HowToContribute

How to Contribute to Apache Hive

This page describes the mechanics of *how* to contribute software to Apache Hive. For ideas about *what* you might contribute, please see open tickets in Jira .

Getting the Source Code

First of all, you need the Hive source code.

Get the source code on your local drive using git. See Understanding Hive Branches below to understand which branch you should be using.

```
git clone https://github.com/apache/hive
```

Setting Up Eclipse Development Environment (Optional)

This is an optional step. Eclipse has a lot of advanced features for Java development, and it makes the life much easier for Hive developers as well.

How do I import into eclipse?

Becoming a Contributor

This checklist tells you how to create accounts and obtain permissions needed by Hive contributors. See the Hive website for additional information.

- [ ] Request an Apache Software Foundation JIRA account, if you do not already have one.
  - The ASF JIRA system dashboard is here.
  - The Hive JIRA is here.
- [ ] To review patches you could do that on Github
  - All Hive patches posted for review are listed here.
  - Individual JIRA tickets provide a link to the issue on the review board when a review request has been made.
- [ ] To contribute to the Hive wiki, follow the instructions in About This Wiki.
- [ ] To edit the Hive website, follow the instructions in How to edit the website.
- [ ] Join the Hive mailing lists to receive email about issues and discussions.

Making Changes

If you're a newcomer, feel free to contribute by working on a newbie task.

Before you start, send a message to the Hive developer mailing list, or file a bug report in JIRA. Describe your proposed changes and check that they fit in with what others are doing and have planned for the project. Be patient, it may take folks a while to understand your requirements.

Modify the source code and add some features using your favorite IDE.

Coding Conventions

Please take care about the following points.

- All public classes and methods should have informative Javadoc comments.
  - Do not use @author tags.
- Code should be formatted according to Sun's conventions, with two exceptions:
  - Indent two (2) spaces per level, not four (4).
  - Line length limit is 120 chars, instead of 80 chars.

  An Eclipse formatter is provided in the dev-support folder – this can be used with both Eclipse and Intellij. Please consider importing this before editing the source code.

  - For Eclipse:
    - Go to Preferences -> Java -> Code Style -> Formatter; Import eclipse-styles.xml; Apply.
    - In addition update save actions: Java -> Editor -> Save Actions; Check the following: Perform the following actions on save; Format Source Code; Format edited lines.
  - For Intellij:
    - Go to Settings -> Editor -> Code Style -> Java -> Scheme; Click manage; Import eclipse-styles.xml; Apply.
- Contributions should not introduce new Checkstyle violations.
  - Check for new Checkstyle violations by running `mvn checkstyle:checkstyle-aggregate`, and then inspect the results in the `target/site` directory. It is possible to run the checks for a specific module, if the `mvn` command is issued in the root directory of the module.
  - If you use Eclipse you should install the eclipse-cs Checkstyle plugin. This plugin highlights violations in your code and is also able to automatically correct some types of violations.
- Contributions should pass existing unit tests.
- New unit tests should be provided to demonstrate bugs and fixes. JUnit is our test framework:
  - You should create test classes for junit4, whose class name must start with a 'Test' prefix.
  - You can run all the unit tests with the command `mvn test`, or you can run a specific unit test with the command `mvn test -Dtest=<class name without package prefix>` (for example: `mvn test -Dtest=TestFileSystem`).
  - After uploading your patch, it might worthwhile to check if your new test has been executed in the precommit job.

Understanding Maven

Hive is a **multi-module** Maven project. If you are new to Maven, the articles below may be of interest:

- Maven in Five Minutes
- Maven getting started
- Maven by example - a multi-module project

Additionally, Hive actually has two projects, "core" and "itests". The reason that itests is not connected to the core reactor is that itests requires the packages to be built.

The actual Maven commands you will need are discussed on the HiveDeveloperFAQ page.

Understanding Hive Branches

Hive has a few "main lines", master and branch-X.

All new feature work and bug fixes in Hive are contributed to the master branch. Releases are done from branch-X. Major versions like 2.x versions are not necessarily backwards compatible with 1.x versions.backwards compatibility will be accepted on branch-1.

In addition to these main lines Hive has two types of branches, release branches and feature branches.

Release branches are made from branch-1 (for 1.x) or master (for 2.x) when the community is preparing a Hive release. Release branches match the number of the release (e.g., branch-1.2 for Hive 1.2). For patch releases the branch is made from the existing release branch (to avoid picking up new features from the main line). For example, if a 1.2.1 release was being made branch-1.2.1 would be made from the tip of branch-1.2. Once a release branch has been made, inclusion of additional patches on that branch is at the discretion of the release manager. After a release has been made from a branch, additional bug fixes can still be applied to that branch in anticipation of the next patch release. Any bug fix applied to a release branch must first be applied to master (and branch-1 if applicable).

Feature branches are used to develop new features without destabilizing the rest of Hive. The intent of a feature branch is that it will be merged back into master once the feature has stabilized.

For general information about Hive branches, see Hive Versions and Branches.

Hadoop Dependencies

Hadoop dependencies are handled differently in master and branch-1.

branch-1

In branch-1 both Hadoop 1.x and 2.x are supported. The Hive build downloads a number of different Hadoop versions via Maven in order to compile "shims" which allow for compatibility with these Hadoop versions. However, the rest of Hive is only built and tested against a single Hadoop version.

The Maven build has two profiles, `hadoop-1` for Hadoop 1.x and `hadoop-2` for Hadoop 2.x. When building, you must specify which profile you wish to use via Maven's `-P` command line option (see How to build all source).

branch-2

Hadoop 1.x is no longer supported in Hive's master branch. There is no need to specify a profile for most Maven commands, as Hadoop 2.x will always be chosen.

> **Hadoop Version Information**
>
> On this page we assume you are building from the master branch and do not include the profile in the example Maven commands. If you are building on branch-1 you will need to select the appropriate profile for the version of Hadoop you are building against.

Unit Tests

When submitting a patch it's highly recommended you execute tests locally which you believe will be impacted in addition to any new tests. The full test suite can be executed by Hive PreCommit Patch Testing. Hive Developer FAQ describes how to execute a specific set of tests.

```
mvn clean install -DskipTests
mvn test -Dtest=SomeTest
```

Unit tests take a long time (several hours) to run sequentially even on a very fast machine.

Add a Unit Test

There are two kinds of unit tests that can be added: those that test an entire component of Hive, and those that run a query to test a feature.

Java Unit Test

To test a particular component of Hive:

- Add a new class (name must start with `Test`) in the component's `*/src/test/java` directory.
- To test only the new testcase, run `mvn test -Dtest=TestAbc` (where `TestAbc` is the name of the new class), which will be faster than `mvn test` which tests all testcases.

Query Unit Test

If the new feature can be tested using a Hive query in the command line, we just need to add a new `*.q` file and a new `*.q.out` file.

If the feature is added in `ql` (query language):

- Add a new `XXXXXX.q` file in `ql/src/test/queries/clientpositive`. (Optionally, add a new `XXXXXX.q` file for a query that is expected to fail in `ql/src/test/queries/clientnegative`.)
- Run `mvn test -Dtest=TestMiniLlapLocalCliDriver -Dqfile=XXXXXX.q -Dtest.output.overwrite=true`. This will generate a new `XXXXXX.q.out` file in `ql/src/test/results/clientpositive`.
    - If you want to run multiple .q files in the test run, you can specify comma separated .q files, for example `-Dqfile="X1.q,X2.q"`. You can also specify a Java regex, for example `-Dqfile_regex='join.*'`. (Note that it takes Java regex, i.e., `'join.*'` and not `'join*'`.) The regex match first removes the `.q` from the file name before matching regex, so specifying `join*.q` will not work.

If the feature is added in `contrib`:

- Do the steps above, replacing `ql` with `contrib`, and `TestCliDriver` with `TestContribCliDriver`.

See the FAQ "How do I add a test case?" for more details.

Beeline Query Unit Test

Legacy query test Drivers (all of them except TestBeeLineDriver) uses HiveCli to run the tests. TestBeeLineDriver runs the tests using the Beeline client. Creates a specific database for them, so the tests can run parallel. Running the tests you have the following configuration options:

- `-Dqfile=XXXXXX.q` - To run one or more specific query file tests. For the exact format, check the Query Unit Test paragraph. If not provided only those query files from `ql/src/test/queries/clientpositive` directory will be run which are mentioned in `itests/src/test/resources/testconfiguration.properties` in the `beeline.positive.include` parameter.
- `-Dtest.output.overwrite=true` - This will rewrite the output of the q.out files in `ql/src/test/results/clientpositive/beeline`. The default value is false, and it will check the current output against the golden files
- `-Dtest.beeline.compare.portable` - If this parameter is true, the generated and the golden query output files will be filtered before comparing them. This way the existing query tests can be run against different configurations using the same golden output files. The result of the following commands will be filtered out from the output files: EXPLAIN, DESCRIBE, DESCRIBE EXTENDED, DESCRIBE FORMATTED, SHOW

TABLES, SHOW FORMATTED INDEXES and SHOW DATABASES.
The default value is `false`.
- `-Djunit.parallel.threads=1` - The number of the parallel threads running the tests. The default is 1. There were some flakiness caused by parallelization
- `-Djunit.parallel.timeout=10` - The tests are terminated after the given timeout. The parameter is set in minutes and the default is 10 minutes. (As of HIVE 3.0.0.)
- The BeeLine tests could run against an existing cluster. Or if not provided, then against a MiniHS2 cluster created during the tests.
    - `-Dtest.beeline.url` - The jdbc url which should be used to connect to the existing cluster. If not set then a MiniHS2 cluster will be created instead.
    - `-Dtest.beeline.user` - The user which should be used to connect to the cluster. If not set `"user"` will be used.
    - `-Dtest.beeline.password` - The password which should be used to connect to the cluster. If not set `"password"` will be used.
    - `-Dtest.data.dir` - The test data directory on the cluster. If not set `<HIVEROOT>/data/files` will be used.
    - `-Dtest.results.dir` - The test results directory to compare against. If not set the default configuration will be used.
    - `-Dtest.init.script` - The test init script. If not set the default configuration will be used.
    - `-Dtest.beeline.shared.database` - If true, then the default database will be used, otherwise a test-specific database will be created for every run. The default value is false.

Debugging

Please see Debugging Hive code in Development Guide.

Submitting a PR

There are many excellent howtos about how to submit pullrequests for github projects. The following is one way to approach it:

Setting up a repo with 2 remotes; I would recommend to use the github user as the remote name - as it may make things easier if you need to add someone else's repo as well.

```
# clone the apache/hive repo from github
git clone --origin apache https://github.com/apache/hive
cd hive
# add your own fork as a remote
git remote add GITHUB_USER git@github.com:GITHUB_USER/hive
```

You will need a separate branch to make your changes; you need to this for every PR you are doing.

```
# fetch all changes - so you will create your feature branch on top of the current master
git fetch --all
# create a feature branch This branch name can be anything - including the ticket id may help later on
identifying the branch.
git branch HIVE-9999-something apache/master
git checkout HIVE-9999-something
# push your feature branch to your github fork - and set that branch as upstream to this branch
git push GITHUB_USER -u HEAD
```

Make your change

```
# make your changes; you should include the ticketid + message in the first commit message
git commit -m 'HIVE-9999: Something' -a
# a simple push will deliver your changes to the github branch
git push
```

If you think your changes are ready to be tested and reviewed - you could open a PR request on the https://github.com/apache/hive page.

If you need to make changes you just need to push further changes to the branch.

Please do not:

- reformat code unrelated to the issue being fixed: formatting changes should be separate patches/commits;
- comment out code that is now obsolete: just remove it;
- insert comments around each change, marking the change: folks can use git to figure out what's changed and by whom;
- make things public that are not required by end-users.

Please do:

- try to adhere to the coding style of files you edit;
- comment code whose function or rationale is not obvious;
- add one or more unit tests (see Add a Unit Test above);
- update documentation (such as Javadocs including *package.html* files and this wiki).

Fetching a PR from Github

you could do that using:

```
git fetch origin pull/ID/head:BRANCHNAME
```

Suppose you want to pull the changes of PR-1234 into a local branch named "radiator"

```
git fetch origin pull/1234/head:radiator
```

Contributing Your Work

You should open a JIRA ticket about the issue you are about to fix.

Upload your changes to your github fork and open a PR against the hive repo.

If your patch creates an incompatibility with the latest major release, then you must set the **Incompatible change** flag on the issue's JIRA *and* fill in the **Release Note** field with an explanation of the impact of the incompatibility and the necessary steps users must take.

If your patch implements a major feature or improvement, then you must fill in the **Release Note** field on the issue's JIRA with an explanation of the feature that will be comprehensible by the end user.

The **Release Note** field can also document changes in the user interface (such as new HiveQL syntax or configuration parameters) prior to inclusion in the wiki documentation.

A committer should evaluate the patch within a few days and either: commit it; or reject it with an explanation.

Please be patient. Committers are busy people too. If no one responds to your patch after a few days, please make friendly reminders. Please incorporate others' suggestions into your patch if you think they're reasonable. Finally, remember that even a patch that is not committed is useful to the community.

Should your patch receive a "-1" select **Resume Progress** on the issue's JIRA, upload a new patch with necessary fixes, and then select the **Submit Patch** link again.

Committers: for non-trivial changes, it is best to get another committer to review your patches before commit. Use the **Submit Patch** link like other contributors, and then wait for a "+1" from another committer before committing. Please also try to frequently review things in the patch queue.

JIRA

Hive uses JIRA for issues/case management. You must have a JIRA account in order to log cases and issues.

Requests for the creation of new accounts can be submitted via the following form: https://selfserve.apache.org/jira-account.html

Guidelines

Please comment on issues in JIRA, making your concerns known. Please also vote for issues that are a high priority for you.

Please refrain from editing descriptions and comments if possible, as edits spam the mailing list and clutter JIRA's "All" display, which is otherwise very useful. Instead, preview descriptions and comments using the preview button (icon below the comment box) before posting them.

Keep descriptions brief and save more elaborate proposals for comments, since descriptions are included in JIRA's automatically sent messages. If you change your mind, note this in a new comment, rather than editing an older comment. The issue should preserve this history of the discussion.

To open a JIRA issue, click the Create button on the top line of the Hive summary page or any Hive JIRA issue.

Please leave Fix Version/s empty when creating the issue – it should not be tagged until an issue is closed, and then, it is tagged by the committer closing it to indicate the earliest version(s) the fix went into. Instead of Fix Version/s, use Target Version/s to request which versions the new issue's patch should go into. (Target Version/s was added to the Create Issue form in November 2015. You can add target versions to issues created before that with the Edit button, which is in the upper left corner.)

Consider using bi-directional links when referring to other tickets. It is very common and convenient to refer to other tickets by adding the HIVE-XXXXX pattern in summary, description, and comments. The pattern allows someone to navigate quickly to an older JIRA from the current one but not the other way around. Ideally, along with the mention (HIVE-XXXXX) pattern, it helps to add an explicit link (relates to, causes, depends upon, etc.) so that the relationship between tickets is visible from both ends.

Add the "backward-incompatible" label to tickets changing the behavior of some component or introduce modifications to public APIs. There are various other labels available for similar purposes but this is the most widely used across projects so it is better to stick to it to keep things uniform.

When in doubt about how to fill in the Create Issue form, take a look at what was done for other issues. Here are several Hive JIRA issues that you can use as examples:

- bug: HIVE-8485, HIVE-8600, HIVE-9438, HIVE-11174
- new feature: HIVE-6806, HIVE-7088, HIVE-11103
- improvement: HIVE-7685, HIVE-9858, HIVE-10165
- test: HIVE-8601, HIVE-10637
- wish: HIVE-4563, HIVE-10427
- task: HIVE-7111, HIVE-7789

Many examples of uncommitted issues are available in the "Added recently" list on the issues panel.

Generating Thrift Code

Some portions of the Hive code are generated by Thrift. For most Hive changes, you don't need to worry about this, but if you modify any of the Thrift IDL files (e.g., **standalone-metastore/metastore-common/src/main/thrift/hive_metastore.thrift** and **service-rpc/if/TCLIService.thrift**), then you'll also need to regenerate these files and submit their updated versions as part of your patch.

Here are the steps relevant to `hive_metastore.thrift`:

1. Don't make any changes to `hive_metastore.thrift` until instructed below.
2. Use the approved version of Thrift. This is currently `thrift-0.14.1`, which you can obtain from http://thrift.apache.org/.
    a. For Mac via Homebrew (since the version we need is not available by default):

```
brew tap-new $USER/local-tap
brew extract --version='0.14.1' thrift $USER/local-tap
brew install thrift@0.14.1
mkdir -p /usr/local/share/fb303/if
cp /usr/local/Cellar/thrift@0.14.1/0.14.1/share/fb303/if/fb303.thrift /usr/local/share/fb303/if
```

    b. For Mac, building from sources:

```
wget http://archive.apache.org/dist/thrift/0.14.1/thrift-0.14.1.tar.gz

tar xzf thrift-0.14.1.tar.gz

#If configure fails with "syntax error near unexpected token `QT5", then run "brew install pkg-
config"

./bootstrap.sh

sudo ./configure --with-openssl=/usr/local/Cellar/openssl@1.1/1.1.1j --without-erlang --without-
nodejs --without-python --without-py3 --without-perl --without-php --without-php_extension --
without-ruby --without-haskell --without-go --without-swift --without-dotnetcore --without-qt5

brew install openssl

sudo ln -s /usr/local/opt/openssl/include/openssl/ /usr/local/include/

sudo make

sudo make install

mkdir -p /usr/local/share/fb303/if

cp path/to/thrift-0.14.1/contrib/fb303/if/fb303.thrift /usr/local/share/fb303/if/fb303.thrift
# or alternatively the following command
curl -o /usr/local/share/fb303/if/fb303.thrift https://raw.githubusercontent.com/apache/thrift
/master/contrib/fb303/if/fb303.thrift
```

    c. For Linux:

```
cd /path/to/thrift-0.14.1
/configure -without-erlang --without-nodejs --without-python --without-py3 --without-perl --
without-php --without-php_extension --without-ruby --without-haskell --without-go --without-swift
--without-dotnetcore --without-qt5
sudo make
sudo make install
sudo mkdir -p /usr/local/share/fb303/if
sudo cp /path/to/thrift-0.14.1/contrib/fb303/if/fb303.thrift /usr/local/share/fb303/if/fb303.thrift
```

3. Before proceeding, verify that `which thrift` returns the build of Thrift you just installed (typically `/usr/local/bin` on Linux); if not, edit your PATH and repeat the verification. Also verify that the command 'thrift -version' returns the expected version number of Thrift.
4. Now you can run the Maven 'thriftif' profile to generate the Thrift code:
    a. `cd /path/to/hive/`
    b. `mvn clean install -Pthriftif -DskipTests -Dthrift.home=/usr/local`
5. Verify that the code generation was a no-op, which should be the case if you have the correct Thrift version and everyone has been following these instructions. You may use `git status` for the same. If you can't figure out what is going wrong, ask for help from a committer.
6. Now make your changes to `hive_metastore.thrift`, and then run the compiler again, from /path/to/hive/<hive_metastore.thrift's module>:
    a. `mvn clean install -Pthriftif -DskipTests -Dthrift.home=/usr/local`
7. Now use `git status and git diff` to verify that the regenerated code corresponds only to the changes you made to `hive_metastore.thrift`. You may also need `git add` if new files were generated (and or `git rm` if some files are now obsoleted).
8. `cd /path/to/hive`
9. `mvn clean package -DskiptTests (at the time of writing also "-Dmaven.javadoc.skip" is needed)`
10. Verify that Hive is still working correctly with both embedded and remote metastore configurations.

Stay Involved

Contributors should join the Hive mailing lists. In particular the dev list (to join discussions of changes) and the user list (to help others).

See Also

- Apache contributor documentation
- Apache voting documentation
- How to edit the website