Data Pipeline Language v1

João Pinto <lamego.pinto@gmail.com>

Data Pipeline Language v1

João Pinto – lamego.pinto@gmail.com – Draft

Table of Contents

Introduction

This document fully describes the Data Pipeline Language version 1. DPL is a general purpose language that can be used to describe a data transformation process for both structured and unstructured data. It does not intend to be a replacement for domain/technology specific languages like SQL, it seeks to provide an higher level «but computable» language capable of integrating data from diverse formats and sources.

Prerequisites

DPL is a declarative meta-language fully based on the YAML 1.1 format. The knowledge of YAML is a plus for better understanding of the material in this specification.

The DPL reference execution engine, mdatapipe, is written in Python 3. In general it is not required to use Python programming within a pipeline, however when dealing with data types transformation and more complex operations, it is recommended to have a good understanding on Python's Standard Data Types and Operations.

Pipelines

A data pipeline is an YAML document that describes the sequence of operations required to produce some output from a set of inputs. It can be used to describe both real-time and batch processes.

There are two kinds of pipeline definitions schemas: simple and structured. They will be described in the next section.

> **NOTE** The concept of a data pipeline is similar to the concept of business process, the main difference being that BPs are centered on business tasks and activities, while DPs are centered on general data processing.

Simple Pipelines

A simple pipeline is an YAML document where the top level element is a sequence. Each item in the sequence defines a step. A step is composed using the format: *component name* : *configuration*, where *component name* must be one of the components available from the DPL execution library, while the *configuration* «which is optional» can be any YAML content.

Every step in a pipeline may receive input from the previous step and may deliver output to the next step.

> **NOTE** The first step in a simple pipeline receives a single input item containing the system clock time. That item is delivered by the DPL execution engine.

simple_example.yaml

```
# Read CSV file and parse it to column values
- collect from file: data.csv
# Prints each row
- print:
```

In the previous example the first step uses the component "*collect from file*" and the string "*data.csv*" for its configuration. The second step uses the component "*print*" without any configuration.

Structured Pipelines

A structured pipeline is composed by one or more segments. A segment is similar to a simple pipeline, in the sense that it represents a sequence of steps, but it has a different YAML structure in order to accommodate the following additional features:

- Output that is produced by the last step of a segment, may be selected «via conditional operators» to be delivered as input to the first step of a different segment.
- Failures output from any step of a segment, can be sent as input to other segment.

The YAML of a structured pipeline follows the following schema:

structured.yaml

```
segments:
    start: # This is the name of a segment, "start", which must be present
        steps:
            # Read CSV file and parse it to column values
            - collect from file: people.csv
            # Prints each row
            - print
        on_fail: report_the_failure
        on_success:
            # segment_name: condition
            teen_stuff: age > 9 and age < 30 # on True send to the "teen_stuff" segment
            big_foot: feet > 40              # on True send to "big_foot" segment
            else_do_nothing:  # if empty, and all other conditions are False, send
    report_the_failure: # This is the name of a segment
        - print: "There was an error processing the csv file"
    teen_stuff: # This is the name of a segment
        print: "Will do something for teens"
    big_foot:  # This is the name of a segment
        print "Will do something for big foot people"
    else_do_nothing: # This is the name of a segment
        print "Nothing special"
```

In the previous example, data can flow in multiple segments depending on the following conditions:

Table 1. Data delivered to segments

Segment	**First step of the segment gets one input item when…**
start	The pipeline is started
report_the_failure	Every time any step of the *start* segment fails
	Every time the last step of the *start* segment produces one output item and …
teen_stuff	age > 9 and age < 30
big_foot	feet > 40
else_do_nothing	not teen_stuff and not big_foot

Components

Component is the fundamental processing unit of a pipeline, it provides the logic that is applied when an input is received. When starting a pipeline, the DPL execution engine may create one or more component instances associated with each step. Each instance is started with the configuration provided in the corresponding step.

A component instance may perform some internal logic when starting (eg. initialization of external resources), but it can not produce any output before receiving an input. For this reason, a DPL execution engine may chose to perform lazy initialization or to suspend an instance until input data becomes available.

Input, Output and Configuration items

One of the great challenges of integrating data from different sources is data type diversity. DPL components inputs, outputs, and configurations are Python objects, in DPL we refer to them using the term `item`. Input and output items can be of any type available on Python, configuration can be of any type available on YAML.

The most common data types are: strings, numbers «which can be integer or float», lists «which can contain any type of item» and dictionaries which contain a collection of *key: value* pairs.

Components Polymorphism

DPL components can be polymorphic, the same component may be able to handle different types of input and configuration items, apply different logic based on those types, and produce output in different formats.

While components may be polymorphic, component instances may not, their processing logic and output datatype is set based on their configuration and/or the first input item type. Multiple outputs of a single step in the pipeline must always be of the same type.

Table 2. Example of the polymorphic logic for the "sum" component

Config Type	Input Type	Logic
-	list	Output the sum of the input list items
-	integer,float	Accumulate the value, produce the total only when is end of input
string	dict	Accumulate the value of dict[string], produce the total end of input is reached
dict	dict	For all config keys, accumulate the corresponding input keys, produce dict with tot when end of input is reached

WARNING Instances of a component are not polymorphic, its logic will be set based on its configuration and its first input item. As such, any step in a data pipeline is expected to produce outputs of the same type.

Dynamic Configuration

Step's configuration can include references to parts or the entire content of the input item. Those references will be replaced by the corresponding value every time an input item is received.

Format	Value related to the input item
$.$	Full item content
$*key_name*$	Item[key_name]
$*number*$	Item[n], where *number* is an integer number
$#$	Length of item

WARNING If you need to include a regular "*$*" symbol as part of your configuration text, it needs to be escaped using "*\$*" .

Conclusion

After reading this document you should be familiar with the fundamental concepts required to write a data pipeline using DPL. At this point, is recommended that you read the *DPLv1 Standard Components Reference*, or of you prefer to learn by example, go directly into the *DPLv1 Examples Quick Guide*.

CopyRight and License