## **Understanding Apache Beam**

Apache Beam is a unified model for defining both batch and streaming data-parallel processing that can build portable big data pipelines. The name Beam actually comes from combining **B**atch + Str**eam -> Beam**

Unified model as in beam exposes single API abstraction layer for both batch and stream processing jobs, unlike other frameworks with different APIs for batch and stream processing.

The Apache Beam programming model simplifies the mechanics of large-scale data processing. Using one of the Apache Beam SDKs, you build a program that defines the pipeline. Then, you execute the pipeline on a specific platform such as **Cloud** **Dataflow**. This model lets you concentrate on the logical composition of your data processing job, rather than managing the orchestration of parallel processing.

**Unique Features of Apache Beam**

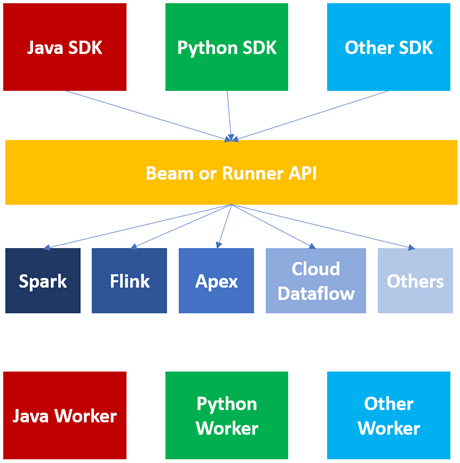
The unique features of the beam are as follows:

* **Unified** — Use a single programming model for both batch and streaming use cases.
* **Portable** — Execute pipelines in multiple execution environments. Here, execution environments mean different runners. Ex. Spark Runner, Dataflow Runner, Direct Runner etc
* **Extensible** — Write custom SDKs, IO connectors, and transformation libraries.

**Advantages of Apache Beam**

1. **Flexible Execution -** Apache Beam supports a variety of execution engines, including Apache Spark, Google Cloud Dataflow, and Apache Flink. This offers you the flexibility to choose the execution engine that best meets your needs.
2. **Unified Model -** Apache Beam aims to provide a unified model for batch and streaming data processing. This means that you can use the same Beam code to process data that is either coming in as a stream or that has already been collected into a batch. This can save you time and effort, as you don't need to learn two different sets of APIs.
3. **Portable Code -** Theoretically, Apache Beam code can be run on any execution engine without modification. This means that you can develop your code once and then run it on any platform that supports Apache Beam. This can save you time and money, as you don't need to develop and maintain separate versions of your code for different platforms.
4. **Scalable -** Apache Beam can scale to process large amounts of data. This is because Apache Beam uses a distributed architecture that can be scaled out to multiple machines. This can help you to process data more quickly and efficiently.
5. **Extensible -** Apache Beam is extensible with a variety of plugins and libraries. This means that you can add new features and functionality to Apache Beam to meet your specific needs.

**Disadvantages of Apache Beam**



**Basics of the Beam model**

1. **Pipeline -** A pipeline is a user-constructed graph of transformations that defines the desired data processing operations.
2. **PCollection -** A PCollection is a data set or data stream. The data that a pipeline processes is part of a PCollection.
3. **PTransform -** A PTransform (or transform) represents a data processing operation, or a step, in your pipeline. A transform is applied to zero or more PCollection objects, and produces zero or more PCollection objects.
4. **Aggregation -** Aggregation is computing a value from multiple (1 or more) input elements.
5. **User-defined function (UDF) -** Some Beam operations allow you to run user-defined code as a way to configure the transform.
6. **Schema -** A schema is a language-independent type definition for a PCollection. The schema for a PCollection defines elements of that PCollection as an ordered list of named fields.
7. **SDK -** A language-specific library that lets pipeline authors build transforms, construct their pipelines, and submit them to a runner.
8. **Runner -** A runner runs a Beam pipeline using the capabilities of your chosen data processing engine.
9. **Window -** A PCollection can be subdivided into windows based on the timestamps of the individual elements. Windows enable grouping operations over collections that grow over time by dividing the collection into windows of finite collections.
10. **Watermark -** A watermark is a guess as to when all data in a certain window is expected to have arrived. This is needed because data isn’t always guaranteed to arrive in a pipeline in time order, or to always arrive at predictable intervals.
11. **Trigger -** A trigger determines when to aggregate the results of each window.
12. **State and timers -** Per-key state and timer callbacks are lower-level primitives that give you full control over aggregating input collections that grow over time.
13. **Splittable DoFn -** Splittable DoFns let you process elements in a non-monolithic way. You can checkpoint the processing of an element, and the runner can split the remaining work to yield additional parallelism.

**Flow of Beam Programming Model**



A typical Beam driver program works as follows

1. **Create a** **Pipeline object** and set the **pipeline execution options**, including the **Pipeline Runner**.
2. **Create an initial PCollection** for pipeline data, either using the IOs to read data from an external storage system, or using a Create transform to build a PCollection from in-memory data.
3. **Apply PTransforms** 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 modifying the input collection.
4. Use **IOs** to write the final, transformed PCollection(s) to an external source.
5. Run the pipeline using the designated **Pipeline Runner**.

**Note -** A typical pipeline applies subsequent transforms to each new output PCollection in turn until the processing is complete. However, note that a pipeline does not have to be a single straight line of transforms applied one after another: think of PCollections as variables and PTransforms as functions applied to these variables: the shape of the pipeline can be an arbitrarily complex processing graph.

**Input -** text file, Bigquery, avro files, database, stream (kafka, google pub/sub) etc.

**Output -** text file, database, hdfs, gcs bucket, stream (kafka, google pub/sub) etc.

**PCollection**

A **PCollection** is a data set or data stream. The data that a pipeline processes is part of a PCollection. It is an abstraction represents a potentially distributed, multi-element data set.

In beam, the data must be in the form of a PCollection. Beam uses these PCollection to perform transformation on top of that PCollection data which is basically an input or output to a PTransfrom.

Multiple pipelines cannot share a PCollection. Beam pipelines process PCollection, and the runner is responsible for storing these elements. A PCollection generally contains “big data” (too much data to fit in memory on a single machine).

***A PCollection can be either bounded or unbounded -***

A **bounded PCollection** is a dataset of a known, fixed size (alternatively, a dataset that is not growing over time). Bounded data are batch data which can be processed by batch pipelines.

An **unbounded PCollection** is a dataset that grows over time, and the elements are processed as they arrive. Unbounded data are streaming data which must be processed by streaming pipelines.

***PCollection Characteristics*** -

* **Immutable** - PCollections are immutable in nature i.e., applying a transformation on a pcollection results in a creation of new pcollection.
* **Element Type** – The elements in pcollection may be of any type, but all must be of same type.
* **Timestamps** – Each element in pcollection has an associated timestamp with it.
* **Operation Type** – PCollection does not support grained operations i.e., transformations cannot be applied on a specific element in pcollection.
* **No Random Access** – Cannot access data using indexes or some specific element. No size limit.
* **Coder** - Every PCollection has a coder, which is a specification of the binary format of the elements. Each PCollection has a declared encoding for its elements, called a ***coder***. A coder has a URN that identifies the encoding, and might have additional sub-coders.
* **Ownership –** PCollection is only owned by a specific Pipeline Object for which it is created, multiple pipelines cannot share a PCollection.

**PTransform**

A **PTransform** (or transform) represents a data processing operation, or a step, in your pipeline. A transform is usually applied to one or more input PCollection objects.

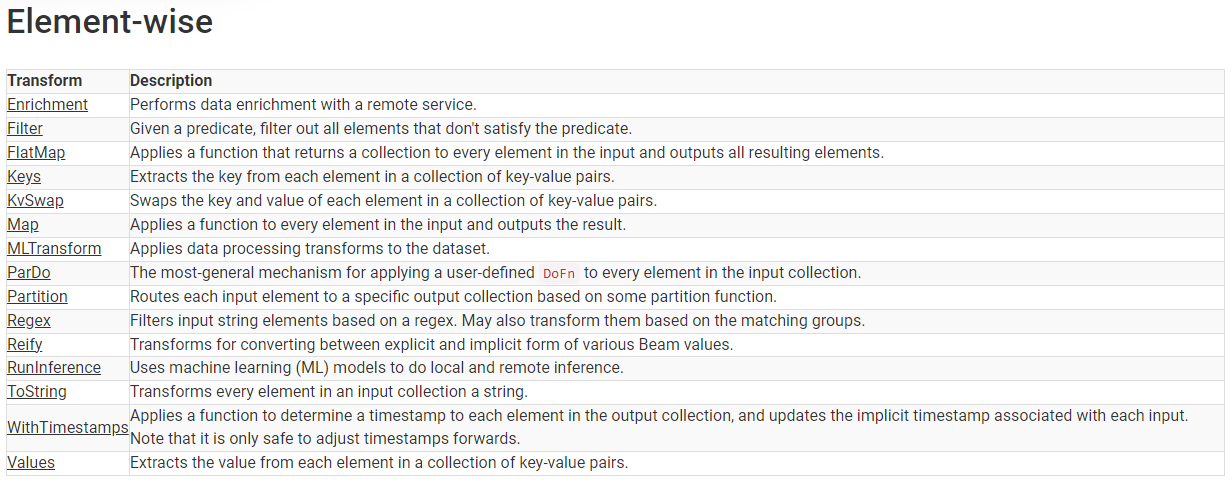
Transforms in Beam are represented by the PTransform object which takes an Input (PCollection) and applies transformations of some kind on the Input and produces an output which is another PCollection object.

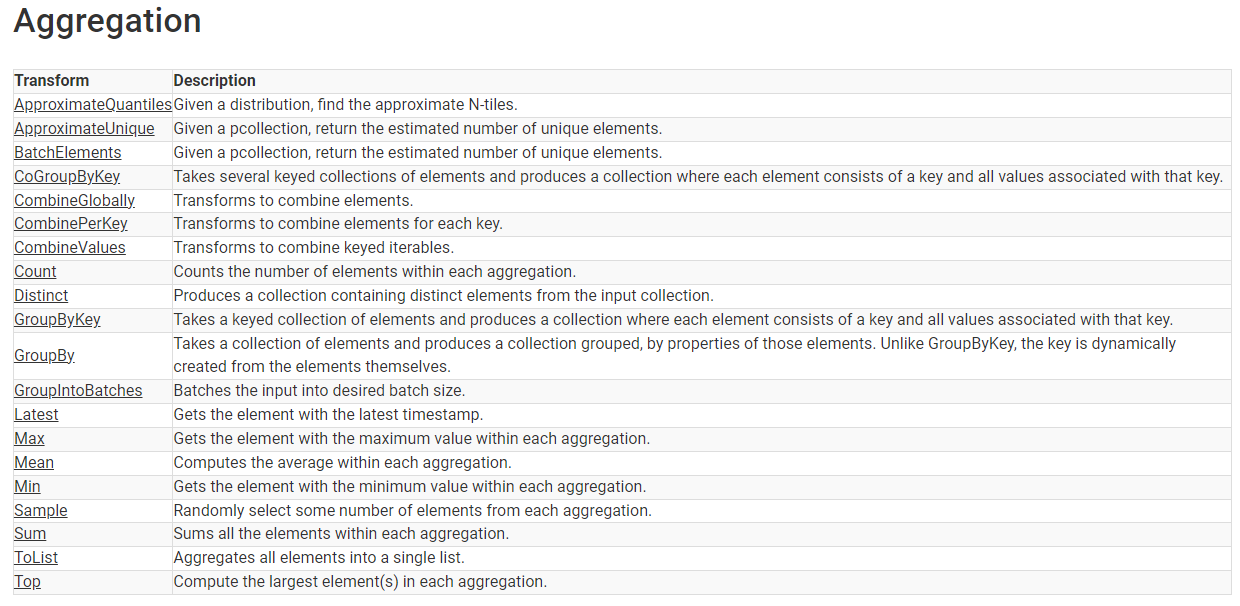
The most common types of transformations are categorized into Element-Wise Transformation and Aggregation. The element-wise transformations are the most common ones like Filter, FlatMap, Keys, Map, ParDo etc. Aggregation transforms are GroupBy, GroupByKey, Min, Sum, Combines, Count, etc.

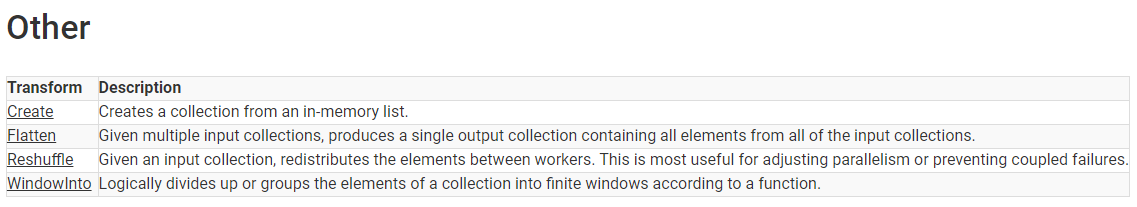
***Transform can be of two kinds –***

1. **Element-wise Transform**
2. **Aggregation Transform**

**Types of Transforms**







**Element-Wise Transforms**

1. **ParDo**

ParDo is a core beam transform for generic parallel processing that takes each element in PCollection and performs a user-defined function (udf) on it which is ‘DoFn’ and produces the result.

A **ParDo Transform** considers each element in the Input PCollection, performs some processing function (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.

**DoFn (DoFunction Object)**

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 DoFns - they’re what define your pipeline’s exact data processing tasks.

* When we apply a ParDo Transform, we will need to provide user code in the form of a DoFn object.
* DoFn is a Beam SDK class that defines a **Distributed Processing Function.**
* Inside our DoFn subclass, we will write a method process where we provide the actual processing logic.
* We don’t need to manually extract the elements from input collection, the Beam SDKs handle that for us.
* Our process method should accept an argument element, which is the input element, and return an iterable with its output value.
* We can accomplish this by emitting individual elements with yield statements, or can also use a return statement with an iterable, like a list or generator.

1. **Map**

* **Map** is a beam transform which applies a simple 1-to-1 mapping function over each element in the collection. The Map function will be applied to each element in the input and produces a result.
* It works like **ParDo**, applied **Map** in multiple ways to transform every element in PCollection.
* Map accepts a function that returns a single element for every input element in the PCollection.

1. **Map Tuple**

* If a PCollection consists of (key, value) pairs, we can use MapTuple to unpack them into different function arguments.

1. **Filter**

* Given a predicate/logic, filter out all elements that don’t satisfy that predicate. May also be used to filter based on an inequality with a given value based on the comparison ordering of the element.

1. **Flat Map**

* Applies a **simple 1-to-many mapping** function over each element in the collection. The many elements are flattened into the resulting collection.

**Note**:

* Lambda - A lambda function is a small anonymous function. A lambda function can take any number of arguments, but can only have one expression.

1. **Keys**

* Takes a collection of key-value pairs and returns the key to each element.

1. **Values**

* Takes a collection of key-value pairs and returns the value of each element.

1. **KVSwap**

* Takes a collection of key-value pairs and returns a collection of key-value pairs which has each key and value swapped.

1. **ToString**

* Transforms every element in an input collection to a string.
* Any non-string element can be converted to a string using standard Python functions and methods.
* Many I/O transforms, such as **textio.WriteToText**, expect their input elements to be strings.
  + Key-value pairs to string
  + Elements to string
  + Iterables to string

1. **Partition**

* Partition is a Beam transform for PCollection objects that store the same data type. It 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.
* Partition accepts a function that receives the number of partitions, and returns the index of the desired partition for the element. The number of partitions passed must be a positive integer, and it must return an integer in the range 0 to num\_partitions-1.

1. **Flatten**

* Flatten is a Beam transform for PCollection objects that store the same data type. Its kind of a union operation.
* Merges multiple PCollection objects into a single logical PCollection.

1. **Regex**
2. **Reify**
3. **Run Inference**
4. **ML Transform**
5. **Enrichment**

**References for element-wise transform:**

* <https://beam.apache.org/documentation/programming-guide/#pardo>
* <https://beam.apache.org/documentation/transforms/python/elementwise/pardo/>
* <https://beam.apache.org/documentation/transforms/python/elementwise/map/>
* <https://beam.apache.org/documentation/transforms/python/elementwise/flatmap/>
* <https://beam.apache.org/documentation/transforms/python/elementwise/filter/#example-2-filtering-with-a-lambda-function>
* <https://beam.apache.org/documentation/transforms/python/elementwise/keys/>
* <https://beam.apache.org/documentation/transforms/python/elementwise/values/>
* <https://beam.apache.org/documentation/transforms/python/elementwise/tostring/>
* <https://beam.apache.org/documentation/programming-guide/#flatten>
* [https://beam.apache.org/documentation/transforms/python/other/flatten/](https://beam.apache.org/documentation/transforms/python/elementwise/tostring/)
* <https://beam.apache.org/documentation/programming-guide/#partition>
* <https://beam.apache.org/documentation/transforms/python/elementwise/partition/>
* <https://beam.apache.org/documentation/transforms/python/elementwise/regex/>
* <https://beam.apache.org/documentation/transforms/python/elementwise/mltransform/>
* <https://beam.apache.org/documentation/transforms/python/elementwise/reify/>
* <https://beam.apache.org/documentation/transforms/python/elementwise/runinference/>
* <https://beam.apache.org/documentation/transforms/python/elementwise/enrichment/>

**Aggregation Transforms**

1. **GroupBy**

* Takes a collection of elements and produces a collection grouped, by properties of those elements.
* Unlike GroupByKey, the key is dynamically created from the elements themselves.

1. **GroupByKey**

* GroupByKey is a Beam transform for processing collections of key/value pairs.
* Takes a keyed collection of elements and produces a collection where each element consists of a key and all values associated with that key.
* It’s a parallel reduction operation, analogous to the Shuffle phase of a Map/Shuffle/Reduce-style algorithm.
* 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).

1. **CoGroupByKey**

* Aggregates all input elements by their key and allows downstream processing to consume all values associated with the key.
* While GroupByKey performs this operation over a single input collection and thus a single type of input values, CoGroupByKey operates over multiple input collections.
* **CoGroupByKey expects a dictionary of named keyed PCollections**, and produces elements joined by their keys. The values of each output element are dictionaries where the names correspond to the input dictionary, with lists of all the values found for that key.
* As a result, the result for each key is a tuple of the values associated with that key in each input collection.

1. **GroupIntoBatches**

* Batches the input into desired batch size.

**Difference between GroupBy, GroupByKey & CoGroupByKey**

* **GroupBy** performs the operation on simple collection data which don’t contain any key-value pair in it.
* **GroupByKey** performs the operations over a single input collection which have key-value pairs as data and thus a single type of input values.
* **CoGroupByKey** operates over multiple input collections which have key-value pairs as data.

1. **Latest**

* Gets the element with the latest timestamp.

1. **Max**

* Gets the element with the maximum value within each aggregation.

1. **Min**

* Gets the element with the minimum value within each aggregation.

1. **Mean**

* Transforms for computing the arithmetic mean of the elements in a collection, or the mean of the values associated with each key in a collection of key-value pairs.

1. **Sample**

* Transforms for taking samples of the elements in a collection, or samples of the values associated with each key in a collection of key-value pairs.

1. **Sum**

* Sums all the elements within each aggregation.

1. **Top**

* Transforms for finding the largest (or smallest) set of elements in a collection, or the largest (or smallest) set of values associated with each key in a collection of key-value pairs.

1. **Distinct**

* We use Distinct to get rid of duplicate elements, which outputs a PCollection of all the unique elements.

1. **Count**

* we apply Count to get the total number of elements in different ways.

**References for aggregation transforms:**

* <https://beam.apache.org/documentation/transforms/python/aggregation/groupby/>
* <https://beam.apache.org/documentation/programming-guide/#groupbykey>
* <https://beam.apache.org/documentation/transforms/python/aggregation/groupbykey/>
* <https://beam.apache.org/documentation/programming-guide/#cogroupbykey>
* <https://beam.apache.org/documentation/transforms/python/aggregation/cogroupbykey/>
* <https://beam.apache.org/documentation/transforms/python/aggregation/groupintobatches/>
* <https://beam.apache.org/documentation/transforms/python/aggregation/latest/>
* <https://beam.apache.org/documentation/transforms/python/aggregation/max/>
* <https://beam.apache.org/documentation/transforms/python/aggregation/min/>
* <https://beam.apache.org/documentation/transforms/python/aggregation/mean/>
* <https://beam.apache.org/documentation/transforms/python/aggregation/sample/>
* <https://beam.apache.org/documentation/transforms/python/aggregation/sum/>
* <https://beam.apache.org/documentation/transforms/python/aggregation/top/>
* <https://beam.apache.org/documentation/transforms/python/aggregation/distinct/>
* <https://beam.apache.org/documentation/transforms/python/aggregation/count/>

**Combine Core Transform**

1. **Combine**

* Combine is a Beam transform for combining collections of elements or values in your data.
* Combine has variants that work on entire PCollections, and some that combine the values for each key in PCollections 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.
* The Beam SDK also provides some pre-built combine functions for common numeric combination operations such as sum, min, and max.
* complex combination operations might require you to create a subclass of CombineFn that has an accumulation type distinct from the input/output type.

**Advanced combinations using CombineFn:**

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:

* **Create Accumulator -** creates a new “local” accumulator
* **Add Input -** adds an input element to an accumulator, returning the accumulator value.
* **Merge Accumulators -** merges several accumulators into a single accumulator; this is how data in multiple accumulators is combined before the final calculation.
* **Extract Output -** performs the final computation.

**Three types of Aggregator function is supported by beam. They are.**

* **CombineGlobally:** Combines all elements in a collection.
* **CombinePerKey:** Combines all elements for each key in a collection.
* **CombineValues:** Combines an iterable of values in a keyed collection of elements.

**References for combine core transform**

* <https://beam.apache.org/documentation/programming-guide/#combine>
* <https://beam.apache.org/documentation/transforms/python/aggregation/combineglobally/>
* <https://beam.apache.org/documentation/transforms/python/aggregation/combineperkey/>
* <https://beam.apache.org/documentation/transforms/python/aggregation/combinevalues/>

**Side Inputs and Additional Outputs**

1. **Side Inputs**

* A side input is an additional input that your DoFn can access each time it processes an element in the input PCollection.
* In addition to the main input PCollection, you can provide additional inputs to a ParDo transform in the form of side inputs.
* 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).
* Side inputs must be small in size and not as big as pcollection because it has to be kept in memory of each worker.
* Such values might be determined by the input data, or depend on a different branch of your pipeline.

1. **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 PCollections.
* If you choose to have multiple outputs, your ParDo returns all of the output PCollections (including the main output) bundled together.

**References for side inputs and additional outputs:**

* <https://beam.apache.org/documentation/patterns/side-inputs/>
* <https://beam.apache.org/documentation/programming-guide/#additional-outputs>
* <https://beam.apache.org/documentation/programming-guide/#side-inputs>

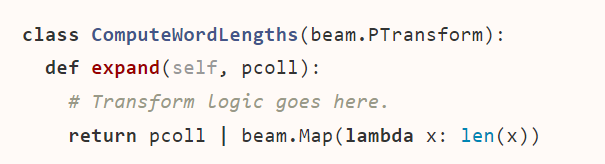
**Composite Transformation in Apache Beam**

**Composite Transform**

* 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.

**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.
* The transforms can include core transforms, composite transforms, or the transforms included in the Beam SDK libraries.
* The following code sample shows how to declare a PTransform that accepts a PCollection of Strings for input, and outputs a PCollection of Integers.



* 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.
* 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.
* 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.

**References for composite transform:**

* <https://beam.apache.org/documentation/programming-guide/#composite-transforms>
* <https://beam.apache.org/releases/pydoc/2.36.0/apache_beam.transforms.html>

**Other Transform**

1. **Create**

* Creates a collection containing a specified set of elements. This is useful for testing, as well as creating an initial input to process in parallel.
* For example, a single element to execute a one-time ParDo or a list of filenames to be read.

1. **Reshuffle**

* Adds a temporary random key to each element in a collection, reshuffles these keys, and removes the temporary key.
* This redistributes the elements between workers and returns a collection equivalent to its input collection. This is most useful for adjusting parallelism or preventing coupled failures.

1. **WindowInto**

* Logically divides up or groups the elements of a collection into finite windows according to a function.

1. **WithTimestamps**

* Assigns timestamps to all the elements of a collection.
* When windowing and late data play an important role in streaming pipelines, timestamps are especially useful.

**References for other transforms:**

* <https://beam.apache.org/documentation/transforms/python/other/create/>
* <https://beam.apache.org/documentation/transforms/python/other/reshuffle/>
* <https://beam.apache.org/documentation/transforms/python/other/windowinto/>
* <https://beam.apache.org/documentation/transforms/python/elementwise/withtimestamps/>

**Pipeline Options**

* To create a pipeline, we need to instantiate the pipeline object, eventually pass some options, and declaring the steps/transforms of the pipeline.
* 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.

|  |
| --- |
| import apache\_beam as beam  from apache\_beam.options.pipeline\_options import PipelineOptions  options = PipelineOptions()  p = beam.Pipeline(options=options) |

The PipelineOptions() method above is a command line parser that will read any standard option passed the following way:

|  |
| --- |
| **--<option>=<value>** |

**Custom options**

* You can also build your custom options. In this example I set an input and an output folder for my pipeline:

|  |
| --- |
| class MyOptions(PipelineOptions):  @classmethod  def \_add\_argparse\_args(cls, parser):  parser.add\_argument('--input',  help='Input for the pipeline',  default='./data/')  parser.add\_argument('--output',  help='Output for the pipeline',  default='./output/') |