Contribution Testing Guide

- Beam Testing
  - Testing Scenarios
    - Pre-commit
    - Post-commit
  - Testing Types
    - Unit
      - How to run Python unit tests
      - How to run Java NeedsRunner tests
      - ValidatesRunner
      - E2E
    - E2E Testing Framework
      - ValidatesRunner Tests
      - Effective use of the TestPipeline JUnit rule
      - API Surface testing
      - Tests of Core Apache Beam Operations
        - What are they?
        - Running instructions
          - Running with Dataflow runner
          - Running with Portable Flink Runner
          - Jenkins jobs
          - Dashboards
    - Best practices for writing tests
      - Aim for one failure path
      - Avoid non-deterministic code
      - Use descriptive test names
      - Use a pre-commit test if possible

Beam Testing

This document outlines how to write tests, which tests are appropriate where, and when tests are run, with some additional information about the testing systems at the bottom.

Testing Scenarios

Ideally, all available tests should be run against a pull request (PR) before it’s allowed to be committed to Beam’s Github repo. This is not possible, however, due to a combination of time and resource constraints. Running all tests for each PR would take hours or even days using available resources, which would slow down development considerably.

Thus tests are split into pre-commit and post-commit suites. Pre-commit is fast, while post-commit is comprehensive. As their names imply, pre-commit tests are run on each PR before it is committed, while post-commit runs periodically against the master branch (i.e. on already committed PRs).

Beam uses Jenkins to run pre-commit and post-commit tests.

Pre-commit

The pre-commit test suite verifies correctness via two testing tools: unit tests and end-to-end (E2E) tests. Unit tests ensure correctness at a basic level, while WordCount E2E tests are run against each supported SDK / runner combination as a smoke test, to verify that a basic level of functionality exists.

This combination of tests hits the appropriate tradeoff between a desire for short (ideally <30m) pre-commit times and a desire to verify that PRs going into Beam function in the way in which they are intended.

Pre-commit jobs are kicked off when a contributor makes a PR against the apache/beam repository. Job statuses are displayed at the bottom of the PR page. Clicking on “Details” will open the status page in the selected tool; there, you can view test status and output.

Post-commit

Running in post-commit removes as stringent of a time constraint, which gives us the ability to do some more comprehensive testing. In post-commit we have a test suite running the ValidatesRunner tests against each supported runner, and another for running the full set of E2E tests against each runner. Currently-supported runners are Dataflow, Flink, Spark, and Gearpump, with others soon to follow. Work is ongoing to enable Flink, Spark, and Gearpump in the E2E framework, with full support targeted for end of August 2016. Post-commit tests run periodically, with timing defined in their Jenkins configurations.

Adding new post-commit E2E tests is generally as easy as adding a *IT.java file to the repository - Failsafe will notice it and run it - but if you want to do more interesting things, take a look at WordCountIT.java.

Post-commit test results can be found in Jenkins.

Testing Types
Unit

Unit tests are, in Beam as everywhere else, the first line of defense in ensuring software correctness. As all of the contributors to Beam understand the importance of testing, Beam has a robust set of unit tests, as well as testing coverage measurement tools, which protect the codebase from simple to moderate breakages. Beam Java unit tests are written in JUnit.

How to run Python unit tests

Python tests are written using the standard Python unittest library. To run all unit tests, execute the following command in the `sdks/python` subdirectory:

```
$ python setup.py test [-s apache_beam.package.module.TestClass.test_method]
```

We also provide a tox configuration in that same directory to run all the tests, including lint, cleanly in all desired configurations.

How to run Java NeedsRunner tests

NeedsRunner is a category of tests that require a Beam runner. To run NeedsRunner tests:

```
$ ./gradlew runners:direct-java:needsRunnerTests
```

To run a single NeedsRunner test use the `test` property, e.g.

```
```

will run the `MapElementsTest.testMapBasicProcessFunction()` test.

NeedsRunner tests in modules that are not required to build runners (e.g. `sdks/java/io/google-cloud-platform`) can be executed with the gradle command:

```
$ ./gradlew sdks:java:io:google-cloud-platform:test --tests org.apache.beam.sdk.io.gcp.spanner.SpannerIOWriteTest
```

ValidatesRunner

ValidatesRunner tests contain components of both component and end-to-end tests. They fulfill the typical purpose of a component test - they are meant to test a well-scoped piece of Beam functionality or the interactions between two such pieces and can be run in a component-test-type fashion against the DirectRunner. Additionally, they are built with the ability to run in an end-to-end fashion against a runner, allowing them to verify not only core Beam functionality, but runner functionality as well. They are more lightweight than a traditional end-to-end test and, because of their well-scoped nature, provide good signal as to what exactly is working or broken against a particular runner.

E2E

End-to-End tests are meant to verify at the very highest level that the Beam codebase is working as intended. Because they are implemented as a thin wrapper around existing pipelines, they can be used to prove that the core Beam functionality is available. They will be used to verify runner correctness, but they can also be used for IO connectors and other core functionality.

Testing Systems

E2E Testing Framework

The Beam end-to-end testing framework is a framework designed in a runner-agnostic fashion to exercise the entire lifecycle of a Beam pipeline. We run a pipeline as a user would and allow it to run to completion in the same way, verifying after completion that it behaved how we expected. Using pipelines from the Beam examples, or custom-built pipelines, the framework will provide hooks during several pipeline lifecycle events, e.g., pipeline creation, pipeline success, and pipeline failure, to allow verification of pipeline state.

The E2E testing framework is currently built to execute the tests on Jenkins, invoked via Gradle tasks. Once it is determined how Python and other future languages will integrate into the overall build/test system (via Gradle or otherwise) we will adjust this. The framework provides a wrapper around actual Beam pipelines, enabling those pipelines to be run in an environment which facilitates verification of pipeline results and details.

Verifiers include:
• Output verification. Output verifiers ensure that the pipeline has produced the expected output. Current verifiers check text-based output, but future verifiers could support other output such as BigQuery and Datastore.
• Aggregator verification. Aggregator verifiers ensure that the user-defined aggregators present in the pipelines under test finish in the expected state.

The E2E framework will support running on various different configurations of environments. We currently provide the ability to run against the DirectRunner, against a local Spark instance, a local Flink instance, and against the Google Cloud Dataflow service.

ValidatesRunner Tests

ValidatesRunner tests are tests built to use the Beam TestPipeline class, which enables test authors to write simple functionality verification. They are meant to use some of the built-in utilities of the SDK, namely PAssert, to verify that the simple pipelines they run end in the correct state.

Effective use of the TestPipeline JUnit rule

TestPipeline is JUnit rule designed to facilitate testing pipelines. In combination with PAssert, the two can be used for testing and writing assertions over pipelines. However, in order for these assertions to be effective, the constructed pipeline must be run by a pipeline runner. If the pipeline is not run (i.e., executed) then the constructed PAssert statements will not be triggered, and will thus be ineffective.

To prevent such cases, TestPipeline has some protection mechanisms in place.

Abandoned node detection (performed automatically)

Abandoned nodes are PTransforms, PAsserts included, that were not executed by the pipeline runner. Abandoned nodes are most likely to occur due to the one of the following scenarios:

1. Lack of a pipeline.run() statement at the end of a test.
2. Addition of PTransforms after the pipeline has already run.

Abandoned node detection is automatically enabled when a real pipeline runner (i.e. not a CrashingRunner) and/or a @NeedsRunner / @ValidatesRunner annotation are detected.

Consider the following test:

Java

```java
// Note the @Rule annotation here
@Rule
public final transient TestPipeline pipeline = TestPipeline.create();

@Test
@Category(NeedsRunner.class)
public void myPipelineTest() throws Exception {
    final PCollection<String> pCollection =
        pipeline
            .apply("Create", Create.of(WORDS).withCoder(StringUtf8Coder.of()))
            .apply("Map1",
                MapElements.via(
                    new SimpleFunction<String, String>() {
                        @Override
                        public String apply(final String input) {
                            return WHATEVER;
                        }
                    }));

    PAssert.that(pCollection).containsInAnyOrder(WHATEVER);
    /* ERROR: pipeline.run() is missing, PAsserts are ineffective */
}
```

Python

```python
# The suggested pattern of using pipelines as targets of with statements
# eliminates the possibility for this kind of error or the need for a
# framework to catch it.
```
with beam.Pipeline(...) as p:
    [...arbitrary construction...]
    # p.run() is automatically called on successfully exiting the context

The PAssert at the end of this test method will not be executed, since pipeline is never run, making this test ineffective. If this test method is run using an actual pipeline runner, an exception will be thrown indicating that there was no run() invocation in the test.

Exceptions that are thrown prior to executing a pipeline, will fail the test unless handled by an ExpectedException rule.

Consider the following test:

Java

```java
// Note the @Rule annotation here
@Rule
public final transient TestPipeline pipeline = TestPipeline.create();

@Test
public void testReadingFailsTableDoesNotExist() throws Exception {
    final String table = "TEST-TABLE";
    BigtableIO.Read read =
        BigtableIO.read()
            .withBigtableOptions(BIGTABLE_OPTIONS)
            .withTableId(table)
            .withBigtableService(service);
    // Exception will be thrown by read.validate() when read is applied.
    thrown.expect(IllegalArgumentException.class);
    thrown.expectMessage(String.format("Table %s does not exist", table));
    p.apply(read);
}
```

Python

```python
# Use standard Python tools such as unittest's TestCase.assertRaises

The application of the read transform throws an exception, which is then handled by the thrown ExpectedException rule. In light of this exception, the fact this test has abandoned nodes (the read transform) does not play a role since the test fails before the pipeline would have been executed (had there been a run() statement).

Auto-add pipeline.run() (disabled by default)

A TestPipeline instance can be configured to auto-add a missing run() statement by setting testPipeline.enableAutoRunIfMissing(true/false). If this feature is enabled, no exception will be thrown in case of a missing run() statement, instead, one will be added automatically.

API Surface testing

The surface of an API is the set of public classes that are exposed to the outer world. In order to keep the API tight and avoid unnecessarily exposing classes, Beam provides the ApiSurface utility class. Using the ApiSurface class, we can assert the API surface against an expected set of classes.

Consider the following snippet:

Java

```java
@Test
public void testMyApiSurface() throws Exception {
    final Package thisPackage = getClass().getPackage();
    final ClassLoader thisClassLoader = getClass().getClassLoader();
    final ApiSurface apiSurface =
        ApiSurface.ofPackage(thisPackage, thisClassLoader)
            .pruningPattern("org[].apache[].beam[].*Test.*")
            .pruningPattern("org[].apache[].beam[].*IT")
            .pruningPattern("java[].lang.");
```
final Set<Matcher<Class<?>>> allowed =
    ImmutableSet.of(
    classesInPackage("org.apache.beam.x"),
    classesInPackage("org.apache.beam.y"),
    classesInPackage("org.apache.beam.z"),
    Matchers.<Class<?>>equalTo(Other.class));

assertThat(apiSurface, containsOnlyClassesMatching(allowed));

# Python doesn't have the notion of public vs. private members or type declarations that may accidentally leak.

This test will fail if the classes exposed by getClass().getPackage(), except classes which reside under "org\.apache\.beam\..*Test\.*", "org\.apache\.beam\..*IT", or "java\.lang\..*", belong to neither of the packages: org\.apache\.beam\.x, org\.apache\.beam\.y, org\.apache\.beam\.z, nor equal to Other.class.

Tests of Core Apache Beam Operations

What are they?

Tests of Core Apache Beam Operations are a set of tests aiming to exercise Core Beam transforms to see how do they behave in stressful conditions. They operate on synthetic data of KV<byte[], byte[]> type that is generated deterministically and can be shaped with different distributions, generation delays and the size of records itself.

For example, thanks to the tests users can check things like:

- Performance impact of hotkeys in input data,
- Record size impact,
- State cache capacity,
- Inter operation overhead,
- Influence of extensive metrics API usage

...and many more.

For a more detailed description of tests and testing scenarios, see the initial proposal: https://s.apache.org/load-test-basic-operations

Running instructions

Load test specific parameters:

<table>
<thead>
<tr>
<th>Name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>loadTest.mainClass</td>
<td>The fully qualified name of the testing class</td>
</tr>
<tr>
<td>runner</td>
<td>Runner to be used</td>
</tr>
<tr>
<td>loadTest.args</td>
<td>Pipeline options for the test</td>
</tr>
</tbody>
</table>

Running with Dataflow runner

Java:

```
./gradlew :sdks:java:testing:load-tests:run \
-PlLoadTest.mainClass=org.apache.beam.sdk.loadtests.GroupByKeyLoadTest \
-Prunner=runners:google-cloud-dataflow-java \
-PlLoadTest.args=-project=apache-beam-testing --appName=load_tests_Java_Dataflow_batch_GBK_1 --tempLocation=gs://temp-storage-for-perf-tests/loadtests --publishToBigQuery=true --bigQueryDataset=load_test --bigQueryTable=java_dataflow_batch_GBK_1 --sourceOptions="{"numRecords":200000000,"keySizeBytes":1,"valueSizeBytes":9}" --fanout=1 --iterations=1 --streaming=false --runner=DataflowRunner'
```

Running with Portable Flink Runner

As Flink is not a managed service like Dataflow, you need to setup the cluster before you start the test. We prepared scripts to do that for you using Google Cloud Dataproc. You can see instructions on how to set up a full-blown Flink cluster in flink_cluster.sh script.
After you set up the cluster, run the tests with the following command:

```bash
./gradlew --publish_to_big_query=true --project=apache-beam-testing --metrics_dataset=load_test --metrics_table=python_flink_batch_GBK_1 --parallelism=5 --job_endpoint=localhost:8099 --environment_config=gcr.io/apache-beam-testing/beam_portability/python2.7_sdk:latest --environment_type=DOCKER --iterations=1 --fanout=1 --input_options='{"num_records": 20000000, "key_size": 10, "value_size": 90}';
```

For the purpose of local testing, you can also run the test on an embedded Flink cluster. In this case, you don't need to build and push SDK harness image and setup Flink cluster using Dataproc.

Before the first run, build a job server:

```bash
./gradlew :runners:flink:1.9:job-server:shadowJar
```

Jenkins jobs

There are several Jenkins jobs that are configured to run every 24h. Other than that it is possible to trigger them on demand using Github Pull Request Build Plugin (ghprb).

You can find all definitions of the jobs in .test-infra/jenkins directory. Other than that, all the jobs are listed in README.md file.

Dashboards

The load tests are run daily by a set of Jenkins jobs. To be able to assess the current performance of the operations and detect regressions, we created a set of dashboards for the tests.

Link for the dashboards: https://apache-beam-testing.appspot.com/dashboard-admin (please click “all dashboards” to see the list of currently available dashboards, including the ones for load-tests).

Best practices for writing tests

The following best practices help you to write reliable and maintainable tests.

Aim for one failure path

An ideal test has one failure path. When you create your tests, minimize the possible reasons for a test failure. A developer can debug a problem more easily when there are fewer failure paths.

Avoid non-deterministic code

Reliable tests are predictable and deterministic. Tests that contain non-deterministic code are hard to debug and are often flaky. Non-deterministic code includes the use of randomness, time, and multithreading.

To avoid non-deterministic code, mock the corresponding methods or classes.

Use descriptive test names

Helpful test names contain details about your test, such as test parameters and the expected result. Ideally, a developer can read the test name and know where the buggy code is and how to reproduce the bug.

An easy and effective way to name your methods is to use these three questions:

- What you are testing?
- What are the parameters of the test?
- What is the expected result of the test?

For example, consider a scenario where you want to add a test for the `divide` method:

Java

```java
float Divide(float dividend, float divisor) {
    return dividend / divisor;
}
...

@Test
void <--TestMethodName-->() {
    assertThrows(Divide(10, 0))
}
```

If you use a simple test name, such as `testDivide()`, you are missing important information such as the expected action, parameter information, and expected test result. As a result, triaging a test failure requires you to look at the test implementation to see what the test does.

Instead, use a name such as `invokingDivideWithDivisorEqualToZeroThrowsException()`, which specifies:

- the expected action of the test (`invokingDivide`)
- details about important parameters (the divisor is zero)
- the expected result (the test throws an exception)

If this test fails, you can look at the descriptive test name to find the most probable cause of the failure. In addition, test frameworks and test result dashboards use the test name when reporting test results. Descriptive names enable contributors to look at test suite results and easily see what features are failing.

Long method names are not a problem for test code. Test names are rarely used (usually when you triage and debug), and when you do need to look at a test, it is helpful to have descriptive names.

Use a pre-commit test if possible

Post-commit tests validate that Beam works correctly in broad variety of scenarios. The tests catch errors that are hard to predict in the design and implementation stages.

However, we often write a test to verify a specific scenario. In this situation, it is usually possible to implement the test as a unit test or a component test. You can add your unit tests or component tests to the pre-commit test suite, and the pre-commit test results give you faster code health feedback during the development stage, when a bug is cheap to fix.