HowToContribute

Getting the source code

First of all, you need the Pig source code.

Get the source code on your local drive using SVN. Most development is done on the "trunk":

svn checkout http://svn.apache.org/repos/asf/pig/trunk/

or

git clone https://github.com/apache/pig.git

Making Changes

Before you start, send a message to the Pig 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 (very) nice features or fix some (nasty) bugs using your favorite IDE.

But 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. We use four spaces (**not** tabs) for indentation.
- Contributions should pass unit tests.
- New unit tests should be provided to demonstrate bugs and fixes. JUnit is our test framework:
    - You must implement a class whose class name contains `Test`.
    - If an `HDFS` cluster and/or a `MapReduce` cluster is needed by your test, add a field of type `MiniGenericCluster` to the class and initialize it with a statement like the following (the name of the field is not important). `TestAlgebraicEval.java` is an example of a test that uses cluster. The test will then run on a cluster created on the local machine.
      `MiniGenericCluster cluster = MiniGenericCluster.buildCluster();`
- Define methods within your class and annotate it with `@Test`, and call JUnit's many assert methods to verify conditions; these methods will be executed when you run `ant test`.
- Place your class in the `test` tree.
- You can then run the core unit test with the command `ant test-commit`. Similarly, you can run a specific unit test with the command `ant test -Dtestcase=<ClassName>` (For example `ant test -Dtestcase=TestPigFile`)

Generating a patch

Compilation

Make sure that your code introduces no new warnings into the javac compilation.

To compile with Hadoop 2.x:

```
> ant clean jar
```

The latest Pig codebase only supports Hadoop 2.x which is based on YARN and has separate Resource Manager and Application Masters instead of a single JobTracker that managed both resources (cpu, memory) and running of mapreduce applications. The exact versions of Hadoop 2.x pig compiles against is configured in `ivy/libraries.properties` and is usually updated to compile against the latest stable releases.

Please note that in earlier versions Pig used to support older Hadoop versions too, and there was an option to select a certain Hadoop version at build time. If you would like to contribute to older release branches (0.16.0 or below) you will have to set the hadoopversion property. It has 2 values - 20 and 23. `-Dhadoopversion=20` which is the default denotes the Hadoop 0.20.x and 1.x releases which are the old versions with JobTracker. The other option, `-Dhadoopversion=23` denotes the Hadoop 0.23.x and Hadoop 2.x releases.

Unit Tests

The full suite of pig unit tests has a huge number of tests and there are multiple execution modes - mapreduce (default), spark, tez against which the whole test suite can be run. Since it takes a really long time, you are not expected to run the full suite of tests before submitting the patch. You can just run and verify the test classes affected by your patch and also run test-commit which runs a core set of tests that takes 20 mins. If the fix is specific to a particular execution mode (For eg: tez or spark), run the tests with that exectype. The Pig commit build (https://builds.apache.org/job/Pig-trunk-commit) which runs daily will report any additional failures on the committed patch and a new patch can be submitted that fixes those failures later. Some of the different test goals are `test` - full suite of unit tests in mapreduce mode, `test-tez` - full suite of unit tests in tez mode, `test-commit` - core set of tests in mapreduce mode.

To run the full suite of testcases in mapreduce mode with Hadoop 2.x. Usually you don't have to run this unless you are doing major changes.

```
> ant clean test
```

To run the full suite of testcases in tez mode with Hadoop 2.x. This is a shortcut which takes care of adding `-Dexectype=tez` . Usually you don't have to run this unless you are doing major changes.

```
> ant clean test-tez
```

To run a single testcase with Hadoop 2.x. You can do this to verify the new tests that you have added or run specific testcases affected by your patch.

```
> ant clean test -Dtestcase=TestEvalPipeline
```

To run a single testcase with Hadoop 2.x and tez as execution engine

```
> ant clean test -Dtestcase=TestEvalPipeline2 -Dexectype=tez
```

To run the core set of unit tests follow below steps. Please make sure that all the core unit tests and the tests you wrote succeed before constructing your patch.

```
> cd trunk
> ant -Djavac.args="-Xlint -Xmaxwarns 1000" clean test-commit
```

This should run in around 20 minutes.

After a while, if you see

```
BUILD SUCCESSFUL
```

all is ok, but if you see

```
BUILD FAILED
```

then please examine error messages in build/test and fix things before proceeding.

Javadoc

Please also check the javadoc.

```
> ant docs
> firefox build/docs/api/index.html
```

Examine all public classes you've changed to see that documentation is complete and informative. Your patch must not generate any javadoc warnings.

Creating a patch

Check to see what files you have modified with:

```
svn stat
```

Add any new files with:

```
svn add src/.../MyNewClass.java
```

In order to create a patch, just type:

```
svn diff > myBeautifulPatch.patch
```

This will report all modifications done on Pig sources on your local disk and save them into the myBeautifulPath.patch file. Read the patch file. Make sure it includes ONLY the modifications required to fix a single issue.

Please do not:

- reformat code unrelated to the bug 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 subversion to figure out what's changed and by whom.
- make things public which 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;
- update documentation (e.g., package.html files, this wiki, etc.)

If you need to rename files in your patch:

1. Write a shell script that uses 'svn mv' to rename the original files.
2. Edit files as needed (e.g., to change package names).
3. Create a patch file with 'svn diff --no-diff-deleted --notice-ancestry'.
4. Submit both the shell script and the patch file.
   This way other developers can preview your change by running the script and then applying the patch.

Creating a patch with git

If working from a git repo, please be aware the the default diff format will not apply in SVN repos. Please generate patches with the `--no-prefix` option so they apply cleanly.

```
git diff --no-prefix
```

~~h3. Testing a patch~~ (**Ignore this section for now**)

You can run the same tools that the automated Jenkins patch test system will run on a patch. This enables you to fix problems with your patch once Jenkins or a committer points them out. The `test-patch` Ant target will run your patch through the same checks that Jenkins currently does *except* for executing the core and contrib unit tests.

To use this target, you must run it from a clean workspace (ie `svn stat` shows no modifications or additions). From your clean workspace, run:

```
ant \
  -Dpatch.file=/patch/to/my.patch \
  -Dforrest.home=/path/to/forrest/ \
  -Dfindbugs.home=/path/to/findbugs \
  -Djava5.home=/patch/to/java5home \
  -Dscratch.dir=/path/to/a/temp/dir \ (optional)
  -Dsvn.cmd=/path/to/subversion/bin/svn \ (optional)
  -Dgrep.cmd=/path/to/grep \ (optional)
  -Dpatch.cmd=/path/to/patch \ (optional)
  test-patch
```

At the end, you should get a message on your console that is similar to the comment added to Jira by Jenkins' automated patch test system. The scratch directory (which defaults to the value of ${user.home}/tmp) will contain some output files that will be useful in determining what issues were found in the patch.

Some things to note:

- the optional cmd parameters will default to the ones in your `PATH` environment variable
- the `grep` command must support the -o flag (GNU does)
- the `patch` command must support the -E flag
- you may need to explicitly set ANT_HOME. Running `ant -diagnostics` will tell you the default value on your system.

Applying a patch

To apply a patch either you generated or found from JIRA, from the `trunk` directory you can issue

```
patch -p0 <cool_patch.patch
```

if you just want to check whether the patch applies you can run patch with --dry-run option

```
patch -p0 --dry-run <cool_patch.patch
```

If you are an Eclipse user, you can apply a patch by :

1. Right click project name in Package Explorer,
   #. Team -> Apply Patch

Contributing your work

Finally, patches should be attached to a bug report in Jira via the Attach File link on the jira. Please note that the attachment should be granted license to ASF for inclusion in ASF works (as per the Apache License subsection 5).

When you believe that your patch is ready to be committed, select the **Submit Patch** link on the issue's Jira. Submitted patches will be automatically tested against "trunk" by WWW Jenkins. Upon test completion, Jenkins will add a success ("+1") message or failure ("-1") to your bug report in Jira. If your issue contains multiple patch versions, Hudson tests the last patch uploaded. (Note: currently this is not working and developers are running `test-patch` manually. We hope to have this fixed soon.)

Folks should run 'ant clean test-commit javadoc' before selecting 'Submit Patch'. Tests should all pass. Javadoc should report no warnings or errors. Hudson's tests should only double-check things, and not be used as a primary patch tester, which would create too much noise on the mailing list and in Jira.

If your patch involves performance optimizations, they should be validated by benchmarks that demonstrate an improvement.

Once a "+1" comment is received from the automated patch testing system and a "+1, code reviewed" comment is received from a code reviewer, a committer should then evaluate it 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 other's suggestions into 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 earn a -1 on the Jenkins test, set the issue status to 'Resume Progress', upload a patch with necessary fixes and then set the status to 'Submit Patch' again.

Committers: for non-trivial changes, you must get another committer to review your patches before commit. Use "Submit Patch" 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 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 (on the right) 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.

Stay involved

Contributors should join the Pig mailing lists. In particular, the commit list (to see changes as they are made), the dev list (to join discussions of changes) and the user list (to help others).

See Also

- Apache contributor documentation
- Apache voting documentation

Picking Something to Work On

Looking for a place to start? A great first place is to peruse the JIRA and find an issue that needs resolved. Especially, here is a list of Jiras marked as "newbie". If you're looking for a bigger project, try the Pig Journal. This gives a list of projects the Pig team would like to see worked on.