########################### Essential documents of Beam ############################

What is apache beam?

Apache Beam is an open source unified programming model to define and execute data processing pipelines, including ETL, batch and stream (continuous) processing.[2] Beam Pipelines are defined using one of the provided SDKs and executed in one of the Beam’s supported runners (distributed processing back-ends) including Apache Flink, Apache Samza, Apache Spark, and Google Cloud Dataflow.

Source: <https://beam.apache.org/documentation/>

**Map:** Applies a simple 1-to-1 mapping function over each element in the collection.

Eg:

import apache\_beam as beam

with beam.Pipeline() as pipeline:

plants = (

pipeline

| 'Gardening plants' >> beam.Create([

' 🍓Strawberry \n',

' 🥕Carrot \n',

' 🍆Eggplant \n',

' 🍅Tomato \n',

' 🥔Potato \n',

])

| 'Strip' >> beam.Map(str.strip)

| beam.Map(print))

O/P: 🍓Strawberry

🥕Carrot

🍆Eggplant

🍅Tomato

🥔Potato

**Filter:**

Given a predicate, filter out all elements that don’t satisfy that predicate. May also be used to filter based on an inequality with a given value based on the comparison ordering of the element.

Eg:

import apache\_beam as beam

def is\_perennial(plant):

return plant['duration'] == 'perennial'

with beam.Pipeline() as pipeline:

perennials = (

pipeline

| 'Gardening plants' >> beam.Create([

{ 'icon': '🍓', 'name': 'Strawberry', 'duration': 'perennial'

},

{ 'icon': '🥕', 'name': 'Carrot', 'duration': 'biennial'

},

{ 'icon': '🍆', 'name': 'Eggplant', 'duration': 'perennial'

},

{ 'icon': '🍅', 'name': 'Tomato', 'duration': 'annual'

},

{

'icon': '🥔', 'name': 'Potato', 'duration': 'perennial'

}

])

| 'Filter perennials' >> beam.Filter(is\_perennial)

| beam.Map(print))

perennials.run()

O/P:

{'icon': '🍓', 'name': 'Strawberry', 'duration': 'perennial'}

{'icon': '🍆', 'name': 'Eggplant', 'duration': 'perennial'}

{'icon': '🥔', 'name': 'Potato', 'duration': 'perennial'}

**Flat Map:**

Applies a simple 1-to-many mapping function over each element in the collection. The many elements are flattened into the resulting collection.

E.g:

import apache\_beam as beam

with beam.Pipeline() as pipeline:

plants = (

pipeline

| 'Gardening plants' >> beam.Create([

'🍓Strawberry 🥕Carrot 🍆Eggplant',

'🍅Tomato 🥔Potato',

])

| 'Split words' >> beam.FlatMap(str.split)

| beam.Map(print))

O/P:

🍓Strawberry

🥕Carrot

🍆Eggplant

🍅Tomato

🥔Potato

Eg: FlatMap with lambda function:

import apache\_beam as beam

with beam.Pipeline() as pipeline:

plants = (

pipeline

| 'Gardening plants' >> beam.Create([

['🍓Strawberry', '🥕Carrot', '🍆Eggplant'],

['🍅Tomato', '🥔Potato'],

])

| 'Flatten lists' >> beam.FlatMap(lambda elements: elements)

| beam.Map(print))

Eg:

🍓Strawberry

🥕Carrot

🍆Eggplant

🍅Tomato

🥔Potato

**ParDo functions:**

Pardo transform takes each element of the input PCollection performs processing function on it and emits 0,1 or multiple elements

Functionalities:

1. Filtering: ParDo takes each elements of the PCollection and can decide whether to output or discard it
2. Formatting or Type Conversion: Pardo can change the type or Format of each elements
3. Extracting individual elements: ParDo can extract underlying elements from each parent elements
4. Computations:ParDo can perform any processing function on the input element and output PCollection

**DoFn:**

DoFn is a Beam Class that defines distributed processing function. It contains all the logic to run the user provided function parallelly on different machines.

Eg:

*# import apache\_beam as beam*

*# """ DoFn is a Beam Class that defines distributed processing function.*

*#  It contains all the logic to run the user provided function parallelly on different machines."""*

*# class SplitHRRow(beam.DoFn):*

*#     def process(self,element):*

*#         return element.split(",")*

*# class FilterHREmployee(beam.DoFn):*

*#     def process(self,element):*

*#         if element[3] == 'Finance':*

*#             return [element]*

*# class PairHREmployees(beam.DoFn):*

*#     def process(self,element):*

*#         return [(element[3]+","+element[1]),1]*

*# class CountingHR(beam.DoFn):*

*#     def process(self,element):*

*#         (key,value)=element*

*#         return [key,sum(value)]*

class SplitHRRow(beam.DoFn):

  def process(*self*, *element*):

*# return type -> list*

*return*  [*element*.split(',')]

class FilterHREmployee(beam.DoFn):

  def process(*self*, *element*):

*if* *element*[3] == 'HR':

*return* [*element*]

class PairHREmployees(beam.DoFn):

  def process(*self*, *element*):

*return* [(*element*[3]+":"+*element*[1], 1)]

class CountingHR(beam.DoFn):

  def process(*self*, *element*):

*# return type -> list*

    (key, values) = *element*           *# [Marco, HR  [1,1,1,1....] , Rebekah, HR [1,1,1,1,....] ]*

*return* [(key, sum(values))]

p2=beam.Pipeline()

attendance\_count=(

   p2

    |beam.io.ReadFromText('dept\_data\_v1.txt')

    |beam.ParDo(SplitHRRow())

*# | 'Compute WordLength' >> beam.ParDo(lambda element: [ element.split(',') ])*

    |beam.ParDo(FilterHREmployee())

    |beam.ParDo(PairHREmployees())

    | 'Group ' >> beam.GroupByKey()

    | 'Sum using ParDo' >> beam.ParDo(CountingHR())

    |beam.io.WriteToText('data\output\_new\_hr\_final.txt')

)

p2.run()

O/P:

('HR:Beryl', 62)

('HR:Olga', 31)

('HR:Leslie', 31)

('HR:Mindy', 31)

('HR:Vicky', 31)

('HR:Richard', 31)

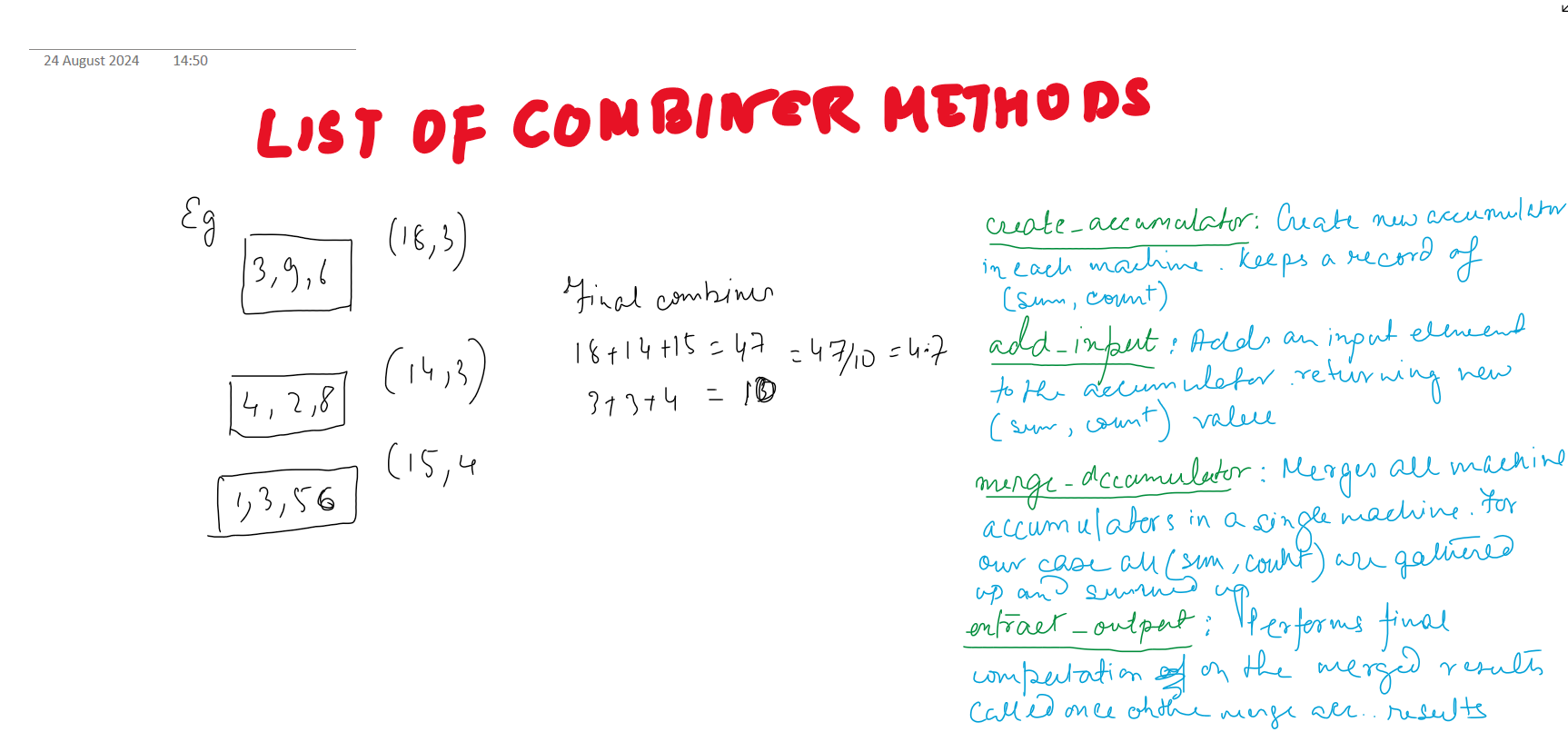
('HR:Kirk', 31)

('HR:Kaori', 31)

('HR:Oscar', 31)

**Combiner**

Combiner is a mini reducer which does the reduce task locally to a mapper machine



*import* apache\_beam *as* beam

class AverageFn(beam.CombineFn):

    def create\_accumulator(*self*):

*return* (0.0, 0)   *# initialize (sum, count)*

    def add\_input(*self*, *sum\_count*, *input*):

        (sum, count) = *sum\_count*

*return* sum + *input*, count + 1

    def merge\_accumulators(*self*, *accumulators*):

        ind\_sums, ind\_counts = zip(\**accumulators*)       *# zip - [(27, 3), (39, 3), (18, 2)]  -->   [(27,39,18), (3,3,2)]*

*return* sum(ind\_sums), sum(ind\_counts)        *# (84,8)*

    def extract\_output(*self*, *sum\_count*):

        (sum, count) = *sum\_count*    *# combine globally using CombineFn*

*return* sum / count *if* count *else* float('NaN')

p=beam.Pipeline()

small\_sum = (

                p

                 | beam.Create([15,5,7,7,9,23,13,5])

                 | "Combine Globally" >> beam.CombineGlobally(AverageFn())

                 | "Write output" >> beam.io.WriteToText("data\combine.txt")

)

p.run()

**Composite Transformation:**

Transforms can have a nested structure, where a complex transform performs multiple simpler transforms (such as more than one ParDo, Combine, GroupByKey, or even other composite transforms). These transforms are called composite transforms. Nesting multiple transforms inside a single composite transform can make your code more modular and easier to understand.

E.g:

*import* apache\_beam *as* beam

class MyTransform(beam.PTransform):

  def expand(*self*, *input\_coll*):

    a = (

*input\_coll*

                       | 'Group and sum1' >> beam.CombinePerKey(sum)

                       | 'count filter accounts' >> beam.Filter(filter\_on\_count)

                       | 'Regular accounts employee' >> beam.Map(format\_output)

    )

*return* a

def SplitRow(*element*):

*return* *element*.split(',')

def filter\_on\_count(*element*):

  name, count = *element*

*if* count > 30:

*return* *element*

def format\_output(*element*):

  name, count = *element*

*#return ', '.join((name.encode('ascii'),str(count),'Regular employee'))*

*return* ', '.join((name,str(count),'Regular employee'))

p = beam.Pipeline()

input\_collection = (

                      p

                      | "Read from text file" >> beam.io.ReadFromText('dept\_data\_v1.txt')

                      | "Split rows" >> beam.Map(SplitRow)

                   )

accounts\_count = (

                      input\_collection

                      | 'Get all Accounts dept persons' >> beam.Filter(lambda *record*: *record*[3] == 'Accounts')

                      | 'Pair each accounts employee with 1' >> beam.Map(lambda *record*: ("Accounts, " +*record*[1], 1))

                      | 'composite accounts' >> MyTransform()

                      | 'Write results for account' >> beam.io.WriteToText('data/Account')

                 )

hr\_count = (

                input\_collection

                | 'Get all HR dept persons' >> beam.Filter(lambda *record*: *record*[3] == 'HR')

                | 'Pair each hr employee with 1' >> beam.Map(lambda *record*: ("HR, " +*record*[1], 1))

                | 'composite HR' >> MyTransform()

                | 'Write results for hr' >> beam.io.WriteToText('data/HR')

           )

p.run()

*# # Sample the first 20 results, remember there are no ordering guarantees.*

*# !{('head -n 20 data/Account-00000-of-00001')}*

*# !{('head -n 20 data/HR-00000-of-00001')}*

O/P:

Accounts, Marco, 31, Regular employee

Accounts, Rebekah, 31, Regular employee

Accounts, Itoe, 31, Regular employee

Accounts, Edouard, 31, Regular employee

Accounts, Kyle, 62, Regular employee

Accounts, Kumiko, 31, Regular employee

Accounts, Gaston, 31, Regular employee

**CoGroupByKey:**

Aggregates all input elements by their key and allows downstream processing to consume all values associated with the key. While GroupByKey performs this operation over a single input collection and thus a single type of input values, CoGroupByKey operates over multiple input collections. As a result, the result for each key is a tuple of the values associated with that key in each input collection.

Easy Eg:

In the following example, we create a pipeline with two PCollections of produce, one with icons and one with durations, both with a common key of the produce name. Then, we apply CoGroupByKey to join both PCollections using their keys.

CoGroupByKey expects a dictionary of named keyed PCollections, and produces elements joined by their keys. The values of each output element are dictionaries where the names correspond to the input dictionary, with lists of all the values found for that key.

import apache\_beam as beam

with beam.Pipeline() as pipeline:

icon\_pairs = pipeline | 'Create icons' >> beam.Create([

('Apple', '🍎'),

('Apple', '🍏'),

('Eggplant', '🍆'),

('Tomato', '🍅'),

])

duration\_pairs = pipeline | 'Create durations' >> beam.Create([

('Apple', 'perennial'),

('Carrot', 'biennial'),

('Tomato', 'perennial'),

('Tomato', 'annual'),

])

plants = (({

'icons': icon\_pairs, 'durations': duration\_pairs

})

| 'Merge' >> beam.CoGroupByKey()

| beam.Map(print))

O/P:

('Apple', {'icons': ['🍎', '🍏'], 'durations': ['perennial']})

('Carrot', {'icons': [], 'durations': ['biennial']})

('Tomato', {'icons': ['🍅'], 'durations': ['perennial', 'annual']})

('Eggplant', {'icons': ['🍆'], 'durations': []})

Real life example with dept data and location:

*import* apache\_beam *as* beam

def retTuple(*element*):

  thisTuple=*element*.split(',')

*return* (thisTuple[0],thisTuple[1:])

p1 = beam.Pipeline()

*# Apply a ParDo to the PCollection "words" to compute lengths for each word.*

dep\_rows = (

                p1

                | "Reading File 1" >> beam.io.ReadFromText('dept\_data\_v2.txt')

                | 'Pair each employee with key' >> beam.Map(retTuple)          *# {149633CM : [Marco,10,Accounts,1-01-2019]}*

               )

loc\_rows = (

                p1

                | "Reading File 2" >> beam.io.ReadFromText('location.txt')

                | 'Pair each loc with key' >> beam.Map(retTuple)                *# {149633CM : [9876843261,New York]}*

               )

results = ({'dep\_data': dep\_rows, 'loc\_data': loc\_rows}

           | beam.CoGroupByKey()

           | 'Write results' >> beam.io.WriteToText('data/result')

          )

p1.run()

*#!{('head -n 20 data/result-00000-of-00001')}*

O/P:

(**'149633CM'**, {**'dep\_data'**: [['Marco', '10', 'Accounts', '1-01-2019'], ['Marco', '10', 'Accounts', '2-01-2019'], ['Marco', '10', 'Accounts', '3-01-2019'], ['Marco', '10', 'Accounts', '4-01-2019'], ['Marco', '10', 'Accounts', '5-01-2019'], ['Marco', '10', 'Accounts', '6-01-2019'], ['Marco', '10', 'Accounts', '7-01-2019'], ['Marco', '10', 'Accounts', '8-01-2019'], ['Marco', '10', 'Accounts', '9-01-2019'], ['Marco', '10', 'Accounts', '10-01-2019'], ['Marco', '10', 'Accounts', '11-01-2019'], ['Marco', '10', 'Accounts', '12-01-2019'], ['Marco', '10', 'Accounts', '13-01-2019'], ['Marco', '10', 'Accounts', '14-01-2019'], ['Marco', '10', 'Accounts', '15-01-2019'], ['Marco', '10', 'Accounts', '16-01-2019'], ['Marco', '10', 'Accounts', '17-01-2019'], ['Marco', '10', 'Accounts', '18-01-2019'], ['Marco', '10', 'Accounts', '19-01-2019'], ['Marco', '10', 'Accounts', '20-01-2019'], ['Marco', '10', 'Accounts', '21-01-2019'], ['Marco', '10', 'Accounts', '22-01-2019'], ['Marco', '10', 'Accounts', '23-01-2019'], ['Marco', '10', 'Accounts', '24-01-2019'], ['Marco', '10', 'Accounts', '25-01-2019'], ['Marco', '10', 'Accounts', '26-01-2019'], ['Marco', '10', 'Accounts', '27-01-2019'], ['Marco', '10', 'Accounts', '28-01-2019'], ['Marco', '10', 'Accounts', '29-01-2019'], ['Marco', '10', 'Accounts', '30-01-2019'], ['Marco', '10', 'Accounts', '31-01-2019']], **'loc\_data'**: [['9876843261', 'New York'], ['9204232778', 'New York']]})

(**'212539MU'**, {**'dep\_data'**: [['Rebekah', '10', 'Accounts', '1-01-2019'], ['Rebekah', '10', 'Accounts', '2-01-2019'], ['Rebekah', '10', 'Accounts', '3-01-2019'], ['Rebekah', '10', 'Accounts', '4-01-2019'], ['Rebekah', '10', 'Accounts', '5-01-2019'], ['Rebekah', '10', 'Accounts', '6-01-2019'], ['Rebekah', '10', 'Accounts', '7-01-2019'], ['Rebekah', '10', 'Accounts', '8-01-2019'], ['Rebekah', '10', 'Accounts', '9-01-2019'], ['Rebekah', '10', 'Accounts', '10-01-2019'], ['Rebekah', '10', 'Accounts', '11-01-2019'], ['Rebekah', '10', 'Accounts', '12-01-2019'], ['Rebekah', '10', 'Accounts', '13-01-2019'], ['Rebekah', '10', 'Accounts', '14-01-2019'], ['Rebekah', '10', 'Accounts', '15-01-2019'], ['Rebekah', '10', 'Accounts', '16-01-2019'], ['Rebekah', '10', 'Accounts', '17-01-2019'], ['Rebekah', '10', 'Accounts', '18-01-2019'], ['Rebekah', '10', 'Accounts', '19-01-2019'], ['Rebekah', '10', 'Accounts', '20-01-2019'], ['Rebekah', '10', 'Accounts', '21-01-2019'], ['Rebekah', '10', 'Accounts', '22-01-2019'], ['Rebekah', '10', 'Accounts', '23-01-2019'], ['Rebekah', '10', 'Accounts', '24-01-2019'], ['Rebekah', '10', 'Accounts', '25-01-2019'], ['Rebekah', '10', 'Accounts', '26-01-2019'], ['Rebekah', '10', 'Accounts', '27-01-2019'], ['Rebekah', '10', 'Accounts', '28-01-2019'], ['Rebekah', '10', 'Accounts', '29-01-2019'], ['Rebekah', '10', 'Accounts', '30-01-2019'], ['Rebekah', '10', 'Accounts', '31-01-2019']], **'loc\_data'**: [['9995440673', 'Denver']]})

(**'231555ZZ'**, {**'dep\_data'**: [['Itoe', '10', 'Accounts', '1-01-2019'], ['Itoe', '10', 'Accounts', '2-01-2019'], ['Itoe', '10', 'Accounts', '3-01-2019'], ['Itoe', '10', 'Accounts', '4-01-2019'], ['Itoe', '10', 'Accounts', '5-01-2019'], ['Itoe', '10', 'Accounts', '6-01-2019'], ['Itoe', '10', 'Accounts', '7-01-2019'], ['Itoe', '10', 'Accounts', '8-01-2019'], ['Itoe', '10', 'Accounts', '9-01-2019'], ['Itoe', '10', 'Accounts', '10-01-2019'], ['Itoe', '10', 'Accounts', '11-01-2019'], ['Itoe', '10', 'Accounts', '12-01-2019'], ['Itoe', '10', 'Accounts', '13-01-2019'], ['Itoe', '10', 'Accounts', '14-01-2019'], ['Itoe', '10', 'Accounts', '15-01-2019'], ['Itoe', '10', 'Accounts', '16-01-2019'], ['Itoe', '10', 'Accounts', '17-01-2019'], ['Itoe', '10', 'Accounts', '18-01-2019'], ['Itoe', '10', 'Accounts', '19-01-2019'], ['Itoe', '10', 'Accounts', '20-01-2019'], ['Itoe', '10', 'Accounts', '21-01-2019'], ['Itoe', '10', 'Accounts', '22-01-2019'], ['Itoe', '10', 'Accounts', '23-01-2019'], ['Itoe', '10', 'Accounts', '24-01-2019'], ['Itoe', '10', 'Accounts', '25-01-2019'], ['Itoe', '10', 'Accounts', '26-01-2019'], ['Itoe', '10', 'Accounts', '27-01-2019'], ['Itoe', '10', 'Accounts', '28-01-2019'], ['Itoe', '10', 'Accounts', '29-01-2019'], ['Itoe', '10', 'Accounts', '30-01-2019'], ['Itoe', '10', 'Accounts', '31-01-2019']], **'loc\_data'**: [['9196597290', 'Boston']]})

**Side Input**

In addition to the main input PCollection, you can provide additional inputs to a ParDo transform in the form of side inputs. A **side input** is an additional input that your DoFn can access each time it processes an element in the input PCollection. When you specify a side input, you create a view of some other data that can be read from within the ParDo transform’s DoFn while processing each element.

**Side inputs** are useful if your ParDo needs to inject additional data when processing each element in the input PCollection, but the additional data needs to be determined at runtime (and not hard-coded). Such values might be determined by the input data, or depend on a different branch of your pipeline.

Additional Input provided to a DoFn object

Can be provided to a ParaDo function or a child Transformation(Map,Filter etc)

E.g:

Side Input:

Do attendace count with below conditions:

1. Accounts dept employees excluding few

2. Also employee name should be between 3-10 charecters

*import* apache\_beam *as* beam

side\_list=[]

*with* open('exclude\_list.txt','r') *as* file:

*for* line *in* file:

        side\_list.append(line.rstrip())

p =beam.Pipeline()

class FilterUsingLength(beam.DoFn):

    def process(*self*,*element*,*side\_list*,*lower\_bound*,*upper\_bound*=float('inf')):

        id=*element*.split(',')[0]

        name=*element*.split(',')[1]

*#id=id.decode('utf-8','ignore').encode("utf-8")*

        element\_list=*element*.split(',')

*if* (*lower\_bound*<=len(name)<=*upper\_bound* and id not in *side\_list*):

*return* [element\_list]

small\_length=(

            p

             |"Read from the text">> beam.io.ReadFromText("dept\_data\_v2.txt")

             |"Filter the Rows on basis of name length">>beam.ParDo(FilterUsingLength(),side\_list,3,10)

             |"Filter accounts people" >> beam.Filter(lambda *record*: *record*[3]=="Accounts" )

             |"Combine the Key of Id and Name and assign them a count" >> beam.Map(lambda *record*:(*record*[0]+" "+*record*[1],1))

             |"Sum of combination of Id and Key" >> beam.CombinePerKey(sum)

             |"Write the output in a result text file" >> beam.io.WriteToText("data\output\_side\_input.txt")

)

p.run()

O/P:

('503996WI Edouard', 31)

('957149WC Kyle', 31)

('241316NX Kumiko', 31)

('796656IE Gaston', 31)

('718737IX Ayumi', 30)

**Additional Outputs**

*import* apache\_beam *as* beam

*# DoFn function*

class ProcessWords(beam.DoFn):

  def process(*self*, *element*, *cutoff\_length*, *marker*):

    name = *element*.split(',')[1]

*if* len(name) <= *cutoff\_length*:

*return* [beam.pvalue.TaggedOutput('Short\_Names', name)]

*else*:

*return* [beam.pvalue.TaggedOutput('Long\_Names', name)]

*if* name.startswith(marker):

*return* name

p = beam.Pipeline()

results = (

            p

            | beam.io.ReadFromText('dept\_data.txt')

            | beam.ParDo(ProcessWords(), *cutoff\_length*=4, *marker*='A').with\_outputs('Short\_Names', 'Long\_Names', *main*='Names\_A')

          )

short\_collection = results.Short\_Names

long\_collection = results.Long\_Names

startA\_collection = results.Names\_A

*# write to file*

short\_collection | 'Write 1'>> beam.io.WriteToText('short')

*# write to file*

long\_collection | 'Write 2'>> beam.io.WriteToText('long')

*# write to file*

startA\_collection | 'Write 3'>> beam.io.WriteToText('start\_a')

p.run()

!{'head -n 5 short-00000-of-00001'}

!{'head -n 5 long-00000-of-00001'}

!{'head -n 5 start\_a-00000-of-00001'}

**Coder in pipeline**

However, the coder itself doesn't exist in the processing pipeline. It's exposed through another object called **CoderRegistry**. As its name indicates, the object represents a registry with all coders that can be used in the processing.

But the coder is not created by the registry. The creation is delegated to another object called **CoderProvider**. Different providers are registered and it's the order of registration that determines what provider will be responsible for producing the coder for a particular class or type. In the first place are used providers registered explicitly. After them are used the providers related to common Java types. At the end of resolution chain are placed the providers registered automatically with ServiceLoader.

Among the implementations of CoderProvider we can distinguish: a provider for common Java types (**CommonTypes**, a provider for Java's serializable types (**SerializableCoderProvider**) or for Protobuf messages (**ProtoCoderProvider**).

The coder in the pipeline is resolved before the physical execution. It's used during the construction of transform hierarchy (read more about in the post [TransformHierarchy in Apache Beam](https://www.waitingforcode.com/apache-beam/transformhierarchy-apache-beam/read" \t "_blank)), when the outputs for each transforms are built.

Code:

*from* apache\_beam *import* coders

coders.registry.get\_coder(int)

O/P:

VarIntCoder

Str: StrUtf8Coder

Custom Coder:

# CLASS CHANGE FRENCH CHARACTERS

class IgnoreUnicode(Coder):

def encode(self, value):

return value.encode('utf-8','ignore') #encode objects into byte stream

def decode(self, value):

return value.decode('utf-8','ignore') #decode bytestream into objects

def is\_deterministic(self):

return True # deterministic means values and datas will remain same across distributed machines which yields same value for scenarios like wide transformations eg.Group By or CoGroupby

Type Safety:

With type safety, programming languages prevents type errors, or we can say that type safety means the compiler will validate type while compiling, and throw an error when we try to assign a wrong type to a variable

Example:

*from* apache\_beam *import* beam

p=beam.Pipeline()

type\_hint=(

    p

     | "create word array" >> beam.create(["one","two","three"])

     | "Map the values" >> beam.Filter(lambda *x*:*x*%2==0).with\_input\_type(int)

)

p.run()

O/P:

Error: **TypeCheckError**

*# import beam module*

*import* apache\_beam *as* beam

p = beam.Pipeline()

@beam.typehints.with\_input\_types(int)

class FilterEvensDoFn(beam.DoFn):

  def process(*self*, *element*):

*if* *element* % 2 == 0:

*yield* *element*

evens = ( p

         | beam.Create(['1','2','3'])

         | beam.ParDo(FilterEvensDoFn())

        )

p.run()

It will also give same error

Example where Typehint and coders is useful:

import apache\_beam as beam

import typing

class Employee(object):

def \_\_init\_\_(self, id, name):

self.id = id

self.name = name

class EmployeeCoder(beam.coders.Coder):

def encode(self, employee):

return ('%s:%s' % (employee.id, employee.name)).encode('utf-8')

def decode(self, s):

return Employee(\*s.decode('utf-8').split(':'))

def is\_deterministic(self):

return True

beam.coders.registry.register\_coder(Employee, EmployeeCoder)

def split\_file(input):

name, id, salary = input.split(',')

return Employee(id, name), int(salary)

result = (

p

| beam.io.ReadFromText('data.txt')

| beam.Map(split\_file)

| beam.CombinePerKey(sum).with\_input\_types(typing.Tuple[Employee, int])

)

p.run()