Apache Beam Programming Guide

The **Beam Programming Guide** is intended for Beam users who want to use the Beam SDKs to create data processing pipelines. It provides guidance for using the Beam SDK classes to build and test your pipeline. It is not intended as an exhaustive reference, but as a language-agnostic, high-level guide to programmatically building your Beam pipeline. As the programming guide is filled out, the text will include code samples in multiple languages to help illustrate how to implement Beam concepts in your pipelines.

Adapt for: Java SDK Python SDK

Table of Contents:

1. Overview

To use Beam, you need to first create a driver program using the classes in one of the Beam SDKs. Your driver program *defines* your pipeline, including all of the inputs, transforms, and outputs; it also sets execution options for your pipeline (typically passed in using command-line options). These include the Pipeline Runner, which, in turn, determines what back-end your pipeline will run on.

The Beam SDKs provide a number of abstractions that simplify the mechanics of large-scale distributed data processing. The same Beam abstractions work with both batch and streaming data sources. When you create your Beam pipeline, you can think about your data processing task in terms of these abstractions. They include:

- `Pipeline` : A `Pipeline` encapsulates your entire data processing task, from start to finish. This includes reading input data, transforming that data, and writing output data. All Beam driver programs must create a `Pipeline` . When you create the `Pipeline` , you must also specify the execution options that tell the `Pipeline` where and how to run.
- `PCollection` : A `PCollection` represents a distributed data set that your Beam pipeline operates on. The data set can be *bounded*, meaning it comes from a fixed source like a file, or *unbounded*, meaning it comes from a continuously updating source via a subscription or other mechanism. Your pipeline typically creates an initial `PCollection` by reading data from an external data source, but you can also create a `PCollection` from in-memory data within your driver program. From there, `PCollection` s are the inputs and outputs for each step in your pipeline.
- `Transform` : A `Transform` represents a data processing operation, or a step, in your pipeline. Every `Transform` takes one or more `PCollection` objects as input, performs a processing function that you provide on the elements of that `PCollection` , and produces one or more output `PCollection` objects.
- I/O `Source` and `Sink` : Beam provides `Source` and `Sink` APIs to represent reading and writing data, respectively. `Source` encapsulates the code necessary to read data into your Beam pipeline from some external source, such as cloud file storage or a subscription to a streaming data source. `Sink` likewise encapsulates the code necessary to write the elements of a `PCollection` to an external data sink.

A typical Beam driver program works as follows:

- Create a `Pipeline` object and set the pipeline execution options, including the Pipeline Runner.
- Create an initial `PCollection` for pipeline data, either using the `Source` API to read data from an external source, or using a `Create` transform to build a `PCollection` from in-memory data.
- Apply **Transforms** to each `PCollection` . Transforms can change, filter, group, analyze, or otherwise process the elements in a `PCollection` . A transform creates a new output `PCollection` *without consuming the input collection*. A typical pipeline applies subsequent transforms to the each new output `PCollection` in turn until processing is complete.
- Output the final, transformed `PCollection` (s), typically using the `Sink` API to write data to an external source.
- **Run** the pipeline using the designated Pipeline Runner.

When you run your Beam driver program, the Pipeline Runner that you designate constructs a **workflow graph** of your pipeline based on the `PCollection` objects you've created and transforms that you've applied. That graph is then executed using the appropriate distributed processing back-end, becoming an asynchronous "job" (or equivalent) on that back-end.

2. Creating a pipeline

The `Pipeline` abstraction encapsulates all the data and steps in your data processing task. Your Beam driver program typically starts by constructing a Pipeline (/documentation/sdks/javadoc/2.1.0/index.html?org/apache/beam/sdk/Pipeline.html) object, and then using that object as the basis for creating the pipeline's data sets as `PCollection` s and its operations as `Transform` s.

To use Beam, your driver program must first create an instance of the Beam SDK class `Pipeline` (typically in the `main()` function). When you create your `Pipeline` , you'll also need to set some **configuration options**. You can set your pipeline's configuration options programatically, but it's often easier to set the options ahead of time (or read them from the command line) and pass them to the `Pipeline` object when you create the object.

Java Python

```
// Start by defining the options for the pipeline.
PipelineOptions options = PipelineOptionsFactory.create();

// Then create the pipeline.
Pipeline p = Pipeline.create(options);
```

2.1. Configuring pipeline options

Use the pipeline options to configure different aspects of your pipeline, such as the pipeline runner that will execute your pipeline and any runner-specific configuration required by the chosen runner. Your pipeline options will potentially include information such as your project ID or a location for storing files.

When you run the pipeline on a runner of your choice, a copy of the PipelineOptions will be available to your code. For example, you can read PipelineOptions from a DoFn's Context.

2.1.1. Setting PipelineOptions from command-line arguments

While you can configure your pipeline by creating a `PipelineOptions` object and setting the fields directly, the Beam SDKs include a command-line parser that you can use to set fields in `PipelineOptions` using command-line arguments.

To read options from the command-line, construct your `PipelineOptions` object as demonstrated in the following example code:

Java Python

```
MyOptions options = PipelineOptionsFactory.fromArgs(args).withValidation().create();
```

This interprets command-line arguments that follow the format:

```
--<option>=<value>
```

> **Note:** Appending the method `.withValidation` will check for required command-line arguments and validate argument values.

Building your `PipelineOptions` this way lets you specify any of the options as a command-line argument.

> **Note:** The WordCount example pipeline (/get-started/wordcount-example) demonstrates how to set pipeline options at runtime by using command-line options.

2.1.2. Creating custom options

You can add your own custom options in addition to the standard `PipelineOptions`. To add your own options, define an interface with getter and setter methods for each option, as in the following example:

Java Python

```
public interface MyOptions extends PipelineOptions {
    String getMyCustomOption();
    void setMyCustomOption(String myCustomOption);
  }
```

You can also specify a description, which appears when a user passes `--help` as a command-line argument, and a default value.

You set the description and default value using annotations, as follows:

Java Python

```
public interface MyOptions extends PipelineOptions {
    @Description("My custom command line argument.")
    @Default.String("DEFAULT")
    String getMyCustomOption();
    void setMyCustomOption(String myCustomOption);
  }
```

It's recommended that you register your interface with `PipelineOptionsFactory` and then pass the interface when creating the `PipelineOptions` object. When you register your interface with `PipelineOptionsFactory`, the `--help` can find your custom options interface and add it to the output of the `--help` command. `PipelineOptionsFactory` will also validate that your custom options are compatible with all other registered options.

The following example code shows how to register your custom options interface with `PipelineOptionsFactory`:

```
PipelineOptionsFactory.register(MyOptions.class);
MyOptions options = PipelineOptionsFactory.fromArgs(args)
                                                .withValidation()
                                                .as(MyOptions.class);
```

Now your pipeline can accept `--myCustomOption=value` as a command-line argument.

3. PCollections

The PCollection (/documentation/sdks/javadoc/2.1.0/index.html?org/apache/beam/sdk/values/PCollection.html) abstraction represents a potentially distributed, multi-element data set. You can think of a `PCollection` as "pipeline" data; Beam transforms use `PCollection` objects as inputs and outputs. As such, if you want to work with data in your pipeline, it must be in the form of a `PCollection`.

After you've created your `Pipeline`, you'll need to begin by creating at least one `PCollection` in some form. The `PCollection` you create serves as the input for the first operation in your pipeline.

3.1. Creating a PCollection

You create a `PCollection` by either reading data from an external source using Beam's Source API, or you can create a `PCollection` of data stored in an in-memory collection class in your driver program. The former is typically how a production pipeline would ingest data; Beam's Source APIs contain adapters to help you read from external sources

like large cloud-based files, databases, or subscription services. The latter is primarily useful for testing and debugging purposes.

3.1.1. Reading from an external source

To read from an external source, you use one of the Beam-provided I/O adapters. The adapters vary in their exact usage, but all of them from some external data source and return a `PCollection` whose elements represent the data records in that source.

Each data source adapter has a `Read` transform; to read, you must apply that transform to the `Pipeline` object itself. `TextIO.Read` , for example, reads from an external text file and returns a `PCollection` whose elements are of type `String` , each `String` represents one line from the text file. Here's how you would apply `TextIO.Read` to your `Pipeline` to create a `PCollection` :

Java Python

```
public static void main(String[] args) {
    // Create the pipeline.
    PipelineOptions options =
        PipelineOptionsFactory.fromArgs(args).create();
    Pipeline p = Pipeline.create(options);

    // Create the PCollection 'lines' by applying a 'Read' transform.
    PCollection<String> lines = p.apply(
      "ReadMyFile", TextIO.read().from("protocol://path/to/some/inputData.txt"));
}
```

See the section on I/O to learn more about how to read from the various data sources supported by the Beam SDK.

3.1.2. Creating a PCollection from in-memory data

To create a `PCollection` from an in-memory Java `Collection` , you use the Beam-provided `Create` transform. Much like a data adapter's `Read` , you apply `Create` directly to your `Pipeline` object itself.

As parameters, `Create` accepts the Java `Collection` and a `Coder` object. The `Coder` specifies how the elements in the `Collection` should be encoded.

The following example code shows how to create a `PCollection` from an in-memory `List` :

Java Python

```
public static void main(String[] args) {
    // Create a Java Collection, in this case a List of Strings.
    static final List<String> LINES = Arrays.asList(
      "To be, or not to be: that is the question: ",
      "Whether 'tis nobler in the mind to suffer ",
      "The slings and arrows of outrageous fortune, ",
      "Or to take arms against a sea of troubles, ");

    // Create the pipeline.
    PipelineOptions options =
        PipelineOptionsFactory.fromArgs(args).create();
    Pipeline p = Pipeline.create(options);

    // Apply Create, passing the list and the coder, to create the PCollection.
    p.apply(Create.of(LINES)).setCoder(StringUtf8Coder.of())
}
```

3.2. PCollection characteristics

A `PCollection` is owned by the specific `Pipeline` object for which it is created; multiple pipelines cannot share a `PCollection`. In some respects, a `PCollection` functions like a collection class. However, a `PCollection` can differ in a few key ways:

3.2.1. Element type

The elements of a `PCollection` may be of any type, but must all be of the same type. However, to support distributed processing, Beam needs to be able to encode each individual element as a byte string (so elements can be passed around to distributed workers). The Beam SDKs provide a data encoding mechanism that includes built-in encoding for commonly-used types as well as support for specifying custom encodings as needed.

3.2.2. Immutability

A `PCollection` is immutable. Once created, you cannot add, remove, or change individual elements. A Beam Transform might process each element of a `PCollection` and generate new pipeline data (as a new `PCollection`), *but it does not consume or modify the original input collection*.

3.2.3. Random access

A `PCollection` does not support random access to individual elements. Instead, Beam Transforms consider every element in a `PCollection` individually.

3.2.4. Size and boundedness

A `PCollection` is a large, immutable “bag” of elements. There is no upper limit on how many elements a `PCollection` can contain; any given `PCollection` might fit in memory on a single machine, or it might represent a very large distributed data set backed by a persistent data store.

A `PCollection` can be either **bounded** or **unbounded** in size. A **bounded** `PCollection` represents a data set of a known, fixed size, while an **unbounded** `PCollection` represents a data set of unlimited size. Whether a `PCollection` is bounded or unbounded depends on the source of the data set that it represents. Reading from a batch data source, such as a file or a database, creates a bounded `PCollection`. Reading from a streaming or continously-updating data source, such as Pub/Sub or Kafka, creates an unbounded `PCollection` (unless you explicitly tell it not to).

The bounded (or unbounded) nature of your `PCollection` affects how Beam processes your data. A bounded `PCollection` can be processed using a batch job, which might read the entire data set once, and perform processing in a job of finite length. An unbounded `PCollection` must be processed using a streaming job that runs continuously, as the entire collection can never be available for processing at any one time.

When performing an operation that groups elements in an unbounded `PCollection`, Beam requires a concept called **windowing** to divide a continuously updating data set into logical windows of finite size. Beam processes each window as a bundle, and processing continues as the data set is generated. These logical windows are determined by some characteristic associated with a data element, such as a **timestamp**.

3.2.5. Element timestamps

Each element in a `PCollection` has an associated intrinsic **timestamp**. The timestamp for each element is initially assigned by the Source that creates the `PCollection`. Sources that create an unbounded `PCollection` often assign each new element a timestamp that corresponds to when the element was read or added.

> **Note**: Sources that create a bounded `PCollection` for a fixed data set also automatically assign timestamps, but the most common behavior is to assign every element the same timestamp (`Long.MIN_VALUE`).

Timestamps are useful for a `PCollection` that contains elements with an inherent notion of time. If your pipeline is reading a stream of events, like Tweets or other social media messages, each element might use the time the event was posted as the element timestamp.

You can manually assign timestamps to the elements of a `PCollection` if the source doesn't do it for you. You'll want to do this if the elements have an inherent timestamp, but the timestamp is somewhere in the structure of the element itself (such as a "time" field in a server log entry). Beam has Transforms that take a `PCollection` as input and output an identical `PCollection` with timestamps attached; see Assigning Timestamps for more information about how to do so.

4. Transforms

Transforms are the operations in your pipeline, and provide a generic processing framework. You provide processing logic in the form of a function object (colloquially referred to as "user code"), and your user code is applied to each element of an input `PCollection` (or more than one `PCollection`). Depending on the pipeline runner and back-end that you choose, many different workers across a cluster may execute instances of your user code in parallel. The user code running on each worker generates the output elements that are ultimately added to the final output `PCollection` that the transform produces.

The Beam SDKs contain a number of different transforms that you can apply to your pipeline's `PCollection` s. These include general-purpose core transforms, such as ParDo or Combine. There are also pre-written composite transforms included in the SDKs, which combine one or more of the core transforms in a useful processing pattern, such as counting or combining elements in a collection. You can also define your own more complex composite transforms to fit your pipeline's exact use case.

4.1. Applying transforms

To invoke a transform, you must **apply** it to the input `PCollection` . Each transform in the Beam SDKs has a generic `apply` method . Invoking multiple Beam transforms is similar to *method chaining*, but with one slight difference: You apply the transform to the input `PCollection` , passing the transform itself as an argument, and the operation returns the output `PCollection` . This takes the general form:

Java Python

```
[Output PCollection] = [Input PCollection].apply([Transform])
```

Because Beam uses a generic `apply` method for `PCollection` , you can both chain transforms sequentially and also apply transforms that contain other transforms nested within (called composite transforms in the Beam SDKs).

How you apply your pipeline's transforms determines the structure of your pipeline. The best way to think of your pipeline is as a directed acyclic graph, where the nodes are `PCollection` s and the edges are transforms. For example, you can chain transforms to create a sequential pipeline, like this one:

Java Python

```
[Final Output PCollection] = [Initial Input PCollection].apply([First Transform])
.apply([Second Transform])
.apply([Third Transform])
```

The resulting workflow graph of the above pipeline looks like this:

[Sequential Graph Graphic]

However, note that a transform *does not consume or otherwise alter* the input collection–remember that a `PCollection` is immutable by definition. This means that you can apply multiple transforms to the same input `PCollection` to create a branching pipeline, like so:

Java Python

```
[Output PCollection 1] = [Input PCollection].apply([Transform 1])
[Output PCollection 2] = [Input PCollection].apply([Transform 2])
```

The resulting workflow graph from the branching pipeline above looks like this:

[Branching Graph Graphic]

You can also build your own composite transforms that nest multiple sub-steps inside a single, larger transform. Composite transforms are particularly useful for building a reusable sequence of simple steps that get used in a lot of different places.

4.2. Core Beam transforms

Beam provides the following core transforms, each of which represents a different processing paradigm:

- `ParDo`
- `GroupByKey`
- `CoGroupByKey`
- `Combine`
- `Flatten`
- `Partition`

4.2.1. ParDo

`ParDo` is a Beam transform for generic parallel processing. The `ParDo` processing paradigm is similar to the “Map” phase of a Map/Shuffle/Reduce-style algorithm: a `ParDo` transform considers each element in the input `PCollection`, performs some processing function (your user code) on that element, and emits zero, one, or multiple elements to an output `PCollection`.

`ParDo` is useful for a variety of common data processing operations, including:

- **Filtering a data set.** You can use `ParDo` to consider each element in a `PCollection` and either output that element to a new collection, or discard it.
- **Formatting or type-converting each element in a data set.** If your input `PCollection` contains elements that are of a different type or format than you want, you can use `ParDo` to perform a conversion on each element and output the result to a new `PCollection`.
- **Extracting parts of each element in a data set.** If you have a `PCollection` of records with multiple fields, for example, you can use a `ParDo` to parse out just the fields you want to consider into a new `PCollection`.
- **Performing computations on each element in a data set.** You can use `ParDo` to perform simple or complex computations on every element, or certain elements, of a `PCollection` and output the results as a new `PCollection`.

In such roles, `ParDo` is a common intermediate step in a pipeline. You might use it to extract certain fields from a set of raw input records, or convert raw input into a different format; you might also use `ParDo` to convert processed data into a format suitable for output, like database table rows or printable strings.

When you apply a `ParDo` transform, you’ll need to provide user code in the form of a `DoFn` object. `DoFn` is a Beam SDK class that defines a distributed processing function.

> When you create a subclass of `DoFn`, note that your subclass should adhere to the Requirements for writing user code for Beam transforms.

4.2.1.1. Applying ParDo

Like all Beam transforms, you apply `ParDo` by calling the `apply` method on the input `PCollection` and passing `ParDo` as an argument, as shown in the following example code:

Java | Python

```
// The input PCollection of Strings.
PCollection<String> words = ...;

// The DoFn to perform on each element in the input PCollection.
static class ComputeWordLengthFn extends DoFn<String, Integer> { ... }

// Apply a ParDo to the PCollection "words" to compute lengths for each word.
PCollection<Integer> wordLengths = words.apply(
    ParDo
    .of(new ComputeWordLengthFn()));        // The DoFn to perform on each element, which
                                            // we define above.
```

In the example, our input `PCollection` contains `String` values. We apply a `ParDo` transform that specifies a function (`ComputeWordLengthFn`) to compute the length of each string, and outputs the result to a new `PCollection` of `Integer` values that stores the length of each word.

4.2.1.2. Creating a DoFn

The `DoFn` object that you pass to `ParDo` contains the processing logic that gets applied to the elements in the input collection. When you use Beam, often the most important pieces of code you'll write are these `DoFn` s–they're what define your pipeline's exact data processing tasks.

> **Note:** When you create your `DoFn` , be mindful of the Requirements for writing user code for Beam transforms and ensure that your code follows them.

A `DoFn` processes one element at a time from the input `PCollection` . When you create a subclass of `DoFn` , you'll need to provide type parameters that match the types of the input and output elements. If your `DoFn` processes incoming `String` elements and produces `Integer` elements for the output collection (like our previous example, `ComputeWordLengthFn`), your class declaration would look like this:

```
static class ComputeWordLengthFn extends DoFn<String, Integer> { ... }
```

Inside your `DoFn` subclass, you'll write a method annotated with `@ProcessElement` where you provide the actual processing logic. You don't need to manually extract the elements from the input collection; the Beam SDKs handle that for you. Your `@ProcessElement` method should accept an object of type `ProcessContext` . The `ProcessContext` object gives you access to an input element and a method for emitting an output element:

```
static class ComputeWordLengthFn extends DoFn<String, Integer> {
  @ProcessElement
  public void processElement(ProcessContext c) {
    // Get the input element from ProcessContext.
    String word = c.element();
    // Use ProcessContext.output to emit the output element.
    c.output(word.length());
  }
}
```

> **Note:** If the elements in your input `PCollection` are key/value pairs, you can access the key or value by using `ProcessContext.element().getKey()` or `ProcessContext.element().getValue()`, respectively.

A given `DoFn` instance generally gets invoked one or more times to process some arbitrary bundle of elements. However, Beam doesn't guarantee an exact number of invocations; it may be invoked multiple times on a given worker node to account for failures and retries. As such, you can cache information across multiple calls to your processing method, but if you do so, make sure the implementation **does not depend on the number of invocations**.

In your processing method, you'll also need to meet some immutability requirements to ensure that Beam and the processing back-end can safely serialize and cache the values in your pipeline. Your method should meet the following requirements:

- You should not in any way modify an element returned by `ProcessContext.element()` or `ProcessContext.sideInput()` (the incoming elements from the input collection).
- Once you output a value using `ProcessContext.output()` or `ProcessContext.sideOutput()`, you should not modify that value in any way.

4.2.1.3. Lightweight DoFns and other abstractions

If your function is relatively straightforward, you can simplify your use of `ParDo` by providing a lightweight `DoFn` in-line, as an anonymous inner class instance .

Here's the previous example, `ParDo` with `ComputeLengthWordsFn`, with the `DoFn` specified as an anonymous inner class instance :

Java Python

```
// The input PCollection.
PCollection<String> words = ...;

// Apply a ParDo with an anonymous DoFn to the PCollection words.
// Save the result as the PCollection wordLengths.
PCollection<Integer> wordLengths = words.apply(
  "ComputeWordLengths",                     // the transform name
  ParDo.of(new DoFn<String, Integer>() {    // a DoFn as an anonymous inner class instance
      @ProcessElement
      public void processElement(ProcessContext c) {
        c.output(c.element().length());
      }
    }));
```

If your `ParDo` performs a one-to-one mapping of input elements to output elements–that is, for each input element, it applies a function that produces *exactly one* output element, you can use the higher-level `MapElements` transform. `MapElements` can accept an anonymous Java 8 lambda function for additional brevity.

Here's the previous example using `MapElements` :

Java Python

```
// The input PCollection.
PCollection<String> words = ...;

// Apply a MapElements with an anonymous lambda function to the PCollection words.
// Save the result as the PCollection wordLengths.
PCollection<Integer> wordLengths = words.apply(
  MapElements.into(TypeDescriptors.integers())
             .via((String word) -> word.length()));
```

> **Note:** You can use Java 8 lambda functions with several other Beam transforms, including `Filter`, `FlatMapElements`, and `Partition`.

4.2.2. GroupByKey

`GroupByKey` is a Beam transform for processing collections of key/value pairs. It's a parallel reduction operation, analogous to the Shuffle phase of a Map/Shuffle/Reduce-style algorithm. The input to `GroupByKey` is a collection of key/value pairs that represents a *multimap*, where the collection contains multiple pairs that have the same key, but different values. Given such a collection, you use `GroupByKey` to collect all of the values associated with each unique key.

`GroupByKey` is a good way to aggregate data that has something in common. For example, if you have a collection that stores records of customer orders, you might want to group together all the orders from the same postal code (wherein the "key" of the key/value pair is the postal code field, and the "value" is the remainder of the record).

Let's examine the mechanics of `GroupByKey` with a simple example case, where our data set consists of words from a text file and the line number on which they appear. We want to group together all the line numbers (values) that share the same word (key), letting us see all the places in the text where a particular word appears.

Our input is a `PCollection` of key/value pairs where each word is a key, and the value is a line number in the file where the word appears. Here's a list of the key/value pairs in the input collection:

```
cat, 1
dog, 5
and, 1
jump, 3
tree, 2
cat, 5
dog, 2
and, 2
cat, 9
and, 6
...
```

`GroupByKey` gathers up all the values with the same key and outputs a new pair consisting of the unique key and a collection of all of the values that were associated with that key in the input collection. If we apply `GroupByKey` to our input collection above, the output collection would look like this:

```
cat, [1,5,9]
dog, [5,2]
and, [1,2,6]
jump, [3]
tree, [2]
...
```

Thus, `GroupByKey` represents a transform from a multimap (multiple keys to individual values) to a uni-map (unique keys to collections of values).

4.2.3. CoGroupByKey

`CoGroupByKey` joins two or more key/value `PCollection` s that have the same key type, and then emits a collection of `KV<K, CoGbkResult>` pairs. Design Your Pipeline (/documentation/pipelines/design-your-pipeline/#multiple-sources) shows an example pipeline that uses a join.

Given the input collections below:

```
// collection 1
user1, address1
user2, address2
user3, address3

// collection 2
user1, order1
user1, order2
user2, order3
guest, order4
...
```

`CoGroupByKey` gathers up the values with the same key from all `PCollection` s, and outputs a new pair consisting of the unique key and an object `CoGbkResult` containing all values that were associated with that key. If you apply `CoGroupByKey` to the input collections above, the output collection would look like this:

```
user1, [[address1], [order1, order2]]
user2, [[address2], [order3]]
user3, [[address3], []]
guest, [[], [order4]]
...
```

A Note on Key/Value Pairs: Beam represents key/value pairs slightly differently depending on the language and SDK you're using. In the Beam SDK for Java, you represent a key/value pair with an object of type `KV<K, V>` . In Python, you represent key/value pairs with 2-tuples.

4.2.4. Combine

`Combine` (/documentation/sdks/javadoc/2.1.0/index.html?org/apache/beam/sdk/transforms/Combine.html) is a Beam transform for combining collections of elements or values in your data. `Combine` has variants that work on entire `PCollection` s, and some that combine the values for each key in `PCollection` s of key/value pairs.

When you apply a `Combine` transform, you must provide the function that contains the logic for combining the elements or values. The combining function should be commutative and associative, as the function is not necessarily invoked exactly once on all values with a given key. Because the input data (including the value collection) may be distributed across multiple workers, the combining function might be called multiple times to perform partial combining on subsets of the value collection. The Beam SDK also provides some pre-built combine functions for common numeric combination operations such as sum, min, and max.

Simple combine operations, such as sums, can usually be implemented as a simple function. More complex combination operations might require you to create a subclass of `CombineFn` that has an accumulation type distinct from the input/output type.

4.2.4.1. Simple combinations using simple functions

The following example code shows a simple combine function.

Java Python

```
// Sum a collection of Integer values. The function SumInts implements the interface SerializableFunction.
public static class SumInts implements SerializableFunction<Iterable<Integer>, Integer> {
  @Override
  public Integer apply(Iterable<Integer> input) {
    int sum = 0;
    for (int item : input) {
      sum += item;
    }
    return sum;
  }
}
```

4.2.4.2. Advanced combinations using CombineFn

For more complex combine functions, you can define a subclass of `CombineFn`. You should use `CombineFn` if the combine function requires a more sophisticated accumulator, must perform additional pre- or post-processing, might change the output type, or takes the key into account.

A general combining operation consists of four operations. When you create a subclass of `CombineFn`, you must provide four operations by overriding the corresponding methods:

1. **Create Accumulator** creates a new "local" accumulator. In the example case, taking a mean average, a local accumulator tracks the running sum of values (the numerator value for our final average division) and the number of values summed so far (the denominator value). It may be called any number of times in a distributed fashion.
2. **Add Input** adds an input element to an accumulator, returning the accumulator value. In our example, it would update the sum and increment the count. It may also be invoked in parallel.
3. **Merge Accumulators** merges several accumulators into a single accumulator; this is how data in multiple accumulators is combined before the final calculation. In the case of the mean average computation, the accumulators representing each portion of the division are merged together. It may be called again on its outputs any number of times.
4. **Extract Output** performs the final computation. In the case of computing a mean average, this means dividing the combined sum of all the values by the number of values summed. It is called once on the final, merged accumulator.

The following example code shows how to define a `CombineFn` that computes a mean average:

Java Python

```
public class AverageFn extends CombineFn<Integer, AverageFn.Accum, Double> {
  public static class Accum {
    int sum = 0;
    int count = 0;
  }

  @Override
  public Accum createAccumulator() { return new Accum(); }

  @Override
  public Accum addInput(Accum accum, Integer input) {
      accum.sum += input;
      accum.count++;
      return accum;
  }

  @Override
  public Accum mergeAccumulators(Iterable<Accum> accums) {
    Accum merged = createAccumulator();
    for (Accum accum : accums) {
      merged.sum += accum.sum;
      merged.count += accum.count;
    }
    return merged;
  }

  @Override
  public Double extractOutput(Accum accum) {
    return ((double) accum.sum) / accum.count;
  }
}
```

If you are combining a `PCollection` of key-value pairs, per-key combining is often enough. If you need the combining strategy to change based on the key (for example, MIN for some users and MAX for other users), you can define a `KeyedCombineFn` to access the key within the combining strategy.

4.2.4.3. Combining a PCollection into a single value

Use the global combine to transform all of the elements in a given `PCollection` into a single value, represented in your pipeline as a new `PCollection` containing one element. The following example code shows how to apply the Beam provided sum combine function to produce a single sum value for a `PCollection` of integers.

Java | Python

```
// Sum.SumIntegerFn() combines the elements in the input PCollection. The resulting PCollec
tion, called sum,
// contains one value: the sum of all the elements in the input PCollection.
PCollection<Integer> pc = ...;
PCollection<Integer> sum = pc.apply(
   Combine.globally(new Sum.SumIntegerFn()));
```

4.2.4.4. Combine and global windowing

If your input `PCollection` uses the default global windowing, the default behavior is to return a `PCollection` containing one item. That item's value comes from the accumulator in the combine function that you specified when applying `Combine` . For example, the Beam provided sum combine function returns a zero value (the sum of an empty input), while the max combine function returns a maximal or infinite value.

To have `Combine` instead return an empty `PCollection` if the input is empty, specify `.withoutDefaults` when you apply your `Combine` transform, as in the following code example:

Java | Python

```
PCollection<Integer> pc = ...;
PCollection<Integer> sum = pc.apply(
  Combine.globally(new Sum.SumIntegerFn()).withoutDefaults());
```

4.2.4.5. Combine and non-global windowing

If your `PCollection` uses any non-global windowing function, Beam does not provide the default behavior. You must specify one of the following options when applying `Combine`:

- Specify `.withoutDefaults`, where windows that are empty in the input `PCollection` will likewise be empty in the output collection.
- Specify `.asSingletonView`, in which the output is immediately converted to a `PCollectionView`, which will provide a default value for each empty window when used as a side input. You'll generally only need to use this option if the result of your pipeline's `Combine` is to be used as a side input later in the pipeline.

4.2.4.6. Combining values in a keyed PCollection

After creating a keyed PCollection (for example, by using a `GroupByKey` transform), a common pattern is to combine the collection of values associated with each key into a single, merged value. Drawing on the previous example from `GroupByKey`, a key-grouped `PCollection` called `groupedWords` looks like this:

```
  cat, [1,5,9]
  dog, [5,2]
  and, [1,2,6]
  jump, [3]
  tree, [2]
  ...
```

In the above `PCollection`, each element has a string key (for example, "cat") and an iterable of integers for its value (in the first element, containing [1, 5, 9]). If our pipeline's next processing step combines the values (rather than considering them individually), you can combine the iterable of integers to create a single, merged value to be paired with each key. This pattern of a `GroupByKey` followed by merging the collection of values is equivalent to Beam's Combine PerKey transform. The combine function you supply to Combine PerKey must be an associative reduction function or a subclass of `CombineFn`.

Java | Python

```
// PCollection is grouped by key and the Double values associated with each key are combine
d into a Double.
PCollection<KV<String, Double>> salesRecords = ...;
PCollection<KV<String, Double>> totalSalesPerPerson =
  salesRecords.apply(Combine.<String, Double, Double>perKey(
    new Sum.SumDoubleFn()));

// The combined value is of a different type than the original collection of values per ke
y. PCollection has
// keys of type String and values of type Integer, and the combined value is a Double.
PCollection<KV<String, Integer>> playerAccuracy = ...;
PCollection<KV<String, Double>> avgAccuracyPerPlayer =
  playerAccuracy.apply(Combine.<String, Integer, Double>perKey(
    new MeanInts()));
```

4.2.5. Flatten

`Flatten` (/documentation/sdks/javadoc/2.1.0/index.html?org/apache/beam/sdk/transforms/Flatten.html) and is a Beam transform for `PCollection` objects that store the same data type. `Flatten` merges multiple `PCollection` objects into a single logical `PCollection`.

The following example shows how to apply a `Flatten` transform to merge multiple `PCollection` objects.

Java | Python

```
// Flatten takes a PCollectionList of PCollection objects of a given type.
// Returns a single PCollection that contains all of the elements in the PCollection object
s in that list.
PCollection<String> pc1 = ...;
PCollection<String> pc2 = ...;
PCollection<String> pc3 = ...;
PCollectionList<String> collections = PCollectionList.of(pc1).and(pc2).and(pc3);

PCollection<String> merged = collections.apply(Flatten.<String>pCollections());
```

4.2.5.1. Data encoding in merged collections

By default, the coder for the output `PCollection` is the same as the coder for the first `PCollection` in the input `PCollectionList`. However, the input `PCollection` objects can each use different coders, as long as they all contain the same data type in your chosen language.

4.2.5.2. Merging windowed collections

When using `Flatten` to merge `PCollection` objects that have a windowing strategy applied, all of the `PCollection` objects you want to merge must use a compatible windowing strategy and window sizing. For example, all the collections you're merging must all use (hypothetically) identical 5-minute fixed windows or 4-minute sliding windows starting every 30 seconds.

If your pipeline attempts to use `Flatten` to merge `PCollection` objects with incompatible windows, Beam generates an `IllegalStateException` error when your pipeline is constructed.

4.2.6. Partition

`Partition` (/documentation/sdks/javadoc/2.1.0/index.html?org/apache/beam/sdk/transforms/Partition.html) is a Beam transform for `PCollection` objects that store the same data type. `Partition` splits a single `PCollection` into a fixed number of smaller collections.

`Partition` divides the elements of a `PCollection` according to a partitioning function that you provide. The partitioning function contains the logic that determines how to split up the elements of the input `PCollection` into each resulting partition `PCollection`. The number of partitions must be determined at graph construction time. You can, for example, pass the number of partitions as a command-line option at runtime (which will then be used to build your pipeline graph), but you cannot determine the number of partitions in mid-pipeline (based on data calculated after your pipeline graph is constructed, for instance).

The following example divides a `PCollection` into percentile groups.

Java | Python

```
// Provide an int value with the desired number of result partitions, and a PartitionFn tha
t represents the
// partitioning function. In this example, we define the PartitionFn in-line. Returns a PCo
llectionList
// containing each of the resulting partitions as individual PCollection objects.
PCollection<Student> students = ...;
// Split students up into 10 partitions, by percentile:
PCollectionList<Student> studentsByPercentile =
    students.apply(Partition.of(10, new PartitionFn<Student>() {
        public int partitionFor(Student student, int numPartitions) {
            return student.getPercentile()  // 0..99
                 * numPartitions / 100;
        }}));

// You can extract each partition from the PCollectionList using the get method, as follow
s:
PCollection<Student> fortiethPercentile = studentsByPercentile.get(4);
```

4.3. Requirements for writing user code for Beam transforms

When you build user code for a Beam transform, you should keep in mind the distributed nature of execution. For example, there might be many copies of your function running on a lot of different machines in parallel, and those copies function independently, without communicating or sharing state with any of the other copies. Depending on the Pipeline Runner and processing back-end you choose for your pipeline, each copy of your user code function may be retried or run multiple times. As such, you should be cautious about including things like state dependency in your user code.

In general, your user code must fulfill at least these requirements:

- Your function object must be **serializable**.
- Your function object must be **thread-compatible**, and be aware that *the Beam SDKs are not thread-safe*.

In addition, it's recommended that you make your function object **idempotent**.

> **Note:** These requirements apply to subclasses of `DoFn` (a function object used with the ParDo transform), `CombineFn` (a function object used with the Combine transform), and `WindowFn` (a function object used with the Window transform).

4.3.1. Serializability

Any function object you provide to a transform must be **fully serializable**. This is because a copy of the function needs to be serialized and transmitted to a remote worker in your processing cluster. The base classes for user code, such as `DoFn`, `CombineFn`, and `WindowFn`, already implement `Serializable`; however, your subclass must not add any non-serializable members.

Some other serializability factors you should keep in mind are:

- Transient fields in your function object are *not* transmitted to worker instances, because they are not automatically serialized.
- Avoid loading a field with a large amount of data before serialization.
- Individual instances of your function object cannot share data.
- Mutating a function object after it gets applied will have no effect.
- Take care when declaring your function object inline by using an anonymous inner class instance. In a non-static context, your inner class instance will implicitly contain a pointer to the enclosing class and that class' state. That enclosing class will also be serialized, and thus the same considerations that apply to the function object itself also apply to this outer class.

4.3.2. Thread-compatibility

Your function object should be thread-compatible. Each instance of your function object is accessed by a single thread on a worker instance, unless you explicitly create your own threads. Note, however, that **the Beam SDKs are not thread-safe**. If you create your own threads in your user code, you must provide your own synchronization. Note that static members in your function object are not passed to worker instances and that multiple instances of your function may be accessed from different threads.

4.3.3. Idempotence

It's recommended that you make your function object idempotent–that is, that it can be repeated or retried as often as necessary without causing unintended side effects. The Beam model provides no guarantees as to the number of times your user code might be invoked or retried; as such, keeping your function object idempotent keeps your pipeline's output deterministic, and your transforms' behavior more predictable and easier to debug.

4.4. Side inputs

In addition to the main input `PCollection`, you can provide additional inputs to a `ParDo` transform in the form of side inputs. A side input is an additional input that your `DoFn` can access each time it processes an element in the input `PCollection`. When you specify a side input, you create a view of some other data that can be read from within the `ParDo` transform's `DoFn` while procesing each element.

Side inputs are useful if your `ParDo` needs to inject additional data when processing each element in the input `PCollection`, but the additional data needs to be determined at runtime (and not hard-coded). Such values might be determined by the input data, or depend on a different branch of your pipeline.

4.4.1. Passing side inputs to ParDo

Java | Python

```
  // Pass side inputs to your ParDo transform by invoking .withSideInputs.
  // Inside your DoFn, access the side input by using the method DoFn.ProcessContext.sideIn
put.

  // The input PCollection to ParDo.
  PCollection<String> words = ...;

  // A PCollection of word lengths that we'll combine into a single value.
  PCollection<Integer> wordLengths = ...; // Singleton PCollection

  // Create a singleton PCollectionView from wordLengths using Combine.globally and View.as
Singleton.
  final PCollectionView<Integer> maxWordLengthCutOffView =
     wordLengths.apply(Combine.globally(new Max.MaxIntFn()).asSingletonView());


  // Apply a ParDo that takes maxWordLengthCutOffView as a side input.
  PCollection<String> wordsBelowCutOff =
  words.apply(ParDo
      .of(new DoFn<String, String>() {
          public void processElement(ProcessContext c) {
            String word = c.element();
            // In our DoFn, access the side input.
            int lengthCutOff = c.sideInput(maxWordLengthCutOffView);
            if (word.length() <= lengthCutOff) {
              c.output(word);
            }
          }
      }).withSideInputs(maxWordLengthCutOffView)
  );
```

4.4.2. Side inputs and windowing

A windowed `PCollection` may be infinite and thus cannot be compressed into a single value (or single collection class). When you create a `PCollectionView` of a windowed `PCollection`, the `PCollectionView` represents a single entity per window (one singleton per window, one list per window, etc.).

Beam uses the window(s) for the main input element to look up the appropriate window for the side input element. Beam projects the main input element's window into the side input's window set, and then uses the side input from the resulting window. If the main input and side inputs have identical windows, the projection provides the exact corresponding window. However, if the inputs have different windows, Beam uses the projection to choose the most appropriate side input window.

For example, if the main input is windowed using fixed-time windows of one minute, and the side input is windowed using fixed-time windows of one hour, Beam projects the main input window against the side input window set and selects the side input value from the appropriate hour-long side input window.

If the main input element exists in more than one window, then `processElement` gets called multiple times, once for each window. Each call to `processElement` projects the "current" window for the main input element, and thus might provide a different view of the side input each time.

If the side input has multiple trigger firings, Beam uses the value from the latest trigger firing. This is particularly useful if you use a side input with a single global window and specify a trigger.

4.5. Additional outputs

While `ParDo` always produces a main output `PCollection` (as the return value from `apply`), you can also have your `ParDo` produce any number of additional output `PCollection`s. If you choose to have multiple outputs, your `ParDo` returns all of the output `PCollection`s (including the main output) bundled together.

4.5.1. Tags for multiple outputs

Java | Python

```
// To emit elements to multiple output PCollections, create a TupleTag object to identify e
ach collection
// that your ParDo produces. For example, if your ParDo produces three output PCollections
 (the main output
// and two additional outputs), you must create three TupleTags. The following example code
 shows how to
// create TupleTags for a ParDo with three output PCollections.

  // Input PCollection to our ParDo.
  PCollection<String> words = ...;

  // The ParDo will filter words whose length is below a cutoff and add them to
  // the main ouput PCollection<String>.
  // If a word is above the cutoff, the ParDo will add the word length to an
  // output PCollection<Integer>.
  // If a word starts with the string "MARKER", the ParDo will add that word to an
  // output PCollection<String>.
  final int wordLengthCutOff = 10;

  // Create three TupleTags, one for each output PCollection.
  // Output that contains words below the length cutoff.
  final TupleTag<String> wordsBelowCutOffTag =
      new TupleTag<String>(){};
  // Output that contains word lengths.
  final TupleTag<Integer> wordLengthsAboveCutOffTag =
      new TupleTag<Integer>(){};
  // Output that contains "MARKER" words.
  final TupleTag<String> markedWordsTag =
      new TupleTag<String>(){};

// Passing Output Tags to ParDo:
// After you specify the TupleTags for each of your ParDo outputs, pass the tags to your Pa
rDo by invoking
// .withOutputTags. You pass the tag for the main output first, and then the tags for any a
dditional outputs
// in a TupleTagList. Building on our previous example, we pass the three TupleTags for our
 three output
// PCollections to our ParDo. Note that all of the outputs (including the main output PColl
ection) are
// bundled into the returned PCollectionTuple.

  PCollectionTuple results =
      words.apply(ParDo
          .of(new DoFn<String, String>() {
            // DoFn continues here.
            ...
          })
          // Specify the tag for the main output.
          .withOutputTags(wordsBelowCutOffTag,
          // Specify the tags for the two additional outputs as a TupleTagList.
                          TupleTagList.of(wordLengthsAboveCutOffTag)
                                      .and(markedWordsTag)));
```

4.5.2. Emitting to multiple outputs in your DoFn

Java | Python

```
// Inside your ParDo's DoFn, you can emit an element to a specific output PCollection by pa
ssing in the
// appropriate TupleTag when you call ProcessContext.output.
// After your ParDo, extract the resulting output PCollections from the returned PCollectio
nTuple.
// Based on the previous example, this shows the DoFn emitting to the main output and two a
dditional outputs.

  .of(new DoFn<String, String>() {
     public void processElement(ProcessContext c) {
       String word = c.element();
       if (word.length() <= wordLengthCutOff) {
         // Emit short word to the main output.
         // In this example, it is the output with tag wordsBelowCutOffTag.
         c.output(word);
       } else {
         // Emit long word length to the output with tag wordLengthsAboveCutOffTag.
         c.output(wordLengthsAboveCutOffTag, word.length());
       }
       if (word.startsWith("MARKER")) {
         // Emit word to the output with tag markedWordsTag.
         c.output(markedWordsTag, word);
       }
     }}));
```

4.6. Composite transforms

Transforms can have a nested structure, where a complex transform performs multiple simpler transforms (such as more than one `ParDo`, `Combine`, `GroupByKey`, or even other composite transforms). These transforms are called composite transforms. Nesting multiple transforms inside a single composite transform can make your code more modular and easier to understand.

The Beam SDK comes packed with many useful composite transforms. See the API reference pages for a list of transforms:

- Pre-written Beam transforms for Java (/documentation/sdks/javadoc/2.1.0/index.html?org/apache/beam/sdk/transforms/package-summary.html)
- Pre-written Beam transforms for Python (/documentation/sdks/pydoc/2.1.0/apache_beam.transforms.html)

4.6.1. An example composite transform

The `CountWords` transform in the WordCount example program (/get-started/wordcount-example/) is an example of a composite transform. `CountWords` is a `PTransform` subclass that consists of multiple nested transforms.

In its `expand` 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 Beam 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.

Note that this is also an example of nested composite transforms, as `Count` is, by itself, a composite transform.

Your composite transform's parameters and return value must match the initial input type and final return type for the entire transform, even if the transform's intermediate data changes type multiple times.

Java | Python

```
public static class CountWords extends PTransform<PCollection<String>,
    PCollection<KV<String, Long>>> {
  @Override
  public PCollection<KV<String, Long>> expand(PCollection<String> lines) {

    // Convert lines of text into individual words.
    PCollection<String> words = lines.apply(
        ParDo.of(new ExtractWordsFn()));

    // Count the number of times each word occurs.
    PCollection<KV<String, Long>> wordCounts =
        words.apply(Count.<String>perElement());

    return wordCounts;
  }
}
```

4.6.2. Creating a composite transform

To create your own composite transform, create a subclass of the `PTransform` class and override the `expand` method to specify the actual processing logic. You can then use this transform just as you would a built-in transform from the Beam SDK.

For the `PTransform` class type parameters, you pass the `PCollection` types that your transform takes as input, and produces as output. To take multiple `PCollection` s as input, or produce multiple `PCollection` s as output, use one of the multi-collection types 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:

Java Python

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

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

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

Java Python

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

As long as you override the `expand` 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 Beam SDK libraries.

Note: The `expand` method of a `PTransform` is not meant to be invoked directly by the user of a transform. Instead, you should call the `apply` method on the `PCollection` itself, with the transform as an argument. This allows transforms to be nested within the structure of your pipeline.

4.6.3. PTransform Style Guide

The PTransform Style Guide (/contribute/ptransform-style-guide/) contains additional information not included here, such as style guidelines, logging and testing guidance, and language-specific considerations. The guide is a useful starting point when you want to write new composite PTransforms.

5. Pipeline I/O

When you create a pipeline, you often need to read data from some external source, such as a file in external data sink or a database. Likewise, you may want your pipeline to output its result data to a similar external data sink. Beam provides read and write transforms for a number of common data storage types (/documentation/io/built-in/). If you want your pipeline to read from or write to a data storage format that isn't supported by the built-in transforms, you can implement your own read and write transforms (/documentation/io/io-toc/).

5.1. Reading input data

Read transforms read data from an external source and return a `PCollection` representation of the data for use by your pipeline. You can use a read transform at any point while constructing your pipeline to create a new `PCollection`, though it will be most common at the start of your pipeline.

Java | Python

```
PCollection<String> lines = p.apply(TextIO.read().from("gs://some/inputData.txt"));
```

5.2. Writing output data

Write transforms write the data in a `PCollection` to an external data source. You will most often use write transforms at the end of your pipeline to output your pipeline's final results. However, you can use a write transform to output a `PCollection`'s data at any point in your pipeline.

Java | Python

```
output.apply(TextIO.write().to("gs://some/outputData"));
```

5.3. File-based input and output data

5.3.1. Reading from multiple locations

Many read transforms support reading from multiple input files matching a glob operator you provide. Note that glob operators are filesystem-specific and obey filesystem-specific consistency models. The following TextIO example uses a glob operator (*) to read all matching input files that have prefix "input-" and the suffix ".csv" in the given location:

Java | Python

```
p.apply(“ReadFromText”,
    TextIO.read().from("protocol://my_bucket/path/to/input-*.csv");
```

To read data from disparate sources into a single `PCollection`, read each one independently and then use the Flatten transform to create a single `PCollection`.

5.3.2. Writing to multiple output files

For file-based output data, write transforms write to multiple output files by default. When you pass an output file name to a write transform, the file name is used as the prefix for all output files that the write transform produces. You can append a suffix to each output file by specifying a suffix.

The following write transform example writes multiple output files to a location. Each file has the prefix “numbers”, a numeric tag, and the suffix “.csv”.

Java | Python

```
records.apply("WriteToText",
    TextIO.write().to("protocol://my_bucket/path/to/numbers")
                .withSuffix(".csv"));
```

5.4. Beam-provided I/O transforms

See the Beam-provided I/O Transforms (/documentation/io/built-in/) page for a list of the currently available I/O transforms.

6. Data encoding and type safety

When Beam runners execute your pipeline, they often need to materialize the intermediate data in your `PCollection`s, which requires converting elements to and from byte strings. The Beam SDKs use objects called `Coder`s to describe how the elements of a given `PCollection` may be encoded and decoded.

> Note that coders are unrelated to parsing or formatting data when interacting with external data sources or sinks. Such parsing or formatting should typically be done explicitly, using transforms such as `ParDo` or `MapElements`.

In the Beam SDK for Java, the type `Coder` provides the methods required for encoding and decoding data. The SDK for Java provides a number of Coder subclasses that work with a variety of standard Java types, such as Integer, Long, Double, StringUtf8 and more. You can find all of the available Coder subclasses in the Coder package (https://github.com/apache/beam/tree/master/sdks/java/core/src/main/java/org/apache/beam/sdk/coders).

> Note that coders do not necessarily have a 1:1 relationship with types. For example, the Integer type can have multiple valid coders, and input and output data can use different Integer coders. A transform might have Integer-typed input data that uses BigEndianIntegerCoder, and Integer-typed output data that uses VarIntCoder.

6.1. Specifying coders

The Beam SDKs require a coder for every `PCollection` in your pipeline. In most cases, the Beam SDK is able to automatically infer a `Coder` for a `PCollection` based on its element type or the transform that produces it, however, in some cases the pipeline author will need to specify a `Coder` explicitly, or develop a `Coder` for their custom type.

You can explicitly set the coder for an existing `PCollection` by using the method `PCollection.setCoder`. Note that you cannot call `setCoder` on a `PCollection` that has been finalized (e.g. by calling `.apply` on it).

You can get the coder for an existing `PCollection` by using the method `getCoder`. This method will fail with an `IllegalStateException` if a coder has not been set and cannot be inferred for the given `PCollection`.

Beam SDKs use a variety of mechanisms when attempting to automatically infer the `Coder` for a `PCollection`.

Each pipeline object has a `CoderRegistry`. The `CoderRegistry` represents a mapping of Java types to the default coders that the pipeline should use for `PCollection`s of each type.

By default, the Beam SDK for Java automatically infers the `Coder` for the elements of a `PCollection` produced by a `PTransform` using the type parameter from the transform's function object, such as `DoFn`. In the case of `ParDo`, for example, a `DoFn<Integer, String>` function object accepts an input element of type `Integer` and produces an output element of type `String`. In such a case, the SDK for Java will automatically infer the default `Coder` for the output `PCollection<String>` (in the default pipeline `CoderRegistry`, this is `StringUtf8Coder`).

> NOTE: If you create your `PCollection` from in-memory data by using the `Create` transform, you cannot rely on coder inference and default coders. `Create` does not have access to any typing information for its arguments, and may not be able to infer a coder if the argument list contains a value whose exact run-time class doesn't have a default coder registered.

When using `Create`, the simplest way to ensure that you have the correct coder is by invoking `withCoder` when you apply the `Create` transform.

6.2. Default coders and the CoderRegistry

Each Pipeline object has a `CoderRegistry` object, which maps language types to the default coder the pipeline should use for those types. You can use the `CoderRegistry` yourself to look up the default coder for a given type, or to register a new default coder for a given type.

`CoderRegistry` contains a default mapping of coders to standard Java types for any pipeline you create using the Beam SDK for Java. The following table shows the standard mapping:

Java Type	Default Coder
Double	DoubleCoder
Instant	InstantCoder
Integer	VarIntCoder
Iterable	IterableCoder
KV	KvCoder
List	ListCoder
Map	MapCoder
Long	VarLongCoder
String	StringUtf8Coder
TableRow	TableRowJsonCoder
Void	VoidCoder
byte[]	ByteArrayCoder
TimestampedValue	TimestampedValueCoder

6.2.1. Looking up a default coder

You can use the method `CoderRegistry.getDefaultCoder` to determine the default Coder for a Java type. You can access the `CoderRegistry` for a given pipeline by using the method `Pipeline.getCoderRegistry`. This allows you to determine (or set) the default Coder for a Java type on a per-pipeline basis: i.e. "for this pipeline, verify that Integer values are encoded using `BigEndianIntegerCoder`."

6.2.2. Setting the default coder for a type

To set the default Coder for a Java type for a particular pipeline, you obtain and modify the pipeline's `CoderRegistry`. You use the method `Pipeline.getCoderRegistry` to get the `CoderRegistry` object, and then use the method `CoderRegistry.registerCoder` to register a new `Coder` for the target type.

The following example code demonstrates how to set a default Coder, in this case `BigEndianIntegerCoder`, for Integer values for a pipeline.

Java | Python

```
PipelineOptions options = PipelineOptionsFactory.create();
Pipeline p = Pipeline.create(options);

CoderRegistry cr = p.getCoderRegistry();
cr.registerCoder(Integer.class, BigEndianIntegerCoder.class);
```

6.2.3. Annotating a custom data type with a default coder

If your pipeline program defines a custom data type, you can use the `@DefaultCoder` annotation to specify the coder to use with that type. For example, let's say you have a custom data type for which you want to use `SerializableCoder`. You can use the `@DefaultCoder` annotation as follows:

```
@DefaultCoder(AvroCoder.class)
public class MyCustomDataType {
  ...
}
```

If you've created a custom coder to match your data type, and you want to use the `@DefaultCoder` annotation, your coder class must implement a static `Coder.of(Class<T>)` factory method.

```
public class MyCustomCoder implements Coder {
  public static Coder<T> of(Class<T> clazz) {...}
  ...
}

@DefaultCoder(MyCustomCoder.class)
public class MyCustomDataType {
  ...
}
```

7. Windowing

Windowing subdivides a `PCollection` according to the timestamps of its individual elements. Transforms that aggregate multiple elements, such as `GroupByKey` and `Combine`, work implicitly on a per-window basis — they process each `PCollection` as a succession of multiple, finite windows, though the entire collection itself may be of unbounded size.

A related concept, called **triggers**, determines when to emit the results of aggregation as unbounded data arrives. You can use triggers to refine the windowing strategy for your `PCollection`. Triggers allow you to deal with late-arriving data or to provide early results. See the triggers section for more information.

7.1. Windowing basics

Some Beam transforms, such as `GroupByKey` and `Combine`, group multiple elements by a common key. Ordinarily, that grouping operation groups all of the elements that have the same key within the entire data set. With an unbounded data set, it is impossible to collect all of the elements, since new elements are constantly being added and may be infinitely many (e.g. streaming data). If you are working with unbounded `PCollection`s, windowing is especially useful.

In the Beam model, any `PCollection` (including unbounded `PCollection`s) can be subdivided into logical windows. Each element in a `PCollection` is assigned to one or more windows according to the `PCollection`'s windowing function, and each individual window contains a finite number of elements. Grouping transforms then consider each `PCollection`'s elements on a per-window basis. `GroupByKey`, for example, implicitly groups the elements of a `PCollection` by *key and window*.

Caution: Beam's default windowing behavior is to assign all elements of a `PCollection` to a single, global window and discard late data, *even for unbounded `PCollection`s*. Before you use a grouping transform such as `GroupByKey` on an unbounded `PCollection`, you must do at least one of the following:

- Set a non-global windowing function. See Setting your PCollection's windowing function.
- Set a non-default trigger. This allows the global window to emit results under other conditions, since the default windowing behavior (waiting for all data to arrive) will never occur.

If you don't set a non-global windowing function or a non-default trigger for your unbounded `PCollection` and subsequently use a grouping transform such as `GroupByKey` or `Combine`, your pipeline will generate an error upon construction and your job will fail.

7.1.1. Windowing constraints

After you set the windowing function for a `PCollection`, the elements' windows are used the next time you apply a grouping transform to that `PCollection`. Window grouping occurs on an as-needed basis. If you set a windowing function using the `Window` transform, each element is assigned to a window, but the windows are not considered until `GroupByKey` or `Combine` aggregates across a window and key. This can have different effects on your pipeline. Consider the example pipeline in the figure below:

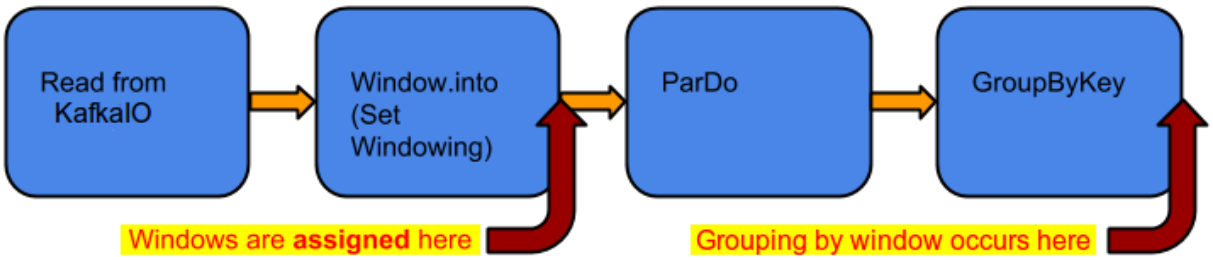


Figure: Pipeline applying windowing

In the above pipeline, we create an unbounded `PCollection` by reading a set of key/value pairs using `KafkaIO`, and then apply a windowing function to that collection using the `Window` transform. We then apply a `ParDo` to the the collection, and then later group the result of that `ParDo` using `GroupByKey`. The windowing function has no effect on the `ParDo` transform, because the windows are not actually used until they're needed for the `GroupByKey`. Subsequent transforms, however, are applied to the result of the `GroupByKey` – data is grouped by both key and window.

7.1.2. Using windowing with bounded PCollections

You can use windowing with fixed-size data sets in **bounded** `PCollection`s. However, note that windowing considers only the implicit timestamps attached to each element of a `PCollection`, and data sources that create fixed data sets (such as `TextIO`) assign the same timestamp to every element. This means that all the elements are by default part of a single, global window.

To use windowing with fixed data sets, you can assign your own timestamps to each element. To assign timestamps to elements, use a `ParDo` transform with a `DoFn` that outputs each element with a new timestamp (for example, the WithTimestamps (/documentation/sdks/javadoc/2.1.0/index.html?org/apache/beam/sdk/transforms/WithTimestamps.html) transform in the Beam SDK for Java).

To illustrate how windowing with a bounded `PCollection` can affect how your pipeline processes data, consider the following pipeline:

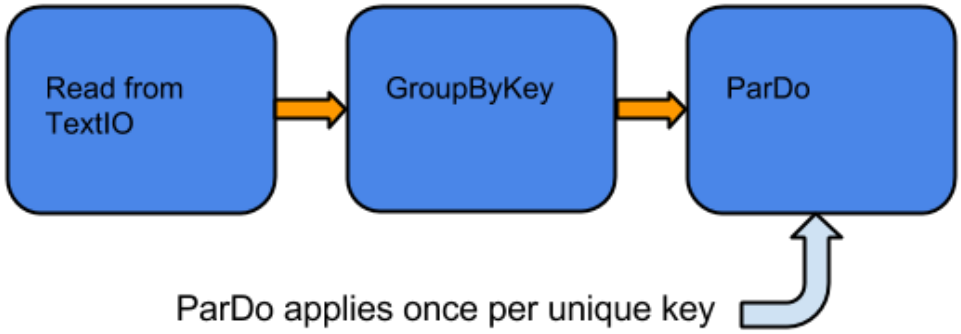


Figure: `GroupByKey` and `ParDo` without windowing, on a bounded collection.

In the above pipeline, we create a bounded `PCollection` by reading a set of key/value pairs using `TextIO`. We then group the collection using `GroupByKey`, and apply a `ParDo` transform to the grouped `PCollection`. In this example, the `GroupByKey` creates a collection of unique keys, and then `ParDo` gets applied exactly once per key.

Note that even if you don't set a windowing function, there is still a window – all elements in your `PCollection` are assigned to a single global window.

Now, consider the same pipeline, but using a windowing function:

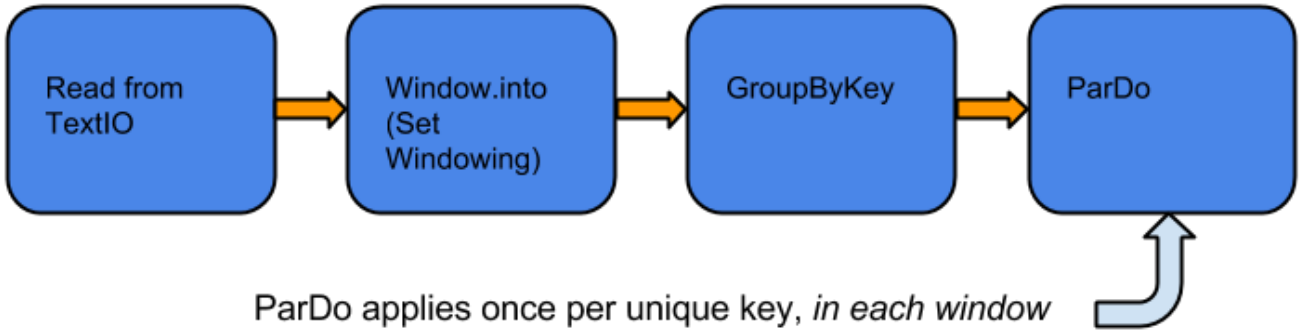


Figure: `GroupByKey` and `ParDo` with windowing, on a bounded collection.

As before, the pipeline creates a bounded `PCollection` of key/value pairs. We then set a windowing function for that `PCollection`. The `GroupByKey` transform groups the elements of the `PCollection` by both key and window, based on the windowing function. The subsequent `ParDo` transform gets applied multiple times per key, once for each window.

7.2. Provided windowing functions

You can define different kinds of windows to divide the elements of your `PCollection`. Beam provides several windowing functions, including:

- Fixed Time Windows
- Sliding Time Windows
- Per-Session Windows
- Single Global Window
- Calendar-based Windows (not supported by the Beam SDK for Python)

You can also define your own `WindowFn` if you have a more complex need.

Note that each element can logically belong to more than one window, depending on the windowing function you use. Sliding time windowing, for example, creates overlapping windows wherein a single element can be assigned to multiple windows.

7.2.1. Fixed time windows

The simplest form of windowing is using **fixed time windows**: given a timestamped `PCollection` which might be continuously updating, each window might capture (for example) all elements with timestamps that fall into a five minute interval.

A fixed time window represents a consistent duration, non overlapping time interval in the data stream. Consider windows with a five-minute duration: all of the elements in your unbounded `PCollection` with timestamp values from 0:00:00 up to (but not including) 0:05:00 belong to the first window, elements with timestamp values from 0:05:00 up to (but not including) 0:10:00 belong to the second window, and so on.

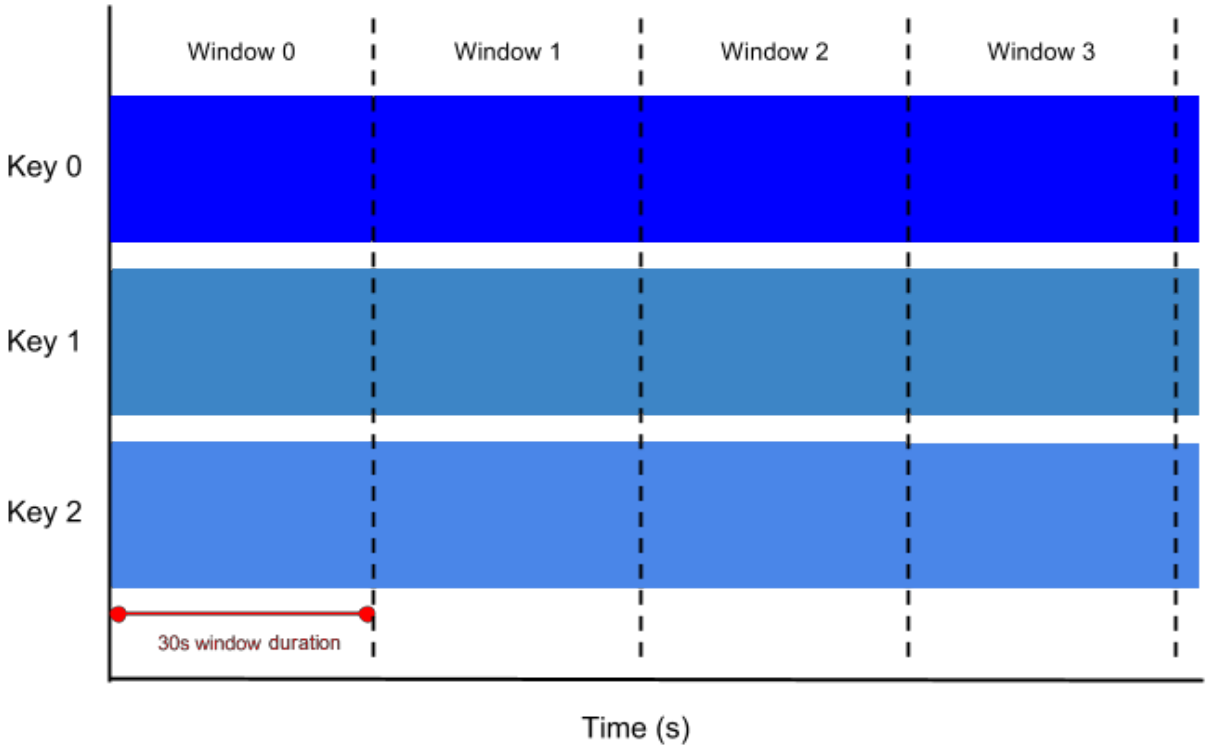


Figure: Fixed time windows, 30s in duration.

7.2.2. Sliding time windows

A **sliding time window** also represents time intervals in the data stream; however, sliding time windows can overlap. For example, each window might capture five minutes worth of data, but a new window starts every ten seconds. The frequency with which sliding windows begin is called the *period*. Therefore, our example would have a window *duration* of five minutes and a *period* of ten seconds.

Because multiple windows overlap, most elements in a data set will belong to more than one window. This kind of windowing is useful for taking running averages of data; using sliding time windows, you can compute a running average of the past five minutes' worth of data, updated every ten seconds, in our example.

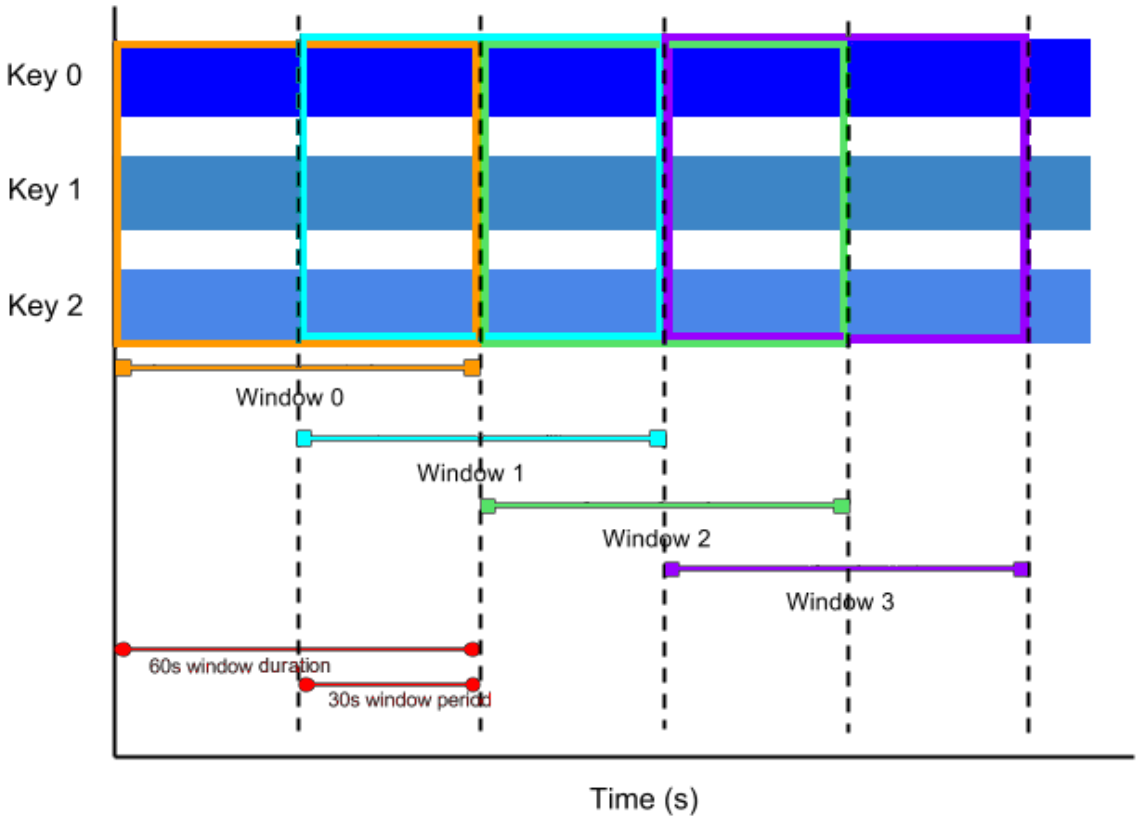


Figure: Sliding time windows, with 1 minute window duration and 30s window period.

7.2.3. Session windows

A **session window** function defines windows that contain elements that are within a certain gap duration of another element. Session windowing applies on a per-key basis and is useful for data that is irregularly distributed with respect to time. For example, a data stream representing user mouse activity may have long periods of idle time interspersed with high concentrations of clicks. If data arrives after the minimum specified gap duration time, this initiates the start of a new window.

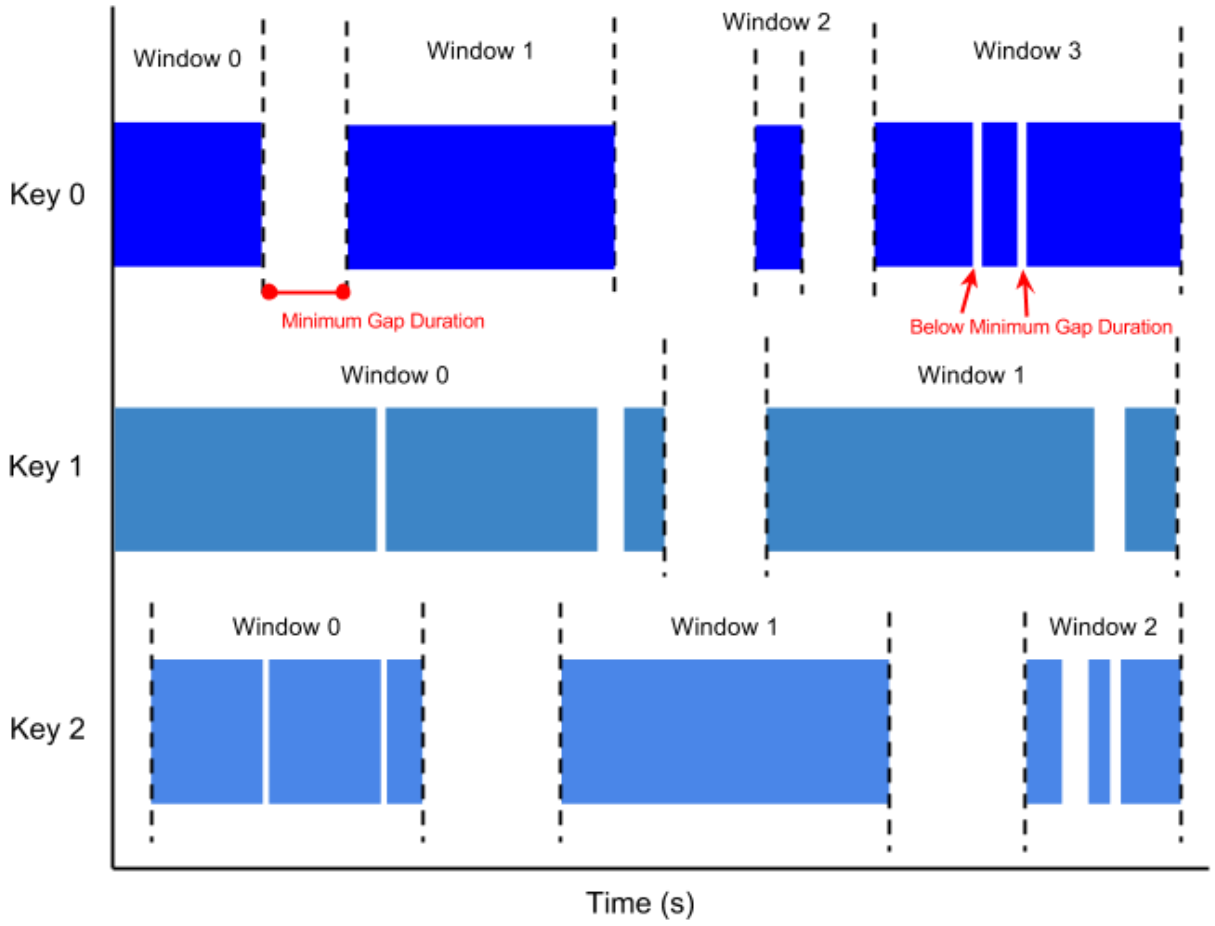


Figure: Session windows, with a minimum gap duration. Note how each data key has different windows, according to its data distribution.

7.2.4. The single global window

By default, all data in a `PCollection` is assigned to the single global window, and late data is discarded. If your data set is of a fixed size, you can use the global window default for your `PCollection`.

You can use the single global window if you are working with an unbounded data set (e.g. from a streaming data source) but use caution when applying aggregating transforms such as `GroupByKey` and `Combine`. The single global window with a default trigger generally requires the entire data set to be available before processing, which is not possible with continuously updating data. To perform aggregations on an unbounded `PCollection` that uses global windowing, you should specify a non-default trigger for that `PCollection`.

7.3. Setting your PCollection's windowing function

You can set the windowing function for a `PCollection` by applying the `Window` transform. When you apply the `Window` transform, you must provide a `WindowFn`. The `WindowFn` determines the windowing function your `PCollection` will use for subsequent grouping transforms, such as a fixed or sliding time window.

When you set a windowing function, you may also want to set a trigger for your `PCollection`. The trigger determines when each individual window is aggregated and emitted, and helps refine how the windowing function performs with respect to late data and computing early results. See the triggers section for more information.

7.3.1. Fixed-time windows

The following example code shows how to apply `Window` to divide a `PCollection` into fixed windows, each one minute in length:

Java Python

```
    PCollection<String> items = ...;
    PCollection<String> fixed_windowed_items = items.apply(
        Window.<String>into(FixedWindows.of(Duration.standardMinutes(1))));
```

7.3.2. Sliding time windows

The following example code shows how to apply `Window` to divide a `PCollection` into sliding time windows. Each window is 30 minutes in length, and a new window begins every five seconds:

Java Python

```
    PCollection<String> items = ...;
    PCollection<String> sliding_windowed_items = items.apply(
        Window.
<String>into(SlidingWindows.of(Duration.standardMinutes(30)).every(Duration.standardSeconds(5
```

7.3.3. Session windows

The following example code shows how to apply `Window` to divide a `PCollection` into session windows, where each session must be separated by a time gap of at least 10 minutes:

Java Python

```
    PCollection<String> items = ...;
    PCollection<String> session_windowed_items = items.apply(
        Window.<String>into(Sessions.withGapDuration(Duration.standardMinutes(10))));
```

Note that the sessions are per-key — each key in the collection will have its own session groupings depending on the data distribution.

7.3.4. Single global window

If your `PCollection` is bounded (the size is fixed), you can assign all the elements to a single global window. The following example code shows how to set a single global window for a `PCollection`:

Java Python

```
    PCollection<String> items = ...;
    PCollection<String> batch_items = items.apply(
        Window.<String>into(new GlobalWindows()));
```

7.4. Watermarks and late data

In any data processing system, there is a certain amount of lag between the time a data event occurs (the “event time”, determined by the timestamp on the data element itself) and the time the actual data element gets processed at any stage in your pipeline (the “processing time”, determined by the clock on the system processing the element). In addition, there are no guarantees that data events will appear in your pipeline in the same order that they were generated.

For example, let’s say we have a `PCollection` that’s using fixed-time windowing, with windows that are five minutes long. For each window, Beam must collect all the data with an *event time* timestamp in the given window range (between 0:00 and 4:59 in the first window, for instance). Data with timestamps outside that range (data from 5:00 or

later) belong to a different window.

However, data isn't always guaranteed to arrive in a pipeline in time order, or to always arrive at predictable intervals. Beam tracks a *watermark*, which is the system's notion of when all data in a certain window can be expected to have arrived in the pipeline. Data that arrives with a timestamp after the watermark is considered **late data**.

From our example, suppose we have a simple watermark that assumes approximately 30s of lag time between the data timestamps (the event time) and the time the data appears in the pipeline (the processing time), then Beam would close the first window at 5:30. If a data record arrives at 5:34, but with a timestamp that would put it in the 0:00-4:59 window (say, 3:38), then that record is late data.

Note: For simplicity, we've assumed that we're using a very straightforward watermark that estimates the lag time. In practice, your `PCollection` 's data source determines the watermark, and watermarks can be more precise or complex.

Beam's default windowing configuration tries to determines when all data has arrived (based on the type of data source) and then advances the watermark past the end of the window. This default configuration does *not* allow late data. Triggers allow you to modify and refine the windowing strategy for a `PCollection` . You can use triggers to decide when each individual window aggregates and reports its results, including how the window emits late elements.

7.4.1. Managing late data

> **Note:** Managing late data is not supported in the Beam SDK for Python.

You can allow late data by invoking the `.withAllowedLateness` operation when you set your `PCollection` 's windowing strategy. The following code example demonstrates a windowing strategy that will allow late data up to two days after the end of a window.

Java

```
PCollection<String> items = ...;
PCollection<String> fixed_windowed_items = items.apply(
    Window.<String>into(FixedWindows.of(Duration.standardMinutes(1)))
          .withAllowedLateness(Duration.standardDays(2)));
```

When you set `.withAllowedLateness` on a `PCollection` , that allowed lateness propagates forward to any subsequent `PCollection` derived from the first `PCollection` you applied allowed lateness to. If you want to change the allowed lateness later in your pipeline, you must do so explictly by applying `Window.configure().withAllowedLateness()` .

7.5. Adding timestamps to a PCollection's elements

An unbounded source provides a timestamp for each element. Depending on your unbounded source, you may need to configure how the timestamp is extracted from the raw data stream.

However, bounded sources (such as a file from `TextIO`) do not provide timestamps. If you need timestamps, you must add them to your `PCollection` 's elements.

You can assign new timestamps to the elements of a `PCollection` by applying a ParDo transform that outputs new elements with timestamps that you set.

An example might be if your pipeline reads log records from an input file, and each log record includes a timestamp field; since your pipeline reads the records in from a file, the file source doesn't assign timestamps automatically. You can parse the timestamp field from each record and use a `ParDo` transform with a `DoFn` to attach the timestamps to each element in your `PCollection` .

Java Python

```
PCollection<LogEntry> unstampedLogs = ...;
PCollection<LogEntry> stampedLogs =
    unstampedLogs.apply(ParDo.of(new DoFn<LogEntry, LogEntry>() {
      public void processElement(ProcessContext c) {
        // Extract the timestamp from log entry we're currently processing.
        Instant logTimeStamp = extractTimeStampFromLogEntry(c.element());
        // Use ProcessContext.outputWithTimestamp (rather than
        // ProcessContext.output) to emit the entry with timestamp attached.
        c.outputWithTimestamp(c.element(), logTimeStamp);
      }
    }));
```

8. Triggers

> **NOTE:** This content applies only to the Beam SDK for Java. The Beam SDK for Python does not support triggers.

When collecting and grouping data into windows, Beam uses **triggers** to determine when to emit the aggregated results of each window (referred to as a *pane*). If you use Beam's default windowing configuration and default trigger, Beam outputs the aggregated result when it estimates all data has arrived, and discards all subsequent data for that window.

You can set triggers for your `PCollection` s to change this default behavior. Beam provides a number of pre-built triggers that you can set:

- **Event time triggers**. These triggers operate on the event time, as indicated by the timestamp on each data element. Beam's default trigger is event time-based.
- **Processing time triggers**. These triggers operate on the processing time – the time when the data element is processed at any given stage in the pipeline.
- **Data-driven triggers**. These triggers operate by examining the data as it arrives in each window, and firing when that data meets a certain property. Currently, data-driven triggers only support firing after a certain number of data elements.
- **Composite triggers**. These triggers combine multiple triggers in various ways.

At a high level, triggers provide two additional capabilities compared to simply outputting at the end of a window:

- Triggers allow Beam to emit early results, before all the data in a given window has arrived. For example, emitting after a certain amouint of time elapses, or after a certain number of elements arrives.
- Triggers allow processing of late data by triggering after the event time watermark passes the end of the window.

These capabilities allow you to control the flow of your data and balance between different factors depending on your use case:

- **Completeness:** How important is it to have all of your data before you compute your result?
- **Latency:** How long do you want to wait for data? For example, do you wait until you think you have all data? Do you process data as it arrives?
- **Cost:** How much compute power/money are you willing to spend to lower the latency?

For example, a system that requires time-sensitive updates might use a strict time-based trigger that emits a window every *N* seconds, valuing promptness over data completeness. A system that values data completeness more than the exact timing of results might choose to use Beam's default trigger, which fires at the end of the window.

You can also set a trigger for an unbounded `PCollection` that uses a single global window for its windowing function. This can be useful when you want your pipeline to provide periodic updates on an unbounded data set — for example, a running average of all data provided to the present time, updated every N seconds or every N elements.

8.1. Event time triggers

The `AfterWatermark` trigger operates on *event time*. The `AfterWatermark` trigger emits the contents of a window after the watermark passes the end of the window, based on the timestamps attached to the data elements. The watermark is a global progress metric, and is Beam's notion of input completeness within your pipeline at any given point. `AfterWatermark.pastEndOfWindow()` *only* fires when the watermark passes the end of the window.

In addition, you can use `.withEarlyFirings(trigger)` and `.withLateFirings(trigger)` to configure triggers that fire if your pipeline receives data before or after the end of the window.

The following example shows a billing scenario, and uses both early and late firings:

Java | Python

```
// Create a bill at the end of the month.
AfterWatermark.pastEndOfWindow()
    // During the month, get near real-time estimates.
    .withEarlyFirings(
        AfterProcessingTime
            .pastFirstElementInPane()
            .plusDuration(Duration.standardMinutes(1))
    // Fire on any late data so the bill can be corrected.
    .withLateFirings(AfterPane.elementCountAtLeast(1))
```

8.1.1. The default trigger

The default trigger for a `PCollection` is based on event time, and emits the results of the window when the Beam's watermark passes the end of the window, and then fires each time late data arrives.

However, if you are using both the default windowing configuration and the default trigger, the default trigger emits exactly once, and late data is discarded. This is because the default windowing configuration has an allowed lateness value of 0. See the Handling Late Data section for information about modifying this behavior.

8.2. Processing time triggers

The `AfterProcessingTime` trigger operates on *processing time*. For example, the `AfterProcessingTime.pastFirstElementInPane()` trigger emits a window after a certain amount of processing time has passed since data was received. The processing time is determined by the system clock, rather than the data element's timestamp.

The `AfterProcessingTime` trigger is useful for triggering early results from a window, particularly a window with a large time frame such as a single global window.

8.3. Data-driven triggers

Beam provides one data-driven trigger, `AfterPane.elementCountAtLeast()`. This trigger works on an element count; it fires after the current pane has collected at least *N* elements. This allows a window to emit early results (before all the data has accumulated), which can be particularly useful if you are using a single global window.

It is important to note that if, for example, you use `.elementCountAtLeast(50)` and only 32 elements arrive, those 32 elements sit around forever. If the 32 elements are important to you, consider using composite triggers to combine multiple conditions. This allows you to specify multiple firing conditions such as "fire either when I receive 50 elements, or every 1 second".

8.4. Setting a trigger

When you set a windowing function for a `PCollection` by using the `Window` transform, you can also specify a trigger.

You set the trigger(s) for a `PCollection` by invoking the method `.triggering()` on the result of your `Window.into()` transform, as follows:

Java | Python

```
  PCollection<String> pc = ...;
  pc.apply(Window.<String>into(FixedWindows.of(1, TimeUnit.MINUTES))
                                .triggering(AfterProcessingTime.pastFirstElementInPane()
                                                               .plusDelayOf(Duration.standar
dMinutes(1)))
                                .discardingFiredPanes());
```

This code sample sets a time-based trigger for a `PCollection`, which emits results one minute after the first element in that window has been processed. The last line in the code sample, `.discardingFiredPanes()`, is the window's **accumulation mode**.

8.4.1. Window accumulation modes

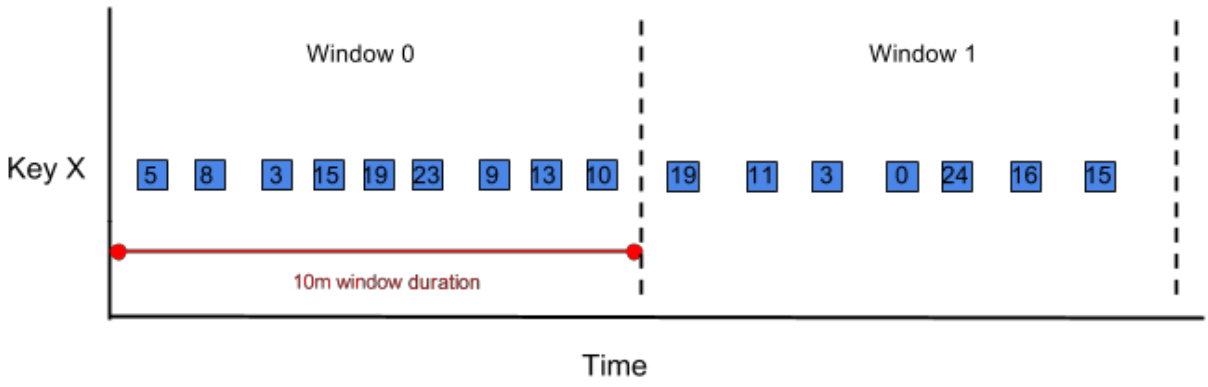
When you specify a trigger, you must also set the the window's **accumulation mode**. When a trigger fires, it emits the current contents of the window as a pane. Since a trigger can fire multiple times, the accumulation mode determines whether the system *accumulates* the window panes as the trigger fires, or *discards* them.

To set a window to accumulate the panes that are produced when the trigger fires, invoke `.accumulatingFiredPanes()` when you set the trigger. To set a window to discard fired panes, invoke `.discardingFiredPanes()`.

Let's look an example that uses a `PCollection` with fixed-time windowing and a data-based trigger. This is something you might do if, for example, each window represented a ten-minute running average, but you wanted to display the current value of the average in a UI more frequently than every ten minutes. We'll assume the following conditions:

- The `PCollection` uses 10-minute fixed-time windows.
- The `PCollection` has a repeating trigger that fires every time 3 elements arrive.

The following diagram shows data events for key X as they arrive in the PCollection and are assigned to windows. To keep the diagram a bit simpler, we'll assume that the events all arrive in the pipeline in order.



8.4.1.1. Accumulating mode

If our trigger is set to `.accumulatingFiredPanes`, the trigger emits the following values each time it fires. Keep in mind that the trigger fires every time three elements arrive:

```
  First trigger firing:  [5, 8, 3]
  Second trigger firing: [5, 8, 3, 15, 19, 23]
  Third trigger firing:  [5, 8, 3, 15, 19, 23, 9, 13, 10]
```

8.4.1.2. Discarding mode

If our trigger is set to `.discardingFiredPanes`, the trigger emits the following values on each firing:

```
First trigger firing:  [5, 8, 3]
Second trigger firing:           [15, 19, 23]
Third trigger firing:                         [9, 13, 10]
```

8.4.2. Handling late data

If you want your pipeline to process data that arrives after the watermark passes the end of the window, you can apply an *allowed lateness* when you set your windowing configuration. This gives your trigger the opportunity to react to the late data. If allowed lateness is set, the default trigger will emit new results immediately whenever late data arrives.

You set the allowed lateness by using `.withAllowedLateness()` when you set your windowing function:

Java | Python

```
PCollection<String> pc = ...;
pc.apply(Window.<String>into(FixedWindows.of(1, TimeUnit.MINUTES))
                            .triggering(AfterProcessingTime.pastFirstElementInPane()
                                                           .plusDelayOf(Duration.standard
Minutes(1)))
                            .withAllowedLateness(Duration.standardMinutes(30));
```

This allowed lateness propagates to all `PCollection` s derived as a result of applying transforms to the original `PCollection`. If you want to change the allowed lateness later in your pipeline, you can apply `Window.configure().withAllowedLateness()` again, explicitly.

8.5. Composite triggers

You can combine multiple triggers to form **composite triggers**, and can specify a trigger to emit results repeatedly, at most once, or under other custom conditions.

8.5.1. Composite trigger types

Beam includes the following composite triggers:

- You can add additional early firings or late firings to `AfterWatermark.pastEndOfWindow` via `.withEarlyFirings` and `.withLateFirings`.
- `Repeatedly.forever` specifies a trigger that executes forever. Any time the trigger's conditions are met, it causes a window to emit results and then resets and starts over. It can be useful to combine `Repeatedly.forever` with `.orFinally` to specify a condition that causes the repeating trigger to stop.
- `AfterEach.inOrder` combines multiple triggers to fire in a specific sequence. Each time a trigger in the sequence emits a window, the sequence advances to the next trigger.
- `AfterFirst` takes multiple triggers and emits the first time *any* of its argument triggers is satisfied. This is equivalent to a logical OR operation for multiple triggers.
- `AfterAll` takes multiple triggers and emits when *all* of its argument triggers are satisfied. This is equivalent to a logical AND operation for multiple triggers.
- `orFinally` can serve as a final condition to cause any trigger to fire one final time and never fire again.

8.5.2. Composition with AfterWatermark.pastEndOfWindow

Some of the most useful composite triggers fire a single time when Beam estimates that all the data has arrived (i.e. when the watermark passes the end of the window) combined with either, or both, of the following:

- Speculative firings that precede the watermark passing the end of the window to allow faster processing of partial results.
- Late firings that happen after the watermark passes the end of the window, to allow for handling late-arriving data

You can express this pattern using `AfterWatermark.pastEndOfWindow`. For example, the following example trigger code fires on the following conditions:

- On Beam's estimate that all the data has arrived (the watermark passes the end of the window)
- Any time late data arrives, after a ten-minute delay
- After two days, we assume no more data of interest will arrive, and the trigger stops executing

Java Python

```
  .apply(Window
      .configure()
      .triggering(AfterWatermark
           .pastEndOfWindow()
           .withLateFirings(AfterProcessingTime
                .pastFirstElementInPane()
                .plusDelayOf(Duration.standardMinutes(10))))
      .withAllowedLateness(Duration.standardDays(2)));
```

8.5.3. Other composite triggers

You can also build other sorts of composite triggers. The following example code shows a simple composite trigger that fires whenever the pane has at least 100 elements, or after a minute.

Java Python

```
  Repeatedly.forever(AfterFirst.of(
      AfterPane.elementCountAtLeast(100),

AfterProcessingTime.pastFirstElementInPane().plusDelayOf(Duration.standardMinutes(1))))
```