17.9.2.3. Maximum region size

For large tables in production scenarios, maximum region size is mostly limited by compactions - very large compactions, esp. major, can degrade cluster performance. Currently, the recommended maximum region size is 10-20Gb, and 5-10Gb is optimal. For older 0.90.x codebase, the upper-bound of regionsize is about 4Gb, with a default of 256Mb.

The size at which the region is split into two is generally configured via `hbase.hregion.max.filesize`; for details, see Section 9.7.4, "Region Splits".

If you cannot estimate the size of your tables well, when starting off, it's probably best to stick to the default region size, perhaps going smaller for hot tables (or manually split hot regions to spread the load over the cluster), or go with larger region sizes if your cell sizes tend to be largish (100k and up).

In HBase 0.98, experimental stripe compactions feature was added that would allow for larger regions, especially for log data. See Section 9.7.6.7.3, "Experimental: Stripe Compactions".

17.9.2.4. Total data size per region server

According to above numbers for region size and number of regions per region server, in an optimistic estimate 10 GB x 100 regions per RS will give up to 1TB served per region server, which is in line with some of the reported multi-PB use cases. However, it is important to think about the data vs cache size ratio at the RS level. With 1TB of data per server and 10 GB block cache, only 1% of the data will be cached, which may barely cover all block indices.

17.9.3. Initial configuration and tuning

First, see Section 2.6, "The Important Configurations". Note that some configurations, more than others, depend on specific scenarios. Pay special attention to:

- `hbase.regionserver.handler.count` - request handler thread count, vital for high-throughput workloads.
- Section 2.6.2.6, "Configuring the size and number of WAL files" - the blocking number of WAL files depends on your memstore configuration and should be set accordingly to prevent potential blocking when doing high volume of writes.

Then, there are some considerations when setting up your cluster and tables.

17.9.3.1. Compactions

Depending on read/write volume and latency requirements, optimal compaction settings may be different. See Section 9.7.6.7, "Compaction" for some details.

When provisioning for large data sizes, however, it's good to keep in mind that compactions can affect write throughput. Thus, for write-intensive workloads, you may opt for less frequent compactions and more store files per regions. Minimum number of files for compactions (`hbase.hstore.compaction.min`) can be set to higher value; `hbase.hstore.blockingStoreFiles` should also be increased, as more files might accumulate in such case. You may also consider manually managing compactions: Section 2.6.2.8, "Managed Compactions"

17.9.3.2. Pre-splitting the table

Based on the target number of the regions per RS (see above) and number of RSes, one can pre-split the table at creation time. This would both avoid some costly splitting as the table starts to fill up, and ensure that the table starts out already distributed across many servers.

If the table is expected to grow large enough to justify that, at least one region per RS should be created. It is not recommended to split immediately into the full target number of regions (e.g. 50 * number of RSes), but a low intermediate value can be chosen. For multiple tables, it is recommended to be conservative with presplitting (e.g. pre-split 1 region per RS at most), especially if you don't know how much each table will grow. If you split too much, you may end up with too many regions, with some tables having too many small regions.

For pre-splitting howto, see Section 14.8.2, " Table Creation: Pre-Creating Regions ".

17.10. Table Rename

In versions 0.90.x of hbase and earlier, we had a simple script that would rename the hdfs table directory and then do an edit of the hbase:meta table replacing all mentions of the old table name with the new. The script was called **./bin/rename_table.rb**. The script was deprecated and removed mostly because it was unmaintained and the operation performed by the script was brutal.

As of hbase 0.94.x, you can use the snapshot facility renaming a table. Here is how you would do it using the hbase shell:

```
hbase shell> disable 'tableName'
hbase shell> snapshot 'tableName', 'tableSnapshot'
hbase shell> clone_snapshot 'tableSnapshot', 'newTableName'
hbase shell> delete_snapshot 'tableSnapshot'
hbase shell> drop 'tableName'
```

or in code it would be as follows:

```
void rename(HBaseAdmin admin, String oldTableName, String newTableName) {
  String snapshotName = randomName();
  admin.disableTable(oldTableName);
  admin.snapshot(snapshotName, oldTableName);
  admin.cloneSnapshot(snapshotName, newTableName);
  admin.deleteSnapshot(snapshotName);
  admin.deleteTable(oldTableName);
}
```

---

[28] See this blog post for more details.

Chapter 18. Building and Developing Apache HBase

Table of Contents

This chapter contains information and guidelines for building and releasing HBase code and documentation. Being familiar with these guidelines will help the HBase committers to use your contributions more easily.

18.1. Getting Involved

Apache HBase gets better only when people contribute! If you are looking to contribute to Apache HBase, look for issues in JIRA tagged with the label 'beginner'. These are issues HBase contributors have deemed worthy but not of immediate priority and a good way to ramp on HBase internals. See What label is used for issues that are good on ramps for new contributors? from the dev mailing list for background.

As Apache HBase is an Apache Software Foundation project, see Appendix I, *HBase and the Apache Software Foundation* for more information about how the ASF functions.

18.1.1. Mailing Lists

Sign up for the dev-list and the user-list. See the mailing lists page. Posing questions - and helping to answer other people's questions - is encouraged! There are varying levels of experience on both lists so patience and politeness are encouraged (and please stay on topic.)

18.1.2. Internet Relay Chat (IRC)

For real-time questions and discussions, use the `#hbase` IRC channel on the FreeNode IRC network. FreeNode offers a web-based client, but most people prefer a native client, and several clients are available for each operating system.

18.1.3. Jira

Check for existing issues in Jira. If it's either a new feature request, enhancement, or a bug, file a ticket.

To check for existing issues which you can tackle as a beginner, search for issues in JIRA tagged with the label 'beginner'. See Section 18.1, “Getting Involved” for more information.

JIRA Priorities

- Blocker: Should only be used if the issue WILL cause data loss or cluster instability reliably.
- Critical: The issue described can cause data loss or cluster instability in some cases.
- Major: Important but not tragic issues, like updates to the client API that will add a lot of much-needed functionality or significant bugs that need to be fixed but that don't cause data loss.
- Minor: Useful enhancements and annoying but not damaging bugs.
- Trivial: Useful enhancements but generally cosmetic.

Example 18.1. Code Blocks in Jira Comments

A commonly used macro in Jira is {code}. Everything inside the tags is preformatted, as in this example.

```
{code}
code snippet
{code}
```

18.2. Apache HBase Repositories

There are two different repositories for Apache HBase: Subversion (SVN) and Git. GIT is our repository of record for all but the Apache HBase website. We used to be on SVN. We migrated. See Migrate Apache HBase SVN Repos to Git. Updating hbase.apache.org still requires use of SVN (See Section 18.8, “Updating hbase.apache.org”). See Source Code Management page for contributor and committer links or seach for HBase on the Apache Git page.

18.3. IDEs

18.3.1. Eclipse

18.3.1.1. Code Formatting

Under the `dev-support/` folder, you will find `hbase_eclipse_formatter.xml`. We encourage you to have this formatter in place in eclipse when editing HBase code.

Procedure 18.1. Load the HBase Formatter Into Eclipse

1. Open the Eclipse → Preferences menu item.
2. In Preferences, click the Java → Code Style → Formatter menu item.
3. Click Import and browse to the location of the `hbase_eclipse_formatter.xml` file, which is in the `dev-support/` directory. Click Apply.
4. Still in Preferences, click Java Editor → Save Actions. Be sure the following options are selected:
   - Perform the selected actions on save
   - Format source code
   - Format edited lines

   Click Apply. Close all dialog boxes and return to the main window.

In addition to the automatic formatting, make sure you follow the style guidelines explained in Section 18.12.4, “Code Formatting Conventions”

Also, no `@author` tags - that's a rule. Quality Javadoc comments are appreciated. And include the Apache license.

18.3.1.2. Eclipse Git Plugin

If you cloned the project via git, download and install the Git plugin (EGit). Attach to your local git repo (via the Git Repositories window) and you'll be able to see file revision history, generate patches, etc.

18.3.1.3. HBase Project Setup in Eclipse using `m2eclipse`

The easiest way is to use the **m2eclipse** plugin for Eclipse. Eclipse Indigo or newer includes **m2eclipse**, or you can download it from http://www.eclipse.org/m2e//. It provides Maven integration for Eclipse, and even lets you use the direct Maven commands from within Eclipse to compile and test your project.

To import the project, click File → Import → Maven → Existing Maven Projects and select the HBase root directory. `m2eclipse` locates all the hbase modules for you.

If you install **m2eclipse** and import HBase in your workspace, do the following to fix your eclipse Build Path.

1. Remove `target` folder
2. Add `target/generated-jamon` and `target/generated-sources/java` folders.
3. Remove from your Build Path the exclusions on the `src/main/resources` and `src/test/resources` to avoid error message in the console, such as the following:

   ```
   Failed to execute goal
   org.apache.maven.plugins:maven-antrun-plugin:1.6:run (default) on project hbase:
   'An Ant BuildException has occured: Replace: source file .../target/classes/hbase-default.xml
   ```

```
doesn't exist
```

This will also reduce the eclipse build cycles and make your life easier when developing.

18.3.1.4. HBase Project Setup in Eclipse Using the Command Line

Instead of using `m2eclipse`, you can generate the Eclipse files from the command line.

1. First, run the following command, which builds HBase. You only need to do this once.

```
mvn clean install -DskipTests
```

2. Close Eclipse, and execute the following command from the terminal, in your local HBase project directory, to generate new `.project` and `.classpath` files.

```
mvn eclipse:eclipse
```

3. Reopen Eclipse and import the `.project` file in the HBase directory to a workspace.

18.3.1.5. Maven Classpath Variable

The `$M2_REPO` classpath variable needs to be set up for the project. This needs to be set to your local Maven repository, which is usually `~/.m2/repository`

If this classpath variable is not configured, you will see compile errors in Eclipse like this:

```
Description     Resource        Path    Location        Type
The project cannot be built until build path errors are resolved        hbase           Unknown Java Problem
Unbound classpath variable: 'M2_REPO/asm/asm/3.1/asm-3.1.jar' in project 'hbase'        hbase           Build path      Build Path Prc
Unbound classpath variable: 'M2_REPO/com/google/guava/guava/r09/guava-r09.jar' in project 'hbase'       hbase           Build path
Unbound classpath variable: 'M2_REPO/com/google/protobuf/protobuf-java/2.3.0/protobuf-java-2.3.0.jar' in project 'hbase'        hbase
```

18.3.1.6. Eclipse Known Issues

Eclipse will currently complain about `Bytes.java`. It is not possible to turn these errors off.

```
Description     Resource        Path    Location        Type
Access restriction: The method arrayBaseOffset(Class) from the type Unsafe is not accessible due to restriction on required library /s
Access restriction: The method arrayIndexScale(Class) from the type Unsafe is not accessible due to restriction on required library /s
Access restriction: The method getLong(Object, long) from the type Unsafe is not accessible due to restriction on required library /Sy
```

18.3.1.7. Eclipse - More Information

For additional information on setting up Eclipse for HBase development on Windows, see Michael Morello's blog on the topic.

18.3.2. Other IDEs

It would be userful to mirror the Section 18.3.1, "Eclipse" set-up instructions for other IDEs. If you would like to assist, please have a look at HBASE-11704.

18.4. Building Apache HBase

18.4.1. Basic Compile

HBase is compiled using Maven. You must use Maven 3.x. To check your Maven version, run the command **mvn -version**.

You can read about the various maven commands in Section 18.10, "Maven Build Commands", but the simplest command to compile HBase from its java source code is:

```
mvn package -DskipTests
```

Or, to clean up before compiling:

```
mvn clean package -DskipTests
```

With Eclipse set up as explained above in Section 18.3.1, "Eclipse", you can also use the Build command in Eclipse. To create the full installable HBase package takes a little bit more work, so read on.

JDK Version Requirements

Starting with HBase 1.0 you must use Java 7 or later to build from source code. See Table 2.1, "Java" for more complete information about supported JDK versions.

18.4.2. Build Protobuf

You may need to change the protobuf definitions that reside in the `hbase-protocol` module or other modules.

The protobuf files are located `hbase-protocol/src/main/protobuf`. For the change to be effective, you will need to regenerate the classes. You can use maven profile `compile-protobuf` to do this.

```
mvn compile -Pcompile-protobuf
```

You may also want to define `protoc.path` for the protoc binary, using the following command:

```
mvn compile -Pcompile-protobuf -Dprotoc.path=/opt/local/bin/protoc
```

Read the `hbase-protocol/README.txt` for more details.

18.4.3. Build a Tarball

You can build a tarball without going through the release process described in Section 18.5, "Releasing Apache HBase", by running the following command:

```
mvn -DskipTests clean install && mvn -DskipTests package assembly:single
```

The distribution tarball is built in `hbase-assembly/target/hbase-<version>-bin.tar.gz`.

18.4.4. Build Gotchas

If you see `Unable to find resource 'VM_global_library.vm'`, ignore it. Its not an error. It is officially ugly though.

18.4.5. Building in snappy compression support

Pass `-Psnappy` to trigger the `snappy` maven profile for building Google Snappy native libraries into HBase. See also ???

18.5. Releasing Apache HBase

Building against HBase 1.x. HBase 1.x requires Java 7 to build. See Table 2.1, "Java" for Java requirements per HBase release.

18.5.1. Building against HBase 0.96-0.98

HBase 0.96.x will run on Hadoop 1.x or Hadoop 2.x. HBase 0.98 still runs on both, but HBase 0.98 deprecates use of Hadoop 1. HBase 1.x will *not* run on Hadoop 1. In the following procedures, we make a distinction between HBase 1.x builds and the awkward process involved building HBase 0.96/0.98 for either Hadoop 1 or Hadoop 2 targets.

You must choose which Hadoop to build against. It is not possible to build a single HBase binary that runs against both Hadoop 1 and Hadoop 2. Hadoop is included in the build, because it is needed to run HBase in standalone mode. Therefore, the set of modules included in the tarball changes, depending on the build target. To determine which HBase you have, look at the HBase version. The Hadoop version is embedded within it.

Maven, our build system, natively does not allow a single product to be built against different dependencies. Also, Maven cannot change the set of included modules and write out the correct `pom.xml` files with appropriate dependencies, even using two build targets, one for Hadoop 1 and another for Hadoop 2. A prerequisite step is required, which takes as input the current `pom.xml`s and generates Hadoop 1 or Hadoop 2 versions using a script in the `dev-tools/` directory, called `generate-hadoopX-poms.sh` where *X* is either `1` or `2`. You then reference these generated poms when you build. For now, just be aware of the difference between HBase 1.x builds and those of HBase 0.96-0.98. This difference is important to the build instructions.

Example 18.2. Example `~/.m2/settings.xml` File

Publishing to maven requires you sign the artifacts you want to upload. For the build to sign them for you, you a properly configured `settings.xml` in your local repository under `.m2`, such as the following.

```
<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0
                      http://maven.apache.org/xsd/settings-1.0.0.xsd">
  <servers>
    <!- To publish a snapshot of some part of Maven -->
    <server>
      <id>apache.snapshots.https</id>
      <username>YOUR_APACHE_ID
      </username>
      <password>YOUR_APACHE_PASSWORD
      </password>
    </server>
    <!-- To publish a website using Maven -->
    <!-- To stage a release of some part of Maven -->
    <server>
      <id>apache.releases.https</id>
      <username>YOUR_APACHE_ID
      </username>
      <password>YOUR_APACHE_PASSWORD
      </password>
    </server>
  </servers>
  <profiles>
    <profile>
      <id>apache-release</id>
      <properties>
    <gpg.keyname>YOUR_KEYNAME</gpg.keyname>
    <!--Keyname is something like this ... 00A5F21E... do gpg --list-keys to find it-->
    <gpg.passphrase>YOUR_KEY_PASSWORD
    </gpg.passphrase>
      </properties>
    </profile>
  </profiles>
</settings>
```

18.5.2. Making a Release Candidate

> **Note**
>
> These instructions are for building HBase 1.0.x. For building earlier versions, the process is different. See this section under the respective release documentation folders.

Point Releases. If you are making a point release (for example to quickly address a critical incompatability or security problem) off of a release branch instead of a development branch, the tagging instructions are slightly different. I'll prefix those special steps with *Point Release Only*.

Before You Begin. Before you make a release candidate, do a practice run by deploying a snapshot. Before you start, check to be sure recent builds have been passing for the branch from where you are going to take your release. You should also have tried recent branch tips out on a cluster under load, perhaps by running the `hbase-`

`it` integration test suite for a few hours to 'burn in' the near-candidate bits.

Point Release Only

At this point you should tag the previous release branch (ex: 0.96.1) with the new point release tag (e.g. 0.96.1.1 tag). Any commits with changes for the point release should be appled to the new tag.

The Hadoop How To Release wiki page is used as a model for most of the instructions below, and may have more detail on particular sections, so it is worth review.

Specifying the Heap Space for Maven on OSX

On OSX, you may need to specify the heap space for Maven commands, by setting the `MAVEN_OPTS` variable to `-Xmx3g`. You can prefix the variable to the Maven command, as in the following example:

```
MAVEN_OPTS="-Xmx2g" mvn package
```

You could also set this in an environment variable or alias in your shell.

Procedure 18.2. Release Procedure

The script `dev-support/make_rc.sh` automates many of these steps. It does not do the modification of the `CHANGES.txt` for the release, the close of the staging repository in Apache Maven (human intervention is needed here), the checking of the produced artifacts to ensure they are 'good' -- e.g. extracting the produced tarballs, verifying that they look right, then starting HBase and checking that everything is running correctly, then the signing and pushing of the tarballs to people.apache.org. The script handles everything else, and comes in handy.

1. **Update the `CHANGES.txt` file and the POM files.**

   Update `CHANGES.txt` with the changes since the last release. Make sure the URL to the JIRA points to the proper location which lists fixes for this release. Adjust the version in all the POM files appropriately. If you are making a release candidate, you must remove the `-SNAPSHOT` label from all versions. If you are running this receipe to publish a snapshot, you must keep the `-SNAPSHOT` suffix on the hbase version. The Versions Maven Plugin can be of use here. To set a version in all the many poms of the hbase multi-module project, use a command like the following:

   ```
   $ mvn clean org.codehaus.mojo:versions-maven-plugin:1.3.1:set -DnewVersion=0.96.0
   ```

   Checkin the `CHANGES.txt` and any version changes.

2. **Update the documentation.**

   Update the documentation under `src/main/docbkx`. This usually involves copying the latest from trunk and making version-particular adjustments to suit this release candidate version.

3. **Build the source tarball.**

   Now, build the source tarball. This tarball is Hadoop-version-independent. It is just the pure source code and documentation without a particular hadoop taint, etc. Add the `-Prelease` profile when building. It checks files for licenses and will fail the build if unlicensed files are present.

   ```
   $ mvn clean install -DskipTests assembly:single -Dassembly.file=hbase-assembly/src/main/assembly/src.xml -Prelease
   ```

   Extract the tarball and make sure it looks good. A good test for the src tarball being 'complete' is to see if you can build new tarballs from this source bundle. If the source tarball is good, save it off to a *version directory*, a directory somewhere where you are collecting all of the tarballs you will publish as part of the release candidate. For example if you were building a hbase-0.96.0 release candidate, you might call the directory `hbase-0.96.0RC0`. Later you will publish this directory as our release candidate up on people.apache.org/*~you*/.

4. **Build the binary tarball.**

   Next, build the binary tarball. Add the `-Prelease` profile when building. It checks files for licenses and will fail the build if unlicensed files are present. Do it in two steps.

   a. First install into the local repository

      ```
      $ mvn clean install -DskipTests -Prelease
      ```

   b. Next, generate documentation and assemble the tarball.

      ```
      $ mvn install -DskipTests site assembly:single -Prelease
      ```

   Otherwise, the build complains that hbase modules are not in the maven repository when you try to do it at once, especially on fresh repository. It seems that you need the install goal in both steps.

   Extract the generated tarball and check it out. Look at the documentation, see if it runs, etc. If good, copy the tarball to the above mentioned *version directory*.

5. **Create a new tag.**

   ### Point Release Only

   The following step that creates a new tag can be skipped since you've already created the point release tag

   Tag the release at this point since it looks good. If you find an issue later, you can delete the tag and start over. Release needs to be tagged for the next step.

6. **Deploy to the Maven Repository.**

   Next, deploy HBase to the Apache Maven repository, using the `apache-release` profile instead of the `release` profile when running the **mvn deploy** command. This profile invokes the Apache pom referenced by our pom files, and also signs your artifacts published to Maven, as long as the `settings.xml` is configured correctly, as described in Example 18.2, “Example `~/.m2/settings.xml` File”.

   ```
   $ mvn deploy -DskipTests -Papache-release
   ```

   This command copies all artifacts up to a temporary staging Apache mvn repository in an 'open' state. More work needs to be done on these maven artifacts to

make them generally available.

7. **Make the Release Candidate available.**

   The artifacts are in the maven repository in the staging area in the 'open' state. While in this 'open' state you can check out what you've published to make sure all is good. To do this, login at repository.apache.org using your Apache ID. Find your artifacts in the staging repository. Browse the content. Make sure all artifacts made it up and that the poms look generally good. If it checks out, 'close' the repo. This will make the artifacts publically available. You will receive an email with the URL to give out for the temporary staging repository for others to use trying out this new release candidate. Include it in the email that announces the release candidate. Folks will need to add this repo URL to their local poms or to their local `settings.xml` file to pull the published release candidate artifacts. If the published artifacts are incomplete or have problems, just delete the 'open' staged artifacts.

   #### hbase-downstreamer

   See the hbase-downstreamer test for a simple example of a project that is downstream of HBase an depends on it. Check it out and run its simple test to make sure maven artifacts are properly deployed to the maven repository. Be sure to edit the pom to point to the proper staging repository. Make sure you are pulling from the repository when tests run and that you are not getting from your local repository, by either passing the `-U` flag or deleting your local repo content and check maven is pulling from remote out of the staging repository.

   See Publishing Maven Artifacts for some pointers on this maven staging process.

   #### Note

   We no longer publish using the maven release plugin. Instead we do **mvn deploy**. It seems to give us a backdoor to maven release publishing. If there is no *-SNAPSHOT* on the version string, then we are 'deployed' to the apache maven repository staging directory from which we can publish URLs for candidates and later, if they pass, publish as release (if a *-SNAPSHOT* on the version string, deploy will put the artifacts up into apache snapshot repos).

   If the HBase version ends in `-SNAPSHOT`, the artifacts go elsewhere. They are put into the Apache snapshots repository directly and are immediately available. Making a SNAPSHOT release, this is what you want to happen.

8. **If you used the `make_rc.sh` script instead of doing the above manually, do your sanity checks now.**

   At this stage, you have two tarballs in your 'version directory' and a set of artifacts in a staging area of the maven repository, in the 'closed' state. These are publicly accessible in a temporary staging repository whose URL you should have gotten in an email. The above mentioned script, `make_rc.sh` does all of the above for you minus the check of the artifacts built, the closing of the staging repository up in maven, and the tagging of the release. If you run the script, do your checks at this stage verifying the src and bin tarballs and checking what is up in staging using hbase-downstreamer project. Tag before you start the build. You can always delete it if the build goes haywire.

9. **Sign and upload your version directory to people.apache.org.**

   If all checks out, next put the *version directory* up on people.apache.org. You will need to sign and fingerprint them before you push them up. In the *version directory* run the following commands:

   ```
   $ for i in *.tar.gz; do echo $i; gpg --print-mds $i > $i.mds ; done
   $ for i in *.tar.gz; do echo $i; gpg --armor --output $i.asc --detach-sig $i  ; done
   $ cd ..
   # Presuming our 'version directory' is named 0.96.0RC0, now copy it up to people.apache.org.
   $ rsync -av 0.96.0RC0 people.apache.org:public_html
   ```

   Make sure the people.apache.org directory is showing and that the mvn repo URLs are good. Announce the release candidate on the mailing list and call a vote.

18.5.3. Publishing a SNAPSHOT to maven

Make sure your `settings.xml` is set up properly, as in Example 18.2, "Example `~/.m2/settings.xml` File". Make sure the hbase version includes `-SNAPSHOT` as a suffix. Following is an example of publishing SNAPSHOTS of a release that had an hbase version of 0.96.0 in its poms.

```
 $ mvn clean install -DskipTests  javadoc:aggregate site assembly:single -Prelease
 $ mvn -DskipTests  deploy -Papache-release
```

The `make_rc.sh` script mentioned above (see Section 18.5.2, "Making a Release Candidate") can help you publish `SNAPSHOTS`. Make sure your `hbase.version` has a `-SNAPSHOT` suffix before running the script. It will put a snapshot up into the apache snapshot repository for you.

18.6. Voting on Release Candidates

Everyone is encouraged to try and vote on HBase release candidates. Only the votes of PMC members are binding. PMC members, please read this WIP doc on policy voting for a release candidate, Release Policy. "Before casting +1 binding votes, individuals are required to download the signed source code package onto their own hardware, compile it as provided, and test the resulting executable on their own platform, along with also validating cryptographic signatures and verifying that the package meets the requirements of the ASF policy on releases." Regards the latter, run **mvn apache-rat:check** to verify all files are suitably licensed. See HBase, mail # dev - On recent discussion clarifying ASF release policy. for how we arrived at this process.

18.7. Generating the HBase Reference Guide

The manual is marked up using docbook. We then use the docbkx maven plugin to transform the markup to html. This plugin is run when you specify the **site** goal as in when you run **mvn site** or you can call the plugin explicitly to just generate the manual by doing **mvn docbkx:generate-html**. When you run **mvn site**, the documentation is generated twice, once to generate the multipage manual and then again for the single page manual, which is easier to search. See Appendix A, *Contributing to Documentation* for more information on building the documentation.

18.8. Updating hbase.apache.org

18.8.1. Contributing to hbase.apache.org

See Appendix A, *Contributing to Documentation* for more information on contributing to the documentation or website.

18.8.2. Publishing hbase.apache.org

As of INFRA-5680 Migrate apache hbase website, to publish the website, build it, and then deploy it over a checkout of `https://svn.apache.org/repos/asf/hbase/hbase.apache.org/trunk`. Finally, check it in. For example, if trunk is checked out out at `/Users/stack/checkouts/trunk` and the hbase website, hbase.apache.org, is checked out at `/Users/stack/checkouts/hbase.apache.org/trunk`, to update the site, do the following:

```
# Build the site and deploy it to the checked out directory
# Getting the javadoc into site is a little tricky.  You have to build it before you invoke 'site'.
$ mvn clean install -DskipTests javadoc:aggregate site  \
  site:stage -DstagingDirectory=/Users/stack/checkouts/hbase.apache.org/trunk
```

Now check the deployed site by viewing in a brower, browse to file:////Users/stack/checkouts/hbase.apache.org/trunk/index.html and check all is good. If all checks out, commit it and your new build will show up immediately at http://hbase.apache.org.

```
$ cd /Users/stack/checkouts/hbase.apache.org/trunk
$ svn status
# Do an svn add of any new content...
$ svn add ....
$ svn commit -m 'Committing latest version of website...'
```

18.9. Tests

Developers, at a minimum, should familiarize themselves with the unit test detail; unit tests in HBase have a character not usually seen in other projects.

This information is about unit tests for HBase itself. For developing unit tests for your HBase applications, see Chapter 19, *Unit Testing HBase Applications*.

18.9.1. Apache HBase Modules

As of 0.96, Apache HBase is split into multiple modules. This creates "interesting" rules for how and where tests are written. If you are writing code for `hbase-server`, see Section 18.9.2, “Unit Tests” for how to write your tests. These tests can spin up a minicluster and will need to be categorized. For any other module, for example `hbase-common`, the tests must be strict unit tests and only test the class under test - no use of the **HBaseTestingUtility** or minicluster is allowed (or even possible given the dependency tree).

18.9.1.1. Running Tests in other Modules

If the module you are developing in has no other dependencies on other HBase modules, then you can cd into that module and run the `mvn test` command, which will only run the tests *in that module*. If there are other dependencies on other modules, then you must run the command from the *root hbase directory*. This will run the tests in all modules unless you specify to skip the tests in a given module. For instance, to skip the tests in the `hbase-server` module, you would run the following command, to run all the tests in modules other than `hbase-server`:

```
mvn clean test -PskipServerTests
```

You can specify to skip tests in multiple modules as well as just for a single module. For example, to skip the tests in `hbase-server` and `hbase-common`, you would run the following command:

```
        mvn clean test -PskipServerTests -PskipCommonTests
```

Also, keep in mind that if you are running tests in the `hbase-server` module, you need to apply the maven profiles discussed in Section 18.9.3, “Running tests” to get the tests to run properly.

18.9.2. Unit Tests

Apache HBase unit tests are subdivided into four categories: `small`, `medium`, `large`, and `integration` with corresponding JUnit categories: `SmallTests`, `MediumTests`, `LargeTests`, `IntegrationTests`. JUnit categories are denoted using java annotations and look like this in your unit test code.

```
...
@Category(SmallTests.class)
public class TestHRegionInfo {
  @Test
  public void testCreateHRegionInfoName() throws Exception {
    // ...
  }
}
```

The above example shows how to mark a unit test as belonging to the `small` category. All unit tests in HBase have a categorization.

The first three categories, `small`, `medium`, and `large`, are for tests run when you type `$ mvn test`. In other words, these three categorizations are for HBase unit tests. The `integration` category is not for unit tests, but for integration tests. These are run when you invoke `$ mvn verify`. Integration tests are described in Section 18.9.5, “Integration Tests”.

HBase uses a patched maven surefire plugin and maven profiles to implement its unit test characterizations.

Keep reading to figure which annotation of the set small, medium, and large to put on your new HBase unit test.

Categorizing Tests

Small Tests

> *Small* tests are executed in a shared JVM. We put in this category all the tests that can be executed quickly in a shared JVM. The maximum execution time for a small test is 15 seconds, and small tests should not use a (mini)cluster.

Medium Tests

> *Medium* tests represent tests that must be executed before proposing a patch. They are designed to run in less than 30 minutes altogether, and are quite stable in their results. They are designed to last less than 50 seconds individually. They can use a cluster, and each of them is executed in a separate JVM.

Large Tests

> *Large* tests are everything else. They are typically large-scale tests, regression tests for specific bugs, timeout tests, performance tests. They are executed before a commit on the pre-integration machines. They can be run on the developer machine as well.

Integration Tests

Integration tests are system level tests. See Section 18.9.5, “Integration Tests” for more info.

18.9.3. Running tests

18.9.3.1. Default: small and medium category tests

Running `mvn test` will execute all small tests in a single JVM (no fork) and then medium tests in a separate JVM for each test instance. Medium tests are NOT executed if there is an error in a small test. Large tests are NOT executed. There is one report for small tests, and one report for medium tests if they are executed.

18.9.3.2. Running all tests

Running `mvn test -P runAllTests` will execute small tests in a single JVM then medium and large tests in a separate JVM for each test. Medium and large tests are NOT executed if there is an error in a small test. Large tests are NOT executed if there is an error in a small or medium test. There is one report for small tests, and one report for medium and large tests if they are executed.

18.9.3.3. Running a single test or all tests in a package

To run an individual test, e.g. `MyTest`, rum `mvn test -Dtest=MyTest` You can also pass multiple, individual tests as a comma-delimited list: `mvn test -Dtest=MyTest1,MyTest2,MyTest3` You can also pass a package, which will run all tests under the package: `mvn test '-Dtest=org.apache.hadoop.hbase.client.*'`

When `-Dtest` is specified, the `localTests` profile will be used. It will use the official release of maven surefire, rather than our custom surefire plugin, and the old connector (The HBase build uses a patched version of the maven surefire plugin). Each junit test is executed in a separate JVM (A fork per test class). There is no parallelization when tests are running in this mode. You will see a new message at the end of the -report: `"[INFO] Tests are skipped"`. It's harmless. However, you need to make sure the sum of `Tests run:` in the `Results :` section of test reports matching the number of tests you specified because no error will be reported when a non-existent test case is specified.

18.9.3.4. Other test invocation permutations

Running **mvn test -P runSmallTests** will execute "small" tests only, using a single JVM.

Running **mvn test -P runMediumTests** will execute "medium" tests only, launching a new JVM for each test-class.

Running **mvn test -P runLargeTests** will execute "large" tests only, launching a new JVM for each test-class.

For convenience, you can run **mvn test -P runDevTests** to execute both small and medium tests, using a single JVM.

18.9.3.5. Running tests faster

By default, `$ mvn test -P runAllTests` runs 5 tests in parallel. It can be increased on a developer's machine. Allowing that you can have 2 tests in parallel per core, and you need about 2Gb of memory per test (at the extreme), if you have an 8 core, 24Gb box, you can have 16 tests in parallel. but the memory available limits it to 12 (24/2), To run all tests with 12 tests in parallel, do this: **mvn test -P runAllTests -Dsurefire.secondPartThreadCount=12**. To increase the speed, you can as well use a ramdisk. You will need 2Gb of memory to run all tests. You will also need to delete the files between two test run. The typical way to configure a ramdisk on Linux is:

```
$ sudo mkdir /ram2G
sudo mount -t tmpfs -o size=2048M tmpfs /ram2G
```

You can then use it to run all HBase tests with the command:

```
mvn test
                        -P runAllTests -Dsurefire.secondPartThreadCount=12
                        -Dtest.build.data.basedirectory=/ram2G
```

18.9.3.6. hbasetests.sh

It's also possible to use the script **hbasetests.sh**. This script runs the medium and large tests in parallel with two maven instances, and provides a single report. This script does not use the hbase version of surefire so no parallelization is being done other than the two maven instances the script sets up. It must be executed from the directory which contains the `pom.xml`.

For example running **./dev-support/hbasetests.sh** will execute small and medium tests. Running **./dev-support/hbasetests.sh runAllTests** will execute all tests. Running **./dev-support/hbasetests.sh replayFailed** will rerun the failed tests a second time, in a separate jvm and without parallelisation.

18.9.3.7. Test Resource Checker

A custom Maven SureFire plugin listener checks a number of resources before and after each HBase unit test runs and logs its findings at the end of the test output files which can be found in `target/surefire-reports` per Maven module (Tests write test reports named for the test class into this directory. Check the `*-out.txt` files). The resources counted are the number of threads, the number of file descriptors, etc. If the number has increased, it adds a *LEAK?* comment in the logs. As you can have an HBase instance running in the background, some threads can be deleted/created without any specific action in the test. However, if the test does not work as expected, or if the test should not impact these resources, it's worth checking these log lines `...hbase.ResourceChecker(157): before...` and `...hbase.ResourceChecker(157): after...`. For example:

```
2012-09-26 09:22:15,315 INFO [pool-1-thread-1]
hbase.ResourceChecker(157): after:
regionserver.TestColumnSeeking#testReseeking Thread=65 (was 65),
OpenFileDescriptor=107 (was 107), MaxFileDescriptor=10240 (was 10240),
ConnectionCount=1 (was 1)
```

18.9.4. Writing Tests

18.9.4.1. General rules

- As much as possible, tests should be written as category small tests.

- All tests must be written to support parallel execution on the same machine, hence they should not use shared resources as fixed ports or fixed file names.
- Tests should not overlog. More than 100 lines/second makes the logs complex to read and use i/o that are hence not available for the other tests.
- Tests can be written with `HBaseTestingUtility`. This class offers helper functions to create a temp directory and do the cleanup, or to start a cluster.

18.9.4.2. Categories and execution time

- All tests must be categorized, if not they could be skipped.
- All tests should be written to be as fast as possible.
- Small category tests should last less than 15 seconds, and must not have any side effect.
- Medium category tests should last less than 50 seconds.
- Large category tests should last less than 3 minutes. This should ensure a good parallelization for people using it, and ease the analysis when the test fails.

18.9.4.3. Sleeps in tests

Whenever possible, tests should not use `Thread.sleep`, but rather waiting for the real event they need. This is faster and clearer for the reader. Tests should not do a `Thread.sleep` without testing an ending condition. This allows understanding what the test is waiting for. Moreover, the test will work whatever the machine performance is. Sleep should be minimal to be as fast as possible. Waiting for a variable should be done in a 40ms sleep loop. Waiting for a socket operation should be done in a 200 ms sleep loop.

18.9.4.4. Tests using a cluster

Tests using a HRegion do not have to start a cluster: A region can use the local file system. Start/stopping a cluster cost around 10 seconds. They should not be started per test method but per test class. Started cluster must be shutdown using `HBaseTestingUtility#shutdownMiniCluster`, which cleans the directories. As most as possible, tests should use the default settings for the cluster. When they don't, they should document it. This will allow to share the cluster later.

18.9.5. Integration Tests

HBase integration/system tests are tests that are beyond HBase unit tests. They are generally long-lasting, sizeable (the test can be asked to 1M rows or 1B rows), targetable (they can take configuration that will point them at the ready-made cluster they are to run against; integration tests do not include cluster start/stop code), and verifying success, integration tests rely on public APIs only; they do not attempt to examine server internals asserting success/fail. Integration tests are what you would run when you need to more elaborate proofing of a release candidate beyond what unit tests can do. They are not generally run on the Apache Continuous Integration build server, however, some sites opt to run integration tests as a part of their continuous testing on an actual cluster.

Integration tests currently live under the `src/test` directory in the hbase-it submodule and will match the regex: `**/IntegrationTest*.java`. All integration tests are also annotated with `@Category(IntegrationTests.class)`.

Integration tests can be run in two modes: using a mini cluster, or against an actual distributed cluster. Maven failsafe is used to run the tests using the mini cluster. IntegrationTestsDriver class is used for executing the tests against a distributed cluster. Integration tests SHOULD NOT assume that they are running against a mini cluster, and SHOULD NOT use private API's to access cluster state. To interact with the distributed or mini cluster uniformly, `IntegrationTestingUtility`, and `HBaseCluster` classes, and public client API's can be used.

On a distributed cluster, integration tests that use ChaosMonkey or otherwise manipulate services thru cluster manager (e.g. restart regionservers) use SSH to do it. To run these, test process should be able to run commands on remote end, so ssh should be configured accordingly (for example, if HBase runs under hbase user in your cluster, you can set up passwordless ssh for that user and run the test also under it). To facilitate that, `hbase.it.clustermanager.ssh.user`, `hbase.it.clustermanager.ssh.opts` and `hbase.it.clustermanager.ssh.cmd` configuration settings can be used. "User" is the remote user that cluster manager should use to perform ssh commands. "Opts" contains additional options that are passed to SSH (for example, "-i /tmp/my-key"). Finally, if you have some custom environment setup, "cmd" is the override format for the entire tunnel (ssh) command. The default string is `{/usr/bin/ssh %1$s %2$s%3$s%4$s "%5$s"}` and is a good starting point. This is a standard Java format string with 5 arguments that is used to execute the remote command. The argument 1 (%1$s) is SSH options set the via opts setting or via environment variable, 2 is SSH user name, 3 is "@" if username is set or "" otherwise, 4 is the target host name, and 5 is the logical command to execute (that may include single quotes, so don't use them). For example, if you run the tests under non-hbase user and want to ssh as that user and change to hbase on remote machine, you can use `{/usr/bin/ssh %1$s %2$s%3$s%4$s "su hbase - -c \"%5$s\""}`. That way, to kill RS (for example) integration tests may run `{/usr/bin/ssh some-hostname "su hbase - -c \"ps aux | ... | kill ...\""}`. The command is logged in the test logs, so you can verify it is correct for your environment.

18.9.5.1. Running integration tests against mini cluster

HBase 0.92 added a `verify` maven target. Invoking it, for example by doing `mvn verify`, will run all the phases up to and including the verify phase via the maven failsafe plugin, running all the above mentioned HBase unit tests as well as tests that are in the HBase integration test group. After you have completed **mvn install -DskipTests** You can run just the integration tests by invoking:

```
cd hbase-it
mvn verify
```

If you just want to run the integration tests in top-level, you need to run two commands. First: **mvn failsafe:integration-test** This actually runs ALL the integration tests.

> **Note**
>
> This command will always output `BUILD SUCCESS` even if there are test failures.

At this point, you could grep the output by hand looking for failed tests. However, maven will do this for us; just use: **mvn failsafe:verify** The above command basically looks at all the test results (so don't remove the 'target' directory) for test failures and reports the results.

18.9.5.1.1. Running a subset of Integration tests

This is very similar to how you specify running a subset of unit tests (see above), but use the property `it.test` instead of `test`. To just run `IntegrationTestClassXYZ.java`, use: **mvn failsafe:integration-test -Dit.test=IntegrationTestClassXYZ** The next thing you might want to do is run groups of integration tests, say all integration tests that are named IntegrationTestClassX*.java: **mvn failsafe:integration-test -Dit.test=*ClassX*** This runs everything that is an integration test that matches *ClassX*. This means anything matching: "**/IntegrationTest*ClassX*". You can also run multiple groups of integration tests using comma-delimited lists (similar to unit tests). Using a list of matches still supports full regex matching for each of the groups.This would look something like: **mvn failsafe:integration-test -Dit.test=*ClassX*, *ClassY**

18.9.5.2. Running integration tests against distributed cluster

If you have an already-setup HBase cluster, you can launch the integration tests by invoking the class `IntegrationTestsDriver`. You may have to run test-compile first. The configuration will be picked by the bin/hbase script.

```
mvn test-compile
```

Then launch the tests with:

```
bin/hbase [--config config_dir] org.apache.hadoop.hbase.IntegrationTestsDriver
```

Pass `-h` to get usage on this sweet tool. Running the IntegrationTestsDriver without any argument will launch tests found under `hbase-it/src/test`, having `@Category(IntegrationTests.class)` annotation, and a name starting with `IntegrationTests`. See the usage, by passing -h, to see how to filter test classes. You can pass a regex which is checked against the full class name; so, part of class name can be used. IntegrationTestsDriver uses Junit to run the tests. Currently there is no support for running integration tests against a distributed cluster using maven (see HBASE-6201).

The tests interact with the distributed cluster by using the methods in the `DistributedHBaseCluster` (implementing `HBaseCluster`) class, which in turn uses a pluggable `ClusterManager`. Concrete implementations provide actual functionality for carrying out deployment-specific and environment-dependent tasks (SSH, etc). The default `ClusterManager` is `HBaseClusterManager`, which uses SSH to remotely execute start/stop/kill/signal commands, and assumes some posix commands (ps, etc). Also assumes the user running the test has enough "power" to start/stop servers on the remote machines. By default, it picks up `HBASE_SSH_OPTS, HBASE_HOME, HBASE_CONF_DIR` from the env, and uses `bin/hbase-daemon.sh` to carry out the actions. Currently tarball deployments, deployments which uses hbase-daemons.sh, and Apache Ambari deployments are supported. /etc/init.d/ scripts are not supported for now, but it can be easily added. For other deployment options, a ClusterManager can be implemented and plugged in.

18.9.5.3. Destructive integration / system tests

In 0.96, a tool named `ChaosMonkey` has been introduced. It is modeled after the same-named tool by Netflix. Some of the tests use ChaosMonkey to simulate faults in the running cluster in the way of killing random servers, disconnecting servers, etc. ChaosMonkey can also be used as a stand-alone tool to run a (misbehaving) policy while you are running other tests.

ChaosMonkey defines Action's and Policy's. Actions are sequences of events. We have at least the following actions:

- Restart active master (sleep 5 sec)
- Restart random regionserver (sleep 5 sec)
- Restart random regionserver (sleep 60 sec)
- Restart META regionserver (sleep 5 sec)
- Restart ROOT regionserver (sleep 5 sec)
- Batch restart of 50% of regionservers (sleep 5 sec)
- Rolling restart of 100% of regionservers (sleep 5 sec)

Policies on the other hand are responsible for executing the actions based on a strategy. The default policy is to execute a random action every minute based on predefined action weights. ChaosMonkey executes predefined named policies until it is stopped. More than one policy can be active at any time.

To run ChaosMonkey as a standalone tool deploy your HBase cluster as usual. ChaosMonkey uses the configuration from the bin/hbase script, thus no extra configuration needs to be done. You can invoke the ChaosMonkey by running:

```
bin/hbase org.apache.hadoop.hbase.util.ChaosMonkey
```

This will output smt like:

```
12/11/19 23:21:57 INFO util.ChaosMonkey: Using ChaosMonkey Policy: class org.apache.hadoop.hbase.util.ChaosMonkey$PeriodicRandomActior
12/11/19 23:21:57 INFO util.ChaosMonkey: Sleeping for 26953 to add jitter
12/11/19 23:22:24 INFO util.ChaosMonkey: Performing action: Restart active master
12/11/19 23:22:24 INFO util.ChaosMonkey: Killing master:master.example.com,60000,1353367210440
12/11/19 23:22:24 INFO hbase.HBaseCluster: Aborting Master: master.example.com,60000,1353367210440
12/11/19 23:22:24 INFO hbase.ClusterManager: Executing remote command: ps aux | grep master | grep -v grep | tr -s ' ' | cut -d ' ' -f
12/11/19 23:22:25 INFO hbase.ClusterManager: Executed remote command, exit code:0 , output:
12/11/19 23:22:25 INFO hbase.HBaseCluster: Waiting service:master to stop: master.example.com,60000,1353367210440
12/11/19 23:22:25 INFO hbase.ClusterManager: Executing remote command: ps aux | grep master | grep -v grep | tr -s ' ' | cut -d ' ' -f
12/11/19 23:22:25 INFO hbase.ClusterManager: Executed remote command, exit code:0 , output:
12/11/19 23:22:25 INFO util.ChaosMonkey: Killed master server:master.example.com,60000,1353367210440
12/11/19 23:22:25 INFO util.ChaosMonkey: Sleeping for:5000
12/11/19 23:22:30 INFO util.ChaosMonkey: Starting master:master.example.com
12/11/19 23:22:30 INFO hbase.HBaseCluster: Starting Master on: master.example.com
12/11/19 23:22:30 INFO hbase.ClusterManager: Executing remote command: /homes/enis/code/hbase-0.94/bin/../bin/hbase-daemon.sh --config
12/11/19 23:22:31 INFO hbase.ClusterManager: Executed remote command, exit code:0 , output:starting master, logging to /homes/enis/coc
....
12/11/19 23:22:33 INFO util.ChaosMonkey: Started master: master.example.com,60000,1353367210440
12/11/19 23:22:33 INFO util.ChaosMonkey: Sleeping for:51321
12/11/19 23:23:24 INFO util.ChaosMonkey: Performing action: Restart random region server
12/11/19 23:23:24 INFO util.ChaosMonkey: Killing region server:rs3.example.com,60020,1353367027826
12/11/19 23:23:24 INFO hbase.HBaseCluster: Aborting RS: rs3.example.com,60020,1353367027826
12/11/19 23:23:24 INFO hbase.ClusterManager: Executing remote command: ps aux | grep regionserver | grep -v grep | tr -s ' ' | cut -d
12/11/19 23:23:25 INFO hbase.ClusterManager: Executed remote command, exit code:0 , output:
12/11/19 23:23:25 INFO hbase.HBaseCluster: Waiting service:regionserver to stop: rs3.example.com,60020,1353367027826
12/11/19 23:23:25 INFO hbase.ClusterManager: Executing remote command: ps aux | grep regionserver | grep -v grep | tr -s ' ' | cut -d
12/11/19 23:23:25 INFO hbase.ClusterManager: Executed remote command, exit code:0 , output:
12/11/19 23:23:25 INFO util.ChaosMonkey: Killed region server:rs3.example.com,60020,1353367027826. Reported num of rs:6
12/11/19 23:23:25 INFO util.ChaosMonkey: Sleeping for:60000
12/11/19 23:24:25 INFO util.ChaosMonkey: Starting region server:rs3.example.com
12/11/19 23:24:25 INFO hbase.HBaseCluster: Starting RS on: rs3.example.com
12/11/19 23:24:25 INFO hbase.ClusterManager: Executing remote command: /homes/enis/code/hbase-0.94/bin/../bin/hbase-daemon.sh --config
12/11/19 23:24:26 INFO hbase.ClusterManager: Executed remote command, exit code:0 , output:starting regionserver, logging to /homes/er

12/11/19 23:24:27 INFO util.ChaosMonkey: Started region server:rs3.example.com,60020,1353367027826. Reported num of rs:6
```

As you can see from the log, ChaosMonkey started the default PeriodicRandomActionPolicy, which is configured with all the available actions, and ran RestartActiveMaster and RestartRandomRs actions. ChaosMonkey tool, if run from command line, will keep on running until the process is killed.

18.9.5.4. Passing individual Chaos Monkey per-test Settings/Properties

Since HBase version 1.0.0 (HBASE-11348), the chaos monkeys is used to run integration tests can be configured per test run. Users can create a java properties file and and pass this to the chaos monkey with timing configurations. The properties file needs to be in the HBase classpath. The various properties that can be configured and their default values can be found listed in the `org.apache.hadoop.hbase.chaos.factories.MonkeyConstants` class. If any chaos monkey configuration is missing from the property file, then the default values are assumed. For example:

```
$bin/hbase org.apache.hadoop.hbase.IntegrationTestIngest -m slowDeterministic -monkeyProps monkey.properties
```

The above command will start the integration tests and chaos monkey passing the properties file `monkey.properties`. Here is an example chaos monkey file:

```
sdm.action1.period=120000
sdm.action2.period=40000
move.regions.sleep.time=80000
move.regions.max.time=1000000
move.regions.sleep.time=80000
batch.restart.rs.ratio=0.4f
```

18.10. Maven Build Commands

All commands executed from the local HBase project directory.

Note: use Maven 3 (Maven 2 may work but we suggest you use Maven 3).

Example 18.3. Compile

```
mvn compile
```

Running all or individual Unit Tests. See the Section 18.9.3, "Running tests" section above in Section 18.9.2, "Unit Tests"

18.10.1. Building against various hadoop versions.

As of 0.96, Apache HBase supports building against Apache Hadoop versions: 1.0.3, 2.0.0-alpha and 3.0.0-SNAPSHOT. By default, in 0.96 and earlier, we will build with Hadoop-1.0.x. As of 0.98, Hadoop 1.x is deprecated and Hadoop 2.x is the default. To change the version to build against, add a hadoop.profile property when you invoke **mvn**:

```
mvn -Dhadoop.profile=1.0 ...
```

The above will build against whatever explicit hadoop 1.x version we have in our `pom.xml` as our '1.0' version. Tests may not all pass so you may need to pass `-DskipTests` unless you are inclined to fix the failing tests.

> **'dependencyManagement.dependencies.dependency.artifactId' for org.apache.hbase:${compat.module}:test-jar with value '${compat.module}' does not match a valid id pattern**
>
> You will see ERRORs like the above title if you pass the *default* profile; e.g. if you pass hadoop.profile=1.1 when building 0.96 or hadoop.profile=2.0 when building hadoop 0.98; just drop the hadoop.profile stipulation in this case to get your build to run again. This seems to be a maven pecularity that is probably fixable but we've not spent the time trying to figure it.

Similarly, for 3.0, you would just replace the profile value. Note that Hadoop-3.0.0-SNAPSHOT does not currently have a deployed maven artificat - you will need to build and install your own in your local maven repository if you want to run against this profile.

In earilier versions of Apache HBase, you can build against older versions of Apache Hadoop, notably, Hadoop 0.22.x and 0.23.x. If you are running, for example HBase-0.94 and wanted to build against Hadoop 0.23.x, you would run with:

```
mvn -Dhadoop.profile=22 ...
```

18.11. Developing

18.11.1. Codelines

Most development is done on the master branch, which is named `master` in the Git repository. Previously, HBase used Subversion, in which the master branch was called `TRUNK`. Branches exist for minor releases, and important features and bug fixes are often back-ported.

18.11.2. Code Standards

See Section 18.3.1.1, "Code Formatting" and Section 18.12.4, "Code Formatting Conventions".

18.11.2.1. Interface Classifications

Interfaces are classified both by audience and by stability level. These labels appear at the head of a class. The conventions followed by HBase are inherited by its parent project, Hadoop.

The following interface classifications are commonly used:

@InterfaceAudience

`@InterfaceAudience.Public`

> APIs for users and HBase applications. These APIs will be deprecated through major versions of HBase.

`@InterfaceAudience.Private`

> APIs for HBase internals developers. No guarantees on compatibility or availability in future versions. Private interfaces do not need an `@InterfaceStability`

classification.

`@InterfaceAudience.LimitedPrivate(HBaseInterfaceAudience.COPROC)`

APIs for HBase coprocessor writers. As of HBase 0.92/0.94/0.96/0.98 this api is still unstable. No guarantees on compatibility with future versions.

No `@InterfaceAudience` Classification

Packages without an `@InterfaceAudience` label are considered private. Mark your new packages if publicly accessible.

Excluding Non-Public Interfaces from API Documentation

Only interfaces classified `@InterfaceAudience.Public` should be included in API documentation (Javadoc). Committers must add new package excludes `ExcludePackageNames` section of the `pom.xml` for new packages which do not contain public classes.

@InterfaceStability

`@InterfaceStability` is important for packages marked `@InterfaceAudience.Public`.

`@InterfaceStability.Stable`

Public packages marked as stable cannot be changed without a deprecation path or a very good reason.

`@InterfaceStability.Unstable`

Public packages marked as unstable can be changed without a deprecation path.

`@InterfaceStability.Evolving`

Public packages marked as evolving may be changed, but it is discouraged.

No `@InterfaceStability` Label

Public classes with no `@InterfaceStability` label are discouraged, and should be considered implicitly unstable.

If you are unclear about how to mark packages, ask on the development list.

18.11.3. Invariants

We don't have many but what we have we list below. All are subject to challenge of course but until then, please hold to the rules of the road.

18.11.3.1. No permanent state in ZooKeeper

ZooKeeper state should transient (treat it like memory). If ZooKeeper state is deleted, hbase should be able to recover and essentially be in the same state.

Exceptions

- There are currently a few exceptions that we need to fix around whether a table is enabled or disabled.
- Replication data is currently stored only in ZooKeeper. Deleting ZooKeeper data related to replication may cause replication to be disabled. Do not delete the replication tree, `/hbase/replication/`.

Warning

Replication may be disrupted and data loss may occur if you delete the replication tree (`/hbase/replication/`) from ZooKeeper. Follow progress on this issue at HBASE-10295.

18.11.4. Running In-Situ

If you are developing Apache HBase, frequently it is useful to test your changes against a more-real cluster than what you find in unit tests. In this case, HBase can be run directly from the source in local-mode. All you need to do is run:

```
${HBASE_HOME}/bin/start-hbase.sh
```

This will spin up a full local-cluster, just as if you had packaged up HBase and installed it on your machine.

Keep in mind that you will need to have installed HBase into your local maven repository for the in-situ cluster to work properly. That is, you will need to run:

```
mvn clean install -DskipTests
```

to ensure that maven can find the correct classpath and dependencies. Generally, the above command is just a good thing to try running first, if maven is acting oddly.

18.11.5. Adding Metrics

After adding a new feature a developer might want to add metrics. HBase exposes metrics using the Hadoop Metrics 2 system, so adding a new metric involves exposing that metric to the hadoop system. Unfortunately the API of metrics2 changed from hadoop 1 to hadoop 2. In order to get around this a set of interfaces and implementations have to be loaded at runtime. To get an in-depth look at the reasoning and structure of these classes you can read the blog post located here. To add a metric to an existing MBean follow the short guide below:

18.11.5.1. Add Metric name and Function to Hadoop Compat Interface.

Inside of the source interface the corresponds to where the metrics are generated (eg MetricsMasterSource for things coming from HMaster) create new static strings for metric name and description. Then add a new method that will be called to add new reading.

18.11.5.2. Add the Implementation to Both Hadoop 1 and Hadoop 2 Compat modules.

Inside of the implementation of the source (eg. MetricsMasterSourceImpl in the above example) create a new histogram, counter, gauge, or stat in the init method. Then in the method that was added to the interface wire up the parameter passed in to the histogram.

Now add tests that make sure the data is correctly exported to the metrics 2 system. For this the MetricsAssertHelper is provided.

18.12. Submitting Patches

HBase moved to GIT from SVN. Until we develop our own documentation for how to contribute patches in our new GIT context, caveat the fact that we have a different branching model and that we don't currently do the merge practice described in the following, the accumulo doc on how to contribute and develop after our move to GIT is worth a read.

If you are new to submitting patches to open source or new to submitting patches to Apache, start by reading the On Contributing Patches page from Apache Commons Project. It provides a nice overview that applies equally to the Apache HBase Project.

18.12.1. Create Patch

Patching Workflow

- Always patch against the master branch first, even if you want to patch in another branch. HBase committers always apply patches first to the master branch, and backport if necessary.
- Submit one single patch for a fix. If necessary, squash local commits to merge local commits into a single one first. See this Stack Overflow question for more information about squashing commits.
- The patch should have the JIRA ID in the name. If you generating from a branch, include the target branch in the filename. A common naming scheme for patches is:

  ```
  HBASE-XXXX.patch

  HBASE-XXXX-0.90.patch     # to denote that the patch is against branch 0.90
  ```

- To submit a patch, first create it using one of the methods in Methods to Create Patches. Next, attach the patch to the JIRA (one patch for the whole fix), using the More → Attach Files dialog. Next, click the Patch Available button, which triggers the Hudson job which checks the patch for validity.

  Please understand that not every patch may get committed, and that feedback will likely be provided on the patch.
- If your patch is longer than a single screen, also attach a Review Board to the case. See Section 18.12.5, “ReviewBoard”.
- If you need to revise your patch, leave the previous patch file(s) attached to the JIRA, and upload the new one, as in ???. Cancel the Patch Available flag and then re-trigger it, by toggling the Patch Available button in JIRA. JIRA sorts attached files by the time they were attached, and has no problem with multiple attachments with the same name.
- If you need to submit yuor patch against multiple branches, rather than just master, name each version of the patch with the branch it is for, as in ???.

Methods to Create Patches

Eclipse

Select the Team → Create Patch menu item.

Git

`git format-patch` is preferred because it preserves commit messages. Use `git squash` first, to combine smaller commits into a single larger one.

- `git format-patch --no-prefix origin/master --stdout > HBASE-XXXX.patch`
- `git diff --no-prefix origin/master > HBASE-XXXX.patch`

Subversion

```
svn diff > HBASE-XXXX.patch
```

Make sure you review Section 18.3.1.1, “Code Formatting” and Section 18.12.4, “Code Formatting Conventions” for code style.

18.12.2. Unit Tests

Yes, please. Please try to include unit tests with every code patch (and especially new classes and large changes). Make sure unit tests pass locally before submitting the patch.

Also, see Section 19.2, “Mockito”.

If you are creating a new unit test class, notice how other unit test classes have classification/sizing annotations at the top and a static method on the end. Be sure to include these in any new unit test files you generate. See Section 18.9, “Tests” for more on how the annotations work.

18.12.3. Integration Tests

Significant new features should provide an integration test in addition to unit tests, suitable for exercising the new feature at different points in its configuration space.

18.12.4. Code Formatting Conventions

Please adhere to the following guidelines so that your patches can be reviewed more quickly. These guidelines have been developed based upon common feedback on patches from new submitters.

See the Code Conventions for the Java Programming Language for more information on coding conventions in Java.

Example 18.4. Space Invaders

Do not use extra spaces around brackets. Use the second style, rather than the first.

```
if ( foo.equals( bar ) ) {     // don't do this

if (foo.equals(bar)) {
```

```
foo = barArray[ i ];     // don't do this

foo = barArray[i];
```

Example 18.5. Auto Generated Code

Auto-generated code in Eclipse often uses bad variable names such as `arg0`. Use more informative variable names. Use code like the second example here.

```
 public void readFields(DataInput arg0) throws IOException {    // don't do this
   foo = arg0.readUTF();                                        // don't do this

 public void readFields(DataInput di) throws IOException {
   foo = di.readUTF();
```

Example 18.6. Long Lines

Keep lines less than 100 characters. You can configure your IDE to do this automatically.

```
Bar bar = foo.veryLongMethodWithManyArguments(argument1, argument2, argument3, argument4, argument5, argument6, argument7, argument8,

Bar bar = foo.veryLongMethodWithManyArguments(
 argument1, argument2, argument3,argument4, argument5, argument6, argument7, argument8, argument9);
```

Example 18.7. Trailing Spaces

Trailing spaces are a common problem. Be sure there is a line break after the end of your code, and avoid lines with nothing but whitespace. This makes diffs more meaningful. You can configure your IDE to help with this.

```
Bar bar = foo.getBar();     <--- imagine there is an extra space(s) after the semicolon.
```

18.12.4.1. Implementing Writable

> ### Applies pre-0.96 only
>
> In 0.96, HBase moved to protocol buffers (protobufs). The below section on Writables applies to 0.94.x and previous, not to 0.96 and beyond.

Every class returned by RegionServers must implement the `Writable` interface. If you are creating a new class that needs to implement this interface, do not forget the default constructor.

18.12.4.2. API Documentation (Javadoc)

This is also a very common feedback item. Don't forget Javadoc!

Javadoc warnings are checked during precommit. If the precommit tool gives you a '-1', please fix the javadoc issue. Your patch won't be committed if it adds such warnings.

18.12.4.3. Findbugs

`Findbugs` is used to detect common bugs pattern. It is checked during the precommit build by Apache's Jenkins. If errors are found, please fix them. You can run findbugs locally with **mvn findbugs:findbugs**, which will generate the `findbugs` files locally. Sometimes, you may have to write code smarter than `findbugs`. You can annotate your code to tell `findbugs` you know what you're doing, by annotating your class with the following annotation:

```
@edu.umd.cs.findbugs.annotations.SuppressWarnings(
value="HE_EQUALS_USE_HASHCODE",
justification="I know what I'm doing")
```

It is important to use the Apache-licensed version of the annotations.

18.12.4.4. Javadoc - Useless Defaults

Don't just leave the @param arguments the way your IDE generated them.:

```
  /**
   *
   * @param bar             <---- don't do this!!!!
   * @return                <---- or this!!!!
   */
  public Foo getFoo(Bar bar);
```

Either add something descriptive to the `@param` and `@return` lines, or just remove them. The preference is to add something descriptive and useful.

18.12.4.5. One Thing At A Time, Folks

If you submit a patch for one thing, don't do auto-reformatting or unrelated reformatting of code on a completely different area of code.

Likewise, don't add unrelated cleanup or refactorings outside the scope of your Jira.

18.12.4.6. Ambigious Unit Tests

Make sure that you're clear about what you are testing in your unit tests and why.

18.12.5. ReviewBoard

Patches larger than one screen, or patches that will be tricky to review, should go through ReviewBoard.

Procedure 18.3. Use ReviewBoard

1. Register for an account if you don't already have one. It does not use the credentials from issues.apache.org. Log in.
2. Click New Review Request.
3. Choose the `hbase-git` repository. Click Choose File to select the diff and optionally a parent diff. Click Create Review Request.
4. Fill in the fields as required. At the minimum, fill in the Summary and choose `hbase` as the Review Group. If you fill in the Bugs field, the review board is attached to the relevant JIRA automatically. The more fields you fill in, the better. Click Publish to make your review request public. An email will be sent to everyone in the `hbase` group, to review the patch.
5. Back in your JIRA, click More → Link → Web Link, and paste in the URL of your ReviewBoard request.
6. To cancel the request, click Close → Discarded.

For more information on how to use ReviewBoard, see the ReviewBoard documentation.

18.12.6. Guide for HBase Committers

18.12.6.1. New committers

New committers are encouraged to first read Apache's generic committer documentation:

- Apache New Committer Guide
- Apache Committer FAQ

18.12.6.2. Review

HBase committers should, as often as possible, attempt to review patches submitted by others. Ideally every submitted patch will get reviewed by a committer *within a few days*. If a committer reviews a patch they have not authored, and believe it to be of sufficient quality, then they can commit the patch, otherwise the patch should be cancelled with a clear explanation for why it was rejected.

The list of submitted patches is in the HBase Review Queue, which is ordered by time of last modification. Committers should scan the list from top to bottom, looking for patches that they feel qualified to review and possibly commit.

For non-trivial changes, it is required to get another committer to review your own patches before commit. Use the Submit Patch button in JIRA, just like other contributors, and then wait for a `+1` response from another committer before committing.

18.12.6.3. Reject

Patches which do not adhere to the guidelines in HowToContribute and to the code review checklist should be rejected. Committers should always be polite to contributors and try to instruct and encourage them to contribute better patches. If a committer wishes to improve an unacceptable patch, then it should first be rejected, and a new patch should be attached by the committer for review.

18.12.6.4. Commit

Committers commit patches to the Apache HBase GIT repository.

> **Before you commit!!!!**
>
> Make sure your local configuration is correct, especially your identity and email. Examine the output of the **$ git config --list** command and be sure it is correct. See this GitHub article, Set Up Git if you need pointers.

When you commit a patch, please:

1. Include the Jira issue id in the commit message, along with a short description of the change and the name of the contributor if it is not you. Be sure to get the issue id right, as this causes Jira to link to the change in Subversion (use the issue's "All" tab to see these).
2. Resolve the issue as fixed, thanking the contributor. Always set the "Fix Version" at this point, but please only set a single fix version, the earliest release in which the change will appear.

18.12.6.4.1. Commit Message Format

The commit message should contain the JIRA ID and a description of what the patch does. The preferred commit message format is:

```
<jira-id> <jira-title> (<contributor-name-if-not-commit-author>)

HBASE-12345 Fix All The Things (jane@example.com)
```

If the submitter used **git format-patch** to generate the patch, their commit message is in their patch and you can use that.

18.12.6.4.2. Add Amending-Author when a conflict cherrypick backporting

We've established the practice of committing to trunk and then cherry picking back to branches whenever possible. When there is a minor conflict we can fix it up and just proceed with the commit. The resulting commit retains the original author. When the amending author is different from the original committer, add notice of this at the end of the commit message as: `Amending-Author: Author <committer&apache>` See discussion at HBase, mail # dev - [DISCUSSION] Best practice when amending commits cherry picked from master to branch.

18.12.6.4.3. Committers are responsible for making sure commits do not break the build or tests

If a committer commits a patch, it is their responsibility to make sure it passes the test suite. It is helpful if contributors keep an eye out that their patch does not break the hbase build and/or tests, but ultimately, a contributor cannot be expected to be aware of all the particular vagaries and interconnections that occur in a project like

HBase. A committer should.

18.12.6.4.4. Patching Etiquette

In the thread HBase, mail # dev - ANNOUNCEMENT: Git Migration In Progress (WAS => Re: Git Migration), it was agreed on the following patch flow

1. Develop and commit the patch against trunk/master first.
2. Try to cherry-pick the patch when backporting if possible.
3. If this does not work, manually commit the patch to the branch.

18.12.6.4.5. Merge Commits

Avoid merge commits, as they create problems in the git history.

18.12.6.4.6. Committing Documentation

See Appendix A, *Contributing to Documentation*.

18.12.7. Dialog

Committers should hang out in the #hbase room on irc.freenode.net for real-time discussions. However any substantive discussion (as with any off-list project-related discussion) should be re-iterated in Jira or on the developer list.

18.12.8. Do not edit JIRA comments

Misspellings and/or bad grammar is preferable to the disruption a JIRA comment edit causes: See the discussion at Re:(HBASE-451) Remove HTableDescriptor from HRegionInfo

Chapter 19. Unit Testing HBase Applications

Table of Contents

This chapter discusses unit testing your HBase application using JUnit, Mockito, MRUnit, and HBaseTestingUtility. Much of the information comes from a community blog post about testing HBase applications. For information on unit tests for HBase itself, see Section 18.9, “Tests”.

19.1. JUnit

HBase uses JUnit 4 for unit tests

This example will add unit tests to the following example class:

```
public class MyHBaseDAO {

    public static void insertRecord(HTableInterface table, HBaseTestObj obj)
    throws Exception {
        Put put = createPut(obj);
        table.put(put);
    }

    private static Put createPut(HBaseTestObj obj) {
        Put put = new Put(Bytes.toBytes(obj.getRowKey()));
        put.add(Bytes.toBytes("CF"), Bytes.toBytes("CQ-1"),
                    Bytes.toBytes(obj.getData1()));
        put.add(Bytes.toBytes("CF"), Bytes.toBytes("CQ-2"),
                    Bytes.toBytes(obj.getData2()));
        return put;
    }
}
```

The first step is to add JUnit dependencies to your Maven POM file:

```
<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.11</version>
    <scope>test</scope>
</dependency>
```

Next, add some unit tests to your code. Tests are annotated with `@Test`. Here, the unit tests are in bold.

```
public class TestMyHbaseDAOData {
  @Test
  public void testCreatePut() throws Exception {
  HBaseTestObj obj = new HBaseTestObj();
  obj.setRowKey("ROWKEY-1");
  obj.setData1("DATA-1");
  obj.setData2("DATA-2");
  Put put = MyHBaseDAO.createPut(obj);
  assertEquals(obj.getRowKey(), Bytes.toString(put.getRow()));
  assertEquals(obj.getData1(), Bytes.toString(put.get(Bytes.toBytes("CF"), Bytes.toBytes("CQ-1")).get(0).getValue()));
  assertEquals(obj.getData2(), Bytes.toString(put.get(Bytes.toBytes("CF"), Bytes.toBytes("CQ-2")).get(0).getValue()));
```

```
  }
}
```

These tests ensure that your `createPut` method creates, populates, and returns a `Put` object with expected values. Of course, JUnit can do much more than this. For an introduction to JUnit, see https://github.com/junit-team/junit/wiki/Getting-started.

19.2. Mockito

Mockito is a mocking framework. It goes further than JUnit by allowing you to test the interactions between objects without having to replicate the entire environment. You can read more about Mockito at its project site, https://code.google.com/p/mockito/.

You can use Mockito to do unit testing on smaller units. For instance, you can mock a `org.apache.hadoop.hbase.Server` instance or a `org.apache.hadoop.hbase.master.MasterServices` interface reference rather than a full-blown `org.apache.hadoop.hbase.master.HMaster`.

This example builds upon the example code in Chapter 19, *Unit Testing HBase Applications*, to test the `insertRecord` method.

First, add a dependency for Mockito to your Maven POM file.

```
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-all</artifactId>
    <version>1.9.5</version>
    <scope>test</scope>
</dependency>
```

Next, add a `@RunWith` annotation to your test class, to direct it to use Mockito.

```
@RunWith(MockitoJUnitRunner.class)
public class TestMyHBaseDAO{
  @Mock
  private HTableInterface table;
  @Mock
  private HTablePool hTablePool;
  @Captor
  private ArgumentCaptor putCaptor;

  @Test
  public void testInsertRecord() throws Exception {
    //return mock table when getTable is called
    when(hTablePool.getTable("tablename")).thenReturn(table);
    //create test object and make a call to the DAO that needs testing
    HBaseTestObj obj = new HBaseTestObj();
    obj.setRowKey("ROWKEY-1");
    obj.setData1("DATA-1");
    obj.setData2("DATA-2");
    MyHBaseDAO.insertRecord(table, obj);
    verify(table).put(putCaptor.capture());
    Put put = putCaptor.getValue();

    assertEquals(Bytes.toString(put.getRow()), obj.getRowKey());
    assert(put.has(Bytes.toBytes("CF"), Bytes.toBytes("CQ-1")));
    assert(put.has(Bytes.toBytes("CF"), Bytes.toBytes("CQ-2")));
    assertEquals(Bytes.toString(put.get(Bytes.toBytes("CF"),Bytes.toBytes("CQ-1")).get(0).getValue()), "DATA-1");
    assertEquals(Bytes.toString(put.get(Bytes.toBytes("CF"),Bytes.toBytes("CQ-2")).get(0).getValue()), "DATA-2");
  }
}
```

This code populates `HBaseTestObj` with “ROWKEY-1”, “DATA-1”, “DATA-2” as values. It then inserts the record into the mocked table. The Put that the DAO would have inserted is captured, and values are tested to verify that they are what you expected them to be.

The key here is to manage htable pool and htable instance creation outside the DAO. This allows you to mock them cleanly and test Puts as shown above. Similarly, you can now expand into other operations such as Get, Scan, or Delete.

19.3. MRUnit

Apache MRUnit is a library that allows you to unit-test MapReduce jobs. You can use it to test HBase jobs in the same way as other MapReduce jobs.

Given a MapReduce job that writes to an HBase table called `MyTest`, which has one column family called `CF`, the reducer of such a job could look like the following:

```
public class MyReducer extends TableReducer<Text, Text, ImmutableBytesWritable> {
   public static final byte[] CF = "CF".getBytes();
   public static final byte[] QUALIFIER = "CQ-1".getBytes();
   public void reduce(Text key, Iterable<Text> values, Context context) throws IOException, InterruptedException {
     //bunch of processing to extract data to be inserted, in our case, lets say we are simply
     //appending all the records we receive from the mapper for this particular
     //key and insert one record into HBase
     StringBuffer data = new StringBuffer();
     Put put = new Put(Bytes.toBytes(key.toString()));
     for (Text val : values) {
         data = data.append(val);
     }
     put.add(CF, QUALIFIER, Bytes.toBytes(data.toString()));
     //write to HBase
     context.write(new ImmutableBytesWritable(Bytes.toBytes(key.toString())), put);
   }
 }
```

To test this code, the first step is to add a dependency to MRUnit to your Maven POM file.

```
<dependency>
   <groupId>org.apache.mrunit</groupId>
   <artifactId>mrunit</artifactId>
   <version>1.0.0 </version>
```

```
   <scope>test</scope>
</dependency>
```

Next, use the ReducerDriver provided by MRUnit, in your Reducer job.

```
public class MyReducerTest {
    ReduceDriver<Text, Text, ImmutableBytesWritable, Writable> reduceDriver;
    byte[] CF = "CF".getBytes();
    byte[] QUALIFIER = "CQ-1".getBytes();

    @Before
    public void setUp() {
      MyReducer reducer = new MyReducer();
      reduceDriver = ReduceDriver.newReduceDriver(reducer);
    }

   @Test
   public void testHBaseInsert() throws IOException {
      String strKey = "RowKey-1", strValue = "DATA", strValue1 = "DATA1",
strValue2 = "DATA2";
      List<Text> list = new ArrayList<Text>();
      list.add(new Text(strValue));
      list.add(new Text(strValue1));
      list.add(new Text(strValue2));
      //since in our case all that the reducer is doing is appending the records that the mapper
      //sends it, we should get the following back
      String expectedOutput = strValue + strValue1 + strValue2;
     //Setup Input, mimic what mapper would have passed
      //to the reducer and run test
      reduceDriver.withInput(new Text(strKey), list);
      //run the reducer and get its output
      List<Pair<ImmutableBytesWritable, Writable>> result = reduceDriver.run();

      //extract key from result and verify
      assertEquals(Bytes.toString(result.get(0).getFirst().get()), strKey);

      //extract value for CF/QUALIFIER and verify
      Put a = (Put)result.get(0).getSecond();
      String c = Bytes.toString(a.get(CF, QUALIFIER).get(0).getValue());
      assertEquals(expectedOutput,c );
   }

}
```

Your MRUnit test verifies that the output is as expected, the Put that is inserted into HBase has the correct value, and the ColumnFamily and ColumnQualifier have the correct values.

MRUnit includes a MapperDriver to test mapping jobs, and you can use MRUnit to test other operations, including reading from HBase, processing data, or writing to HDFS,

19.4. Integration Testing with a HBase Mini-Cluster

HBase ships with HBaseTestingUtility, which makes it easy to write integration tests using a *mini-cluster*. The first step is to add some dependencies to your Maven POM file. Check the versions to be sure they are appropriate.

```
<dependency>
    <groupId>org.apache.hadoop</groupId>
    <artifactId>hadoop-common</artifactId>
    <version>2.0.0</version>
    <type>test-jar</type>
    <scope>test</scope>
</dependency>

<dependency>
    <groupId>org.apache.hbase</groupId>
    <artifactId>hbase</artifactId>
    <version>0.98.3</version>
    <type>test-jar</type>
    <scope>test</scope>
</dependency>

<dependency>
    <groupId>org.apache.hadoop</groupId>
    <artifactId>hadoop-hdfs</artifactId>
    <version>2.0.0</version>
    <type>test-jar</type>
    <scope>test</scope>
</dependency>

<dependency>
    <groupId>org.apache.hadoop</groupId>
    <artifactId>hadoop-hdfs</artifactId>
    <version>2.0.0</version>
    <scope>test</scope>
</dependency>
```

This code represents an integration test for the MyDAO insert shown in Chapter 19, *Unit Testing HBase Applications*.

```
public class MyHBaseIntegrationTest {
    private static HBaseTestingUtility utility;
    byte[] CF = "CF".getBytes();
    byte[] QUALIFIER = "CQ-1".getBytes();

    @Before
    public void setup() throws Exception {
        utility = new HBaseTestingUtility();
        utility.startMiniCluster();
    }
```

```
    @Test
        public void testInsert() throws Exception {
         HTableInterface table = utility.createTable(Bytes.toBytes("MyTest"),
                         Bytes.toBytes("CF"));
         HBaseTestObj obj = new HBaseTestObj();
         obj.setRowKey("ROWKEY-1");
         obj.setData1("DATA-1");
         obj.setData2("DATA-2");
         MyHBaseDAO.insertRecord(table, obj);
         Get get1 = new Get(Bytes.toBytes(obj.getRowKey()));
         get1.addColumn(CF, CQ1);
         Result result1 = table.get(get1);
         assertEquals(Bytes.toString(result1.getRow()), obj.getRowKey());
         assertEquals(Bytes.toString(result1.value()), obj.getData1());
         Get get2 = new Get(Bytes.toBytes(obj.getRowKey()));
         get2.addColumn(CF, CQ2);
         Result result2 = table.get(get2);
         assertEquals(Bytes.toString(result2.getRow()), obj.getRowKey());
         assertEquals(Bytes.toString(result2.value()), obj.getData2());
    }
}
```

This code creates an HBase mini-cluster and starts it. Next, it creates a table called `MyTest` with one column family, `CF`. A record is inserted, a Get is performed from the same table, and the insertion is verified.

Note

Starting the mini-cluster takes about 20-30 seconds, but that should be appropriate for integration testing.

To use an HBase mini-cluster on Microsoft Windows, you need to use a Cygwin environment.

See the paper at HBase Case-Study: Using HBaseTestingUtility for Local Testing and Development (2010) for more information about HBaseTestingUtility.

Chapter 20. ZooKeeper

Table of Contents

A distributed Apache HBase installation depends on a running ZooKeeper cluster. All participating nodes and clients need to be able to access the running ZooKeeper ensemble. Apache HBase by default manages a ZooKeeper "cluster" for you. It will start and stop the ZooKeeper ensemble as part of the HBase start/stop process. You can also manage the ZooKeeper ensemble independent of HBase and just point HBase at the cluster it should use. To toggle HBase management of ZooKeeper, use the `HBASE_MANAGES_ZK` variable in `conf/hbase-env.sh`. This variable, which defaults to `true`, tells HBase whether to start/stop the ZooKeeper ensemble servers as part of HBase start/stop.

When HBase manages the ZooKeeper ensemble, you can specify ZooKeeper configuration 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 HBase Default Configuration. Look for the `hbase.zookeeper.property` prefix [29]

You must at least list the ensemble servers in `hbase-site.xml` using the `hbase.zookeeper.quorum` property. This property defaults to a single ensemble member at `localhost` which is not suitable for a fully distributed HBase. (It binds to the local machine only and remote clients will not be able to connect).

How many ZooKeepers should I run?

You can run a ZooKeeper ensemble that comprises 1 node only but in production it is recommended that you run a ZooKeeper ensemble of 3, 5 or 7 machines; the more members an ensemble has, the more tolerant the ensemble is of host failures. Also, run an odd number of machines. In ZooKeeper, an even number of peers is supported, but it is normally not used because an even sized ensemble requires, proportionally, more peers to form a quorum than an odd sized ensemble requires. For example, an ensemble with 4 peers requires 3 to form a quorum, while an ensemble with 5 also requires 3 to form a quorum. Thus, an ensemble of 5 allows 2 peers to fail, and thus is more fault tolerant than the ensemble of 4, which allows only 1 down peer.

Give each ZooKeeper server around 1GB of RAM, and if possible, its own dedicated disk (A dedicated disk is the best thing you can do to ensure a performant ZooKeeper ensemble). For very heavily loaded clusters, run ZooKeeper servers on separate machines from RegionServers (DataNodes and TaskTrackers).

For example, to have HBase manage a ZooKeeper quorum on nodes *rs{1,2,3,4,5}.example.com*, bound to port 2222 (the default is 2181) ensure `HBASE_MANAGE_ZK` is commented out or set to `true` in `conf/hbase-env.sh` and then edit `conf/hbase-site.xml` and set `hbase.zookeeper.property.clientPort` and `hbase.zookeeper.quorum`. You should also set `hbase.zookeeper.property.dataDir` to other than the default as the default has ZooKeeper persist data under `/tmp` which is often cleared on system restart. In the example below we have ZooKeeper persist to `/user/local/zookeeper`.

```
  <configuration>
    ...
    <property>
      <name>hbase.zookeeper.property.clientPort</name>
      <value>2222</value>
      <description>Property from ZooKeeper's config zoo.cfg.
      The port at which the clients will connect.
      </description>
    </property>
```