

Apache Beam

Part One...

Introduction



Using Apache beam is helpful for the ETL tasks, specially if you are running some transformation on the data before loading it into its final destination.

Currently there are limited SDKs:

- **Java**: Most complete SDK
- **Python**: Most of the features already implemented in the Python SDK, but not all and everything. It is usually updated after the Java SDK.
- **Go**: Experimental, does not yet offer any compatibility guarantees and is not recommended for production usage. It supports most features, but not all.
- **SQL**

We will use the **Python SDK** in the next few sections.

Installation

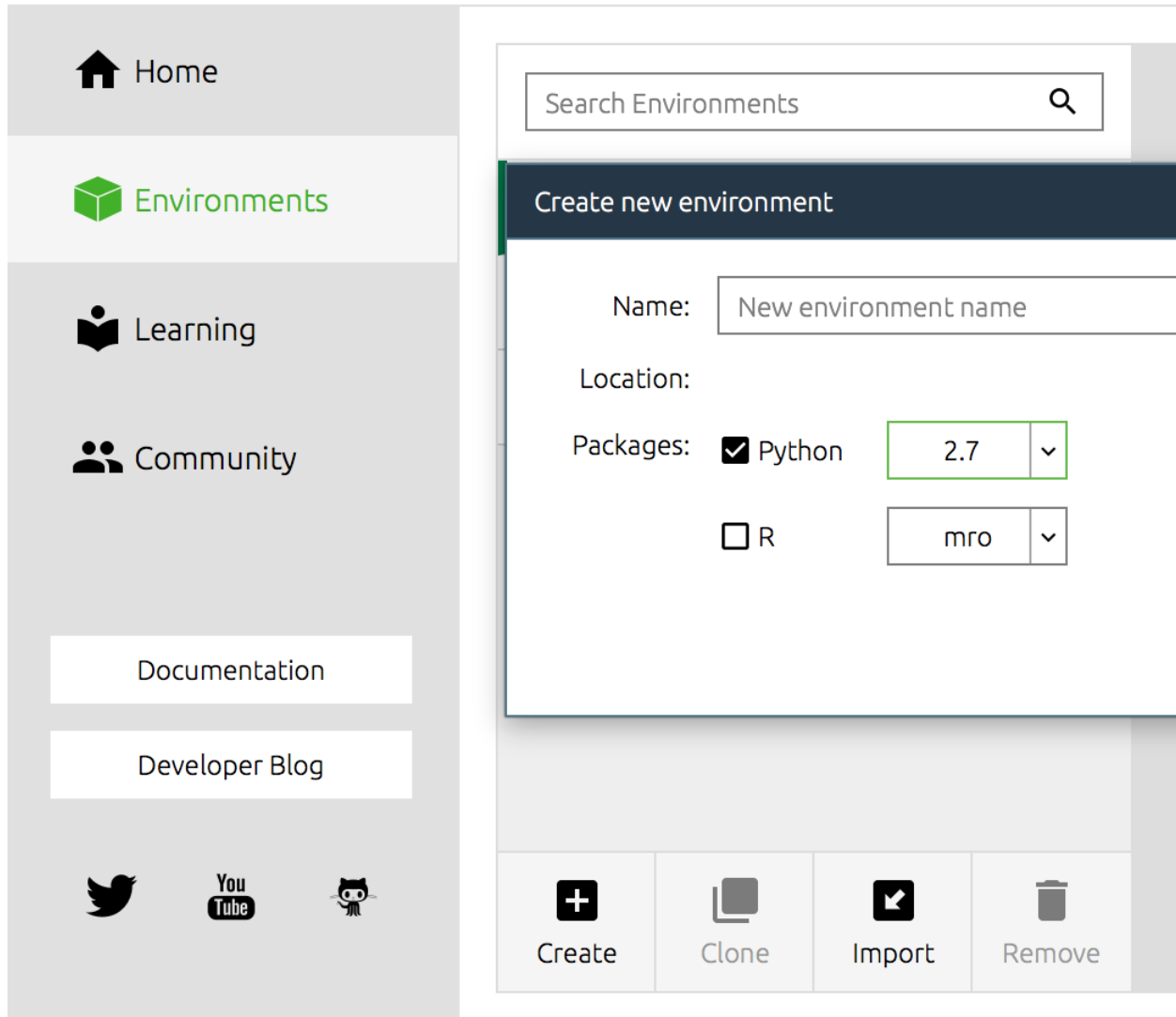
Preparing the Environment:

Lets first notice that beam currently working with Python 2.7, so if you don't have Python 2.7 environment, please setup one.

For that, you can do it easily if you are using Anaconda-Navigator.

Create a new Environment, and select the Python 2.7 release.

ANACONDA[®] NAVIGATOR



The screenshot shows the Anaconda Navigator application interface. On the left is a sidebar with navigation options: Home (house icon), Environments (cube icon), Learning (book icon), and Community (people icon). Below these are buttons for Documentation and Developer Blog, and social media icons for Twitter, YouTube, and GitHub. The main panel on the right features a 'Search Environments' search bar at the top. Below it is a 'Create new environment' dialog box. This dialog has a 'Name' field with the placeholder 'New environment name'. The 'Location' field is currently empty. Under 'Packages', there are two options: 'Python' which is checked and has a version dropdown set to '2.7', and 'R' which is unchecked and has a version dropdown set to 'mro'. At the bottom of the main panel are four buttons: 'Create' (plus icon), 'Clone' (document icon), 'Import' (arrow icon), and 'Remove' (trash icon).

Now you created the environment, swap to it if it is not active. (How to do that? Read next line...)

From your terminal, check which python version is running by writing the next command:

```
which python
```

Ensure that the command output contains the path to the environment you just created. Something like: `/anaconda3/envs/Python2-7-X/bin/python` Where Python2-7-X is my new created environment.

Or you can check the Python version using:

```
python --version
```

Ensure it is 2.7 or 2.7.x (x is any number... easy)

It also required [pip](#), Python package manager. Check that you have version 7.0.0 or newer by running:

```
pip --version
```

Upgrade it if required, run the following command to install it.

```
pip install --upgrade pip
```

Other option, you can have a Vagrant machine, and have your isolated work environment.

If you are going to run a vagrant machine, ensure installing Python 2.7.X, and pip.

For **ubuntu-trusty-64**, run the next commands in your terminal. (You may need `sudo` in some commands)

```
apt-get update
curl "https://bootstrap.pypa.io/get-pip.py" -o "get-pip.py"
python get-pip.py
pip -V
```

Installing apache-beam:

That is easy so far, now lets install beam itself. You can follow the steps [here](#).

Which are:

Run the next command in your terminal:

```
pip install apache-beam
```

One thing to notice here, that command will not install everything, it will only install what is required to run it standing alone. But let's say that you need GCP dependences as example, then you need to install it.

How to do that? You can just list the features you are looking for:

```
pip install apache-beam[feature1,feature2]
```

So, in our case, we are targeting writing to and reading from GCP BigQuery, so we need to install are related features.

```
pip install apache-beam[gcp]
```

If you want to just have everything, then install all (so easy):

```
pip install apache-beam[all]
```

Now, installation done...

Now, let's start writing codes...

Create your new python file, call it **counter.py** as we will do a simple counting operations. For that, I am using a simple dataset, from [Kaggle](#) about some **transactions from a bakery**, which was [here](#) before, but you can still find it [here](#). You will find it online.

The dataset contains four variables, Date, Time, Transaction ID and Item.

We simply will get some integrated information from it:

1. The count of transactions happened in different days. **(Part One)**
2. The list of items had been sold in different days. **(Part Two)**
3. The max and minimum number of transactions per item. **(Part Two)**

So, we will basically cover few important concepts here:

- Loading the data from different resources:
 - CSV/TXT **(Part One)**
 - CSV **(Part Two)**
 - BigQuery **(Part Three)**
- Transforming the data, running the operations:
 - Map (ParDo)
 - Shuffle\Reduce (GroupBy)
- Inserting the final data into the destination:
 - TXT **(Part One)**
 - CSV **(Part Two)**
 - BigQuery **(Part Three)**

Which are simply our **ETL** steps.

First, we need to create and configure our [pipeline](#), which will encapsulate all the data and steps in our data processing task.

Pipeline configurations

Import the required libraries, basically beam, and its configuration options:

```
import apache_beam as beam
from apache_beam.options.pipeline_options import PipelineOptions

p = beam.Pipeline(options=PipelineOptions())
```

For the configuration, you can have your own configuration by creating an element from the `PipelineOptions` class:

```
class MyOptions(PipelineOptions):
    @classmethod
    def _add_argparse_args(cls, parser):
        parser.add_argument('--input',
                            help='Input for the pipeline',
                            default='gs://my-bucket/input')
        parser.add_argument('--output',
                            help='Output for the pipeline',
                            default='gs://my-bucket/output')
```

In the example here, for input and output, the default path pointing to a file in Google Cloud Storage, you can change it with your default URL:

```
parser.add_argument('--input',
                    help='Input for the pipeline',
                    default='https://your/path/to/the/input')
```

We will see how to load the local files soon.

Then you can use this options in creating your pipeline:

```
p = beam.Pipeline(options=MyOptions)
```

Reading the input data

Now, let's read the input data from a file...

First, import the text reader and writer from `apache_beam.io`

```
from apache_beam.io import ReadFromText
from apache_beam.io import WriteToText
```

Then the first thing that our pipeline will do is loading the data from the source file:

```
data_from_source = (p
    | 'ReadMyFile' >> ReadFromText('./input/BreadBasket_DMS.csv')
    )
```

Here we read the CSV file as text, it is now stored in the **data_from_source** variable.

Now, how to do some processing on this data within apache beam?

As we mentioned earlier, the Map\Shuffle\Reduce here will be implemented here by ParDo, GroupByKey, and CoGroupByKey... which are part of the core Beam transformation.

How to write ParDo?

There two ways to write your ParDo in beam:

- Define it as a function:

```
def printer(data_item):
    print data_item
```

And we use it like this:

```
data_from_source = (p
    | 'ReadMyFile' >> ReadFromText('./input/BreadBasket_DMS.csv')
    | 'Print the data' >> beam.ParDo(printer)
    )
```

- Inherit as a class from **beam.DoFn** :

```
class Printer(beam.DoFn):
```

```
def process(self,data_item):  
    print data_item
```

And we use it like:

```
data_from_source = (p  
    | 'ReadMyFile' >> ReadFromText('./input/BreadBasket_DMS.csv')  
    | 'Print the data' >> beam.ParDo(Printer())  
)
```

The two will do exactly the same thing, but you call them in different ways as we saw.

Now our file looks like:

```
# coding: utf-8  
# Python 2.7  
  
import apache_beam as beam  
  
from apache_beam.options.pipeline_options import PipelineOptions  
  
from apache_beam.io import ReadFromText  
from apache_beam.io import WriteToText  
  
p = beam.Pipeline(options=PipelineOptions())  
  
class Printer(beam.DoFn):  
    def process(self,data_item):  
        print data_item  
  
def printer(data_item):  
    print data_item  
  
data_from_source = (p  
    | 'ReadMyFile' >> ReadFromText('./input/BreadBasket_DMS.csv')  
    | 'Printer the data' >> beam.ParDo(Printer())  
    )  
  
result = p.run()
```

If we run this file, it will just print the data from the source to our terminal.

Please notice that we used the default `PipelineOptions()` here, as we don't have a specific options to set for now.

Running the Transformation on the data

We are reading the raw data to process it... so let's see how to manipulate the data and extract more meaningful statistics from it.

When working with the pipeline, the data sets we have called **PCollection**, and the operations we run called **Transformation**.

As we noticed that the data is processed line by line. That is because **ReadFromText** generates a **PCollection** as an output, where each element in the output PCollection represents one line of text from the input file.

We used **ParDo** function to run a specific operation on the data, this operation run on every element in the PCollection. As we saw, we printed every line of the data alone.

Now, we will only get the dates from the input data. To do that, we will have a new **DoFn** to return the date only from each element.

If you check the type of the data passed from the PCollection to the DoFn from the ParDo, it will return **<type 'unicode'>** which are string. To check that, create and use the next DoFn:

```
class TypeOf(beam.DoFn):
    def process(self, data_item):
        print type(data_item)

data_from_source = (p
    | 'ReadMyFile' >> ReadFromText('./input/BreadBasket_DMS.csv')
    | 'Check data type' >> beam.ParDo(TypeOf())
)
```

OK, it is a string. We can split it by the comma operator, which will return an array of elements. We can easily use `beam.Map` which accepts lambda function and implements it on every record of the PCollection.

Our lambda function is simple here, split the input by the comma operator, then return the first element:

```
beam.Map(lambda record: (record.split(',')[0]))
```

So, our pipeline we looks like this:

```
data_from_source = (p
    | 'ReadMyFile' >> ReadFromText('./input/BreadBasket_DMS.csv')
    | 'Splitter using beam.Map' >> beam.Map(lambda record:
(record.split(',')[0])
    | 'Print the date' >> beam.ParDo(Printer())
)
```

The output should be something like this:

```
...
2017-04-03
2017-04-03
2017-04-04
2017-04-04
...
...
2017-04-08
2017-04-08
2017-04-09
2017-04-09
...
```

Great, now we have only the dates. What we can do by this data?

Let's count how many order made everyday...

To do that, we will need to do three simple steps. **First**, Map each record with 1 as a counter. **Second**, group the records with the similar data. **Third**, get the sum of the 1s.

First:

```
| 'Map record to 1' >> beam.Map(lambda record: (record, 1))
```

Which will return:

```
...
```

```
(u'2017-04-09', 1)
(u'2017-04-09', 1)
(u'2017-04-09', 1)
...
```

Second:

```
| 'GroupBy the data' >> beam.GroupByKey()
```

The date is the key from the previous step. That will return:

```
(u'2016-11-24', [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,  
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,  
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,  
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,  
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1])
```

Third:

We have two ways to do this, `beam.Map` OR `beam.ParDo`

```
| 'Sum using beam.Map' >> beam.Map(lambda record:
(record[0],sum(record[1])))
```

OR:

```
class GetTotal(beam.DoFn):
    def process(self, element):
        # get the total transactions for one item
        return [(str(element[0]),sum(element[1]))]
...
...
# In your pipeline
| 'Get the total in each day' >> beam.ParDo(GetTotal())
```

When you run the application, you will get result like this in the terminal:

```
(u'2016-11-12', 227)
(u'2016-11-11', 174)
(u'2016-11-10', 155)
(u'2016-11-29', 102)
(u'2017-03-04', 265)
```

We now know how many orders happened in each day.

Writing the output data

We need to store this data somewhere for future usage... so we will write it in a new text file for now.

```
| 'Export results to new file' >> WriteToText( 'output/day-list','.txt')
```

If you are using `beam.Map`, your code should look like:

```
data_from_source = (p
    | 'ReadMyFile' >> ReadFromText('./input/BreadBasket_DMS.csv')
    | 'Splitter using beam.Map' >> beam.Map(lambda record: (record.split(',')[0])
    | 'Map record to 1' >> beam.Map(lambda record: (record, 1))
    | 'GroupBy the data' >> beam.GroupByKey()
    | 'Sum using beam.Map' >> beam.Map(lambda record: (record[0],sum(record[1])))
    | 'Export results to new file' >> WriteToText( 'output/day-list','.txt')
)
```

And if you are using `beam.ParDo`, your code will look like:

```
class GetTotal(beam.DoFn):
    def process(self, element):
        # get the total transactions for one item
        return [(str(element[0]),sum(element[1]))]

data_from_source = (p
    | 'ReadMyFile' >> ReadFromText('./input/BreadBasket_DMS.csv')
    | 'Splitter using beam.Map' >> beam.Map(lambda record: (record.split(',')[0])
    | 'Map record to 1' >> beam.Map(lambda record: (record, 1))
    | 'GroupBy the data' >> beam.GroupByKey()
    | 'Get the total in each day' >> beam.ParDo(GetTotal())
    | 'Export results to new file' >> WriteToText( 'output/day-list','.txt')
)
```

Review the Code:

Your file will look like:

```
# coding: utf-8
# Python 2.7

import apache_beam as beam

from apache_beam.options.pipeline_options import PipelineOptions

from apache_beam.io import ReadFromText
from apache_beam.io import WriteToText

p = beam.Pipeline(options=PipelineOptions())

class GetTotal(beam.DoFn):
    def process(self, element):
        # get the total transactions for one item
        return [(str(element[0]), sum(element[1]))]

data_from_source = (p
    | 'Read data file' >> ReadFromText('./input/BreadBasket_DMS.csv')
    | 'Splitter using beam.Map' >> beam.Map(lambda record:
(record.split(',')[0])
    | 'Map record to 1' >> beam.Map(lambda record: (record, 1))
    | 'GroupBy the data' >> beam.GroupByKey()
    | 'Get the total in each day' >> beam.ParDo(GetTotal())
    | 'Export results to new file' >> WriteToText('output/day-list', '.txt')
)

result = p.run()
```

That will run a simple counting operation on the data and will create an output file contains everyday and the count of transactions happened in this day.

Run your code:

```
$ python counter.py
```

Then check the output directory, you should find a new created file named `day-list-00000-of-00001.txt` where you will find the final output of your transformation process.