Simpler Python Pipelines with Schemas, SQL, and Dataframes

Digital Beam Summit 2020

Robert Bradshaw - Brian Hulette

s.apache.org/simpler-python-pipelines-2020

Who are we?

Brian Hulette

- Software Engineer at Google
- Apache Beam Committer
- github.com/TheNeuralBit
- bhulette@apache.org

Robert Bradshaw

- Software Engineer at Google
- Apache Beam PMC
- github.com/robertwb
- robertwb@apache.org

Agenda

- Demo!
- How to use Schemas, SQL, and Dataframes?
- How do they work?

Using Beam Schemas in Python

Beam Schemas in Python

- SqlTransform, GroupBy, and DataframeTransform require input PCollections to have a schema.
- A schema represents the “type” of structured data. Ordered list of `(name, type)` pairs.
- Currently three ways to achieve this:
  - Register and yield instances of a `NamedTuple` type.
  - Yield `beam.Row` instances.
  - Follow a PTransform that produces a schema.

Defining a Schema - NamedTuple

```
class FruitRecipe(typing.NamedTuple):
  recipe: str
  fruit: str
  quantity: int
  cost: float

coders.registry.register_coder(FruitRecipe, coders.RowCoder)

pc = (p | beam.Create([
    FruitRecipe("pie", "strawberry", 3, 1.5),
    ...,
    FruitRecipe("muffin", "blueberry", 2, 2.),
  ]).with_output_types(FruitRecipe));

pc | SqlTransform("SELECT * FROM PCOLLECTION WHERE quantity > 1"));
```

Defining a Schema - beam.Row

```
pc = (p | beam.Create([
    ("pie", "strawberry", 3, 1.5),
     ...,
    ("muffin", "blueberry", 2, 2.),
  ])
  | beam.Map(lambda x: beam.Row(recipe=x[0],                    # str
                                fruit=x[1],                     # str
                                quantity=x[2],                  # int
                                unit_cost=x[3],                 # float
                                is_berry=x[1].endswith('berry')))) # bool
```

Types inferred by inspecting each field's expression

```
pc | SqlTransform("SELECT * FROM PCOLLECTION WHERE quantity > 1");
```

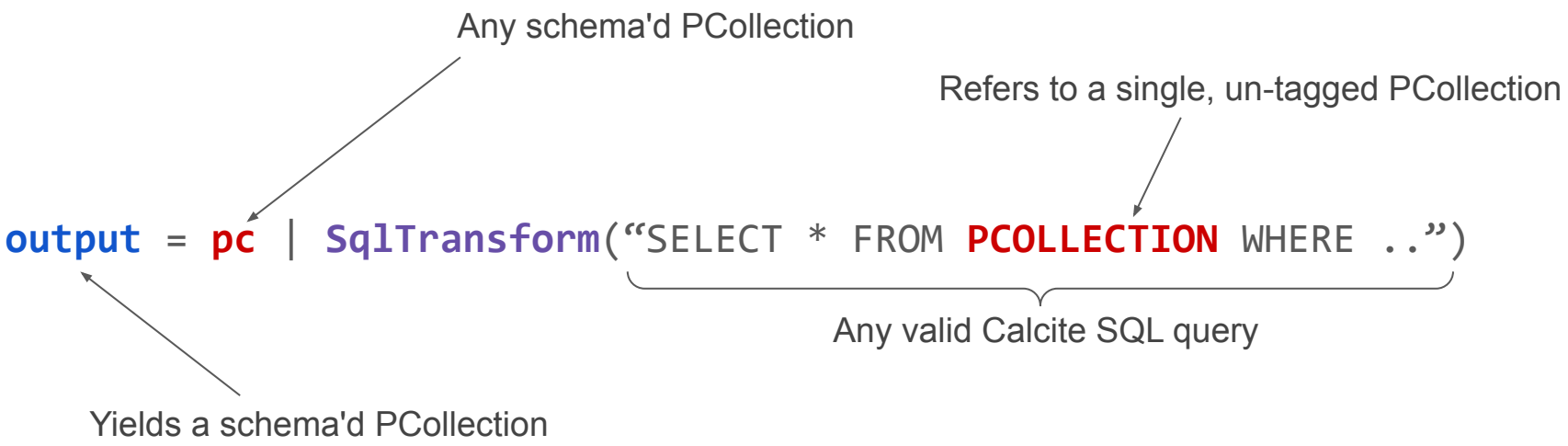
When to use `NamedTuple` vs. `beam.Row`

- `NamedTuple` makes types explicit, which can be better for documenting interfaces and sharing code.
- `beam.Row` may feel more natural and allows for one-off dynamic declarations.
- `beam.Row` relies on inference, it may give up and produce `typing.Any`. Can be helpful to use explicit casts:

```
| beam.Map(lambda x: beam.Row(recipe=str(x[0]),
                              fruit=str(x[1]),
                              quantity=int(x[2]),
                              unit_cost=float(x[3]),
                              is_berry=bool(x[1].endswith('berry'))))
```

SqlTransform

SqlTransform



SqlTransform - Output Type

```
output = pc | SqlTransform(
              “SELECT quantity AS q, unit_cost AS c FROM PCOLLECTION”)

revenue = output | beam.Map(lambda row: row.q * row.c)
```

Access output columns as attributes

SqlTransform - Tagged PCollection

```
output = {a: pc} | SqlTransform(“SELECT * FROM a WHERE ..”)
```

SqlTransform - Join Tagged PCollections

```
output = {a: pc_a, b: pc_b} | SqlTransform(
                              “SELECT * FROM a JOIN b ON a.id = b.id”)
```

SqlTransform - Windowing

```
output = (pc | beam.WindowInto(beam.window.FixedWindows(15))
             # SQL processed per window
             | SqlTransform("SELECT * FROM PCOLLECTION WHERE ..."))
```

End-to-end example of this: sql_taxi.py

SqlTransform Limitations

- Limited type support
  - Available now (2.23.0): INT32, INT64, DOUBLE, STRING, ARRAY, ROW
  - Coming soon (2.24.0): BOOLEAN, BYTES, MAP
  - In progress (2.25.0): TIMESTAMP
- Field names must be valid Python identifiers
  - `"SELECT COUNT(*)"` produces a field named `"$col1"`. It must be renamed, e.g. `"SELECT COUNT(*) AS `count`"`.
- Requires Java runtime

How to learn more

- Python API docs
- More python examples:
  - wordcount_xlang_sql.py
  - sql_taxi.py - process streaming NYC taxi data with windowing
- Beam SQL Overview

GroupBy

GroupBy

```
pc = p | beam.Create(['strawberry', 'raspberry', 'blackberry',
                      'blueberry', 'banana'])

pc | GroupBy(lambda name: name[0])
```

Arbitrary expression

GroupBy

```
pc = p | beam.Create(['strawberry', 'raspberry', 'blackberry',
                      'blueberry', 'banana'])

pc | GroupBy(lambda name: name[0])
```

Arbitrary expression

```
   ('s', ['strawberry'])
   ('r', ['raspberry'])
   ('b', ['blackberry', 'blueberry', 'banana'])
```

GroupBy

Multiple named expressions

```
pc = p | beam.Create(['strawberry', 'raspberry', 'blackberry',
                      'blueberry', 'banana'])

pc | GroupBy(first_letter=lambda name: name[0],
             is_berry=lambda name: 'berry' in name)
```

GroupBy

Multiple named expressions

```
pc = p | beam.Create(['strawberry', 'raspberry', 'blackberry',
                      'blueberry', 'banana'])

pc | GroupBy(first_letter=lambda name: name[0],
             is_berry=lambda name: 'berry' in name)
```

```
((first_letter='s', is_berry=True), ['strawberry'])
((first_letter='r', is_berry=True), ['raspberry'])
((first_letter='b', is_berry=True), ['blackberry', 'blueberry'])
((first_letter='b', is_berry=False), ['banana'])
```

GroupBy

```
pc = p | beam.Create(
```

recipe	fruit	quantity	unit_cost
pie	strawberry	3	$1.50
pie	raspberry	1	$3.50
pie	blackberry	1	$4.00
pie	blueberry	2	$2.00
muffin	banana	3	$2.00
muffin	blueberry	2	$2.00

```
)
```

Shorthand for attributes

```
pc | GroupBy('recipe',
           is_berry=lambda x: 'berry' in x.fruit)
```

GroupBy

```
pc = p | beam.Create( )
```

recipe	fruit	quantity	unit_cost
pie	strawberry	3	$1.50
pie	raspberry	1	$3.50
pie	blackberry	1	$4.00
pie	blueberry	2	$2.00
muffin	banana	3	$2.00
muffin	blueberry	2	$2.00

Shorthand for attributes

```
pc | GroupBy('recipe',
          is_berry=lambda x: 'berry' in x.fruit)
```

Full elements

```
((recipe='pie', is_berry=True), [
```

pie	strawberry	3	$1.50
pie	raspberry	1	$3.50
pie	blackberry	1	$4.00
pie	blueberry	2	$2.00

```
])
```

```
((recipe='muffin', is_berry=True), [
```

muffin	blueberry	2	$2.00

```
])
((recipe='muffin', is_berry=False), [
```

muffin	banana	3	$2.00

```
])
```

GroupBy

```
pc = p | beam.Create(
```

recipe	fruit	quantity	unit_cost
pie	strawberry	3	$1.50
pie	raspberry	1	$3.50
pie	blackberry	1	$4.00
pie	blueberry	2	$2.00
muffin	banana	3	$2.00
muffin	blueberry	2	$2.00

```
)
```

```
pc | GroupBy('recipe')
        .aggregate_field('quantity', sum, 'total_quantity')
```

GroupBy

```
pc = p | beam.Create(
```

recipe	fruit	quantity	unit_cost
pie	strawberry	3	$1.50
pie	raspberry	1	$3.50
pie	blackberry	1	$4.00
pie	blueberry	2	$2.00
muffin	banana	3	$2.00
muffin	blueberry	2	$2.00

```
)
```

```
pc | GroupBy('recipe')
        .aggregate_field('quantity', sum, 'total_quantity')
```

recipe	total_quantity
pie	7
muffin	5

Aggregated and flattened data

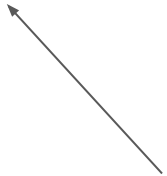
GroupBy

```
pc = p | beam.Create(
```

recipe	fruit	quantity	unit_cost
pie	strawberry	3	$1.50
pie	raspberry	1	$3.50
pie	blackberry	1	$4.00
pie	blueberry	2	$2.00
muffin	banana	3	$2.00
muffin	blueberry	2	$2.00

```
)
```

```
pc | GroupBy('recipe', is_berry=lambda x: 'berry' in x.fruit)
        .aggregate_field('quantity', sum, 'total_quantity')
        .aggregate_field(lambda x: x.quantity*x.unit_price,
                         sum, 'total_price')
```



Multiple expressions and lambdas

GroupBy

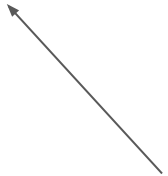
```
pc = p | beam.Create(
```

recipe	fruit	quantity	unit_cost
pie	strawberry	3	$1.50
pie	raspberry	1	$3.50
pie	blackberry	1	$4.00
pie	blueberry	2	$2.00
muffin	banana	3	$2.00
muffin	blueberry	2	$2.00

```
)
```

```
pc | GroupBy('recipe', is_berry=lambda x: 'berry' in x.fruit)
        .aggregate_field('quantity', sum, 'total_quantity')
        .aggregate_field(lambda x: x.quantity*x.unit_price,
                         sum, 'total_price')
```

recipe	is_berry	total_quantity	total_price
pie	True	7	$16.00
muffin	False	3	$6.00
muffin	True	2	$4.00



Multiple expressions and lambdas

GroupBy

```
pc = p | beam.Create(
```

recipe	fruit	quantity	unit_cost
pie	strawberry	3	$1.50
pie	raspberry	1	$3.50
pie	blackberry	1	$4.00
pie	blueberry	2	$2.00
muffin	banana	3	$2.00
muffin	blueberry	2	$2.00

```
)

pc | Map(lambda x: ((x.recipe, 'berry' in x.fruit),
                    (x.quantity, x.quantity*x.unit_price)))
   | CombinePerKey(TupleCombineFn(sum, sum))
   | MapTuple(lambda key, value: Row(
            recipe=key[0], is_berry=key[1],
            total_quantity=value[0], total_price=value[1]))
```

recipe	is_berry	total_quantity	total_price
pie	True	7	$16.00
muffin	False	3	$6.00
muffin	True	2	$4.00

GroupBy

```
pc = p | beam.Create(
```

recipe	fruit	quantity	unit_cost
pie	strawberry	3	$1.50
pie	raspberry	1	$3.50
pie	blackberry	1	$4.00
pie	blueberry	2	$2.00
muffin	banana	3	$2.00
muffin	blueberry	2	$2.00

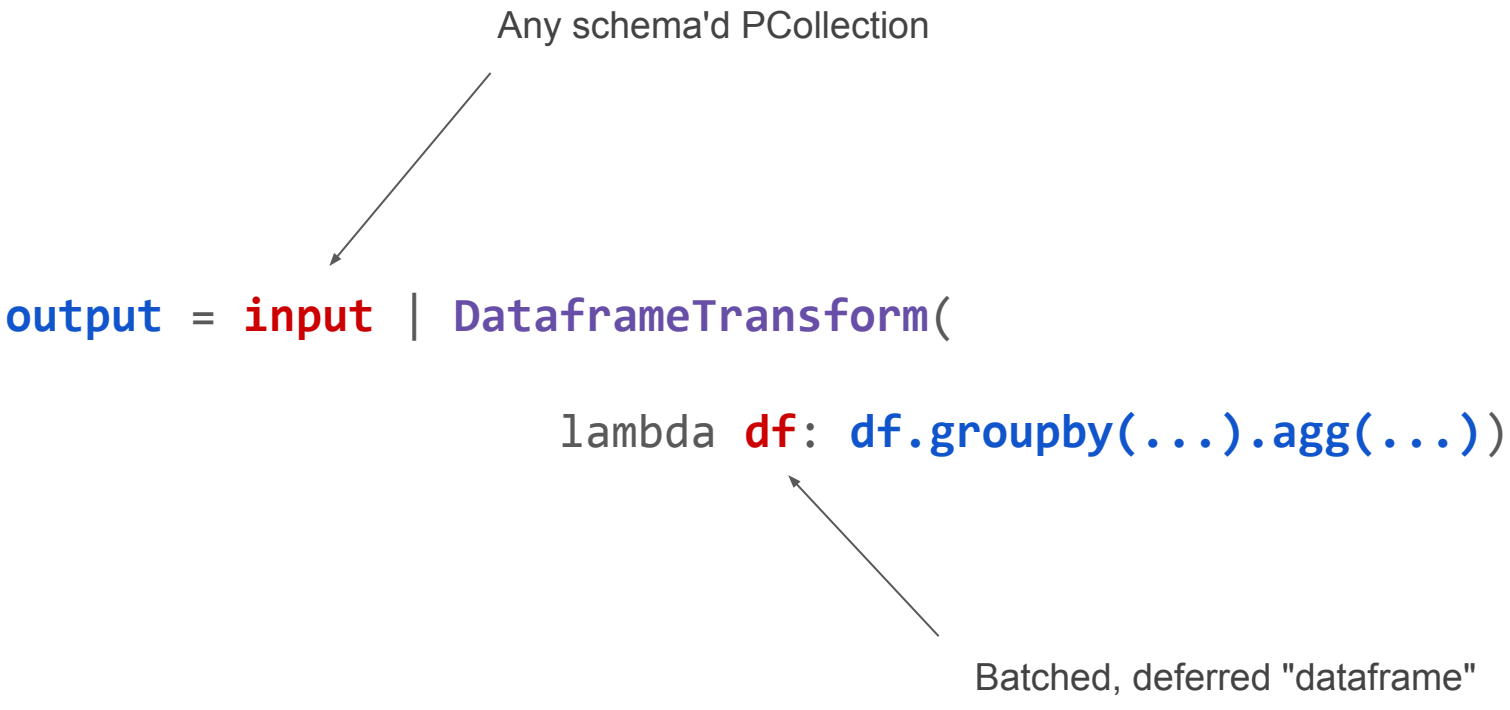
```
)
```

```
pc | GroupBy('recipe', is_berry=lambda x: 'berry' in x.fruit)
        .aggregate_field('quantity', sum, 'total_quantity')
        .aggregate_field(lambda x: x.quantity*x.unit_price,
                         sum, 'total_price')
```

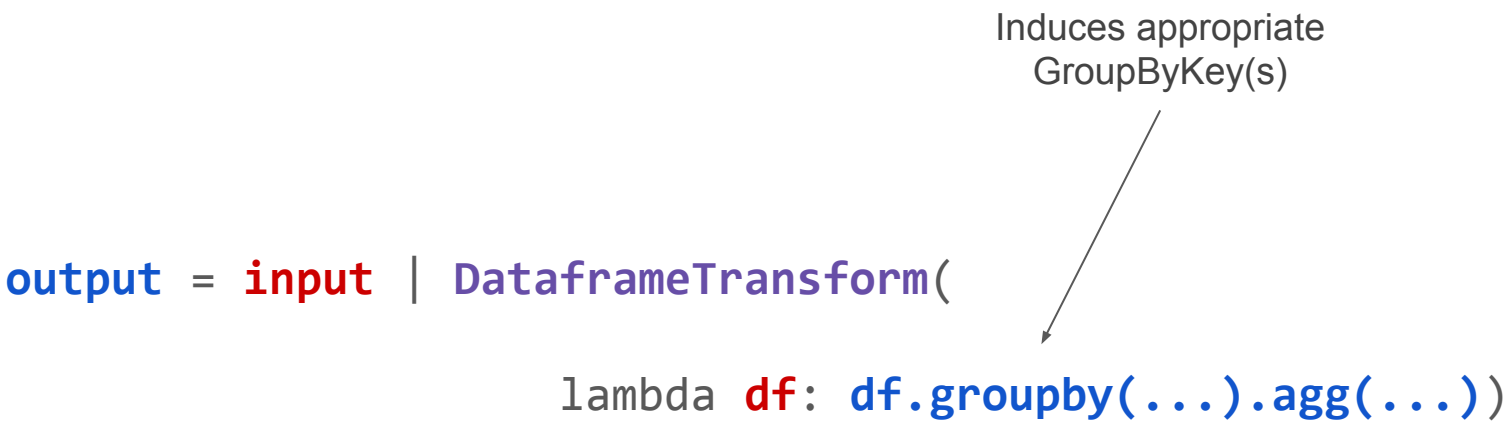
recipe	is_berry	total_quantity	total_price
pie	True	7	$16.00
muffin	False	3	$6.00
muffin	True	2	$4.00

DataframeTransform

Dataframe Transform



Dataframe Transform



Dataframe Transform

```
def my_function(df):
    ...
    return result


output = input | DataframeTransform(my_function)
```

Dataframe Transform

```
def my_function(df):
    df['C'] = df.A + 2*df.B
    result = df.groupby('C').sum().filter('A < 0')
    return result

output = input | DataframeTransform(my_function)
```

Dataframe Transform

Some restrictions apply

- All operations deferred
  - E.g. you can compute a **sum**, but can't ~~**branch**~~ on the result.
- Result columns must be computable
  - E.g. no ~~**transpose**~~
- Non-parallel operations must be guarded
  - `with beam.dataframe.allow_non_parallel_operations(True): ...`

Dataframe Transform

Some restrictions apply

- All operations deferred
  - E.g. you can compute a **sum**, but can't ~~**branch**~~ on the result.
- Result columns must be computable
  - E.g. no ~~**transpose**~~
- Non-parallel operations must be guarded
  - `with beam.dataframe.allow_non_parallel_operations(True): ...`

But, what is implemented is faithful

- Under the hood, actual Pandas dataframe methods are called.

Dataframe Transform

```
output = input | DataframeTransform(lambda df: ...)
```

Dataframe Transform

```
output = (pc1, pc2) | DataframeTransform(lambda df1, df2: ...)
```

Dataframe Transform

```
output = {a: pc, ...} | DataframeTransform(lambda a, ...: ...)
```

Dataframe Transform

```
pc1, pc2 = {a: pc} | DataframeTransform(lambda a: expr1, expr2)
```

Dataframe Transform

```
{...} = {a: pc} | DataframeTransform(lambda a: {...})
```

Dataframe/PCollection Conversion

```
with beam.Pipeline() as p:
  pc1 = ...
  pc2 = ...

  df1 = to_dataframe(pc1)
  df2 = to_dataframe(pc2)
  ...
  result = ...

  result_pc = to_pcollection(result)

  result_pc | beam.WriteToText(...)
```

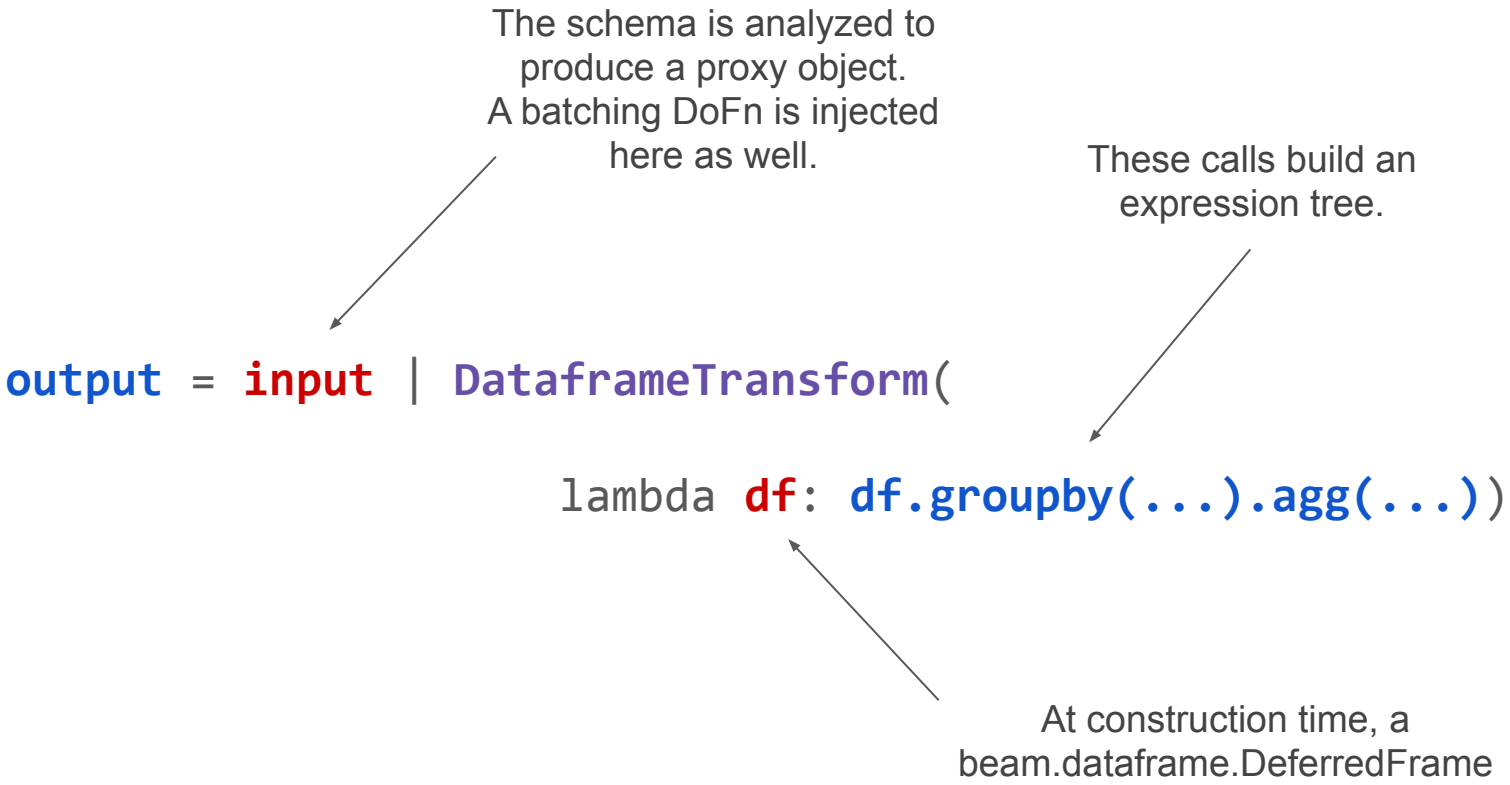
Dataframe Transform - Status

- Currently in Active Development
  - Not ready for production use
  - Targeting 2.25 release
  - Try it out on master
- Basic API and framework in place
- Running against Pandas doctests
  - DataFrame 223 / 112 / 237 Skipped/WontImplement/Passed
  - Series 172 / 87 / 152 Skipped/WontImplement/Passed
  - Also investigating IOs, etc. but nothing implemented yet.

DataframeTransform

A peek under the Hood

Dataframe Transform - Under the Hood



Dataframe Transform - Under the Hood

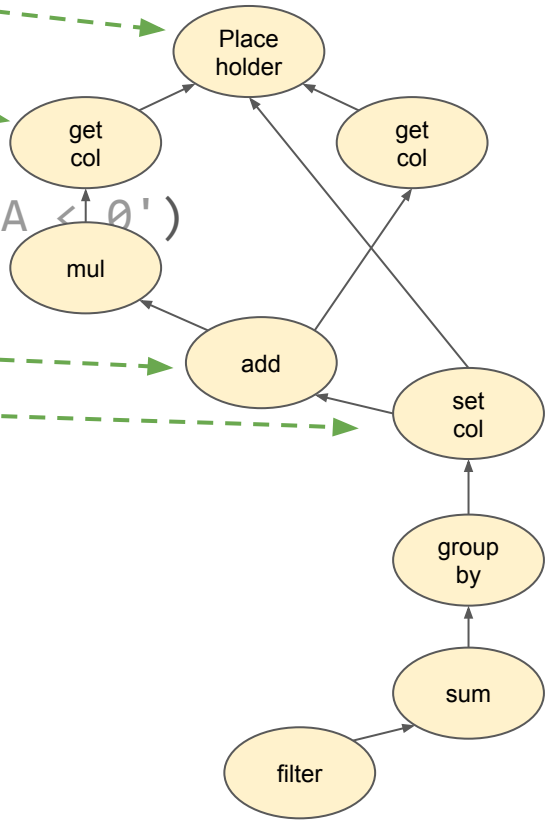
```
def my_function(df):
    df['C'] = df.A + 2*df.B
    result = df.groupby('C').sum().filter('A < 0')
    return result

output = input | DataframeTransform(my_function)
```

Expression Trees recorded, validated, and returned.

Dataframe Transform - Under the Hood

```
def my_function(df):
    df['C'] = df.A + 2*df.B
    result = df.groupby('C').sum().filter('A < 0')
    return result
```

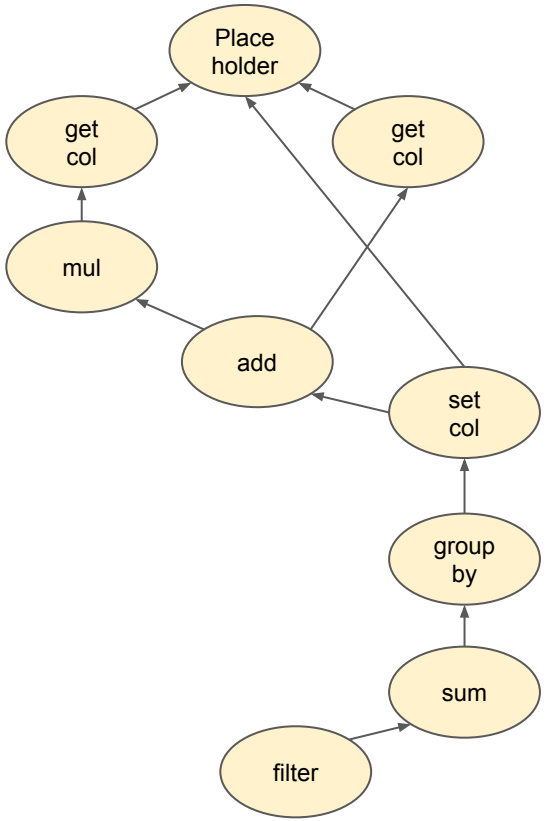


Expression Trees recorded, validated, and returned.

Dataframe Transform - Under the Hood

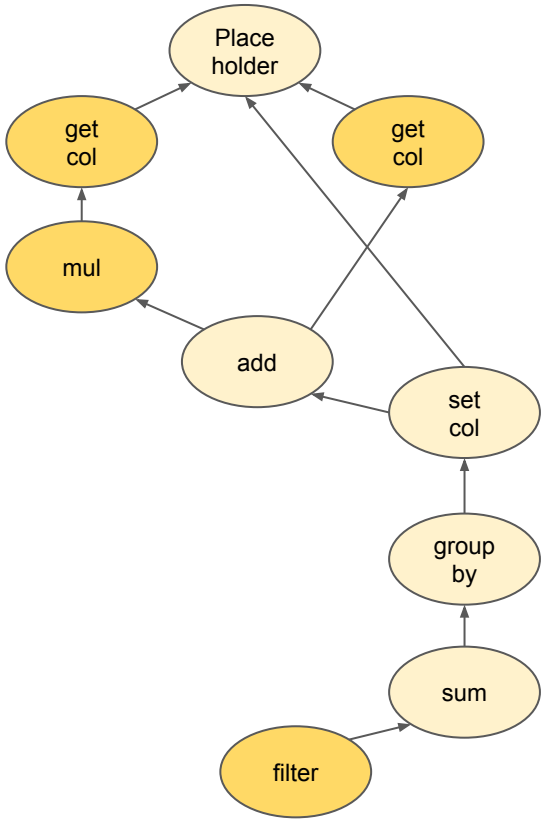
Classification of Operations

- Elementwise
- Grouping
- Zipping
- Order-sensitive



Dataframe Transform - Under the Hood

Elementwise Operations map naturally onto ParDo operations in a distributed system, and can be executed by applying the given operation to each partition.

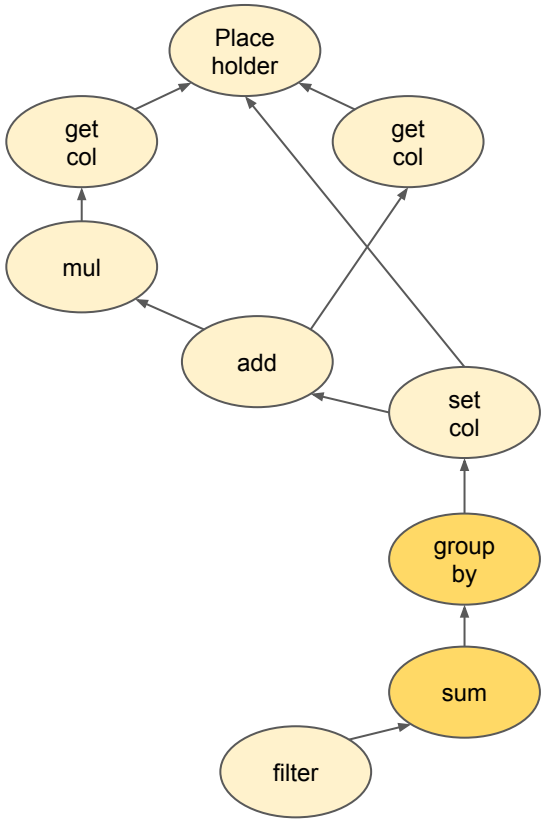


Dataframe Transform - Under the Hood

Grouping Operations collocate rows with identical values in indices/columns, analogous to the GroupByKey and Combine operations in Beam.

The key insight is that one can perform these operations locally if all required rows are in the same partition, so we inject a GroupByKey to colocate all required rows, then apply the pandas grouping operation directly.

Combining operations lifted when possible.

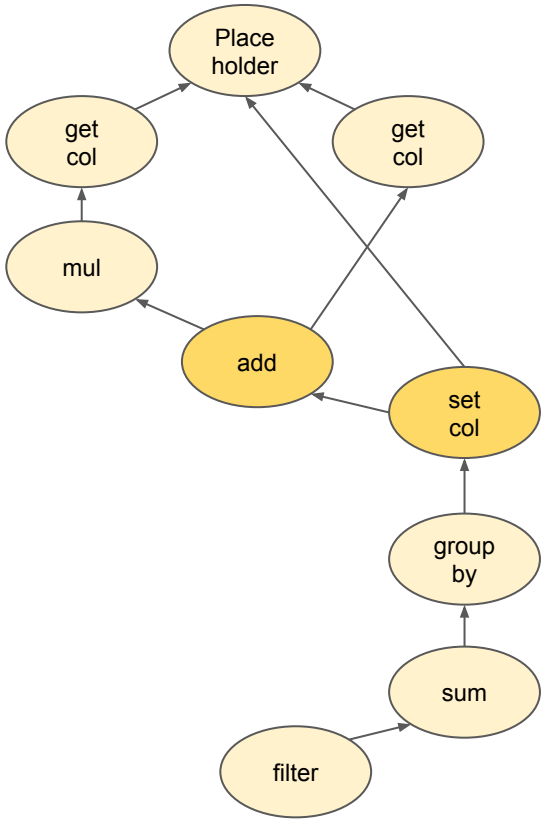


Dataframe Transform - Under the Hood

Zipping Operations take advantage of the fact that *all dataframes are keyed* giving a natural 1:1 relationship between the rows of multiple dataframes.

CoGBK or Join come the closest in Beam.

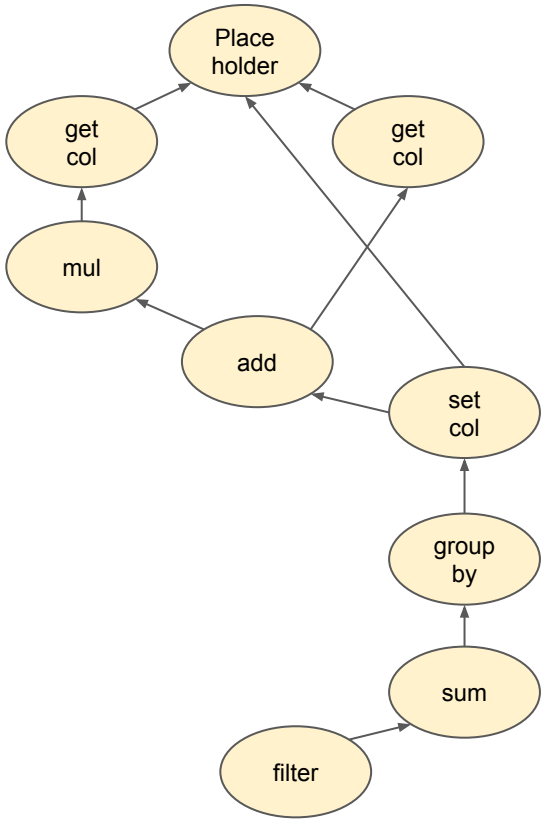
An essential optimization is avoiding shuffles when the inputs are *already* both partitioned by index (e.g. common ancestor).



Dataframe Transform - Under the Hood

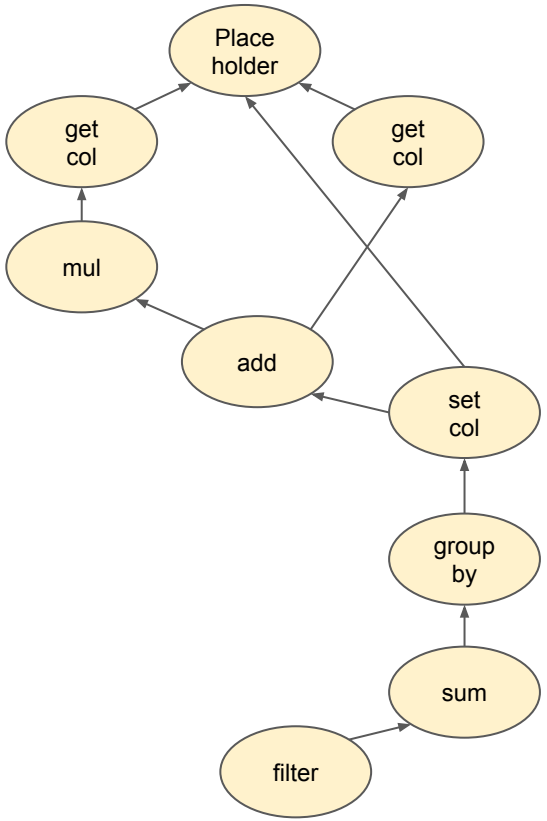
Order-sensitive Operations (e.g. `iloc`) are not (yet?) supported, as PCollections are unordered and we use hash partitioning for good distributions.

We have considered doing this in the future for Dataframes whose order has been explicitly declared (e.g. via a sort). This may have performance implications.

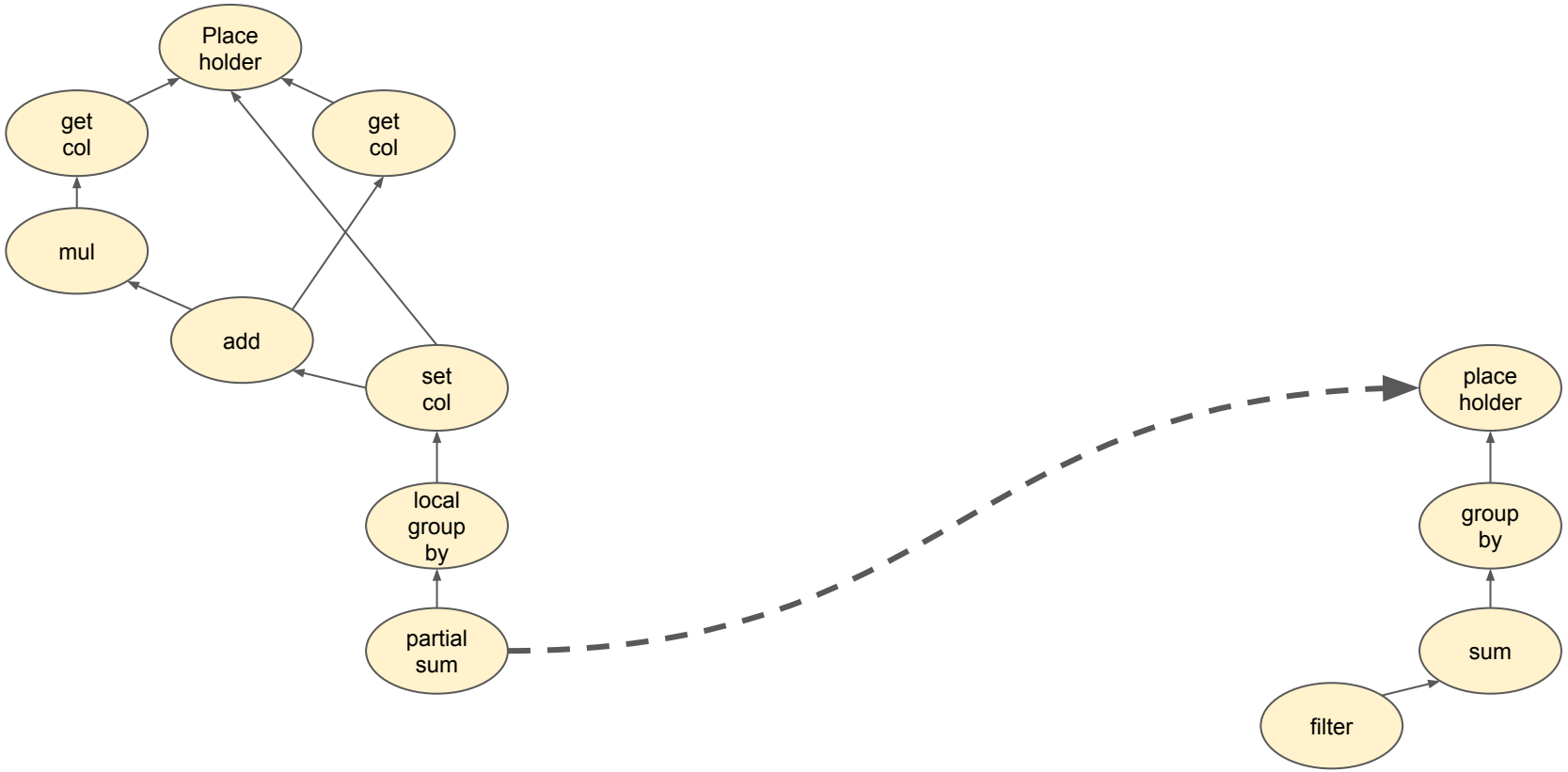


Dataframe Transform - Under the Hood

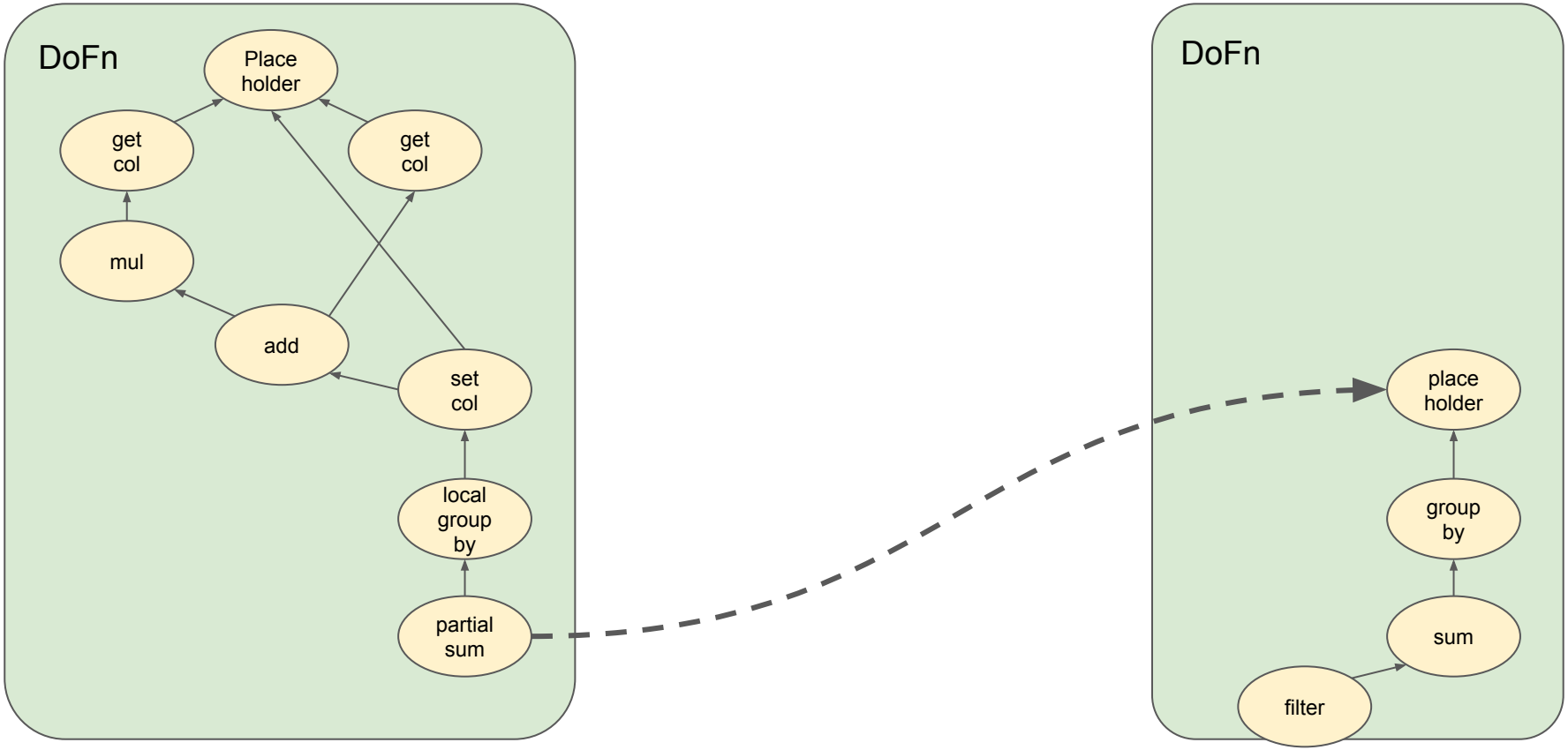
The expression tree is then broken up into the minimal number of **DoFns**, interleaved with **partitioning shuffles**.



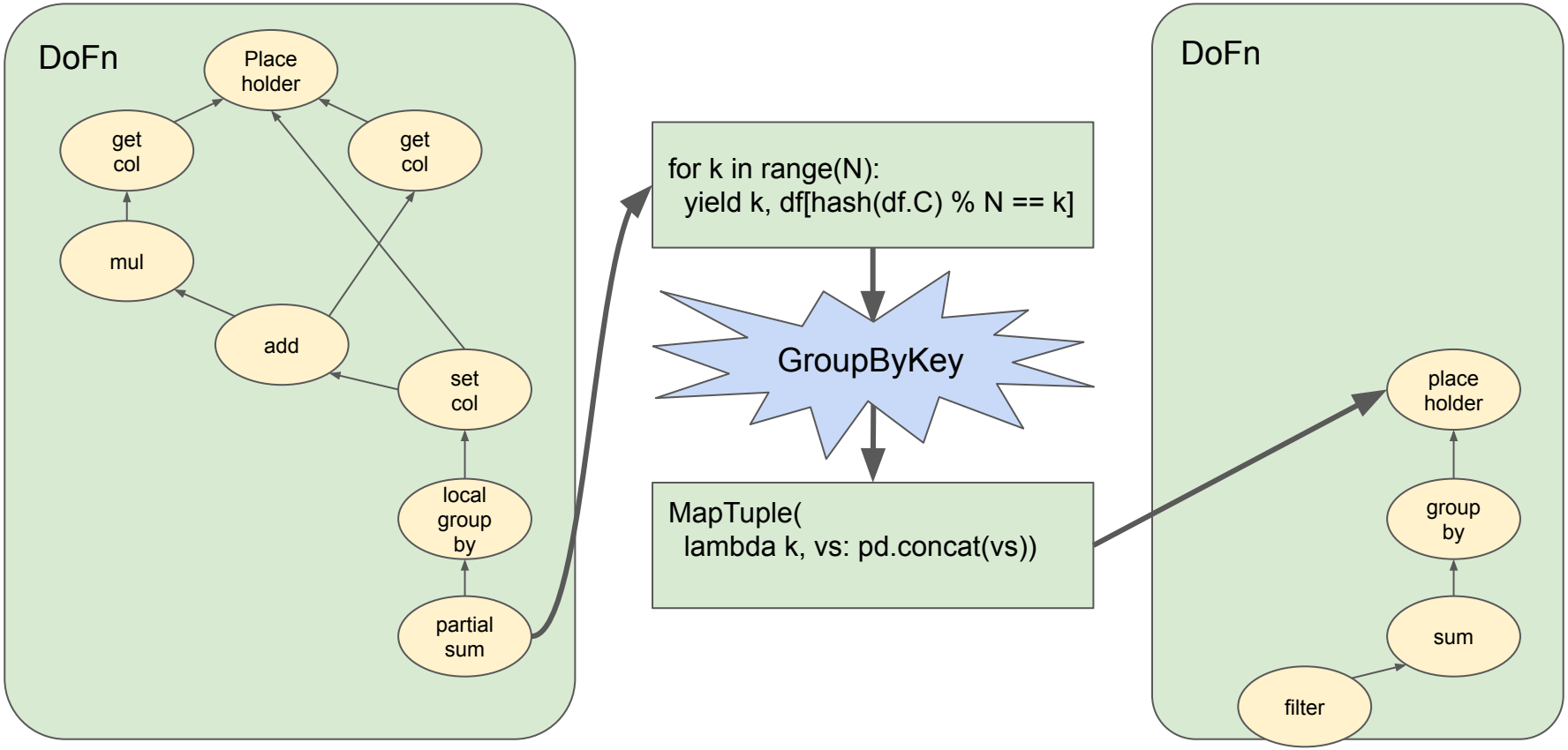
Dataframe Transform - Under the Hood



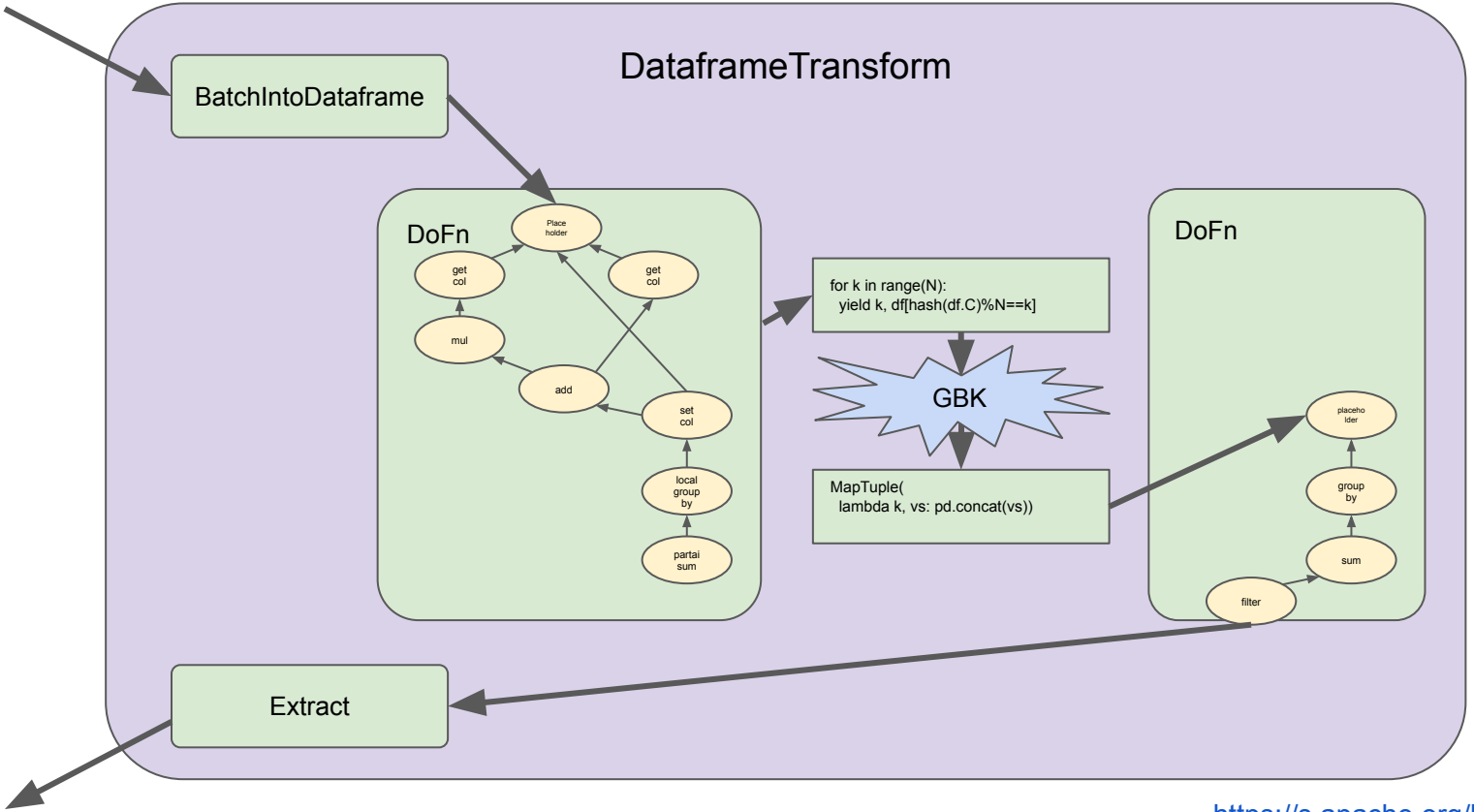
Dataframe Transform - Under the Hood



Dataframe Transform - Under the Hood



Dataframe Transform - Under the Hood



https://s.apache.org/beam-dataframes

SqlTransform

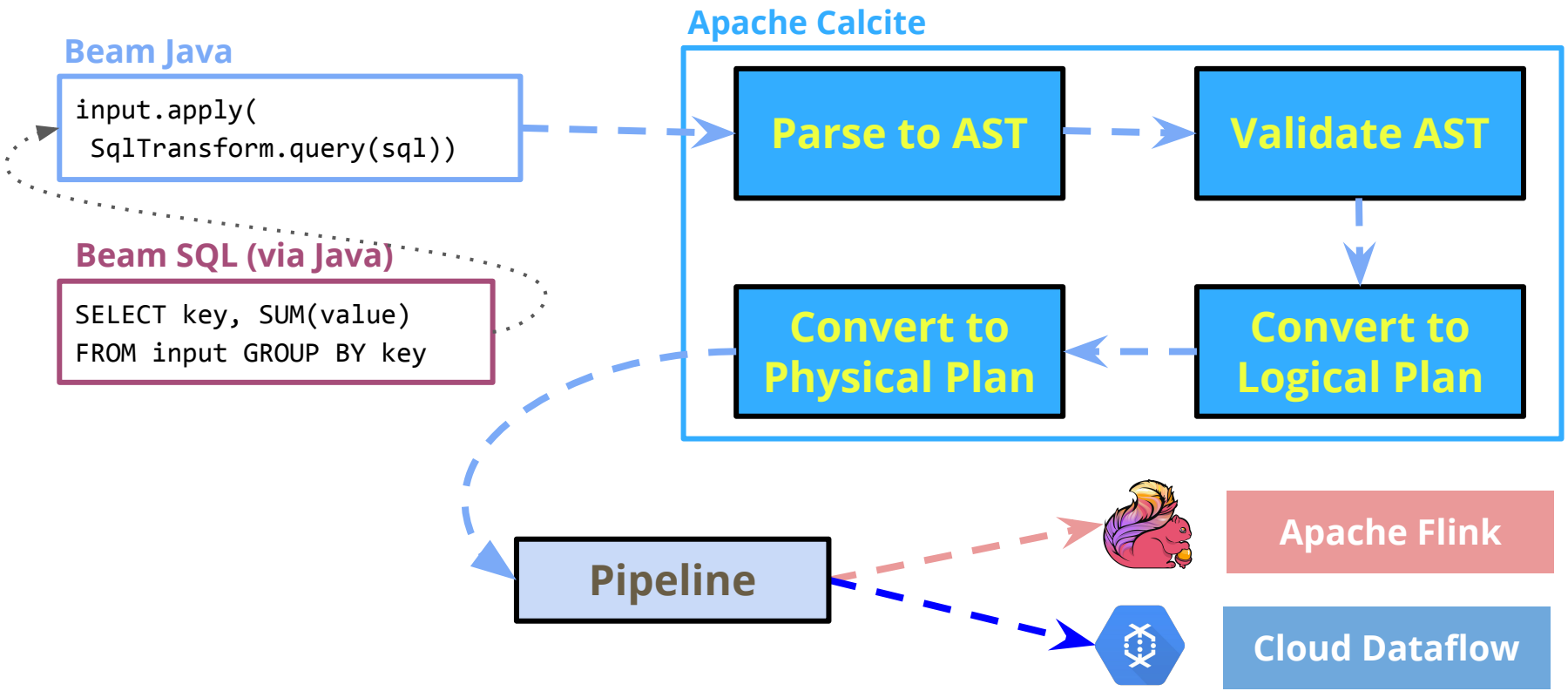
A peek under the Hood

(Slides adapted from s.apache.org/beam-sql-vegas courtesy of Andrew Pilloud)

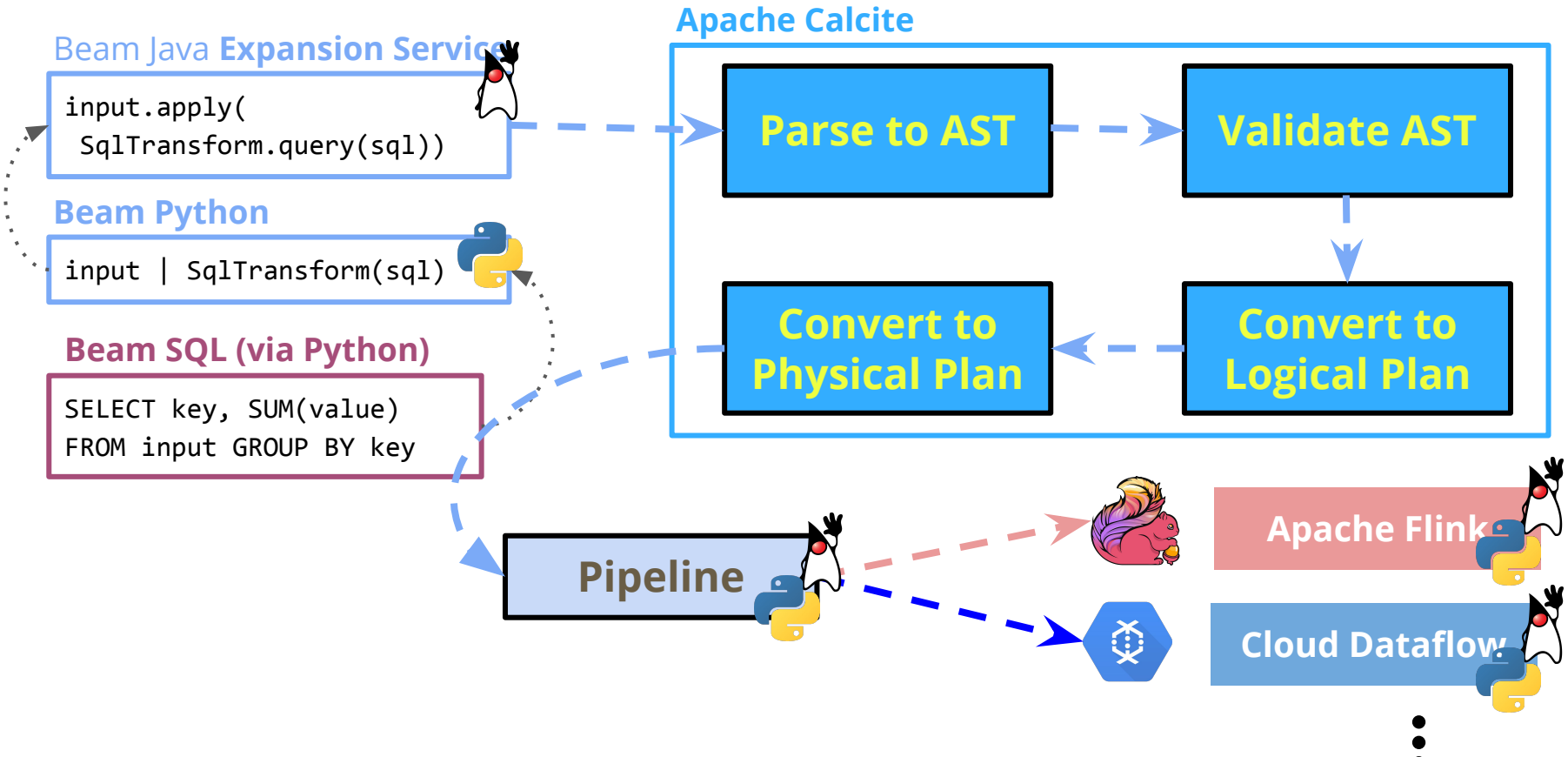
Beam SQL: It’s Apache Calcite, essentially.

- SQL Parsing and Validation*
- Conversion to Relational Algebra*
- Conversion to Physical Execution Plan
- JDBC Driver
- Implementation of Built-in SQL operators
- Project and Filter Code Generation

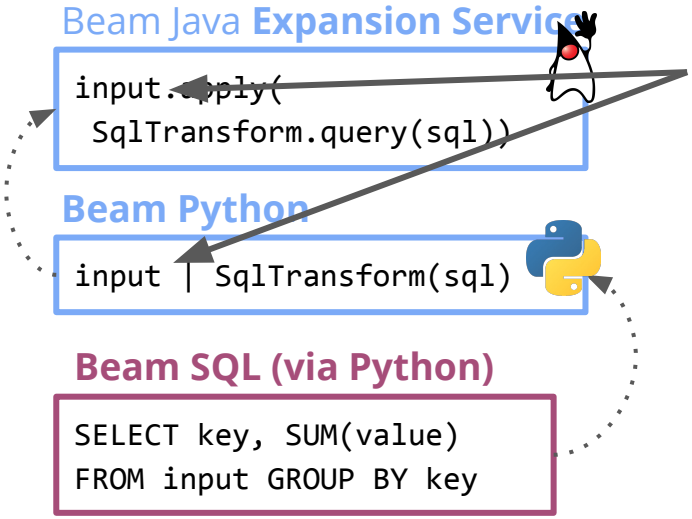
Beam SQL Java



Beam SQL Python

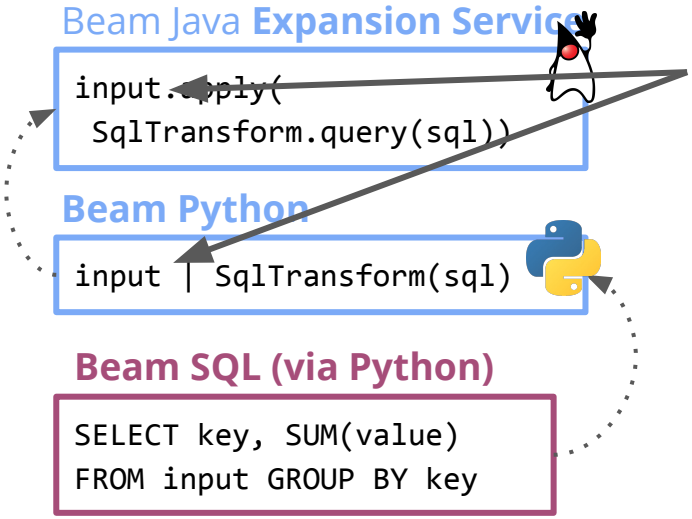


Beam SQL Python



How is `input` PCollection understood by both the Java Expansion Service and the Python SDK?

Beam SQL Python



How is `input` PCollection understood by both the Java Expansion Service and the Python SDK?

- Portable Beam Schemas!
- Defined a new standard coder, `beam:coder:row:v1`, for communicating structured data between SDKs.

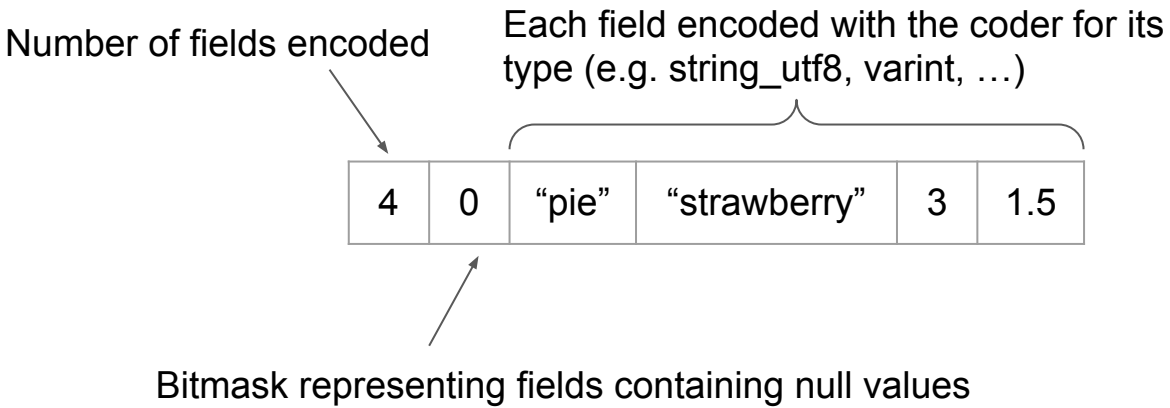
From `class RowCoder` to `beam:coder:row:v1`

- Java RowCoder implementation …
  - … codified as a standard coder with integration tests: beam:coder:row:v1 specification
  - ... implemented in Python: row_coder.py

Construction time

```
Coder {
  urn = “beam:coder:row:v1”;
  payload = Schema {
    fields = [{
      name = “recipe”,
      type = STRING
    },
    ...
    ]
  }
}
```

Execution time



Questions?