HowToTest

How To Test Pig

This document covers how to test Pig. It is intended for Pig developers who need to know how to test their work. It can also be used by Pig users who wish to verify their instance of Pig.

Pig currently provides tools for two types of testing: unit testing and end-to-end testing.

Unit Testing

Unit tests are executed via JUnit. Currently, many "unit tests" are really end-to-end tests. We are in the process of changing this so that all of end-to-end tests will be run by the e2e harness (see below). See PigTestProposal for details.

Preparation

Prior to running unit tests, make sure to set `umask 0022`.

We also see unit tests fail due to extended acl, so use `setfacl -b` to remove extended acl if applicable.

Running all unit tests

To run the unit tests do `ant test` in the top level Pig directory. Currently this takes 8 hours to run. We intend to drive this to under five minutes. Until this is done it is not expected that contributors will run all these tests before submitting their patch.

Running pre-commit tests

You can also run `ant clean test-commit` to run true unit tests plus a few very basic end-to-end tests to assure nothing fundamental has been broken. We ask that committers run this before committing patches, and that contributors run it before uploading their patch.

Running a single unit test

A single unit test can be run by setting the `testcase` property. For example:

```
ant -Dtestcase=TestRegisteredJarVisibility clean test
```

Generating Code Coverage Numbers for Unit Tests

Pig is integrated with clover, which can be used to measure code coverage of the unit tests. First, you will need to obtain clover:

1. Download clover from Atlassian. You want Clover for Ant 2.6.3. The license from Apache doesn't work with the latest 3.x versions of Clover.
2. Unzip clover somewhere on the machine where you will be doing the tests. We'll call this location `$CLOVER_HOME`
3. Download the license file from Apache. You must be an Apache committer to access this file.
4. Place the license file in `$CLOVER_HOME/lib`

Now, to run the unit tests with clover:

```
ant clean
ant -Dclover.home=<clover_home> -Drun.clover=true clover jar test
ant -Dclover.home=<clover_home> -Drun.clover=true generate-clover-reports
ant -Dclover.home=<clover_home> -Drun.clover=true generate-pdf-clover-reports
```

The detailed report is now available as a collection of HTML files under `build/test/clover/reports`
A summary report is now available in `build/test/clover/pdf/reports/clover_coverage.pdf`

End-to-end Testing

End-to-end tests (e2e) are run with a custom test harness. For information on this harness see TestHarnessDesign. The goal of these tests is to test Pig on a real Hadoop cluster. These tests try to cover all functionality provided by Pig.

Currently running the entire suite of e2e tests takes about 10 hours. It is not expected that contributors committers will run all these tests before contributing or committing a patch. They should run the Checkin tests, plus any tests they have added, plus tests that cover the area of changes they are making. For example, if one was making a change to the merge join algorithm, he should run the Checkin tests plus the MergeJoin tests before uploading and checking in his patch. These tests should be run in both cluster and local mode.

How to Run e2e Tests

Running the e2e tests requires three things: a cluster to run them on, an old version of Pig to use to generate expected results, and perl (plus a few CPAN modules) on your client machine. The cluster can be quite small. A single machine will enough. Since performance is not the goal it is fine if this is a virtual machine. If you do not have access to a cluster, see below for information on how to run the tests on EC2.

You will need the following CPAN modules:

- IPC::Run
- Parallel::ForkManager
- DBI

For help installing CPAN modules, see cpan module install instructions .

Currently we are using Pig 0.8.1 for the old version of Pig that generates expected results. Once 0.9 is widely adopted we will likely switch to it as the source of truth. You can download Pig 0.8.1 from http://www.apache.org/dyn/closer.cgi/pig.

If you are running in a secure Hadoop environment (with kerberos in use), you must do kinit to obtain a ticket before running ant. Otherwise your tests will all fail to gain access to the cluster.

Before you can run the test harness against your cluster for the first time, you must generate the test data in your cluster. To do this, do:

```
ant -Dharness.old.pig=old_pig -Dharness.cluster.conf=hadoop_conf_dir -Dharness.cluster.bin=hadoop_script -
Dharness.hadoop.home=hadoop_home_dir test-e2e-deploy
```

Where `old_pig` is where you installed the old version of Pig, `hadoop_conf_dir` is the directory where your `hadoop-site.xml` or `mapred-site.xml` file is, and `hadoop_script` is where your `hadoop` executable is located. For example, if you have installed Pig 0.8.1 in `/usr/local/pig/pig-0.8.1` and Hadoop in `/usr/local/hadoop`, then your command line would look like:

```
ant -Dharness.old.pig=/usr/local/pig/pig-0.8.1 -Dharness.cluster.conf=/usr/local/hadoop/conf -Dharness.cluster.
bin=/usr/local/hadoop/bin/hadoop -Dharness.hadoop.home=hadoop_home_dir test-e2e-deploy
```

This takes a couple of minutes and only needs to be run once. After building Pig itself it will display information on the data it is generating.

Once you have loaded your cluster with data, you can run the tests by doing:

```
ant -Dharness.old.pig=old_pig -Dharness.cluster.conf=hadoop_conf_dir -Dharness.cluster.bin=hadoop_script -
Dharness.hadoop.home=hadoop_home_dir test-e2e
```

Run with `test-e2e-tez` instead of `test-e2e` to run tests with Tez as execution engine.

Running the full test suite is rarely what you want, as it takes around 10 hours. If you are running against a cluster with more capacity, you can speedup the execution of the tests by parallelizing it. The `fork.factor.conf.file` property tells how many test conf files to run in parallel. The `fork.factor.group` property tells how many groups to run in parallel within each test file. Within a group, each tests are run sequentially. For eg: `-Dfork.factor.conf.file=2 -Dfork.factor.group=5` will run 2 test files and 5 groups in each totaling 10 tests in parallel.

To run only some tests, set the `tests.to.run` property. This value can be passed a group of tests (e.g. Checkin), or a single test (e.g. Checkin_1). You can pass multiple tests or groups in this property. Each test or group of tests must be proceeded by a {{-t }}. For example, to run the Checkin tests and the first MergeJoin test, do:

```
ant -Dharness.old.pig=old_pig -Dharness.cluster.conf=hadoop_conf_dir -Dharness.cluster.bin=hadoop_script -
Dharness.hadoop.home=hadoop_home_dir -Dtests.to.run="-t Checkin -t MergeJoin_1" test-e2e
```

Status will be provided as each test is run. Tests either succeed, fail, or abort. A test fails when actual results do not match expected results. A test aborts when the test or expected results generation failed to execute. The harness prints out the path to the log file where details of the test run are provided.

If you want to clean the data off of your cluster, you can use the undeploy target:

```
ant -Dharness.old.pig=old_pig -Dharness.cluster.conf=hadoop_conf_dir -Dharness.cluster.bin=hadoop_script -
Dharness.hadoop.home=hadoop_home_dir test-e2e-undeploy
```

There is no need to do this on a regular basis.

If you want to generate a junit format xml file out of the e2e test log and use it for displaying test results in Jenkins, you can run test/e2e/harness/xmlReport.pl against the log file.

```
test/e2e/harness/xmlReport.pl testdist/out/log/test_harnesss_1411157020 > test-report.xml
```

Running e2e in Local Mode

End-to-end tests can also be run in local mode. While this is not a substitute for running them on a cluster, it is a good idea to run your tests in both cluster and local mode to assure both modes work. Running in local mode is nearly identical to running cluster mode, except that the test data in local mode is placed in a directory under the test harness. So you must generate the local mode data each time you are working in a different source tree or after doing an `ant clean`.

To generate the test data in local mode, do:

```
ant -Dharness.old.pig=old_pig -Dharness.cluster.conf=hadoop_conf_dir -Dharness.cluster.bin=hadoop_script -
Dharness.hadoop.home=hadoop_home_dir test-e2e-deploy-local
```

(Yes you still have to give cluster information even though you aren't using a cluster. Pig doesn't use it in this case and you can pass bogus info if you want.)

To run the local mode tests themselves, do:

```
ant -Dharness.old.pig=old_pig -Dharness.cluster.conf=hadoop_conf_dir -Dharness.cluster.bin=hadoop_script -
Dharness.hadoop.home=hadoop_home_dir test-e2e-local
```

Running on EC2

If you do not have access to a cluster you can run the tests in EC2. In the directory `tests/e2e/pig/whirr` there are tools that will help you run on EC2.

In the following text any value that starts `your_` is a value you should fill in.

Prerequisites:

1. An account in Amazon's AWS
2. An Amazon Access Key ID and Secret Access Key. These are not ssh keys. See http://aws-portal.amazon.com/gp/aws/developer/account/index.html?action=access-key under Access Credentials. You need an Access Key.
3. An RSA SSH key pair that is passphraseless. You may want to generate a pair just for use with the tool to avoid forcing your regular ssh key pair to be passphraseless. They must be RSA; Whirr does not work with any of the other key types. You can generate a pair with the command `ssh-keygen -f your_private_rsa_key_file -t rsa -P ''` where `your_private_rsa_key_file` is the file to store the private key in.
4. Apache Whirr version 0.5 or newer.

To Start a Cluster:

```
export AWS_ACCESS_KEY_ID=your_amazon_access_key
export AWS_SECRET_ACCESS_KEY=your_secret_amazon_access_key
export SSH_PRIVATE_KEY_FILE=your_private_rsa_key_file
cd your_path_to_apache_whirr/bin
./whirr launch-cluster --config your_path_to_pig_trunk/test/e2e/pig/whirr/pigtest.properties
```

This will take ~5 minutes and spew various messages on your screen.

DO NOT FORGET TO SHUTDOWN YOUR CLUSTER (see below) (unless you think Amazon a worthy cause and wish to donate your extra cash to them).

Running the tests:
Open the file `~/.whirr/pigtest/hadoop-site.xml` and find the line that has `mapred.job.tracker`. The next line should have the hostname that is running your Job Tracker. Copy that host name, but NOT the port numbers (ie the `:nnnn` where `nnnn` is `9001` or something similar). This value will be referred to as `your_namenode`.

```
cd your_path_to_pig_src
scp -i your_private_rsa_key_file test/e2e/pig/whirr/whirr_test_patch.sh your_namenode:~

if you have a patch you want to run
    scp -i your_private_rsa_key_file your_patch your_namenode:~

ssh -i your_private_rsa_key_file your_namenode
```

Now you can run `whirr_test_patch` to run some or all of the tests against trunk or against your patch. To run all the tests against trunk, do `./whirr_test_patch.sh`

To apply your patch and then run the tests, do `./whirr_test_patch.sh -p your_patch`

To run just some of the tests, do `./whirr_test_patch.sh -t test_group_or_name` where `test_group_or_name` is a test or group of tests you want to run. Multiple -t options can be passed.

`whirr_test_patch` is not idempotent. It downloads necessary packages, checks out trunk, applies your patch if appropriate, and generates the test data and loads into your cluster. Once you have successfully run it once, you should not run it again. If you wish to do additional testing `cd src/trunk` and run the end-to-end tests via ant as you normally would.

Initial setup takes around 5 minutes. Running all of the nightly tests currently (August 2011) takes about 10 hours. When you are just testing a patch for submission your are not expected to run the full suite of tests. Checkin, plus any tests you've added, plus all that cover the area of your change is sufficient.

Shutting down your cluster:
In the same shell you started the cluster:

```
./whirr destroy-cluster --config your_path_to_pig_trunk/test/e2e/pig/whirr/pigtest.properties
```

How to Write an e2e Test

Writing a new e2e test does not require writing any new Java code (assuming you don't need to write a UDF for your job). The e2e test harness is written in Perl, and the tests are stored in .conf files, each of which is one big Perl hash (if you squint just right, it almost looks like JSON). These files are in `test/e2e/pig/tests/`. This hash is expected to have a `groups` key, which is an array. Each element in the array describes a collection of tests, usually oriented around a particular feature. For example the group `FilterBoolean` tests boolean predicates in filters. Every group in the array is a hash. It must have `name` and a `tests` keys. `tests` is expected to be an array of tests. Each test is again a hash, and must have a `num`, the test number and `pig`, the Pig Latin code to run. As an example look at the following, taken from `nightly.conf`:

```
$cfg = {
    'driver' => 'Pig',

    'groups' => [
        {
            'name' => 'Checkin',
            'tests' => [
                {
                    'num' => 1,
                    'pig' => q\a = load ':INPATH:/singlefile/studenttab10k' as (name, age, gpa);
                              store a into ':OUTPATH:';\,
                },
                {
                    'num' => 2,
                    'pig' => q\a = load ':INPATH:/singlefile/studenttab10k' as (name, age, gpa);
                              b = load ':INPATH:/singlefile/votertab10k' as (name, age, registration,
contributions);
                              c = filter a by age < 50;
                              d = filter b by age < 50;
                              e = cogroup c by (name, age), d by (name, age) ;
                              f = foreach e generate flatten(c), flatten(d);
                              g = group f by registration;
                              h = foreach g generate group, SUM(f.d::contributions);
                              i = order h by $1;
                              store i into ':OUTPATH:';\,
                    'sortArgs' => ['-t', '  ', '+1', '-2'],
                }
            ]
        },
        {
            'name' => 'LoaderPigStorageArg',
            'tests' => [
                {
                    'num' => 1,
                    'pig' => q\a = load ':INPATH:/singlefile/studentcolon10k' using PigStorage(':') as (name,
age, gpa);
                              store a into ':OUTPATH:';\,
                },
            ]
        }
    ]
};
```

This has two groups `Checkin` and `LoaderPigStorageArg`. `Checkin` has two tests.

In these simple cases the test harness runs the specified Pig Latin and generates a result. It then runs the same script against the version of Pig you specified as old (we are currently using Pig 0.8.1 in our nightly tests) and generates an expected result. These two results are then sorted and an md5 checksum taken. If this matches, the test is declared to have succeeded, otherwise, it failed.

For tests where sort order is important (as in `Checkin_2` above), you can check that data is sorted using the Unix `sort` utility. This is done by passing an array of the arguments for `sort`. `sort` is then invoked before the harness sorts the data for comparison to the expected results to see if the data is sorted as specified. The `sortArgs` given above says to use tab as a delimiter and check that the output is sorted on the second (or first if you count from zero) field. See `sort`'s man page for details.

For features that are new, you cannot test against old versions of Pig. For example, macros in 0.9 cannot be tested against 0.8.1. As an alternate to running the same Pig Latin script against an old version, you can run a different script. This script will be run using the current version, not the old one. To specify a different script, you need a key `verify_pig_script`. For example:

```
      {
         # simple macro, no args
         'num' => 1,
         'pig' => q#define simple_macro() returns void {
                       a = load ':INPATH:/singlefile/studenttab10k' as (name, age, gpa);
                       b = foreach a generate age, name;
                       store b into ':OUTPATH:';
                    }

                    simple_macro();#,
         'verify_pig_script' => q#a = load ':INPATH:/singlefile/studenttab10k' as (name, age, gpa);
                                  b = foreach a generate age, name;
                                  store b into ':OUTPATH:';#,
       }
```

Some tests need to check that Pig outputs correct errors messages, returns correct codes, etc. These tests can define `expected_err_regex`, `execpted_out_regex`, `not_expected_out_regex`, `not_expected_err_regex` or `rc`. The first four search `stderr` or `stdout` for the provided regular expression (or lack thereof in the not cases). The last checks the return code against the provided value. Success or failure of the test is determined by whether these regular expressions or returns codes match. When these tags are present no expected results are generated via an old version of Pig or an alternate Pig Latin script.

An exhaustive list of keys supported in the test hash:

Key	What it Does	Example	Required?
delimiter	Provides `floatpostprocess` with delimiter to use	`'delimiter' => ':'`	Only with `flostpostprocess`
execonly	This test will only be executed in specified mode; options are `local` and `mapred`	`'execonly' => 'mapred'`	No
expected_err_regex	Checks stderr error for the provided regular expression	`'expected_err_regex' => "Out of bound access."`	No
expected_out_regex	Checks stdout error for the provided regular expression	`'expected_out_regex' => "A: {name: bytearray,age: bytearray,gpa: bytearray}"`	No
floatpostprocess	Run floating point numbers through a post processor, since due to precision issues different runs of the same script will produce slightly different values. All floating point numbers are rounded to 3 decimal places. This must be used in conjunction with `delimiter`	`'floatpostprocess' => 1`	For outputs that include calculated floating point values.
ignore	Do not run this test, used when a test is failing but we don't want to remove it because it will be needed once the issue is fixed. A reason for ignoring the test should be given.	`'ignore' => 'JIRA-19999'`	No
java_params	Values to be passed on the `pig` command line before other Pig parameters; useful for passing properties.	`'java_params' => ['-Dpig.cachedbag.memusage=0']`	No
not_expected_err_regex	Checks that stderr does not match the provided regular expression	`'not_expected_err_regex' => "ERROR"`	No
not_expected_out_regex	Checks that stdout does not match the provided regular expression	`'not_expected_out_regex' => "datafile"`	No
notmq	Tells the test harness this is not a multi-query test; only necessary when a test has multiple `store` operators but should not be verified as if it were multi-query.	`'notmq' => 1`	No
num	Test number; must be unique in the test group	`'num' => 1`	Yes
pig	The Pig Latin script to run in the test	`q#a = load ':INPATH:/dir /studenttab10k' as (name, age, gpa); store a into ':OUTPATH:';#`	Yes
pig_params	Command line arguments to pass to `pig` when running this test.	`'pig_params' => ['-p', qq (fname='studenttab10k')]`	No
rc	Expected return code	`'rc' => 0`	No
sortArgs	Arguments to pass to the Unix `sort` utility. When these are given, sort will be called before data is sorted for comparison with the expected results.	`'sortArgs' => ['-t', ' ', '+0', '-1']`	Only when job output should be sorted

verify_pig_script	Alternate Pig Latin script to use to generate the expected results	`'verify_pig_script' => q\A = load ':INPATH:/singlefile/studenttab10k' as (name, age, gpa); store A into ':OUTPATH:';\,`	No

Almost all test Pig Latin scripts need to make references to file paths. Rather than hardwire these paths in the tests, variables are provided. Whenever referencing paths in your scripts you should use these variables. The following table describes the most commonly used variables:

Variable	Used For	Example
:FUNCPATH:	Location of test UDF jars constructed by the test harness	`register :FUNCPATH:/testudf.jar;`
:HADOOPHOME:	Location where MapReduce jars used for testing the `mapreduce` operator are kept	`b = mapreduce ':HADOOPHOME:/hadoop-examples.jar'`
:INPATH:	The path where Pig test data is stored	`a = load ':INPATH:/singlefile/studenttab10k'`
:OUTPATH:	The path where Pig will write the results of a test	`store a into ':OUTPATH:';`
:SCRIPTHOMEPATH:	Location external script modules (e.g. Python) are kept	`register ':SCRIPTHOMEPATH:/python/scriptingudf.py' using jython as myfuncs;`
:TMP:	Temporary directory specific to a given test, can be used to store temporary files	`pig_script = ":TMP:/script.pig"`

The following files contain tests, and are located in `test/e2e/pig/nightly`:

conf File	Tests	Comments
bigdata.conf	larger size data	We keep these to a minimum as they take much longer than other tests.
cmdline.conf	Pig command line output (such as describe)	
grunt.conf	grunt operators, like `ls`	
macro.conf	macro and import	
multiquery.conf	multiquery scripts	
negative.conf	negative conditions where Pig is expected to return an error	
nightly.conf	general positive tests	Your test goes here if it doesn't fit anywhere else
orc.conf	OrcStorage tests	
streaming.conf	streaming feature	
streaming_local.conf	streaming feature in local mode	
turing_jython.conf	Pig scripts embedded in Python scripts	

When writing new tests, if a group already exists that tests the functionality you are testing, add your tests to that group. There is a `Regression` group for tests that check bug fixes. If you are writing a new feature or writing tests for a previously un-tested feature, create a new group. When creating new groups, try to place it in an existing conf file that contains related tests. If you are creating a feature that will require many tests and multiple groups of tests, such as support for embedding Pig in a new language, create a new conf file for those tests.