Testing Your Pipeline

ng: Dataflow SDK 1.x for Java is unsupported as of October 16, 2018. After August 12, 2020, Dataflow will no
sing Dataflow 1.x and below. See Migrating from Dataflow SDK 1.x for Java
aflow/docs/guides/migrate-java-1-to-2) for migration guidance.

ocumentation on this page applies only to the Dataflow SDK 1.x for Java.

ataflow SDK 2.x for Java and the Dataflow SDK for Python are based on Apache Beam. See the documentatio
aflow/pipelines/creating-a-pipeline-beam) for those SDKs.

Testing your pipeline is a particularly important step in developing an effective data processing solution. The indirect nature of Cloud Dataflow's model, in which your user code constructs a pipeline graph to be executed remotely on Google Cloud Platform, can make debugging failed runs a non-trivial task. Often it is faster and simpler to perform local unit testing on your pipeline code than to debug a pipeline's remote execution.

Unit testing your pipeline code locally, before performing full runs with the Cloud Dataflow service, is often the best and most direct way to identify and fix bugs in your pipeline code. Unit testing your pipeline locally also allows you to use your familiar/favorite local debugging tools.

When testing your code with the Dataflow service, consider limiting the number of worker instances for your p
r the minimum number appropriate for your test. Limiting the number of worker instances used during repeate
an provide significant time and cost savings.

n limit the number of workers your pipeline uses during test runs by setting the `--maxNumWorkers` execution
aflow/pipelines/specifying-exec-params) when you run your test pipeline.

The Dataflow SDKs provide a number of ways to unit test your pipeline code, from the lowest to the highest levels. From the lowest to the highest level, these are:

- You can test the individual function objects, such as DoFn (/dataflow/java-sdk/JavaDoc/com/google/cloud/dataflow/sdk/transforms/DoFn)s, inside your pipeline's core transforms.

- You can test an entire Composite Transform (/dataflow/model/composite-transforms) as a unit.
- You can perform an end-to-end test for an entire pipeline.

Java

To support unit testing, the Dataflow SDK for Java provides a number of test classes in the package com.google.cloud.dataflow.sdk.testing (/dataflow/java-sdk/JavaDoc/com/google/cloud/dataflow/sdk/testing/package-summary). In addition, the transforms included in the SDK have unit tests (https://github.com/GoogleCloudPlatform/DataflowJavaSDK/tree/master-1.x/sdk/src/test/java/com/google/cloud/dataflow/sdk/transforms), and the example programs in the SDK also contain tests (https://github.com/GoogleCloudPlatform/DataflowSDK-examples/tree/master-1.x/src/test/java/com/google/cloud/dataflow/examples). You can use these tests as references and guides.

Testing Individual DoFn Objects

The code in your pipeline's `DoFn` functions runs often, and often across multiple Compute Engine instances. Unit-testing your `DoFn` objects before using them in a Dataflow run can save a great deal of debugging time and energy.

Java

The Dataflow SDK for Java provides a convenient way to test an individual `DoFn` called DoFnTester (/dataflow/java-sdk/JavaDoc/com/google/cloud/dataflow/sdk/transforms/DoFnTester), which is included in the SDK `Transforms` package.

`DoFnTester` uses the JUnit (http://junit.org) framework. To use `DoFnTester`, you'll need to do the following:

1. Create a `DoFnTester`. You'll need to pass an instance of the `DoFn` you want to test to the static factory method for `DoFnTester`.
2. Create one or more main test inputs of the appropriate type for your `DoFn`. If your `DoFn` takes side inputs and/or produces side outputs, you should also create the side inputs and the side output tags.

3. Call `DoFnTester.processBatch` to process the main inputs.
4. Use JUnit's `Assert.assertThat` method to ensure the test outputs returned from `processBatch` match your expected values.

Creating a DoFnTester

To create a `DoFnTester`, first create an instance of the `DoFn` you want to test. You then use that instance when you create a `DoFnTester` using the `.of()` static factory method:

```
static class MyDoFn extends DoFn<String, Integer> { ... }
MyDoFn myDoFn = ...;

DoFnTester<String, Integer> fnTester = DoFnTester.of(myDoFn);
```

Creating Test Inputs

You'll need to create one or more test inputs for `DoFnTester` to send to your `DoFn`. To create test inputs, simply create one or more input variables of the same input type that your `DoFn` accepts. In the case above:

```
static class MyDoFn extends DoFn<String, Integer> { ... }
MyDoFn myDoFn = ...;
DoFnTester<String, Integer> fnTester = DoFnTester.of(myDoFn);

String testInput = "test1";
```

Side Inputs and Outputs

If your `DoFn` accepts side inputs, you can create those side inputs by using the method `DoFnTester.setSideInputs`.

```
static class MyDoFn extends DoFn<String, Integer> { ... }
MyDoFn myDoFn = ...;
DoFnTester<String, Integer> fnTester = DoFnTester.of(myDoFn);

PCollectionView<List<Integer>> sideInput = ...;
```

```
Iterable<Integer> value = ...;
fnTester.setSideInputInGlobalWindow(sideInput, value);
```

If your `DoFn` produces side outputs, you'll need to set the appropriate `TupleTag` objects that you'll use to access each output. A `DoFn` with side outputs produces a `PCollectionTuple` for each side output; you'll need to provide a `TupleTagList` that corresponds to each side output in that tuple.

Suppose your `DoFn` produces side outputs of type `String` and `Integer`. You create `TupleTag` objects for each, and bundle them into a `TupleTagList`, then set it for the `DoFnTester` as follows:

```
static class MyDoFn extends DoFn<String, Integer> { ... }
MyDoFn myDoFn = ...;
DoFnTester<String, Integer> fnTester = DoFnTester.of(myDoFn);

TupleTag<String> tag1 = ...;
TupleTag<Integer> tag2 = ...;
TupleTagList tags = TupleTagList.of(tag1).and(tag2);

fnTester.setSideOutputTags(tags);
```

See the `ParDo` documentation on side inputs (/dataflow/model/par-do#side-inputs) for more information.

Processing Test Inputs and Checking Results

To process the inputs (and thus run the test on your `DoFn`), you call the method `DoFnTester.processBatch`. When you call `processBatch`, you pass one or more main test input values for your `DoFn`. If you set side inputs, the side inputs are available to each batch of main inputs that you provide.

`DoFnTester.processBatch` returns a `List` of outputs—that is, objects of the same type as the `DoFn`'s specified output type. For a `DoFn<String, Integer>`, `processBatch` returns a `List<Integer>`:

```
static class MyDoFn extends DoFn<String, Integer> { ... }
MyDoFn myDoFn = ...;
DoFnTester<String, Integer> fnTester = DoFnTester.of(myDoFn);

String testInput = "test1";
List<Integer> testOutputs = fnTester.processBatch(testInput);
```

To check the results of `processBatch`, you use JUnit's `Assert.assertThat` method to test if the `List` of outputs contains the values you expect:

```
String testInput = "test1";
List<Integer> testOutputs = fnTester.processBatch(testInput);

Assert.assertThat(testOutputs, Matchers.hasItems(...));

// Process a larger batch in a single step.
Assert.assertThat(fnTester.processBatch("input1", "input2", "input3"), Matchers.ha
```

Testing Composite Transforms

To test a composite transform you've created, you can use the following pattern:

- Create a `TestPipeline`.
- Create some static, known test input data.
- Use the `Create` transform to create a `PCollection` of your input data.
- `Apply` your composite transform to the input `PCollection` and save the resulting output `PCollection`.
- Use `DataflowAssert` and its subclasses to verify that the output `PCollection` contains the elements that you expect.

Java

Using the SDK Test Classes

TestPipeline (/dataflow/java-sdk/JavaDoc/com/google/cloud/dataflow/sdk/testing/TestPipeline) and DataflowAssert (/dataflow/java-sdk/JavaDoc/com/google/cloud/dataflow/sdk/testing/DataflowAssert) are classes included in the Cloud Dataflow Java SDK specifically for testing transforms. `TestPipeline` and `DataflowAssert` work with tests configured to run both locally or against the remote Cloud Dataflow service.

TestPipeline

For tests, use `TestPipeline` in place of `Pipeline` when you create the pipeline object. Unlike `Pipeline.create`, `TestPipeline.create` handles setting `PipelineOptions` internally.

You create a `TestPipeline` as follows:

```
Pipeline p = TestPipeline.create();
```

DataflowAssert

`DataflowAssert` is an assertion on the contents of a `PCollection`. You can use `DataflowAssert` to verify that a `PCollection` contains a specific set of expected elements.

For a given `PCollection`, you can use to `DataflowAssert` to verify the contents as follows:

```
PCollection<String> output = ...;

// Check whether a PCollection contains some elements in any order.
DataflowAssert.that(output)
  .containsInAnyOrder(
    "elem1",
    "elem3",
    "elem2");
```

Any code that uses **`DataflowAssert`** must link in **`JUnit`** and **`Hamcrest`**. If you're using Maven, you can link in **`Hamcrest`** by adding the following dependency to your project's **`pom.xml`** file:

```
<dependency>
  <groupId>org.hamcrest</groupId>
  <artifactId>hamcrest-all</artifactId>
  <version>1.3</version>
  <scope>test</scope>
</dependency>
```

For more information on how these classes work, see the com.google.cloud.dataflow.sdk.testing (/dataflow/java-sdk/JavaDoc/com/google/cloud/dataflow/sdk/testing/package-summary)

package documentation.

Using the Create Transform

You can use the `Create` transform to create a `PCollection` out of a standard in-memory collection class, such as Java `List`. See Creating a PCollection (/dataflow/model/pcollection#Creating) for more information.

An Example Test for a Composite Transform

Java

The following code shows a complete test for a composite transform. The test applies the `Count` transform to an input `PCollection` of `String` elements. The test uses the `Create` transform to create the input `PCollection` from a Java `List<String>`.

```
@RunWith(JUnit4.class)
public class CountTest {

// Our static input data, which will make up the initial PCollection.
static final String[] WORDS_ARRAY = new String[] {
  "hi", "there", "hi", "hi", "sue", "bob",
  "hi", "sue", "", "", "ZOW", "bob", ""};

static final List<String> WORDS = Arrays.asList(WORDS_ARRAY);

  @Test
  public void testCount() {
    // Create a test pipeline.
    Pipeline p = TestPipeline.create();

    // Create an input PCollection.
    PCollection<String> input = p.apply(Create.of(WORDS)).setCoder(StringUtf8Coder

    // Apply the Count transform under test.
    PCollection<KV<String, Long>> output =
      input.apply(Count.<String>perElement());

    // Assert on the results.
    DataflowAssert.that(output)
```

```
      .containsInAnyOrder(
          KV.of("hi", 4L),
          KV.of("there", 1L),
          KV.of("sue", 2L),
          KV.of("bob", 2L),
          KV.of("", 3L),
          KV.of("ZOW", 1L));

    // Run the pipeline.
    p.run();
}
```

Testing a Pipeline End-to-End

You can use the test classes in the Dataflow SDKs (such as `TestPipeline` and `DataflowAssert` in the Dataflow SDK for Java) to test an entire pipeline end-to-end. Typically, to test an entire pipeline, you do the following:

- For every source of input data to your pipeline, create some known static test input data.
- Create some static test output data that matches what you expect in your pipeline's final output `PCollection`(s).
- Create a `TestPipeline` in place of the standard `Pipeline.create`.
- In place of your pipeline's `Read` transform(s), use the `Create` transform to create one or more `PCollection`s from your static input data.
- Apply your pipeline's transforms.
- In place of your pipeline's `Write` transform(s), use `DataflowAssert` to verify that the contents of the final `PCollection`s your pipeline produces match the expected values in your static output data.

Testing the WordCount Pipeline

Java

The following example code shows how one might test the WordCount example pipeline (/dataflow/examples/wordcount-example). `WordCount` usually reads lines from a text file for input data; instead, the test creates a Java `List<String>` containing some text lines and uses a `Create` transform to create an initial `PCollection`.

`WordCount`'s final transform (from the composite transform `CountWords`) produces a `PCollection<String>` of formatted word counts suitable for printing. Rather than write that `PCollection` to an output text file, our test pipeline uses `DataflowAssert` to verify that the elements of the `PCollection` match those of a static `String` array containing our expected output data.

```
@RunWith(JUnit4.class)
public class WordCountTest {

  // Our static input data, which will comprise the initial PCollection.
  static final String[] WORDS_ARRAY = new String[] {
    "hi there", "hi", "hi sue bob",
    "hi sue", "", "bob hi"};

  static final List<String> WORDS = Arrays.asList(WORDS_ARRAY);

  // Our static output data, which is the expected data that the final PCollectior
  static final String[] COUNTS_ARRAY = new String[] {
      "hi: 5", "there: 1", "sue: 2", "bob: 2"};

  // Example test that tests the pipeline's transforms.
  @Test
  @Category(com.google.cloud.dataflow.sdk.testing.RunnableOnService.class)
  public void testCountWords() throws Exception {
    Pipeline p = TestPipeline.create();

    // Create a PCollection from the WORDS static input data.
    PCollection<String> input = p.apply(Create.of(WORDS)).setCoder(StringUtf8Coder

    // Run ALL the pipeline's transforms (in this case, the CountWords composite t
    PCollection<String> output = input.apply(new CountWords());

    // Assert that the output PCollection matches the COUNTS_ARRAY known static ou
    DataflowAssert.that(output).containsInAnyOrder(COUNTS_ARRAY);

    // Run the pipeline.
    p.run();
  }
}
```

Last updated 2020-06-22 UTC.