch, their commit message is in their patch and you can use that, but be sure the JIRA ID is at the front of the commit message, even if the contributor left it out.

Use GitHub's "Co-authored-by" when there are multiple authors

We've established the practice of committing to master and then cherry picking back to branches whenever possible, unless

- it's breaking compat: In which case, if it can go in minor releases, backport to branch-1 and branch-2.
- it's a new feature: No for maintenance releases, For minor releases, discuss and arrive at consensus.

There are occasions when there are multiple author for a patch. For example when there is a minor conflict we can fix it up and just proceed with the commit. The amending author will be different from the original committer, so you should also attribute to the original author by adding one or more `Co-authored-by` trailers to the commit's message. See the GitHub documentation for "Creating a commit with multiple authors".

In short, these are the steps to add Co-authors that will be tracked by GitHub:

1. Collect the name and email address for each co-author.
2. Commit the change, but after your commit description, instead of a closing quotation, add two empty lines. (Do not close the commit message with a quotation mark)
3. On the next line of the commit message, type `Co-authored-by: name <name@example.com>`. After the co-author information, add a closing quotation mark.

Here is the example from the GitHub page, using 2 Co-authors:

```
$ git commit -m "Refactor usability tests.
>
>
Co-authored-by: name <name@example.com>
Co-authored-by: another-name <another-name@example.com>"
```

Note: `Amending-Author: Author <committer@apache>` was used prior to this DISCUSSION.

Close related GitHub PRs

As a project we work to ensure there's a JIRA associated with each change, but we don't mandate any particular tool be used for reviews. Due to implementation details of the ASF's integration between hosted git repositories and GitHub, the PMC has no ability to directly close PRs on our GitHub repo. In the event that a contributor makes a Pull Request on GitHub, either because the contributor finds that easier than attaching a patch to JIRA or because a reviewer prefers that UI for examining changes, it's important to make note of the PR in the commit that goes to the master branch so that PRs are kept up to date.

To read more about the details of what kinds of commit messages will work with the GitHub "close via keyword in commit" mechanism see the GitHub documentation for "Closing issues using keywords". In summary, you should include a line with the phrase "closes #XXX", where the XXX is the pull request id. The pull request id is usually given in the GitHub UI in grey at the end of the subject heading.

Committers are responsible for making sure commits do not break the build or tests

If a committer commits a patch, it is their responsibility to make sure it passes the test suite. It is helpful if contributors keep an eye out that their patch does not break the hbase build and/or tests, but ultimately, a contributor cannot be expected to be aware of all the particular vagaries and interconnections that occur in a project like HBase. A committer should.

Patching Etiquette

In the thread HBase, mail # dev - ANNOUNCEMENT: Git Migration In Progress (WAS ⇒ Re: Git Migration), it was agreed on the following patch flow

1. Develop and commit the patch against master first.
2. Try to cherry-pick the patch when backporting if possible.

3. If this does not work, manually commit the patch to the branch.

Merge Commits

Avoid merge commits, as they create problems in the git history.

Committing Documentation

See appendix contributing to documentation.

How to re-trigger github Pull Request checks/re-build

A Pull Request (PR) submission triggers the hbase yetus checks. The checks make sure the patch doesn't break the build or introduce test failures. The checks take around four hours to run (They are the same set run when you submit a patch via HBASE JIRA). When finished, they add a report to the PR as a comment. If a problem w/ the patch — failed compile, checkstyle violation, or an added findbugs -- the original author makes fixes and pushes a new patch. This re-runs the checks to produce a new report.

Sometimes though, the patch is good but a flakey, unrelated test has the report vote -1 on the patch. In this case, **committers** can retrigger the check run by doing a force push of the exact same patch. Or, click on the `Console output` link which shows toward the end of the report (For example `https://builds.apache.org/job/HBase-PreCommit-GitHub-PR/job/PR-289/1/console`). This will take you to `builds.apache.org`, to the build run that failed. See the "breadcrumbs" along the top (where breadcrumbs is the listing of the directories that gets us to this particular build page). It'll look something like `Jenkins > HBase-PreCommit-GitHub-PR > PR-289 > #1`. Click on the PR number — i.e. PR-289 in our example — and then, when you've arrived at the PR page, find the 'Build with Parameters' menu-item (along top left-hand menu). Click here and then `Build` leaving the JIRA_ISSUE_KEY empty. This will re-run your checks.

Dialog

Committers should hang out in the #hbase room on irc.freenode.net for real-time discussions. However any substantive discussion (as with any off-list project-related discussion) should be re-iterated in Jira or on the developer list.

Do not edit JIRA comments

Misspellings and/or bad grammar is preferable to the disruption a JIRA comment edit.

The hbase-thirdparty dependency and shading/relocation

A new project was created for the release of hbase-2.0.0. It was called `hbase-thirdparty`. This project exists only to provide the main hbase project with relocated — or shaded — versions of popular thirdparty libraries such as guava, netty, and protobuf. The mainline HBase project relies on the relocated versions of these libraries gotten from hbase-thirdparty rather than on finding these classes in their usual locations. We do this so we can specify whatever the version we wish. If we don't relocate, we must harmonize our version to match that which hadoop, spark, and other projects use.

For developers, this means you need to be careful referring to classes from netty, guava, protobuf, gson, etc. (see the hbase-thirdparty pom.xml for what it provides). Devs must refer to the hbase-thirdparty provided classes. In practice, this is usually not an issue (though it can be a bit of a pain). You will have to hunt for the relocated version of your particular class. You'll find it by prepending the general relocation prefix of `org.apache.hbase.thirdparty.`. For example if you are looking for `com.google.protobuf.Message`, the relocated version used by HBase internals can be found at `org.apache.hbase.thirdparty.com.google.protobuf.Message`.

For a few thirdparty libs, like protobuf (see the protobuf chapter in this book for the why), your IDE may give you both options — the `com.google.protobuf.*` and the `org.apache.hbase.thirdparty.com.google.protobuf.*` — because both classes are on your CLASSPATH. Unless you are doing the particular juggling required in Coprocessor Endpoint development (again see above cited protobuf chapter), you'll want to use the shaded version, always.

The `hbase-thirdparty` project has groupid of `org.apache.hbase.thirdparty`. As of this writing, it provides three jars; one for netty with an artifactid of `hbase-thirdparty-netty`, one for protobuf at `hbase-thirdparty-protobuf` and then a jar for all else — gson, guava — at `hbase-thirdpaty-miscellaneous`.

The hbase-thirdparty artifacts are a product produced by the Apache HBase project under the aegis of the HBase Project Management Committee. Releases are done via the usual voting project on the hbase dev mailing list. If issue in the hbase-thirdparty, use the hbase JIRA and mailing lists to post notice.

Development of HBase-related Maven archetypes

The development of HBase-related Maven archetypes was begun with HBASE-14876. For an overview of the hbase-archetypes infrastructure and instructions for developing new HBase-related Maven archetypes, please see `hbase/hbase-archetypes/README.md`.

Unit Testing HBase Applications

Much of the information comes from a community blog post about testing HBase applications. For information on unit tests for HBase itself, see hbase.tests.

Starting from HBase 2.5.0, HBaseTestingUtility is deprecated and should only be used when writing UTs inside HBase. End users should use org.apache.hadoop.hbase.testing.TestingHBaseCluster instead.

JUnit

HBase uses JUnit for unit tests

This example will add unit tests to the following example class:

```
public class MyHBaseDAO {

    public static void insertRecord(Table.getTable(table), HBaseTestObj obj)
    throws Exception {
        Put put = createPut(obj);
        table.put(put);
    }

    private static Put createPut(HBaseTestObj obj) {
        Put put = new Put(Bytes.toBytes(obj.getRowKey()));
        put.add(Bytes.toBytes("CF"), Bytes.toBytes("CQ-1"),
                    Bytes.toBytes(obj.getData1()));
        put.add(Bytes.toBytes("CF"), Bytes.toBytes("CQ-2"),
                    Bytes.toBytes(obj.getData2()));
        return put;
    }
}
```

The first step is to add JUnit dependencies to your Maven POM file:

```
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.11</version>
  <scope>test</scope>
</dependency>
```

Next, add some unit tests to your code. Tests are annotated with `@Test`. Here, the unit tests are in bold.

```
public class TestMyHbaseDAOData {
  @Test
  public void testCreatePut() throws Exception {
  HBaseTestObj obj = new HBaseTestObj();
  obj.setRowKey("ROWKEY-1");
  obj.setData1("DATA-1");
  obj.setData2("DATA-2");
  Put put = MyHBaseDAO.createPut(obj);
  assertEquals(obj.getRowKey(), Bytes.toString(put.getRow()));
  assertEquals(obj.getData1(), Bytes.toString(put.get(Bytes.toBytes("CF"), Bytes.
toBytes("CQ-1")).get(0).getValue()));
  assertEquals(obj.getData2(), Bytes.toString(put.get(Bytes.toBytes("CF"), Bytes.
toBytes("CQ-2")).get(0).getValue()));
  }
}
```

These tests ensure that your `createPut` method creates, populates, and returns a `Put` object with expected values. Of course, JUnit can do much more than this. For an introduction to JUnit, see https://github.com/junit-team/junit/wiki/Getting-started.

Mockito

Mockito is a mocking framework. It goes further than JUnit by allowing you to test the interactions between objects without having to replicate the entire environment. You can read more about Mockito at its project site, https://code.google.com/p/mockito/.

You can use Mockito to do unit testing on smaller units. For instance, you can mock a `org.apache.hadoop.hbase.Server` instance or a `org.apache.hadoop.hbase.master.MasterServices` interface reference rather than a full-blown `org.apache.hadoop.hbase.master.HMaster`.

This example builds upon the example code in unit.tests, to test the `insertRecord` method.

First, add a dependency for Mockito to your Maven POM file.

```
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-core</artifactId>
    <version>2.1.0</version>
    <scope>test</scope>
```

```
</dependency>
```

Next, add a `@RunWith` annotation to your test class, to direct it to use Mockito.

```
@RunWith(MockitoJUnitRunner.class)
public class TestMyHBaseDAO{
  @Mock
  Configuration config = HBaseConfiguration.create();
  @Mock
  Connection connection = ConnectionFactory.createConnection(config);
  @Mock
  private Table table;
  @Captor
  private ArgumentCaptor putCaptor;

  @Test
  public void testInsertRecord() throws Exception {
    //return mock table when getTable is called
    when(connection.getTable(TableName.valueOf("tablename")).thenReturn(table);
    //create test object and make a call to the DAO that needs testing
    HBaseTestObj obj = new HBaseTestObj();
    obj.setRowKey("ROWKEY-1");
    obj.setData1("DATA-1");
    obj.setData2("DATA-2");
    MyHBaseDAO.insertRecord(table, obj);
    verify(table).put(putCaptor.capture());
    Put put = putCaptor.getValue();

    assertEquals(Bytes.toString(put.getRow()), obj.getRowKey());
    assert(put.has(Bytes.toBytes("CF"), Bytes.toBytes("CQ-1")));
    assert(put.has(Bytes.toBytes("CF"), Bytes.toBytes("CQ-2")));
    assertEquals(Bytes.toString(put.get(Bytes.toBytes("CF"),Bytes.toBytes("CQ-
1")).get(0).getValue()), "DATA-1");
    assertEquals(Bytes.toString(put.get(Bytes.toBytes("CF"),Bytes.toBytes("CQ-
2")).get(0).getValue()), "DATA-2");
  }
}
```

This code populates `HBaseTestObj` with "ROWKEY-1", "DATA-1", "DATA-2" as values. It then inserts the record into the mocked table. The Put that the DAO would have inserted is captured, and values are tested to verify that they are what you expected them to be.

The key here is to manage Connection and Table instance creation outside the DAO. This allows you to mock them cleanly and test Puts as shown above. Similarly, you can now expand into other operations such as Get, Scan, or Delete.

MRUnit

Apache MRUnit is a library that allows you to unit-test MapReduce jobs. You can use it to test HBase jobs in the same way as other MapReduce jobs.

Given a MapReduce job that writes to an HBase table called `MyTest`, which has one column family called `CF`, the reducer of such a job could look like the following:

```
public class MyReducer extends TableReducer<Text, Text, ImmutableBytesWritable> {
   public static final byte[] CF = "CF".getBytes();
   public static final byte[] QUALIFIER = "CQ-1".getBytes();
   public void reduce(Text key, Iterable<Text> values, Context context) throws IO
Exception, InterruptedException {
     //bunch of processing to extract data to be inserted, in our case, let's say
we are simply
     //appending all the records we receive from the mapper for this particular
     //key and insert one record into HBase
     StringBuffer data = new StringBuffer();
     Put put = new Put(Bytes.toBytes(key.toString()));
     for (Text val : values) {
         data = data.append(val);
     }
     put.add(CF, QUALIFIER, Bytes.toBytes(data.toString()));
     //write to HBase
     context.write(new ImmutableBytesWritable(Bytes.toBytes(key.toString())), pu
t);
   }
 }
```

To test this code, the first step is to add a dependency to MRUnit to your Maven POM file.

```
<dependency>
   <groupId>org.apache.mrunit</groupId>
   <artifactId>mrunit</artifactId>
   <version>1.0.0 </version>
   <scope>test</scope>
</dependency>
```

Next, use the ReducerDriver provided by MRUnit, in your Reducer job.

```
public class MyReducerTest {
    ReduceDriver<Text, Text, ImmutableBytesWritable, Writable> reduceDriver;
    byte[] CF = "CF".getBytes();
    byte[] QUALIFIER = "CQ-1".getBytes();

    @Before
    public void setUp() {
```

```
      MyReducer reducer = new MyReducer();
      reduceDriver = ReduceDriver.newReduceDriver(reducer);
    }

   @Test
   public void testHBaseInsert() throws IOException {
      String strKey = "RowKey-1", strValue = "DATA", strValue1 = "DATA1",
strValue2 = "DATA2";
      List<Text> list = new ArrayList<Text>();
      list.add(new Text(strValue));
      list.add(new Text(strValue1));
      list.add(new Text(strValue2));
      //since in our case all that the reducer is doing is appending the records
that the mapper
      //sends it, we should get the following back
      String expectedOutput = strValue + strValue1 + strValue2;
     //Setup Input, mimic what mapper would have passed
      //to the reducer and run test
      reduceDriver.withInput(new Text(strKey), list);
      //run the reducer and get its output
      List<Pair<ImmutableBytesWritable, Writable>> result = reduceDriver.run();

      //extract key from result and verify
      assertEquals(Bytes.toString(result.get(0).getFirst().get()), strKey);

      //extract value for CF/QUALIFIER and verify
      Put a = (Put)result.get(0).getSecond();
      String c = Bytes.toString(a.get(CF, QUALIFIER).get(0).getValue());
      assertEquals(expectedOutput,c );
   }
}
```

Your MRUnit test verifies that the output is as expected, the Put that is inserted into HBase has the correct value, and the ColumnFamily and ColumnQualifier have the correct values.

MRUnit includes a MapperDriver to test mapping jobs, and you can use MRUnit to test other operations, including reading from HBase, processing data, or writing to HDFS,

Integration Testing with an HBase Mini-Cluster

HBase ships with HBaseTestingUtility, which makes it easy to write integration tests using a *mini-cluster*. The first step is to add some dependencies to your Maven POM file. Check the versions to be sure they are appropriate.

```
<properties>
  <hbase.version>2.0.0-SNAPSHOT</hbase.version>
</properties>
```

```
<dependencies>
  <dependency>
    <groupId>org.apache.hbase</groupId>
    <artifactId>hbase-testing-util</artifactId>
    <version>${hbase.version}</version>
    <scope>test</scope>
  </dependency>
</dependencies>
```

This code represents an integration test for the MyDAO insert shown in unit.tests.

```
public class MyHBaseIntegrationTest {
    private static HBaseTestingUtility utility;
    byte[] CF = "CF".getBytes();
    byte[] CQ1 = "CQ-1".getBytes();
    byte[] CQ2 = "CQ-2".getBytes();

    @Before
    public void setup() throws Exception {
        utility = new HBaseTestingUtility();
        utility.startMiniCluster();
    }

    @Test
    public void testInsert() throws Exception {
        Table table = utility.createTable(Bytes.toBytes("MyTest"), CF);
        HBaseTestObj obj = new HBaseTestObj();
        obj.setRowKey("ROWKEY-1");
        obj.setData1("DATA-1");
        obj.setData2("DATA-2");
        MyHBaseDAO.insertRecord(table, obj);
        Get get1 = new Get(Bytes.toBytes(obj.getRowKey()));
        get1.addColumn(CF, CQ1);
        Result result1 = table.get(get1);
        assertEquals(Bytes.toString(result1.getRow()), obj.getRowKey());
        assertEquals(Bytes.toString(result1.value()), obj.getData1());
        Get get2 = new Get(Bytes.toBytes(obj.getRowKey()));
        get2.addColumn(CF, CQ2);
        Result result2 = table.get(get2);
        assertEquals(Bytes.toString(result2.getRow()), obj.getRowKey());
        assertEquals(Bytes.toString(result2.value()), obj.getData2());
    }
}
```

Starting from HBase 2.5.0, it is recommended to use TestingHBaseCluster instead.

```
public class MyHBaseIntegrationTest {

  private TestingHBaseCluster cluster;

  private Connection conn;
```

```
  private Admin admin;

  private TableName tableName = TableName.valueOf("MyTest");

  byte[] CF = "CF".getBytes();
  byte[] CQ1 = "CQ-1".getBytes();
  byte[] CQ2 = "CQ-2".getBytes();

  @Before
  public void setUp() throws Exception {
    cluster = TestingHBaseCluster.create(TestingHBaseClusterOption.builder().buil
d());
    cluster.start();
    conn = ConnectionFactory.createConnection(cluster.getConf());
    admin = conn.getAdmin();
    admin.createTable(TableDescriptorBuilder.newBuilder(tableName)
      .setColumnFamily(ColumnFamilyDescriptorBuilder.of(CF)).build());
  }

  @After
  public void tearDown() throws Exception {
    admin.close();
    conn.close();
    cluster.stop();
  }

  @Test
  public void testInsert() throws Exception {
    try (Table table = conn.getTable(tableName)) {
      HBaseTestObj obj = new HBaseTestObj();
      obj.setRowKey("ROWKEY-1");
      obj.setData1("DATA-1");
      obj.setData2("DATA-2");
      MyHBaseDAO.insertRecord(table, obj);
      Get get1 = new Get(Bytes.toBytes(obj.getRowKey()));
      get1.addColumn(CF, CQ1);
      Result result1 = table.get(get1);
      assertEquals(Bytes.toString(result1.getRow()), obj.getRowKey());
      assertEquals(Bytes.toString(result1.value()), obj.getData1());
      Get get2 = new Get(Bytes.toBytes(obj.getRowKey()));
      get2.addColumn(CF, CQ2);
      Result result2 = table.get(get2);
      assertEquals(Bytes.toString(result2.getRow()), obj.getRowKey());
      assertEquals(Bytes.toString(result2.value()), obj.getData2());
    }
  }
}
```

This code creates an HBase mini-cluster and starts it. Next, it creates a table called `MyTest` with one column family, `CF`. A record is inserted, a Get is performed from the same table, and the insertion is verified.

> Starting the mini-cluster takes about 20-30 seconds, but that should be appropriate for integration testing.

See the paper at **HBase Case-Study: Using HBaseTestingUtility for Local Testing and Development** (2010) for more information about HBaseTestingUtility.

Protobuf in HBase

Protobuf

HBase uses Google's protobufs wherever it persists metadata — in the tail of hfiles or Cells written by HBase into the system hbase:meta table or when HBase writes znodes to zookeeper, etc. — and when it passes objects over the wire making RPCs. HBase uses protobufs to describe the RPC Interfaces (Services) we expose to clients, for example the `Admin` and `Client` Interfaces that the RegionServer fields, or specifying the arbitrary extensions added by developers via our Coprocessor Endpoint mechanism.

With protobuf, you describe serializations and services in a `.protos` file. You then feed these descriptors to a protobuf tool, the `protoc` binary, to generate classes that can marshall and unmarshall the described serializations and field the specified Services.

See the `README.txt` in the HBase sub-modules for details on how to run the class generation on a per-module basis; e.g. see `hbase-protocol/README.txt` for how to generate protobuf classes in the hbase-protocol module.

In HBase, `.proto` files are either in the `hbase-protocol` module; a module dedicated to hosting the common proto files and the protoc generated classes that HBase uses internally serializing metadata. For extensions to hbase such as REST or Coprocessor Endpoints that need their own descriptors; their protos are located inside the function's hosting module: e.g. `hbase-rest` is home to the REST proto files and the `hbase-rsgroup` table grouping Coprocessor Endpoint has all protos that have to do with table grouping.

Protos are hosted by the module that makes use of them. While this makes it so generation of protobuf classes is distributed, done per module, we do it this way so modules encapsulate all to do with the functionality they bring to hbase.

Extensions whether REST or Coprocessor Endpoints will make use of core HBase protos found back in the hbase-protocol module. They'll use these core protos when they want to serialize a Cell or a Put or refer to a particular node via ServerName, etc., as part of providing the CPEP Service. Going forward, after the release of hbase-2.0.0, this practice needs to whither. We'll explain why in the later hbase-2.0.0 section.

hbase-2.0.0 and the shading of protobufs (HBASE-15638)

As of hbase-2.0.0, our protobuf usage gets a little more involved. HBase core protobuf references are offset so as to refer to a private, bundled protobuf. Core stops referring to protobuf classes at com.google.protobuf._ and instead references protobuf at the HBase-specific offset org.apache.hadoop.hbase.shaded.com.google.protobuf._. We do this indirection so hbase core can evolve its protobuf version independent of whatever our dependencies rely on. For instance, HDFS serializes using protobuf. HDFS is on our CLASSPATH. Without the above described indirection, our protobuf versions would have to align. HBase would be stuck on the HDFS protobuf version until HDFS decided to upgrade. HBase and HDFS versions would be tied.

We had to move on from protobuf-2.5.0 because we need facilities added in protobuf-3.1.0; in particular being able to save on copies and avoiding bringing protobufs onheap for serialization/deserialization.

In hbase-2.0.0, we introduced a new module, `hbase-protocol-shaded` inside which we contained all to do with protobuf and its subsequent relocation/shading. This module is in essence a copy of much of the old `hbase-protocol` but with an extra shading/relocation step. Core was moved to depend on this new module.

That said, a complication arises around Coprocessor Endpoints (CPEPs). CPEPs depend on public HBase APIs that reference protobuf classes at `com.google.protobuf.*` explicitly. For example, in our Table Interface we have the below as the means by which you obtain a CPEP Service to make invocations against:

```
...
  <T extends com.google.protobuf.Service,R> Map<byte[],R> coprocessorService(
   Class<T> service, byte[] startKey, byte[] endKey,
     org.apache.hadoop.hbase.client.coprocessor.Batch.Call<T,R> callable)
  throws com.google.protobuf.ServiceException, Throwable
```

Existing CPEPs will have made reference to core HBase protobufs specifying ServerNames or carrying Mutations. So as to continue being able to service CPEPs and their references to `com.google.protobuf.*` across the upgrade to hbase-2.0.0 and beyond, HBase needs to be able to deal with both `com.google.protobuf.*` references and its internal offset `org.apache.hadoop.hbase.shaded.com.google.protobuf.*` protobufs.

The `hbase-protocol-shaded` module hosts all protobufs used by HBase core.

But for the vestigial CPEP references to the (non-shaded) content of `hbase-protocol`, we keep around most of this module going forward just so it is available to CPEPs. Retaining the most of `hbase-protocol` makes for overlapping, 'duplicated' proto instances where some exist as non-shaded/non-relocated here in their old module location but also in the new location, shaded under `hbase-protocol-shaded`. In other words, there is an instance of the generated protobuf class `org.apache.hadoop.hbase.protobuf.generated.ServerName` in hbase-protocol and another generated instance that is the same in all regards except its protobuf references are to the internal shaded version at `org.apache.hadoop.hbase.shaded.protobuf.generated.ServerName` (note the 'shaded' addition in the middle of the package name).

If you extend a proto in `hbase-protocol-shaded` for internal use, consider extending it also in `hbase-protocol` (and regenerating).

Going forward, we will provide a new module of common types for use by CPEPs that will have the same guarantees against change as does our public API. TODO.

protobuf changes for hbase-3.0.0 (HBASE-23797)

Since hadoop(start from 3.3.x) also shades protobuf and bumps the version to 3.x, there is no reason for us to stay on protobuf 2.5.0 any more.

In HBase 3.0.0, the hbase-protocol module has been purged, the CPEP implementation should use the protos in hbase-protocol-shaded module, and also make use of the shaded protobuf in hbase-thirdparty. In general, we will keep the protobuf version compatible for a whole major release, unless there are critical problems, for example, a critical CVE on protobuf.

Add this dependency to your pom:

```
<dependency>
  <groupId>org.apache.hbase.thirdparty</groupId>
  <artifactId>hbase-shaded-protobuf</artifactId>
  <!-- use the version that your target hbase cluster uses -->
  <version>${hbase-thirdparty.version}</version>
  <scope>provided</scope>
</dependency>
```

And typically you also need to add this plugin to your pom to make your generated protobuf code also use the shaded and relocated protobuf version in hbase-thirdparty.

```
<plugin>
  <groupId>com.google.code.maven-replacer-plugin</groupId>
  <artifactId>replacer</artifactId>
  <version>1.5.3</version>
  <executions>
    <execution>
      <phase>process-sources</phase>
      <goals>
        <goal>replace</goal>
      </goals>
    </execution>
  </executions>
  <configuration>
    <basedir>${basedir}/target/generated-sources/</basedir>
      <includes>
        <include>**/*.java</include>
      </includes>
      <!-- Ignore errors when missing files, because it means this build
           was run with -Dprotoc.skip and there is no -Dreplacer.skip -->
      <ignoreErrors>true</ignoreErrors>
      <replacements>
        <replacement>
          <token>([^\.])com.google.protobuf</token>
          <value>$1org.apache.hbase.thirdparty.com.google.protobuf</value>
        </replacement>
        <replacement>
          <token>(public)(\W+static)?(\W+final)?(\W+class)</token>
          <value>@javax.annotation.Generated("proto") $1$2$3$4</value>
        </replacement>
        <!-- replacer doesn't support anchoring or negative lookbehind -->
        <replacement>
          <token>(@javax.annotation.Generated\("proto"\) ){2}</token>
          <value>$1</value>
        </replacement>
      </replacements>
  </configuration>
</plugin>
```

In hbase-examples module, we have some examples under the `org.apache.hadoop.hbase.coprocessor.example` package. You can see `BulkDeleteEndpoint` and `BulkDelete.proto` for more details, and you can also check the `pom.xml` of hbase-examples module to see how to make use of the above plugin.

Procedure Framework (Pv2)

See HBASE-12439 for the original implementation.

With Pv2 you can build and run state machines. It was built by Matteo to make distributed state transitions in HBase resilient in the face of process failures. Previous to Pv2, state transition handling was spread about the codebase with implementation varying by transition-type and context. Pv2 was inspired by FATE, of Apache Accumulo.

Early Pv2 aspects have been shipping in HBase with a good while now but it has continued to evolve as it takes on more involved scenarios. What we have now is powerful but intricate in operation and incomplete, in need of cleanup and hardening. In this doc we have given overview on the system so you can make use of it (and help with its polishing).

This system has the awkward name of Pv2 because HBase already had the notion of a Procedure used in snapshots (see hbase-server *org.apache.hadoop.hbase.procedure* as opposed to hbase-procedure *org.apache.hadoop.hbase.procedure2*). Pv2 supercedes and is to replace Procedure.

Procedures

A Procedure is a transform made on an HBase entity. Examples of HBase entities would be Regions and Tables. Procedures are run by a ProcedureExecutor instance. Procedure current state is kept in the ProcedureStore. The ProcedureExecutor has but a primitive view on what goes on inside a Procedure. From its PoV, Procedures are submitted and then the ProcedureExecutor keeps calling *#execute(Object)* until the Procedure is done. Execute may be called multiple times in the case of failure or restart, so Procedure code must be idempotent yielding the same result each time it run. Procedure code can also implement *rollback* so steps can be undone if failure. A call to *execute()* can result in one of following possibilities:

- *execute()* returns
  - *null*: indicates we are done.
  - *this*: indicates there is more to do so, persist current procedure state and re-*execute()*.
  - *Array* of sub-procedures: indicates a set of procedures needed to be run to completion before we can proceed (after which we expect the framework to call our

execute again).

- *execute()* throws exception
  - *suspend*: indicates execution of procedure is suspended and can be resumed due to some external event. The procedure state is persisted.
  - *yield*: procedure is added back to scheduler. The procedure state is not persisted.
  - *interrupted*: currently same as *yield*.
  - Any *exception* not listed above: Procedure *state* is changed to *FAILED* (after which we expect the framework will attempt rollback).

The ProcedureExecutor stamps the frameworks notions of Procedure State into the Procedure itself; e.g. it marks Procedures as INITIALIZING on submit. It moves the state to RUNNABLE when it goes to execute. When done, a Procedure gets marked FAILED or SUCCESS depending. Here is the list of all states as of this writing:

- *INITIALIZING* Procedure in construction, not yet added to the executor
- *RUNNABLE* Procedure added to the executor, and ready to be executed.
- *WAITING* The procedure is waiting on children (subprocedures) to be completed
- *WAITING_TIMEOUT* The procedure is waiting a timeout or an external event
- *ROLLEDBACK* The procedure failed and was rolledback.
- *SUCCESS* The procedure execution completed successfully.
- *FAILED* The procedure execution failed, may need to rollback.

After each execute, the Procedure state is persisted to the ProcedureStore. Hooks are invoked on Procedures so they can preserve custom state. Post-fault, the ProcedureExecutor re-hydrates its pre-crash state by replaying the content of the ProcedureStore. This makes the Procedure Framework resilient against process failure.

Implementation

In implementation, Procedures tend to divide transforms into finer-grained tasks and while some of these work items are handed off to sub-procedures, the bulk are done as processing *steps* in-Procedure; each invocation of the execute is used to perform a single step, and then the Procedure relinquishes returning to the framework. The Procedure does its own tracking of where it is in the processing.

What comprises a sub-task, or *step* in the execution is up to the Procedure author but generally it is a small piece of work that cannot be further decomposed and that moves the processing forward toward its end state. Having procedures made of many small steps rather than a few large ones allows the Procedure framework give out insight on where we are in the processing. It also allows the framework be more fair in its execution. As stated per above, each step may be called multiple times (failure/restart) so steps must be implemented idempotent. It is easy to confuse the state that the Procedure itself is keeping with that of the Framework itself. Try to keep them distinct.

Rollback

Rollback is called when the procedure or one of the sub-procedures has failed. The rollback step is supposed to cleanup the resources created during the execute() step. In case of failure and restart, rollback() may be called multiple times, so again the code must be idempotent.

Metrics

There are hooks for collecting metrics on submit of the procedure and on finish.

- updateMetricsOnSubmit()
- updateMetricsOnFinish()

Individual procedures can override these methods to collect procedure specific metrics. The default implementations of these methods try to get an object implementing an interface ProcedureMetrics which encapsulates following set of generic metrics:

- SubmittedCount (Counter): Total number of procedure instances submitted of a type.
- Time (Histogram): Histogram of runtime for procedure instances.
- FailedCount (Counter): Total number of failed procedure instances.

Individual procedures can implement this object and define these generic set of metrics.

Baggage

Procedures can carry baggage. One example is the *step* the procedure last attained (see previous section); procedures persist the enum that marks where they are currently. Other examples might be the Region or Server name the Procedure is currently working. After

each call to execute, the Procedure#serializeStateData is called. Procedures can persist whatever.

Result/State and Queries

(From Matteo's ProcedureV2 and Notification Bus doc)
In the case of asynchronous operations, the result must be kept around until the client asks for it. Once we receive a "get" of the result we can schedule the delete of the record. For some operations the result may be "unnecessary" especially in case of failure (e.g. if the create table fail, we can query the operation result or we can just do a list table to see if it was created) so in some cases we can schedule the delete after a timeout. On the client side the operation will return a "Procedure ID", this ID can be used to wait until the procedure is completed and get the result/exception.

```
Admin.doOperation() { longprocId=master.doOperation(); master.waitCompletion(proc
Id); }
```

If the master goes down while performing the operation the backup master will pickup the half inprogress operation and complete it. The client will not notice the failure.

Subprocedures

Subprocedures are *Procedure* instances created and returned by *#execute(Object)* method of a procedure instance (parent procedure). As subprocedures are of type *Procedure*, they can instantiate their own subprocedures. As its a recursive, procedure stack is maintained by the framework. The framework makes sure that the parent procedure does not proceed till all sub-procedures and their subprocedures in a procedure stack are successfully finished.

ProcedureExecutor

ProcedureExecutor uses *ProcedureStore* and *ProcedureScheduler* and executes procedures submitted to it. Some of the basic operations supported are:

- *abort(procId)*: aborts specified procedure if its not finished
- *submit(Procedure)*: submits procedure for execution

- *retrieve:* list of get methods to get *Procedure* instances and results
- *register/ unregister* listeners: for listening on Procedure related notifications

When *ProcedureExecutor* starts it loads procedure instances persisted in *ProcedureStore* from previous run. All unfinished procedures are resumed from the last stored state.

Nonces

You can pass the nonce that came in with the RPC to the Procedure on submit at the executor. This nonce will then be serialized along w/ the Procedure on persist. If a crash, on reload, the nonce will be put back into a map of nonces to pid in case a client tries to run same procedure for a second time (it will be rejected). See the base Procedure and how nonce is a base data member.

Wait/Wake/Suspend/Yield

'suspend' means stop processing a procedure because we can make no more progress until a condition changes; i.e. we sent RPC and need to wait on response. The way this works is that a Procedure throws a suspend exception from down in its guts as a GOTO the end-of-the-current-processing step. Suspend also puts the Procedure back on the scheduler. Problematic is we do some accounting on our way out even on suspend making it so it can take time exiting (We have to update state in the WAL).

RegionTransitionProcedure#reportTransition is called on receipt of a report from a RS. For Assign and Unassign, this event response from the server we sent an RPC wakes up suspended Assign/Unassigns.

Locking

Procedure Locks are not about concurrency! They are about giving a Procedure read/write access to an HBase Entity such as a Table or Region so that is possible to shut out other Procedures from making modifications to an HBase Entity state while the current one is running.

Locking is optional, up to the Procedure implementor but if an entity is being operated on by a Procedure, all transforms need to be done via Procedures using the same locking scheme else havoc.

Two ProcedureExecutor Worker threads can actually end up both processing the same Procedure instance. If it happens, the threads are meant to be running different parts of the one Procedure — changes that do not stamp on each other (This gets awkward around the procedure frameworks notion of ‘suspend'. More on this below).

Locks optionally may be held for the life of a Procedure. For example, if moving a Region, you probably want to have exclusive access to the HBase Region until the Region completes (or fails). This is used in conjunction with {@link #holdLock(Object)}. If {@link #holdLock(Object)} returns true, the procedure executor will call acquireLock() once and thereafter not call {@link #releaseLock(Object)} until the Procedure is done (Normally, it calls release/acquire around each invocation of {@link #execute(Object)}.

Locks also may live the life of a procedure; i.e. once an Assign Procedure starts, we do not want another procedure meddling w/ the region under assignment. Procedures that hold the lock for the life of the procedure set Procedure#holdLock to true. AssignProcedure does this as do Split and Move (If in the middle of a Region move, you do not want it Splitting).

Locking can be for life of Procedure.

Some locks have a hierarchy. For example, taking a region lock also takes (read) lock on its containing table and namespace to prevent another Procedure obtaining an exclusive lock on the hosting table (or namespace).

Procedure Types

StateMachineProcedure

One can consider each call to *#execute(Object)* method as transitioning from one state to another in a state machine. Abstract class *StateMachineProcedure* is wrapper around base *Procedure* class which provides constructs for implementing a state machine as a *Procedure*. After each state transition current state is persisted so that, in case of crash/ restart, the state transition can be resumed from the previous state of a procedure before

crash/ restart. Individual procedures need to define initial and terminus states and hooks *executeFromState()* and *setNextState()* are provided for state transitions.

RemoteProcedureDispatcher

A new RemoteProcedureDispatcher (+ subclass RSProcedureDispatcher) primitive takes care of running the Procedure-based Assignments 'remote' component. This dispatcher knows about 'servers'. It does aggregation of assignments by time on a time/count basis so can send procedures in batches rather than one per RPC. Procedure status comes back on the back of the RegionServer heartbeat reporting online/offline regions (No more notifications via ZK). The response is passed to the AMv2 to 'process'. It will check against the in-memory state. If there is a mismatch, it fences out the RegionServer on the assumption that something went wrong on the RS side. Timeouts trigger retries (Not Yet Implemented!). The Procedure machine ensures only one operation at a time on any one Region/Table using entity *locking* and smarts about what is serial and what can be run concurrently (Locking was zk-based — you'd put a znode in zk for a table — but now has been converted to be procedure-based as part of this project).

References

- Matteo had a slide deck on what it the Procedure Framework would look like and the problems it addresses initially attached to the Pv2 issue.
- A good doc by Matteo on problem and how Pv2 addresses it w/ roadmap (from the Pv2 JIRA). We should go back to the roadmap to do the Notification Bus, convertion of log splitting to Pv2, etc.

AMv2 Description for Devs

The AMv2 project is a redo of Assignment in an attempt at addressing the root cause of many of our operational issues in production, namely slow assignment and problematic accounting such that Regions are misplaced stuck offline in the notorious *Regions-In-Transition (RIT)* limbo state.

Below are notes for devs on key aspects of AMv2 in no particular order.

Background

Assignment in HBase 1.x has been problematic in operation. It is not hard to see why. Region state is kept at the other end of an RPC in ZooKeeper (Terminal states — i.e. OPEN or CLOSED — are published to the *hbase:meta* table). In HBase-1.x.x, state has multiple writers with Master and RegionServers all able to make state edits concurrently (in *hbase:meta* table and out on ZooKeeper). If clocks are awry or watchers missed, state changes can be skipped or overwritten. Locking of HBase Entities — tables, regions — is not comprehensive so a table operation — disable/enable — could clash with a region-level operation; a split or merge. Region state is distributed and hard to reason about and test. Assignment is slow in operation because each assign involves moving remote znodes through transitions. Cluster size tends to top out at a couple of hundred thousand regions; beyond this, cluster start/stop takes hours and is prone to corruption.

AMv2 (AssignmentManager Version 2) is a refactor (HBASE-14350) of the hbase-1.x AssignmentManager putting it up on a ProcedureV2 (HBASE-12439) basis. ProcedureV2 (Pv2), is an awkwardly named system that allows describing and running multi-step state machines. It is performant and persists all state to a Store which is recoverable post crash. See the companion chapter on Procedure Framework (Pv2), to learn more about the ProcedureV2 system.

In AMv2, all assignment, crash handling, splits and merges are recast as Procedures(v2). ZooKeeper is purged from the mix. As before, the final assignment state gets published to *hbase:meta* for non-Master participants to read (all-clients) with intermediate state kept in the local Pv2 WAL-based 'store' but only the active Master, a single-writer, evolves state. The Master's in-memory cluster image is the authority and if disagreement, RegionServers are forced to comply. Pv2 adds shared/exclusive locking of all core HBase Entities —

namespace, tables, and regions — to ensure one actor at a time access and to prevent operations contending over resources (move/split, disable/assign, etc.).

This redo of AM atop of a purposed, performant state machine with all operations taking on the common Procedure form with a single state writer only moves our AM to a new level of resilience and scale.

New System

Each Region Assign or Unassign of a Region is now a Procedure. A Move (Region) Procedure is a compound of Procedures; it is the running of an Unassign Procedure followed by an Assign Procedure. The Move Procedure spawns the Assign and Unassign in series and then waits on their completions.

And so on. ServerCrashProcedure spawns the WAL splitting tasks and then the reassign of all regions that were hosted on the crashed server as subprocedures.

AMv2 Procedures are run by the Master in a ProcedureExecutor instance. All Procedures make use of utility provided by the Pv2 framework.

For example, Procedures persist each state transition to the frameworks' Procedure Store. The default implementation is done as a WAL kept on HDFS. On crash, we reopen the Store and rerun all WALs of Procedure transitions to put the Assignment State Machine back into the attitude it had just before crash. We then continue Procedure execution.

In the new system, the Master is the Authority on all things Assign. Previous we were ambiguous; e.g. the RegionServer was in charge of Split operations. Master keeps an in-memory image of Region states and servers. If disagreement, the Master always prevails; at an extreme it will kill the RegionServer that is in disagreement.

A new RegionStateStore class takes care of publishing the terminal Region state, whether OPEN or CLOSED, out to the *hbase:meta* table.

RegionServers now report their run version on Connection. This version is available inside the AM for use running migrating rolling restarts.

Procedures Detail

Assign/Unassign

Assign and Unassign subclass a common RegionTransitionProcedure. There can only be one RegionTransitionProcedure per region running at a time since the RTP instance takes a lock on the region. The RTP base Procedure has three steps; a store the procedure step (REGION_TRANSITION_QUEUE); a dispatch of the procedure open or close followed by a suspend waiting on the remote regionserver to report successful open or fail (REGION_TRANSITION_DISPATCH) or notification that the server fielding the request crashed; and finally registration of the successful open/close in hbase:meta (REGION_TRANSITION_FINISH).

Here is how the assign of a region 56f985a727afe80a184dac75fbf6860c looks in the logs. The assign was provoked by a Server Crash (Process ID 1176 or pid=1176 which when it is the parent of a procedure, it is identified as ppid=1176). The assign is pid=1179, the second region of the two being assigned by this Server Crash.

```
2017-05-23 12:04:24,175 INFO  [ProcExecWrkr-30] procedure2.ProcedureExecutor: Ini
tialized subprocedures=[{pid=1178, ppid=1176, state=RUNNABLE:REGION_TRANSITION_QU
EUE; AssignProcedure table=IntegrationTestBigLinkedList, region=bfd57f0b72fd3ca77
e9d3c5e3ae48d76, target=ve0540.halxg.example.org,16020,1495525111232}, {pid=1179,
ppid=1176, state=RUNNABLE:REGION_TRANSITION_QUEUE; AssignProcedure table=Integrat
ionTestBigLinkedList, region=56f985a727afe80a184dac75fbf6860c, target=ve0540.halx
g.example.org,16020,1495525111232}]
```

Next we start the assign by queuing ('registering') the Procedure with the framework.

```
2017-05-23 12:04:24,241 INFO  [ProcExecWrkr-30] assignment.AssignProcedure: Start
pid=1179, ppid=1176, state=RUNNABLE:REGION_TRANSITION_QUEUE; AssignProcedure tabl
e=IntegrationTestBigLinkedList, region=56f985a727afe80a184dac75fbf6860c, target=v
e0540.halxg.example.org,16020,1495525111232; rit=OFFLINE, location=ve0540.halxg.e
xample.org,16020,1495525111232; forceNewPlan=false, retain=false
```

Track the running of Procedures in logs by tracing their process id — here pid=1179.

Next we move to the dispatch phase where we update hbase:meta table setting the region state as OPENING on server ve540. We then dispatch an rpc to ve540 asking it to open the region. Thereafter we suspend the Assign until we get a message back from ve540 on whether it has opened the region successfully (or not).

```
2017-05-23 12:04:24,494 INFO  [ProcExecWrkr-38] assignment.RegionStateStore: pid=
1179 updating hbase:meta row=IntegrationTestBigLinkedList,H\xE3@\x8D\x964\x9D\xDF
\x8F@9\x0F\xC8\xCC\xC2,1495566261066.56f985a727afe80a184dac75fbf6860c., regionSta
te=OPENING, regionLocation=ve0540.halxg.example.org,16020,1495525111232
2017-05-23 12:04:24,498 INFO  [ProcExecWrkr-38] assignment.RegionTransitionProced
ure: Dispatch pid=1179, ppid=1176, state=RUNNABLE:REGION_TRANSITION_DISPATCH; Ass
ignProcedure table=IntegrationTestBigLinkedList, region=56f985a727afe80a184dac75f
bf6860c, target=ve0540.halxg.example.org,16020,1495525111232; rit=OPENING, locati
on=ve0540.halxg.example.org,16020,1495525111232
```

Below we log the incoming report that the region opened successfully on ve540. The Procedure is woken up (you can tell it the procedure is running by the name of the thread, its a ProcedureExecutor thread, ProcExecWrkr-9). The woken up Procedure updates state in hbase:meta to denote the region as open on ve0540. It then reports finished and exits.

```
2017-05-23 12:04:26,643 DEBUG [RpcServer.default.FPBQ.Fifo.handler=46,queue=1,por
t=16000] assignment.RegionTransitionProcedure: Received report OPENED seqId=11984
985, pid=1179, ppid=1176, state=RUNNABLE:REGION_TRANSITION_DISPATCH; AssignProced
ure table=IntegrationTestBigLinkedList, region=56f985a727afe80a184dac75fbf6860c,
target=ve0540.halxg.example.org,16020,1495525111232; rit=OPENING, location=ve054
0.halxg.example.org,16020,1495525111232
2017-05-23 12:04:26,643 INFO  [ProcExecWrkr-9] assignment.RegionStateStore: pid=1
179 updating hbase:meta row=IntegrationTestBigLinkedList,H\xE3@\x8D\x964\x9D\xDF
\x8F@9\x0F\xC8\xCC\xC2,1495566261066.56f985a727afe80a184dac75fbf6860c., regionSta
te=OPEN, openSeqNum=11984985, regionLocation=ve0540.halxg.example.org,16020,14955
25111232
2017-05-23 12:04:26,836 INFO  [ProcExecWrkr-9] procedure2.ProcedureExecutor: Fini
sh suprocedure pid=1179, ppid=1176, state=SUCCESS; AssignProcedure table=Integrat
ionTestBigLinkedList, region=56f985a727afe80a184dac75fbf6860c, target=ve0540.halx
g.example.org,16020,1495525111232
```

Unassign looks similar given it is based on the base RegionTransitionProcedure. It has the same state transitions and does basically the same steps but with different state name (CLOSING, CLOSED).

Most other procedures are subclasses of a Pv2 StateMachine implementation. We have both Table and Region focused StateMachines types.

UI

Along the top-bar on the Master, you can now find a 'Procedures&Locks' tab which takes you to a page that is ugly but useful. It dumps currently running procedures and framework locks. Look at this when you can't figure what stuff is stuck; it will at least

identify problematic procedures (take the pid and grep the logs...). Look for ROLLEDBACK or pids that have been RUNNING for a long time.

Logging

Procedures log their process ids as pid= and their parent ids (ppid=) everywhere. Work has been done so you can grep the pid and see history of a procedure operation.

Implementation Notes

In this section we note some idiosyncrasies of operation as an attempt at saving you some head-scratching.

Region Transition RPC and RS Heartbeat can arrive at ~same time on Master

Reporting Region Transition on a RegionServer is now a RPC distinct from RS heartbeating ('RegionServerServices' Service). An heartbeat and a status update can arrive at the Master at about the same time. The Master will update its internal state for a Region but this same state is checked when heartbeat processing. We may find the unexpected; i.e. a Region just reported as CLOSED so heartbeat is surprised to find region OPEN on the back of the RS report. In the new system, all slaves must cow to the Masters' understanding of cluster state; the Master will kill/close any misaligned entities.

To address the above, we added a lastUpdate for in-memory Master state. Let a region state have some vintage before we act on it (one second currently).

Master as RegionServer or as RegionServer that just does system tables

AMv2 enforces current master branch default of HMaster carrying system tables only; i.e. the Master in an HBase cluster acts also as a RegionServer only it is the exclusive host for tables such as *hbase:meta*, *hbase:namespace*, etc., the core system tables. This is causing a couple of test failures as AMv1, though it is not supposed to, allows moving hbase:meta off Master while AMv2 does not.

New Configs

These configs all need doc on when you'd change them.

hbase.procedure.remote.dispatcher.threadpool.size

Defaults 128

hbase.procedure.remote.dispatcher.delay.msec

Default 150ms

hbase.procedure.remote.dispatcher.max.queue.size

Default 32

hbase.regionserver.rpc.startup.waittime

Default 60 seconds.

Tools

HBASE-15592 Print Procedure WAL Content

Patch in HBASE-18152 [AMv2] Corrupt Procedure WAL file; procedure data stored out of order https://issues.apache.org/jira/secure/attachment/12871066/reading_bad_wal.patch

MasterProcedureSchedulerPerformanceEvaluation

Tool to test performance of locks and queues in procedure scheduler independently from other framework components. Run this after any substantial changes in proc system. Prints nice output:

```
*****************************************
Time - addBack     : 5.0600sec
Ops/sec - addBack  : 1.9M
Time - poll        : 19.4590sec
Ops/sec - poll     : 501.9K
Num Operations     : 10000000

Completed          : 10000006
Yield              : 22025876

Num Tables         : 5
Regions per table  : 10
```

```
Operations type    : both
Threads            : 10
****************************************
Raw format for scripts

RESULT [num_ops=10000000, ops_type=both, num_table=5, regions_per_table=10, threa
ds=10, num_yield=22025876, time_addback_ms=5060, time_poll_ms=19459]
```

ZooKeeper

Apache HBase by default manages a ZooKeeper "cluster" for you. It will start and stop the ZooKeeper ensemble as part of the HBase start/stop process. You can also manage the ZooKeeper ensemble independent of HBase and just point HBase at the cluster it should use. To toggle HBase management of ZooKeeper, use the `HBASE_MANAGES_ZK` variable in *conf/hbase-env.sh*. This variable, which defaults to `true`, tells HBase whether to start/stop the ZooKeeper ensemble servers as part of HBase start/stop.

When HBase manages the ZooKeeper ensemble, you can specify ZooKeeper configuration directly in *conf/hbase-site.xml*. A ZooKeeper configuration option can be set as a property in the HBase *hbase-site.xml* XML configuration file by prefacing the ZooKeeper option name with `hbase.zookeeper.property`. For example, the `clientPort` setting in ZooKeeper can be changed by setting the `hbase.zookeeper.property.clientPort` property. For all default values used by HBase, including ZooKeeper configuration, see **hbase default configurations**. Look for the `hbase.zookeeper.property` prefix. For the full list of ZooKeeper configurations, see ZooKeeper's *zoo.cfg*. HBase does not ship with a *zoo.cfg* so you will need to browse the *conf* directory in an appropriate ZooKeeper download.

You must at least list the ensemble servers in *hbase-site.xml* using the `hbase.zookeeper.quorum` property. This property defaults to a single ensemble member at `localhost` which is not suitable for a fully distributed HBase. (It binds to the local machine only and remote clients will not be able to connect).

> **How many ZooKeepers should I run?**
>
> You can run a ZooKeeper ensemble that comprises 1 node only but in production it is recommended that you run a ZooKeeper ensemble of 3, 5 or 7 machines; the more members an ensemble has, the more tolerant the ensemble is of host failures. Also, run an odd number of machines. In ZooKeeper, an even number of peers is supported, but it is normally not used because an even sized ensemble requires, proportionally, more peers to form a quorum than an odd sized ensemble requires. For example, an ensemble with 4 peers requires 3 to form a quorum, while an ensemble with 5 also requires 3 to form a quorum. Thus, an ensemble of 5 allows 2 peers to fail, and thus is more fault tolerant than the ensemble of 4, which allows only 1 down peer.
>
> Give each ZooKeeper server around 1GB of RAM, and if possible, its own dedicated disk (A dedicated disk is the best thing you can do to ensure a performant ZooKeeper ensemble). For very heavily loaded clusters, run ZooKeeper servers on separate machines from RegionServers (DataNodes and TaskTrackers).

For example, to have HBase manage a ZooKeeper quorum on nodes *rs{1,2,3,4,5}.example.com*, bound to port 2222 (the default is 2181) ensure `HBASE_MANAGE_ZK` is commented out or set to `true` in *conf/hbase-env.sh* and then edit *conf/hbase-site.xml* and set `hbase.zookeeper.property.clientPort` and `hbase.zookeeper.quorum`. You should also set `hbase.zookeeper.property.dataDir` to other than the default as the default has ZooKeeper persist data under */tmp* which is often cleared on system restart. In the example below we have ZooKeeper persist to */user/local/zookeeper*.

```
<configuration>
  ...
  <property>
    <name>hbase.zookeeper.property.clientPort</name>
    <value>2222</value>
    <description>Property from ZooKeeper's config zoo.cfg.
    The port at which the clients will connect.
    </description>
  </property>
  <property>
    <name>hbase.zookeeper.quorum</name>
    <value>rs1.example.com,rs2.example.com,rs3.example.com,rs4.example.com,rs5.ex
ample.com</value>
    <description>Comma separated list of servers in the ZooKeeper Quorum.
    For example, "host1.mydomain.com,host2.mydomain.com,host3.mydomain.com".
    By default this is set to localhost for local and pseudo-distributed modes
    of operation. For a fully-distributed setup, this should be set to a full
    list of ZooKeeper quorum servers. If HBASE_MANAGES_ZK is set in hbase-env.sh
    this is the list of servers which we will start/stop ZooKeeper on.
    </description>
  </property>
  <property>
    <name>hbase.zookeeper.property.dataDir</name>
    <value>/usr/local/zookeeper</value>
    <description>Property from ZooKeeper's config zoo.cfg.
    The directory where the snapshot is stored.
    </description>
  </property>
  ...
</configuration>
```

What version of ZooKeeper should I use?

The newer version, the better. ZooKeeper 3.4.x is required as of HBase 1.0.0

ZooKeeper Maintenance

Be sure to set up the data dir cleaner described under ZooKeeper Maintenance else you could have 'interesting' problems a couple of months in; i.e. zookeeper could start dropping sessions if it has to run through a directory of hundreds of thousands of logs which is wont to do around

> leader reelection time — a process rare but run on occasion whether because a machine is dropped or happens to hiccup.

Using existing ZooKeeper ensemble

To point HBase at an existing ZooKeeper cluster, one that is not managed by HBase, set `HBASE_MANAGES_ZK` in *conf/hbase-env.sh* to false

```
...
# Tell HBase whether it should manage its own instance of ZooKeeper or not.
export HBASE_MANAGES_ZK=false
```

Next set ensemble locations and client port, if non-standard, in *hbase-site.xml*.

When HBase manages ZooKeeper, it will start/stop the ZooKeeper servers as a part of the regular start/stop scripts. If you would like to run ZooKeeper yourself, independent of HBase start/stop, you would do the following

```
${HBASE_HOME}/bin/hbase-daemons.sh {start,stop} zookeeper
```

Note that you can use HBase in this manner to spin up a ZooKeeper cluster, unrelated to HBase. Just make sure to set `HBASE_MANAGES_ZK` to `false` if you want it to stay up across HBase restarts so that when HBase shuts down, it doesn't take ZooKeeper down with it.

For more information about running a distinct ZooKeeper cluster, see the ZooKeeper **Getting Started Guide**. Additionally, see the **ZooKeeper Wiki** or the **ZooKeeper documentation** for more information on ZooKeeper sizing.

SASL Authentication with ZooKeeper

Newer releases of Apache HBase (>= 0.92) will support connecting to a ZooKeeper Quorum that supports SASL authentication (which is available in ZooKeeper versions 3.4.0 or later).

This describes how to set up HBase to mutually authenticate with a ZooKeeper Quorum. ZooKeeper/HBase mutual authentication (**HBASE-2418**) is required as part of a complete

secure HBase configuration (HBASE-3025). For simplicity of explication, this section ignores additional configuration required (Secure HDFS and Coprocessor configuration). It's recommended to begin with an HBase-managed ZooKeeper configuration (as opposed to a standalone ZooKeeper quorum) for ease of learning.

Operating System Prerequisites

You need to have a working Kerberos KDC setup. For each `$HOST` that will run a ZooKeeper server, you should have a principle `zookeeper/$HOST`. For each such host, add a service key (using the `kadmin` or `kadmin.local` tool's `ktadd` command) for `zookeeper/$HOST` and copy this file to `$HOST`, and make it readable only to the user that will run zookeeper on `$HOST`. Note the location of this file, which we will use below as *$PATH_TO_ZOOKEEPER_KEYTAB*.

Similarly, for each `$HOST` that will run an HBase server (master or regionserver), you should have a principle: `hbase/$HOST`. For each host, add a keytab file called *hbase.keytab* containing a service key for `hbase/$HOST`, copy this file to `$HOST`, and make it readable only to the user that will run an HBase service on `$HOST`. Note the location of this file, which we will use below as *$PATH_TO_HBASE_KEYTAB*.

Each user who will be an HBase client should also be given a Kerberos principal. This principal should usually have a password assigned to it (as opposed to, as with the HBase servers, a keytab file) which only this user knows. The client's principal's `maxrenewlife` should be set so that it can be renewed enough so that the user can complete their HBase client processes. For example, if a user runs a long-running HBase client process that takes at most 3 days, we might create this user's principal within `kadmin` with: `addprinc -maxrenewlife 3days`. The ZooKeeper client and server libraries manage their own ticket refreshment by running threads that wake up periodically to do the refreshment.

On each host that will run an HBase client (e.g. `hbase shell`), add the following file to the HBase home directory's *conf* directory:

```
Client {
  com.sun.security.auth.module.Krb5LoginModule required
  useKeyTab=false
  useTicketCache=true;
};
```

We'll refer to this JAAS configuration file as *$CLIENT_CONF* below.

HBase-managed ZooKeeper Configuration

On each node that will run a zookeeper, a master, or a regionserver, create a JAAS configuration file in the conf directory of the node's *HBASE_HOME* directory that looks like the following:

```
Server {
  com.sun.security.auth.module.Krb5LoginModule required
  useKeyTab=true
  keyTab="$PATH_TO_ZOOKEEPER_KEYTAB"
  storeKey=true
  useTicketCache=false
  principal="zookeeper/$HOST";
};
Client {
  com.sun.security.auth.module.Krb5LoginModule required
  useKeyTab=true
  useTicketCache=false
  keyTab="$PATH_TO_HBASE_KEYTAB"
  principal="hbase/$HOST";
};
```

where the *$PATH_TO_HBASE_KEYTAB* and *$PATH_TO_ZOOKEEPER_KEYTAB* files are what you created above, and `$HOST` is the hostname for that node.

The `Server` section will be used by the ZooKeeper quorum server, while the `Client` section will be used by the HBase master and regionservers. The path to this file should be substituted for the text *$HBASE_SERVER_CONF* in the *hbase-env.sh* listing below.

The path to this file should be substituted for the text *$CLIENT_CONF* in the *hbase-env.sh* listing below.

Modify your *hbase-env.sh* to include the following:

```
export HBASE_OPTS="-Djava.security.auth.login.config=$CLIENT_CONF"
export HBASE_MANAGES_ZK=true
export HBASE_ZOOKEEPER_OPTS="-Djava.security.auth.login.config=$HBASE_SERVER_CON
F"
export HBASE_MASTER_OPTS="-Djava.security.auth.login.config=$HBASE_SERVER_CONF"
export HBASE_REGIONSERVER_OPTS="-Djava.security.auth.login.config=$HBASE_SERVER_C
ONF"
```

where *$HBASE_SERVER_CONF* and *$CLIENT_CONF* are the full paths to the JAAS configuration files created above.

Modify your *hbase-site.xml* on each node that will run zookeeper, master or regionserver to contain:

```
<configuration>
  <property>
    <name>hbase.zookeeper.quorum</name>
    <value>$ZK_NODES</value>
  </property>
  <property>
    <name>hbase.cluster.distributed</name>
    <value>true</value>
  </property>
  <property>
    <name>hbase.zookeeper.property.authProvider.1</name>
    <value>org.apache.zookeeper.server.auth.SASLAuthenticationProvider</value>
  </property>
  <property>
    <name>hbase.zookeeper.property.kerberos.removeHostFromPrincipal</name>
    <value>true</value>
  </property>
  <property>
    <name>hbase.zookeeper.property.kerberos.removeRealmFromPrincipal</name>
    <value>true</value>
  </property>
</configuration>
```

where `$ZK_NODES` is the comma-separated list of hostnames of the ZooKeeper Quorum hosts.

Start your hbase cluster by running one or more of the following set of commands on the appropriate hosts:

```
bin/hbase zookeeper start
bin/hbase master start
bin/hbase regionserver start
```

External ZooKeeper Configuration

Add a JAAS configuration file that looks like:

```
Client {
  com.sun.security.auth.module.Krb5LoginModule required
  useKeyTab=true
```

```
  useTicketCache=false
  keyTab="$PATH_TO_HBASE_KEYTAB"
  principal="hbase/$HOST";
};
```

where the *$PATH_TO_HBASE_KEYTAB* is the keytab created above for HBase services to run on this host, and `$HOST` is the hostname for that node. Put this in the HBase home's configuration directory. We'll refer to this file's full pathname as *$HBASE_SERVER_CONF* below.

Modify your hbase-env.sh to include the following:

```
export HBASE_OPTS="-Djava.security.auth.login.config=$CLIENT_CONF"
export HBASE_MANAGES_ZK=false
export HBASE_MASTER_OPTS="-Djava.security.auth.login.config=$HBASE_SERVER_CONF"
export HBASE_REGIONSERVER_OPTS="-Djava.security.auth.login.config=$HBASE_SERVER_C
ONF"
```

Modify your *hbase-site.xml* on each node that will run a master or regionserver to contain:

```
<configuration>
  <property>
    <name>hbase.zookeeper.quorum</name>
    <value>$ZK_NODES</value>
  </property>
  <property>
    <name>hbase.cluster.distributed</name>
    <value>true</value>
  </property>
  <property>
    <name>hbase.zookeeper.property.authProvider.1</name>
    <value>org.apache.zookeeper.server.auth.SASLAuthenticationProvider</value>
  </property>
  <property>
    <name>hbase.zookeeper.property.kerberos.removeHostFromPrincipal</name>
    <value>true</value>
  </property>
  <property>
    <name>hbase.zookeeper.property.kerberos.removeRealmFromPrincipal</name>
    <value>true</value>
  </property>
</configuration>
```

where `$ZK_NODES` is the comma-separated list of hostnames of the ZooKeeper Quorum hosts.

Also on each of these hosts, create a JAAS configuration file containing:

```
Server {
  com.sun.security.auth.module.Krb5LoginModule required
  useKeyTab=true
  keyTab="$PATH_TO_ZOOKEEPER_KEYTAB"
  storeKey=true
  useTicketCache=false
  principal="zookeeper/$HOST";
};
```

where `$HOST` is the hostname of each Quorum host. We will refer to the full pathname of this file as *$ZK_SERVER_CONF* below.

Start your ZooKeepers on each ZooKeeper Quorum host with:

```
SERVER_JVMFLAGS="-Djava.security.auth.login.config=$ZK_SERVER_CONF" bin/zkServer
start
```

Start your HBase cluster by running one or more of the following set of commands on the appropriate nodes:

```
bin/hbase master start
bin/hbase regionserver start
```

ZooKeeper Server Authentication Log Output

If the configuration above is successful, you should see something similar to the following in your ZooKeeper server logs:

```
11/12/05 22:43:39 INFO zookeeper.Login: successfully logged in.
11/12/05 22:43:39 INFO server.NIOServerCnxnFactory: binding to port 0.0.0.0/0.0.
0.0:2181
11/12/05 22:43:39 INFO zookeeper.Login: TGT refresh thread started.
11/12/05 22:43:39 INFO zookeeper.Login: TGT valid starting at:        Mon Dec 05
22:43:39 UTC 2011
11/12/05 22:43:39 INFO zookeeper.Login: TGT expires:                  Tue Dec 06
22:43:39 UTC 2011
11/12/05 22:43:39 INFO zookeeper.Login: TGT refresh sleeping until: Tue Dec 06 1
8:36:42 UTC 2011
..
11/12/05 22:43:59 INFO auth.SaslServerCallbackHandler:
  Successfully authenticated client: authenticationID=hbase/ip-10-166-175-249.us-
west-1.compute.internal@HADOOP.LOCALDOMAIN;
```

```
  authorizationID=hbase/ip-10-166-175-249.us-west-1.compute.internal@HADOOP.LOCAL
DOMAIN.
11/12/05 22:43:59 INFO auth.SaslServerCallbackHandler: Setting authorizedID: hbas
e
11/12/05 22:43:59 INFO server.ZooKeeperServer: adding SASL authorization for auth
orizationID: hbase
```

ZooKeeper Client Authentication Log Output

On the ZooKeeper client side (HBase master or regionserver), you should see something similar to the following:

```
11/12/05 22:43:59 INFO zookeeper.ZooKeeper: Initiating client connection, connect
String=ip-10-166-175-249.us-west-1.compute.internal:2181 sessionTimeout=180000 wa
tcher=master:60000
11/12/05 22:43:59 INFO zookeeper.ClientCnxn: Opening socket connection to server
/10.166.175.249:2181
11/12/05 22:43:59 INFO zookeeper.RecoverableZooKeeper: The identifier of this pro
cess is 14851@ip-10-166-175-249
11/12/05 22:43:59 INFO zookeeper.Login: successfully logged in.
11/12/05 22:43:59 INFO client.ZooKeeperSaslClient: Client will use GSSAPI as SASL
mechanism.
11/12/05 22:43:59 INFO zookeeper.Login: TGT refresh thread started.
11/12/05 22:43:59 INFO zookeeper.ClientCnxn: Socket connection established to ip-
10-166-175-249.us-west-1.compute.internal/10.166.175.249:2181, initiating session
11/12/05 22:43:59 INFO zookeeper.Login: TGT valid starting at:        Mon Dec 05
22:43:59 UTC 2011
11/12/05 22:43:59 INFO zookeeper.Login: TGT expires:                  Tue Dec 06
22:43:59 UTC 2011
11/12/05 22:43:59 INFO zookeeper.Login: TGT refresh sleeping until: Tue Dec 06 1
8:30:37 UTC 2011
11/12/05 22:43:59 INFO zookeeper.ClientCnxn: Session establishment complete on se
rver ip-10-166-175-249.us-west-1.compute.internal/10.166.175.249:2181, sessionid
= 0x134106594320000, negotiated timeout = 180000
```

Configuration from Scratch

This has been tested on the current standard Amazon Linux AMI. First setup KDC and principals as described above. Next checkout code and run a sanity check.

```
git clone https://gitbox.apache.org/repos/asf/hbase.git
cd hbase
mvn clean test -Dtest=TestZooKeeperACL
```

Then configure HBase as described above. Manually edit target/cached_classpath.txt (see below):

```
bin/hbase zookeeper &
bin/hbase master &
bin/hbase regionserver &
```

Future improvements

Fix target/cached_classpath.txt

You must override the standard hadoop-core jar file from the `target/cached_classpath.txt` file with the version containing the HADOOP-7070 fix. You can use the following script to do this:

```
echo `find ~/.m2 -name "*hadoop-core*7070*SNAPSHOT.jar"` ':' `cat target/cached_c
lasspath.txt` | sed 's/ //g' > target/tmp.txt
mv target/tmp.txt target/cached_classpath.txt
```

Set JAAS configuration programmatically

This would avoid the need for a separate Hadoop jar that fixes HADOOP-7070.

Elimination of `kerberos.removeHostFromPrincipal` and `kerberos.removeRealmFromPrincipal`

TLS connection to ZooKeeper

Apache ZooKeeper also supports SSL/TLS client connections to encrypt the data in transmission. This is particularly useful when the ZooKeeper ensemble is running on a host different from HBase and data has to be sent over the wire.

Java system properties

The ZooKeeper client supports the following Java system properties to set up TLS connection:

```
zookeeper.client.secure=true
zookeeper.clientCnxnSocket=org.apache.zookeeper.ClientCnxnSocketNetty
zookeeper.ssl.keyStore.location="/path/to/your/keystore"
zookeeper.ssl.keyStore.password="keystore_password"
zookeeper.ssl.trustStore.location="/path/to/your/truststore"
zookeeper.ssl.trustStore.password="truststore_password"
```

Setting up KeyStore is optional and only required if ZooKeeper server requests for client certificate.

Find more detailed information in the ZooKeeper SSL User Guide.

> ⚠ These're standard Java properties which should be set in the HBase command line and are effective in the entire Java process. All ZooKeeper clients running in the same process will pick them up including co-processors.

> ℹ Since ZooKeeper version 3.8 the following two properties are useful to store the keystore and truststore passwords in protected text files rather than exposing them in the command line.

```
zookeeper.ssl.keyStore.passwordPath=/path/to/secure/file
zookeeper.ssl.trustStore.passwordPath=/path/to/secure/file
```

HBase configuration

By adding HBASE-28038, ZooKeeper client TLS settings are also available in *hbase-site.xml* via `hbase.zookeeper.property` prefix. In contrast to Java system properties this could be more convenient under some circumstances.

```
<configuration>
  <property>
    <name>hbase.zookeeper.property.client.secure</name>
    <value>true</value>
  </property>
  <property>
    <name>hbase.zookeeper.property.clientCnxnSocket</name>
    <value>org.apache.zookeeper.ClientCnxnSocketNetty</value>
  </property>
  <property>
    <name>hbase.zookeeper.property.ssl.trustStore.location</name>
    <value>/path/to/your/truststore</value>
  </property>
  ...
</configuration>
```

> ℹ These settings are eventually transformed into Java system properties, it's just a convenience feature. So, the same rules that mentioned in the previous point, applies to them as well.

Community

Decisions

Feature Branches

Feature Branches are easy to make. You do not have to be a committer to make one. Just request the name of your branch be added to JIRA up on the developer's mailing list and a committer will add it for you. Thereafter you can file issues against your feature branch in Apache HBase JIRA. Your code you keep elsewhere — it should be public so it can be observed — and you can update dev mailing list on progress. When the feature is ready for commit, 3 +1s from committers will get your feature merged. See HBase, mail # dev - Thoughts about large feature dev branches

How to set fix version in JIRA on issue resolve

Here is how we agreed to set versions in JIRA when we resolve an issue. If master is going to be 3.0.0, branch-2 will be 2.4.0, and branch-1 will be 1.7.0 then:

- Commit only to master (i.e., backward-incompatible new feature): Mark with 3.0.0
- Commit only to master and branch-2 (i.e., backward-compatible new feature, applicable only to 2.x+): Mark with 3.0.0 and 2.4.0
- Commit to master, branch-2, and branch-1 (i.e., backward-compatible new feature, applicable everywhere): Mark with 3.0.0, 2.4.0, and 1.7.0
- Commit to master, branch-2, and branch-2.3, branch-1, branch-1.4 (i.e., bug fix applicable to all active release lines): Mark with 3.0.0, 2.4.0, 2.3.x, 1.7.0, and 1.4.x
- Commit a fix to the website: no version

Policy on when to set a RESOLVED JIRA as CLOSED

We agreed that for issues that list multiple releases in their *Fix Version/s* field, CLOSE the issue on the release of any of the versions listed; subsequent change to the issue must happen in a new JIRA.

Only transient state in ZooKeeper!

You should be able to kill the data in zookeeper and hbase should ride over it recreating the zk content as it goes. This is an old adage around these parts. We just made note of it now. We also are currently in violation of this basic tenet — replication at least keeps permanent state in zk — but we are working to undo this breaking of a golden rule.

Community Roles

Release Managers

Each maintained release branch has a release manager, who volunteers to coordinate new features and bug fixes are backported to that release. The release managers are committers. If you would like your feature or bug fix to be included in a given release, communicate with that release manager. If this list goes out of date or you can't reach the listed person, reach out to someone else on the list.

Release Managers:

Release	Release Manager	Latest Release	EOL
0.94	Lars Hofhansl	0.94.27	April 2017
0.96	Michael Stack	0.96.2	September 2014
0.98	Andrew Purtell	0.98.24	April 2017
1.0	Enis Soztutar	1.0.3	January 2016
1.1	Nick Dimiduk	1.1.13	December 2017
1.2	Sean Busbey	1.2.12	June 2019
1.3	Mikhail Antonov	1.3.6	August 2020
1.4	Andrew Purtell	1.4.14	October 2021
1.5	Andrew Purtell	1.5.0	October 2019
1.6	Andrew Purtell	1.6.0	February 2020
1.7	Reid Chan	1.7.2	August 2022
2.0	Michael Stack	2.0.6	September 2019
2.1	Duo Zhang	2.1.10	May 2020

Release	Release Manager	Latest Release	EOL
2.2	Guanghao Zhang	2.2.7	April 2021
2.3	Nick Dimiduk	2.3.7	October 2021
2.4	Andrew Purtell	2.4.18	June 2024
2.5	Andrew Purtell	Check the download page	NOT YET
2.6	Bryan Beaudreault	Check the download page	NOT YET

Commit Message format

We agreed to the following Git commit message format:

```
HBASE-xxxxx <title>. (<contributor>)
```

If the person making the commit is the contributor, leave off the '(<contributor>)' element.

hbtop

Usage

You can run hbtop with the following command:

```
$ hbase hbtop
```

In this case, the values of `hbase.client.zookeeper.quorum` and `zookeeper.znode.parent` in `hbase-site.xml` in the classpath or the default values of them are used to connect.

Or, you can specify your own zookeeper quorum and znode parent as follows:

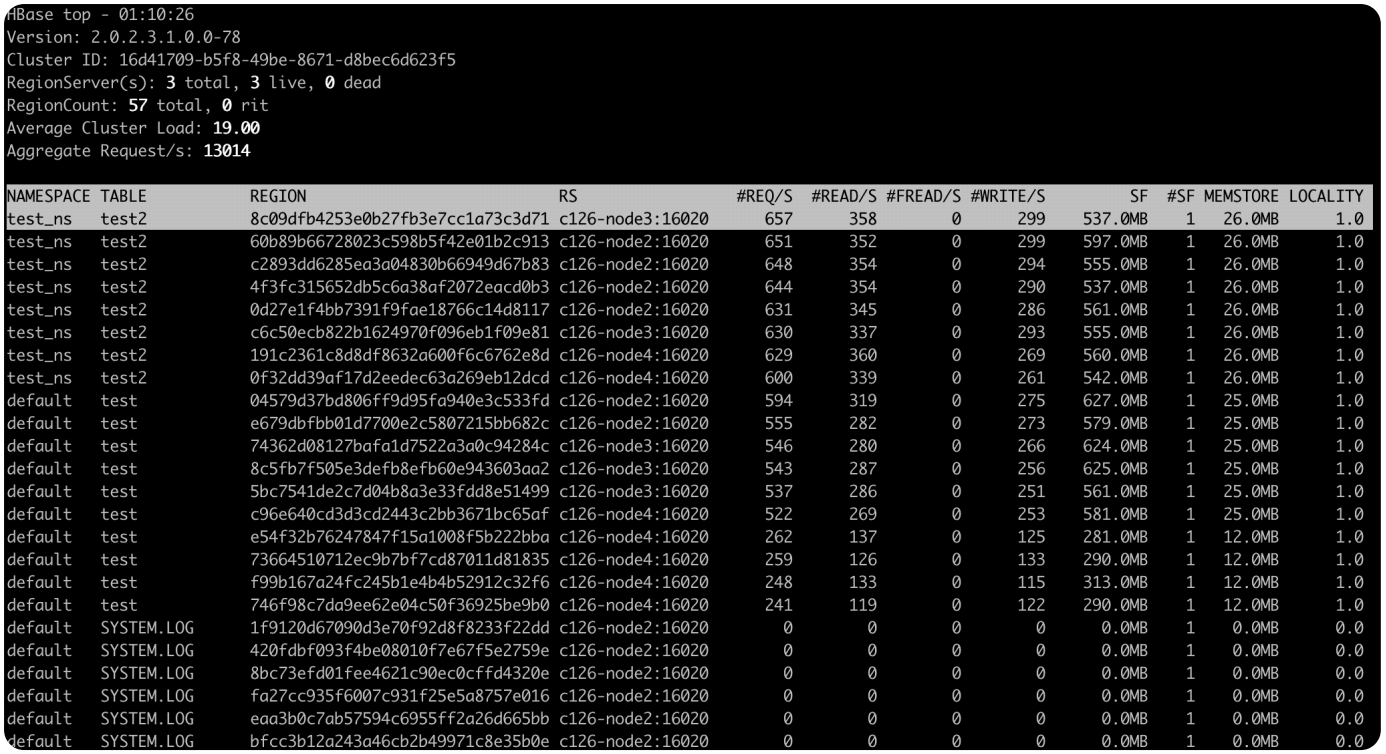
```
$ hbase hbtop -Dhbase.client.zookeeper.quorum=<zookeeper quorum> -Dzookeeper.znode.parent=<znode parent>
```



The top screen consists of a summary part and of a metrics part. In the summary part, you can see `HBase Version`, `Cluster ID`, `The number of region servers`, `Region count`, `Average Cluster Load` and `Aggregated Request/s`. In the metrics part, you can see metrics per Region/Namespace/Table/RegionServer depending on the selected mode. The top screen is refreshed in a certain period – 3 seconds by default.

Scrolling metric records

You can scroll the metric records in the metrics part.



Command line arguments

Argument	Description
-d,--delay `<arg>`	The refresh delay (in seconds); default is 3 seconds
-h,--help	Print usage; for help while the tool is running press `h` key
-m,--mode `<arg>`	The mode; `n` (Namespace)| `t` (Table)| `r` (Region)| `s` (RegionServer), default is `r`
-n,--numberOfIterations `<arg>`	The number of iterations
-O,--outputFieldNames	Print each of the available field names on a separate line, then quit
-f,--fields `<arg>`	Show only the given fields. Specify comma separated fields to show multiple fields

Argument	Description
-s,--sortField `<arg>`	The initial sort field. You can prepend a `+` or `-` to the field name to also override the sort direction. A leading `+` will force sorting high to low, whereas a `-` will ensure a low to high ordering
-i,--filters `<arg>`	The initial filters. Specify comma separated filters to set multiple filters
-b,--batchMode	Starts hbtop in Batch mode, which could be useful for sending output from hbtop to other programs or to a file. In this mode, hbtop will not accept input and runs until the iterations limit you've set with the `-n` command-line option or until killed

Modes

There are the following modes in hbtop:

Mode	Description
Region	Showing metric records per region
Namespace	Showing metric records per namespace
Table	Showing metric records per table
RegionServer	Showing metric records per region server
User	Showing metric records per user
Client	Showing metric records per client

Region mode

In Region mode, the default sort field is `#REQ/S`.

The fields in this mode are as follows:

Field	Description	Displayed by default
RNAME	Region Name	false
NAMESPACE	Namespace Name	true
TABLE	Table Name	true

Field	Description	Displayed by default
SCODE	Start Code	false
REPID	Replica ID	false
REGION	Encoded Region Name	true
RS	Short Region Server Name	true
LRS	Long Region Server Name	false
#REQ/S	Request Count per second	true
#READ/S	Read Request Count per second	true
#FREAD/S	Filtered Read Request Count per second	true
#WRITE/S	Write Request Count per second	true
SF	StoreFile Size	true
USF	Uncompressed StoreFile Size	false
#SF	Number of StoreFiles	true
MEMSTORE	MemStore Size	true
LOCALITY	Block Locality	true
SKEY	Start Key	false
#COMPingCELL	Compacting Cell Count	false
#COMPedCELL	Compacted Cell Count	false
%COMP	Compaction Progress	false
LASTMCOMP	Last Major Compaction Time	false

Namespace mode

In Namespace mode, the default sort field is `#REQ/S`.

The fields in this mode are as follows:

Field	Description	Displayed by default
NAMESPACE	Namespace Name	true

Field	Description	Displayed by default
#REGION	Region Count	true
#REQ/S	Request Count per second	true
#READ/S	Read Request Count per second	true
#FREAD/S	Filtered Read Request Count per second	true
#WRITE/S	Write Request Count per second	true
SF	StoreFile Size	true
USF	Uncompressed StoreFile Size	false
#SF	Number of StoreFiles	true
MEMSTORE	MemStore Size	true

Table mode

In Table mode, the default sort field is `#REQ/S`.

The fields in this mode are as follows:

Field	Description	Displayed by default
NAMESPACE	Namespace Name	true
TABLE	Table Name	true
#REGION	Region Count	true
#REQ/S	Request Count per second	true
#READ/S	Read Request Count per second	true
#FREAD/S	Filtered Read Request Count per second	true
#WRITE/S	Write Request Count per second	true
SF	StoreFile Size	true
USF	Uncompressed StoreFile Size	false

Field	Description	Displayed by default
#SF	Number of StoreFiles	true
MEMSTORE	MemStore Size	true

RegionServer mode

In RegionServer mode, the default sort field is `#REQ/S`.

The fields in this mode are as follows:

Field	Description	Displayed by default
RS	Short Region Server Name	true
LRS	Long Region Server Name	false
#REGION	Region Count	true
#REQ/S	Request Count per second	true
#READ/S	Read Request Count per second	true
#FREAD/S	Filtered Read Request Count per second	true
#WRITE/S	Write Request Count per second	true
SF	StoreFile Size	true
USF	Uncompressed StoreFile Size	false
#SF	Number of StoreFiles	true
MEMSTORE	MemStore Size	true
UHEAP	Used Heap Size	true
MHEAP	Max Heap Size	true

User mode

In User mode, the default sort field is `#REQ/S`.

The fields in this mode are as follows:

Field	Description	Displayed by default
USER	user Name	true
#CLIENT	Client Count	true
#REQ/S	Request Count per second	true
#READ/S	Read Request Count per second	true
#WRITE/S	Write Request Count per second	true
#FREAD/S	Filtered Read Request Count per second	true

Client mode

In Client mode, the default sort field is `#REQ/S`.

The fields in this mode are as follows:

Field	Description	Displayed by default
CLIENT	Client Hostname	true
#USER	User Count	true
#REQ/S	Request Count per second	true
#READ/S	Read Request Count per second	true
#WRITE/S	Write Request Count per second	true
#FREAD/S	Filtered Read Request Count per second	true

Changing mode

You can change mode by pressing `m` key in the top screen.

```
HBase top - 01:35:13
Version: 2.0.2.3.1.0.0-78
Cluster ID: 16d41709-b5f8-49be-8671-d8bec6d623f5
RegionServer(s): 3 total, 3 live, 0 dead
RegionCount: 57 total, 0 rit
Average Cluster Load: 19.00
Aggregate Request/s: 11995

NAMESPACE TABLE           REGION                           RS                  #REQ/S  #READ/S #FREAD/S #WRITE/S       SF  #SF MEMSTORE LOCALITY
test_ns   test2           8c09dfb4253e0b27fb3e7cc1a73c3d71 c126-node3:16020       532      285        0      247 572.0MB   1    4.0MB      1.0
default   test            e679dbfbb01d7700e2c5807215bb682c c126-node2:16020       527      297        0      230 616.0MB   1    4.0MB      1.0
default   test            04579d37bd806ff9d95fa940e3c533fd c126-node2:16020       519      297        0      222 665.0MB   1    3.0MB      1.0
default   test            c96e640cd3d3cd2443c2bb3671bc65af c126-node4:16020       516      299        0      217 618.0MB   1    4.0MB      1.0
test_ns   test2           c6c50ecb822b1624970f096eb1f09e81 c126-node3:16020       486      271        0      215 593.0MB   1    5.0MB      1.0
default   test            8c5fb7f505e3defb8efb60e943603aa2 c126-node3:16020       485      263        0      222 661.0MB   1    4.0MB      1.0
default   test            5bc7541de2c7d04b8a3e33fdd8e51499 c126-node3:16020       473      249        0      224 595.0MB   1    2.0MB      1.0
default   test            74362d08127bafa1d7522a3a0c94284c c126-node3:16020       472      252        0      220 662.0MB   1    4.0MB      1.0
test_ns   test2           191c2361c8d8df8632a600f6c6762e8d c126-node4:16020       451      196        0      255 601.0MB   1    3.0MB      1.0
test_ns   test2           60b89b66728023c598b5f42e01b2c913 c126-node2:16020       440      195        0      245 638.0MB   1    3.0MB      1.0
test_ns   test2           0f32dd39af17d2eedec63a269eb12dcd c126-node4:16020       437      191        0      246 578.0MB   1    3.0MB      1.0
test_ns   test2           0d27e1f4bb7391f9fae18766c14d8117 c126-node2:16020       430      186        0      244 600.0MB   1    5.0MB      1.0
test_ns   test2           c2893dd6285ea3a04830b66949d67b83 c126-node2:16020       427      179        0      248 594.0MB   1    5.0MB      1.0
test_ns   test2           4f3fc315652db5c6a38af2072eacd0b3 c126-node2:16020       410      179        0      231 571.0MB   1    5.0MB      1.0
default   test            73664510712ec9b7bf7cd87011d81835 c126-node4:16020       283      161        0      122 310.0MB   1    1.0MB      1.0
default   test            f99b167a24fc245b1e4b4b52912c32f6 c126-node4:16020       268      146        0      122 332.0MB   1    1.0MB      1.0
default   test            746f98c7da9ee62e04c50f36925be9b0 c126-node4:16020       268      157        0      111 309.0MB   1    2.0MB      1.0
default   test            e54f32b76247847f15a1008f5b222bba c126-node4:16020       262      146        0      116 297.0MB   1    2.0MB      1.0
default   SYSTEM.LOG      1f9120d67090d3e70f92d8f8233f22dd c126-node2:16020         0        0        0        0   0.0MB   1    0.0MB      0.0
default   SYSTEM.LOG      420fdbf093f4be08010f7e67f5e2759e c126-node2:16020         0        0        0        0   0.0MB   1    0.0MB      0.0
default   SYSTEM.LOG      8bc73efd01fee4621c90ec0cffd4320e c126-node2:16020         0        0        0        0   0.0MB   1    0.0MB      0.0
default   SYSTEM.LOG      fa27cc935f6007c931f25e5a8757e016 c126-node2:16020         0        0        0        0   0.0MB   1    0.0MB      0.0
default   SYSTEM.LOG      eaa3b0c7ab57594c6955ff2a26d665bb c126-node2:16020         0        0        0        0   0.0MB   1    0.0MB      0.0
default   SYSTEM.LOG      bfcc3b12a243a46cb2b49971c8e35b0e c126-node2:16020         0        0        0        0   0.0MB   1    0.0MB      0.0
```

Changing the refresh delay

You can change the refresh by pressing `d` key in the top screen.

```
HBase top - 01:43:19
Version: 2.0.2.3.1.0.0-78
Cluster ID: 16d41709-b5f8-49be-8671-d8bec6d623f5
RegionServer(s): 3 total, 3 live, 0 dead
RegionCount: 57 total, 0 rit
Average Cluster Load: 19.00
Aggregate Request/s: 11629

NAMESPACE TABLE           REGION                           RS                  #REQ/S  #READ/S #FREAD/S #WRITE/S       SF  #SF MEMSTORE LOCALITY
test_ns   test2           60b89b66728023c598b5f42e01b2c913 c126-node2:16020       559      298        0      261 638.0MB   1    8.0MB      1.0
test_ns   test2           0d27e1f4bb7391f9fae18766c14d8117 c126-node2:16020       548      284        0      264 600.0MB   1   10.0MB      1.0
test_ns   test2           c2893dd6285ea3a04830b66949d67b83 c126-node2:16020       537      293        0      244 594.0MB   1   10.0MB      1.0
test_ns   test2           4f3fc315652db5c6a38af2072eacd0b3 c126-node2:16020       528      274        0      254 571.0MB   1   10.0MB      1.0
test_ns   test2           8c09dfb4253e0b27fb3e7cc1a73c3d71 c126-node3:16020       524      285        0      239 572.0MB   1    9.0MB      1.0
test_ns   test2           c6c50ecb822b1624970f096eb1f09e81 c126-node3:16020       513      281        0      232 593.0MB   1   10.0MB      1.0
default   test            74362d08127bafa1d7522a3a0c94284c c126-node3:16020       506      271        0      235 662.0MB   1    8.0MB      1.0
test_ns   test2           191c2361c8d8df8632a600f6c6762e8d c126-node4:16020       506      271        0      235 601.0MB   1    8.0MB      1.0
test_ns   test2           0f32dd39af17d2eedec63a269eb12dcd c126-node4:16020       505      270        0      235 578.0MB   1    8.0MB      1.0
default   test            8c5fb7f505e3defb8efb60e943603aa2 c126-node3:16020       502      264        0      238 661.0MB   1    9.0MB      1.0
default   test            c96e640cd3d3cd2443c2bb3671bc65af c126-node4:16020       477      245        0      232 618.0MB   1    9.0MB      1.0
default   test            5bc7541de2c7d04b8a3e33fdd8e51499 c126-node3:16020       475      266        0      209 595.0MB   1    7.0MB      1.0
default   test            e679dbfbb01d7700e2c5807215bb682c c126-node2:16020       467      211        0      256 616.0MB   1    9.0MB      1.0
default   test            04579d37bd806ff9d95fa940e3c533fd c126-node2:16020       456      191        0      265 665.0MB   1    8.0MB      1.0
default   test            e54f32b76247847f15a1008f5b222bba c126-node4:16020       243      132        0      111 297.0MB   1    4.0MB      1.0
default   test            73664510712ec9b7bf7cd87011d81835 c126-node4:16020       241      133        0      108 310.0MB   1    4.0MB      1.0
default   test            f99b167a24fc245b1e4b4b52912c32f6 c126-node4:16020       238      121        0      117 332.0MB   1    3.0MB      1.0
default   test            746f98c7da9ee62e04c50f36925be9b0 c126-node4:16020       223      121        0      102 309.0MB   1    4.0MB      1.0
default   SYSTEM.LOG      1f9120d67090d3e70f92d8f8233f22dd c126-node2:16020         0        0        0        0   0.0MB   1    0.0MB      0.0
default   SYSTEM.LOG      420fdbf093f4be08010f7e67f5e2759e c126-node2:16020         0        0        0        0   0.0MB   1    0.0MB      0.0
default   SYSTEM.LOG      8bc73efd01fee4621c90ec0cffd4320e c126-node2:16020         0        0        0        0   0.0MB   1    0.0MB      0.0
default   SYSTEM.LOG      fa27cc935f6007c931f25e5a8757e016 c126-node2:16020         0        0        0        0   0.0MB   1    0.0MB      0.0
default   SYSTEM.LOG      eaa3b0c7ab57594c6955ff2a26d665bb c126-node2:16020         0        0        0        0   0.0MB   1    0.0MB      0.0
default   SYSTEM.LOG      bfcc3b12a243a46cb2b49971c8e35b0e c126-node2:16020         0        0        0        0   0.0MB   1    0.0MB      0.0
```

Changing the displayed fields

You can move to the field screen by pressing `f` key in the top screen. In the fields screen, you can change the displayed fields by choosing a field and pressing `d` key or `space` key.

```
HBase top - 00:40:57
Version: 2.0.2.3.1.0.0-78
Cluster ID: 16d41709-b5f8-49be-8671-d8bec6d623f5
RegionServer(s): 3 total, 3 live, 0 dead
RegionCount: 57 total, 0 rit
Average Cluster Load: 19.00
Aggregate Request/s: 12470

NAMESPACE TABLE           REGION                           RS                #REQ/S  #READ/S #FREAD/S #WRITE/S        SF  #SF MEMSTORE LOCALITY
default   test            5bc7541de2c7d04b8a3e33fdd8e51499 c126-node3:16020     514     259        0      255  631.0MB   1    5.0MB      1.0
default   test            e679dbfbb01d7700e2c5807215bb682c c126-node2:16020     506     257        0      249  650.0MB   1    5.0MB      1.0
test_ns   test2           4f3fc315652db5c6a38af2072eacd0b3 c126-node2:16020     506     262        0      244  619.0MB   1    6.0MB      1.0
default   test            04579d37bd806ff9d95fa940e3c533fd c126-node2:16020     505     271        0      234  698.0MB   1    5.0MB      1.0
test_ns   test2           60b89b66728023c598b5f42e01b2c913 c126-node2:16020     501     250        0      251  673.0MB   1    6.0MB      1.0
default   test            c96e640cd3d3cd2443c2bb3671bc65af c126-node4:16020     501     252        0      249  653.0MB   1    5.0MB      1.0
test_ns   test2           0d27e1f4bb7391f9fae18766c14d8117 c126-node2:16020     496     258        0      238  637.0MB   1    6.0MB      1.0
default   test            74362d08127bafa1d7522a3a0c94284c c126-node3:16020     493     241        0      252  538.0MB   1    5.0MB      1.0
default   test            8c5fb7f505e3defb8efb60e943603aa2 c126-node3:16020     481     245        0      236  696.0MB   1    5.0MB      1.0
test_ns   test2           8c09dfb4253e0b27fb3e7cc1a73c3d71 c126-node3:16020     477     245        0      232  619.0MB   1    6.0MB      1.0
test_ns   test2           c2893dd6285ea3a04830b66949d67b83 c126-node2:16020     470     243        0      227  630.0MB   1    6.0MB      1.0
test_ns   test2           c6c50ecb822b1624970f096eb1f09e81 c126-node3:16020     466     232        0      234  630.0MB   1    6.0MB      1.0
test_ns   test2           0f32dd39af17d2eedec63a269eb12dcd c126-node4:16020     459     237        0      222  624.0MB   1    6.0MB      1.0
test_ns   test2           191c2361c8d8df8632a600f6c6762e8d c126-node4:16020     434     211        0      223  635.0MB   1    6.0MB      1.0
default   test            e54f32b76247847f15a1008f5b222bba c126-node4:16020     265     133        0      132  316.0MB   1    2.0MB      1.0
default   test            746f98c7da9ee62e04c50f36925be9b0 c126-node4:16020     257     133        0      124  326.0MB   1    2.0MB      1.0
default   test            73664510712ec9b7bf7cd87011d81835 c126-node4:16020     253     125        0      128  326.0MB   1    2.0MB      1.0
default   test            f99b167a24fc245b1e4b4b52912c32f6 c126-node4:16020     251     123        0      128  349.0MB   1    2.0MB      1.0
default   SYSTEM.LOG      1f9120d67090d3e70f92d8f8233f22dd c126-node2:16020       0       0        0        0    0.0MB   1    0.0MB      0.0
default   SYSTEM.LOG      420fdbf093f4be08010f7e67f5e2759e c126-node2:16020       0       0        0        0    0.0MB   1    0.0MB      0.0
default   SYSTEM.LOG      8bc73efd01fee4621c90ec0cffd4320e c126-node2:16020       0       0        0        0    0.0MB   1    0.0MB      0.0
default   SYSTEM.LOG      fa27cc935f6007c931f25e5a8757e016 c126-node2:16020       0       0        0        0    0.0MB   1    0.0MB      0.0
default   SYSTEM.LOG      eaa3b0c7ab57594c6955ff2a26d665bb c126-node2:16020       0       0        0        0    0.0MB   1    0.0MB      0.0
default   SYSTEM.LOG      bfcc3b12a243a46cb2b49971c8e35b0e c126-node2:16020       0       0        0        0    0.0MB   1    0.0MB      0.0
```

Changing the sort field

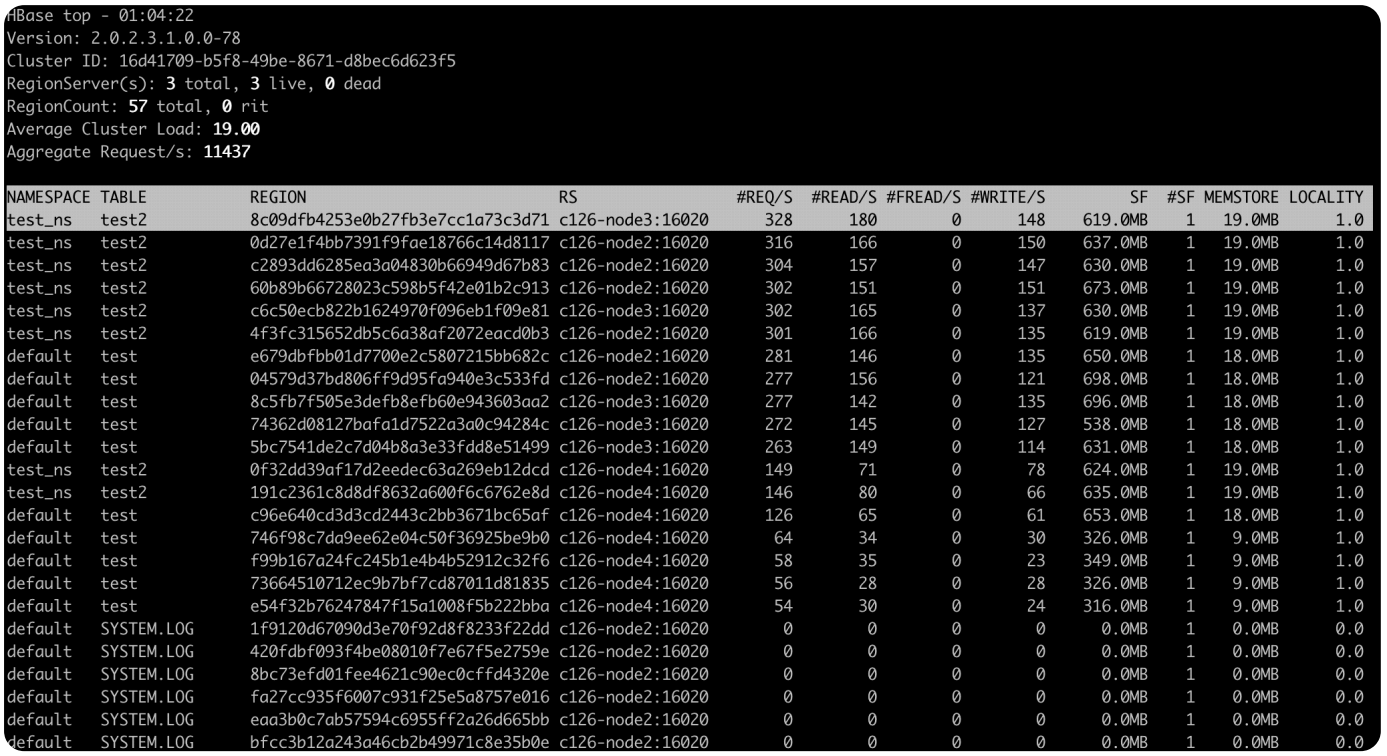
You can move to the fields screen by pressing `f` key in the top screen. In the field screen, you can change the sort field by choosing a field and pressing `s`. Also, you can change the sort order (ascending or descending) by pressing `R` key.

```
HBase top - 00:56:05
Version: 2.0.2.3.1.0.0-78
Cluster ID: 16d41709-b5f8-49be-8671-d8bec6d623f5
RegionServer(s): 3 total, 3 live, 0 dead
RegionCount: 57 total, 0 rit
Average Cluster Load: 19.00
Aggregate Request/s: 11604

NAMESPACE TABLE           REGION                           RS                #REQ/S  #READ/S #FREAD/S #WRITE/S        SF  #SF MEMSTORE LOCALITY
test_ns   test2           c2893dd6285ea3a04830b66949d67b83 c126-node2:16020     481     269        0      212  630.0MB   1   14.0MB      1.0
test_ns   test2           60b89b66728023c598b5f42e01b2c913 c126-node2:16020     460     239        0      221  673.0MB   1   14.0MB      1.0
test_ns   test2           0d27e1f4bb7391f9fae18766c14d8117 c126-node2:16020     462     247        0      215  637.0MB   1   14.0MB      1.0
test_ns   test2           4f3fc315652db5c6a38af2072eacd0b3 c126-node2:16020     474     257        0      217  619.0MB   1   14.0MB      1.0
test_ns   test2           8c09dfb4253e0b27fb3e7cc1a73c3d71 c126-node3:16020     434     221        0      213  619.0MB   1   14.0MB      1.0
test_ns   test2           c6c50ecb822b1624970f096eb1f09e81 c126-node3:16020     427     205        0      222  630.0MB   1   14.0MB      1.0
test_ns   test2           191c2361c8d8df8632a600f6c6762e8d c126-node4:16020     474     250        0      224  635.0MB   1   14.0MB      1.0
test_ns   test2           0f32dd39af17d2eedec63a269eb12dcd c126-node4:16020     460     246        0      214  624.0MB   1   14.0MB      1.0
default   test            04579d37bd806ff9d95fa940e3c533fd c126-node2:16020     515     253        0      262  698.0MB   1   14.0MB      1.0
default   test            e679dbfbb01d7700e2c5807215bb682c c126-node2:16020     502     247        0      255  650.0MB   1   14.0MB      1.0
default   test            74362d08127bafa1d7522a3a0c94284c c126-node3:16020     473     219        0      254  538.0MB   1   14.0MB      1.0
default   test            5bc7541de2c7d04b8a3e33fdd8e51499 c126-node3:16020     489     233        0      256  631.0MB   1   14.0MB      1.0
default   test            8c5fb7f505e3defb8efb60e943603aa2 c126-node3:16020     488     235        0      253  696.0MB   1   14.0MB      1.0
default   test            73664510712ec9b7bf7cd87011d81835 c126-node4:16020     240     111        0      129  326.0MB   1    7.0MB      1.0
default   test            f99b167a24fc245b1e4b4b52912c32f6 c126-node4:16020     225     105        0      120  349.0MB   1    7.0MB      1.0
default   test            e54f32b76247847f15a1008f5b222bba c126-node4:16020     237     127        0      110  316.0MB   1    7.0MB      1.0
default   test            746f98c7da9ee62e04c50f36925be9b0 c126-node4:16020     228     114        0      114  326.0MB   1    7.0MB      1.0
default   test            c96e640cd3d3cd2443c2bb3671bc65af c126-node4:16020     465     221        0      244  653.0MB   1   14.0MB      1.0
hbase     namespace       f648887eb345405fd03668a7559f4a09 c126-node4:16020       0       0        0        0    0.0MB   1    0.0MB      1.0
hbase     meta            1588230740                       c126-node3:16020       0       0        0        0    0.0MB   2    0.0MB      1.0
default   SYSTEM.STATS    03f0f895cd572fedb97eaec4a6050d78 c126-node3:16020       0       0        0        0    0.0MB   1    0.0MB      0.0
default   SYSTEM.SEQUENCE 42be37839fea614f95dded5d082a2f3f c126-node3:16020       0       0        0        0    0.0MB   1    0.0MB      0.0
default   SYSTEM.MUTEX    d16d5989d9b0d11a6cff71a25f2280cf c126-node2:16020       0       0        0        0    0.0MB   1    0.0MB      1.0
default   SYSTEM.LOG      1f9120d67090d3e70f92d8f8233f22dd c126-node2:16020       0       0        0        0    0.0MB   1    0.0MB      0.0
```

Changing the order of the fields

You can move to the fields screen by pressing `f` key in the top screen. In the field screen, you can change the order of the fields.



Filters

You can filter the metric records with the filter feature. We can add filters by pressing `o` key for ignoring case or `O` key for case sensitive.

```
HBase top - 01:35:53
Version: 2.0.2.3.1.0.0-78
Cluster ID: 16d41709-b5f8-49be-8671-d8bec6d623f5
RegionServer(s): 3 total, 3 live, 0 dead
RegionCount: 57 total, 0 rit
Average Cluster Load: 19.00
Aggregate Request/s: 12428

NAMESPACE TABLE          REGION                           RS                  #REQ/S  #READ/S #FREAD/S #WRITE/S        SF  #SF MEMSTORE LOCALITY
test_ns   test2          4f3fc315652db5c6a38af2072eacd0b3 c126-node2:16020      574     299        0      275   639.0MB   1    2.0MB      1.0
test_ns   test2          c2893dd6285ea3a04830b66949d67b83 c126-node2:16020      560     299        0      261   508.0MB   1    2.0MB      1.0
test_ns   test2          60b89b66728023c598b5f42e01b2c913 c126-node2:16020      553     285        0      268   709.0MB   1    1.0MB      1.0
test_ns   test2          8c09dfb4253e0b27fb3e7cc1a73c3d71 c126-node3:16020      552     288        0      264   639.0MB   1    2.0MB      1.0
test_ns   test2          0d27e1f4bb7391f9fae18766c14d8117 c126-node2:16020      549     285        0      264   674.0MB   1    1.0MB      1.0
test_ns   test2          c6c50ecb822b1624970f096eb1f09e81 c126-node3:16020      547     282        0      265   668.0MB   1    0.0MB      1.0
test_ns   test2          191c2361c8d8df8632a600f6c6762e8d c126-node4:16020      545     281        0      264   672.0MB   1    1.0MB      1.0
test_ns   test2          0f32dd39af17d2eedec63a269eb12dcd c126-node4:16020      525     266        0      259   624.0MB   1   35.0MB      1.0
default   test           c96e640cd3d3cd2443c2bb3671bc65af c126-node4:16020      523     267        0      256   689.0MB   1    0.0MB      1.0
default   test           04579d37bd806ff9d95fa940e3c533fd c126-node2:16020      509     266        0      243   735.0MB   1    0.0MB      1.0
default   test           8c5fb7f505e3defb8efb60e943603aa2 c126-node3:16020      506     260        0      246   732.0MB   1    1.0MB      1.0
default   test           5bc7541de2c7d04b8a3e33fdd8e51499 c126-node3:16020      496     254        0      242   665.0MB   1    2.0MB      1.0
default   test           74362d08127bafa1d7522a3a0c94284c c126-node3:16020      494     245        0      249   573.0MB   1    1.0MB      1.0
default   test           e679dbfbb01d7700e2c5807215bb682c c126-node2:16020      489     241        0      248   650.0MB   1   33.0MB      1.0
default   test           f99b167a24fc245b1e4b4b52912c32f6 c126-node4:16020      269     130        0      139   360.0MB   1    0.0MB      1.0
default   test           73664510712ec9b7bf7cd87011d81835 c126-node4:16020      266     137        0      129   338.0MB   1    0.0MB      1.0
default   test           746f98c7da9ee62e04c50f36925be9b0 c126-node4:16020      263     137        0      126   338.0MB   1    0.0MB      1.0
default   test           e54f32b76247847f15a1008f5b222bba c126-node4:16020      256     137        0      119   334.0MB   1    0.0MB      1.0
default   SYSTEM.LOG     1f9120d67090d3e70f92d8f8233f22dd c126-node2:16020        0       0        0        0     0.0MB   1    0.0MB      0.0
default   SYSTEM.LOG     420fdbf093f4be08010f7e67f5e2759e c126-node2:16020        0       0        0        0     0.0MB   1    0.0MB      0.0
default   SYSTEM.LOG     8bc73efd01fee4621c90ec0cffd4320e c126-node2:16020        0       0        0        0     0.0MB   1    0.0MB      0.0
default   SYSTEM.LOG     fa27cc935f6007c931f25e5a8757e016 c126-node2:16020        0       0        0        0     0.0MB   1    0.0MB      0.0
default   SYSTEM.LOG     eaa3b0c7ab57594c6955ff2a26d665bb c126-node2:16020        0       0        0        0     0.0MB   1    0.0MB      0.0
default   SYSTEM.LOG     bfcc3b12a243a46cb2b49971c8e35b0e c126-node2:16020        0       0        0        0     0.0MB   1    0.0MB      0.0
```

The syntax is as follows:

```
<Field><Operator><Value>
```

For example, we can add filters like the following:

```
NAMESPACE==default
REQ/S>1000
```

The operators we can specify are as follows:

Operator	Description
=	Partial match
==	Exact match
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than and equal to

You can see the current filters by pressing `^o` key and clear them by pressing `=` key.

```
HBase top - 01:43:07
Version: 2.0.2.3.1.0.0-78
Cluster ID: 16d41709-b5f8-49be-8671-d8bec6d623f5
RegionServer(s): 3 total, 3 live, 0 dead
RegionCount: 57 total, 0 rit
Average Cluster Load: 19.00
Aggregate Request/s: 11724

NAMESPACE TABLE           REGION                              RS                #REQ/S  #READ/S #FREAD/S #WRITE/S        SF  #SF MEMSTORE LOCALITY
default   test            5bc7541de2c7d04b8a3e33fdd8e51499 c126-node3:16020      485      253        0      232   665.0MB   1    6.0MB      1.0
default   test            8c5fb7f505e3defb8efb60e943603aa2 c126-node3:16020      463      241        0      222   732.0MB   1    5.0MB      1.0
default   test            74362d08127bafa1d7522a3a0c94284c c126-node3:16020      453      233        0      220   573.0MB   1    5.0MB      1.0
default   test            c96e640cd3d3cd2443c2bb3671bc65af c126-node4:16020      452      217        0      235   689.0MB   1    4.0MB      1.0
default   test            04579d37bd806ff9d95fa940e3c533fd c126-node2:16020      426      200        0      226   735.0MB   1    4.0MB      1.0
default   test            e679dbfbb01d7700e2c5807215bb682c c126-node2:16020      422      205        0      217   687.0MB   1    4.0MB      1.0
```

Drilling down

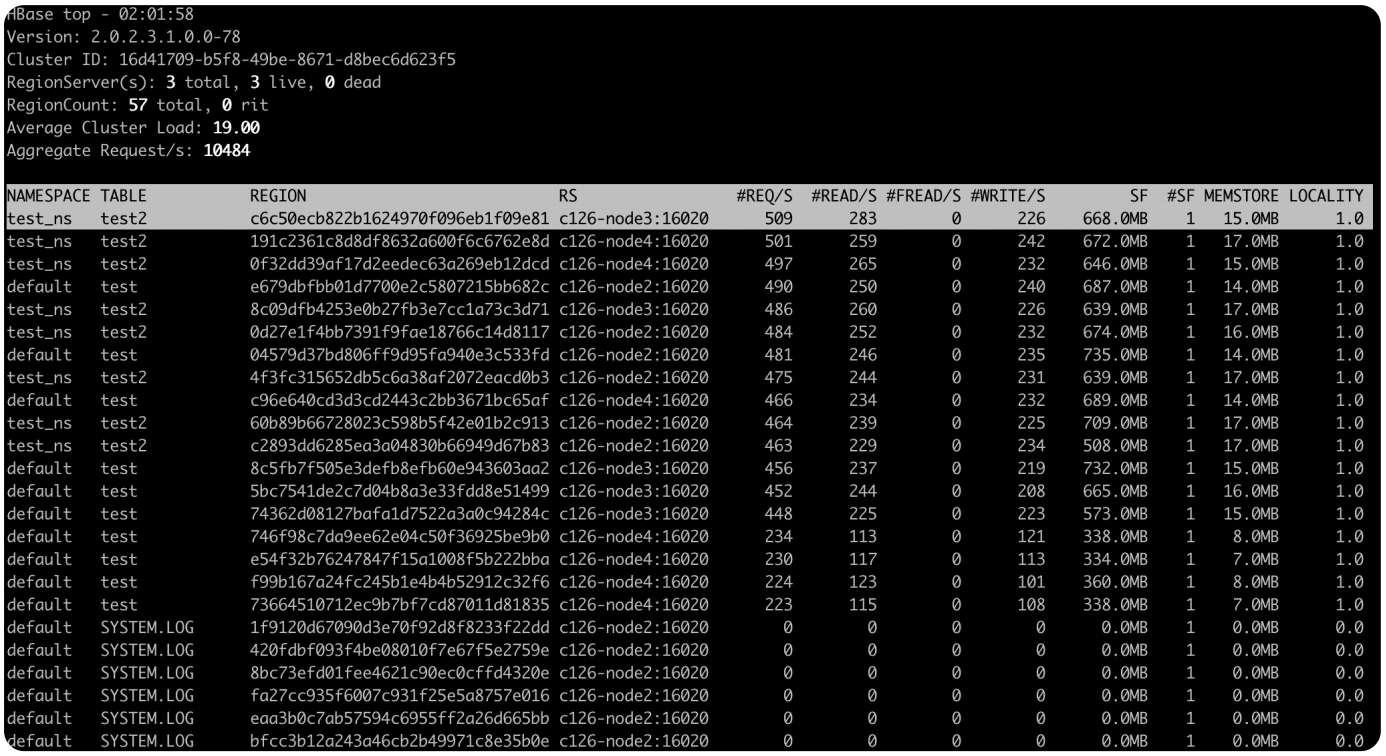
You can drill down the metric record by choosing a metric record that you want to drill down and pressing `i` key in the top screen. With this feature, you can find hot regions easily in a top-down manner.

```
HBase top - 01:53:13
Version: 2.0.2.3.1.0.0-78
Cluster ID: 16d41709-b5f8-49be-8671-d8bec6d623f5
RegionServer(s): 3 total, 3 live, 0 dead
RegionCount: 57 total, 0 rit
Average Cluster Load: 19.00
Aggregate Request/s: 11838

NAMESPACE TABLE           REGION                              RS                #REQ/S  #READ/S #FREAD/S #WRITE/S        SF  #SF MEMSTORE LOCALITY
test_ns   test2           c6c50ecb822b1624970f096eb1f09e81 c126-node3:16020      539      272        0      267   668.0MB   1   10.0MB      1.0
test_ns   test2           0d27e1f4bb7391f9fae18766c14d8117 c126-node2:16020      536      267        0      269   674.0MB   1   11.0MB      1.0
test_ns   test2           c2893dd6285ea3a04830b66949d67b83 c126-node2:16020      530      258        0      272   508.0MB   1   12.0MB      1.0
test_ns   test2           8c09dfb4253e0b27fb3e7cc1a73c3d71 c126-node3:16020      527      273        0      254   639.0MB   1   12.0MB      1.0
test_ns   test2           60b89b66728023c598b5f42e01b2c913 c126-node2:16020      522      265        0      257   709.0MB   1   12.0MB      1.0
test_ns   test2           4f3fc315652db5c6a38af2072eacd0b3 c126-node2:16020      510      257        0      253   639.0MB   1   12.0MB      1.0
default   test            5bc7541de2c7d04b8a3e33fdd8e51499 c126-node3:16020      508      247        0      261   665.0MB   1   11.0MB      1.0
default   test            04579d37bd806ff9d95fa940e3c533fd c126-node2:16020      500      252        0      248   735.0MB   1   10.0MB      1.0
default   test            8c5fb7f505e3defb8efb60e943603aa2 c126-node3:16020      498      244        0      254   732.0MB   1   10.0MB      1.0
default   test            74362d08127bafa1d7522a3a0c94284c c126-node3:16020      496      243        0      253   573.0MB   1   11.0MB      1.0
default   test            e679dbfbb01d7700e2c5807215bb682c c126-node2:16020      481      238        0      243   687.0MB   1    9.0MB      1.0
test_ns   test2           0f32dd39af17d2eedec63a269eb12dcd c126-node4:16020      430      223        0      207   646.0MB   1   10.0MB      1.0
test_ns   test2           191c2361c8d8df8632a600f6c6762e8d c126-node4:16020      425      221        0      204   672.0MB   1   12.0MB      1.0
default   test            c96e640cd3d3cd2443c2bb3671bc65af c126-node4:16020      407      224        0      183   689.0MB   1   10.0MB      1.0
default   test            e54f32b76247847f15a1008f5b222bba c126-node4:16020      204      102        0      102   334.0MB   1    5.0MB      1.0
default   test            f99b167a24fc245b1e4b4b52912c32f6 c126-node4:16020      202      109        0       93   360.0MB   1    5.0MB      1.0
default   test            73664510712ec9b7bf7cd87011d81835 c126-node4:16020      196      102        0       94   338.0MB   1    5.0MB      1.0
default   test            746f98c7da9ee62e04c50f36925be9b0 c126-node4:16020      194      103        0       91   338.0MB   1    5.0MB      1.0
default   SYSTEM.LOG      1f9120d67090d3e70f92d8f8233f22dd c126-node2:16020        0        0        0        0     0.0MB   1    0.0MB      0.0
default   SYSTEM.LOG      420fdbf093f4be08010f7e67f5e2759e c126-node2:16020        0        0        0        0     0.0MB   1    0.0MB      0.0
default   SYSTEM.LOG      8bc73efd01fee4621c90ec0cffd4320e c126-node2:16020        0        0        0        0     0.0MB   1    0.0MB      0.0
default   SYSTEM.LOG      fa27cc935f6007c931f25e5a8757e016 c126-node2:16020        0        0        0        0     0.0MB   1    0.0MB      0.0
default   SYSTEM.LOG      eaa3b0c7ab57594c6955ff2a26d665bb c126-node2:16020        0        0        0        0     0.0MB   1    0.0MB      0.0
default   SYSTEM.LOG      bfcc3b12a243a46cb2b49971c8e35b0e c126-node2:16020        0        0        0        0     0.0MB   1    0.0MB      0.0
```

Help screen

You can see the help screen by pressing `h` key in the top screen.



```
HBase top - 02:01:58
Version: 2.0.2.3.1.0.0-78
Cluster ID: 16d41709-b5f8-49be-8671-d8bec6d623f5
RegionServer(s): 3 total, 3 live, 0 dead
RegionCount: 57 total, 0 rit
Average Cluster Load: 19.00
Aggregate Request/s: 10484

NAMESPACE TABLE          REGION                          RS                #REQ/S  #READ/S #FREAD/S #WRITE/S       SF  #SF MEMSTORE LOCALITY
test_ns   test2          c6c50ecb822b1624970f096eb1f09e81 c126-node3:16020     509      283        0      226  668.0MB   1   15.0MB      1.0
test_ns   test2          191c2361c8d8df8632a600f6c6762e8d c126-node4:16020     501      259        0      242  672.0MB   1   17.0MB      1.0
test_ns   test2          0f32dd39af17d2eedec63a269eb12dcd c126-node4:16020     497      265        0      232  646.0MB   1   15.0MB      1.0
default   test           e679dbfbb01d7700e2c5807215bb682c c126-node2:16020     490      250        0      240  687.0MB   1   14.0MB      1.0
test_ns   test2          8c09dfb4253e0b27fb3e7cc1a73c3d71 c126-node3:16020     486      260        0      226  639.0MB   1   17.0MB      1.0
test_ns   test2          0d27e1f4bb7391f9fae18766c14d8117 c126-node2:16020     484      252        0      232  674.0MB   1   16.0MB      1.0
default   test           04579d37bd806ff9d95fa940e3c533fd c126-node2:16020     481      246        0      235  735.0MB   1   14.0MB      1.0
test_ns   test2          4f3fc315652db5c6a38af2072eacd0b3 c126-node2:16020     475      244        0      231  639.0MB   1   17.0MB      1.0
default   test           c96e640cd3d3cd2443c2bb3671bc65af c126-node4:16020     466      234        0      232  689.0MB   1   14.0MB      1.0
test_ns   test2          60b89b66728023c598b5f42e01b2c913 c126-node2:16020     464      239        0      225  709.0MB   1   17.0MB      1.0
test_ns   test2          c2893dd6285ea3a04830b66949d67b83 c126-node2:16020     463      229        0      234  508.0MB   1   17.0MB      1.0
default   test           8c5fb7f505e3defb8efb60e943603aa2 c126-node3:16020     456      237        0      219  732.0MB   1   15.0MB      1.0
default   test           5bc7541de2c7d04b8a3e33fdd8e51499 c126-node3:16020     452      244        0      208  665.0MB   1   16.0MB      1.0
default   test           74362d08127bafa1d7522a3a0c94284c c126-node3:16020     448      225        0      223  573.0MB   1   15.0MB      1.0
default   test           746f98c7da9ee62e04c50f36925be9b0 c126-node4:16020     234      113        0      121  338.0MB   1    8.0MB      1.0
default   test           e54f32b76247847f15a1008f5b222bba c126-node4:16020     230      117        0      113  334.0MB   1    7.0MB      1.0
default   test           f99b167a24fc245b1e4b4b52912c32f6 c126-node4:16020     224      123        0      101  360.0MB   1    8.0MB      1.0
default   test           73664510712ec9b7bf7cd87011d81835 c126-node4:16020     223      115        0      108  338.0MB   1    7.0MB      1.0
default   SYSTEM.LOG     1f9120d67090d3e70f92d8f8233f22dd c126-node2:16020       0        0        0        0    0.0MB   1    0.0MB      0.0
default   SYSTEM.LOG     420fdbf093f4be08010f7e67f5e2759e c126-node2:16020       0        0        0        0    0.0MB   1    0.0MB      0.0
default   SYSTEM.LOG     8bc73efd01fee4621c90ec0cffd4320e c126-node2:16020       0        0        0        0    0.0MB   1    0.0MB      0.0
default   SYSTEM.LOG     fa27cc935f6007c931f25e5a8757e016 c126-node2:16020       0        0        0        0    0.0MB   1    0.0MB      0.0
default   SYSTEM.LOG     eaa3b0c7ab57594c6955ff2a26d665bb c126-node2:16020       0        0        0        0    0.0MB   1    0.0MB      0.0
default   SYSTEM.LOG     bfcc3b12a243a46cb2b49971c8e35b0e c126-node2:16020       0        0        0        0    0.0MB   1    0.0MB      0.0
```

Others

How hbtop gets the metrics data

hbtop gets the metrics from ClusterMetrics which is returned as the result of a call to Admin#getClusterMetrics() on the current HMaster. To add metrics to hbtop, they will need to be exposed via ClusterMetrics.

Tracing

Overview

The basic support for tracing has been done, where we added tracing for async client, rpc, region read/write/scan operation, and WAL. We use opentelemetry-api to implement the tracing support manually by code, as our code base is way too complicated to be instrumented through a java agent. But notice that you still need to attach the opentelemetry java agent to enable tracing. Please see the official site for OpenTelemetry and the documentation for opentelemetry-java-instrumentation for more details on how to properly configure opentelemetry instrumentation.

Usage

Enable Tracing

See this section in hbase-env.sh

```
# Uncomment to enable trace, you can change the options to use other exporters su
ch as jaeger or
# zipkin. See https://github.com/open-telemetry/opentelemetry-java-instrumentatio
n on how to
# configure exporters and other components through system properties.
# export HBASE_TRACE_OPTS="-Dotel.resource.attributes=service.name=HBase -Dotel.t
races.exporter=logging otel.metrics.exporter=none"
```

Uncomment this line to enable tracing. The default config is to output the tracing data to log. Please see the documentation for opentelemetry-java-instrumentation for more details on how to export tracing data to other tracing system such as OTel collector, jaeger or zipkin, what does the *service.name* mean, and how to change the sampling rate, etc.

> The LoggingSpanExporter uses java.util.logging(jul) for logging tracing data, and the logger is initialized in opentelemetry java agent, which seems to be ahead of our jul to slf4j bridge initialization, so it will always log the tracing data to console. We highly suggest that you use other tracing systems to collect and view tracing data instead of logging.

Performance Impact

According to the result in HBASE-25658, the performance impact is minimal. Of course the test cluster is not under heavy load, so if you find out that enabling tracing would impact the performance, try to lower the sampling rate. See documentation for configuring sampler for more details.

Store File Tracking

Overview

Historically, HBase internals have relied on creating hfiles on temporary directories first, renaming those files to the actual store directory at operation commit time. That's a simple and convenient way to separate transient from already finalised files that are ready to serve client reads with data. This approach works well with strong consistent file systems, but with the popularity of less consistent file systems, mainly Object Store which can be used like file systems, dependency on atomic rename operations starts to introduce performance penalties. The Amazon S3 Object Store, in particular, has been the most affected deployment, due to its lack of atomic renames. The HBase community temporarily bypassed this problem by building a distributed locking layer called HBOSS, to guarantee atomicity of operations against S3.

With **Store File Tracking**, decision on where to originally create new hfiles and how to proceed upon commit is delegated to the specific Store File Tracking implementation. The implementation can be set at the HBase service leve in **hbase-site.xml** or at the Table or Column Family via the TableDescriptor configuration.

> When the store file tracking implementation is specified in **hbase_site.xml**, this configuration is also propagated into a tables configuration at table creation time. This is to avoid dangerous configuration mismatches between processes, which could potentially lead to data loss.

Available Implementations

Store File Tracking initial version provides three builtin implementations:

- DEFAULT
- FILE
- MIGRATION

DEFAULT

As per the name, this is the Store File Tracking implementation used by default when no explicit configuration has been defined. The DEFAULT tracker implements the standard

approach using temporary directories and renames. This is how all previous (implicit) implementation that HBase used to track store files.

FILE

A file tracker implementation that creates new files straight in the store directory, avoiding the need for rename operations. It keeps a list of committed hfiles in memory, backed by meta files, in each store directory. Whenever a new hfile is committed, the list of *tracked files* in the given store is updated and a new meta file is written with this list contents, discarding the previous meta file now containing an out dated list.

MIGRATION

A special implementation to be used when swapping between Store File Tracking implementations on pre-existing tables that already contain data, and therefore, files being tracked under an specific logic.

Usage

For fresh deployments that don't yet contain any user data, **FILE** implementation can be just set as value for **hbase.store.file-tracker.impl** property in global **hbase-site.xml** configuration, prior to the first hbase start. Omitting this property sets the **DEFAULT** implementation.

For clusters with data that are upgraded to a version of HBase containing the store file tracking feature, the Store File Tracking implementation can only be changed with the **MIGRATION** implementation, so that the *new tracker* can safely build its list of tracked files based on the list of the *current tracker*.

> MIGRATION tracker should NOT be set at global configuration. To use it, follow below section about setting Store File Tacking at Table or Column Family configuration.

Configuring for Table or Column Family

Setting Store File Tracking configuration globally may not always be possible or desired, for example, in the case of upgraded clusters with pre-existing user data. Store File Tracking can be set at Table or Column Family level configuration. For example, to specify

FILE implementation in the table configuration at table creation time, the following should be applied:

```
create 'my-table', 'f1', 'f2', {CONFIGURATION => {'hbase.store.file-tracker.impl'
=> 'FILE'}}
```

To define **FILE** for an specific Column Family:

```
create 'my-table', {NAME=> '1', CONFIGURATION => {'hbase.store.file-tracker.impl'
=> 'FILE'}}
```

Switching trackers at Table or Column Family

A very common scenario is to set Store File Tracking on pre-existing HBase deployments that have been upgraded to a version that supports this feature. To apply the FILE tracker, tables effectively need to be migrated from the DEFAULT tracker to the FILE tracker. As explained previously, such process requires the usage of the special MIGRATION tracker implementation, which can only be specified at table or Column Family level.

For example, to switch *tracker* from **DEFAULT** to **FILE** in a table configuration:

```
alter 'my-table', CONFIGURATION => {'hbase.store.file-tracker.impl' => 'MIGRATIO
N',
'hbase.store.file-tracker.migration.src.impl' => 'DEFAULT',
'hbase.store.file-tracker.migration.dst.impl' => 'FILE'}
```

To apply similar switch at column family level configuration:

```
alter 'my-table', {NAME => 'f1', CONFIGURATION => {'hbase.store.file-tracker.imp
l' => 'MIGRATION',
'hbase.store.file-tracker.migration.src.impl' => 'DEFAULT',
'hbase.store.file-tracker.migration.dst.impl' => 'FILE'}}
```

Once all table regions have been onlined again, don't forget to disable MIGRATION, by now setting **hbase.store.file-tracker.migration.dst.impl** value as the **hbase.store.file-tracker.impl**. In the above example, that would be as follows:

```
alter 'my-table', CONFIGURATION => {'hbase.store.file-tracker.impl' => 'FILE'}
```

Specifying trackers during snapshot recovery

It's also possible to specify a given store file tracking implementation when recovering a snapshot using the *CLONE_SFT* option of *clone_snasphot* command. This is useful when recovering old snapshots, taken prior to a change in the global configuration, or if the snapshot has been imported from a different cluster that had a different store file tracking setting. Because snapshots preserve table and colum family descriptors, a simple restore would reload the original configuration, requiring the additional steps described above to convert the table/column family to the desired tracker implementation. An example of how to use *clone_snapshot* to specify the **FILE** tracker implementation is shown below:

```
clone_snapshot 'snapshotName', 'namespace:tableName', {CLONE_SFT=>'FILE'}
```

> The option to specify the tracker during snapshot recovery is only available for the *clone_snapshot* command. The *restore_snapshot* command does not support this parameter.

Bulk Data Generator Tool

Usage

```
usage: hbase org.apache.hadoop.hbase.util.bulkdatagenerator.BulkDataGeneratorTool
<OPTIONS> [-D<property=value>]*
 -d,--delete-if-exist         If it's set, the table will be deleted if already e
xist.
 -h,--help                    Show help message for the tool
 -mc,--mapper-count <arg>     The number of mapper containers to be launched.
 -o,--table-options <arg>     Table options to be set while creating the table.
 -r,--rows-per-mapper <arg>   The number of rows to be generated PER mapper.
 -sc,--split-count <arg>      The number of regions/pre-splits to be created for
the table.
 -t,--table <arg>             The table name for which data need to be generated.
```

Examples:

```
hbase org.apache.hadoop.hbase.util.bulkdatagenerator.BulkDataGeneratorTool -t TES
T_TABLE -mc 10 -r 100 -sc 10

hbase org.apache.hadoop.hbase.util.bulkdatagenerator.BulkDataGeneratorTool -t TES
T_TABLE -mc 10 -r 100 -sc 10 -d -o "BACKUP=false,NORMALIZATION_ENABLED=false"

hbase org.apache.hadoop.hbase.util.bulkdatagenerator.BulkDataGeneratorTool -t TES
T_TABLE -mc 10 -r 100 -sc 10 -Dmapreduce.map.memory.mb=8192
```

Overview

Table Schema

Tool generates a HBase table with single column family, i.e. **cf** and 9 columns i.e.

```
ORG_ID, TOOL_EVENT_ID, EVENT_ID, VEHICLE_ID, SPEED, LATITUDE, LONGITUDE, LOCATIO
N, TIMESTAMP
```

with row key as

```
<TOOL_EVENT_ID>:<ORGANIZATION_ID>
```

Table Creation

Tool creates a pre-splited HBase Table having "**split-count**" splits (i.e. **split-count** + 1 regions) with sequential 6 digit region boundary prefix. Example: If a table is generated with "**split-count**" as 10, it will have (10+1) regions with following start-end keys.

```
(-000001, 000001-000002, 000002-000003, ...., 000009-000010, 0000010-)
```

Data Generation

Tool creates and run a MR job to generate the HFiles, which are bulk loaded to table regions via `org.apache.hadoop.hbase.tool.BulkLoadHFilesTool`. The number of mappers is defined in input as "**mapper-count**". Each mapper generates "**records-per-mapper**" rows.

`org.apache.hadoop.hbase.util.bulkdatageneratorBulkDataGeneratorRecordReader` ensures that each record generated by mapper is associated with index (added to the key) ranging from 1 to "**records-per-mapper**".

The TOOL_EVENT_ID column for each row has a 6 digit prefix as

```
(index) mod ("split-count" + 1)
```

Example, if 10 records are to be generated by each mapper and "**split-count**" is 4, the TOOL_EVENT_IDs for each record will have a prefix as

Record Index	TOOL_EVENT_ID's first six characters
1	000001
2	000002
3	000003
4	000004
5	000000
6	000001
7	000002
8	000003

Record Index	TOOL_EVENT_ID's first six characters
9	000004
10	000005

Since TOOL_EVENT_ID is first attribute of row key and table region boundaries are also having start-end keys as 6 digit sequential prefixes, this ensures that each mapper generates (nearly) same number of rows for each region, making the data uniformly distributed. TOOL_EVENT_ID suffix and other columns of the row are populated with random data.

Number of rows generated is

```
rows-per-mapper * mapper-count
```

Size of each rows is (approximately)

```
850 B
```

Experiments

These results are from a 11 node cluster having HBase and Hadoop service running within self-managed test environment

Data Size	Time to Generate Data (mins)
100 GB	6 minutes
340 GB	13 minutes
3.5 TB	3 hours 10 minutes

Appendix: Contributing to Documentation

Contributing to Documentation

The Apache HBase project welcomes contributions to all aspects of the project, including the documentation.

In HBase, documentation includes the following areas, and probably some others:

- The HBase Reference Guide (this book)
- The HBase website
- API documentation
- Command-line utility output and help text
- Web UI strings, explicit help text, context-sensitive strings, and others
- Log messages
- Comments in source files, configuration files, and others
- Localization of any of the above into target languages other than English

No matter which area you want to help out with, the first step is almost always to download (typically by cloning the Git repository) and familiarize yourself with the HBase source code. For information on downloading and building the source, see developer.

Contributing to Other Strings

If you spot an error in a string in a UI, utility, script, log message, or elsewhere, or you think something could be made more clear, or you think text needs to be added where it doesn't currently exist, the first step is to file a JIRA. Be sure to set the component to `Documentation` in addition to any other involved components. Most components have one or more default owners, who monitor new issues which come into those queues. Regardless of whether you feel able to fix the bug, you should still file bugs where you see them.

If you want to try your hand at fixing your newly-filed bug, assign it to yourself. You will need to clone the HBase Git repository to your local system and work on the issue there. When you have developed a potential fix, submit it for review. If it addresses the issue and

is seen as an improvement, one of the HBase committers will commit it to one or more branches, as appropriate.

Procedure: Suggested Work flow for Submitting Patches

This procedure goes into more detail than Git pros will need, but is included in this appendix so that people unfamiliar with Git can feel confident contributing to HBase while they learn.

1. If you have not already done so, clone the Git repository locally. You only need to do this once.
2. Fairly often, pull remote changes into your local repository by using the `git pull` command, while your tracking branch is checked out.
3. For each issue you work on, create a new branch. One convention that works well for naming the branches is to name a given branch the same as the JIRA it relates to:

```
$ git checkout -b HBASE-123456
```

4. Make your suggested changes on your branch, committing your changes to your local repository often. If you need to switch to working on a different issue, remember to check out the appropriate branch.
5. When you are ready to submit your patch, first be sure that HBase builds cleanly and behaves as expected in your modified branch.
6. If you have made documentation or website changes, verify that the site builds correctly by running the development server from the `hbase-website/` directory.
7. If it takes you several days or weeks to implement your fix, or you know that the area of the code you are working in has had a lot of changes lately, make sure you rebase your branch against the remote master and take care of any conflicts before submitting your patch.

```
$ git checkout HBASE-123456
$ git rebase origin/master
```

8. Generate your patch against the remote master. Run the following command from the top level of your git repository (usually called `hbase`):

```
$ git format-patch --stdout origin/master > HBASE-123456.patch
```

   The name of the patch should contain the JIRA ID.

9. Look over the patch file to be sure that you did not change any additional files by accident and that there are no other surprises.

10. When you are satisfied, attach the patch to the JIRA and click the **Patch Available** button. A reviewer will review your patch.

11. If you need to submit a new version of the patch, leave the old one on the JIRA and add a version number to the name of the new patch.

12. After a change has been committed, there is no need to keep your local branch around.

Editing the HBase Website and Documentation

The HBase website and documentation are now part of a single application built with Remix and Fumadocs. The source files are located in the `hbase-website/` directory:

- **Documentation pages:** `hbase-website/app/pages/_docs/docs/_mdx/(multi-page)/` - individual MDX files for each documentation section
- **Single-page view:** `hbase-website/app/pages/_docs/docs/_mdx/single-page/index.mdx` - combines all documentation into one page
- **Website components:** `hbase-website/app/components/` - React components used throughout the site
- **Images:** `hbase-website/public/` - static assets including images

You can edit MDX files in any text editor or IDE. To preview your changes locally, run the development server from the `hbase-website/` directory and navigate to the documentation pages in your browser. When you are satisfied with your changes, follow the procedure in submit doc patch procedure to submit your patch.

Publishing the HBase Website and Documentation

The HBase website and documentation are built and deployed as a single Remix application. The deployment process is managed through the project's CI/CD pipeline, which builds the site from the `hbase-website/` directory and deploys it automatically when changes are merged to the main branch.

MDX and Fumadocs Components

The HBase documentation is written in MDX (Markdown with JSX), which allows you to use standard Markdown syntax along with React components. For comprehensive documentation on Markdown formatting and MDX features, refer to:

- Fumadocs Markdown Documentation - Complete guide to MDX syntax and Fumadocs features
- CommonMark specification - Standard Markdown syntax reference
- GFM (GitHub Flavored Markdown) - GitHub-style Markdown extensions

Fumadocs Components

Fumadocs provides several components that enhance the documentation:

Steps Component

Use `<Steps>` to create numbered step-by-step instructions:

```
<Steps>

<Step>First, do this thing.</Step>

<Step>Then, do this other thing.</Step>

</Steps>
```

Example output:

1. First, do this thing.
2. Then, do this other thing.

Callout Component

Use `<Callout>` for notes, warnings, and important information:

```
<Callout type="info">This is an informational callout.</Callout>

<Callout type="warning">This is a warning callout.</Callout>
```

Example output:

> ℹ This is an informational callout.

> ⚠ This is a warning callout.

Include Directive

The single-page documentation view uses `<include>` tags to combine multiple MDX files:

```
<include>../(multi-page)/getting-started.mdx</include>
```

See `hbase-website/app/pages/_docs/docs/_mdx/single-page/index.mdx` for examples of how all documentation sections are included in the single-page view.

Auto-Generated Content

Some parts of the HBase documentation, such as the **default configuration**, are generated automatically to stay in sync with the code. The configuration documentation is generated from the `hbase-common/src/main/resources/hbase-default.xml` file.

To add or modify configuration parameters, update the source XML file. To regenerate the documentation from the updated configuration, run:

```
npm run extract-hbase-config
```

This command is also executed automatically when you run `npm ci`.

Images in the Documentation

You can include images in the HBase documentation using standard Markdown syntax. Always include descriptive alt text for accessibility:

```
![Alt text describing the image](/path/to/image.png)
```

Save images to the `hbase-website/public/` directory or an appropriate subdirectory. Reference them in your MDX files using absolute paths from the public directory:

```
![Architecture diagram](/images/architecture-diagram.png)
```

When submitting a patch that includes images, attach the images to the JIRA issue.

Adding a New Section to the Documentation

To add a new section to the HBase documentation:

1. Create a new MDX file in `hbase-website/app/pages/_docs/docs/_mdx/(multi-page)/` with a descriptive name (e.g., `my-new-section.mdx`)
2. Add frontmatter at the top of the file with a title and description:

```
---
title: "My New Section"
description: "Brief description of what this section covers"
---

## My New Section

Your content here...
```

3. Add your new file to `hbase-website/app/pages/_docs/docs/_mdx/(multi-page)/meta.json` in the appropriate location within the `pages` array (without the `.mdx` extension):

```
{
  "pages": [
    "---My Category---",
    "my-new-section",
    ...
```

```
  ]
}
```

4. Add an `<include>` directive to `hbase-website/app/pages/_docs/docs/_mdx/single-page/index.mdx` in the appropriate location:

```
<include>../(multi-page)/my-new-section.mdx</include>
```

5. Add your new file to Git before creating your patch.

Unique Headings Requirement

Since all documentation files are merged into a single-page view, **all heading IDs must be unique across the entire documentation**. A test will fail during the build if duplicate heading IDs are detected, marking the problematic headings.

Headings don't have to be visually unique, but their link IDs must be unique. You can customize the heading ID using Fumadocs syntax:

```
## Configuration [#server-configuration]
```

This creates a heading that displays as "Configuration" but has the unique ID `#server-configuration` for linking purposes.

Hiding Headings from Table of Contents

You can hide specific headings from the right-side table of contents:

```
## Internal Implementation Details [!toc]
```

This heading will still appear in the document but won't show up in the table of contents navigation.

> ⚠ Note: `[!toc]` becomes part of the heading ID. For example, `## Usage [!toc]` will have the ID `#usage-toc`.

Combining Custom IDs and TOC Hiding

You can combine both attributes:

```
## Configuration Details [#server-config] [!toc]
```

Common Documentation Issues

The following documentation issues come up often:

1. Isolate Changes for Easy Diff Review

   Avoid reformatting entire files when making content changes. If you need to reformat a file, do that in a separate JIRA where you do not change any content.

2. Syntax Highlighting

   MDX supports syntax highlighting for code blocks. Specify the language after the opening triple backticks:

   ````
   ```java
   public class Example {
       // your code here
   }
   ```
   ````

3. Component Syntax

   Remember to properly close Fumadocs components. Components like `<Steps>` and `<Callout>` must be properly closed:

   ```
   <Callout>Your content here</Callout>
   ```

4. Unique Heading IDs

   Ensure all heading IDs are unique across the entire documentation. If you get a test failure about duplicate headings, customize the heading ID using `[#custom-id]` syntax as described in the Unique Headings Requirement section.

FAQ

General

When should I use HBase?

See Overview in the Architecture chapter.

Does HBase support SQL?

Not really. SQL-ish support for HBase via Hive is in development, however Hive is based on MapReduce which is not generally suitable for low-latency requests. See the Data Model section for examples on the HBase client.

How can I find examples of NoSQL/HBase?

See the link to the BigTable paper in Other Information About HBase, as well as the other papers.

What is the history of HBase?

See HBase History.

Why are the cells above 10MB not recommended for HBase?

Large cells don't fit well into HBase's approach to buffering data. First, the large cells bypass the MemStoreLAB when they are written. Then, they cannot be cached in the L2 block cache during read operations. Instead, HBase has to allocate on-heap memory for them each time. This can have a significant impact on the garbage collector within the RegionServer process.

Upgrading

How do I upgrade Maven-managed projects from HBase 0.94 to HBase 0.96+?

In HBase 0.96, the project moved to a modular structure. Adjust your project's dependencies to rely upon the `hbase-client` module or another module as appropriate, rather than a single JAR. You can model your Maven dependency after one of the following, depending on your targeted version of HBase. See Section 3.5, "Upgrading from 0.94.x to 0.96.x" or Section 3.3, "Upgrading from 0.96.x to 0.98.x" for more information.

Maven Dependency for HBase 0.98

```
<dependency>
  <groupId>org.apache.hbase</groupId>
  <artifactId>hbase-client</artifactId>
  <version>0.98.5-hadoop2</version>
</dependency>
```

Maven Dependency for HBase 0.96

```
<dependency>
  <groupId>org.apache.hbase</groupId>
  <artifactId>hbase-client</artifactId>
  <version>0.96.2-hadoop2</version>
</dependency>
```

Maven Dependency for HBase 0.94

```
<dependency>
  <groupId>org.apache.hbase</groupId>
  <artifactId>hbase</artifactId>
  <version>0.94.3</version>
</dependency>
```

Architecture

How does HBase handle Region-RegionServer assignment and locality?

See Regions.

Configuration

How can I get started with my first cluster?

See Quick Start - Standalone HBase.

Where can I learn about the rest of the configuration options?

See Apache HBase Configuration.

Schema Design / Data Access

How should I design my schema in HBase?

See Data Model and HBase and Schema Design.

How can I store (fill in the blank) in HBase?

See Supported Datatypes.

How can I handle secondary indexes in HBase?

See Secondary Indexes and Alternate Query Paths.

Can I change a table's rowkeys?

This is a very common question. You can't. See Immutability of Rowkeys.

What APIs does HBase support?

See Data Model, Client, and Apache HBase External APIs.

MapReduce

How can I use MapReduce with HBase?

See HBase and MapReduce.

Performance and Troubleshooting

How can I improve HBase cluster performance?

See Apache HBase Performance Tuning.

How can I troubleshoot my HBase cluster?

See Troubleshooting and Debugging Apache HBase.

Amazon EC2

I am running HBase on Amazon EC2 and...

EC2 issues are a special case. See Amazon EC2 and Amazon EC2.

Operations

How do I manage my HBase cluster?

See Apache HBase Operational Management.

How do I back up my HBase cluster?

See HBase Backup.

HBase in Action

Where can I find interesting videos and presentations on HBase?

See Other Information About HBase.

Access Control Matrix

Interpreting the ACL Matrix Table

The following conventions are used in the ACL Matrix table:

Scopes

Permissions are evaluated starting at the widest scope and working to the narrowest scope.

A scope corresponds to a level of the data model. From broadest to narrowest, the scopes are as follows:

Scopes

- Global
- Namespace (NS)
- Table
- Column Family (CF)
- Column Qualifier (CQ)
- Cell

For instance, a permission granted at table level dominates any grants done at the Column Family, Column Qualifier, or cell level. The user can do what that grant implies at any location in the table. A permission granted at global scope dominates all: the user is always allowed to take that action everywhere.

Permissions

Possible permissions include the following:

Permissions

- Superuser - a special user that belongs to group "supergroup" and has unlimited access
- Admin (A)
- Create (C)

- Write (W)
- Read (R)
- Execute (X)

For the most part, permissions work in an expected way, with the following caveats:

Having Write permission does not imply Read permission.
It is possible and sometimes desirable for a user to be able to write data that same user cannot read. One such example is a log-writing process.

The `hbase:meta` table is readable by every user, regardless of the user's other grants or restrictions.
This is a requirement for HBase to function correctly.

`CheckAndPut` and `CheckAndDelete` operations will fail if the user does not have both Write and Read permission.
`Increment` and `Append` operations do not require Read access.
The `superuser`, as the name suggests has permissions to perform all possible operations.
And for the operations marked with *, the checks are done in post hook and only subset of results satisfying access checks are returned back to the user.
The following table is sorted by the interface that provides each operation. In case the table goes out of date, the unit tests which check for accuracy of permissions can be found in *hbase-server/src/test/java/org/apache/hadoop/hbase/security/access/TestAccessController.java*, and the access controls themselves can be examined in *hbase-server/src/main/java/org/apache/hadoop/hbase/security/access/AccessController.java*.

ACL Matrix

Interface	Operation	Permissions
Master	createTable	superuser|global(C)|NS(C)
	modifyTable	superuser|global(A)|global(C)|NS(A)|NS(C)|TableOwner|table(A)|table(C)

Interface	Operation	Permissions
	deleteTable	superuser|global(A)|global(C)|NS(A)|NS(C)|TableOwner|table(A)|table(C)
	truncateTable	superuser|global(A)|global(C)|NS(A)|NS(C)|TableOwner|table(A)|table(C)
	addColumn	superuser|global(A)|global(C)|NS(A)|NS(C)|TableOwner|table(A)|table(C)
	modifyColumn	superuser|global(A)|global(C)|NS(A)|NS(C)|TableOwner|table(A)|table(C)|column(A)|column(C)
	deleteColumn	superuser|global(A)|global(C)|NS(A)|NS(C)|TableOwner|table(A)|table(C)|column(A)|column(C)
	enableTable	superuser|global(A)|global(C)|NS(A)|NS(C)|TableOwner|table(A)|table(C)
	disableTable	superuser|global(A)|global(C)|NS(A)|NS(C)|TableOwner|table(A)|table(C)
	disableAclTable	Not allowed
	move	superuser|global(A)|NS(A)|TableOwner|table(A)
	assign	superuser|global(A)|NS(A)|TableOwner|table(A)
	unassign	superuser|global(A)|NS(A)|TableOwner|table(A)
	regionOffline	superuser|global(A)|NS(A)|TableOwner|table(A)
	balance	superuser|global(A)
	balanceSwitch	superuser|global(A)

Interface	Operation	Permissions
	shutdown	superuser|global(A)
	stopMaster	superuser|global(A)
	snapshot	superuser|global(A)|NS(A)|TableOwner|table(A)
	listSnapshot	superuser|global(A)|SnapshotOwner
	cloneSnapshot	superuser|global(A)|(SnapshotOwner & TableName matches)
	restoreSnapshot	superuser|global(A)|SnapshotOwner & (NS(A)|TableOwner|table(A))
	deleteSnapshot	superuser|global(A)|SnapshotOwner
	createNamespace	superuser|global(A)
	deleteNamespace	superuser|global(A)
	modifyNamespace	superuser|global(A)
	getNamespaceDescriptor	superuser|global(A)|NS(A)
	listNamespaceDescriptors*	superuser|global(A)|NS(A)
	flushTable	superuser|global(A)|global(C)|NS(A)|NS(C)|TableOwner|table(A)|table(C)
	getTableDescriptors*	superuser|global(A)|global(C)|NS(A)|NS(C)|TableOwner|table(A)|table(C)
	getTableNames*	superuser|TableOwner|Any global or table perm
	setUserQuota(global level)	superuser|global(A)
	setUserQuota(namespace level)	superuser|global(A)
	setUserQuota(Table level)	superuser|global(A)|NS(A)|TableOwner|table(A)

Interface	Operation	Permissions
	setTableQuota	superuser|global(A)|NS(A)|TableOwner|table(A)
	setNamespaceQuota	superuser|global(A)
	addReplicationPeer	superuser|global(A)
	removeReplicationPeer	superuser|global(A)
	enableReplicationPeer	superuser|global(A)
	disableReplicationPeer	superuser|global(A)
	getReplicationPeerConfig	superuser|global(A)
	updateReplicationPeerConfig	superuser|global(A)
	listReplicationPeers	superuser|global(A)
	getClusterStatus	any user
Region	openRegion	superuser|global(A)
	closeRegion	superuser|global(A)
	flush	superuser|global(A)|global(C)|TableOwner|table(A)|table(C)
	split	superuser|global(A)|TableOwner|TableOwner|table(A)
	compact	superuser|global(A)|global(C)|TableOwner|table(A)|table(C)
	getClosestRowBefore	superuser|global(R)|NS(R)|TableOwner|table(R)|CF(R)|CQ(R)
	getOp	superuser|global(R)|NS(R)|TableOwner|table(R)|CF(R)|CQ(R)
	exists	superuser|global(R)|NS(R)|TableOwner|table(R)|CF(R)|CQ(R)
	put	superuser|global(W)|NS(W)|table(W)|TableOwner|CF(W)|CQ(W)
	delete	superuser|global(W)|NS(W)|table(W)|TableOwner|CF(W)|CQ(W)

Interface	Operation	Permissions
	batchMutate	superuser|global(W)|NS(W)|TableOwner|table(W)|CF(W)|CQ(W)
	checkAndPut	superuser|global(RW)|NS(RW)|TableOwner|table(RW)|CF(RW)|CQ(RW)
	checkAndPutAfterRowLock	superuser|global(R)|NS(R)|TableOwner|Table(R)|CF(R)|CQ(R)
	checkAndDelete	superuser|global(RW)|NS(RW)|TableOwner|table(RW)|CF(RW)|CQ(RW)
	checkAndDeleteAfterRowLock	superuser|global(R)|NS(R)|TableOwner|table(R)|CF(R)|CQ(R)
	incrementColumnValue	superuser|global(W)|NS(W)|TableOwner|table(W)|CF(W)|CQ(W)
	append	superuser|global(W)|NS(W)|TableOwner|table(W)|CF(W)|CQ(W)
	appendAfterRowLock	superuser|global(W)|NS(W)|TableOwner|table(W)|CF(W)|CQ(W)
	increment	superuser|global(W)|NS(W)|TableOwner|table(W)|CF(W)|CQ(W)
	incrementAfterRowLock	superuser|global(W)|NS(W)|TableOwner|table(W)|CF(W)|CQ(W)
	scannerOpen	superuser|global(R)|NS(R)|TableOwner|table(R)|CF(R)|CQ(R)
	scannerNext	superuser|global(R)|NS(R)|TableOwner|table(R)|CF(R)|CQ(R)
	scannerClose	superuser|global(R)|NS(R)|TableOwner|table(R)|CF(R)|CQ(R)

Interface	Operation	Permissions
	bulkLoadHFile	superuser|global(C)|TableOwner|table(C)|CF(C)
	prepareBulkLoad	superuser|global(C)|TableOwner|table(C)|CF(C)
	cleanupBulkLoad	superuser|global(C)|TableOwner|table(C)|CF(C)
Endpoint	invoke	superuser|global(X)|NS(X)|TableOwner|table(X)
AccessController	grant(global level)	global(A)
	grant(namespace level)	global(A)|NS(A)
	grant(table level)	global(A)|NS(A)|TableOwner|table(A)|CF(A)|CQ(A)
	revoke(global level)	global(A)
	revoke(namespace level)	global(A)|NS(A)
	revoke(table level)	global(A)|NS(A)|TableOwner|table(A)|CF(A)|CQ(A)
	getUserPermissions(global level)	global(A)
	getUserPermissions(namespace level)	global(A)|NS(A)
	getUserPermissions(table level)	global(A)|NS(A)|TableOwner|table(A)|CF(A)|CQ(A)
	hasPermission(table level)	global(A)|SelfUserCheck
RegionServer	stopRegionServer	superuser|global(A)
	mergeRegions	superuser|global(A)
	rollWALWriterRequest	superuser|global(A)
	replicateLogEntries	superuser|global(W)
RSGroup	addRSGroup	superuser|global(A)
	balanceRSGroup	superuser|global(A)
	getRSGroupInfo	superuser|global(A)

Interface	Operation	Permissions
	getRSGroupInfoOfTable	superuser|global(A)
	getRSGroupOfServer	superuser|global(A)
	listRSGroups	superuser|global(A)
	moveServers	superuser|global(A)
	moveServersAndTables	superuser|global(A)
	moveTables	superuser|global(A)
	removeRSGroup	superuser|global(A)
	removeServers	superuser|global(A)

Compression and Data Block Encoding In HBase

> Codecs mentioned in this section are for encoding and decoding data blocks or row keys. For information about replication codecs, see cluster.replication.preserving.tags.

HBase supports several different compression algorithms which can be enabled on a ColumnFamily. Data block encoding attempts to limit duplication of information in keys, taking advantage of some of the fundamental designs and patterns of HBase, such as sorted row keys and the schema of a given table. Compressors reduce the size of large, opaque byte arrays in cells, and can significantly reduce the storage space needed to store uncompressed data.

Compressors and data block encoding can be used together on the same ColumnFamily.

Changes Take Effect Upon Compaction

If you change compression or encoding for a ColumnFamily, the changes take effect during compaction.

Some codecs take advantage of capabilities built into Java, such as GZip compression. Others rely on native libraries. Native libraries may be available via codec dependencies installed into HBase's library directory, or, if you are utilizing Hadoop codecs, as part of Hadoop. Hadoop codecs typically have a native code component so follow instructions for installing Hadoop native binary support at Making use of Hadoop Native Libraries in HBase.

This section discusses common codecs that are used and tested with HBase.

No matter what codec you use, be sure to test that it is installed correctly and is available on all nodes in your cluster. Extra operational steps may be necessary to be sure that codecs are available on newly-deployed nodes. You can use the compression.test utility to check that a given codec is correctly installed.

To configure HBase to use a compressor, see compressor.install. To enable a compressor for a ColumnFamily, see changing.compression. To enable data block encoding for a ColumnFamily, see data.block.encoding.enable.

Block Compressors

- NONE
  This compression type constant selects no compression, and is the default.
- BROTLI
  Brotli is a generic-purpose lossless compression algorithm that compresses data using a combination of a modern variant of the LZ77 algorithm, Huffman coding, and 2nd order context modeling, with a compression ratio comparable to the best currently available general-purpose compression methods. It is similar in speed with GZ but offers more dense compression.
- BZIP2
  Bzip2 compresses files using the Burrows-Wheeler block sorting text compression algorithm and Huffman coding. Compression is generally considerably better than that achieved by the dictionary- (LZ-) based compressors, but both compression and decompression can be slow in comparison to other options.
- GZ
  gzip is based on the DEFLATE algorithm, which is a combination of LZ77 and Huffman coding. It is universally available in the Java Runtime Environment so is a good lowest common denominator option. However in comparison to more modern algorithms like Zstandard it is quite slow.
- LZ4
  LZ4 is a lossless data compression algorithm that is focused on compression and decompression speed. It belongs to the LZ77 family of compression algorithms, like Brotli, DEFLATE, Zstandard, and others. In our microbenchmarks LZ4 is the fastest option for both compression and decompression in that family, and is our universally recommended option.
- LZMA
  LZMA is a dictionary compression scheme somewhat similar to the LZ77 algorithm that achieves very high compression ratios with a computationally expensive predictive model and variable size compression dictionary, while still maintaining decompression speed similar to other commonly used compression algorithms. LZMA is superior to all other options in general compression ratio but as a compressor it can be extremely slow, especially when configured to operate at higher levels of compression.
- LZO
  LZO is another LZ-variant data compression algorithm, with an implementation focused

on decompression speed. It is almost but not quite as fast as LZ4.

- SNAPPY
  Snappy is based on ideas from LZ77 but is optimized for very high compression speed, achieving only a "reasonable" compression in trade. It is as fast as LZ4 but does not compress quite as well. We offer a pure Java Snappy codec that can be used instead of GZ as the universally available option for any Java runtime on any hardware architecture.
- ZSTD
  Zstandard combines a dictionary-matching stage (LZ77) with a large search window and a fast entropy coding stage, using both Finite State Entropy and Huffman coding. Compression speed can vary by a factor of 20 or more between the fastest and slowest levels, while decompression is uniformly fast, varying by less than 20% between the fastest and slowest levels.
  ZStandard is the most flexible of the available compression codec options, offering a compression ratio similar to LZ4 at level 1 (but with slightly less performance), compression ratios comparable to DEFLATE at mid levels (but with better performance), and LZMA-alike dense compression (and LZMA-alike compression speeds) at high levels; while providing universally fast decompression.

Data Block Encoding Types

Prefix

Often, keys are very similar. Specifically, keys often share a common prefix and only differ near the end. For instance, one key might be `RowKey:Family:Qualifier0` and the next key might be `RowKey:Family:Qualifier1`. In Prefix encoding, an extra column is added which holds the length of the prefix shared between the current key and the previous key. Assuming the first key here is totally different from the key before, its prefix length is 0.

The second key's prefix length is `23`, since they have the first 23 characters in common.

Obviously if the keys tend to have nothing in common, Prefix will not provide much benefit.

The following image shows a hypothetical ColumnFamily with no data block encoding.

Key Len	Val Len	Key	Value
24	...	RowKey:Family:Qualifier0	...
24	...	RowKey:Family:Qualifier1	...
25	...	RowKey:Family:QualifierN	...
25	...	RowKey2:Family:Qualifier1	...
25	...	RowKey2:Family:Qualifier2	...
...	...	...	...

Here is the same data with prefix data encoding.

Key Len	Val Len	Prefix Len	Key	Value
24	...	0	RowKey:Family:Qualifier0	...
1	...	23	1	...
1	...	23	N	...
19	...	6	2:Family:Qualifier1	...
1	...	24	2	...
...	...	...	...	...

Diff

Diff encoding expands upon Prefix encoding. Instead of considering the key sequentially as a monolithic series of bytes, each key field is split so that each part of the key can be compressed more efficiently.

Two new fields are added: timestamp and type.

If the ColumnFamily is the same as the previous row, it is omitted from the current row.

If the key length, value length or type are the same as the previous row, the field is omitted.

In addition, for increased compression, the timestamp is stored as a Diff from the previous row's timestamp, rather than being stored in full. Given the two row keys in the Prefix

example, and given an exact match on timestamp and the same type, neither the value length, or type needs to be stored for the second row, and the timestamp value for the second row is just 0, rather than a full timestamp.

Diff encoding is disabled by default because writing and scanning are slower but more data is cached.

This image shows the same ColumnFamily from the previous images, with Diff encoding.

Flags	Key Len	Val Len	Prefix Len	Key	Timestamp	Type	Value
0	24	512	0	RowKey:Family:Qualifier0	1340466835163	4	...
5		320	23	1	0		...
3			23	N	120	8	...
0	25	576	6	2:Family:Qualifier1	25	4	...
5		384	24	2	1124		...
...	...	...	...	...	...	...	...

Fast Diff

Fast Diff works similar to Diff, but uses a faster implementation. It also adds another field which stores a single bit to track whether the data itself is the same as the previous row. If it is, the data is not stored again.

Fast Diff is the recommended codec to use if you have long keys or many columns.

The data format is nearly identical to Diff encoding, so there is not an image to illustrate it.

Prefix Tree

Prefix tree encoding was introduced as an experimental feature in HBase 0.96. It provides similar memory savings to the Prefix, Diff, and Fast Diff encoder, but provides faster random access at a cost of slower encoding speed. It was removed in hbase-2.0.0. It was a good idea but little uptake. If interested in reviving this effort, write the hbase dev list.

Which Compressor or Data Block Encoder To Use

The compression or codec type to use depends on the characteristics of your data. Choosing the wrong type could cause your data to take more space rather than less, and

can have performance implications.

In general, you need to weigh your options between smaller size and faster compression/decompression. Following are some general guidelines, expanded from a discussion at Documenting Guidance on compression and codecs.

- In most cases, enabling LZ4 or Snappy by default is a good choice, because they have a low performance overhead and provide reasonable space savings. A fast compression algorithm almost always improves overall system performance by trading some increased CPU usage for better I/O efficiency.
- If the values are large (and not pre-compressed, such as images), use a data block compressor.
- For *cold data*, which is accessed infrequently, depending on your use case, it might make sense to opt for Zstandard at its higher compression levels, or LZMA, especially for high entropy binary data, or Brotli for data similar in characteristics to web data. Bzip2 might also be a reasonable option but Zstandard is very likely to offer superior decompression speed.
- For *hot data*, which is accessed frequently, you almost certainly want only LZ4, Snappy, LZO, or Zstandard at a low compression level. These options will not provide as high of a compression ratio but will in trade not unduly impact system performance.
- If you have long keys (compared to the values) or many columns, use a prefix encoder. FAST_DIFF is recommended.
- If enabling WAL value compression, consider LZ4 or SNAPPY compression, or Zstandard at level 1. Reading and writing the WAL is performance critical. That said, the I/O savings of these compression options can improve overall system performance.

Making use of Hadoop Native Libraries in HBase

The Hadoop shared library has a bunch of facility including compression libraries and fast crc'ing — hardware crc'ing if your chipset supports it. To make this facility available to HBase, do the following. HBase/Hadoop will fall back to use alternatives if it cannot find the native library versions — or fail outright if you asking for an explicit compressor and there is no alternative available.

First make sure of your Hadoop. Fix this message if you are seeing it starting Hadoop processes:

```
16/02/09 22:40:24 WARN util.NativeCodeLoader: Unable to load native-hadoop librar
y for your platform... using builtin-java classes where applicable
```

It means is not properly pointing at its native libraries or the native libs were compiled for another platform. Fix this first.

Then if you see the following in your HBase logs, you know that HBase was unable to locate the Hadoop native libraries:

```
2014-08-07 09:26:20,139 WARN  [main] util.NativeCodeLoader: Unable to load native
-hadoop library for your platform... using builtin-java classes where applicable
```

If the libraries loaded successfully, the WARN message does not show. Usually this means you are good to go but read on.

Let's presume your Hadoop shipped with a native library that suits the platform you are running HBase on. To check if the Hadoop native library is available to HBase, run the following tool (available in Hadoop 2.1 and greater):

```
$ ./bin/hbase --config ~/conf_hbase org.apache.hadoop.util.NativeLibraryChecker
2014-08-26 13:15:38,717 WARN  [main] util.NativeCodeLoader: Unable to load native
-hadoop library for your platform... using builtin-java classes where applicable
Native library checking:
hadoop: false
zlib:   false
snappy: false
lz4:    false
bzip2:  false
2014-08-26 13:15:38,863 INFO  [main] util.ExitUtil: Exiting with status 1
```

Above shows that the native hadoop library is not available in HBase context.

The above NativeLibraryChecker tool may come back saying all is hunky-dory — i.e. all libs show 'true', that they are available — but follow the below presecription anyways to ensure the native libs are available in HBase context, when it goes to use them.

To fix the above, either copy the Hadoop native libraries local or symlink to them if the Hadoop and HBase stalls are adjacent in the filesystem. You could also point at their location by setting the `LD_LIBRARY_PATH` environment variable in your hbase-env.sh.

Where the JVM looks to find native libraries is "system dependent" (See `java.lang.System#loadLibrary(name)`). On linux, by default, is going to look in *lib/native/PLATFORM* where `PLATFORM` is the label for the platform your HBase is installed on. On a local linux machine, it seems to be the concatenation of the java properties `os.name` and `os.arch` followed by whether 32 or 64 bit. HBase on startup prints out all of the java system properties so find the os.name and os.arch in the log. For example:

```
...
2014-08-06 15:27:22,853 INFO  [main] zookeeper.ZooKeeper: Client environment:os.n
ame=Linux
2014-08-06 15:27:22,853 INFO  [main] zookeeper.ZooKeeper: Client environment:os.a
rch=amd64
...
```

So in this case, the PLATFORM string is `Linux-amd64-64`. Copying the Hadoop native libraries or symlinking at *lib/native/Linux-amd64-64* will ensure they are found. Rolling restart after you have made this change.

Here is an example of how you would set up the symlinks. Let the hadoop and hbase installs be in your home directory. Assume your hadoop native libs are at ~/hadoop/lib/native. Assume you are on a Linux-amd64-64 platform. In this case, you would do the following to link the hadoop native lib so hbase could find them.

```
...
$ mkdir -p ~/hbaseLinux-amd64-64 -> /home/stack/hadoop/lib/native/lib/native/
$ cd ~/hbase/lib/native/
$ ln -s ~/hadoop/lib/native Linux-amd64-64
$ ls -la
# Linux-amd64-64 -> /home/USER/hadoop/lib/native
...
```

If you see PureJavaCrc32C in a stack track or if you see something like the below in a perf trace, then native is not working; you are using the java CRC functions rather than native:

```
  5.02%  perf-53601.map      [.] Lorg/apache/hadoop/util/PureJavaCrc32C;.update
```

See HBASE-11927 Use Native Hadoop Library for HFile checksum (And flip default from CRC32 to CRC32C), for more on native checksumming support. See in particular the

release note for how to check if your hardware to see if your processor has support for hardware CRCs. Or checkout the Apache Checksums in HBase blog post.

Here is example of how to point at the Hadoop libs with `LD_LIBRARY_PATH` environment variable:

```
$ LD_LIBRARY_PATH=~/hadoop-2.5.0-SNAPSHOT/lib/native ./bin/hbase --config ~/conf_
hbase org.apache.hadoop.util.NativeLibraryChecker
2014-08-26 13:42:49,332 INFO  [main] bzip2.Bzip2Factory: Successfully loaded & in
itialized native-bzip2 library system-native
2014-08-26 13:42:49,337 INFO  [main] zlib.ZlibFactory: Successfully loaded & init
ialized native-zlib library
Native library checking:
hadoop: true /home/stack/hadoop-2.5.0-SNAPSHOT/lib/native/libhadoop.so.1.0.0
zlib:   true /lib64/libz.so.1
snappy: true /usr/lib64/libsnappy.so.1
lz4:    true revision:99
bzip2:  true /lib64/libbz2.so.1
```

Set in *hbase-env.sh* the LD_LIBRARY_PATH environment variable when starting your HBase.

Compressor Configuration, Installation, and Use

Configure HBase For Compressors

Compression codecs are provided either by HBase compressor modules or by Hadoop's native compression support. As described above you choose a compression type in table or column family schema or in site configuration using its short label, e.g. *snappy* for Snappy, or *zstd* for ZStandard. Which codec implementation is dynamically loaded to support what label is configurable by way of site configuration.

Algorithm label	Codec implementation configuration key	Default value
BROTLI	hbase.io.compress.brotli.codec	org.apache.hadoop.hbase.io.compress.brotli.BrotliCodec
BZIP2	hbase.io.compress.bzip2.codec	org.apache.hadoop.io.compress.BZip2Codec

Algorithm label	Codec implementation configuration key	Default value
GZ	hbase.io.compress.gz.codec	org.apache.hadoop.hbase.io.compress.ReusableStreamGzipCodec
LZ4	hbase.io.compress.lz4.codec	org.apache.hadoop.io.compress.Lz4Codec
LZMA	hbase.io.compress.lzma.codec	org.apache.hadoop.hbase.io.compress.xz.LzmaCodec
LZO	hbase.io.compress.lzo.codec	com.hadoop.compression.lzo.LzoCodec
SNAPPY	hbase.io.compress.snappy.codec	org.apache.hadoop.io.compress.SnappyCodec
ZSTD	hbase.io.compress.zstd.codec	org.apache.hadoop.io.compress.ZStandardCodec

The available codec implementation options are:

Label	Codec implementation class	Notes
BROTLI	org.apache.hadoop.hbase.io.compress.brotli.BrotliCodec	Implemented with Brotli4j
BZIP2	org.apache.hadoop.io.compress.BZip2Codec	Hadoop native codec
GZ	org.apache.hadoop.hbase.io.compress.ReusableStreamGzipCodec	Requires the Hadoop native GZ codec
LZ4	org.apache.hadoop.io.compress.Lz4Codec	Hadoop native codec
LZ4	org.apache.hadoop.hbase.io.compress.aircompressor.Lz4Codec	Pure Java implementation
LZ4	org.apache.hadoop.hbase.io.compress.lz4.Lz4Codec	Implemented with lz4-java
LZMA	org.apache.hadoop.hbase.io.compress.xz.LzmaCodec	Implemented with XZ For Java

Label	Codec implementation class	Notes
LZO	com.hadoop.compression.lzo.LzoCodec	Hadoop native codec, requires GPL licensed native dependencies
LZO	org.apache.hadoop.io.compress.LzoCodec	Hadoop native codec, requires GPL licensed native dependencies
LZO	org.apache.hadoop.hbase.io.compress.aircompressor.LzoCodec	Pure Java implementation
SNAPPY	org.apache.hadoop.io.compress.SnappyCodec	Hadoop native codec
SNAPPY	org.apache.hadoop.hbase.io.compress.aircompressor.SnappyCodec	Pure Java implementation
SNAPPY	org.apache.hadoop.hbase.io.compress.xerial.SnappyCodec	Implemented with snappy-java
ZSTD	org.apache.hadoop.io.compress.ZStandardCodec	Hadoop native codec
ZSTD	org.apache.hadoop.hbase.io.compress.aircompressor.ZstdCodec	Pure Java implementation, limited to a fixed compression level, not data compatible with the Hadoop zstd codec
ZSTD	org.apache.hadoop.hbase.io.compress.zstd.ZstdCodec	Implemented with zstd-jni, supports all compression levels, supports custom dictionaries

Specify which codec implementation option you prefer for a given compression algorithm in site configuration, like so:

```
...
<property>
  <name>hbase.io.compress.lz4.codec</name>
  <value>org.apache.hadoop.hbase.io.compress.lz4.Lz4Codec</value>
</property>
...
```

Compressor Microbenchmarks

See https://github.com/apurtell/jmh-compression-tests

256MB (258,126,022 bytes exactly) of block data was extracted from two HFiles containing Common Crawl data ingested using IntegrationLoadTestCommonCrawl, 2,680 blocks in total. This data was processed by each new codec implementation as if the block data were being compressed again for write into an HFile, but without writing any data, comparing only the CPU time and resource demand of the codec itself. Absolute performance numbers will vary depending on hardware and software particulars of your deployment. The relative differences are what are interesting. Measured time is the average time in milliseconds required to compress all blocks of the 256MB file. This is how long it would take to write the HFile containing these contents, minus the I/O overhead of block encoding and actual persistence.

These are the results:

Codec	Level	Time (milliseconds)	Result (bytes)	Improvement
AirCompressor LZ4	-	349.989 ± 2.835	76,999,408	70.17%
AirCompressor LZO	-	334.554 ± 3.243	79,369,805	69.25%
AirCompressor Snappy	-	364.153 ± 19.718	80,201,763	68.93%
AirCompressor Zstandard	3 (effective)	1108.267 ± 8.969	55,129,189	78.64%
Brotli	1	593.107 ± 2.376	58,672,319	77.27%
Brotli	3	1345.195 ± 27.327	53,917,438	79.11%
Brotli	6	2812.411 ± 25.372	48,696,441	81.13%
Brotli	10	74615.936 ± 224.854	44,970,710	82.58%
LZ4 (lz4-java)	-	303.045 ± 0.783	76,974,364	70.18%

Codec	Level	Time (milliseconds)	Result (bytes)	Improvement
LZMA	1	6410.428 ± 115.065	49,948,535	80.65%
LZMA	3	8144.620 ± 152.119	49,109,363	80.97%
LZMA	6	43802.576 ± 382.025	46,951,810	81.81%
LZMA	9	49821.979 ± 580.110	46,951,810	81.81%
Snappy (xerial)	-	360.225 ± 2.324	80,749,937	68.72%
Zstd (zstd-jni)	1	654.699 ± 16.839	56,719,994	78.03%
Zstd (zstd-jni)	3	839.160 ± 24.906	54,573,095	78.86%
Zstd (zstd-jni)	5	1594.373 ± 22.384	52,025,485	79.84%
Zstd (zstd-jni)	7	2308.705 ± 24.744	50,651,554	80.38%
Zstd (zstd-jni)	9	3659.677 ± 58.018	50,208,425	80.55%
Zstd (zstd-jni)	12	8705.294 ± 58.080	49,841,446	80.69%
Zstd (zstd-jni)	15	19785.646 ± 278.080	48,499,508	81.21%
Zstd (zstd-jni)	18	47702.097 ± 442.670	48,319,879	81.28%
Zstd (zstd-jni)	22	97799.695 ± 1106.571	48,212,220	81.32%

Compressor Support On the Master

A new configuration setting was introduced in HBase 0.95, to check the Master to determine which data block encoders are installed and configured on it, and assume that the entire cluster is configured the same. This option, `hbase.master.check.compression`,

defaults to `true`. This prevents the situation described in HBASE-6370, where a table is created or modified to support a codec that a region server does not support, leading to failures that take a long time to occur and are difficult to debug.

If `hbase.master.check.compression` is enabled, libraries for all desired compressors need to be installed and configured on the Master, even if the Master does not run a region server.

Install GZ Support Via Native Libraries

HBase uses Java's built-in GZip support unless the native Hadoop libraries are available on the CLASSPATH. The recommended way to add libraries to the CLASSPATH is to set the environment variable `HBASE_LIBRARY_PATH` for the user running HBase. If native libraries are not available and Java's GZIP is used, `Got brand-new compressor` reports will be present in the logs. See brand.new.compressor).

Install Hadoop Native LZO Support

HBase cannot ship with the Hadoop native LZO codc because of incompatibility between HBase, which uses an Apache Software License (ASL) and LZO, which uses a GPL license. See the Hadoop-LZO at Twitter for information on configuring LZO support for HBase.

If you depend upon LZO compression, consider using the pure Java and ASL licensed AirCompressor LZO codec option instead of the Hadoop native default, or configure your RegionServers to fail to start if native LZO support is not available. See hbase.regionserver.codecs.

Configure Hadoop Native LZ4 Support

LZ4 support is bundled with Hadoop and is the default LZ4 codec implementation. It is not required that you make use of the Hadoop LZ4 codec. Our LZ4 codec implemented with lz4-java offers superior performance, and the AirCompressor LZ4 codec offers a pure Java option for use where native support is not available.

That said, if you prefer the Hadoop option, make sure the hadoop shared library (libhadoop.so) is accessible when you start HBase. After configuring your platform (see hadoop.native.lib), you can make a symbolic link from HBase to the native Hadoop libraries. This assumes the two software installs are colocated. For example, if my 'platform' is Linux-amd64-64:

```
$ cd $HBASE_HOME
$ mkdir lib/native
```

```
$ ln -s $HADOOP_HOME/lib/native lib/native/Linux-amd64-64
```

Use the compression tool to check that LZ4 is installed on all nodes. Start up (or restart) HBase. Afterward, you can create and alter tables to enable LZ4 as a compression codec.:

```
hbase(main):003:0> alter 'TestTable', {NAME => 'info', COMPRESSION => 'LZ4'}
```

Install Hadoop native Snappy Support

Snappy support is bundled with Hadoop and is the default Snappy codec implementation. It is not required that you make use of the Hadoop Snappy codec. Our Snappy codec implemented with Xerial Snappy offers superior performance, and the AirCompressor Snappy codec offers a pure Java option for use where native support is not available.

That said, if you prefer the Hadoop codec option, you can install Snappy binaries (for instance, by using +yum install snappy+ on CentOS) or build Snappy from source. After installing Snappy, search for the shared library, which will be called *libsnappy.so.X* where X is a number. If you built from source, copy the shared library to a known location on your system, such as */opt/snappy/lib/*.

In addition to the Snappy library, HBase also needs access to the Hadoop shared library, which will be called something like *libhadoop.so.X.Y*, where X and Y are both numbers. Make note of the location of the Hadoop library, or copy it to the same location as the Snappy library.

> The Snappy and Hadoop libraries need to be available on each node of your cluster. See compression.test to find out how to test that this is the case.
>
> See hbase.regionserver.codecs to configure your RegionServers to fail to start if a given compressor is not available.

Each of these library locations need to be added to the environment variable `HBASE_LIBRARY_PATH` for the operating system user that runs HBase. You need to restart the RegionServer for the changes to take effect.

CompressionTest

You can use the CompressionTest tool to verify that your compressor is available to HBase:

```
 $ hbase org.apache.hadoop.hbase.util.CompressionTest hdfs://host/path/to/hbase s
nappy
```

Enforce Compression Settings On a RegionServer

You can configure a RegionServer so that it will fail to restart if compression is configured incorrectly, by adding the option hbase.regionserver.codecs to the *hbase-site.xml*, and setting its value to a comma-separated list of codecs that need to be available. For example, if you set this property to `lzo,gz`, the RegionServer would fail to start if both compressors were not available. This would prevent a new server from being added to the cluster without having codecs configured properly.

Enable Compression On a ColumnFamily

To enable compression for a ColumnFamily, use an `alter` command. You do not need to re-create the table or copy data. If you are changing codecs, be sure the old codec is still available until all the old StoreFiles have been compacted.

Enabling Compression on a ColumnFamily of an Existing Table using HBaseShell

```
hbase> alter 'test', {NAME => 'cf', COMPRESSION => 'GZ'}
```

Creating a New Table with Compression On a ColumnFamily

```
hbase> create 'test2', { NAME => 'cf2', COMPRESSION => 'SNAPPY' }
```

Verifying a ColumnFamily's Compression Settings

```
hbase> describe 'test'
DESCRIPTION                                          ENABLED
 'test', {NAME => 'cf', DATA_BLOCK_ENCODING => 'NONE false
 ', BLOOMFILTER => 'ROW', REPLICATION_SCOPE => '0',
 VERSIONS => '1', COMPRESSION => 'GZ', MIN_VERSIONS
 => '0', TTL => 'FOREVER', KEEP_DELETED_CELLS => 'fa
 lse', BLOCKSIZE => '65536', IN_MEMORY => 'false', B
 LOCKCACHE => 'true'}
1 row(s) in 0.1070 seconds
```

Testing Compression Performance

HBase includes a tool called LoadTestTool which provides mechanisms to test your compression performance. You must specify either `-write` or `-update-read` as your first parameter, and if you do not specify another parameter, usage advice is printed for each option.

`LoadTestTool` Usage

```
$ bin/hbase org.apache.hadoop.hbase.util.LoadTestTool -h
usage: bin/hbase org.apache.hadoop.hbase.util.LoadTestTool <options>
Options:
 -batchupdate                 Whether to use batch as opposed to separate
                              updates for every column in a row
 -bloom <arg>                 Bloom filter type, one of [NONE, ROW, ROWCOL]
 -compression <arg>           Compression type, one of [LZO, GZ, NONE, SNAPPY,
                              LZ4]
 -data_block_encoding <arg>   Encoding algorithm (e.g. prefix compression) to
                              use for data blocks in the test column family, one
                              of [NONE, PREFIX, DIFF, FAST_DIFF, ROW_INDEX_V1].
 -encryption <arg>            Enables transparent encryption on the test table,
                              one of [AES]
 -generator <arg>             The class which generates load for the tool. Any
                              args for this class can be passed as colon
                              separated after class name
 -h,--help                    Show usage
 -in_memory                   Tries to keep the HFiles of the CF inmemory as far
                              as possible.  Not guaranteed that reads are always
                              served from inmemory
 -init_only                   Initialize the test table only, don't do any
                              loading
 -key_window <arg>            The 'key window' to maintain between reads and
                              writes for concurrent write/read workload. The
                              default is 0.
 -max_read_errors <arg>       The maximum number of read errors to tolerate
                              before terminating all reader threads. The default
                              is 10.
 -multiput                    Whether to use multi-puts as opposed to separate
                              puts for every column in a row
 -num_keys <arg>              The number of keys to read/write
 -num_tables <arg>            A positive integer number. When a number n is
                              speicfied, load test tool  will load n table
                              parallely. -tn parameter value becomes table name
                              prefix. Each table name is in format
                              <tn>_1...<tn>_n
 -read <arg>                  <verify_percent>[:#threads=20]
 -regions_per_server <arg>    A positive integer number. When a number n is
                              specified, load test tool will create the test
                              table with n regions per server
 -skip_init                   Skip the initialization; assume test table already
                              exists
 -start_key <arg>             The first key to read/write (a 0-based index). The
```

```
                              default value is 0.
 -tn <arg>                    The name of the table to read or write
 -update <arg>                <update_percent>[:#threads=20][:#whether to
                              ignore nonce collisions=0]
 -write <arg>                 <avg_cols_per_key>:<avg_data_size>[:#threads=20]
 -zk <arg>                    ZK quorum as comma-separated host names without
                              port numbers
 -zk_root <arg>               name of parent znode in zookeeper
```

Example Usage of LoadTestTool

```
$ hbase org.apache.hadoop.hbase.util.LoadTestTool -write 1:10:100 -num_keys 10000
00 \
      -read 100:30 -num_tables 1 -data_block_encoding NONE -tn load_test_tool_NON
E
```

Enable Data Block Encoding

Codecs are built into HBase so no extra configuration is needed. Codecs are enabled on a table by setting the `DATA_BLOCK_ENCODING` property. Disable the table before altering its DATA_BLOCK_ENCODING setting. Following is an example using HBase Shell:

Enable Data Block Encoding On a Table

```
hbase> alter 'test', { NAME => 'cf', DATA_BLOCK_ENCODING => 'FAST_DIFF' }
Updating all regions with the new schema...
0/1 regions updated.
1/1 regions updated.
Done.
0 row(s) in 2.2820 seconds
```

Verifying a ColumnFamily's Data Block Encoding

```
hbase> describe 'test'
DESCRIPTION                                          ENABLED
 'test', {NAME => 'cf', DATA_BLOCK_ENCODING => 'FAST true
 _DIFF', BLOOMFILTER => 'ROW', REPLICATION_SCOPE =>
 '0', VERSIONS => '1', COMPRESSION => 'GZ', MIN_VERS
 IONS => '0', TTL => 'FOREVER', KEEP_DELETED_CELLS =
 > 'false', BLOCKSIZE => '65536', IN_MEMORY => 'fals
 e', BLOCKCACHE => 'true'}
1 row(s) in 0.0650 seconds
```

SQL over HBase

Apache Phoenix

Trafodion: Transactional SQL-on-HBase

YCSB

YCSB: The Yahoo! Cloud Serving Benchmark and HBase

TODO: Describe how YCSB is poor for putting up a decent cluster load.

TODO: Describe setup of YCSB for HBase. In particular, presplit your tables before you start a run. See HBASE-4163 Create Split Strategy for YCSB Benchmark for why and a little shell command for how to do it.

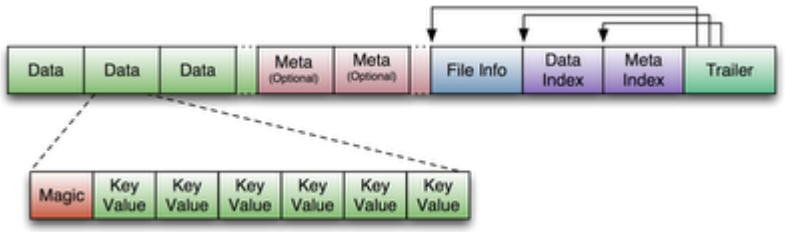
HFile Format

HBase File Format (version 1)

As we will be discussing changes to the HFile format, it is useful to give a short overview of the original (HFile version 1) format.

Overview of Version 1

An HFile in version 1 format is structured as follows:



Block index format in version 1

The block index in version 1 is very straightforward. For each entry, it contains:

1. Offset (long)
2. Uncompressed size (int)
3. Key (a serialized byte array written using Bytes.writeByteArray)
    - Key length as a variable-length integer (VInt)
    - Key bytes

The number of entries in the block index is stored in the fixed file trailer, and has to be passed in to the method that reads the block index. One of the limitations of the block index in version 1 is that it does not provide the compressed size of a block, which turns out to be necessary for decompression. Therefore, the HFile reader has to infer this compressed size from the offset difference between blocks. We fix this limitation in version 2, where we store on-disk block size instead of uncompressed size, and get uncompressed size from the block header.

HBase file format with inline blocks (version 2)

Note: this feature was introduced in HBase 0.92

Motivation

We found it necessary to revise the HFile format after encountering high memory usage and slow startup times caused by large Bloom filters and block indexes in the region server. Bloom filters can get as large as 100 MB per HFile, which adds up to 2 GB when aggregated over 20 regions. Block indexes can grow as large as 6 GB in aggregate size over the same set of regions. A region is not considered opened until all of its block index data is loaded. Large Bloom filters produce a different performance problem: the first get request that requires a Bloom filter lookup will incur the latency of loading the entire Bloom filter bit array.

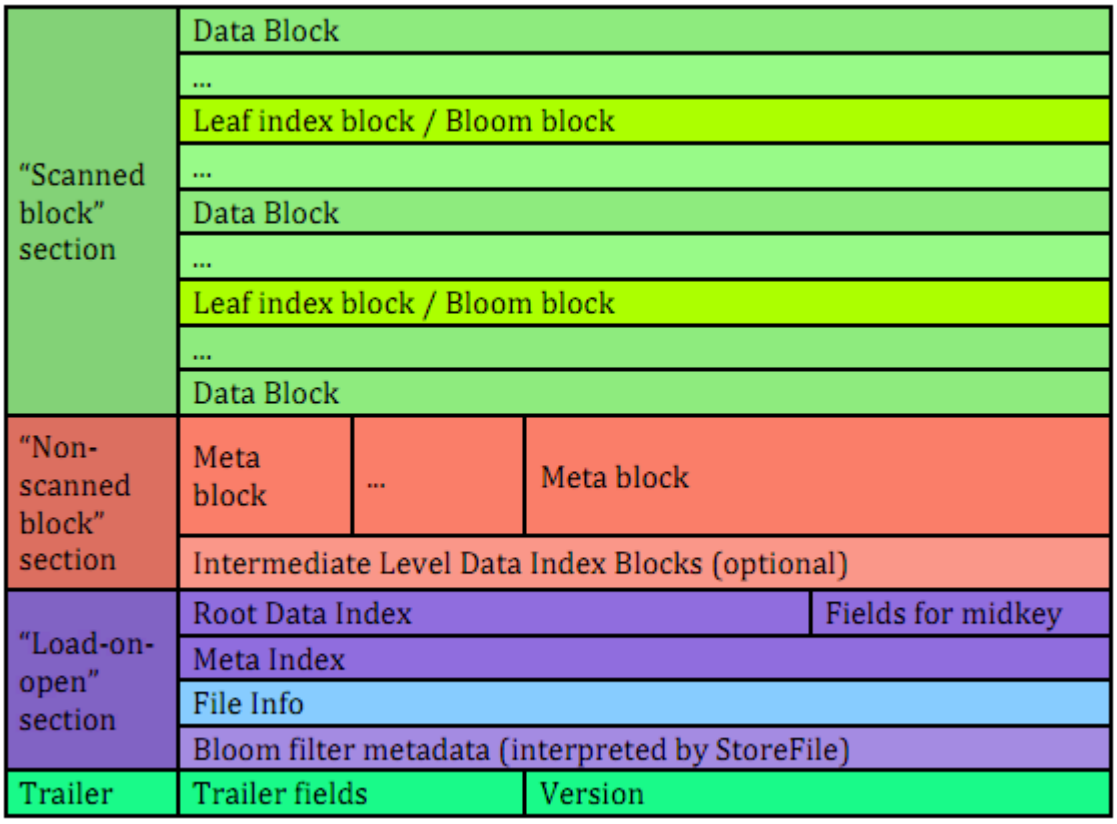
To speed up region server startup we break Bloom filters and block indexes into multiple blocks and write those blocks out as they fill up, which also reduces the HFile writer's memory footprint. In the Bloom filter case, "filling up a block" means accumulating enough keys to efficiently utilize a fixed-size bit array, and in the block index case we accumulate an "index block" of the desired size. Bloom filter blocks and index blocks (we call these "inline blocks") become interspersed with data blocks, and as a side effect we can no longer rely on the difference between block offsets to determine data block length, as it was done in version 1.

HFile is a low-level file format by design, and it should not deal with application-specific details such as Bloom filters, which are handled at StoreFile level. Therefore, we call Bloom filter blocks in an HFile "inline" blocks. We also supply HFile with an interface to write those inline blocks.

Another format modification aimed at reducing the region server startup time is to use a contiguous "load-on-open" section that has to be loaded in memory at the time an HFile is being opened. Currently, as an HFile opens, there are separate seek operations to read the trailer, data/meta indexes, and file info. To read the Bloom filter, there are two more seek operations for its "data" and "meta" portions. In version 2, we seek once to read the trailer and seek again to read everything else we need to open the file from a contiguous block.

Overview of Version 2

The version of HBase introducing the above features reads both version 1 and 2 HFiles, but only writes version 2 HFiles. A version 2 HFile is structured as follows:



Unified version 2 block format

In the version 2 every block in the data section contains the following fields:

1. 8 bytes: Block type, a sequence of bytes equivalent to version 1's "magic records". Supported block types are:
   - DATA – data blocks
   - LEAF_INDEX – leaf-level index blocks in a multi-level-block-index
   - BLOOM_CHUNK – Bloom filter chunks
   - META – meta blocks (not used for Bloom filters in version 2 anymore)
   - INTERMEDIATE_INDEX – intermediate-level index blocks in a multi-level blockindex
   - ROOT_INDEX – root-level index blocks in a multi-level block index
   - FILE_INFO – the "file info" block, a small key-value map of metadata
   - BLOOM_META – a Bloom filter metadata block in the load-on-open section

- TRAILER – a fixed-size file trailer. As opposed to the above, this is not an HFile v2 block but a fixed-size (for each HFile version) data structure
- INDEX_V1 – this block type is only used for legacy HFile v1 block

2. Compressed size of the block's data, not including the header (int).
   Can be used for skipping the current data block when scanning HFile data.
3. Uncompressed size of the block's data, not including the header (int)
   This is equal to the compressed size if the compression algorithm is NONE
4. File offset of the previous block of the same type (long)
   Can be used for seeking to the previous data/index block
5. Compressed data (or uncompressed data if the compression algorithm is NONE).

The above format of blocks is used in the following HFile sections:

Scanned block section

The section is named so because it contains all data blocks that need to be read when an HFile is scanned sequentially. Also contains Leaf index blocks and Bloom chunk blocks.

Non-scanned block section

This section still contains unified-format v2 blocks but it does not have to be read when doing a sequential scan. This section contains "meta" blocks and intermediate-level index blocks.

We are supporting "meta" blocks in version 2 the same way they were supported in version 1, even though we do not store Bloom filter data in these blocks anymore.

Block index in version 2

There are three types of block indexes in HFile version 2, stored in two different formats (root and non-root):

1. Data index — version 2 multi-level block index, consisting of:
   - Version 2 root index, stored in the data block index section of the file
   - Optionally, version 2 intermediate levels, stored in the non-root format in the data index section of the file. Intermediate levels can only be present if leaf level blocks are present
   - Optionally, version 2 leaf levels, stored in the non-root format inline with data blocks

2. Meta index — version 2 root index format only, stored in the meta index section of the file
3. Bloom index — version 2 root index format only, stored in the "load-on-open" section as part of Bloom filter metadata.

Root block index format in version 2

This format applies to:

1. Root level of the version 2 data index
2. Entire meta and Bloom indexes in version 2, which are always single-level.

A version 2 root index block is a sequence of entries of the following format, similar to entries of a version 1 block index, but storing on-disk size instead of uncompressed size.

1. Offset (long)
   This offset may point to a data block or to a deeper-level index block.
2. On-disk size (int)
3. Key (a serialized byte array stored using Bytes.writeByteArray)
4. Key (VInt)
5. Key bytes

A single-level version 2 block index consists of just a single root index block. To read a root index block of version 2, one needs to know the number of entries. For the data index and the meta index the number of entries is stored in the trailer, and for the Bloom index it is stored in the compound Bloom filter metadata.

For a multi-level block index we also store the following fields in the root index block in the load-on-open section of the HFile, in addition to the data structure described above:

- Middle leaf index block offset
- Middle leaf block on-disk size (meaning the leaf index block containing the reference to the "middle" data block of the file)
- The index of the mid-key (defined below) in the middle leaf-level block.

These additional fields are used to efficiently retrieve the mid-key of the HFile used in HFile splits, which we define as the first key of the block with a zero-based index of (n – 1)

/ 2, if the total number of blocks in the HFile is n. This definition is consistent with how the mid-key was determined in HFile version 1, and is reasonable in general, because blocks are likely to be the same size on average, but we don't have any estimates on individual key/value pair sizes.

When writing a version 2 HFile, the total number of data blocks pointed to by every leaf-level index block is kept track of. When we finish writing and the total number of leaf-level blocks is determined, it is clear which leaf-level block contains the mid-key, and the fields listed above are computed. When reading the HFile and the mid-key is requested, we retrieve the middle leaf index block (potentially from the block cache) and get the mid-key value from the appropriate position inside that leaf block.

Non-root block index format in version 2

This format applies to intermediate-level and leaf index blocks of a version 2 multi-level data block index. Every non-root index block is structured as follows.

1. numEntries: the number of entries (int).
2. entryOffsets: the "secondary index" of offsets of entries in the block, to facilitate a quick binary search on the key (`numEntries + 1` int values). The last value is the total length of all entries in this index block. For example, in a non-root index block with entry sizes 60, 80, 50 the "secondary index" will contain the following int array: `{0, 60, 140, 190}`.
3. Entries. Each entry contains:
   - Offset of the block referenced by this entry in the file (long)
   - On-disk size of the referenced block (int)
   - Key. The length can be calculated from entryOffsets.

Bloom filters in version 2

In contrast with version 1, in a version 2 HFile Bloom filter metadata is stored in the load-on-open section of the HFile for quick startup.

- A compound Bloom filter.
- Bloom filter version = 3 (int). There used to be a DynamicByteBloomFilter class that had the Bloom filter version number 2

- The total byte size of all compound Bloom filter chunks (long)
- Number of hash functions (int)
- Type of hash functions (int)
- The total key count inserted into the Bloom filter (long)
- The maximum total number of keys in the Bloom filter (long)
- The number of chunks (int)
- Comparator class used for Bloom filter keys, a UTF-8 encoded string stored using Bytes.writeByteArray
- Bloom block index in the version 2 root block index format

File Info format in versions 1 and 2

The file info block is a serialized map from byte arrays to byte arrays, with the following keys, among others. StoreFile-level logic adds more keys to this.

Key	Description
hfile.LASTKEY	The last key of the file (byte array)
hfile.AVG_KEY_LEN	The average key length in the file (int)
hfile.AVG_VALUE_LEN	The average value length in the file (int)

In version 2, we did not change the file format, but we moved the file info to the final section of the file, which can be loaded as one block when the HFile is being opened.

Also, we do not store the comparator in the version 2 file info anymore. Instead, we store it in the fixed file trailer. This is because we need to know the comparator at the time of parsing the load-on-open section of the HFile.

Fixed file trailer format differences between versions 1 and 2

The following table shows common and different fields between fixed file trailers in versions 1 and 2. Note that the size of the trailer is different depending on the version, so it is "fixed" only within one version. However, the version is always stored as the last four-byte integer in the file.

Differences between HFile Versions 1 and 2

Version 1	Version 2
	File info offset (long)
Data index offset (long)	loadOnOpenOffset (long) *The offset of the sectio n that we need to load when opening the file.*
	Number of data index entries (int)
metaIndexOffset (long) *This field is not being use d by the version 1 reader, so we removed it from version 2.*	uncompressedDataIndexSize (long) *The total un compressed size of the whole data block index, i ncluding root-level, intermediate-level, and leaf-l evel blocks.*
	Number of meta index entries (int)
	Total uncompressed bytes (long)
numEntries (int)	numEntries (long)
Compression codec: 0 = LZO, 1 = GZ, 2 = NONE (int)	Compression codec: 0 = LZO, 1 = GZ, 2 = NONE (int)
	The number of levels in the data block index (int)
	firstDataBlockOffset (long) *The offset of the first data block. Used when scanning.*
	lastDataBlockEnd (long) *The offset of the first by te after the last key/value data block. We don't need to go beyond this offset when scanning.*
Version: 1 (int)	Version: 2 (int)

getShortMidpointKey (an optimization for data index block)

Note: this optimization was introduced in HBase 0.95+

HFiles contain many blocks that contain a range of sorted Cells. Each cell has a key. To save IO when reading Cells, the HFile also has an index that maps a Cell's start key to the offset of the beginning of a particular block. Prior to this optimization, HBase would use the key of the first cell in each data block as the index key.

In HBASE-7845, we generate a new key that is lexicographically larger than the last key of the previous block and lexicographically equal or smaller than the start key of the current block. While actual keys can potentially be very long, this "fake key" or "virtual key" can

be much shorter. For example, if the stop key of previous block is "the quick brown fox", the start key of current block is "the who", we could use "the r" as our virtual key in our hfile index.

There are two benefits to this:

- having shorter keys reduces the hfile index size, (allowing us to keep more indexes in memory), and
- using something closer to the end key of the previous block allows us to avoid a potential extra IO when the target key lives in between the "virtual key" and the key of the first element in the target block.

This optimization (implemented by the getShortMidpointKey method) is inspired by LevelDB's ByteWiseComparatorImpl::FindShortestSeparator() and FindShortSuccessor().

HBase File Format with Security Enhancements (version 3)

Note: this feature was introduced in HBase 0.98

Motivation

Version 3 of HFile makes changes needed to ease management of encryption at rest and cell-level metadata (which in turn is needed for cell-level ACLs and cell-level visibility labels). For more information see hbase.encryption.server, hbase.tags, hbase.accesscontrol.configuration, and hbase.visibility.labels.

Overview

The version of HBase introducing the above features reads HFiles in versions 1, 2, and 3 but only writes version 3 HFiles. Version 3 HFiles are structured the same as version 2 HFiles. For more information see hfilev2.overview.

File Info Block in Version 3

Version 3 added two additional pieces of information to the reserved keys in the file info block.

Key	Description
hfile.MAX_TAGS_LEN	The maximum number of bytes needed to store t he serialized tags for any single cell in this hfile (i nt)
hfile.TAGS_COMPRESSED	Does the block encoder for this hfile compress ta gs? (boolean). Should only be present if hfile.MA X_TAGS_LEN is also present.

When reading a Version 3 HFile the presence of `MAX_TAGS_LEN` is used to determine how to deserialize the cells within a data block. Therefore, consumers must read the file's info block prior to reading any data blocks.

When writing a Version 3 HFile, HBase will always include `MAX_TAGS_LEN` when flushing the memstore to underlying filesystem.

When compacting extant files, the default writer will omit `MAX_TAGS_LEN` if all of the files selected do not themselves contain any cells with tags.

See compaction for details on the compaction file selection algorithm.

Data Blocks in Version 3

Within an HFile, HBase cells are stored in data blocks as a sequence of KeyValues (see hfilev1.overview, or Lars George's excellent introduction to HBase Storage). In version 3, these KeyValue optionally will include a set of 0 or more tags:

	Version 1 & 2, Version 3 witho ut MAX_TAGS_LEN	Version 3 with MAX_TAGS_LE N
Key Length (4 bytes)	✓	✓
Value Length (4 bytes)	✓	✓
Key bytes (variable)	✓	✓
Value bytes (variable)	✓	✓
Tags Length (2 bytes)		✓
Tags bytes (variable)		✓

If the info block for a given HFile contains an entry for `MAX_TAGS_LEN` each cell will have the length of that cell's tags included, even if that length is zero. The actual tags are stored as a sequence of tag length (2 bytes), tag type (1 byte), tag bytes (variable). The format an individual tag's bytes depends on the tag type.

Note that the dependence on the contents of the info block implies that prior to reading any data blocks you must first process a file's info block. It also implies that prior to writing a data block you must know if the file's info block will include `MAX_TAGS_LEN`.

Fixed File Trailer in Version 3

The fixed file trailers written with HFile version 3 are always serialized with protocol buffers. Additionally, it adds an optional field to the version 2 protocol buffer named encryption_key. If HBase is configured to encrypt HFiles this field will store a data encryption key for this particular HFile, encrypted with the current cluster master key using AES. For more information see hbase.encryption.server.

Other Information About HBase

HBase Videos

- Introduction to HBase by Todd Lipcon (Chicago Data Summit 2011).
- Building Real Time Services at Facebook with HBase by Jonathan Gray (Berlin buzzwords 2011)

HBase Presentations (Slides)

Advanced HBase Schema Design by Lars George (Hadoop World 2011).

Introduction to HBase by Todd Lipcon (Chicago Data Summit 2011).

Getting The Most From Your HBase Install by Ryan Rawson, Jonathan Gray (Hadoop World 2009).

HBase Papers

BigTable by Google (2006).

HBase and HDFS Locality by Lars George (2010).

No Relation: The Mixed Blessings of Non-Relational Databases by Ian Varley (2009).

HBase Sites

Cloudera's HBase Blog has a lot of links to useful HBase information.

CAP Confusion is a relevant entry for background information on distributed storage systems.

HBase RefCard from DZone.

HBase Books

HBase: The Definitive Guide by Lars George.

Hadoop Books

Hadoop: The Definitive Guide by Tom White.

HBase History

- 2006: BigTable paper published by Google.
- 2006 (end of year): HBase development starts.
- 2008: HBase becomes Hadoop sub-project.
- 2010: HBase becomes Apache top-level project.

Apache Software Foundation

ASF Development Process

See the Apache Development Process page for all sorts of information on how the ASF is structured (e.g., PMC, committers, contributors), to tips on contributing and getting involved, and how open-source works at ASF.

ASF Board Reporting

Once a quarter, each project in the ASF portfolio submits a report to the ASF board. This is done by the HBase project lead and the committers. See ASF board reporting for more information.

Apache HBase Orca

Default:



For dark theme:



An Orca is the Apache HBase mascot. See NOTICES.txt. Our Orca logo we got here: http://www.vectorfree.com/jumping-orca It is licensed Creative Commons Attribution 3.0. See https://creativecommons.org/licenses/by/3.0/us/ We changed the logo by stripping the colored background, inverting it and then rotating it some.

The 'official' HBase color is "International Orange (Engineering)", the color of the Golden Gate bridge in San Francisco and for space suits used by NASA.

Our 'font' is Bitsumishi.

0.95 RPC Specification

In 0.95, all client/server communication is done with protobuf'ed Messages rather than with Hadoop Writables. Our RPC wire format therefore changes. This document describes the client/server request/response protocol and our new RPC wire-format.

For what RPC is like in 0.94 and previous, see Benoît/Tsuna's Unofficial Hadoop / HBase RPC protocol documentation. For more background on how we arrived at this spec., see HBase RPC: WIP

Goals

1. A wire-format we can evolve
2. A format that does not require our rewriting server core or radically changing its current architecture (for later).

TODO

1. List of problems with currently specified format and where we would like to go in a version2, etc. For example, what would we have to change if anything to move server async or to support streaming/chunking?
2. Diagram on how it works
3. A grammar that succinctly describes the wire-format. Currently we have these words and the content of the rpc protobuf idl but a grammar for the back and forth would help with groking rpc. Also, a little state machine on client/server interactions would help with understanding (and ensuring correct implementation).

RPC

The client will send setup information on connection establish. Thereafter, the client invokes methods against the remote server sending a protobuf Message and receiving a protobuf Message in response. Communication is synchronous. All back and forth is preceded by an int that has the total length of the request/response. Optionally, Cells(KeyValues) can be passed outside of protobufs in follow-behind Cell blocks

(because we can't protobuf megabytes of KeyValues or Cells). These CellBlocks are encoded and optionally compressed.

For more detail on the protobufs involved, see the RPC.proto file in master.

Connection Setup

Client initiates connection.

Client

On connection setup, client sends a preamble followed by a connection header.

`<preamble>`

```
<MAGIC 4 byte integer> <1 byte RPC Format Version> <1 byte auth type>
```

We need the auth method spec. here so the connection header is encoded if auth enabled. E.g.: HBas0x000×50 — 4 bytes of MAGIC — `HBas' — plus one-byte of version, 0 in this case, and one byte, 0x50 (SIMPLE). of an auth type.

`<Protobuf ConnectionHeader Message>`

Has user info, and "protocol", as well as the encoders and compression the client will use sending CellBlocks. CellBlock encoders and compressors are for the life of the connection. CellBlock encoders implement org.apache.hadoop.hbase.codec.Codec. CellBlocks may then also be compressed. Compressors implement org.apache.hadoop.io.compress.CompressionCodec. This protobuf is written using writeDelimited so is prefaced by a pb varint with its serialized length

Server

After client sends preamble and connection header, server does NOT respond if successful connection setup. No response means server is READY to accept requests and to give out response. If the version or authentication in the preamble is not agreeable or the server has trouble parsing the preamble, it will throw a org.apache.hadoop.hbase.ipc.FatalConnectionException explaining the error and will then disconnect. If the client in the connection header — i.e. the protobuf'd Message that comes after the connection preamble — asks for a Service the server does not support or a codec the server does not have, again we throw a FatalConnectionException with explanation.

Request

After a Connection has been set up, client makes requests. Server responds.

A request is made up of a protobuf RequestHeader followed by a protobuf Message parameter. The header includes the method name and optionally, metadata on the optional CellBlock that may be following. The parameter type suits the method being invoked: i.e. if we are doing a getRegionInfo request, the protobuf Message param will be an instance of GetRegionInfoRequest. The response will be a GetRegionInfoResponse. The CellBlock is optionally used ferrying the bulk of the RPC data: i.e. Cells/KeyValues.

Request Parts

`<Total Length>`

The request is prefaced by an int that holds the total length of what follows.

`<Protobuf RequestHeader Message>`

Will have call.id, trace.id, and method name, etc. including optional Metadata on the Cell block IFF one is following. Data is protobuf'd inline in this pb Message or optionally comes in the following CellBlock

`<Protobuf Param Message>`

If the method being invoked is getRegionInfo, if you study the Service descriptor for the client to regionserver protocol, you will find that the request sends a GetRegionInfoRequest protobuf Message param in this position.

`<CellBlock>`

An encoded and optionally compressed Cell block.

Response

Same as Request, it is a protobuf ResponseHeader followed by a protobuf Message response where the Message response type suits the method invoked. Bulk of the data may come in a following CellBlock.

Response Parts

`<Total Length>`

The response is prefaced by an int that holds the total length of what follows.

`<Protobuf ResponseHeader Message>`

Will have call.id, etc. Will include exception if failed processing. Optionally includes metadata on optional, IFF there is a CellBlock following.

`<Protobuf Response Message>`

Return or may be nothing if exception. If the method being invoked is getRegionInfo, if you study the Service descriptor for the client to regionserver protocol, you will find that the response sends a GetRegionInfoResponse protobuf Message param in this position.

`<CellBlock>`

An encoded and optionally compressed Cell block.

Exceptions

There are two distinct types. There is the request failed which is encapsulated inside the response header for the response. The connection stays open to receive new requests. The second type, the FatalConnectionException, kills the connection.

Exceptions can carry extra information. See the ExceptionResponse protobuf type. It has a flag to indicate do-no-retry as well as other miscellaneous payload to help improve client responsiveness.

CellBlocks

These are not versioned. Server can do the codec or it cannot. If new version of a codec with say, tighter encoding, then give it a new class name. Codecs will live on the server for all time so old clients can connect.

Notes

Constraints

In some part, current wire-format — i.e. all requests and responses preceded by a length — has been dictated by current server non-async architecture.

One fat pb request or header+param

We went with pb header followed by pb param making a request and a pb header followed by pb response for now. Doing header+param rather than a single protobuf Message with both header and param content:

1. Is closer to what we currently have
2. Having a single fat pb requires extra copying putting the already pb'd param into the body of the fat request pb (and same making result)
3. We can decide whether to accept the request or not before we read the param; for example, the request might be low priority. As is, we read header+param in one go as server is currently implemented so this is a TODO.

The advantages are minor. If later, fat request has clear advantage, can roll out a v2 later.

RPC Configurations

CellBlock Codecs

To enable a codec other than the default `KeyValueCodec`, set `hbase.client.rpc.codec` to the name of the Codec class to use. Codec must implement hbase's `Codec` Interface. After connection setup, all passed cellblocks will be sent with this codec. The server will return cellblocks using this same codec as long as the codec is on the servers' CLASSPATH (else you will get `UnsupportedCellCodecException`).

To change the default codec, set `hbase.client.default.rpc.codec`.

To disable cellblocks completely and to go pure protobuf, set the default to the empty String and do not specify a codec in your Configuration. So, set `hbase.client.default.rpc.codec` to the empty string and do not set `hbase.client.rpc.codec`. This will cause the client to connect to the server with no codec specified. If a server sees no codec, it will return all responses in pure protobuf. Running pure protobuf all the time will be slower than running with cellblocks.

Compression

Uses hadoop's compression codecs. To enable compressing of passed CellBlocks, set `hbase.client.rpc.compressor` to the name of the Compressor to use. Compressor must implement Hadoop's CompressionCodec Interface. After connection setup, all passed

cellblocks will be sent compressed. The server will return cellblocks compressed using this same compressor as long as the compressor is on its CLASSPATH (else you will get `UnsupportedCompressionCodecException`).

Known Incompatibilities Among HBase Versions

HBase 2.0 Incompatible Changes

This appendix describes incompatible changes from earlier versions of HBase against HBase 2.0. This list is not meant to be wholly encompassing of all possible incompatibilities. Instead, this content is intended to give insight into some obvious incompatibilities which most users will face coming from HBase 1.x releases.

List of Major Changes for HBase 2.0

- HBASE-1912- HBCK is an HBase database checking tool for capturing the inconsistency. As an HBase administrator, you should not use HBase version 1.0 hbck tool to check the HBase 2.0 database. Doing so will break the database and throw an exception error.
- HBASE-16189 and HBASE-18945- You cannot open the HBase 2.0 hfiles through HBase 1.0 version. If you are an admin or an HBase user who is using HBase version 1.x, you must first do a rolling upgrade to the latest version of HBase 1.x and then upgrade to HBase 2.0.
- HBASE-18240 - Changed the ReplicationEndpoint Interface. It also introduces a new hbase-third party 1.0 that packages all the third party utilities, which are expected to run in the hbase cluster.

Coprocessor API changes

- HBASE-16769 - Deprecated PB references from MasterObserver and RegionServerObserver.
- HBASE-17312 - [JDK8] Use default method for Observer Coprocessors. The interface classes of BaseMasterAndRegionObserver, BaseMasterObserver, BaseRegionObserver, BaseRegionServerObserver and BaseWALObserver uses JDK8's 'default' keyword to provide empty and no-op implementations.
- Interface HTableInterface HBase 2.0 introduces following changes to the methods listed below:

interface CoprocessorEnvironment changes

Change	Result
Abstract method getTable (TableName) has been removed.	A client program may be interrupted by NoSuchMethodError exception.
Abstract method getTable (TableName, ExecutorService) has been removed.	A client program may be interrupted by NoSuchMethodError exception.

- Public Audience

The following tables describes the coprocessor changes.

class CoprocessorRpcChannel

Change	Result
This class has become interface.	A client program may be interrupted by IncompatibleClassChangeError or InstantiationError exception depending on the usage of this class.

Class CoprocessorHost<E>

Classes that were Audience Private but were removed.

Change	Result
Type of field coprocessors has been changed from java.util.SortedSet<E> to org.apache.hadoop.hbase.util.SortedList<E>.	A client program may be interrupted by NoSuchFieldError exception.

MasterObserver

HBase 2.0 introduces following changes to the MasterObserver interface.

interface MasterObserver

Change	Result
Abstract method voidpostCloneSnapshot (ObserverContext<MasterCoprocessorEnvironment>, HBaseProtos.SnapshotDescription, HTableDescriptor) has been removed from this interface.	A client program may be interrupted by NoSuchMethodError exception.

Change	Result
Abstract method voidpostCreateTable (ObserverContext<MasterCoprocessorEnvironment>, HTableDescriptor, HRegionInfo[ ]) has been removed from this interface.	A client program may be interrupted by NoSuchMethodErrorexception.
Abstract method voidpostDeleteSnapshot (ObserverContext<MasterCoprocessorEnvironment>, HBaseProtos.SnapshotDescription) has been removed from this interface.	A client program may be interrupted by NoSuchMethodErrorexception.
Abstract method voidpostGetTableDescriptors (ObserverContext<MasterCoprocessorEnvironment>, List<HTableDescriptor>) has been removed from this interface.	A client program may be interrupted by NoSuchMethodErrorexception.
Abstract method voidpostModifyTable (ObserverContext<MasterCoprocessorEnvironment>, TableName, HTableDescriptor) has been removed from this interface.	A client program may be interrupted by NoSuchMethodErrorexception.
Abstract method voidpostRestoreSnapshot (ObserverContext<MasterCoprocessorEnvironment>, HBaseProtos.SnapshotDescription, HTableDescriptor) has been removed from this interface.	A client program may be interrupted by NoSuchMethodErrorexception.
Abstract method voidpostSnapshot (ObserverContext<MasterCoprocessorEnvironment>, HBaseProtos.SnapshotDescription, HTableDescriptor) has been removed from this interface.	A client program may be interrupted by NoSuchMethodErrorexception.
Abstract method voidpreCloneSnapshot (ObserverContext<MasterCoprocessorEnvironment>, HBaseProtos.SnapshotDescription, HTableDescriptor) has been removed from this interface.	A client program may be interrupted by NoSuchMethodErrorexception.
Abstract method voidpreCreateTable (ObserverContext<MasterCoprocessorEnvironment>, HTableDescriptor, HRegionInfo[ ]) has been removed from this interface.	A client program may be interrupted by NoSuchMethodErrorexception.
Abstract method voidpreDeleteSnapshot (ObserverContext<MasterCoprocessorEnvironment>, HBaseProtos.SnapshotDescription) has been removed from this interface.	A client program may be interrupted by NoSuchMethodErrorexception.

Change	Result
Abstract method voidpreGetTableDescriptors (ObserverContext<MasterCoprocessorEnvironment>, List<TableName>, List<HTableDescriptor>) has been removed from this interface.	A client program may be interrupted by NoSuchMethodErrorexception.
Abstract method voidpreModifyTable (ObserverContext<MasterCoprocessorEnvironment>, TableName, HTableDescriptor) has been removed from this interface.	A client program may be interrupted by NoSuchMethodErrorexception.
Abstract method voidpreRestoreSnapshot (ObserverContext<MasterCoprocessorEnvironment>, HBaseProtos.SnapshotDescription, HTableDescriptor) has been removed from this interface.	A client program may be interrupted by NoSuchMethodErrorexception.
Abstract method voidpreSnapshot (ObserverContext<MasterCoprocessorEnvironment>, HBaseProtos.SnapshotDescription, HTableDescriptor) has been removed from this interface.	A client program may be interrupted by NoSuchMethodErrorexception.

RegionObserver

HBase 2.0 introduces following changes to the RegionObserver interface.

interface RegionObserver

Change	Result
Abstract method voidpostCloseRegionOperation (ObserverContext<RegionCoprocessorEnvironment>, HRegion.Operation) has been removed from this interface.	A client program may be interrupted by NoSuchMethodError exception.
Abstract method voidpostCompactSelection (ObserverContext<RegionCoprocessorEnvironment>, Store, ImmutableList<StoreFile>) has been removed from this interface.	A client program may be interrupted by NoSuchMethodError exception.
Abstract method voidpostCompactSelection (ObserverContext<RegionCoprocessorEnvironment>, Store, ImmutableList<StoreFile>, CompactionRequest) has been removed from this interface.	A client program may be interrupted by NoSuchMethodError exception.

Change	Result
Abstract method voidpostGetClosestRowBefore (ObserverContext<RegionCoprocessorEnvironment>, byte[ ], byte[ ], Result) has been removed from this interface.	A client program may be interrupted by NoSuchMethodError exception.
Abstract method DeleteTrackerpostInstantiateDeleteTracker (ObserverContext<RegionCoprocessorEnvironment>, DeleteTracker) has been removed from this interface.	A client program may be interrupted by NoSuchMethodError exception.
Abstract method voidpostSplit (ObserverContext<RegionCoprocessorEnvironment>, HRegion, HRegion) has been removed from this interface.	A client program may be interrupted by NoSuchMethodError exception.
Abstract method voidpostStartRegionOperation (ObserverContext<RegionCoprocessorEnvironment>, HRegion.Operation) has been removed from this interface.	A client program may be interrupted by NoSuchMethodError exception.
Abstract method StoreFile.ReaderpostStoreFileReaderOpen (ObserverContext<RegionCoprocessorEnvironment>, FileSystem, Path, FSDataInputStreamWrapper, long, CacheConfig, Reference, StoreFile.Reader) has been removed from this interface.	A client program may be interrupted by NoSuchMethodError exception.
Abstract method voidpostWALRestore (ObserverContext<RegionCoprocessorEnvironment>, HRegionInfo, HLogKey, WALEdit) has been removed from this interface.	A client program may be interrupted by NoSuchMethodError exception.
Abstract method InternalScannerpreFlushScannerOpen (ObserverContext<RegionCoprocessorEnvironment>, Store, KeyValueScanner, InternalScanner) has been removed from this interface.	A client program may be interrupted by NoSuchMethodError exception.
Abstract method voidpreGetClosestRowBefore (ObserverContext<RegionCoprocessorEnvironment>, byte[ ], byte[ ], Result) has been removed from this interface.	A client program may be interrupted by NoSuchMethodError exception.

Change	Result
Abstract method StoreFile.ReaderpreStoreFileReaderOpen (ObserverContext<RegionCoprocessorEnvironment>, FileSystem, Path, FSDataInputStreamWrapper, long, CacheConfig, Reference, StoreFile.Reader) has been removed from this interface.	A client program may be interrupted by NoSuchMethodError exception.
Abstract method voidpreWALRestore (ObserverContext<RegionCoprocessorEnvironment>, HRegionInfo, HLogKey, WALEdit) has been removed from this interface.	A client program may be interrupted by NoSuchMethodError exception.

WALObserver

HBase 2.0 introduces following changes to the WALObserver interface.

interface WALObserver

Change	Result
Abstract method voidpostWALWrite (ObserverContext<WALCoprocessorEnvironment>, HRegionInfo, HLogKey, WALEdit) has been removed from this interface.	A client program may be interrupted by NoSuchMethodError exception.
Abstract method booleanpreWALWrite (ObserverContext<WALCoprocessorEnvironment>, HRegionInfo, HLogKey, WALEdit) has been removed from this interface.	A client program may be interrupted by NoSuchMethodError exception.

Miscellaneous

HBase 2.0 introduces changes to the following classes:

hbase-server-1.0.0.jar, OnlineRegions.class package org.apache.hadoop.hbase.regionserver

OnlineRegions.getFromOnlineRegions (String p1) [abstract] : HRegion
org/apache/hadoop/hbase/regionserver/OnlineRegions.getFromOnlineRegions: (Ljava/lang/String;)Lorg/apache/hadoop/hbase/regionserver/HRegion;

Change	Result
Return value type has been changed from Regio n to Region.	This method has been removed because the retu rn type is part of the method signature. A client program may be interrupted by NoSuchMethodE rror exception.

hbase-server-1.0.0.jar, RegionCoprocessorEnvironment.class package org.apache.hadoop.hbase.coprocessor

RegionCoprocessorEnvironment.getRegion ( ) [abstract] : HRegion
org/apache/hadoop/hbase/coprocessor/RegionCoprocessorEnvironment.getRegion:()Lorg/apache/hadoop/hbase/regionserver/HRegion;

Change	Result
Return value type has been changed from org.ap ache.hadoop.hbase.regionserver.HRegion to org. apache.hadoop.hbase.regionserver.Region.	This method has been removed because the retu rn type is part of the method signature. A client program may be interrupted by NoSuchMethodE rror exception.

hbase-server-1.0.0.jar, RegionCoprocessorHost.class package org.apache.hadoop.hbase.regionserver

RegionCoprocessorHost.postAppend (Append append, Result result) : void
org/apache/hadoop/hbase/regionserver/RegionCoprocessorHost.postAppend:(Lorg/apache/hadoop/hbase/client/Append;Lorg/apache/hadoop/hbase/client/Result;)V

Change	Result
Return value type has been changed from void t o org.apache.hadoop.hbase.client.Result.	This method has been removed because the retu rn type is part of the method signature. A client program may be interrupted by NoSuchMethodE rror exception.

RegionCoprocessorHost.preStoreFileReaderOpen (FileSystem fs, Path p, FSDataInputStreamWrapper in, long size,CacheConfig cacheConf, Reference r) : StoreFile.Reader
org/apache/hadoop/hbase/regionserver/RegionCoprocessorHost.preStoreFileReaderOpen:(Lorg/apache/hadoop/fs/FileSystem;Lorg/apache/hadoop/fs/Path;Lorg/apache/hadoop/hb

ase/io/FSDataInputStreamWrapper;JLorg/apache/hadoop/hbase/io/hfile/CacheConfig;Lorg/apache/hadoop/hbase/io/Reference;)Lorg/apache/hadoop/hbase/regionserver/StoreFile$Reader;

Change	Result
Return value type has been changed from StoreFile.Reader to StoreFileReader.	This method has been removed because the return type is part of the method signature. A client program may be interrupted by NoSuchMethodError exception.

IPC

Scheduler changes:

1. Following methods became abstract:

package org.apache.hadoop.hbase.ipc

class RpcScheduler

Change	Result
Abstract method void dispatch (CallRunner) has been removed from this class.	A client program may be interrupted by NoSuchMethodError exception.

hbase-server-1.0.0.jar, RpcScheduler.class package org.apache.hadoop.hbase.ipc

RpcScheduler.dispatch (CallRunner p1) [abstract] : void 1

org/apache/hadoop/hbase/ipc/RpcScheduler.dispatch:
(Lorg/apache/hadoop/hbase/ipc/CallRunner;)V

Change	Result
Return value type has been changed from void to boolean.	This method has been removed because the return type is part of the method signature. A client program may be interrupted by NoSuchMethodError exception.

2. Following abstract methods have been removed:

interface PriorityFunction

Change	Result
Abstract method longgetDeadline (RPCProtos.RequestHeader, Message) has been removed from this interface.	A client program may be interrupted by NoSuchMethodError exception.
Abstract method int getPriority (RPCProtos.RequestHeader, Message) has been removed from this interface.	A client program may be interrupted by NoSuchMethodError exception.

Server API changes:

class RpcServer

Change	Result
Type of field CurCall has been changed from java.lang.ThreadLocal<RpcServer.Call> to java.lang.ThreadLocal<RpcCall>.	A client program may be interrupted by NoSuchFieldError exception.
This class became abstract.	A client program may be interrupted by InstantiationError exception.
Abstract method int getNumOpenConnections ( ) has been added to this class.	This class became abstract and a client program may be interrupted by InstantiationError exception.
Field callQueueSize of type org.apache.hadoop.hbase.util.Counter has been removed from this class.	A client program may be interrupted by NoSuchFieldError exception.
Field connectionList of type java.util.List<RpcServer.Connection> has been removed from this class.	A client program may be interrupted by NoSuchFieldError exception.
Field maxIdleTime of type int has been removed from this class.	A client program may be interrupted by NoSuchFieldError exception.
Field numConnections of type int has been removed from this class.	A client program may be interrupted by NoSuchFieldError exception.
Field port of type int has been removed from this class.	A client program may be interrupted by NoSuchFieldError exception.

Change	Result
Field purgeTimeout of type long has been removed from this class.	A client program may be interrupted by NoSuchFieldError exception.
Field responder of type RpcServer.Responder has been removed from this class.	A client program may be interrupted by NoSuchFieldError exception.
Field socketSendBufferSize of type int has been removed from this class.	A client program may be interrupted by NoSuchFieldError exception.
Field thresholdIdleConnections of type int has been removed from this class.	A client program may be interrupted by NoSuchFieldError exception.

Following abstract method has been removed:

Change	Result
Abstract method Pair<Message,CellScanner>call(BlockingService, Descriptors.MethodDescriptor, Message, CellScanner, long, MonitoredRPCHandler) has been removed from this interface.	A client program may be interrupted by NoSuchMethodError exception.

Replication and WAL changes:

HBASE-18733: WALKey has been purged completely in HBase 2.0. Following are the changes to the WALKey:

classWALKey

Change	Result
Access level of field clusterIds has been changed from protected to private.	A client program may be interrupted by IllegalAccessError exception.
Access level of field compressionContext has been changed from protected to private.	A client program may be interrupted by IllegalAccessError exception.
Access level of field encodedRegionName has been changed from protected to private.	A client program may be interrupted by IllegalAccessError exception.
Access level of field tablename has been changed from protectedto private.	A client program may be interrupted by IllegalAccessError exception.
Access level of field writeTime has been changed from protectedto private.	A client program may be interrupted by IllegalAccessError exception.

Following fields have been removed:

Change	Result
Field LOG of type org.apache.commons.logging.Log has been removed from this class.	A client program may be interrupted by NoSuchFieldError exception.
Field VERSION of type WALKey.Version has been removed from this class.	A client program may be interrupted by NoSuchFieldError exception.
Field logSeqNum of type long has been removed from this class.	A client program may be interrupted by NoSuchFieldError exception.

Following are the changes to the WALEdit.class: hbase-server-1.0.0.jar, WALEdit.class package org.apache.hadoop.hbase.regionserver.wal

WALEdit.getCompaction (Cell kv) [static] : WALProtos.CompactionDescriptor

org/apache/hadoop/hbase/regionserver/wal/WALEdit.getCompaction:(Lorg/apache/hadoop/hbase/Cell;)Lorg/apache/hadoop/hbase/protobuf/generated/WALProtos$CompactionDescriptor;

Change	Result
Return value type has been changed from org.apache.hadoop.hbase.protobuf.generated.WALProtos.CompactionDescriptor to org.apache.hadoop.hbase.shaded.protobuf.generated.WALProtos.CompactionDescriptor.	This method has been removed because the return type is part of the method signature. A client program may be interrupted by NoSuchMethodError exception.

WALEdit.getFlushDescriptor (Cell cell) [static] : WALProtos.FlushDescriptor

org/apache/hadoop/hbase/regionserver/wal/WALEdit.getFlushDescriptor:(Lorg/apache/hadoop/hbase/Cell;)Lorg/apache/hadoop/hbase/protobuf/generated/WALProtos$FlushDescriptor;

Change	Result
Return value type has been changed from org.apache.hadoop.hbase.protobuf.generated.WALProtos.FlushDescriptor to org.apache.hadoop.hbase.shaded.protobuf.generated.WALProtos.FlushDescriptor.	This method has been removed because the return type is part of the method signature. A client program may be interrupted by NoSuchMethodError exception.

WALEdit.getRegionEventDescriptor (Cell cell) [static] : WALProtos.RegionEventDescriptor

org/apache/hadoop/hbase/regionserver/wal/WALEdit.getRegionEventDescriptor: (Lorg/apache/hadoop/hbase/Cell;)Lorg/apache/hadoop/hbase/protobuf/generated/WALProtos$RegionEventDescriptor;

Change	Result
Return value type has been changed from org.apache.hadoop.hbase.protobuf.generated.WALProtos.RegionEventDescriptor to org.apache.hadoop.hbase.shaded.protobuf.generated.WALProtos.RegionEventDescriptor.	This method has been removed because the return type is part of the method signature. A client program may be interrupted by NoSuchMethodError exception.

Following is the change to the WALKey.class: package org.apache.hadoop.hbase.wal

WALKey.getBuilder (WALCellCodec.ByteStringCompressor compressor) : WALProtos.WALKey.Builder

org/apache/hadoop/hbase/wal/WALKey.getBuilder: (Lorg/apache/hadoop/hbase/regionserver/wal/WALCellCodec$ByteStringCompressor;)Lorg/apache/hadoop/hbase/protobuf/generated/WALProtos$WALKey$Builder;

Change	Result
Return value type has been changed from org.apache.hadoop.hbase.protobuf.generated.WALProtos.WALKey.Builder to org.apache.hadoop.hbase.shaded.protobuf.generated.WALProtos.WALKey.Builder.	This method has been removed because the return type is part of the method signature. A client program may be interrupted by NoSuchMethodError exception.

Deprecated APIs or coprocessor:

HBASE-16769 - PB references from MasterObserver and RegionServerObserver has been removed.

Admin Interface API changes:

You cannot administer an HBase 2.0 cluster with an HBase 1.0 client that includes RelicationAdmin, ACC, Thrift and REST usage of Admin ops. Methods returning protobufs have been changed to return POJOs instead. pb is not used in the APIs anymore. Returns

have changed from void to Future for async methods. HBASE-18106 - Admin.listProcedures and Admin.listLocks were renamed to getProcedures and getLocks. MapReduce makes use of Admin doing following admin.getClusterStatus() to calcluate Splits.

- Thrift usage of Admin API: compact(ByteBuffer) createTable(ByteBuffer, List<ColumnDescriptor>) deleteTable(ByteBuffer) disableTable(ByteBuffer) enableTable(ByteBuffer) getTableNames() majorCompact(ByteBuffer)

- REST usage of Admin API: hbase-rest org.apache.hadoop.hbase.rest RootResource getTableList() TableName[] tableNames = servlet.getAdmin().listTableNames(); SchemaResource delete(UriInfo) Admin admin = servlet.getAdmin(); update(TableSchemaModel, boolean, UriInfo) Admin admin = servlet.getAdmin(); StorageClusterStatusResource get(UriInfo) ClusterStatus status = servlet.getAdmin().getClusterStatus(); StorageClusterVersionResource get(UriInfo) model.setVersion(servlet.getAdmin().getClusterStatus().getHBaseVersion()); TableResource exists() return servlet.getAdmin().tableExists(TableName.valueOf(table));

Following are the changes to the Admin interface:

interface Admin

Change	Result
Abstract method createTableAsync (HTableDescriptor, byte[ ][ ]) has been removed from this interface.	A client program may be interrupted by NoSuchMethodError exception.
Abstract method disableTableAsync (TableName) has been removed from this interface.	A client program may be interrupted by NoSuchMethodError exception.
Abstract method enableTableAsync (TableName) has been removed from this interface.	A client program may be interrupted by NoSuchMethodError exception.
Abstract method getCompactionState (TableName) has been removed from this interface.	A client program may be interrupted by NoSuchMethodError exception.
Abstract method getCompactionStateForRegion (byte[ ]) has been removed from this interface.	A client program may be interrupted by NoSuchMethodError exception.
Abstract method isSnapshotFinished (HBaseProtos.SnapshotDescription) has been removed from this interface.	A client program may be interrupted by NoSuchMethodError exception.

Change	Result
Abstract method snapshot (String, TableName, HBaseProtos.SnapshotDescription.Type) has been removed from this interface.	A client program may be interrupted by NoSuchMethodError exception.
Abstract method snapshot (HBaseProtos.SnapshotDescription) has been removed from this interface.	A client program may be interrupted by NoSuchMethodError exception.
Abstract method takeSnapshotAsync (HBaseProtos.SnapshotDescription) has been removed from this interface.	A client program may be interrupted by NoSuchMethodError exception.

Following are the changes to the Admin.class: hbase-client-1.0.0.jar, Admin.class package org.apache.hadoop.hbase.client

Admin.createTableAsync (HTableDescriptor p1, byte[ ][ ] p2) [abstract] : void 1

org/apache/hadoop/hbase/client/Admin.createTableAsync: (Lorg/apache/hadoop/hbase/HTableDescriptor;[[B)V

Change	Result
Return value type has been changed from void to java.util.concurrent.Future<java.lang.Void>.	This method has been removed because the return type is part of the method signature. A client program may be interrupted by NoSuchMethodError exception.

Admin.disableTableAsync (TableName p1) [abstract] : void 1

org/apache/hadoop/hbase/client/Admin.disableTableAsync: (Lorg/apache/hadoop/hbase/TableName;)V

Change	Result
Return value type has been changed from void to java.util.concurrent.Future<java.lang.Void>.	This method has been removed because the return type is part of the method signature. A client program may be interrupted by NoSuchMethodError exception.

Admin.enableTableAsync (TableName p1) [abstract] : void

org/apache/hadoop/hbase/client/Admin.enableTableAsync:
(Lorg/apache/hadoop/hbase/TableName;)V

Change	Result
Return value type has been changed from void to java.util.concurrent.Future<java.lang.Void>.	This method has been removed because the return type is part of the method signature. A client program may be interrupted by NoSuchMethodError exception.

Admin.getCompactionState (TableName p1) [abstract] : AdminProtos.GetRegionInfoResponse.CompactionState 1

org/apache/hadoop/hbase/client/Admin.getCompactionState:
(Lorg/apache/hadoop/hbase/TableName;)Lorg/apache/hadoop/hbase/protobuf/generated/AdminProtos$GetRegionInfoResponse$CompactionState;

Change	Result
Return value type has been changed from org.apache.hadoop.hbase.protobuf.generated.AdminProtos.GetRegionInfoResponse.CompactionState to CompactionState.	This method has been removed because the return type is part of the method signature. A client program may be interrupted by NoSuchMethodError exception.

Admin.getCompactionStateForRegion (byte[ ] p1) [abstract] : AdminProtos.GetRegionInfoResponse.CompactionState 1

org/apache/hadoop/hbase/client/Admin.getCompactionStateForRegion:
([B)Lorg/apache/hadoop/hbase/protobuf/generated/AdminProtos$GetRegionInfoResponse$CompactionState;

Change	Result
Return value type has been changed from org.apache.hadoop.hbase.protobuf.generated.AdminProtos.GetRegionInfoResponse.CompactionState to CompactionState.	This method has been removed because the return type is part of the method signature. A client program may be interrupted by NoSuchMethodError exception.

HTableDescriptor and HColumnDescriptor changes

HTableDescriptor and HColumnDescriptor has become interfaces and you can create it through Builders. HCD has become CFD. It no longer implements writable interface. package org.apache.hadoop.hbase

class HColumnDescriptor

Change	Result
Removed super-interface org.apache.hadoop.io.WritableComparable<HColumnDescriptor>.	A client program may be interrupted by NoSuchMethodError exception.

HColumnDescriptor in 1.0.0

```
@InterfaceAudience.Public
@InterfaceStability.Evolving
public class HColumnDescriptor implements WritableComparable<HColumnDescriptor> {
```

HColumnDescriptor in 2.0

```
@InterfaceAudience.Public
@Deprecated // remove it in 3.0
public class HColumnDescriptor implements ColumnFamilyDescriptor, Comparable<HCol
umnDescriptor> {
```

For META_TABLEDESC, the maker method had been deprecated already in HTD in 1.0.0. OWNER_KEY is still in HTD.

class HTableDescriptor

Change	Result
Removed super-interface org.apache.hadoop.io.WritableComparable<HTableDescriptor>.	A client program may be interrupted by NoSuchMethodError exception.
Field META_TABLEDESC of type HTableDescriptor has been removed from this class.	A client program may be interrupted by NoSuchFieldError exception.

hbase-client-1.0.0.jar, HTableDescriptor.class package org.apache.hadoop.hbase

HTableDescriptor.getColumnFamilies ( ) : HColumnDescriptor[ ]

org/apache/hadoop/hbase/HTableDescriptor.getColumnFamilies:()

[Lorg/apache/hadoop/hbase/HColumnDescriptor;

class HColumnDescriptor

Change	Result
Return value type has been changed from HColumnDescriptor[]to client.ColumnFamilyDescriptor[].	This method has been removed because the return type is part of the method signature. A client program may be interrupted by NoSuchMethodError exception.

HTableDescriptor.getCoprocessors ( ) : List<String>
org/apache/hadoop/hbase/HTableDescriptor.getCoprocessors:()Ljava/util/List;

Change	Result
Return value type has been changed from java.util.List<java.lang.String> to java.util.Collection.	This method has been removed because the return type is part of the method signature. A client program may be interrupted by NoSuchMethodError exception.

- HBASE-12990 MetaScanner is removed and it is replaced by MetaTableAccessor.

HTableWrapper changes:

hbase-server-1.0.0.jar, HTableWrapper.class package org.apache.hadoop.hbase.client

HTableWrapper.createWrapper (List<HTableInterface> openTables, TableName tableName, CoprocessorHost.Environment env, ExecutorService pool) [static] : HTableInterface 1

org/apache/hadoop/hbase/client/HTableWrapper.createWrapper:(Ljava/util/List;Lorg/apache/hadoop/hbase/TableName;Lorg/apache/hadoop/hbase/coprocessor/CoprocessorHost$Environment;Ljava/util/concurrent/ExecutorService;)Lorg/apache/hadoop/hbase/client/HTableInterface;

Change	Result
Return value type has been changed from HTableInterface to Table.	This method has been removed because the return type is part of the method signature. A client program may be interrupted by NoSuchMethodError exception.

- HBASE-12586: Delete all public HTable constructors and delete ConnectionManager#{delete,get}Connection.
- HBASE-9117: Remove HTablePool and all HConnection pooling related APIs.
- HBASE-13214: Remove deprecated and unused methods from HTable class Following are the changes to the Table interface:

interface Table

Change	Result
Abstract method batch (List<?>) has been removed from this interface.	A client program may be interrupted by NoSuchMethodError exception.
Abstract method batchCallback (List<?>, Batch.Callback<R>)has been removed from this interface.	A client program may be interrupted by NoSuchMethodError exception.
Abstract method getWriteBufferSize ( ) has been removed from this interface.	A client program may be interrupted by NoSuchMethodError exception.
Abstract method setWriteBufferSize (long) has been removed from this interface.	A client program may be interrupted by NoSuchMethodError exception.

Deprecated buffer methods in Table (in 1.0.1) and removed in 2.0.0

- HBASE-13298- Clarify if Table.{set|get}WriteBufferSize() is deprecated or not.
- LockTimeoutException and OperationConflictException classes have been removed.

class OperationConflictException

Change	Result
This class has been removed.	A client program may be interrupted by NoClassDefFoundErrorexception.

class class LockTimeoutException

Change	Result
This class has been removed.	A client program may be interrupted by NoClassDefFoundErrorexception.

Filter API changes:

Following methods have been removed: package org.apache.hadoop.hbase.filter

class Filter

Change	Result
Abstract method getNextKeyHint (KeyValue) has been removed from this class.	A client program may be interrupted by NoSuchMethodError exception.
Abstract method transform (KeyValue) has been removed from this class.	A client program may be interrupted by NoSuchMethodError exception.

- HBASE-12296 Filters should work with ByteBufferedCell.
- HConnection is removed in HBase 2.0.
- RegionLoad and ServerLoad internally moved to shaded PB.

class RegionLoad

Change	Result
Type of field regionLoadPB has been changed from protobuf.generated.ClusterStatusProtos.RegionLoad to shaded.protobuf.generated.ClusterStatusProtos.RegionLoad.	A client program may be interrupted by NoSuchFieldError exception.

- HBASE-15783:AccessControlConstants#OP_ATTRIBUTE_ACL_STRATEGY_CELL_FIRST is not used any more. package org.apache.hadoop.hbase.security.access

interface AccessControlConstants

Change	Result
Field OP_ATTRIBUTE_ACL_STRATEGY of type java.lang.Stringhas been removed from this interface.	A client program may be interrupted by NoSuchFieldError exception.
Field OP_ATTRIBUTE_ACL_STRATEGY_CELL_FIRST of type byte[] has been removed from this interface.	A client program may be interrupted by NoSuchFieldError exception.
Field OP_ATTRIBUTE_ACL_STRATEGY_DEFAULT of type byte[] has been removed from this interface.	A client program may be interrupted by NoSuchFieldError exception.

ServerLoad returns long instead of int 1

hbase-client-1.0.0.jar, ServerLoad.class package org.apache.hadoop.hbase

ServerLoad.getNumberOfRequests ( ) : int 1

org/apache/hadoop/hbase/ServerLoad.getNumberOfRequests:()I

Change	Result
Return value type has been changed from int to long.	This method has been removed because the return type is part of the method signature. A client program may be interrupted by NoSuchMethodError exception.

ServerLoad.getReadRequestsCount ( ) : int 1

org/apache/hadoop/hbase/ServerLoad.getReadRequestsCount:()I

Change	Result
Return value type has been changed from int to long.	This method has been removed because the return type is part of the method signature. A client program may be interrupted by NoSuchMethodError exception.

ServerLoad.getTotalNumberOfRequests ( ) : int 1

org/apache/hadoop/hbase/ServerLoad.getTotalNumberOfRequests:()I

Change	Result
Return value type has been changed from int to long.	This method has been removed because the return type is part of the method signature. A client program may be interrupted by NoSuchMethodError exception.

ServerLoad.getWriteRequestsCount ( ) : int 1

org/apache/hadoop/hbase/ServerLoad.getWriteRequestsCount:()I

Change	Result
Return value type has been changed from int to long.	This method has been removed because the return type is part of the method signature. A client program may be interrupted by NoSuchMethodError exception.

- HBASE-13636 Remove deprecation for HBASE-4072 (Reading of zoo.cfg)
- HConstants are removed. HBASE-16040 Remove configuration "hbase.replication"

class HConstants

Change	Result
Field DEFAULT_HBASE_CONFIG_READ_ZOOKEEPER_CONFIG of type boolean has been removed from this class.	A client program may be interrupted by NoSuchFieldError exception.
Field HBASE_CONFIG_READ_ZOOKEEPER_CONFIG of type java.lang.String has been removed from this class.	A client program may be interrupted by NoSuchFieldError exception.
Field REPLICATION_ENABLE_DEFAULT of type boolean has been removed from this class.	A client program may be interrupted by NoSuchFieldError exception.
Field REPLICATION_ENABLE_KEY of type java.lang.String has been removed from this class.	A client program may be interrupted by NoSuchFieldError exception.
Field ZOOKEEPER_CONFIG_NAME of type java.lang.String has been removed from this class.	A client program may be interrupted by NoSuchFieldError exception.
Field ZOOKEEPER_USEMULTI of type java.lang.String has been removed from this class.	A client program may be interrupted by NoSuchFieldError exception.

- HBASE-18732: [compat 1-2] HBASE-14047 removed Cell methods without deprecation cycle.

interface Cell 5

Change	Result
Abstract method getFamily ( ) has been removed from this interface.	A client program may be interrupted by NoSuchMethodError exception.
Abstract method getMvccVersion ( ) has been removed from this interface.	A client program may be interrupted by NoSuchMethodError exception.

Change	Result
Abstract method getQualifier ( ) has been removed from this interface.	A client program may be interrupted by NoSuchMethodError exception.
Abstract method getRow ( ) has been removed from this interface.	A client program may be interrupted by NoSuchMethodError exception.
Abstract method getValue ( ) has been removed from this interface.	A client program may be interrupted by NoSuchMethodError exception.

- HBASE-18795:Expose KeyValue.getBuffer() for tests alone. Allows KV#getBuffer in tests only that was deprecated previously.

Region scanner changes:

interface RegionScanner

Change	Result
Abstract method boolean nextRaw (List<Cell>, int) has been removed from this interface.	A client program may be interrupted by NoSuchMethodError exception.

StoreFile changes:

class StoreFile

Change	Result
This class became interface.	A client program may be interrupted by IncompatibleClassChangeError or InstantiationError exception dependent on the usage of this class.

Mapreduce changes:

HFile*Format has been removed in HBase 2.0.

ClusterStatus changes:

HBASE-15843: Replace RegionState.getRegionInTransition() Map with a Set hbase-client-1.0.0.jar, ClusterStatus.class package org.apache.hadoop.hbase

ClusterStatus.getRegionsInTransition ( ) : Map<String,RegionState> 1

org/apache/hadoop/hbase/ClusterStatus.getRegionsInTransition:()Ljava/util/Map;

Change	Result
Return value type has been changed from java.util.Map<java.lang.String,master.RegionState> to java.util.List<master.RegionState>.	This method has been removed because the return type is part of the method signature. A client program may be interrupted by NoSuchMethodError exception.

Other changes in ClusterStatus include removal of convert methods that were no longer necessary after purge of PB from API.

Purge of PBs from API

PBs have been deprecated in APIs in HBase 2.0.

HBaseSnapshotException.getSnapshotDescription ( ) : HBaseProtos.SnapshotDescription 1

org/apache/hadoop/hbase/snapshot/HBaseSnapshotException.getSnapshotDescription:()Lorg/apache/hadoop/hbase/protobuf/generated/HBaseProtos$SnapshotDescription;

Change	Result
Return value type has been changed from org.apache.hadoop.hbase.protobuf.generated.HBaseProtos.SnapshotDescription to org.apache.hadoop.hbase.client.SnapshotDescription.	This method has been removed because the return type is part of the method signature. A client program may be interrupted by NoSuchMethodError exception.

- HBASE-15609: Remove PB references from Result, DoubleColumnInterpreter and any such public facing class for 2.0. hbase-client-1.0.0.jar, Result.class package org.apache.hadoop.hbase.client

Result.getStats ( ) : ClientProtos.RegionLoadStats 1

org/apache/hadoop/hbase/client/Result.getStats:()Lorg/apache/hadoop/hbase/protobuf/generated/ClientProtos$RegionLoadStats;

Change	Result
Return value type has been changed from org.apache.hadoop.hbase.protobuf.generated.ClientProtos.RegionLoadStats to RegionLoadStats.	This method has been removed because the return type is part of the method signature. A client program may be interrupted by NoSuchMethodError exception.

REST changes:

hbase-rest-1.0.0.jar, Client.class package org.apache.hadoop.hbase.rest.client

Client.getHttpClient ( ) : HttpClient 1

org/apache/hadoop/hbase/rest/client/Client.getHttpClient:
()Lorg/apache/commons/httpclient/HttpClient

Change	Result
Return value type has been changed from org.apache.commons.httpclient.HttpClient to org.apache.http.client.HttpClient.	This method has been removed because the return type is part of the method signature. A client program may be interrupted by NoSuchMethodError exception.

hbase-rest-1.0.0.jar, Response.class package org.apache.hadoop.hbase.rest.client

Response.getHeaders ( ) : Header[ ] 1

org/apache/hadoop/hbase/rest/client/Response.getHeaders:()
[Lorg/apache/commons/httpclient/Header;

Change	Result
Return value type has been changed from org.apache.commons.httpclient.Header[] to org.apache.http.Header[].	This method has been removed because the return type is part of the method signature. A client program may be interrupted by NoSuchMethodError exception.

PrettyPrinter changes:

hbase-server-1.0.0.jar, HFilePrettyPrinter.class package org.apache.hadoop.hbase.io.hfile

HFilePrettyPrinter.processFile (Path file) : void 1

org/apache/hadoop/hbase/io/hfile/HFilePrettyPrinter.processFile:
(Lorg/apache/hadoop/fs/Path;)V

Change	Result
Return value type has been changed from void t o int.	This method has been removed because the retu rn type is part of the method signature. A client program may be interrupted by NoSuchMethodE rror exception.

AccessControlClient changes:

HBASE-13171 Change AccessControlClient methods to accept connection object to reduce setup time. Parameters have been changed in the following methods:

- hbase-client-1.2.7-SNAPSHOT.jar, AccessControlClient.class package org.apache.hadoop.hbase.security.access AccessControlClient.getUserPermissions (Configuration conf, String tableRegex) [static] : List<UserPermission> **DEPRECATED** org/apache/hadoop/hbase/security/access/AccessControlClient.getUserPermissions: (Lorg/apache/hadoop/conf/Configuration;Ljava/lang/String;)Ljava/util/List;
- AccessControlClient.grant (Configuration conf, String namespace, String userName, Permission.Action... actions)[static] : void **DEPRECATED** org/apache/hadoop/hbase/security/access/AccessControlClient.grant: (Lorg/apache/hadoop/conf/Configuration;Ljava/lang/String;Ljava/lang/String; [Lorg/apache/hadoop/hbase/security/access/Permission$Action;)V
- AccessControlClient.grant (Configuration conf, String userName, Permission.Action... actions) [static] : void **DEPRECATED** org/apache/hadoop/hbase/security/access/AccessControlClient.grant: (Lorg/apache/hadoop/conf/Configuration;Ljava/lang/String; [Lorg/apache/hadoop/hbase/security/access/Permission$Action;)V
- AccessControlClient.grant (Configuration conf, TableName tableName, String userName, byte[ ] family, byte[ ] qual,Permission.Action... actions) [static] : void **DEPRECATED** org/apache/hadoop/hbase/security/access/AccessControlClient.grant: (Lorg/apache/hadoop/conf/Configuration;Lorg/apache/hadoop/hbase/TableName;Ljava /lang/String;[B[B[Lorg/apache/hadoop/hbase/security/access/Permission$Action;)V

- AccessControlClient.isAccessControllerRunning (Configuration conf) [static] : boolean DEPRECATED org/apache/hadoop/hbase/security/access/AccessControlClient.isAccessControllerRunning:(Lorg/apache/hadoop/conf/Configuration;)Z

- AccessControlClient.revoke (Configuration conf, String namespace, String userName, Permission.Action... actions)[static] : void DEPRECATED org/apache/hadoop/hbase/security/access/AccessControlClient.revoke:(Lorg/apache/hadoop/conf/Configuration;Ljava/lang/String;Ljava/lang/String;[Lorg/apache/hadoop/hbase/security/access/Permission$Action;)V

- AccessControlClient.revoke (Configuration conf, String userName, Permission.Action... actions) [static] : void DEPRECATED org/apache/hadoop/hbase/security/access/AccessControlClient.revoke:(Lorg/apache/hadoop/conf/Configuration;Ljava/lang/String;[Lorg/apache/hadoop/hbase/security/access/Permission$Action;)V

- AccessControlClient.revoke (Configuration conf, TableName tableName, String username, byte[ ] family, byte[ ] qualifier,Permission.Action... actions) [static] : void DEPRECATED org/apache/hadoop/hbase/security/access/AccessControlClient.revoke:(Lorg/apache/hadoop/conf/Configuration;Lorg/apache/hadoop/hbase/TableName;Ljava/lang/String;[B[B[Lorg/apache/hadoop/hbase/security/access/Permission$Action;)V

- HBASE-18731: [compat 1-2] Mark protected methods of QuotaSettings that touch Protobuf internals as IA.Private

1. See 'Source Compatibility' https://wiki.openjdk.org/spaces/csr/pages/32342052/Kinds+of+Compatibility ↩

2. See http://docs.oracle.com/javase/specs/jls/se8/html/jls-13.html. ↩

3. Note that this indicates what could break, not that it will break. We will/should add specifics in our release notes. ↩

4. Running an offline upgrade tool without downgrade might be needed. We will typically only support migrating data from major version X to major version X+1. ↩

5. The Metrics system was redone in HBase 0.96. See Migration to the New Metrics Hotness – Metrics2 by Elliot Clark for detail. ↩