

The Apache [HBase](#) Book

Copyright © 2010 Apache Software Foundation

Revision History		
Revision 0.91.0-SNAPSHOT		
Adding first cuts at Configuration, Getting Started, Data Model		
Revision 0.89.20100924	5 October 2010	stack
Initial layout		

Abstract

This is the official book of [Apache HBase](#), a distributed, versioned, column-oriented database built on top of [Apache Hadoop](#) and [Apache ZooKeeper](#).

Table of Contents

[Preface](#)

[1. Getting Started](#)

[1.1. Introduction](#)

[1.2. Quick Start](#)

[1.2.1. Download and unpack the latest stable release.](#)

[1.2.2. Start HBase](#)

[1.2.3. Shell Exercises](#)

[1.2.4. Stopping HBase](#)

[1.2.5. Where to go next](#)

[2. Configuration](#)

[2.1. Java](#)

[2.2. Operating System](#)

[2.2.1. ssh](#)

[2.2.2. DNS](#)

[2.2.3. NTP](#)

[2.2.4. ulimit and nproc](#)

[2.2.5. Windows](#)

[2.3. Hadoop](#)

[2.3.1. Hadoop Security](#)

[2.3.2. dfs.datanode.max.xcievers](#)

[2.4. HBase run modes: Standalone and Distributed](#)

[2.4.1. Standalone HBase](#)

[2.4.2. Distributed](#)

[2.4.3. Running and Confirming Your Installation](#)

[2.5. ZooKeeper](#)

[2.5.1. Using existing ZooKeeper ensemble](#)[2.6. Configuration Files](#)[2.6.1. hbase-site.xml and hbase-default.xml](#)[2.6.2. hbase-env.sh](#)[2.6.3. log4j.properties](#)[2.6.4. Client configuration and dependencies connecting to an HBase cluster](#)[2.7. Example Configurations](#)[2.7.1. Basic Distributed HBase Install](#)[2.8. The Important Configurations](#)[2.8.1. Required Configurations](#)[2.8.2. Recommended Configurations](#)[3. Upgrading](#)[3.1. Upgrading to HBase 0.90.x from 0.20.x or 0.89.x](#)[4. The HBase Shell](#)[4.1. Scripting](#)[4.2. Shell Tricks](#)[4.2.1. irbrc](#)[4.2.2. LOG data to timestamp](#)[4.2.3. Debug](#)[5. HBase and MapReduce](#)[5.1. The default HBase MapReduce Splitter](#)[5.2. HBase Input MapReduce Example](#)[5.3. Accessing Other HBase Tables in a MapReduce Job](#)[5.4. Speculative Execution](#)[6. HBase and Schema Design](#)[6.1. Schema Creation](#)[6.2. On the number of column families](#)[6.3. Monotonically Increasing Row Keys/Timeseries Data](#)[6.4. Try to minimize row and column sizes](#)[6.5. Number of Versions](#)[6.6. Immutability of Rowkeys](#)[6.7. Supported Datatypes](#)[7. Metrics](#)[7.1. Metric Setup](#)[7.2. RegionServer Metrics](#)[7.2.1. hbase.regionserver.blockCacheCount](#)[7.2.2. hbase.regionserver.blockCacheFree](#)[7.2.3. hbase.regionserver.blockCacheHitRatio](#)[7.2.4. hbase.regionserver.blockCacheSize](#)[7.2.5. hbase.regionserver.compactionQueueSize](#)[7.2.6. hbase.regionserver.fsReadLatency_avg_time](#)

- [7.2.7. hbase.regionserver.fsReadLatency num ops](#)
- [7.2.8. hbase.regionserver.fsSyncLatency avg time](#)
- [7.2.9. hbase.regionserver.fsSyncLatency num ops](#)
- [7.2.10. hbase.regionserver.fsWriteLatency avg time](#)
- [7.2.11. hbase.regionserver.fsWriteLatency num ops](#)
- [7.2.12. hbase.regionserver.memstoreSizeMB](#)
- [7.2.13. hbase.regionserver.regions](#)
- [7.2.14. hbase.regionserver.requests](#)
- [7.2.15. hbase.regionserver.storeFileIndexSizeMB](#)
- [7.2.16. hbase.regionserver.stores](#)
- [7.2.17. hbase.regionserver.storeFiles](#)

[8. Cluster Replication](#)

[9. Data Model](#)

[9.1. Conceptual View](#)

[9.2. Physical View](#)

[9.3. Table](#)

[9.4. Row](#)

[9.5. Column Family](#)

[9.6. Cells](#)

[9.7. Versions](#)

[9.7.1. Versions and HBase Operations](#)

[9.7.2. Current Limitations](#)

[10. Architecture](#)

[10.1. Client](#)

[10.1.1. Connections](#)

[10.1.2. WriteBuffer and Batch Methods](#)

[10.1.3. Filters](#)

[10.2. Daemons](#)

[10.2.1. Master](#)

[10.2.2. RegionServer](#)

[10.3. Regions](#)

[10.3.1. Region Size](#)

[10.3.2. Region Splits](#)

[10.3.3. Region Load Balancer](#)

[10.3.4. Store](#)

[10.4. Write Ahead Log \(WAL\)](#)

[10.4.1. Purpose](#)

[10.4.2. WAL Flushing](#)

[10.4.3. WAL Splitting](#)

[11. Performance Tuning](#)

[11.1. Operating System](#)

[11.1.1. Memory](#)

[11.1.2. 64-bit](#)

[11.1.3. Swapping](#)[11.2. Java](#)[11.2.1. The Garbage Collector and HBase](#)[11.3. Configurations](#)[11.3.1. Number of Regions](#)[11.3.2. Managing Compactions](#)[11.3.3. Compression](#)[11.3.4. hbase.regionserver.handler.count](#)[11.3.5. hfile.block.cache.size](#)[11.3.6. hbase.regionserver.global.memstore.upperLimit](#)[11.3.7. hbase.regionserver.global.memstore.lowerLimit](#)[11.3.8. hbase.hstore.blockingStoreFiles](#)[11.3.9. hbase.hregion.memstore.block.multiplier](#)[11.4. Number of Column Families](#)[11.5. Data Clumping](#)[11.6. Batch Loading](#)[11.6.1. Table Creation: Pre-Creating Regions](#)[11.7. HBase Client](#)[11.7.1. AutoFlush](#)[11.7.2. Scan Caching](#)[11.7.3. Scan Attribute Selection](#)[11.7.4. Close ResultScanners](#)[11.7.5. Block Cache](#)[11.7.6. Optimal Loading of Row Keys](#)[11.7.7. Turn off WAL on Puts](#)[12. Bloom Filters](#)[12.1. Configurations](#)[12.1.1. HColumnDescriptor option](#)[12.1.2. io.hfile.bloom.enabled global kill switch](#)[12.1.3. io.hfile.bloom.error.rate](#)[12.1.4. io.hfile.bloom.max.fold](#)[12.2. Bloom StoreFile footprint](#)[12.2.1. BloomFilter in the StoreFile FileInfo data structure](#)[12.2.2. BloomFilter entries in StoreFile metadata](#)[13. Troubleshooting and Debugging HBase](#)[13.1. General Guidelines](#)[13.2. Logs](#)[13.2.1. Log Locations](#)[13.2.2. JVM Garbage Collection Logs](#)[13.3. Tools](#)[13.3.1. Builtin Tools](#)

[13.3.2. External Tools](#)[13.4. Client](#)[13.4.1. ScannerTimeoutException](#)[13.4.2. Shell or client application throws lots of scary exceptions during normal operation](#)[13.4.3. Long Client Pauses With Compression](#)[13.5. NameNode](#)[13.5.1. HDFS Utilization of Tables and Regions](#)[13.6. RegionServer](#)[13.6.1. Startup Errors](#)[13.6.2. Runtime Errors](#)[13.6.3. Shutdown Errors](#)[13.7. Master](#)[13.7.1. Startup Errors](#)[13.7.2. Shutdown Errors](#)[13.8. ZooKeeper](#)[13.8.1. Startup Errors](#)[13.8.2. ZooKeeper, The Cluster Canary](#)[13.9. Amazon EC2](#)[13.9.1. ZooKeeper does not seem to work on Amazon EC2](#)[13.9.2. Instability on Amazon EC2](#)[14. Building HBase](#)[14.1. Building in snappy compression support](#)[14.2. Adding an HBase release to Apache's Maven Repository](#)[15. Developing HBase](#)[15.1. IDEs](#)[15.1.1. Eclipse](#)[15.2. Unit Tests](#)[15.2.1. Mockito](#)[A. Tools](#)[A.1. HBase hbck](#)[A.2. HFile Tool](#)[A.3. WAL Tools](#)[A.3.1. HLog tool](#)[A.4. Compression Tool](#)[A.5. Node Decommission](#)[A.5.1. Rolling Restart](#)

[B. Compression In HBase](#)

[B.1. CompressionTest Tool](#)

[B.2. hbase.regionserver.codecs](#)

[B.3. LZ0](#)

[B.4. GZIP](#)

[B.5. SNAPPY](#)

[C. FAQ](#)

[D. YCSB: The Yahoo! Cloud Serving Benchmark and HBase](#)

[Index](#)

List of Tables

9.1. [Table webtable](#)

9.2. [ColumnFamily anchor](#)

9.3. [ColumnFamily contents](#)

Preface

This book aims to be the official guide for the [HBase](#) version it ships with. This document describes HBase version *0.9.10-SNAPSHOT*. Herein you will find either the definitive documentation on an HBase topic as of its standing when the referenced HBase version shipped, or this book will point to the location in [javadoc](#), [JIRA](#) or [wiki](#) where the pertinent information can be found.

This book is a work in progress. It is lacking in many areas but we hope to fill in the holes with time. Feel free to add to this book by adding a patch to an issue up in the HBase [JIRA](#).

Heads-up

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 needs span systems (hardware and software) and networking. Your cluster' 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](#).

Chapter 1. Getting Started

Table of Contents

[1.1. Introduction](#)

[1.2. Quick Start](#)

[1.2.1. Download and unpack the latest stable release.](#)

[1.2.2. Start HBase](#)

- [1.2.3. Shell Exercises](#)
- [1.2.4. Stopping HBase](#)
- [1.2.5. Where to go next](#)

1.1. Introduction

[Section 1.2, “Quick Start”](#) will get you up and running on a single-node instance of HBase using the local filesystem. [Chapter 2, Configuration](#) describes setup of HBase in distributed mode running on top of HDFS.

1.2. Quick Start

This guide describes setup of a standalone HBase instance that uses the local filesystem. It leads you through creating a table, inserting rows via the HBase shell, and then cleaning up and shutting down your standalone HBase instance. The below exercise should take no more than ten minutes (not including download time).

1.2.1. Download and unpack the latest stable release.

Choose a download site from this list of [Apache Download Mirrors](#). Click on suggested top link. This will take you to a mirror of *HBase Releases*. Click on the folder named `stable` and then download the file that ends in `.tar.gz` to your local filesystem; e.g. `hbase-0.91.0-SNAPSHOT.tar.gz`.

Decompress and untar your download and then change into the unpacked directory.

```
$ tar xzf hbase-0.91.0-SNAPSHOT.tar.gz
$ cd hbase-0.91.0-SNAPSHOT
```

At this point, you are ready to start HBase. But before starting it, you might want to edit `conf/hbase-site.xml` and set the directory you want HBase to write to, `hbase.rootdir`.

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<configuration>
  <property>
    <name>hbase.rootdir</name>
    <value>file:///DIRECTORY/hbase</value>
  </property>
</configuration>
```

Replace `DIRECTORY` in the above with a path to a directory where you want HBase to store its data. By default, `hbase.rootdir` is set to `/tmp/hbase-{user.name}` which means you'll lose all your data whenever your server reboots (Most operating systems clear `/tmp` on restart).

1.2.2. Start HBase

Now start HBase:

```
$. /bin/start-hbase.sh
starting Master, logging to logs/hbase-user-master-example.org.out
```

You should now have a running standalone HBase instance. In standalone mode, HBase runs all daemons in the the one JVM; i.e. both the HBase and ZooKeeper daemons. HBase logs can be found in the `logs` subdirectory. Check them out especially if HBase had trouble starting.

Is java installed?

All of the above presumes a 1.6 version of Oracle java is installed on your machine and available on your path; i.e. when you type `java`, you see output that describes the options the java program takes (HBase requires java 6). If this is not the case, HBase will not start. Install java, edit `conf/hbase-env.sh`, uncommenting the `JAVA_HOME` line pointing it to your java install. Then, retry the steps above.

1.2.3. Shell Exercises

Connect to your running HBase via the shell.

```
$. /bin/hbase shell
HBase Shell: enter 'help<RETURN>' for list of supported commands.
Type "exit<RETURN>" to leave the HBase Shell
Version: 0.90.0, r1001068, Fri Sep 24 13:55:42 PDT 2010

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

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 emission 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.

Create a table named `test` with a single column family named `cf`. Verify its creation by listing all tables and then insert some values.

```
hbase(main):003:0> create 'test', 'cf'
0 row(s) in 1.2200 seconds
hbase(main):003:0> list 'table'
test
1 row(s) in 0.0550 seconds
hbase(main):004:0> put 'test', 'row1', 'cf:a', 'value1'
0 row(s) in 0.0560 seconds
hbase(main):005:0> put 'test', 'row2', 'cf:b', 'value2'
0 row(s) in 0.0370 seconds
hbase(main):006:0> put 'test', 'row3', 'cf:c', 'value3'
0 row(s) in 0.0450 seconds
```

Above we inserted 3 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).

Verify the data insert.

Run a scan of the table by doing the following


```
hbase(main):007:0> scan 'test'
ROW      COLUMN+CELL
row1     column=cf:a, timestamp=1288380727188, value=value1
row2     column=cf:b, timestamp=1288380738440, value=value2
row3     column=cf:c, timestamp=1288380747365, value=value3
3 row(s) in 0.0590 seconds
```

Get a single row as follows

```
hbase(main):008:0> get 'test', 'row1'
COLUMN    CELL
cf:a      timestamp=1288380727188, value=value1
1 row(s) in 0.0400 seconds
```

Now, disable and drop your table. This will clean up all done above.

```
hbase(main):012:0> disable 'test'
0 row(s) in 1.0930 seconds
hbase(main):013:0> drop 'test'
0 row(s) in 0.0770 seconds
```

Exit the shell by typing exit.

```
hbase(main):014:0> exit
```

1.2.4. Stopping HBase

Stop your hbase instance by running the stop script.

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

1.2.5. Where to go next

The above described standalone setup is good for testing and experiments only. Next move on to [Chapter 2, Configuration](#) where we'll go into depth on the different HBase run modes, requirements and critical configurations needed setting up a distributed HBase deploy.

Chapter 2. Configuration

Table of Contents

[2.1. Java](#)

[2.2. Operating System](#)

[2.2.1. ssh](#)

[2.2.2. DNS](#)

[2.2.3. NTP](#)

[2.2.4. ulimit and nproc](#)

[2.2.5. Windows](#)

[2.3. Hadoop](#)

[2.3.1. Hadoop Security](#)

[2.3.2. `dfs.datanode.max.xcievers`](#)

[2.4. HBase run modes: Standalone and Distributed](#)

[2.4.1. Standalone HBase](#)

[2.4.2. Distributed](#)

[2.4.3. Running and Confirming Your Installation](#)

[2.5. ZooKeeper](#)

[2.5.1. Using existing ZooKeeper ensemble](#)

[2.6. Configuration Files](#)

[2.6.1. `hbase-site.xml` and `hbase-default.xml`](#)

[2.6.2. `hbase-env.sh`](#)

[2.6.3. `log4j.properties`](#)

[2.6.4. Client configuration and dependencies connecting to an HBase cluster](#)

[2.7. Example Configurations](#)

[2.7.1. Basic Distributed HBase Install](#)

[2.8. The Important Configurations](#)

[2.8.1. Required Configurations](#)

[2.8.2. Recommended Configurations](#)

This chapter is the Not-So-Quick start guide to HBase configuration.

Please read this chapter carefully and ensure that all requirements have been satisfied. Failure to do so will cause you (and us) grief debugging strange errors and/or data loss.

HBase uses the same configuration system as Hadoop. To configure a deploy, edit a file of environment variables in `conf/hbase-env.sh` — this configuration is used mostly by the launcher shell scripts getting the cluster off the ground — and then add configuration to an XML file to do things like override HBase defaults, tell HBase what Filesystem to use, and the location of the ZooKeeper ensemble ^[1].

When running in distributed mode, after you make an edit to an HBase configuration, make sure you copy the content of the `conf` directory to all nodes of the cluster. HBase will not do this for you. Use `rsync`.

2.1. Java

Just like Hadoop, HBase requires java 6 from [Oracle](#). Usually you'll want to use the latest version available except the problematic u18 (u24 is the latest version as of this writing).

2.2. Operating System

2.2.1. ssh

ssh must be installed and sshd must be running to use Hadoop's scripts to manage remote Hadoop and HBase daemons. You must be able to ssh to all nodes, including your local node, using passwordless login (Google "ssh passwordless login").

2.2.2. DNS

HBase uses the local hostname to self-report its IP address. Both forward and reverse DNS resolving should work.

If your machine has multiple interfaces, HBase will use the interface that the primary hostname resolves to.

If this is insufficient, you can set `hbase.regionserver.dns.interface` to indicate the primary interface. This only works if your cluster configuration is consistent and every host has the same network interface configuration.

Another alternative is setting `hbase.regionserver.dns.nameserver` to choose a different nameserver than the system wide default.

2.2.3. NTP

The clocks on cluster members should be in basic alignments. Some skew is tolerable but wild skew could generate odd behaviors. Run [NTP](#) on your cluster, or an equivalent.

If you are having problems querying data, or "weird" cluster operations, check system time!

2.2.4. ulimit and nproc

HBase is a database. It uses a lot of files all at the same time. The default `ulimit -n --` i.e. user file limit -- of 1024 on most *nix systems is insufficient (On mac os x its 256). Any significant amount of loading will lead you to [FAQ: Why do I see ".java.io.IOException...\(Too many open files\)" in my logs?](#) You may also notice errors such as

```
2010-04-06 03:04:37,542 INFO org.apache.hadoop.hdfs.DFSClient: Exception increateBlockOutput
2010-04-06 03:04:37,542 INFO org.apache.hadoop.hdfs.DFSClient: Abandoning block blk_+69355246
```

Do yourself a favor and change the upper bound on the number of file descriptors. Set it to north of 10k. See the above referenced FAQ for how. You should also up the hbase users' `nproc` setting; under load, a low-`nproc` setting could manifest as `OutOfMemoryError` [\[2\]](#) [\[3\]](#).

To be clear, upping the file descriptors and `nproc` for the user who is running the HBase process is an operating system configuration, not an HBase configuration. Also, a common mistake is that administrators will up the file descriptors for a particular user but for whatever reason, HBase will be running as some one else. HBase prints in its logs as the first line the `ulimit` its seeing. Ensure its correct. [\[4\]](#)

2.2.4.1. ulimit on Ubuntu

If you are on Ubuntu you will need to make the following changes:

In the file `/etc/security/limits.conf` add a line like:

```
hadoop -nproc 32768
```

Replace `hadoop` with whatever user is running Hadoop and HBase. If you have separate users, you will need 2 entries, one for each user. In the same file set `nproc` hard and soft limits. For example:

```
hadoop soft/hard nproc 32000
```

In the file `/etc/pam.d/common-session` add as the last line in the file:

```
session required pam_limits.so
```

Otherwise the changes in `/etc/security/limits.conf` won't be applied.

Don't forget to log out and back in again for the changes to take effect!

2.2.5. Windows

HBase has been little tested running on Windows. Running a production install of HBase on top of Windows is not recommended.

If you are running HBase on Windows, you must install [Cygwin](#) to have a *nix-like environment for the shell scripts. The full details are explained in the [Windows Installation](#) guide. Also [search our user mailing list](#) to pick up latest fixes figured by Windows users.

2.3. [Hadoop](#)

This version of HBase will only run on [Hadoop 0.20.x](#). It will not run on `hadoop 0.21.x` (nor `0.22.x`). HBase will lose data unless it is running on an HDFS that has a durable sync. Hadoop 0.20.2 and Hadoop 0.20.203.0 DO NOT have this attribute. Currently only the [branch-0.20-append](#) branch has this attribute^[5]. No official releases have been made from the `branch-0.20-append` branch up to now so you will have to build your own Hadoop from the tip of this branch. Michael Noll has written a detailed blog, [Building an Hadoop 0.20.x version for HBase 0.90.2](#), on how to build an Hadoop from `branch-0.20-append`. Recommended.

Or rather than build your own, you could use Cloudera's [CDH3](#). CDH has the `0.20-append` patches needed to add a durable sync (CDH3 betas will suffice; b2, b3, or b4).

Because HBase depends on Hadoop, it bundles an instance of the Hadoop jar under its `lib` directory. The bundled Hadoop was made from the Apache `branch-0.20-append` branch at the time of the HBase's release. The bundled jar is ONLY for use in standalone 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.jar` found in the HBase `lib` directory with the `hadoop.jar` you are running on your cluster to avoid version mismatch issues. Make sure you replace the jar in HBase everywhere on your cluster. Hadoop version mismatch issues have various manifestations but often all looks like its hung up.

2.3.1. Hadoop Security

HBase will run on any Hadoop 0.20.x that incorporates Hadoop security features -- e.g. Y! 0.20S or CDH3B3 -- as long as you do as suggested above and replace the Hadoop jar that ships with HBase with the secure version.

2.3.2. `dfs.datanode.max.xcievers`

An Hadoop HDFS datanode has an upper bound on the number of files that it will serve at any one time. The upper bound parameter is called `xcievers` (yes, this is misspelled). Again, before doing any loading, make sure you have configured Hadoop's `conf/hdfs-site.xml` setting the `xceivers` value to at least the following:

```
<property>
  <name>dfs.datanode.max.xcievers</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. Eventually you'll see a complain in the datanode logs complaining about the `xcievers` exceeded, but on the run up to this one manifestation is complaint about missing blocks. For example:
10/12/08 20:10:31 INFO hdfs.DFSCliet: Could not obtain block
blk_XXXXXXXXXXXXXXXXXXXXX_YYYYYYYY from any node: java.io.IOException: No live
nodes contain current block. Will get new block locations from namenode and retry... ^[6]

2.4. HBase run modes: Standalone and Distributed

HBase has two run modes: [Section 2.4.1, “Standalone HBase”](#) and [Section 2.4.2, “Distributed”](#). Out of the box, HBase runs in standalone mode. To set up a distributed deploy, you will need to configure HBase by editing files in the HBase `conf` directory.

Whatever your mode, you will need to 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.

2.4.1. Standalone HBase

This is the default mode. Standalone mode is what is described in the [Section 1.2, “Quick Start”](#) 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.

2.4.2. 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 ^[7].

Distributed modes require an instance of the *Hadoop Distributed File System* (HDFS). See the Hadoop [requirements and instructions](#) for how to set up a HDFS. Before proceeding, ensure you have an appropriate, working HDFS.

Below we describe the different distributed setups. Starting, verification and exploration of your install, whether a *pseudo-distributed* or *fully-distributed* configuration is described in a section that follows, [Section 2.4.3, “Running and Confirming Your Installation”](#). The same verification script applies to both deploy types.

2.4.2.1. Pseudo-distributed

A pseudo-distributed mode is simply a distributed mode run on a single host. Use this configuration testing and prototyping on HBase. Do not use this configuration for production nor for evaluating HBase performance.

Once you have confirmed your HDFS setup, edit `conf/hbase-site.xml`. This is the file into which you add local customizations and overrides for `<xreg></xreg>` and [Section 2.4.2.3, “HDFS Client Configuration”](#). Point HBase at the running Hadoop HDFS instance by setting the `hbase.rootdir` property. This property points HBase at the Hadoop filesystem instance to use. For example, adding the properties below to your `hbase-site.xml` says that HBase should use the `/hbase` directory in the HDFS whose namenode is at port 9000 on your local machine, and that it should run with one replica only (recommended for pseudo-distributed mode):

```
<configuration>
...
<property>
  <name>hbase.rootdir</name>
  <value>hdfs://localhost:9000/hbase</value>
  <description>The directory shared by RegionServers.
  </description>
</property>
<property>
  <name>dfs.replication</name>
  <value>1</value>
  <description>The replication count for HLog and HFile storage. Should not be greater than HDF
  </description>
</property>
...
</configuration>
```

Note

Let HBase create the `hbase.rootdir` directory. If you don't, you'll get warning saying HBase needs a migration run because the directory is missing files expected by HBase (it'll create them if you let it).

Note

Above we bind to `localhost`. This means that a remote client cannot connect. Amend accordingly, if you want to connect from a remote location.

Now skip to [Section 2.4.3, “Running and Confirming Your Installation”](#) for how to start and verify your pseudo-distributed install. ^[8]

2.4.2.2. Fully-distributed

For running a fully-distributed operation on more than one host, make the following configurations. In `hbase-site.xml`, add the property `hbase.cluster.distributed` and set it

to true and point the HBase `hbase.rootdir` at the appropriate HDFS NameNode and location in HDFS where you would like HBase to write data. For example, if your namenode were running at `namenode.example.org` on port 9000 and you wanted to home your HBase in HDFS at `/hbase`, make the following configuration.

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

2.4.2.2.1. `regionservers`

In addition, a fully-distributed mode requires that you modify `conf/regionservers`. The [Section 2.7.1.2, “`regionservers`”](#) file lists all hosts that you would have running `HRegionServers`, one host per line (This file in HBase is like the Hadoop `slaves` file). All servers listed in this file will be started and stopped when HBase cluster start or stop is run.

2.4.2.2.2. ZooKeeper and HBase

See section [Section 2.5, “ZooKeeper”](#) for ZooKeeper setup for HBase.

2.4.2.2.3. HDFS Client Configuration

Of note, if you have made *HDFS client configuration* on your Hadoop cluster -- i.e. configuration you want HDFS clients to use as opposed to server-side configurations -- HBase will not see this configuration unless you do one of the following:

- 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.

2.4.3. 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 daemons. These do not need to be started.

If you are managing your own ZooKeeper, start it and confirm its 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 its deployed on the Master host at port 60010 (HBase RegionServers listen on port 60020 by default and put up an informational http server at 60030). If the Master were running on a host named master.example.org on the default port, to see the Master's homepage you'd point your browser at <http://master.example.org:60010>.

Once HBase has started, see the [Section 1.2.3, “Shell Exercises”](#) 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.

2.5. ZooKeeper

A distributed HBase depends on a running ZooKeeper cluster. All participating nodes and clients need to be able to access the running ZooKeeper ensemble. 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 using its native [zoo.cfg](#) file, or, the easier option is to just specify ZooKeeper options 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 [Section 2.6.1.1, “HBase Default Configuration”](#). Look for the hbase.zookeeper.property prefix ^[9]

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. There can be no quorum if the number of members is an even number. 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.example.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>
```

2.5.1. 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 it's 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`, or add a suitably configured `zoo.cfg` to HBase's `CLASSPATH`. HBase will prefer the configuration found in `zoo.cfg` over any settings 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

```
S{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.

2.6. Configuration Files

2.6.1. `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 [Section 2.6.1.1, “HBase Default Configuration”](#) 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`. Configuration that it is thought rare anyone would change can exist only in code; the only way to turn up such configurations is via a reading of the source code itself.

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

2.6.1.1. HBase Default Configuration

HBase Default Configuration

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

`hbase.rootdir`

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 HBase writes into /tmp. Change this configuration else all data will be lost on machine restart.

Default: file:///tmp/hbase-\${user.name}/hbase

hbase.master.port

The port the HBase Master should bind to.

Default: 60000

hbase.cluster.distributed

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.tmp.dir

Temporary directory on the local filesystem. Change this setting to point to a location more permanent than '/tmp' (The '/tmp' directory is often cleared on machine restart).

Default: /tmp/hbase-\${user.name}

hbase.master.info.port

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

Default: 60010

hbase.master.info.bindAddress

The bind address for the HBase Master web UI

Default: 0.0.0.0

hbase.client.write.buffer

Default size of the HTable client 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.regionserver.port

The port the HBase RegionServer binds to.

Default: 60020

hbase.regionserver.info.port

The port for the HBase RegionServer web UI Set to -1 if you do not want the RegionServer UI to run.

Default: 60030

`hbase.regionserver.info.port.auto`

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.info.bindAddress`

The address for the HBase RegionServer web UI

Default: 0.0.0.0

`hbase.regionserver.class`

The RegionServer interface to use. Used by the client opening proxy to remote region server.

Default: `org.apache.hadoop.hbase.ipc.HRegionInterface`

`hbase.client.pause`

General client pause value. Used mostly as value to wait before running a retry of a failed get, region lookup, etc.

Default: 1000

`hbase.client.retries.number`

Maximum retries. Used as maximum for all retryable operations such as fetching of the root region from root region server, getting a cell's value, starting a row update, etc. Default: 10.

Default: 10

`hbase.client.scanner.caching`

Number of rows that will be fetched when calling next on a scanner if it is not served from (local, client) memory. 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.regionserver.lease.period`

Default: 1

`hbase.client.keyvalue.maxsize`

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.regionserver.lease.period`

HRegion server lease period in milliseconds. Default is 60 seconds. Clients must report in within this period else they are considered dead.

Default: 60000

`hbase.regionserver.handler.count`

Count of RPC Listener instances spun up on RegionServers. Same property is used by the Master for count of master handlers. Default is 10.

Default: 10

`hbase.regionserver.msginterval`

Interval between messages from the RegionServer to Master in milliseconds.

Default: 3000

`hbase.regionserver.optionallogflushinterval`

Sync the HLog to the HDFS after this interval if it has not accumulated enough entries to trigger a sync. Default 1 second. Units: milliseconds.

Default: 1000

`hbase.regionserver.regionSplitLimit`

Limit for the number of regions after which no more region splitting should take place. This is not a 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 MAX_INT; i.e. do not block splitting.

Default: 2147483647

`hbase.regionserver.logroll.period`

Period at which we will roll the commit log regardless of how many edits it has.

Default: 3600000

`hbase.regionserver.hlog.reader.impl`

The HLog file reader implementation.

Default: `org.apache.hadoop.hbase.regionserver.wal.SequenceFileLogReader`

`hbase.regionserver.hlog.writer.impl`

The HLog file writer implementation.

Default: org.apache.hadoop.hbase.regionserver.wal.SequenceFileLogWriter

hbase.regionserver.nbreservationblocks

The number of resevoir blocks of memory release on OOME so we can cleanup properly before server shutdown.

Default: 4

hbase.zookeeper.dns.interface

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

Default: default

hbase.zookeeper.dns.nameserver

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.regionserver.dns.interface

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

Default: default

hbase.regionserver.dns.nameserver

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.master.dns.interface

The name of the Network Interface from which a master should report its IP address.

Default: default

hbase.master.dns.nameserver

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

Default: default

hbase.balancer.period

Period at which the region balancer runs in the Master.

Default: 300000

hbase.regions.slop

Rebalance if any regionserver has average + (average * slop) regions. Default is 20% slop.

Default: 0.2

hbase.master.logcleaner.ttl

Maximum time a HLog can stay in the .oldlogdir directory, after which it will be cleaned by a Master thread.

Default: 600000

hbase.master.logcleaner.plugins

A comma-separated list of LogCleanerDelegate invoked by the LogsCleaner service. These WAL/HLog cleaners are called in order, so put the HLog cleaner that prunes the most HLog files in front. To implement your own LogCleanerDelegate, 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.TimeToLiveLogCleaner

hbase.regionserver.global.memstore.upperLimit

Maximum size of all memstores in a region server before new updates are blocked and flushes are forced. Defaults to 40% of heap

Default: 0.4

hbase.regionserver.global.memstore.lowerLimit

When memstores are being forced to flush to make room in memory, keep flushing until we hit this mark. Defaults to 35% of heap. This value equal to hbase.regionserver.global.memstore.upperLimit causes the minimum possible flushing to occur when updates are blocked due to memstore limiting.

Default: 0.35

hbase.server.thread.wakefrequency

Time to sleep in between searches for work (in milliseconds). Used as sleep interval by service threads such as log roller.

Default: 10000

hbase.hregion.memstore.flush.size

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: 67108864

hbase.hregion.preclose.flush.size

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

Block updates if memstore has hbase.hregion.block.memstore time hbase.hregion.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: 2

hbase.hregion.memstore.mslab.enabled

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.max.filesize

Maximum HStoreFile size. If any one of a column families' HStoreFiles has grown to exceed this value, the hosting HRegion is split in two. Default: 256M.

Default: 268435456

hbase.hstore.compactionThreshold

If more than this number of HStoreFiles in any one HStore (one HStoreFile is written per flush of memstore) then a compaction is run to rewrite all HStoreFiles files as one. Larger numbers put off compaction but when it runs, it takes longer to complete.

Default: 3

hbase.hstore.blockingStoreFiles

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

Default: 7

hbase.hstore.blockingWaitTime

The time an HRegion will block updates for after hitting the StoreFile limit defined by `hbase.hstore.blockingStoreFiles`. After this time has elapsed, the HRegion will stop blocking updates even if a compaction has not been completed.
Default: 90 seconds.

Default: 90000

`hbase.hstore.compaction.max`

Max number of HStoreFiles to compact per 'minor' compaction.

Default: 10

`hbase.hregion.majorcompaction`

The time (in miliseconds) between 'major' compactons of all HStoreFiles in a region.
Default: 1 day. Set to 0 to disable automated major compactons.

Default: 86400000

`hbase.mapreduce.hfileoutputformat.blocksize`

The mapreduce HFileOutputFormat writes storefiles/hfiles. This is the minimum hfile blocksize to emit. Usually in hbase, writing hfiles, the blocksize is gotten from the table schema (HColumnDescriptor) but in the mapreduce outputformat context, we don't have access to the schema so get blocksize from Configuration. The smaller you make the blocksize, the bigger your index and the less you fetch on a random-access. Set the blocksize down if you have small cells and want faster random-access of individual cells.

Default: 65536

`hfile.block.cache.size`

Percentage of maximum heap (-Xmx setting) to allocate to block cache used by HFile/StoreFile. Default of 0.2 means allocate 20%. Set to 0 to disable.

Default: 0.2

`hbase.hash.type`

The hashing algorithm for use in HashFunction. Two values are supported now: murmur (MurmurHash) and jenkins (JenkinsHash). Used by bloom filters.

Default: murmur

`hbase.rpc.engine`

Implementation of `org.apache.hadoop.hbase.ipc.RpcEngine` to be used for client / server RPC call marshalling.

Default: `org.apache.hadoop.hbase.ipc.WritableRpcEngine`

`hbase.master.keytab.file`

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

Default:

`hbase.master.kerberos.principal`

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:

`hbase.regionserver.keytab.file`

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

Default:

`hbase.regionserver.kerberos.principal`

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:

`zookeeper.session.timeout`

ZooKeeper session timeout. HBase passes this to the zk quorum as suggested maximum time for a session (This setting becomes zookeeper's 'maxSessionTimeout'). See http://hadoop.apache.org/zookeeper/docs/current/zookeeperProgrammers.html#ch_zkSessions "The client sends a requested timeout, the server responds with the timeout that it can give the client." In milliseconds.

Default: 180000

`zookeeper.znode.parent`

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 path are configured with a relative path, so they will all go under this directory unless changed.

Default: /hbase

`zookeeper.znode.rootserver`

Path to ZNode holding root region location. This is written by the master and read by clients and region servers. If a relative path is given, the parent folder will be `$zookeeper.znode.parent`. By default, this means the root location is stored at `/hbase/root-region-server`.

Default: root-region-server

`hbase.coprocessor.region.classes`

A comma-separated list of 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, just put it in HBase's classpath and add the fully qualified class name here. A coprocessor can also be loaded on demand by setting `HTableDescriptor`.

Default:

`hbase.coprocessor.master.classes`

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:

`hbase.zookeeper.quorum`

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.

Default: localhost

`hbase.zookeeper.peerport`

Port used by ZooKeeper peers to talk to each other. See http://hadoop.apache.org/zookeeper/docs/r3.1.1/zookeeperStarted.html#sc_RunningReplicatedZooKeeper for more information.

Default: 2888

`hbase.zookeeper.leaderport`

Port used by ZooKeeper for leader election. See http://hadoop.apache.org/zookeeper/docs/r3.1.1/zookeeperStarted.html#sc_RunningReplicatedZooKeeper for more information.

Default: 3888

`hbase.zookeeper.property.initLimit`

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

Default: 10

`hbase.zookeeper.property.syncLimit`

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

Property from ZooKeeper's config zoo.cfg. The directory where the snapshot is stored.

Default: \${hbase.tmp.dir}/zookeeper

hbase.zookeeper.property.clientPort

Property from ZooKeeper's config zoo.cfg. The port at which the clients will connect.

Default: 2181

hbase.zookeeper.property.maxClientCnxns

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: 30

hbase.rest.port

The port for the HBase REST server.

Default: 8080

hbase.rest.readonly

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.defaults.for.version.skip

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 (***VERSION***), this version is X.X.X-SNAPSHOT"

Default: false

2.6.2. 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.

2.6.3. `log4j.properties`

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.

2.6.4. Client configuration and dependencies connecting to an HBase cluster

Since the HBase Master may move around, clients bootstrap by looking to ZooKeeper for current critical locations. ZooKeeper is where all these values are kept. Thus clients require the location of the ZooKeeper ensemble information before they can do anything else. Usually this the ensemble location is 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 a 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).

Minimally, a client of HBase needs the `hbase`, `hadoop`, `log4j`, `commons-logging`, `commons-lang`, and `ZooKeeper` jars in its CLASSPATH connecting to a cluster.

An example basic `hbase-site.xml` for client only might look 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>The directory shared by region servers.
  </description>
  </property>
</configuration>
```

2.6.4.1. 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 for the cluster programmatically do as follows:

```
Configuration config = HBaseConfiguration.create();
```

```
config.set("hbase.zookeeper.quorum", "localhost"); // Here we are running zookeeper locally
```

If multiple ZooKeeper instances make up your ZooKeeper ensemble, they may be specified in a comma-separated list (just as in the `hbase-site.xml` file). This populated Configuration instance can then be passed to an [HTable](#), and so on.

2.7. Example Configurations

2.7.1. Basic Distributed HBase Install

Here is an example basic configuration 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`, `regionserver`, and `hbase-env.sh` — found in the HBase `conf` directory might look like.

2.7.1.1. `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's 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>
```

2.7.1.2. `regionserver`

In this file you list the nodes that will run RegionServers. In our case we run

RegionServers on all but the head node `example1` which is carrying the HBase Master and the HDFS namenode

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

2.7.1.3. `hbase-env.sh`

Below we use a diff to show the differences from default in the `hbase-env.sh` file. Here we are setting the HBase heap to be 4G instead of the default 1G.

```
$ git diff hbase-env.sh
diff --git a/conf/hbase-env.sh b/conf/hbase-env.sh
index e70ebc6..96f8c27 100644
--- a/conf/hbase-env.sh
+++ b/conf/hbase-env.sh
@@ -31,7 +31,7 @@ export JAVA_HOME=/usr/lib/jvm/java-6-sun/
# export HBASE_CLASSPATH=

# The maximum amount of heap to use, in MB. Default is 1000.
-# export HBASE_HEAPSIZE=1000
+export HBASE_HEAPSIZE=4096

# Extra Java runtime options.
# Below are what we set by default. May only work with SUN JVM.
```

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

2.8. The Important Configurations

Below we list what the *important* Configurations. We've divided this section into required configuration and worth-a-look recommended configs.

2.8.1. Required Configurations

Review the [Section 2.2, “Operating System”](#) and [Section 2.3, “Hadoop”](#) sections.

2.8.2. Recommended Configurations

2.8.2.1. `zookeeper.session.timeout`

The default timeout is three minutes (specified in milliseconds). This means that if a server crashes, it will be three minutes before the Master notices the crash and starts recovery. You might like to tune the timeout down to a minute or even less so the Master notices failures the 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 around the cluster and restart.

We set this value high to save our having to field noob 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.

2.8.2.2. Number of ZooKeeper Instances

See [Section 2.5, “ZooKeeper”](#).

2.8.2.3. `hbase.regionserver.handler.count`

This setting defines the number of threads that are kept open to answer incoming requests to user tables. The default of 10 is rather low in order to prevent users from killing their region servers when using large write buffers with a high number of concurrent clients. 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).

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 region server 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 region server will take longer, which exacerbates the problem even more.

2.8.2.4. 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.

2.8.2.5. 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 [Appendix B, *Compression In HBase*](#) for more information.

2.8.2.6. Bigger Regions

Consider going to larger regions to cut down on the total number of regions on your cluster. Generally less Regions to manage makes for a smoother running cluster (You can always later manually split the big Regions should one prove hot and you want to spread the request load over the cluster). By default, regions are 256MB in size. You could run with 1G. Some run with even larger regions: 4G or even larger. Adjust `hbase.hregion.max.filesize` in your `hbase-site.xml`.

2.8.2.7. Managed Splitting

Rather than let HBase auto-split your Regions, manage the splitting manually ^[10]. With growing amounts of data, splits will continually be needed. Since you always know exactly what regions you have, long-term debugging and profiling is much easier with manual splits. It is hard to trace the logs to understand region level problems if it keeps splitting and getting renamed. Data offlining bugs + unknown number of split regions == oh crap! If an HLog or StoreFile was mistakenly unprocessed by HBase due to a weird bug and you notice it a day or so later, you can be assured that the regions specified in these files are the same as the current regions and you have less headaches trying to restore/replay your data. You can finely tune your compaction algorithm. With roughly uniform data growth, it's easy to cause split / compaction storms as the regions all roughly hit the same data size at the same time. With manual splits, you can let staggered, time-based major compactons spread out your network IO load.

How do I turn off automatic splitting? Automatic splitting is determined by the configuration value `hbase.hregion.max.filesize`. It is not recommended that you set this to `Long.MAX_VALUE` in case you forget about manual splits. A suggested setting is 100GB, which would result in > 1hr major compactons if reached.

What's the optimal number of pre-split regions to create? Mileage will vary depending upon your application. You could start low with 10 pre-split regions / server and watch as data grows over time. It's better to err on the side of too little regions and rolling split later. A more complicated answer is that this depends upon the largest storefile in your region. With a growing data size, this will get larger over time. You want the largest region to be just big enough that the Store compact selection algorithm only compactns it due to a timed major. If you don't, your cluster can be prone to compaction storms as the algorithm decides to run major compactons on a large series of regions all at once. Note that compaction storms are due to the uniform data growth, not the manual split decision.

If you pre-split your regions too thin, you can increase the major compaction interval by configuring `HConstants.MAJOR_COMPACTION_PERIOD`. If your data size grows too large, use the (post-0.90.0 HBase) `org.apache.hadoop.hbase.util.RegionSplitter` script to perform a network IO safe rolling split of all regions.

[1] Be careful editing XML. Make sure you close all elements. Run your file through `xmllint` or similar to ensure well-formedness of your document after an edit session.

[2] See Jack Levin's [major hdfs issues](#) note up on the user list.

[3] The requirement that a database requires upping of system limits is not peculiar to HBase. See for example the section *Setting Shell Limits for the Oracle User* in [Short Guide to install Oracle 10 on Linux](#).

- [4] A useful read setting config on you hadoop cluster is Aaron Kimballs' Configuration Parameters: What can you just ignore?
- [5] See [CHANGES.txt](#) in branch-0.20-append to see list of patches involved adding append on the Hadoop 0.20 branch.
- [6] See [Hadoop HDFS: Deceived by Xciever](#) for an informative rant on xceivering.
- [7] The pseudo-distributed vs fully-distributed nomenclature comes from Hadoop.
- [8] See [Pseudo-distributed mode extras](#) for notes on how to start extra Masters and RegionServers when running pseudo-distributed.
- [9] 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.
- [10] What follows is taken from the javadoc at the head of the `org.apache.hadoop.hbase.util.RegionSplitter` tool added to HBase post-0.90.0 release.

Chapter 3. Upgrading

Table of Contents

[3.1. Upgrading to HBase 0.90.x from 0.20.x or 0.89.x](#)

Review [Chapter 2. Configuration](#), in particular the section on Hadoop version.

3.1. Upgrading to HBase 0.90.x from 0.20.x or 0.89.x

This version of 0.90.x HBase can be started on data written by HBase 0.20.x or HBase 0.89.x. There is no need of a migration step. HBase 0.89.x and 0.90.x does write out the name of region directories differently -- it names them with a md5 hash of the region name rather than a jenkins hash -- so this means that once started, there is no going back to HBase 0.20.x.

Be sure to remove the `hbase-default.xml` from your `conf` directory on upgrade. A 0.20.x version of this file will have sub-optimal configurations for 0.90.x HBase. The `hbase-default.xml` file is now bundled into the HBase jar and read from there. If you would like to review the content of this file, see it in the src tree at `src/main/resources/hbase-default.xml` or see [Section 2.6.1.1, "HBase Default Configuration"](#).

Finally, if upgrading from 0.20.x, check your `.META.` schema in the shell. In the past we would recommend that users run with a 16kb `MEMSTORE_FLUSH_SIZE`. Run `hbase> scan '-ROOT-'` in the shell. This will output the current `.META.` schema. Check `MEMSTORE_FLUSH_SIZE` size. Is it 16kb (16384)? If so, you will need to change this (The 'normal'/default value is 64MB (67108864)). Run the script `bin/set_meta_memstore_size.rb`. This will make the necessary edit to your `.META.` schema. Failure to run this change will make for a slow cluster [\[11\]](#).

[11] See [HBASE-3499 Users upgrading to 0.90.0 need to have their .META. table updated with the right MEMSTORE_SIZE](#)

Chapter 4. The HBase Shell

Table of Contents

[4.1. Scripting](#)

[4.2. Shell Tricks](#)

[4.2.1. irbrc](#)

[4.2.2. LOG data to timestamp](#)

[4.2.3. Debug](#)

The 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 emission 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 [Section 12.3, “Shell Exercises”](#) for example basic shell operation.

4.1. Scripting

For examples scripting 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
```

4.2. Shell Tricks

4.2.1. 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"
```

See the ruby documentation of `.irbrc` to learn about other possible confiurations.

4.2.2. 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 20:56:29")
```

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](#).

4.2.3. Debug

4.2.3.1. 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>
```

4.2.3.2. DEBUG log level

To enable DEBUG level logging in the shell, launch it with the `-d` option.

```
$. /bin/hbase shell -d
```

Chapter 5. HBase and MapReduce

Table of Contents

[5.1. The default HBase MapReduce Splitter](#)

[5.2. HBase Input MapReduce Example](#)

[5.3. Accessing Other HBase Tables in a MapReduce Job](#)

[5.4. Speculative Execution](#)

See [HBase and MapReduce](#) up in javadocs. Start there. Below is some additional help.

5.1. 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.

5.2. HBase Input MapReduce Example

To use HBase as a MapReduce source, the job would be configured via [TableMapReduceUtil](#)

in the following manner...

```
Job job = ...;
Scan scan = new Scan();
scan.setCaching(500); // 1 is the default in Scan, which will be bad for MapReduce jobs
scan.setCacheBlocks(false);
// Now set other scan attrs
...

TableMapReduceUtil.initTableMapperJob(
    tableName,          // input HBase table name
    scan,               // Scan instance to control CF and attribute selection
    MyMapper.class,     // mapper
    Text.class,         // reducer key
    LongWritable.class, // reducer value
    job                // job instance
);
```

...and the mapper instance would extend [TableMapper](#)...

```
public class MyMapper extends TableMapper<Text, LongWritable> {
    public void map(ImmutableBytesWritable row, Result value, Context context)
        throws InterruptedException, IOException {
        // process data for the row from the Result instance.
    }
}
```

5.3. 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 HTable instance in the setup method of the Mapper.

```
public class MyMapper extends TableMapper<Text, LongWritable> {
    private HTable myOtherTable;

    @Override
    public void setup(Context context) {
        myOtherTable = new HTable("myOtherTable");
    }
}
```

5.4. 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, on 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.

Chapter 6. HBase and Schema Design

Table of Contents

[6.1. Schema Creation](#)

[6.2. On the number of column families](#)

[6.3. Monotonically Increasing Row Keys/Timeseries Data](#)

[6.4. Try to minimize row and column sizes](#)

[6.5. Number of Versions](#)

[6.6. Immutability of Rowkeys](#)

[6.7. Supported Datatypes](#)

A good general introduction on the strength and weaknesses modelling on the various non-rdbms datastores is Ian Varleys' Master thesis, [No Relation: The Mixed Blessings of Non-Relational Databases](#). Recommended.

6.1. Schema Creation

HBase schemas can be created or updated with [Chapter 4, The HBase Shell](#) or by using [HBaseAdmin](#) in the Java API.

Tables must be disabled when making ColumnFamily modifications, for example..

```
Configuration config = HBaseConfiguration.create();
HBaseAdmin admin = new HBaseAdmin(config);
String table = "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 [Section 2.6.4, “Client configuration and dependencies connecting to an HBase cluster”](#) for more information about configuring client connections.

6.2. On the number of column families

HBase currently does not do well with anything about two or three column families so keep the number of column families in your schema low. Currently, flushing and compactions are 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 though the amount of data they carry is small. Compaction is currently triggered by the total number of files under a column family. Its not size based. When many column families the flushing and compaction interaction can make for a bunch of needless i/o loading (To be addressed by changing flushing and compaction to work on a per column family basis).

Try to make do with one column famliy 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.

6.3. Monotonically Increasing Row Keys/Timeseries Data

In the HBase chapter of Tom White's book *Hadoop: The Definitive Guide* (O'Reilly) there is 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 its 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.

6.4. Try to minimize row and column sizes

Or why are my storefile indices large?

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 ([Section 10.3.4.2, "StoreFile \(HFile\)"](#)) 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.

6.5. Number of Versions

The number of row versions to store is configured per column family via [HColumnDescriptor](#). The default is 3. This is an important parameter because as described in [Chapter 9, 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 versions may need to be increased or decreased depending on application needs.

6.6. 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).

6.7. 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 be 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 [Chapter 9. Data Model](#), and that includes versioning. Take that into consideration when making your design, as well as block size for the ColumnFamily.

Chapter 7. Metrics

Table of Contents

[7.1. Metric Setup](#)

[7.2. RegionServer Metrics](#)

[7.2.1. hbase.regionserver.blockCacheCount](#)

[7.2.2. hbase.regionserver.blockCacheFree](#)

[7.2.3. hbase.regionserver.blockCacheHitRatio](#)

[7.2.4. hbase.regionserver.blockCacheSize](#)

[7.2.5. hbase.regionserver.compactionQueueSize](#)

[7.2.6. hbase.regionserver.fsReadLatency_avg_time](#)

[7.2.7. hbase.regionserver.fsReadLatency_num_ops](#)

[7.2.8. hbase.regionserver.fsSyncLatency_avg_time](#)

[7.2.9. hbase.regionserver.fsSyncLatency_num_ops](#)

[7.2.10. hbase.regionserver.fsWriteLatency_avg_time](#)

[7.2.11. hbase.regionserver.fsWriteLatency_num_ops](#)

[7.2.12. hbase.regionserver.memstoreSizeMB](#)

[7.2.13. hbase.regionserver.regions](#)

[7.2.14. hbase.regionserver.requests](#)

[7.2.15. hbase.regionserver.storeFileIndexSizeMB](#)

[7.2.16. hbase.regionserver.stores](#)

[7.2.17. hbase.regionserver.storeFiles](#)

7.1. Metric Setup

See [Metrics](#) for an introduction and how to enable Metrics emission.

7.2. RegionServer Metrics

7.2.1. hbase.regionserver.blockCacheCount

Block cache item count in memory. This is the number of blocks of storefiles (HFiles) in the cache.

7.2.2. hbase.regionserver.blockCacheFree

Block cache memory available (bytes).

7.2.3. `hbase.regionserver.blockCacheHitRatio`

Block cache hit ratio (0 to 100). TODO: describe impact to ratio where read requests that have `cacheBlocks=false`

7.2.4. `hbase.regionserver.blockCacheSize`

Block cache size in memory (bytes)

7.2.5. `hbase.regionserver.compactionQueueSize`

Size of the compaction queue. This is the number of stores in the region that have been targeted for compaction.

7.2.6. `hbase.regionserver.fsReadLatency_avg_time`

Filesystem read latency (ms). This is the average time to read from HDFS.

7.2.7. `hbase.regionserver.fsReadLatency_num_ops`

TODO

7.2.8. `hbase.regionserver.fsSyncLatency_avg_time`

Filesystem sync latency (ms)

7.2.9. `hbase.regionserver.fsSyncLatency_num_ops`

TODO

7.2.10. `hbase.regionserver.fsWriteLatency_avg_time`

Filesystem write latency (ms)

7.2.11. `hbase.regionserver.fsWriteLatency_num_ops`

TODO

7.2.12. `hbase.regionserver.memstoreSizeMB`

Sum of all the memstore sizes in this RegionServer (MB)

7.2.13. `hbase.regionserver.regions`

Number of regions served by the RegionServer

7.2.14. `hbase.regionserver.requests`

Total number of read and write requests. Requests correspond to RegionServer RPC calls, thus a single Get will result in 1 request, but a Scan with caching set to 1000 will result in 1 request for each 'next' call (i.e., not each row). A bulk-load request will constitute 1 request per HFile.

7.2.15. `hbase.regionserver.storeFileIndexSizeMB`

Sum of all the storefile index sizes in this RegionServer (MB)

7.2.16. `hbase.regionserver.stores`

Number of stores open on the RegionServer. A store corresponds to a column family. For example, if a table (which contains the column family) has 3 regions on a RegionServer, there will be 3 stores open for that column family.

7.2.17. `hbase.regionserver.storeFiles`

Number of store files open on the RegionServer. A store may have more than one storefile (HFile).

Chapter 8. Cluster Replication

See [Cluster Replication](#).

Chapter 9. Data Model

Table of Contents

[9.1. Conceptual View](#)

[9.2. Physical View](#)

[9.3. Table](#)

[9.4. Row](#)

[9.5. Column Family](#)

[9.6. Cells](#)

[9.7. Versions](#)

[9.7.1. Versions and HBase Operations](#)

[9.7.2. Current Limitations](#)

In short, applications store data into an HBase table. Tables are made of rows and columns. All columns in HBase belong to a particular column family. Table cells — the intersection of row and column coordinates — are versioned. A cell's content is an uninterpreted array of bytes.

Table row keys are also byte arrays so almost anything can serve as a row key from strings to binary representations of longs or even serialized data structures. Rows in HBase tables are sorted by row key. The sort is byte-ordered. All table accesses are via the table row key — its primary key.

9.1. Conceptual View

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 column families named `contents` and `anchor`. In this example, `anchor` contains two columns (`anchor:cssnsi.com`, `anchor:my.look.ca`) and `contents` contains one column (`contents:html`).

Column Names

By convention, a column name is made of its column family prefix and a *qualifier*. For example, the column `contents:html` is of the column family

contents The colon character (:) delimits the column family from the column family *qualifier*.

Table 9.1. Table *webtable*

Row Key	Time Stamp	ColumnFamily contents	ColumnFamily anchor
"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>..."	

9.2. Physical View

Although at a conceptual level tables may be viewed as a sparse set of rows. Physically they are stored on a per-column family basis. New columns (i.e., columnfamily:column) can be added to any column family without pre-announcing them.

Table 9.2. 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"

Table 9.3. 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>..."

It is important to note in the diagram above that the empty cells shown in the conceptual view are not stored since they need not be in a column-oriented storage format. 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 and would also be 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 time stamp t6, the value of anchor:cnnsi.com from time stamp t9, the value of anchor:my.look.ca from time stamp t8.

9.3. Table

Tables are declared up front at schema definition time.

9.4. 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.

9.5. Column Family

Columns in 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 . 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 an 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.

9.6. Cells

A $\{row, column, version\}$ tuple exactly specifies a cell in HBase. Cell content is uninterpreted bytes

9.7. Versions

A $\{row, column, version\}$ tuple exactly specifies a cell in HBase. Its 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, a couple questions that often come up are:

- If multiple writes to a cell have the same version, are all versions maintained or just the last?^[12]
- Is it OK to write cells in a non-increasing version order?^[13]

Below we describe how the version dimension in HBase currently works^[14].

9.7.1. Versions and HBase Operations

In this section we look at the behavior of the version dimension for each of the core HBase operations.

9.7.1.1. 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.setMaxVersions\(\)](#)
- to return versions other than the latest, see [Get.setTimeRange\(\)](#)

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.

9.7.1.2. Default Get Example

The following Get will only retrieve the current version of the row

```
Get get = new Get(Bytes.toBytes("row1"));
Result r = htable.get(get);
byte[] b = r.getValue(Bytes.toBytes("cf"), Bytes.toBytes("attr")); // returns current version
```

9.7.1.3. Versioned Get Example

The following Get will return the last 3 versions of the row.

```
Get get = new Get(Bytes.toBytes("row1"));
get.setMaxVersions(3); // will return last 3 versions of row
Result r = htable.get(get);
byte[] b = r.getValue(Bytes.toBytes("cf"), Bytes.toBytes("attr")); // returns current version
List<KeyValue> kv = r.getColumn(Bytes.toBytes("cf"), Bytes.toBytes("attr")); // returns all v
```

9.7.1.4. 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 would overshadow.

9.7.1.4.1. Implicit Version Example

The following Put will be implicitly versioned by HBase with the current time.

```
Put put = new Put(Bytes.toBytes(row));
put.add(Bytes.toBytes("cf"), Bytes.toBytes("attr1"), Bytes.toBytes(data));
htable.put(put);
```

9.7.1.4.2. Explicit Version Example

The following Put has the version timestamp explicitly set.

```
Put put = new Put( Bytes.toBytes(row ));
long explicitTimeInMs = 555; // just an example
put.add(Bytes.toBytes("cf"), Bytes.toBytes("attr1"), explicitTimeInMs, Bytes.toBytes(data));
htable.put(put);
```

9.7.1.5. Delete

When performing a delete operation in HBase, there are two ways to specify the versions to be deleted

- Delete all versions older than a certain timestamp
- Delete the version at a specific timestamp

A delete can apply to a complete row, a complete column family, or to just one column. It is only in the last case that you can delete explicit versions. For the deletion of a row or all the columns within a family, it always works by deleting all cells older than a certain version.

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^[15]. 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.

9.7.2. Current Limitations

There are still some bugs (or at least 'undecided behavior') with the version dimension that will be addressed by later HBase releases.

9.7.2.1. Deletes mask Puts

Deletes mask puts, even puts that happened after the delete was entered^[16]. Remember that a delete writes a tombstone, which only disappears after then next major compaction has run. Suppose you do a delete of everything $\leq T$. After this you do a new put with a timestamp $\leq T$. 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.

9.7.2.2. 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...”^[17],

^[12] Currently, only the last written is fetchable.

^[13] Yes

^[14] 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.

^[15] When HBase does a major compaction, the tombstones are processed to actually remove the dead values, together with the tombstones themselves.

^[16] [HBASE-2256](#)

^[17] See *Garbage Collection* in [Bending time in HBase](#)

Chapter 10. Architecture

Table of Contents

[10.1. Client](#)

[10.1.1. Connections](#)

[10.1.2. WriteBuffer and Batch Methods](#)

[10.1.3. Filters](#)

[10.2. Daemons](#)

[10.2.1. Master](#)

[10.2.2. RegionServer](#)

[10.3. Regions](#)

[10.3.1. Region Size](#)

[10.3.2. Region Splits](#)

[10.3.3. Region Load Balancer](#)

[10.3.4. Store](#)

[10.4. Write Ahead Log \(WAL\)](#)

[10.4.1. Purpose](#)[10.4.2. WAL Flushing](#)[10.4.3. WAL Splitting](#)

10.1. Client

The HBase client [HTable](#) is responsible for finding RegionServers that are serving the particular row range of interest. It does this by querying the .META. and -ROOT catalog tables (TODO: Explain). After locating the required region(s), the client *directly* contacts the RegionServer serving that region (i.e., it does not go 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.

Administrative functions are handled through [HBaseAdmin](#)

10.1.1. Connections

For connection configuration information, see [Section 2.6.4, “Client configuration and dependencies connecting to an HBase cluster”](#).

[HTable](#) instances are not thread-safe. When creating HTable 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 [HConnectionManager](#).

10.1.2. WriteBuffer and Batch Methods

If [Section 11.7.1, “AutoFlush”](#) is turned off on [HTable](#), Puts are sent to RegionServers when the writebuffer is filled. The writebuffer is 2MB by default. Before an HTable instance is discarded, either `close()` or `flushCommits()` should be invoked so Puts will not be lost.

Note: `htable.delete(Delete);` does not go in the writebuffer! This only applies to Puts.

For additional information on write durability, review the [ACID semantics](#) page.

For fine-grained control of batching of Puts or Deletes, see the [batch](#) methods on HTable.

10.1.3. Filters

[Get](#) and [Scan](#) instances can be optionally configured with [filters](#) which are applied on the RegionServer.

10.2. Daemons

10.2.1. 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.

10.2.1.1. 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.

10.2.1.2. 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 HBaseAdmin method disableTable is invoked, it is serviced by the Master server.

10.2.1.3. Processes

The Master runs several background threads:

- LoadBalancer periodically reassign regions in the cluster.
- CatalogJanitor periodically checks and cleans up the .META. table.

10.2.2. RegionServer

HRegionServer is the RegionServer implementation. It is responsible for serving and managing regions.

10.2.2.1. 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 HBaseAdmin 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.

10.2.2.2. 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 HLog.

10.3. Regions

This chapter is all about Regions.

Note

Regions are comprised of a Store per Column Family.

10.3.1. Region Size

Region size is one of those tricky things, there are a few factors to consider:

- Regions are the basic element of availability and distribution.
- HBase scales by having regions across many servers. Thus if you have 2 regions for 16GB data, on a 20 node machine you are a net loss there.
- High region count has been known to make things slow, this is getting better, but it is probably better to have 700 regions than 3000 for the same amount of data.
- Low region count prevents parallel scalability as per point #2. This really cant be stressed enough, since a common problem is loading 200MB data into HBase then wondering why your awesome 10 node cluster is mostly idle.
- There is not much memory footprint difference between 1 region and 10 in terms of indexes, etc, held by the RegionServer.

Its probably best to stick to the default, perhaps going smaller for hot tables (or manually split hot regions to spread the load over the cluster), or go with a 1GB region size if your cell sizes tend to be largish (100k and up).

10.3.2. Region Splits

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 META, opens daughters on the parent's hosting RegionServer and then reports the split to the Master. See [Section 2.8.2.7, “Managed Splitting”](#) for how to manually manage splits (and for why you might do this)

10.3.3. Region Load Balancer

Periodically, and when there are not any regions in transition, a load balancer will run and move regions around to balance cluster load. The period at which it runs can be configured.

10.3.4. 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.

10.3.4.1. MemStore

The MemStore holds in-memory modifications to the Store. Modifications are KeyValues. When asked to flush, current memstore is moved to snapshot and is cleared. HBase continues to serve edits out of new memstore and backing snapshot until flusher reports in that the flush succeeded. At this point the snapshot is let go.

10.3.4.2. StoreFile (HFile)

10.3.4.2.1. 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](#).

10.3.4.2.2. HFile Tool

To view a textualized version of hfile content, you can do use the `org.apache.hadoop.hbase.io.hfile.HFile` tool. Type the following to see usage:

```
$ ${HBASE_HOME}/bin/hbase org.apache.hadoop.hbase.io.hfile.HFile
```

For example, to view the content of the file `hdfs://10.81.47.41:9000/hbase/TEST/1418428042/DSMP/4759508618286845475`, type the following:

```
$ ${HBASE_HOME}/bin/hbase org.apache.hadoop.hbase.io.hfile.HFile -v -f hdfs://10.81.47.41:9000/ht
```

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.

10.3.4.3. Compaction

There are two types of compactions: minor and major. Minor compactions will usually pick up a couple of the smaller adjacent files and rewrite them as one. Minors do not drop deletes or expired cells, only major compactions do this. Sometimes a minor compaction will pick up all the files in the store and in this case it actually promotes itself to being a major compaction. For a description of how a minor compaction picks files to compact, see the [ascii diagram in the Store source code](#).

After a major compaction runs there will be a single storefile per store, and this will help performance usually. Caution: major compactions rewrite all of the stores data and on a loaded system, this may not be tenable; major compactions will usually have to be [Section 2.8.2.7, “Managed Splitting”](#) on large systems.

10.4. Write Ahead Log (WAL)

10.4.1. Purpose

Each RegionServer adds updates (Puts, Deletes) to its write-ahead log (WAL) first, and then to the [Section 10.3.4.1, “MemStore”](#) for the affected [Section 10.3.4, “Store”](#). This ensures that HBase has durable writes. Without WAL, there is the possibility of data loss in the case of a RegionServer failure before each MemStore is flushed and new StoreFiles are written. [HLog](#) is the HBase WAL implementation, and there is one HLog instance per RegionServer.

The WAL is in HDFS in `/hbase/.logs/` with subdirectories per region.

For more general information about the concept of write ahead logs, see the Wikipedia [Write-Ahead Log](#) article.

10.4.2. WAL Flushing

TODO (describe).

10.4.3. WAL Splitting

10.4.3.1. How edits are recovered from a crashed RegionServer

When a RegionServer crashes, it will lose its ephemeral lease in ZooKeeper...TODO

10.4.3.2. `hbase.hlog.split.skip.errors`

When set to true, the default, any error encountered splitting will be logged, the problematic WAL will be moved into the `.corrupt` directory under the hbase rootdir, and processing will continue. If set to false, the exception will be propagated and the split logged as failed.^[18]

10.4.3.3. How EOFExceptions are treated when splitting a crashed RegionServers' WALs

If we get an EOF while splitting logs, we proceed with the split even when `hbase.hlog.split.skip.errors == false`. An EOF while reading the last log in the set of files to split is near-guaranteed since the RegionServer likely crashed mid-write of a record. But we'll continue even if we got an EOF reading other than the last file in the set.^[19]

^[18] See [HBASE-2958 When hbase.hlog.split.skip.errors is set to false, we fail the split but thats it](#). We need to do more than just fail split if this flag is set.

^[19] For background, see [HBASE-2643 Figure how to deal with eof splitting logs](#)

Chapter 11. Performance Tuning

Table of Contents

[11.1. Operating System](#)

- [11.1.1. Memory](#)
- [11.1.2. 64-bit](#)
- [11.1.3. Swapping](#)

[11.2. Java](#)

- [11.2.1. The Garbage Collector and HBase](#)

[11.3. Configurations](#)

- [11.3.1. Number of Regions](#)
- [11.3.2. Managing Compactions](#)
- [11.3.3. Compression](#)
- [11.3.4. `hbase.regionserver.handler.count`](#)
- [11.3.5. `hfile.block.cache.size`](#)
- [11.3.6. `hbase.regionserver.global.memstore.upperLimit`](#)
- [11.3.7. `hbase.regionserver.global.memstore.lowerLimit`](#)
- [11.3.8. `hbase.hstore.blockingStoreFiles`](#)
- [11.3.9. `hbase.hregion.memstore.block.multiplier`](#)

[11.4. Number of Column Families](#)

[11.5. Data Clumping](#)

[11.6. Batch Loading](#)

- [11.6.1. Table Creation: Pre-Creating Regions](#)

[11.7. HBase Client](#)

- [11.7.1. AutoFlush](#)
- [11.7.2. Scan Caching](#)
- [11.7.3. Scan Attribute Selection](#)
- [11.7.4. Close ResultScanners](#)
- [11.7.5. Block Cache](#)
- [11.7.6. Optimal Loading of Row Keys](#)
- [11.7.7. Turn off WAL on Puts](#)

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 DEBUG for the package `org.apache.hadoop.ipc` (That's right, for `hadoop.ipc`, NOT, `hbase.ipc`). Then tail the RegionServers log. Analyze.

To disable, set the logging level back to INFO level.

11.1. Operating System

11.1.1. Memory

RAM, RAM, RAM. Don't starve HBase.

11.1.2. 64-bit

Use a 64-bit platform (and 64-bit JVM).

11.1.3. Swapping

Watch out for swapping. Set swappiness to 0.

11.2. Java

11.2.1. The Garbage Collector and HBase

11.2.1.1. 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 that must be explicitly enabled in HBase 0.90.x (Its defaulted to be on in 0.92.x HBase). See `hbase.hregion.memstore.mslab.enabled` to true in your Configuration. See the cited slides for background and detail.

For more information about GC logs, see [Section 13.2.2, “JVM Garbage Collection Logs”](#).

11.3. Configurations

See [Section 2.8.2, “Recommended Configurations”](#).

11.3.1. Number of Regions

The number of regions for an HBase table is driven by the [Section 2.8.2.6, “Bigger Regions”](#). Also, see the architecture section on [Section 10.3.1, “Region Size”](#).

11.3.2. Managing Compactions

For larger systems, managing [compactions and splits](#) may be something you want to consider.

11.3.3. Compression

Production systems should use compression with their column family definitions. See [Appendix B, *Compression In HBase*](#) for more information.

11.3.4. `hbase.regionserver.handler.count`

See [hbase.regionserver.handler.count](#). This setting in essence sets how many requests are concurrently being processed inside the RegionServer at any one time. If set too high, then throughput may suffer as the concurrent requests contend; if set too low, requests will be stuck waiting to get into the machine. You can get a sense of whether you have too little or too many handlers by [Enabling RPC-level logging](#) on an individual RegionServer then tailing its logs (Queued requests consume memory).

11.3.5. `hfile.block.cache.size`

See [hfile.block.cache.size](#). A memory setting for the RegionServer process.

11.3.6. `hbase.regionserver.global.memstore.upperLimit`

See [hbase.regionserver.global.memstore.upperLimit](#). This memory setting is often adjusted for the RegionServer process depending on needs.

11.3.7. `hbase.regionserver.global.memstore.lowerLimit`

See [hbase.regionserver.global.memstore.lowerLimit](#). This memory setting is often adjusted for the RegionServer process depending on needs.

11.3.8. `hbase.hstore.blockingStoreFiles`

See [hbase.hstore.blockingStoreFiles](#). If there is blocking in the RegionServer logs, increasing this can help.

11.3.9. `hbase.hregion.memstore.block.multiplier`

See [hbase.hregion.memstore.block.multiplier](#). If there is enough RAM, increasing this can help.

11.4. Number of Column Families

See [Section 6.2, “On the number of column families”](#).

11.5. Data Clumping

If all your data is being written to one region, then re-read the section on processing [timeseries](#) data.

11.6. Batch Loading

Use the bulk load tool if you can. See [Bulk Loads](#). Otherwise, pay attention to the below.

11.6.1. 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. An example of pre-creation using hex-keys is as follows (note: this example may need to be tweaked to the individual applications keys):

```
public static boolean createTable(HBaseAdmin admin, HTableDescriptor table, byte[][] splits)
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 numRegions) {
    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;
}
```

11.7. HBase Client

11.7.1. AutoFlush

When performing a lot of Puts, make sure that `setAutoFlush` is set to `false` on your [HTable](#) instance. Otherwise, the Puts will be sent one at a time to the RegionServer. Puts added via `htable.add(Put)` and `htable.add(<List> Put)` wind up in the same write buffer. If `autoFlush = false`, these messages are not sent until the write-buffer is filled. To explicitly flush the messages, call `flushCommits`. Calling `close` on the `HTable` instance will invoke `flushCommits`.

11.7.2. 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.

11.7.3. 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.

11.7.4. 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 = htable.getScanner(scan);
try {
    for (Result r = rs.next(); r != null; r = rs.next()) {
        // process result...
    } finally {
        rs.close(); // always close the ResultScanner!
    }
}
htable.close();
```

11.7.5. 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.

11.7.6. 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.

11.7.7. Turn off WAL on Puts

A frequently discussed option for increasing throughput on Puts is to call `writeToWAL(false)`. Turning this off means that the RegionServer will *not* write the Put to the Write Ahead Log, only into the memstore, HOWEVER the consequence is that if there is a RegionServer failure *there will be data loss*. If `writeToWAL(false)` is used, do so with extreme caution. You may find in actuality that it makes little difference if your load is well distributed across the cluster.

In general, it is best to use WAL for Puts, and where loading throughput is a concern to use [bulk loading](#) techniques instead.

Chapter 12. Bloom Filters

Table of Contents

[12.1. Configurations](#)

[12.1.1. HColumnDescriptor option](#)

[12.1.2. io.hfile.bloom.enabled global kill switch](#)

[12.1.3. io.hfile.bloom.error.rate](#)

[12.1.4. io.hfile.bloom.max.fold](#)

[12.2. Bloom StoreFile footprint](#)

[12.2.1. BloomFilter in the StoreFile FileInfo data structure](#)

[12.2.2. BloomFilter entries in StoreFile metadata](#)

Bloom filters were developed over in [HBase-1200 Add bloomfilters](#).^{[20][21]}

12.1. Configurations

Blooms are enabled by specifying options on a column family in the HBase shell or in java code as specification on `org.apache.hadoop.hbase.HColumnDescriptor`.

12.1.1. HColumnDescriptor option

Use `HColumnDescriptor.setBloomFilterType(NONE|ROW|ROWCOL)` to enable blooms per Column Family. Default = NONE for no bloom filters. If ROW, the hash of the row will be added to the bloom on each insert. If ROWCOL, the hash of the row + column family + column family qualifier will be added to the bloom on each key insert.

12.1.2. io.hfile.bloom.enabled global kill switch

`io.hfile.bloom.enabled` in Configuration serves as the kill switch in case something goes wrong. Default = true.

12.1.3. io.hfile.bloom.error.rate

`io.hfile.bloom.error.rate` = average false positive rate. Default = 1%. Decrease rate by $\frac{1}{2}$ (e.g. to .5%) == +1 bit per bloom entry.

12.1.4. io.hfile.bloom.max.fold

`io.hfile.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.

12.2. 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.

12.2.1. BloomFilter in the StoreFile FileInfo data structure

12.2.1.1. BLOOM_FILTER_TYPE

FileInfo has a BLOOM_FILTER_TYPE entry which is set to NONE, ROW or ROWCOL.

12.2.2. BloomFilter entries in StoreFile metadata

12.2.2.1. BLOOM_FILTER_META

BLOOM_FILTER_META holds Bloom Size, Hash Function used, etc. Its small in size and is cached on StoreFile.Reader load

12.2.2.2. BLOOM_FILTER_DATA

BLOOM_FILTER_DATA is the actual bloomfilter data. Obtained on-demand. Stored in the LRU cache, if it is enabled (Its enabled by default).

[20] 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](#).

[21] 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.

Chapter 13. Troubleshooting and Debugging HBase

Table of Contents

[13.1. General Guidelines](#)

[13.2. Logs](#)

[13.2.1. Log Locations](#)

[13.2.2. JVM Garbage Collection Logs](#)

[13.3. Tools](#)

[13.3.1. Builtin Tools](#)

[13.3.2. External Tools](#)

[13.4. Client](#)

[13.4.1. ScannerTimeoutException](#)

[13.4.2. Shell or client application throws lots of scary exceptions during normal operation](#)

[13.4.3. Long Client Pauses With Compression](#)

[13.5. NameNode](#)

[13.5.1. HDFS Utilization of Tables and Regions](#)

[13.6. RegionServer](#)

[13.6.1. Startup Errors](#)

[13.6.2. Runtime Errors](#)

[13.6.3. Shutdown Errors](#)

[13.7. Master](#)

[13.7.1. Startup Errors](#)

[13.7.2. Shutdown Errors](#)

[13.8. ZooKeeper](#)

[13.8.1. Startup Errors](#)

[13.8.2. ZooKeeper, The Cluster Canary](#)

[13.9. Amazon EC2](#)

[13.9.1. ZooKeeper does not seem to work on Amazon EC2](#)

[13.9.2. Instability on Amazon EC2](#)

13.1. 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 or [search-hadoop.com](#) should return some hits for those exceptions you're seeing.

An error rarely comes alone in 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 `xcievers` (the two most important initial settings, see [Section 2.2.4, “ulimit and nproc”](#)) 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 [Section 11.2.1.1, “Long GC pauses”](#) above.

13.2. 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>-jobtracker-<hostname>.log`

HMaster: `$HBASE_HOME/logs/hbase-<user>-master-<hostname>.log`

RegionServer: `$HBASE_HOME/logs/hbase-<user>-regionserver-<hostname>.log`

ZooKeeper: `TODO`

13.2.1. 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.

13.2.1.1. 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 is typically run on the NameNode server as well.

13.2.1.2. 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 log for MapReduce task execution.

13.2.2. JVM Garbage Collection 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` add:

```
export HBASE_OPTS="-XX:+UseConcMarkSweepGC -verbose:gc -XX:+PrintGCDetails -XX:+PrintGCTi
```

Adjust the log directory to wherever you log. Note: The GC log does NOT roll automatically, so you'll have to keep an eye on it so it doesn't fill up the disk.

At this point you should see logs like so:

```
64898.952: [GC [1 CMS-initial-mark: 2811538K(3055704K)] 2812179K(3061272K), 0.0007360 secs] [Times: 1
64898.953: [CMS-concurrent-mark-start]
64898.971: [GC 64898.971: [ParNew: 5567K->576K(5568K), 0.0101110 secs] 2817105K->2812715K(3061272K),
```

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, real=2.49 secs]
64901.445: [CMS-concurrent-preclean-start]
64901.453: [GC 64901.453: [ParNew: 5505K->573K(5568K), 0.0062440 secs] 2868746K->2864292K(3061272K)
64901.476: [GC 64901.476: [ParNew: 5563K->575K(5568K), 0.0072510 secs] 2869283K->2864837K(3061272K)
64901.500: [GC 64901.500: [ParNew: 5517K->573K(5568K), 0.0120390 secs] 2869780K->2865267K(3061272K)
64901.529: [GC 64901.529: [ParNew: 5507K->569K(5568K), 0.0086240 secs] 2870200K->2865742K(3061272K)
64901.554: [GC 64901.555: [ParNew: 5516K->575K(5568K), 0.0107130 secs] 2870689K->2866291K(3061272K)
64901.578: [CMS-concurrent-preclean: 0.070/0.133 secs] [Times: user=0.48 sys=0.01, 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)
64901.609: [GC 64901.609: [ParNew: 5512K->569K(5568K), 0.0063370 secs] 2871771K->2867322K(3061272K)
64901.615: [CMS-concurrent-abortable-preclean: 0.007/0.037 secs] [Times: user=0.13 sys=0.00, real=
64901.616: [GC [YG occupancy: 645 K (5568 K)] 64901.616: [Rescan (parallel), 0.0020210 secs] 64901.618: [w
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) , 0.0020210 secs]64901.618:[w
```

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
... lines removed ...
64904.936:[GC 64904.936:[ParNew: 5532K->568K(5568K), 0.0070720 secs] 1365024K->1360689K(3061272
64904.953:[CMS-concurrent-sweep: 2.030/3.332 secs][Times: user=9.57 sys=0.26, real=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 its too small, objects are promoted to old gen too quickly). In the below we constrain new gen size to 64m.

Add this to `HBASE_OPTS`:

```
export HBASE_OPTS="-XX:NewSize=64m -XX:MaxNewSize=64m <cms options from above> <gc logging
```

For more information on GC pauses, see the [3 part blog post](#) by Todd Lipcon and [Section 11.2.1.1, "Long GC pauses"](#) above.

13.3. Tools

13.3.1. Builtin Tools

13.3.1.1. Master Web Interface

The Master starts a web-interface on port 60010 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.

13.3.1.2. RegionServer Web Interface

RegionServers starts a web-interface on port 60030 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 [Chapter 7, Metrics](#) for more information in metric definitions.

13.3.2. External Tools

13.3.2.1. `search-hadoop.com`

search-hadoop.com indexes all the mailing lists and [JIRA](#), it's really helpful when looking for Hadoop/HBase-specific issues.

13.3.2.2. `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).

13.3.2.3. `top`

`top` is probably one of the most important tool 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 "l" will give you the detail of how each CPU is used instead of the average for all of them like shown here.

13.3.2.4. 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<1 Mar04 9855:48 /usr/java/jdk1.6.0_14/bin/java -Xr
```

13.3.2.5. 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's a few example:

The main thread of a RegionServer that's waiting for something to do from the master:


```

"regionserver60020" prio=10 tid=0x0000000040ab4000 nid=0x45cf waiting on condition [0x0000
java.lang.Thread.State: TIMED_WAITING (parking)
  at sun.misc.Unsafe.park(Native Method)
    - parking to wait for <0x00007f16cd5c2f30> (a java.util.concurrent.locks.AbstractQueuee
  at java.util.concurrent.locks.LockSupport.parkNanos(LockSupport.java:198)
  at java.util.concurrent.locks.AbstractQueuedSynchronizer$ConditionObject.awaitNanos(A
  at java.util.concurrent.LinkedBlockingQueue.poll(LinkedBlockingQueue.java:395)
  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.w
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 <0x00007f16cbl4b3a8> (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(DelegatingMethodAccessorImpl.java:
  at java.lang.reflect.Method.invoke(Method.java:597)
  at org.apache.hadoop.io.retry.RetryInvocationHandler.invokeMethod(RetryInvocationHandl
  at org.apache.hadoop.io.retry.RetryInvocationHandler.invoke(RetryInvocationHandler.jav
  at $Proxy1.complete(Unknown Source)
  at org.apache.hadoop.hdfs.DFSClient$DFSOutputStream.closeInternal(DFSCClient.java:3390)
    - locked <0x00007f16cbl4b470> (a org.apache.hadoop.hdfs.DFSClient$DFSOutputStream)
  at org.apache.hadoop.hdfs.DFSClient$DFSOutputStream.close(DFSCClient.java:3304)
  at org.apache.hadoop.fs.FSDataOutputStream$PositionCache.close(FSDataOutputStream.java:
  at org.apache.hadoop.fs.FSDataOutputStream.close(FSDataOutputStream.java:86)
  at org.apache.hadoop.hbase.io.hfile.HFile$Writer.close(HFile.java:650)
  at org.apache.hadoop.hbase.regionserver.StoreFile$Writer.close(StoreFile.java:853)
    - 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(Store.java:135
  at org.apache.hadoop.hbase.regionserver.HRegion.internalFlushCache(HRegion.java:907)
  at org.apache.hadoop.hbase.regionserver.HRegion.internalFlushCache(HRegion.java:834)
  at org.apache.hadoop.hbase.regionserver.HRegion.flushCache(HRegion.java:786)
  at org.apache.hadoop.hbase.regionserver.MemStoreFlusher.flushRegion(MemStoreFlusher.ja
  at org.apache.hadoop.hbase.regionserver.MemStoreFlusher.flushRegion(MemStoreFlusher.ja
  at org.apache.hadoop.hbase.regionserver.MemStoreFlusher.run(MemStoreFlusher.java: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 con
java.lang.Thread.State: WAITING (parking)
  at sun.misc.Unsafe.park(Native Method)
    - parking to wait for <0x00007f16cd3f8dd8> (a java.util.concurrent.locks.AbstractQueuee
  at java.util.concurrent.locks.LockSupport.park(LockSupport.java:158)
  at java.util.concurrent.locks.AbstractQueuedSynchronizer$ConditionObject.await(Abstra
  at java.util.concurrent.LinkedBlockingQueue.take(LinkedBlockingQueue.java:358)
  at org.apache.hadoop.hbase.ipc.HBaseServer$Handler.run(HBaseServer.java: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 [0x0
java.lang.Thread.State: RUNNABLE
    at org.apache.hadoop.hbase.regionserver.KeyValueHeap.<init>(KeyValueHeap.java:56)
    at org.apache.hadoop.hbase.regionserver.StoreScanner.<init>(StoreScanner.java:79)
    at org.apache.hadoop.hbase.regionserver.Store.getScanner(Store.java:1202)
    at org.apache.hadoop.hbase.regionserver.HRegion$RegionScanner.<init>(HRegion.java:2209)
    at org.apache.hadoop.hbase.regionserver.HRegion.instantiateInternalScanner(HRegion.java:
    at org.apache.hadoop.hbase.regionserver.HRegion.getScanner(HRegion.java:1055)
    at org.apache.hadoop.hbase.regionserver.HRegion.getScanner(HRegion.java:1039)
    at org.apache.hadoop.hbase.regionserver.HRegion.getLastIncrement(HRegion.java:2875)
    at org.apache.hadoop.hbase.regionserver.HRegion.incrementColumnValue(HRegion.java:2978)
    at org.apache.hadoop.hbase.regionserver.HRegionServer.incrementColumnValue(HRegionSer
    at sun.reflect.GeneratedMethodAccessor20.invoke(Unknown Source)
    at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:
    at java.lang.reflect.Method.invoke(Method.java:597)
    at org.apache.hadoop.hbase.ipc.HBaseRPC$Server.call(HBaseRPC.java:560)
    at org.apache.hadoop.hbase.ipc.HBaseServer$Handler.run(HBaseServer.java: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=0x00007f16
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$1)
    - locked <0x00007f17d5b68be8> (a java.util.Collections$UnmodifiableSet)
    - locked <0x00007f1877959b50> (a sun.nio.ch.EPollSelectorImpl)
    at sun.nio.ch.SelectorImpl.select(SelectorImpl.java:80)
    at org.apache.hadoop.net.SocketIOWithTimeout$SelectorPool.select(SocketIOWithTimeout.
    at org.apache.hadoop.net.SocketIOWithTimeout.doIO(SocketIOWithTimeout.java:157)
    at org.apache.hadoop.net.SocketInputStream.read(SocketInputStream.java:155)
    at org.apache.hadoop.net.SocketInputStream.read(SocketInputStream.java:128)
    at java.io.FilterInputStream.read(FilterInputStream.java:116)
    at org.apache.hadoop.ipc.Client$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 condition [0x00007
--
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 <0x00007f6dlcd28f80> (a org.apache.hadoop.ipc.Client$Call)
    at org.apache.hadoop.ipc.RPC$Invoker.invoke(RPC.java:220)
    at $Proxy1recoverBlock(Unknown Source)
    at org.apache.hadoop.hdfs.DFSCClient$DFSOutputStream.processDatanodeError(DFSCClient.jav
    at org.apache.hadoop.hdfs.DFSCClient$DFSOutputStream.<init>(DFSCClient.java:2832)
    at org.apache.hadoop.hdfs.DFSCClient.append(DFSCClient.java:529)
```

```

at org.apache.hadoop.hdfs.DistributedFileSystem.append(DistributedFileSystem.java:186)
at org.apache.hadoop.fs.FileSystem.append(FileSystem.java:530)
at org.apache.hadoop.hbase.util.FSUtils.recoverFileLease(FSUtils.java:619)
at org.apache.hadoop.hbase.regionserver.wal.HLog.splitLog(HLog.java:1322)
at org.apache.hadoop.hbase.regionserver.wal.HLog.splitLog(HLog.java:1210)
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)

```

13.3.2.6. OpenTSDB

[OpenTSDB](#) is an excellent alternative to Ganglia as it uses 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 HBase. You can then go down at the machine level and get even more detailed metrics.

13.3.2.7. 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.

13.4. Client

13.4.1. ScannerTimeoutException

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.

13.4.2. 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.
```

13.4.3. Long Client Pauses With Compression

This is a fairly frequent question on the 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 [Section 11.6.1, “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 [Section 11.3, “Configurations”](#) for cluster configuration, particularly `hbase.hstore.blockingStoreFiles`, `hbase.hregion.memstore.block.multiplier`, `MAX_FILESIZE` (region size), and `MEMSTORE_FLUSH_SIZE`.

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, 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 flush size 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.

13.5. NameNode

13.5.1. HDFS Utilization of Tables and Regions

To determine how much space HBase is using on HDFS use the `hadoop fs` 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.

13.6. RegionServer

13.6.1. Startup Errors

13.6.1.1. 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 localhost.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
```

13.6.1.2. Compression Link Errors

Since compression algorithms such as LZ0 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 library
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](#).

13.6.2. Runtime Errors

13.6.2.1. java.io.IOException...(Too many open files)

See the Getting Started section on [ulimit and nproc configuration](#).

13.6.2.2. xceiverCount 258 exceeds the limit of concurrent xcievers 256

This typically shows up in the DataNode logs.

See the Getting Started section on [xceivers configuration](#).

13.6.2.3. System instability, and the presence of "java.lang.OutOfMemoryError: unable to create new 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.

13.6.2.4. 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 ms, ten times longe
2009-02-24 10:01:33,516 WARN org.apache.hadoop.hbase.util.Sleeper: We slept xxx ms, ten times longe
2009-02-24 10:01:36,472 WARN org.apache.hadoop.hbase.regionserver.HRegionServer: unable to repor
```

... or see full GC compactions then you may be experiencing full GC's.

13.6.2.5. "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.

13.6.2.6. 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@35581lec
java.io.IOException: TIMED OUT
    at org.apache.zookeeper.ClientCnxn$SendThread.run(ClientCnxn.java:906)
WARN org.apache.hadoop.hbase.util.Sleeper: We slept 79410ms, ten times longer than scheduled: 5000
INFO org.apache.zookeeper.ClientCnxn: Attempting connection to server hostname/IP:PORT
INFO org.apache.zookeeper.ClientCnxn: Priming connection to java.nio.channels.SocketChannel[conr
INFO org.apache.zookeeper.ClientCnxn: Server connection successful
WARN org.apache.zookeeper.ClientCnxn: Exception closing session 0x278bd16a96000d to sun.nio.ch.S
java.io.IOException: Session Expired
    at org.apache.zookeeper.ClientCnxn$SendThread.readConnectResult(ClientCnxn.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 expired
```

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>1200000</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 to lower the memory load on each machine (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 [Section 13.8.2, “ZooKeeper, The Cluster Canary”](#) for other general information about ZooKeeper troubleshooting.

13.6.2.7. 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 .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.

13.6.3. Shutdown Errors

13.7. Master

13.7.1. Startup Errors

13.7.1.1. 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.

13.7.2. Shutdown Errors

13.8. ZooKeeper

13.8.1. Startup Errors

13.8.1.1. 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.

13.8.2. 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.

13.9. Amazon EC2

13.9.1. 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/10.244.9.171:2181
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.

13.9.2. Instability on Amazon EC2

Questions on HBase and Amazon EC2 come up frequently on the HBase dist-list. Search for old threads using [Search Hadoop](#)

Chapter 14. Building HBase

Table of Contents

[14.1. Building in snappy compression support](#)

[14.2. Adding an HBase release to Apache's Maven Repository](#)

This chapter will be of interest only to those building HBase from source.

14.1. Building in snappy compression support

<p>Pass `-Dsnappy` to trigger the snappy maven profile for building snappy native libs into hbase.</p>

14.2. Adding an HBase release to Apache's Maven Repository

Follow the instructions at [Publishing Maven Artifacts](#). The 'trick' to making it all work is answering the questions put to you by the mvn release plugin properly, making sure it is using the actual branch AND before doing the mvn release:perform step, VERY IMPORTANT, hand edit the release.properties file that was put under \${HBASE_HOME} by the previous step, release:perform. You need to edit it to make it point at right locations in SVN.

If you see run into the below, its because you need to edit version in the pom.xml and add `-SNAPSHOT` to the version (and commit).

```
[INFO] Scanning for projects...
[INFO] Searching repository for plugin with prefix: 'release'.
[INFO] -----
[INFO] Building HBase
[INFO]   task-segment: [release:prepare] (aggregator-style)
[INFO] -----
[INFO] [release:prepare {execution: default-cli}]
[INFO] -----
[ERROR] BUILD FAILURE
[INFO] -----
[INFO] You don't have a SNAPSHOT project in the reactor projects list.
[INFO] -----
[INFO] For more information, run Maven with the -e switch
[INFO] -----
[INFO] Total time: 3 seconds
[INFO] Finished at: Sat Mar 26 18:11:07 PDT 2011
[INFO] Final Memory: 35M/423M
[INFO] -----
```

Chapter 15. Developing HBase

Table of Contents

[15.1. IDEs](#)

[15.1.1. Eclipse](#)

[15.2. Unit Tests](#)

[15.2.1. Mockito](#)

This chapter will be of interest only to those developing HBase (i.e., as opposed to using it).

15.1. IDEs

15.1.1. Eclipse

See [HBASE-3678 Add Eclipse-based Apache Formatter to HBase Wiki](#) for an Eclipse formatter to help ensure your code conforms to HBase's coding conventions. The issue includes instructions for loading the attached formatter.

15.2. Unit Tests

In HBase we use [JUnit](#) 4. If you need to run miniclusters of HDFS, ZooKeeper, HBase, or MapReduce testing, be sure to checkout the `HBaseTestingUtility`. Alex Baranau of Sematext describes how it can be used in [HBase Case-Study: Using HBaseTestingUtility for Local Testing and Development](#) (2010).

15.2.1. Mocito

Sometimes you don't need a full running server unit testing. For example, some methods can make do with 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`. In these cases, you may be able to get away with a mocked `Server` instance. For example:

Appendix A. Tools

Table of Contents

[A.1. HBase hbck](#)

[A.2. HFile Tool](#)

[A.3. WAL Tools](#)

[A.3.1. HLog tool](#)

[A.4. Compression Tool](#)

[A.5. Node Decommission](#)

[A.5.1. Rolling Restart](#)

Here we list HBase tools for administration, analysis, fixup, and debugging.

A.1. HBase hbck

An *fsck* for your HBase install

To run `hbck` against your HBase cluster run

```
$ ./bin/hbase hbck
```

At the end of the commands output it prints *OK* or *INCONSISTENCY*. If your cluster reports inconsistencies, pass `-details` to see more detail emitted. If inconsistencies, run `hbck` a few times because the inconsistency may be transient (e.g. cluster is starting up or a region is splitting). Passing `-fix` may correct the inconsistency (This latter is an experimental feature).

A.2. HFile Tool

See [Section 10.3.4.2.2, “HFile Tool”](#).

A.3. WAL Tools

A.3.1. HLog tool

The main method on HLog offers manual split and dump facilities. Pass it WALs or the product of a split, the content of the `recovered.edits` directory.

You can get a textual dump of a WAL file content by doing the following:

```
$ ./bin/hbase org.apache.hadoop.hbase.regionserver.wal.HLog --dump hdfs://example.org:9000/hba
```

The return code will be non-zero if issues with the file so you can test wholesomeness of file by redirecting STDOUT to `/dev/null` and testing the program return.

Similarly you can force a split of a log file directory by doing:

```
$ ./bin/hbase org.apache.hadoop.hbase.regionserver.wal.HLog --split hdfs://example.org:9000/hba
```

A.4. Compression Tool

See [Section A.4, “Compression Tool”](#).

A.5. 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 [Load Balancer](#) below.

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. In HBase 0.90.2, we added facility for having a node gradually shed its load and then shutdown itself down. HBase 0.90.2 added the

`graceful_stop.sh` script. Here is its usage:

```
$. /bin/graceful_stop.sh
Usage: graceful_stop.sh [--config &conf-dir>] [--restart] [--reload] [--thrift] [--rest] &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 stopped server
debug      Move offloaded regions back on to the stopped server
hostname   Hostname of server we are to stop
```

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. Check the list of RegionServers in the master UI for how HBase is referring to servers. Its usually hostname but can also be FQDN. 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 reenale, do:

```
hbase(main):001:0> balance_switch true
false
0 row(s) in 0.3590 seconds
```

A.5.1. Rolling Restart

You can also ask this script to restart a RegionServer after the shutdown AND move its old regions back into place. The latter you might do to retain data locality. A primitive rolling restart might be effected by running something like the following:

```
$ for i in `cat conf/regionserver|sort`; do ./bin/graceful_stop.sh --restart --reload --debug ;
```

Tail the output of `/tmp/log.txt` to follow the scripts progress. The above does RegionServers only. Be sure to disable the load balancer before doing the above. You'd need to do the master update separately. Do it before you run the above script. Here is a pseudo-script for how you might craft a rolling restart script:

1. Untar your release, make sure of its configuration and then rsync it across the cluster. If this is 0.90.2, patch it with HBASE-3744 and HBASE-3756.

2. Run hbck to ensure the cluster consistent

```
$. /bin/hbase hbck
```

Effect repairs if inconsistent.

3. Restart the Master:

```
$. /bin/hbase-daemon.sh stop master; ./bin/hbase-daemon.sh start master
```

4. Disable the region balancer:

```
$ echo "balance_switch false" | ./bin/hbase
```

5. Run the `graceful_stop.sh` script per RegionServer. For example:

```
$ for i in `cat conf/regionserver|sort`; do ./bin/graceful_stop.sh --restart --reload
```

If you are running thrift or rest servers on the RegionServer, pass `--thrift` or `--rest` options (See usage for `graceful_stop.sh` script).

6. Restart the Master again. This will clear out dead servers list and reenale the balancer.

7. Run hbck to ensure the cluster is consistent.

Appendix B. Compression In HBase

Table of Contents

[B.1. CompressionTest Tool](#)

[B.2. hbase.regionserver.codecs](#)

[B.3. LZ0](#)

[B.4. GZIP](#)[B.5. SNAPPY](#)

B.1. CompressionTest Tool

HBase includes a tool to test compression is set up properly. To run it, type `/bin/hbase org.apache.hadoop.hbase.util.CompressionTest`. This will emit usage on how to run the tool.

B.2. `hbase.regionserver.codecs`

To have a RegionServer test a set of codecs and fail-to-start if any code is missing or misinstalled, add the configuration `hbase.regionserver.codecs` to your `hbase-site.xml` with a value of codecs to test on startup. For example if the `hbase.regionserver.codecs` value is `lzo,gz` and if `lzo` is not present or improperly installed, the misconfigured RegionServer will fail to start.

Administrators might make use of this facility to guard against the case where a new server is added to cluster but the cluster requires install of a particular coded.

B.3. LZ0

Unfortunately, HBase cannot ship with LZ0 because of the licensing issues; HBase is Apache-licensed, LZ0 is GPL. Therefore LZ0 install is to be done post-HBase install. See the [Using LZ0 Compression](#) wiki page for how to make LZ0 work with HBase.

A common problem users run into when using LZ0 is that while initial setup of the cluster runs smooth, a month goes by and some sysadmin goes to add a machine to the cluster only they'll have forgotten to do the LZ0 fixup on the new machine. In versions since HBase 0.90.0, we should fail in a way that makes it plain what the problem is, but maybe not.

See [Section B.2, “`hbase.regionserver.codecs`”](#) for a feature to help protect against failed LZ0 install.

B.4. GZIP

GZIP will generally compress better than LZ0 though slower. For some setups, better compression may be preferred. Java will use java's GZIP unless the native Hadoop libs are available on the CLASSPATH; in this case it will use native compressors instead (If the native libs are NOT present, you will see lots of *Got brand-new compressor* reports in your logs; see [Q:](#)).

B.5. SNAPPY

If snappy is installed, HBase can make use of it (courtesy of [hadoop-snappy](#)).

1. Build and install [snappy](#) on all nodes of your cluster.
2. Use `CompressionTest` to verify snappy support is enabled and the libs can be loaded ON ALL NODES of your cluster:

```
$ hbase org.apache.hadoop.hbase.util.CompressionTest hdfs://host/path/to/hbase snapp
```

3. Create a column family with snappy compression and verify it in the hbase shell:

```
$ hbase> create 't1',{ NAME => 'cf1', COMPRESSION => 'SNAPPY' }  
hbase> describe 't1'
```

In the output of the "describe" command, you need to ensure it lists "COMPRESSION => 'SNAPPY'"

Appendix C. FAQ

C.1. [General](#)

[Are there other HBase FAQs?](#)

[Does HBase support SQL?](#)

[How does HBase work on top of HDFS?](#)

[Can I change a table's rowkeys?](#)

[Why are logs flooded with '2011-01-10 12:40:48.407 INFO](#)

[org.apache.hadoop.io.compress.CodecPool: Got brand-new compressor' messages?](#)

C.2. [EC2](#)

[Why doesn't my remote java connection into my ec2 cluster work?](#)

C.3. [Building HBase](#)

[When I build, why do I always get Unable to find resource 'VM_global_library.vm'?](#)

C.4. [Runtime](#)

[Loading, why do I see pauses when loading HBase?](#)

[Why does my RegionServer hang inexplicitly?](#)

[Why do I see double the actual count of region servers, once by domain name and then by IP?](#)

C.5. [How do I...?](#)

[Secondary Indexes in HBase?](#)

[Store \(fill in the blank\) in HBase?](#)

[Back up my HBase Cluster?](#)

C.1. General

[Are there other HBase FAQs?](#)

[Does HBase support SQL?](#)

[How does HBase work on top of HDFS?](#)

[Can I change a table's rowkeys?](#)

[Why are logs flooded with '2011-01-10 12:40:48.407 INFO](#)

[org.apache.hadoop.io.compress.CodecPool: Got brand-new compressor' messages?](#)

[Are there other HBase FAQs?](#)

See the FAQ that is up on the wiki, [HBase Wiki FAQ](#).

[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 [Chapter 9, Data Model](#) section for examples on the HBase client.

How does HBase work on top of HDFS?

[HDFS](#) is a distributed file system that is well suited for the storage of large files. It's 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. See the [Chapter 9, Data Model](#) and [Chapter 10, Architecture](#) sections for more information on how HBase achieves its goals.

Can I change a table's rowkeys?

No. See [Section 6.6, "Immutability of Rowkeys"](#).

Why are logs flooded with '2011-01-10 12:40:48,407 INFO org.apache.hadoop.io.compress.CodecPool: Got brand-new compressor' messages?

Because 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.

C.2. EC2

[Why doesn't my remote .java connection into my ec2 cluster work?](#)

Why doesn't my remote java connection into my ec2 cluster work?

See Andrew's answer here, up on the user list: [Remote Java client connection into EC2 instance](#).

C.3. Building HBase

[When I build, why do I always get Unable to find resource 'VM_global_library.vm'?](#)

When I build, why do I always get Unable to find resource 'VM_global_library.vm'?

Ignore it. Its not an error. It is [officially ugly](#) though.

C.4. Runtime

[Loading, why do I see pauses when loading HBase?](#)

[Why does my RegionServer hang inexplicitly?](#)

[Why do I see double the actual count of region servers, once by domain name and then by IP?](#)

Loading, why do I see pauses when loading HBase?

If compression is enabled, see this thread up on the user list, [Long client pauses with compression](#).

Why does my RegionServer hang inexplicitly?

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.

Why do I see double the actual count of regionserver, once by domain name and then by IP?

Fix your DNS. In versions of HBase before 0.92.x, reverse DNS needs to give same answer as forward lookup. See [HBASE 3431 Regionserver is not using the name given it by the master: double entry in master listing of servers](#) for gorey details.

C.5. How do I...?

[Secondary Indexes in HBase?](#)

[Store \(fill in the blank\) in HBase?](#)

[Back up my HBase Cluster?](#)

[Secondary Indexes in HBase?](#)

For a useful introduction to the issues involved maintaining a secondary Index in a store like HBase, see the David Butler message in this thread, [HBase, mail # user - Stargate+hbase](#)

[Store \(fill in the blank\) in HBase?](#)

See [Section 6.7, “Supported Datatypes”](#).

[Back up my HBase Cluster?](#)

See [HBase Backup Options](#) over on the Sematext Blog.

[Appendix D. 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

Ted Dunning redid YCSB so its mavenized and added facility for verifying workloads. See [Ted Dunning's YCSB](#).

Index

C

Cells, [Cells](#)

Column Family, [Column Family](#)

Column Family Qualifier, [Column Family](#)

Compression, [Compression In HBase](#)

H

Hadoop, [Hadoop](#)

N

nproc, [ulimit and nproc](#)

U

ulimit, [ulimit and nproc](#)

V

Versions, [Versions](#)

X

xcievers, [dfs.datanode.max.xcievers](#)

Z

ZooKeeper, [ZooKeeper](#)