# How to load, run, and create new Impala tests

## Loading test data

```
./buildall.sh -noclean -testdata
```

Note: Sometimes your may get a messed up terminal after data loading, e.g. not displaying any typed in characters or carriage return. In this case, type ctrl +c, then type `stty sane`, then press `enter` to recover.

## Run all tests:

```
MAX_PYTEST_FAILURES=12345678 ./bin/run-all-tests.sh
```

## Run just front-end tests

**Debugging**

See Debugging front-end test if you need to look into a test behavior such as unexpected error, hanging and so on.

```
# For the front-end tests, the minicluster and Impala cluster must be running:
./testdata/bin/run-all.sh
./bin/start-impala-cluster.py

(pushd fe && mvn -fae test)

# To run one group of tests:
(pushd fe && mvn -fae test -Dtest=AnalyzeStmtsTest)

# To run a single test:
(pushd fe && mvn -fae test -Dtest=AnalyzeStmtsTest#TestStar)


# To run a specific parameterized test:
(pushd fe && mvn test -Dtest=AuthorizationStmtTest#testPrivilegeRequests[0])
# or
(pushd fe && mvn test -Dtest=AuthorizationStmtTest#testPrivilegeRequests[*])
```

## Run just back-end tests

```
 # For the back-end tests, the Impala cluster must not be running:
./bin/start-impala-cluster.py --kill

./bin/run-backend-tests.sh
# or
ctest

# To run just one test (and show what broke) (make sure to source ./bin/set-classpath.sh first):
ctest --output-on-failure -R expr-test

# You can also run just a single test at a time. If this fails, be sure
# that you've run set-classpath.sh before-hand.
# You can then use command line parameters to run a subset of tests:
# To get help with parameters:
be/build/latest/runtime/mem-pool-test --help

# To run just one test within a test:
be/build/latest/runtime/mem-pool-test --gtest_filter=MemPoolTest.TryAllocateAligned
```

## Run most end-to-end tests (but not custom cluster tests)

```
# For the end-to-end tests, Impala must be running:
./bin/start-impala-cluster.py

./tests/run-tests.py

# To run the tests in just one directory:
./tests/run-tests.py metadata

# To run the tests in that directory with a matching name:
./tests/run-tests.py metadata -k test_partition_metadata_compatibility

# To run the tests in a file with a matching name:
./tests/run-tests.py query_test/test_insert.py -k test_limit

# To run the tests that don't match a name:
./tests/run-tests.py query_test/test_insert.py -k "-test_limit".

# To run the tests with more restrictive name-matching
./bin/impala-py.test tests/shell/test_shell_commandline.py::TestImpalaShell::test_multiple_queries

# To run tests only against a specific file format(s):
./tests/run-tests.py --table_formats=seq/snap/block,text/none

# To change the impalad instance to connect to (default is localhost:21000):
./tests/run-tests.py --impalad=hostname:port

# To use a different exploration strategy (default is core):
./tests/run-tests.py --exploration_strategy=exhaustive

# To run a particular .test file, e.g. testdata/workloads/functional-query/queries/QueryTest/exprs.test
# search for a .py test file which refers to the test, (try 'grep -r QueryTest/exprs tests')
# then run that test file (in this case tests/query_test/test_exprs.py) using run-tests.py
./tests/run-tests.py query_test/test_exprs.py

# To update the results of tests (The new test files will be located in ${IMPALA_EE_TEST_LOGS_DIR}
/test_file_name.test):
./tests/run-tests.py --update_results
```

## Run custom cluster tests

```
# To run all custom cluster tests:
./tests/run-custom-cluster-tests.sh

# To run specific custom cluster tests:
impala-py.test tests/custom_cluster/test_alloc_fail.py
```

## Run stress test

```
# Run 10000 TPC-H queries on your 3-daemon minicluster against the tpch_parquet database with
# memory overcommitted 300%. Tolerate up to 100 query failures in a row. Assumes that a minicluster
# has been started with 3 impalads (the default).
./tests/stress/concurrent_select.py --minicluster-num-impalads 3 --max-queries=10000 --tpch-db=tpch_parquet --
mem-overcommit-pct=300 --fail-upon-successive-errors 100

# For more options:
./tests/stress/concurrent_select.py --help
```

## Things to look for in Test Failures

1. Look for Minidump crashes. These look like: ./Impala/logs_static/logs/ee_tests/minidumps/impalad/71bae77a-07d2-4b6e-0885eca5-adc7ee36.dmp. Because some tests run in parallel (especially in the "ee" test suite), if one of the Impala daemons crashed, subsequent test results are meaningless. Start with the crash. If you see a crash dump, it's possible that there's a stack trace in the logs:

   ```
   # ag is a recursive grep tool; see https://github.com/ggreer/the_silver_searcher
   $ag 'Check failure stack trace:'
   Impala/logs_static/logs/ee_tests/impalad.ip-172-31-24-168.ubuntu.log.ERROR.20171023-180534.38031
   101743:*** Check failure stack trace: ***
   101746:*** Check failure stack trace: ***

   Impala/logs_static/logs/custom_cluster_tests/impalad.ip-172-31-24-168.ubuntu.log.ERROR.20171023-193700.44118
   12:*** Check failure stack trace: ***

   Impala/logs_static/logs/custom_cluster_tests/impalad.ip-172-31-24-168.ubuntu.log.ERROR.20171023-193658.43993
   12:*** Check failure stack trace: ***

   Impala/logs_static/logs/custom_cluster_tests/impalad.ip-172-31-24-168.ubuntu.log.ERROR.20171023-193659.44062
   12:*** Check failure stack trace: ***
   ```

2. Look for "FAILED" (case insensitive) in the log output of the Jenkins job:

   ```
   $cat console | egrep -o -i 'FAILED +[^ ]+ +[^ ]+'
   FAILED query_test/test_udfs.py::TestUdfExecution::test_ir_functions[exec_option:
   {'disable_codegen_rows_threshold':
   FAILED query_test/test_spilling.py::TestSpillingDebugActionDimensions::test_spilling_large_rows
   [exec_option: {'debug_action':
   FAILED query_test/test_udfs.py::TestUdfExecution::test_native_functions[exec_option:
   {'disable_codegen_rows_threshold':
   failed to connect:
   failed in 12443.43
   failed in 4077.50
   ```

3. The tests produce xunit XML files; you can look for failures in these like so:

   ```
   # Search for and extract 'failure message="..."' in all XML files; uses
   # a multiline RE.
   $grep -Pzir -o 'failure message=\"[^"]*' $(find . -name '*.xml') | tr '\0' ' ' | head
   ./Impala/logs_static/logs/ee_tests/results/TEST-impala-parallel.xml:failure message="query_test
   /test_nested_types.py:70: in test_subplan
    self.run_test_case(&apos;QueryTest/nested-types-subplan&apos;, vector)
   common/impala_test_suite.py:391: in run_test_case
    result = self.__execute_query(target_impalad_client, query, user=user)
   common/impala_test_suite.py:600: in __execute_query
    return impalad_client.execute(query, user=user)
   common/impala_connection.py:160: in execute
    return self.__beeswax_client.execute(sql_stmt, user=user)
   beeswax/impala_beeswax.py:173: in execute
    handle = self.__execute_query(query_string.strip(), user=user)
   ```

## New tests should be created in areas similar to existing tests.

In general, try to keep similar tests with similar scope together.

- Any changes to the sql-parser.cup should have corresponding tests created in ParserTest.java
  - ParsesOk() and ParserError() are the two validation methods used for tests.
- Any changes in the analysis phase (e.g. under the package org.apache.impala.analysis) should have corresponding tests created in Analyze*Test.java (e.g. AnalyzeDDLTest.java, AnalyzeSubqueriesTests.java,
  - AnalyzesOk() and AnalysisError() are the main two validation methods used for tests but there are other utility methods available.
- Any changes that would involve authorization checks involving Sentry should have corresponding tests created in AuthorizationTests.java
  - AuthzOk() and AuthzError() are the two validation methods used for tests.
  - Note: These test are different than ones in AnalyzeAuthStmtsTest.java.  In AnalyzeAithStmtsTest.java, tests are validating the analysis phase of SQL statements related to authorization, where as AuthorizationTests.java uses Sentry to validate the authorization code granted by specific roles.
  - The comments at the top of this file indicate the various privileges that are granted on different objects.  Any changes or additions to the setup should be reflected here to maintain a single source of truth for the authorization tests.
  - These are currently the only set of tests that utilize Sentry.
- Any changes that require multiple SQL statements to execute, or require validation of the result should be created in the appropriate *.test file under testdata/workloads/...
  - .test files should contain blocks that contain only one of any section, e.g. "----QUERY", "----RESULTS", "----TYPES", etc.  Look at the other files for examples.

## What is the difference between impala-py.test and test/run-tests.py?

py.test (https://docs.pytest.org/en/latest/) is a framework for writing tests in Python. It's got a bunch of features.

impala-py.test is a small wrapper that invokes py.test in the right environment (i.e., from the "virtualenv") and with LD_LIBRARY_PATH updated. When you're using impala-py.test, you're using py.test's mechanisms.

Meanwhile, test/run-tests.py is a wrapper that invokes py.test a few times. It uses environment variables for control. Then it executes any "serial" tests, and then it executes some stress tests, and then, it instructs py.test to execute the "parallel" tests (those not marked serial or stress) with a bunch of parallelism. If you plotted your CPU graph while this was happening, you'd see something like. You can tell where the parallel testing happens about 3/4 of the way through.

To make things even more complicated, there's bin/run-all-tests.sh, which invokes run-tests.py (but also "mvn test" and the C++ tests), and buildall.sh, which invokes run-all-tests.sh.