Licensed to the Apache Software Foundation (ASF), Version 2.0 (the "License")

Show code

Getting started: *Tour of Beam*

Apache Beam is a library for parallel data processing.

Beam is commonly used for Extract-Transform-Load (ETL) jobs, where we *extract* data from a data source, *transform* that data, and *load* it into a data sink like a database. It does particularly well with large amounts of data since it can use mutliple machines to process everything at the same time.

Let's begin by installing the `apache-beam` package with `pip`.

```
# Install apache-beam with pip.
!pip install --quiet apache-beam
```

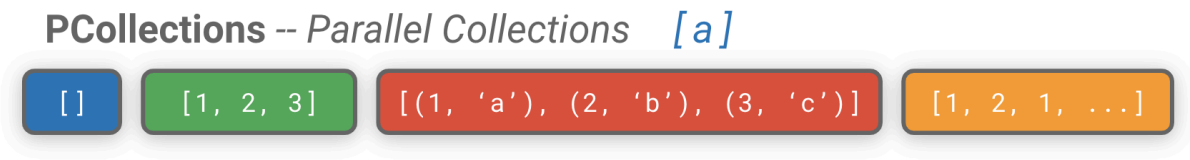
```
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 89.7/89.7 kB 2.6 MB/s eta 0:00:00
  Preparing metadata (setup.py) ... done
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 43.5/43.5 kB 4.3 MB/s eta 0:00:00
  Preparing metadata (setup.py) ... done
  Preparing metadata (setup.py) ... done
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 17.2/17.2 MB 67.8 MB/s eta 0:00:00
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 1.2/1.2 MB 77.9 MB/s eta 0:00:00
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 3.5/3.5 MB 130.1 MB/s eta 0:00:00
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 5.6/5.6 MB 100.9 MB/s eta 0:00:00
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 96.9/96.9 kB 11.4 MB/s eta 0:00:00
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 46.3/46.3 kB 4.6 MB/s eta 0:00:00
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 1.7/1.7 MB 95.6 MB/s eta 0:00:00
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 272.8/272.8 kB 27.1 MB/s eta 0:00:00
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 331.1/331.1 kB 30.7 MB/s eta 0:00:00
  Building wheel for crcmod (setup.py) ... done
  Building wheel for hdfs (setup.py) ... done
  Building wheel for docopt (setup.py) ... done
ERROR: pip's dependency resolver does not currently take into account all the packages that are installed. This behaviour is
grpcio-status 1.71.2 requires grpcio>=1.71.2, but you have grpcio 1.65.5 which is incompatible.
```

You can express a *data processing pipeline*, and then run it on the *runner* of your choice. For now, we use the `DirectRunner` which runs locally for simplicity.

What is a *pipeline*?

A **pipeline** is a **sequence of data transformations**. You can think of it like a production line, data comes in from one end, it gets transformed by each step. The outputs from one step are passed as inputs to the next step.

In Beam, your data lives in a **`PCollection`**, which stands for *Parallel Collection*. A `PCollection` is like a **list of elements**, but without any order guarantees. This allows Beam to easily parallelize and distribute the `PCollection`'s elements.



Once you have your data, the next step is to transform it. In Beam, you transform data using **`PTransform`**s, which stands for *Parallel Transform*. A `PTransform` is like a **function**, they take some inputs, transform them and create some outputs.

Now let's dive into creating our first pipeline.

For this first pipeline, let's just feed it some data from a Python list and print the results.

Each *step* in the pipeline is delimited by the *pipe operator* `|`. The outputs of each transform are passed to the next transform as inputs. And we can save the final results into a `PCollection` variable.

```
# We pass the elements from step1 through step3 and save the results into `outputs`.
outputs = pipeline | step1 | step2 | step3
```

Pipelines can quickly grow long, so it's sometimes easier to read if we surround them with parentheses and break them into multiple lines.

```
# This is equivalent to the example above.
outputs = (
  pipeline
  | step1
  | step2
  | step3
)
```

Also, Beam expects each transform, or step, to have a unique *label*, or description. This makes it a lot easier to debug, and it's in general a good practice to start. You can use the *right shift operator* `>>` to add a label to your transforms, like `'My description' >> MyTransform`.

```
# Try to give short but descriptive labels.
# These serve both as comments and help debug later on.
outputs = (
  pipeline
  | 'First step' >> step1
  | 'Second step' >> step2
  | 'Third step' >> step3
)
```

> ℹ The syntax might seem a little different at first, but you'll become familiar with it.

We use the `Create` transform to feed the pipeline with an `iterable` of elements, like a `list`.

Let's try to see what happens if we try to `print` a PCollection.

```
import apache_beam as beam

inputs = [0, 1, 2, 3]

# Create a pipeline.
with beam.Pipeline() as pipeline:
  # Feed it some input elements with `Create`.
  outputs = (
      pipeline
      | 'Create initial values' >> beam.Create(inputs)
  )

  # `outputs` is a PCollection with our input elements.
  # But printing it directly won't show us its contents :(
  print(f"outputs: {outputs}")
```

```
outputs: PCollection[[27]: Create initial values/Map(decode).None]
```

> ℹ In Beam, you can **NOT** access the elements from a `PCollection` directly like a Python list. This means, we can't simply `print` the output `PCollection` to see the elements.
>
> This is because, depending on the runner, the `PCollection` elements might live in multiple worker machines.

To print the elements in the PCollection, we'll do a little trick, but we'll explain it shortly.

```
import apache_beam as beam

inputs = [0, 1, 2, 3]

with beam.Pipeline() as pipeline:
  outputs = (
      pipeline
      | 'Create initial values' >> beam.Create(inputs)
  )

  # We can only access the elements through another transform.
  # Don't worry if you don't know what's happening here.
```

```
# we'll get to it just next :)
outputs | beam.Map(print)
```

```
0
1
2
3
```

Transforming data

Apache Beam is designed with a functional paradigm. This means that, instead of *loops*, it uses `PTransform`s alongside with *functions* to process each element in a `PCollection`.

Let's go through some of the most common and basic data transforms in Beam.

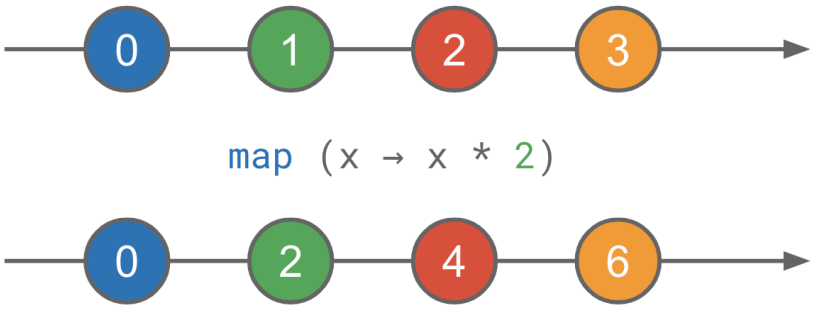
Map: *one-to-one*

Let's say we have some elements and we want to do something with each element.

We want to `map` a function to each element of the collection.

`map` takes a *function* that transforms a single input `a` into a single output `b`.

> For example, we want to multiply each element by 2.



In Python, this is commonly done with the built-in `map` function, or with list comprehensions.

```
inputs = [0, 1, 2, 3]

# Using the `map` function.
outputs = map(lambda x: x * 2, inputs)
print(list(outputs))

# Using a list comprehension.
outputs = [x * 2 for x in inputs]
print(outputs)

# Roughly equivalent for loop.
outputs = []
for x in inputs:
  outputs.append(x * 2)
print(outputs)
```

```
[0, 2, 4, 6]
[0, 2, 4, 6]
[0, 2, 4, 6]
```

In Beam, there is the `Map` transform, but we must use it within a pipeline.

First we create a pipeline and feed it our input elements. Then we *pipe* those elements into a `Map` transform where we apply our function.

```
import apache_beam as beam

inputs = [0, 1, 2, 3]

with beam.Pipeline() as pipeline:
  outputs = (
      pipeline
      | 'Create values' >> beam.Create(inputs)
```

```
      | 'Multiply by 2' >> beam.Map(lambda x: x * 2)
  )

  outputs | beam.Map(print)
```

```
0
2
4
6
```

> ℹ Now that we know how `Map` works, we can see what's happening when we print the elements.
>
> We have our outputs stored in the `outputs` `PCollection`, so we *pipe* it to a `Map` transform to apply the `print` function.
>
> Note that `print` returns `None`, so we get an output `PCollection` of all `None` elements. But we are not saving its results to any variable, so they get discarded.
>
> This does *not* affect the values in `outputs` in any way.

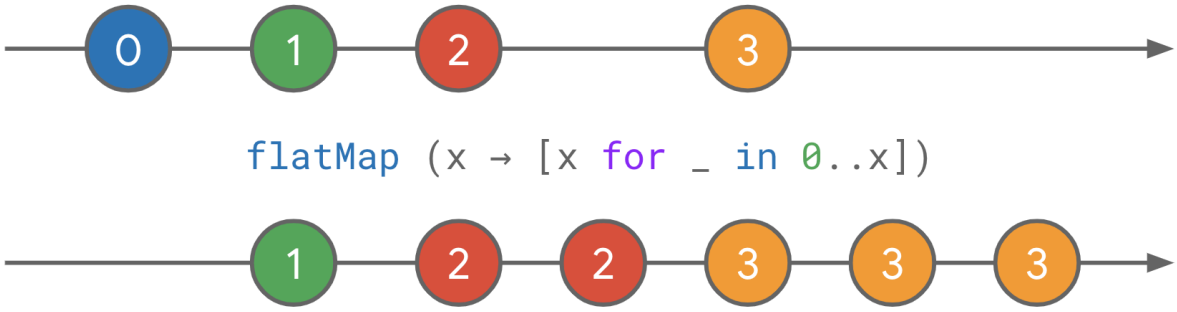
FlatMap: *one-to-many*

`Map` allows us to transform each individual element, but we can't change the number of elements with it.

We want to `map` a function to each element of a collection. That function returns a *list of output elements*, so we would get a *list of lists of elements*. Then we want to *flatten* the *list of lists* into a single *list*.

`flatMap` takes a function that transforms a single input `a` into an `iterable` of outputs `b`. But we get a *single collection* containing the outputs of *all* the elements.

> ℹ For example, we want to have as many elements as the element's value. For a value `1` we want one element, and three elements for a value `3`.



In Python this could be done with a *nested list comprehension*, but it's a little tricky to read.

```
inputs = [0, 1, 2, 3]

# Using a list comprehensions.
mapOutputs = [[x for _ in range(x)] for x in inputs]
# After the map function, flatten the results.
outputs = [x for xs in mapOutputs for x in xs]
print(outputs)

# Roughly equivalent for loop.
outputs = []
for x in inputs:
  outputs += [x for _ in range(x)]
print(outputs)
```

```
[1, 2, 2, 3, 3, 3]
[1, 2, 2, 3, 3, 3]
```

The good news is that Beam already has a `FlatMap` transform built-in, so it's actually easier than plain Python.

`FlatMap` accepts a function that takes a single input element and outputs an `iterable` of elements.

```
import apache_beam as beam

inputs = [0, 1, 2, 3]
```

```
with beam.Pipeline() as pipeline:
  outputs = (
      pipeline
      | 'Create values' >> beam.Create(inputs)
      | 'Expand elements' >> beam.Map(lambda x: [x for _ in range(x)]) # [[1], [2, 2], [3,3,3]]
      # | 'Expand elements' >> beam.FlatMap(lambda x: [x for _ in range(x)]) # [1, 2, 2, 3, 3,3]
  )

  outputs | beam.Map(print)
```

```
[]
[1]
[2, 2]
[3, 3, 3]
```

> Try replacing the `FlatMap` transform with `Map` to see how they behave differently.

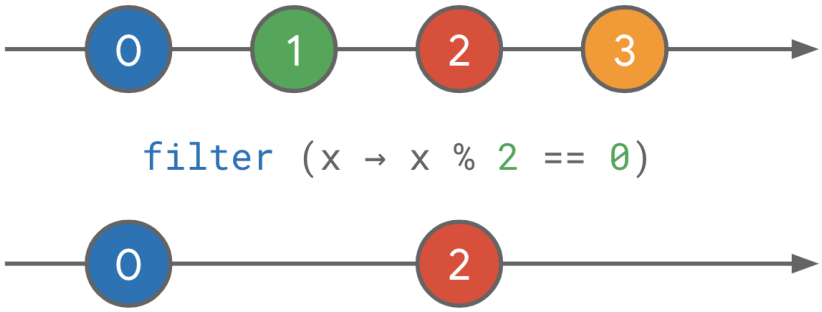
Filter: *one-to-zero*

Sometimes we want to *only* process certain elements while ignoring others.

We want to `filter` each element in a collection using a function.

`filter` takes a function that checks a single element `a`, and returns `True` to keep the element, or `False` to discard it.

> For example, we only want to keep number that are *even*, or divisible by two. We can use the [modulo operator `%`]



In Python we can do this with *list comprehensions* as well.

```
inputs = [0, 1, 2, 3]

# Using a list comprehensions.
outputs = [x for x in inputs if x % 2 == 0]
print(outputs)

# Roughly equivalent for loop.
outputs = []
for x in inputs:
  if x % 2 == 0:
    outputs.append(x)
print(outputs)
```

```
[0, 2]
[0, 2]
```

In Beam, there is the `Filter` transform.

```
import apache_beam as beam

inputs = [0, 1, 2, 3]

with beam.Pipeline() as pipeline:
  outputs = (
      pipeline
      | 'Create values' >> beam.Create(inputs)
      | 'Keep only even numbers' >> beam.Filter(lambda x: x % 2 == 0)
  )

  outputs | beam.Map(print)
```

```
0
2
```

Combine: *many-to-one*

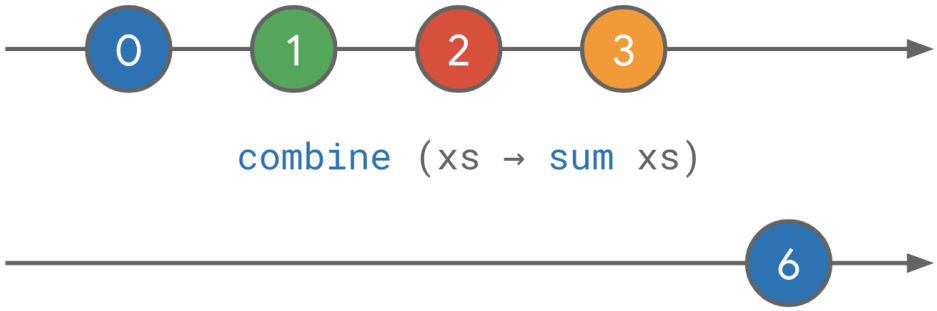
We also need a way to get a single value from an entire `PCollection`. We might want to get the total number of elements, or the average value, or any other type of *aggregation* of values.

We want to `combine` the elements in a collection into a single output.

`combine` takes a function that transforms an `iterable` of inputs `a`, and returns a single output `a`.

Other common names for this function are `fold` and `reduce`.

> For example, we want to add all numbers together.



In Python this is usually achieved with the `reduce` function.

```
from functools import reduce

inputs = [0, 1, 2, 3]

# Using reduce (most general way).
output = reduce(lambda x, y: x + y, inputs, 0)
print(output)

# Using the built-in sum function, which is itself a "reduce" function.
output = sum(inputs)
print(output)

# Roughly equivalent for loop.
output = 0
for x in inputs:
  y = output
  output = x + y
print(output)
```

```
6
6
6
```

In Beam, there are aggregation transforms.

For this particular example, we can use the `CombineGlobally` transform which accepts a function that takes an iterable of elements as an input and outputs a single value.

We can pass the built-in function `sum` into `CombineGlobally`.

```
import apache_beam as beam

inputs = [0, 1, 2, 3]

with beam.Pipeline() as pipeline:
  outputs = (
      pipeline
      | 'Create values' >> beam.Create(inputs)
      | 'Sum all values together' >> beam.CombineGlobally(sum)
  )

  outputs | beam.Map(print)
```

```
6
```

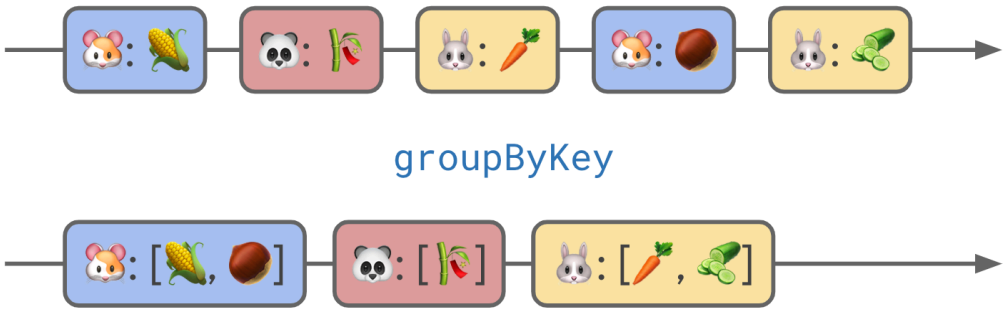
> ℹ There are many ways to combine values in Beam. You could even combine them into a different data type by defining a custom `CombineFn`.
>
> You can learn more about them by checking the available aggregation transforms.

GroupByKey: *group related elements*

Sometimes it's useful to pair each element with a *key* that we can use to group related elements together.

Think of it as creating a Python `dict` from a list of `(key, value)` pairs, but instead of replacing the value on a "duplicate" key, you would get a list of all the values associated with that key.

> ℹ For example, we want to group each animal with the list of foods they like, and we start with `(animal, food)` pairs.



There's no built-in function for `groupByKey` in plain Python, but here's a simple implementation.

```
from functools import reduce

inputs = [
  ('🐹', '🌽'),
  ('🐼', '🥬'),
  ('🐰', '🥕'),
  ('🐹', '🥔'),
  ('🐰', '🥒'),
]


# Since we're getting a single dict from all the elements,
# we can use reduce for this.
def groupByKey(result, keyValue):
  key, value = keyValue
  values = result.get(key, []) + [value]
  return {**result, key: values}
output = reduce(groupByKey, inputs, {})
print(output)


# Roughly equivalent for loop.
output = {}
for key, value in inputs:
  values = output.get(key, []) + [value]
  output = {**output, key: values}
print(output)
```

```
{'🐹': ['🌽', '🥔'], '🐼': ['🥬'], '🐰': ['🥕', '🥒']}
{'🐹': ['🌽', '🥔'], '🐼': ['🥬'], '🐰': ['🥕', '🥒']}
```

In Beam, there is the `GroupByKey` transform.

```
import apache_beam as beam

inputs = [
  ('🐹', '🌽'),
  ('🐼', '🥬'),
  ('🐰', '🥕'),
  ('🐹', '🥔'),
  ('🐰', '🥒'),
]

with beam.Pipeline() as pipeline:
  outputs = (
```

```
      pipeline
      | 'Create (animal, food) pairs' >> beam.Create(inputs)
      | 'Group foods by animals' >> beam.GroupByKey()
  )

  outputs | beam.Map(print)
```

```
('🐯', ['🥕', '🧅'])
('🐼', ['🎋'])
('🐰', ['🥕', '🥒'])
```

```
inputs = ["ChatGPT’s result is far more detailed than our outline. Here, you should adopt the parts of ChatGPT’s outline you
```

```
# Mini Word Count example
import apache_beam as beam
import re

WORD_RE = re.compile(r"[A-Za-z]+(?:'[A-Za-z]+)?")
with beam.Pipeline() as pipeline:
  outputs = (
      pipeline
      | 'Create values' >> beam.Create(inputs)
      | 'ExtractWords' >> beam.FlatMap(lambda x: WORD_RE.findall(x.lower()))
      | 'PairWithOne' >> beam.Map(lambda w: (w, 1)) # Map, (w, 1)
      | 'CountPerWord' >> beam.CombinePerKey(sum)   # Reduce, (w, count)
      | 'TopByCount' >> beam.combiners.Top.Of(20, key=lambda kv: kv[1])
      | 'Flatten' >> beam.FlatMap(lambda words: words)
  )

  outputs | beam.Map(print)
```

```
WARNING:apache_beam.transforms.core:('No iterator is returned by the process method in %s.', <class 'apache_beam.transforms.
('you', 10)
('the', 9)
('to', 6)
('your', 6)
('section', 4)
('that', 4)
('outline', 4)
('chatgpt', 3)
('best', 3)
('in', 3)
('with', 2)
('our', 2)
('as', 2)
('will', 2)
('of', 2)
('sections', 2)
('like', 2)
('can', 2)
('an', 2)
('and', 2)
```