Beam YAML: Advanced topics

Presented by:
Jeff Kinard

Software Engineer at Google
Working on Apache Beam and Dataflow

BEAM SUMMIT

September 4-5, 2024

Sunnyvale, CA. USA

Agenda

01

Background

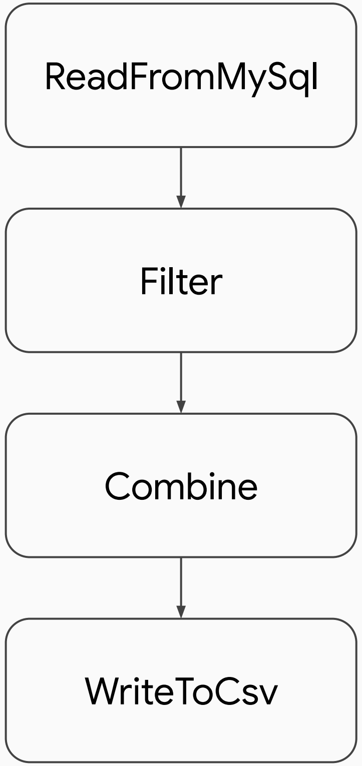
BEAM SUMMIT

What is Beam YAML?

- Apache Beam’s newest SDK
- Declarative YAML syntax
  - Effortless pipeline construction in no-code (or low-code) environment
  - Easily copy, modify, share existing YAML components
  - Better maintainability
- Leverage existing powerful Beam features
  - Rich IO’s
  - Turnkey transforms

Example pipeline

```yaml
pipeline:
  type: chain
  transforms:
    - type: ReadFromMySql
      config:
        url: jdbc:mysql://host:port/database
        table: transactions
        username: 'username'
        password: 'password'
    - type: Filter
      config:
        language: python
        keep: category == "Electronics"
    - type: Combine
      name: CountNumberSold
      input: FilterWithCategory
      config:
        group_by: product_name
        combine:
          num_sold:
            value: product_name
            fn: count
    - type: WriteToCsv
      config:
        path: electronics.csv
```



02

Mapping Transforms

Mapping Transforms

- MapToFields
  - Map the input collection to a schema where each field can be defined by a UDF (user-defined function)
    - Expressions (Generic/Python/Java/SQL/JS*)
    - Callables (Python/Java/JS*)
    - File (Python/Java/JS*)
- Filter
  - Filter records in a collection given a predicate
    - Same expression, callable, file capabilities as MapToFields
- Explode
  - Produce elements for each iterable field specified
    - {name='a', iter=[1, 2, 3]} ⟶ {name='a', iter=1}, {name='a', iter=2}, and {name='a', iter=3}
- Partition
  - Split input collection into multiple output collections based on condition
- AssignTimestamps
  - Mark field in collection as Timestamp - useful for streaming pipeline with embedded timestamps

* JavaScript support is experimental

Generic MapToFields

```yaml
- type: MapToFields
  name: RenameAndMapCustomFields
  input: ReadFromCsv
  config:
    fields:
      new_col: col1
      int_literal: 389
      float_literal: 1.90216
      str_literal: '"example"'
```

col1
1
2
3

new_col	int_literal	float_literal	str_literal
1	389	1.90216	"example"
2	389	1.90216	"example"
3	389	1.90216	"example"

MapToFields

```yaml
- type: MapToFields
  name: RenameAndMapCustomFields
  input: ReadFromCsv
  config:
    language: python
    fields:
      myNewStr:
        expression: "myOldStr"
      myNewNum:
        callable: "lambda row: row.myOldNum * 2"
      myNewName:
        path: "udf.py"
        name: "to_uppercase"
```

myOldNum	myOldStr	myOldName
1	“a”	“John”
2	“b”	“Jane”
3	“c”	“Apache Beam”

$\downarrow$

myNewNum	myNewStr	myNewName
2	“a”	“JOHN”
4	“b”	“JANE”
6	“c”	“APACHE BEAM”

BEAM SUMMIT

MapToFields Callable

```
- type: MapToFields
  name: RenameAndMapCustomFields
  input: ReadFromCsv
  config:
    language: python
    fields:
      json_str:
        callable: |
          import json
          def process(row):
            json_str = json.dumps(row._asdict())
            return json_str
```

col1	col2	col3
1	“a”	“John”
2	“b”	“Jane”
3	“c”	“Apache Beam”

↓

json_str
'{"col1": 1, "col2": "a", "col3": "John"}'
'{"col1": 2, "col2": "b", "col3": "Jane"}'
'{"col1": 3, "col2": "c", "col3": "Apache Beam"}'

BEAM SUMMIT

MapToFields Output Types

```
- type: MapToFields
  name: RenameAndMapCustomFields
  input: ReadFromCsv
  config:
    language: python
    fields:
      json_str: # -------> Any
        callable: |
          import json
          def process(row):
            json_str = json.dumps(row._asdict())
            return json_str
- type: SomeJavaTransform
  config:
    ...
```

```
java.lang.IllegalArgumentException:
Failed to decode Schema due to an error
decoding Field proto:

name: "json_str"
type {
  nullable: true
  logical_type {
    urn: "beam:logical:pythonsdk_any:v1"
  }
}
```

MapToFields Output Types

```
- type: MapToFields
  name: RenameAndMapCustomFields
  input: ReadFromCsv
  config:
    language: python
    fields:
      json_str:
        callable: |
          import json
          def process(row):
            json_str = json.dumps(row._asdict())
            return json_str
        output_type: string # -------> String
- type: SomeJavaTransform
  config:
    ...
```

✓

MapToFields Create Schema

```
- type: MapToFields
  name: RenameAndMapCustomFields
  input: ReadFromCsv
  config:
    language: python
    fields:
      col1:
        callable: 'lambda row: row.col1'
        output_type: integer
      col2:
        callable: 'lambda row: row.col2'
        output_type: string
      col3:
        callable: 'lambda row: row.col3'
        output_type: string
```

col1	col2	col3
1	"a"	"John"
2	"b"	"Jane"
3	"c"	"Apache Beam"

↓

col1	col2	col3
1	"a"	"John"
2	"b"	"Jane"
3	"c"	"Apache Beam"

BEAM SUMMIT

MapToFields Java

```
- type: MapToFields
  name: RenameAndMapCustomFields
  input: ReadFromCsv
  config:
    language: java
    fields:
      myNewStr:
        expression: "myOldStr"
      myNewNum:
        callable: |
          import org.apache.beam.sdk.values.Row;
          import java.util.function.Function;
          public class MyFunction implements Function<Row, String> {
            public String apply(Row row) {
              return row.getString("myOldName").toUpperCase();
            }
          }
      myNewName:
        path: "udf.java"
        name: "to_uppercase"
```

myOldNum	myOldStr	myOldName
1	“a”	“John”
2	“b”	“Jane”
3	“c”	“Apache Beam”

$\downarrow$

myNewNum	myNewStr	myNewName
2	“a”	“JOHN”
4	“b”	“JANE”
6	“c”	“APACHE BEAM”

BEAM SUMMIT

MapToFields SQL

```sql
SELECT
  `timestamp`,
  UPPER(myOldName) AS myNewName,
  "myOldNum + 1" AS myNewNum
FROM PCOLLECTION;
```

```yaml
- type: MapToFields
  name: RenameAndMapCustomFields
  input: ReadFromCsv
  config:
    language: sql
    fields:
      timestamp:
        expression: "`timestamp`"
      myNewNum:
        expression: "myOldNum + 1"
      myNewName:
        expression: "UPPER(myOldName)"
```

timestamp	myOldNum	myOldName
1	1	“John”
2	2	“Jane”
3	3	“Apache Beam”

timestamp	myNewNum	myNewName
1	2	“JOHN”
2	3	“JANE”
3	4	“APACHE BEAM”

BEAM SUMMIT

03

Aggregation Transforms

Aggregation Transforms

- Combine
  - Aggregate the input collection according to a given aggregation method
    - Built-in
      - sum, max, min, all, any, mean, count, group, concat
    - Custom transform
      - Some function that implements `core.CombineFn`
  - Supports multiple languages - Python, SQL

Basic Combine

```yaml
- type: Combine
  config:
    group_by: col1
    combine:
      col2:
        value: col2
        fn:
          type: sum
      count:
        value: col1
        fn:
          type: count
```

col1	col2
'a'	1
'b'	2
'a'	3

col1	col2	count
'a'	4	2
'b'	2	1

Basic Combine

```
- type: Combine
  config:
    group_by: col1
    combine:
      col2:
        value: col2
        fn: sum
      count:
        value: col1
        fn: count
```

col1	col2
'a'	1
'b'	2
'a'	3

$\downarrow$

col1	col2	count
'a'	4	2
'b'	2	1

Basic Combine

```yaml
- type: Combine
  config:
    group_by: col1
    combine:
      col2: sum
      count:
        value: col1
        fn: count
```

col1	col2
'a'	1
'b'	2
'a'	3

col1	col2	count
'a'	4	2
'b'	2	1

SQL Combine

```yaml
- type: Combine
  config:
    language: sql
    group_by: id
    combine:
      num_values: "count(*)"
      total: "sum(col1)"
```

id	col1
1	1
2	2
1	3

id	num_values	total
1	2	4
2	1	2

BEAM SUMMIT

Custom Combine Fn

```yaml
- type: Combine
  config:
    language: python
    group_by: id
    combine:
      top_two:
        value: "col1 + col2"
        fn:
          type: 'apache_beam.transforms.combiners.TopCombineFn'
          config:
            n: 2
```

id	col1	col2
1	1	5
2	2	6
1	3	7
1	4	8

id	top_two
1	[12,10]
2	[8]

A set of Beam's built-in CombineFn's can be found at
https://beam.apache.org/releases/pydoc/current/apache_beam.transforms.combiners.html

04

Providers

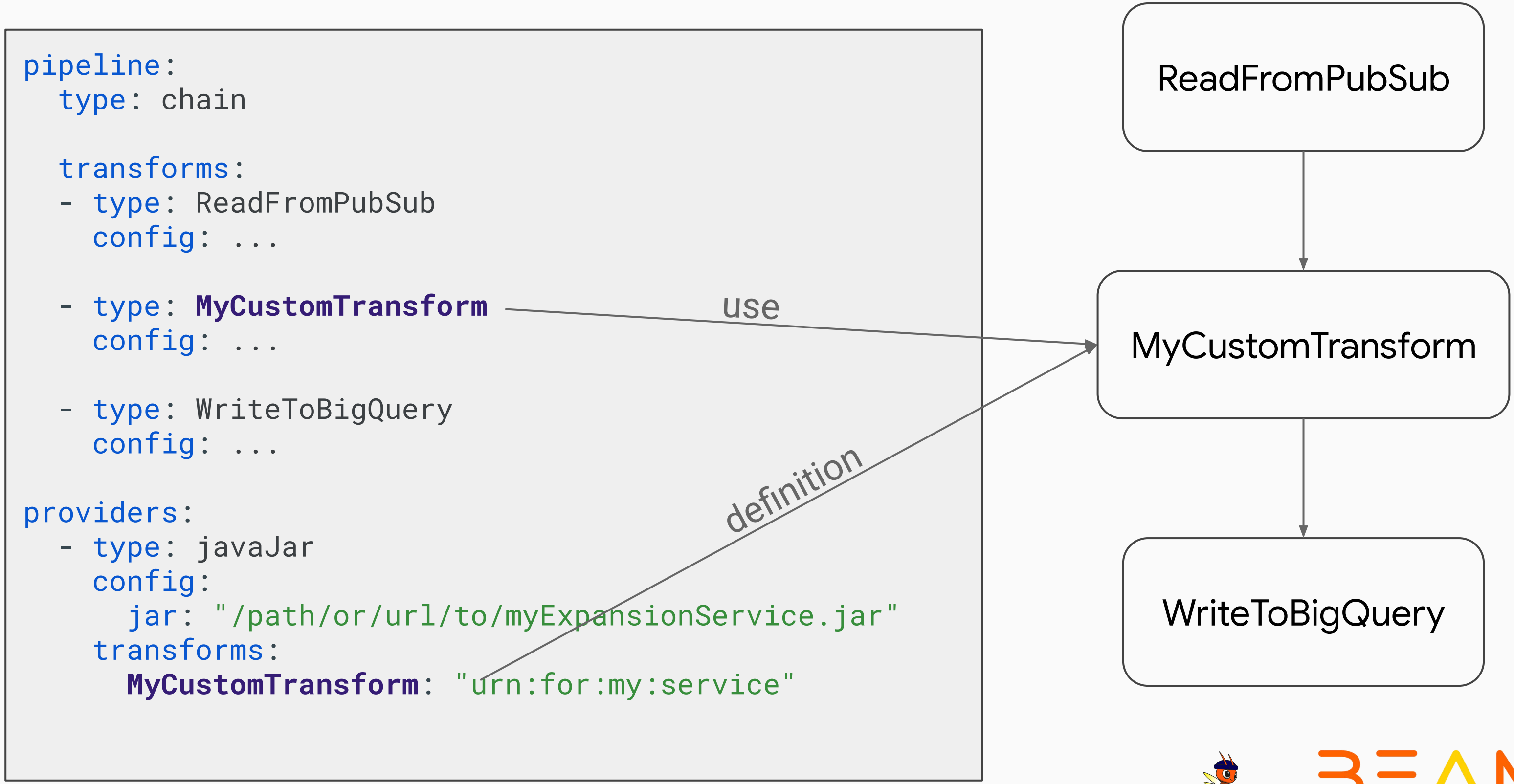
Custom Transforms

Though we aim to provide a rich set of built-in transforms, invariable customers will want to provide their own custom transformations.

- We provide this extensibility via *Providers*
- Allows one to use the full expressivity of Beam SDKs
- Custom transforms can be authored and deployed in multiple ways
  - Inline
  - PyPi packages
  - Java jars
  - Maven/gradle targets
  - YAML
- Customers can use this to provide their own custom transforms, and their users can then reference and use them.

A tutorial on creating a custom Java provider can be found at https://github.com/Polber/beam-yaml-xlang

Provider Example (Java Jar)



```yaml
pipeline:
  type: chain

  transforms:
  - type: ReadFromPubSub
    config: ...

  - type: MyCustomTransform
    config: ...

  - type: WriteToBigQuery
    config: ...

providers:
  - type: javaJar
    config:
      jar: "/path/or/url/to/myExpansionService.jar"
    transforms:
      MyCustomTransform: "urn:for:my:service"
```

05

Inlining Python

Custom Transforms

There are cases where the overhead of supplying a custom packaged provider is not worth the time investment or overhead. In those cases, Beam YAML provides a method for creating in-line Python transforms.

There are currently two ways to implement these in-line Python transforms

- Using PyTransform
- Leveraging the Providers framework (python Provider)

Best Practice:

- Output a Beam Row - outputting schema'd data helps integration with existing Beam YAML transforms

PyTransform

- Typically used to call some arbitrary Transform that is not specifically wrapped for Beam YAML
- This could be a transform built into Beam, or one packaged with the pipeline

```
- type: PyTransform
  config:
    constructor: apache_beam.pkg.module.SomeTransform
    args: [1, 'foo']
    kwargs:
       baz: 3
```

PyTransform __constructor__

```yaml
- type: PyTransform
  config:
    constructor: __constructor__
    kwargs:
      source: |
        class MyPTransform(beam.PTransform):
          def __init__(self, inc):
            self._inc = inc
          def expand(self, pcoll):
            return pcoll | beam.Map(lambda x: beam.Row(out=x.col2 + self._inc))

      inc: 10
```

col1	col2
1	4
2	5
3	6

out
14
15
16

PyTransform __callable__

```yaml
- type: PyTransform
  config:
    constructor: __callable__
    kwargs:
      source: |
        def my_ptransform(pcoll, inc):
          return pcoll | beam.Map(lambda x: beam.Row(out=x.col2 + inc))
      inc: 10
```

col1	col2
1	4
2	5
3	6

out
14
15
16

Python Provider

```
pipeline:
  transforms:
    - ...
    - type: MyTransform
      input: ...
      config:
        inc: 10
    - ...

providers:
  - type: python
    config: {}
    transforms:
      MyTransform: |
        @beam.ptransform_fn
        def my_ptransform(pcoll, inc):
          return pcoll | beam.Map(lambda x: beam.Row(out=x.col2 + inc))
```

col1	col2
1	4
2	5
3	6

out
14
15
16

06

Jinja Templatization

Jinja Templatization

There are many cases where a static YAML file, or components, may be cumbersome to share, negating the benefit of YAML being easy to share across teams.

- Sensitive information embedded (SPII)
- Different teams/users need minor variations (though possibly clearly defined subsets of use-cases)
- Copy/paste-ing YAML blocks can be disorganized and difficult to maintain across an organization
- etc.

Jinja Templatization

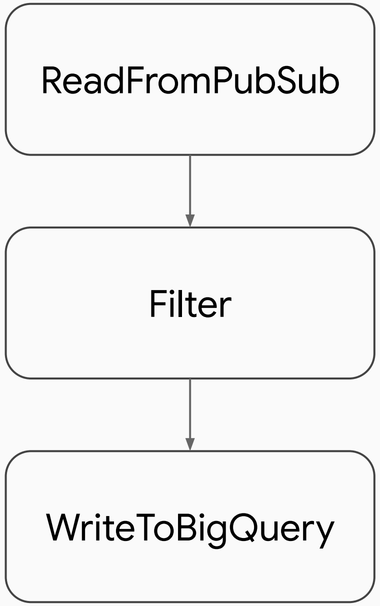
Jinja framework provides standardized method for creating template YAML pipelines

- Inject variables, such as SPII, when running the pipeline, rather than embedding
- Allows for dynamic construction of pipeline graphs based on runtime parameters
- Import central-hosted YAML blocks/files

More information on Jinja syntax can be found at https://jinja.palletsprojects.com/en/3.1.x/templates/#

Variable injection

```
pipeline:
  type: chain
  transforms:
    - type: ReadFromPubSub
      config:
        subscription: ...
        format: ...
        schema: ...
    - type: Filter
      config:
        language: python
        keep: "age > {{threshold}}"
    - type: WriteToBigQuery
      config:
        table: "my_project.my_dataset.my_table_staging"
```

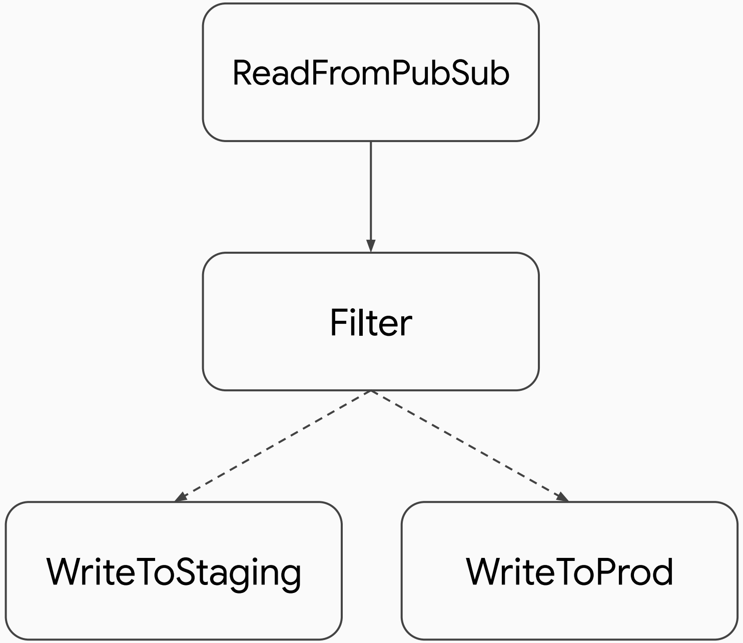


```
python -m apache_beam.yaml.main --yaml_pipeline_file=pipeline.yaml   --jinja_variables='{"threshold": "5"}'
```

Dynamic graph construction

```
pipeline:
  type: chain
  transforms:
    - type: ReadFromPubSub
      config:
        subscription: ...
        format: ...
        schema: ...
    - type: Filter
      config:
        language: python
        keep: "age > {{threshold}}"

{% if use_staging == "true" %}
    - type: WriteToBigQuery
      name: WriteToStaging
      config:
        table: "my_project.my_dataset.my_table_staging"
{% else %}
    - type: WriteToBigQuery
      name: WriteToProd
      config:
        table: "my_project.my_dataset.my_table"
{% endif %}
```



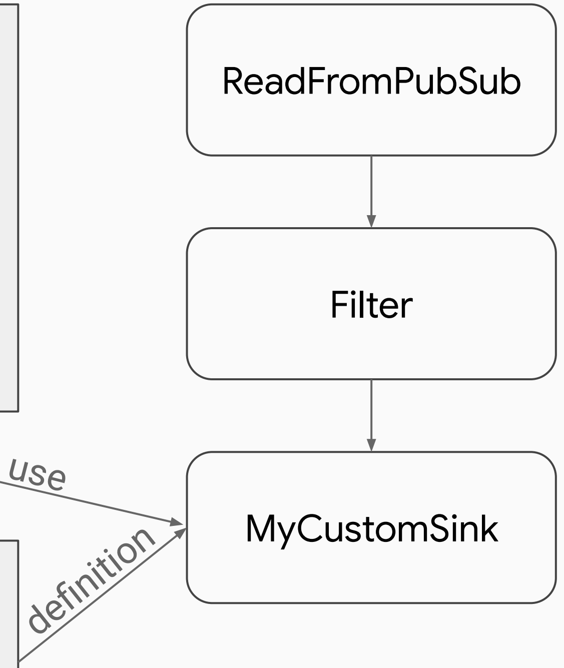
```
python -m apache_beam.yaml.main --yaml_pipeline_file=pipeline.yaml   --jinja_variables='{"threshold": "5", "use_staging":
"true"}'
```

Easily share transforms catalogs

```
{% include 'gs://my-bucket/path/to/providers.yaml' %}
pipeline:
  type: chain
  transforms:
    - type: ReadFromPubSub
      config:
        subscription: ...
        format: ...
        schema: ...
    - type: Filter
      config:
        language: python
        keep: "age > {{threshold}}"
    - type: MyCustomSink
      config:
        ...
```

providers.yaml

```
providers:
  - type: pythonPackage
    config:
       packages:
            - my_pypi_package>=version
            - /path/to/local/package.zip
    transforms:
       MyCustomSink: "pkg.subpkg.PTransformClassOrCallable"
```



More Information

- Beam YAML docs:
  - https://beam.apache.org/documentation/sdks/yaml/
- Beam YAML Getting Started Notebook:
  - https://colab.sandbox.google.com/github/apache/beam/blob/master/examples/notebooks/get-started/try-apache-beam-yaml.ipynb
- Creating a custom Beam Java transform for Beam YAML:
  - https://github.com/Polber/beam-yaml-xlang

Thank you!

Questions?

Please reach out with any questions!

Email:
jkinard@google.com

LinkedIn:
https://www.linkedin.com/in/jeffrey-kinard-92637214a/