Apache Beam DataFlow Pipeline

Summary	Scraping EDGAR data and compute word count for various lists using a Google Dataflow Pipeline
URL	your-first-pwapp
Category	Web
Environment	web, kiosk, io2016, pwa-dev-summit, pwa-roadshow, chrome-dev-summit-2016, io2017, typtwd17, gdd17, cds17, io2018, tag-web, jsconfeu, devfest18
Status	Draft
Feedback Link	https://github.com/googlecodelabs/your-first-pwapp/issues
Author	Manogna Mantripragada, Kashish Shah, Dhruv Panchal

Abstract

- Creating a Google Dataflow pipeline to extract the EDGAR website links, and store it in the specified CSV format,
- Scrape the data using Beautiful Soup and cleaning using Regular Expression.
- Tokenization: NLTK Tokenizer
- Count the words and process the word count on based on the dictionary of words.

Apache Beam

- **Apache Beam** is an open source, **unified model for defining both batch and streaming data-parallel processing pipelines**
- It is **portable** and executes pipelines on **multiple execution environments**.
- Apache Beam comes with **Java and Python SDK** as of now and a **Scala** interface is also available.
- The programs built can be executed by one of Beam's supported distributed processing back-ends, which include **Apache Apex**, **Apache Flink, Apache Spark**, and **Google Cloud Dataflow**.

Apache Beam Pipeline

A Pipeline manages a directed acyclic graph of PTransforms(Transformations) , and the PCollections (Databases) that the PTransforms consume and produce.

Each Pipeline is **self-contained** and **isolated from any other Pipeline**. The PValues that are inputs and outputs of each of a Pipeline's PTransforms are also owned by that Pipeline. A PValue owned by one Pipeline can be read only by PTransforms also owned by that Pipeline. Pipelines can safely be executed concurrently.

Building an Apache Beam Pipeline

What to consider when designing your pipeline

Where is your input data stored?
How many sets of input data do we have?

What does your data look like?
It might be plaintext, formatted log files, or rows in a database table.

What do you want to do with your data?
The core transforms in the Beam SDKs are general purpose. Knowing how you need to change or manipulate your data will determine how you build core transforms like ParDo, or when you use pre-written transforms included with the Beam SDKs.

What does your output data look like, and where should it go?
This will determine what kinds of Write transforms you'll need to apply at the end of your pipeline.

Installation of Apache Beam:

1. Installing Apache Beam

$pip install apache-beam[gcp]

2. Creating a basic pipeline ingesting CSV

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

```
import apache_beam as beam
from apache_beam.options.pipeline_options import PipelineOptions
options = PipelineOptions()
p = beam.Pipeline(options=options)
```

PCollection and Transform

PCollection<T> is the data abstraction used in Apache Beam. Its type corresponds to the type of the values stored inside. The whole class can be described in following points:

- **immutable** - the data stored in one PCollection can't be modified. If some transformation is applied on it, a new PCollection instance is created.
- **distributed** - the PCollection represents the data stored in distributed dataset, that is the parts of whole data resides on different processing nodes.
- **boundary(less)** - PCollection is a data abstraction shared by stream and batch processing. It means that its size can be unbounded or bounded.
- **universal** - one PCollection can be an input for a lot of different transformations.

Transforms principles

In Beam, data is represented as a PCollection object. So to start ingesting data, we need to read from the csv and store this as a PCollection to which we can then apply transformations. The Read operation is considered as a transform and follows the syntax of all transformations:

[Output PCollection] = [Input PCollection] | [Transform]

These transforms can then be chained like this:

[Final Output PCollection] = ([Initial Input PCollection] | [First Transform]
| [Second Transform]
| [Third Transform])

The pipe is the equivalent of an apply method.

Reading input data and writing output data

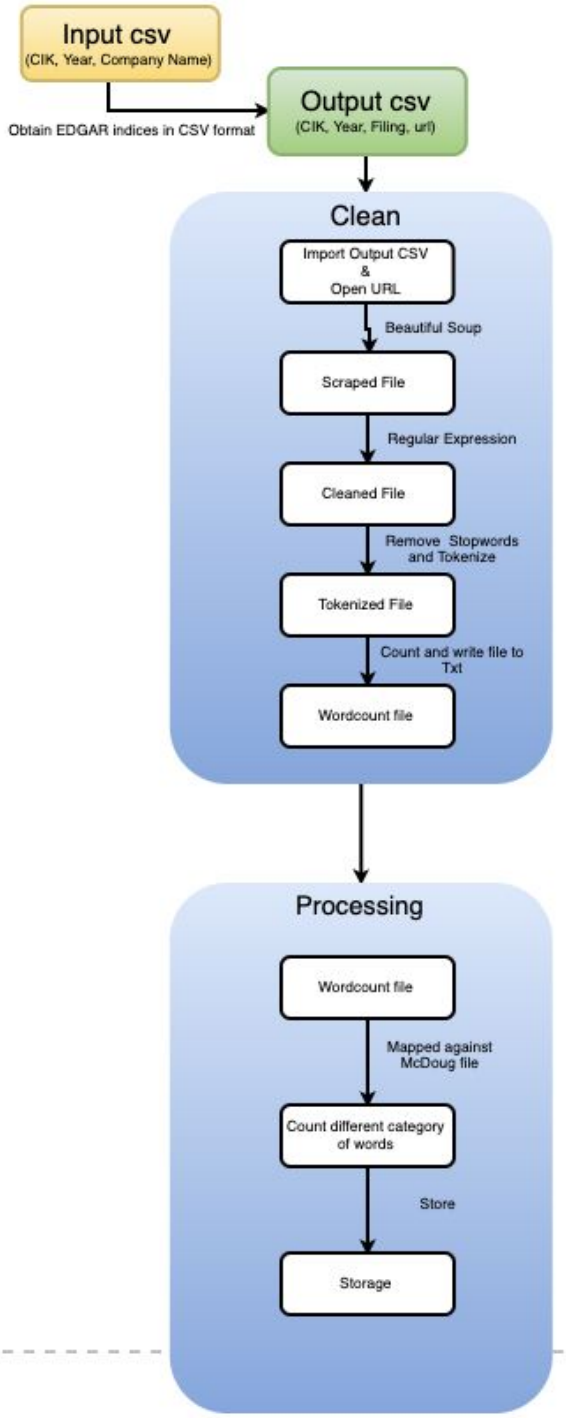
csv_lines = (p | ReadFromText(input_filename, skip_header_lines=1) | ...

At the other end of our pipeline we want to output a text file. So let's use the standard writer:

... | beam.io.WriteToText(output_filename)

What are we trying to do?

- Creating a Google Dataflow pipeline to extract the EDGAR website links, and store it in the specified CSV format,
- Scrape the data using Beautiful Soup and cleaning using Regular Expression.
- Tokenization: NLTK Tokenizer
- Count the words and process the word count on based on the dictionary of words.


Input csv
(CIK, Year, Company Name)
Obtain EDGAR indices in CSV format
Output csv
(CIK, Year, Filing, url)
Clean
Import Output CSV
&
Open URL
Beautiful Soup
Scraped File
Regular Expression
Cleaned File
Remove Stopwords
and Tokenize
Tokenized File
Count and write file to
Txt
Wordcount file
Processing
Wordcount file
Mapped against
McDoug file
Count different category
of words
Store
Storage

Our Approach:

- Started with creating basic python definitions to execute the logic
- Implemented the pipeline and run the code using Default runner(DirectRunner)
- Create Google cloud project, enable API's , create instance, storage bucket and create a pipeline on the Cloud using a DataFlow Runner.
- Store the output files in the storage bucket.

Hiccups:

1. Tokenizing: Tokenizer wasn't able to efficiently tokenize all the words
2. Accessing the cloud storage bucket
3. Inadequate documentation

What would have been helpful?

1. Tutorial to understand the working and implementation
2. Better Documentation