Programming with Apache Beam, pipelines and stream data

April 16, 2024

Basics of Apache Beam

Material taken from apache beam documentation

- Pipeline: graph of transformations
- PCollection: data being processed
- PTransforms: operations on PCollections
- SDK: language (in our case, Python)
- Runner: takes a beam pipeline and executes it

Basics of Apache Beam: PTransform

- PTransforms can be one of the 5 primitives:
  - Read: parallel conectors to external systems
  - ParDo: per element processing
  - GroupByKey: aggregating elements
  - Flatten: union of PCollections
  - Window: set the windowing strategy for a PCollection

Basics of Apache Beam: PCollections

- may be:
  - Bounded: finite, as in batch use cases
  - Unbounded: it may be infinite, as in streaming use cases

Basics of Apache Beam: Timestamps

- Every element in a PCollection has a timestamp associated with it
- If elements denote events, timestamps are important
- In case the timestamp is not important it is set to “negative infinity”

Basics of Apache Beam: Watermarks

- Estimates how complete a PCollection is
- The contents of a PCollection are complete when a watermark advances to “infinity”
  → this way we know that an unbounded PCollection is finite (has ended)

Basics of Apache Beam: Windowed elements

- Windows define the size (number of elements) that will be processed in the pipeline at once
- When elements are read from external sources they arrive in the global window
- When they are written to the outside world, they are placed back into the global window
  → any writing transform that doesn't obey it may risk data loss
- A window has a maximum timestamp
- All data related to an expired window may be discarded at any time

Basics of Apache Beam: Coder

- Specifies the binary format of the elements of a PCollection
- Can be just bytes or some encoding system (for example, graphical accents, depending on the language)

Basics of Apache Beam: Windowing strategy

- Specify essential information for grouping and triggering operations → operate on a one-by-one element basis may be very inefficient, depending on the operation
- For example, GroupByKey is governed by a windowing strategy

Basics of Apache Beam: User defined functions (UDF)

- beam pipeline may contain UDFs different from the current runner
- DoFn: per-element processing function (used in ParDo)
- WindowFn: places elements in windows and merges windows (used in Window and GroupByKey)
- ViewFn: adapts a PCollection to a particular interface
- WindowMappingFn: maps one element's window to another, and specifies bounds on how far in the past the result window will be
- CombineFn: associative and commutative aggregation (used in Combine and state)
- Coder: encodes user data

Basics of Apache Beam: Runner

- is used for a couple of things
- it generally refers to the software that takes a beam pipeline and runs it
- it usually includes some customized operators for your data processing engine and it sometimes refers to the full stack
- a runner has a single method run(pipeline)
- run(pipeline) methods should be **asynchronous** and result in a PipelineResult which is a job descriptor. It provides methods:
  - for checking job status
  - canceling
  - waiting for termination

Apache Beam: Execution model

(https://beam.apache.org/documentation/runtime/model/)

- runners can execute a pipeline in different ways
- Processing of elements:
  - serialization[1] and communication between machines is one of the most expensive operations
  - avoiding serialization may require re-processing elements after failures or may limit the distribution of output to other machines

[1] process of translating a data structure to be stored or transmitted

Apache Beam: Serialization and Communication

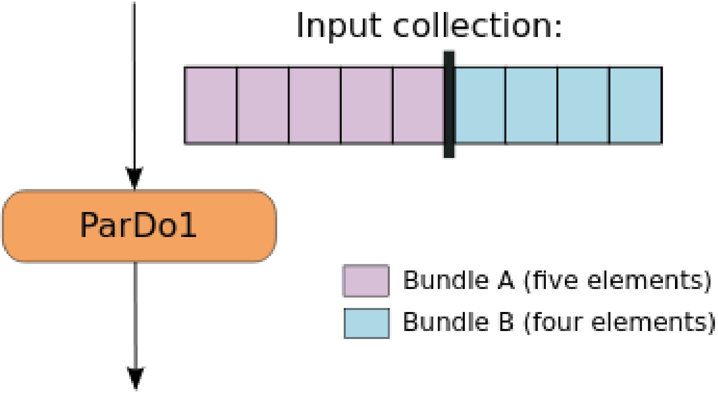
- runner may serialize elements between machines for communication or persistence
- runner may decide transfer elements between transforms in a variety of ways:
  - grouping operation: may involve serializing elements and grouping or sorting them by key
  - redistributing elements between workers to adjust parallelism
  - using elements in a side input to a ParDo: may require serializing the elements and broadcasting them to all workers executing the ParDo
  - passing elements between transforms that are running on the same worker

Apache Beam: Bundling and persistence

- situations for persistence: stateful app or checkpointing
- elements of a PCollection are processed in “bundles”
  - runner chooses appropriate middle-ground between persisting results
  - for example, streaming runnners may prefer to process and commit small bundles, while a batch runner may prefer to process larger bundles

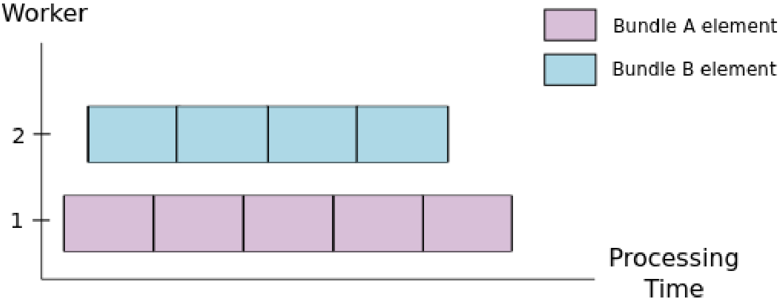
Failures and parallelism within and between transforms

When executing a single ParDo, a runner might divide an example input collection of 9 elements into two bundles



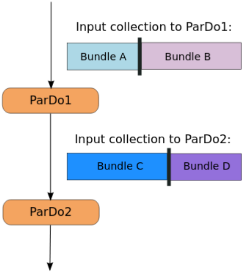
Failures and parallelism within and between transforms

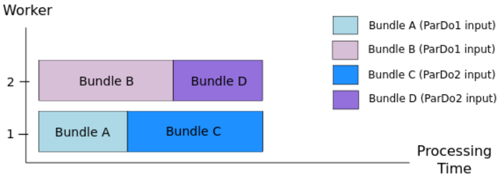
Parallelism within transform: when the ParDo executes, workers can process bundles in parallel



Failures and parallelism within and between transforms

Dependent parallelism between transforms:



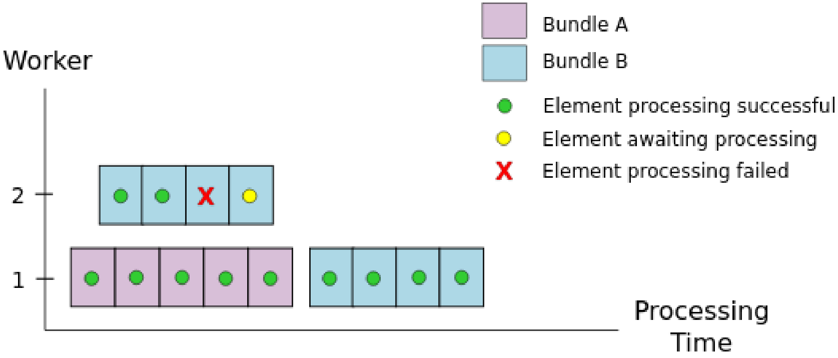


Failures and parallelism within and between transforms

- If processing of an element within a bundle fails, the entire bundle fails
- The elements in the bundle must be retried, otherwise the entire pipeline fails
- but they do not need to be retried in the same worker

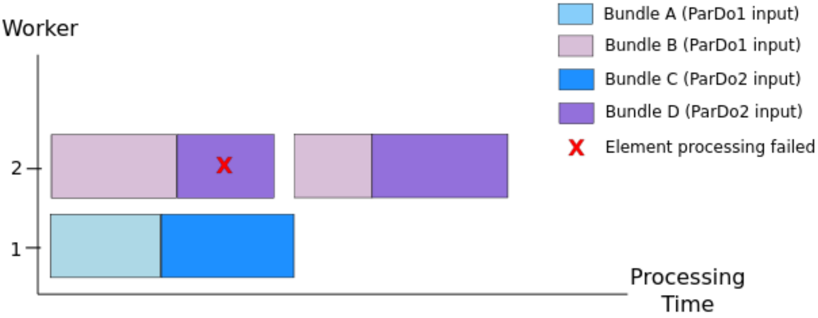
Failures and parallelism within and between transforms

Failures within one transform: input collection with 9 elements, divided in two bundles. First run: worker 2 fails element 3 of its bundle B and worker 1 succeeds with its bundle A. Retry: worker 1 retries **all** bundle B and succeeds.


Bundle A
Bundle B
Element processing successful
Element awaiting processing
Element processing failed
Worker
2
1
Processing Time

Failures and parallelism within and between transforms

Failures between transforms (in this case: *coupled failure*): worker 1 succeeds processing bundle A and produces bundle C which is input to another ParDo. Worker 2 failed processing bundle D, therefore the input used to produce bundle D (bundle B) needs to be recomputed. Therefore a full recomputation of B and D needs to be done.



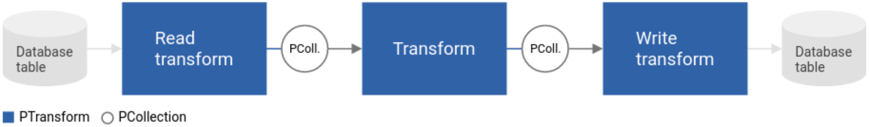
Notice that keeping bundles A-C, B-D in the same worker makes the processing more efficient.

Pipeline development lifecycle

What to consider in the design?

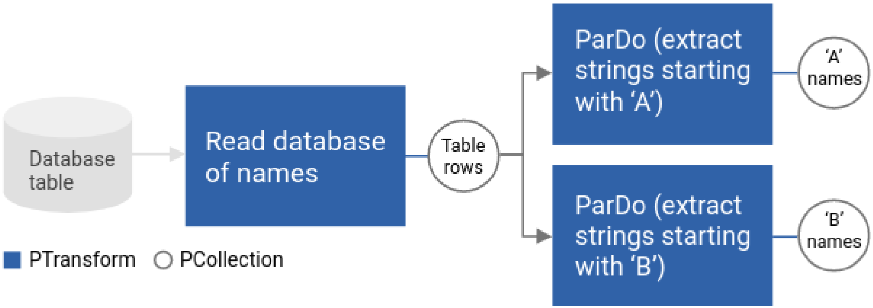
- Where is your input data stored?
  → will define what kind of Read transform to use
- What does your data look like?
  → will define which transform to apply and allow for more efficient data handling
- What do you want to do with your data?
  → will define the transformations, functions etc that you want to apply to your data
- What does your output data look like and where should it go?
  → will define what kind of Write transform to use

Basic Pipeline example



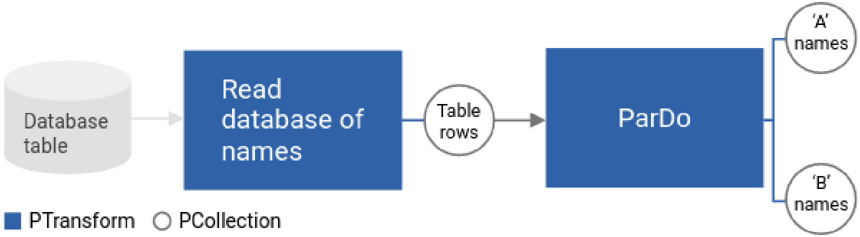
```
[Final Output PCollection] = ([Initial Input PCollection]
                | [First Transform]
                | [Second Transform]
                | [Third Transform])
```

Branching PCollections



```
[PCollection of database table rows] = [Database Table Reader] |
                                       [Read Transform]
[PCollection of 'A' names] = [PCollection of database table rows] |
                             [Transform A]
[PCollection of 'B' names] = [PCollection of database table rows] |
                             [Transform B]
```

Producing multiple outputs (1)



```
results = (
    words
    | beam.ParDo(ProcessWords(), cutoff_length=2,
                                 marker='x').with_outputs(
        'above_cutoff_lengths',
        'marked strings',
         main='below_cutoff_strings'))

below = results.below_cutoff_strings
above = results.above_cutoff_lengths
marked = results['marked strings']  # indexing works as well
```

Producing multiple outputs (2)

OR...

```
below, above, marked = (
    words
    | beam.ParDo(ProcessWords(), cutoff_length=2,
                                 marker='x').with_outputs(
         'above_cutoff_lengths',
         'marked strings',
          main='below_cutoff_strings'))
```

Producing multiple outputs (3)

What does ProcessWords do?

```
class ProcessWords(beam.DoFn):
  def process(self, element, cutoff_length, marker):
    if len(element) <= cutoff_length:
      # Emit this short word to the main output.
      yield element
    else:
      # Emit this word’s long length to the ’above_cutoff_lengths’ output.
      yield pvalue.TaggedOutput(’above_cutoff_lengths’, len(element))
    if element.startswith(marker):
      # Emit this word to a different output with the ’marked strings’ tag.
      yield pvalue.TaggedOutput(’marked strings’, element)
```

Modifying a pipeline to use stream processing

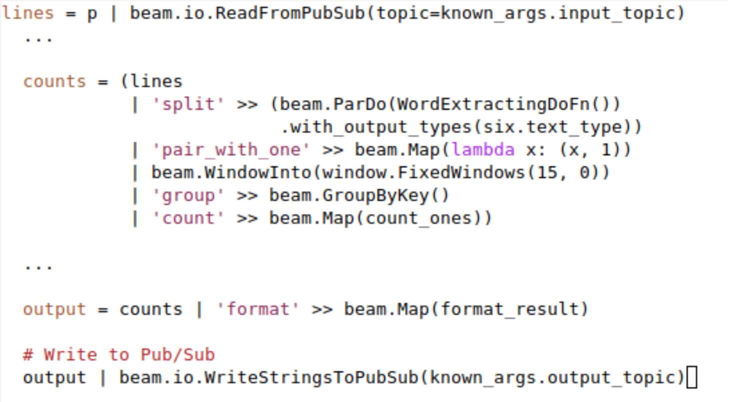
Material from beam python streaming

You need to make the following code changes:

- use an I/O connector that supports reading from an unbounded source → ReadFromText and others do not support unbounded sources!
- use an I/O connector that supports writing to an unbounded source
- choose a windowing strategy

Modifying a pipeline to use stream processing

- beam SDK for python includes 2 of these I/O connectors: Google Cloud PubSub (reading and writing) and Google BigQuery (writing)
- changing code for counting words:



```
lines = p | beam.io.ReadFromPubSub(topic=known_args.input_topic)
  ...

  counts = (lines
            | 'split' >> (beam.ParDo(WordExtractingDoFn())
                          .with_output_types(six.text_type))
            | 'pair_with_one' >> beam.Map(lambda x: (x, 1))
            | beam.WindowInto(window.FixedWindows(15, 0))
            | 'group' >> beam.GroupByKey()
            | 'count' >> beam.Map(count_ones))


  ...

  output = counts | 'format' >> beam.Map(format_result)

  # Write to Pub/Sub
  output | beam.io.WriteStringsToPubSub(known_args.output_topic)
```

Modifying a pipeline to use stream processing

Material from quickstart Google pubsub

- to run a streaming pipeline you must create input and output topics (channels) in the Google Cloud Pub/Sub
- authenticate to the GCP first
- to create a channel called `my-topic`:

  ```
  gcloud pubsub subscriptions create my-sub --topic=my-topic
  ```
- send a msg:

  ```
  gcloud pubsub topics publish my-topic --message="hello"
  ```
- receive the msg:

  ```
  gcloud pubsub subscriptions pull my-sub --auto-ack
  ```

Example

Material from python streaming with GCP

- Sending text through channel my-topic

```
cat amazon_review_polarity_csv/train.csv |
   while read line
   do
      gcloud pubsub topics publish \
            my-topic --message "$line" --limit 30
   done
```

- receiving text (open in another shell)

```
gcloud pubsub subscriptions pull my-sub --auto-ack
```

Modifying a pipeline to use stream processing

- GCP provides a guide to implement stream processing using Pub/Sub (see here)
- communication can be one-to-many (fan-out), many-to-one (fan-in) and many-to-many

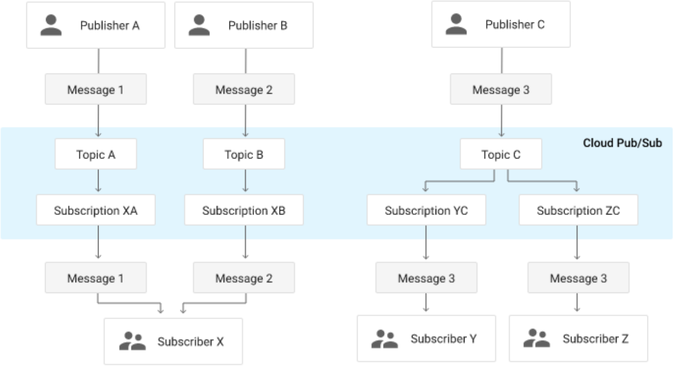


image source: GCP Pub/Sub

Pub/Sub

- GCP provides various streaming templates that can export Pub/Sub data to different destinations:
  - Pub/Sub subscription to BigQuery
  - Pub/Sub to Pub/Sub relay
  - Pub/Sub to Cloud Storage Avro
  - Pub/Sub to Cloud Storage Text
  - Storage Text to Pub/Sub (Stream)

(for templates, see here)