Creating Composite Transforms

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/model/programming-model-beam) for those SDKs.

Transforms in the Dataflow SDK can have a nested structure, in which you can compose a complex transform from multiple simpler transforms. Such a transform might be composed of multiple other transform operations (i.e., they might perform more than one `ParDo`, `Combine`, or `GroupByKey`). These transforms are called **composite transforms**. Composite transforms are useful if you want to create a reusable transform consisting of multiple steps.

Nesting multiple transforms inside a single composite transform can provide multiple benefits to your Dataflow pipeline:

- Composite transforms can make your code more modular and easier to understand, promoting code reuse.
- The Dataflow Monitoring Interface (/dataflow/pipelines/dataflow-monitoring-intf) can refer to composite transforms by name, making it easier for you to track and understand your pipeline's progress at runtime.

An Example of a Composite Transform

Many of the pre-written transforms (/dataflow/model/library-transforms) in the Dataflow SDKs are composite transforms.

The `CountWords` transform in the Dataflow SDK WordCount example program (/dataflow/examples/wordcount-example) is an example of a composite transform. `CountWords` is a `PTransform` subclass that is made up of multiple nested transforms.

In its `apply` method, the `CountWords` transform applies the following transform operations:

1. It applies a `ParDo` on the input `PCollection` of text lines, producing an output `PCollection` of individual words.
2. It applies the Dataflow SDK library transform `Count`* on the `PCollection` of words, producing a `PCollection` of key/value pairs. Each key represents a word in the text, and each value represents the number of times that word appeared in the original data.
3. It applies a final `ParDo` to the `PCollection` of key/value pairs to produce a `PCollection` of printable strings suitable for writing to an output file.

Figure 1 shows a diagram of how the pipeline containing `CountWords` is structured using composite transforms.

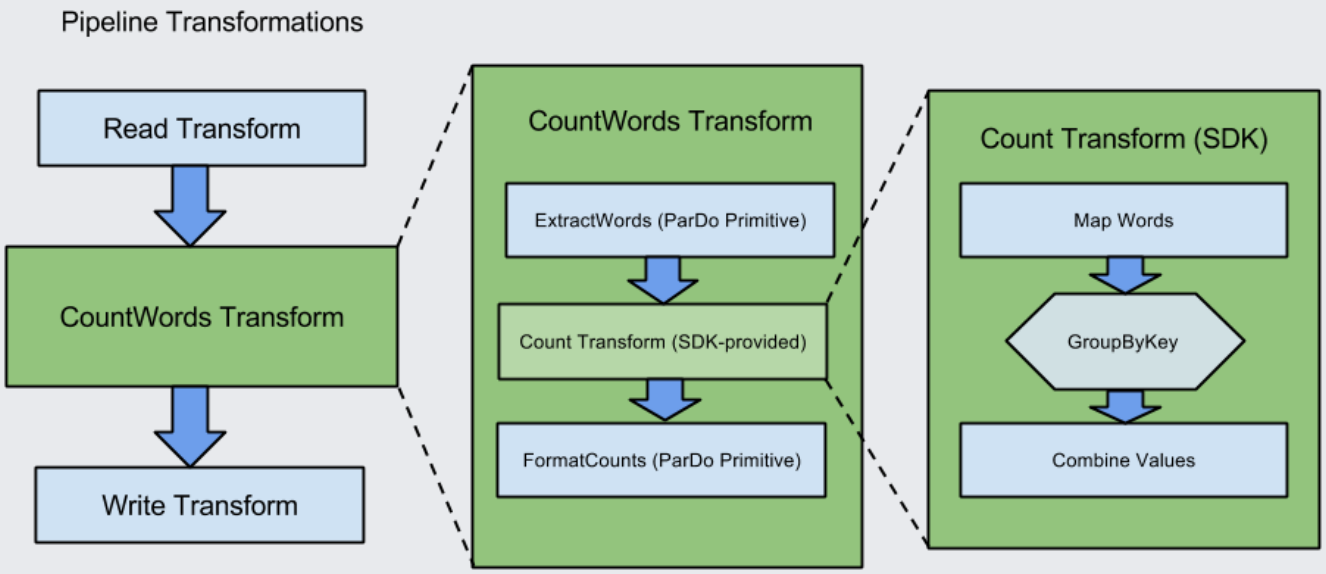

Figure 1: A breakdown of the composite CountWords transform

Java

Your composite transform's parameters and return value must match the initial input type and final return type for the entire transform. For example, `CountWords.apply` accepts an input `PCollection<String>` and returns a `PCollection<String>`, even though the transform's intermediate data changes type multiple times:

```
static class CountWords
    extends PTransform<PCollection<String>, PCollection<String>> {
  @Override
```

```
  public PCollection<String> apply(PCollection<String> lines) {
    PCollection<String> words = lines.apply(
        ParDo
        .named("ExtractWords")
        .of(new ExtractWordsFn()));

    PCollection<KV<String, Integer>> wordCounts =
        words.apply(Count.<String>perElement());

    PCollection<String> results = wordCounts.apply(
        ParDo
        .named("FormatCounts")
        .of(new DoFn<KV<String, Integer>, String>() {
            @Override
            public void processElement(ProcessContext c) {
              c.output(c.element().getKey() + ": " + c.element().getValue());
            }
          }));

    return results;
  }
}
```

Creating a Composite Transform

You can create your own composite transform by creating a subclass of the `Ptransform` class in the Dataflow SDK and overriding the apply method to specify the actual processing logic. You can then use this transform just as you would a built-in transform from the SDK.

Java

For the `PTransform` class type parameters, you pass the `PCollection` types that your transform takes as input and produces as output. To take multiple PCollections as input, or produce multiple PCollections as output, use one of the multi-collection types (/dataflow/model/multiple-pcollections) for the relevant type parameter.

The following code sample shows how to declare a `PTransform` that accepts a `PCollection` of `String`s for input and outputs a `PCollection` of `Integer`s:

```
static class ComputeWordLengths
  extends PTransform<PCollection<String>, PCollection<Integer>> {
  ...
}
```

Overriding the Apply Method

Within your `PTransform` subclass, you'll need to override the `apply` method. `apply` is where you add the processing logic for the `PTransform`. Your override of `apply` must accept the appropriate type of input `PCollection` as a parameter, and specify the output `PCollection` as the return value.

Java

The following code sample shows how to override `apply` for the `ComputeWordLengths` class declared in the previous example:

```
static class ComputeWordLengths
    extends PTransform<PCollection<String>, PCollection<Integer>> {
  @Override
  public PCollection<Integer> apply(PCollection<String>) {
    ...
    // transform logic goes here
    ...
  }
```

As long as you override the `apply` method in your `PTransform` subclass to accept the appropriate input `PCollection`(s) and return the corresponding output `PCollection`(s), you can include as many transforms as you want. These transforms can include core transforms, composite transforms, or the transforms included in the libraries in the Dataflow SDKs.

Java

The **`apply`** method of a **`PTransform`** is not meant to be invoked directly by the user of a transform. Instead, you should call the **`apply`** method (/dataflow/model/transforms) on the **`PCollection`** itself

with the transform as an argument. This allows transforms to be nested within the structure of your pipeline.

Last updated 2020-06-22 UTC.