Scaling up pandas with the Beam DataFrame API

By Brian Hulette (bhulette@apache.org)

https://s.apache.org/beam-dataframes-2022
Brian Hulette

Software Engineer at Google
Apache Arrow Committer
Apache Beam Committer

bhulette@apache.org
@BrianHulette
linkedin.com/in/brian-hulette
github.com/TheNeuralBit
Agenda

- What are pandas DataFrames? Why put them in Beam?
- Tour of the Beam DataFrame API
- How it works
- Lessons Learned and Future Work
Agenda

- What are pandas DataFrames? Why put them in Beam?
- Tour of the Beam DataFrame API
- How it works
- Lessons Learned and Future Work
In [1]: import pandas as pd

In [2]: df = pd.DataFrame(
    ....:     {
    ....:         "A": ["foo", "bar", "foo", "bar", "foo", "bar", "foo"],
    ....:         "B": ["one", "one", "two", "three", "two", "two", "one", "three"],
    ....:         "C": np.random.randn(8),
    ....:         "D": np.random.randn(8),
    ....:     }
    ....: )

In [3]: df
Out[3]:
   A    B         C         D
0  foo   one  1.346061  -1.577585
1  bar   one  1.511763   0.396823
2  foo   two  1.627081  -0.105381
3  bar  three  0.990582  -0.532532
4  foo  three -0.441652   1.453749
5  bar   two  1.211526   1.208843
6  foo   one  0.268520  -0.080952
7  foo  three  0.024580  -0.264610
In [3]: df
Out[3]:
    A    B       C       D
0  foo  one  1.346061 -1.577585
1  bar  one  1.511763  0.396823
2  foo  two  1.627081 -0.105381
3  bar  three -0.990582 -0.532532
4  foo  two -0.441652  1.453749
5  bar  two  1.211526  1.208843
6  foo  one  0.268520 -0.080952
7  foo  three  0.024580 -0.264610

In [4]: df.groupby("A").sum()
Out[4]:
    C     D
A
bar 1.732707  1.073134
foo 2.824590 -0.574779

In [5]: df.C
Out[5]:
    0    0.359797
    1    0.371583
    2   -1.849233
    3   -1.880074
    4   -0.689943
    5   -1.024726
    6   -1.492432
    7   -0.650677
Name: C, dtype: float64

In [6]: df.C.mean()
Out[6]: -0.8569631996465763
In [3]: df

Out[3]:
   A    B     C     D
0  foo   one  1.3461  -1.5776
1  bar   one  1.5118   0.3968
2  foo   two  1.6271  -0.1054
3  bar  three  0.9906  -0.5325
4  foo   two  0.4417  1.4537
5  bar   two  1.2115  1.2088
6  foo   one  0.2685  -0.0809
7  foo  three  0.0246 -0.2646

In [4]: df.groupby("A").sum()

Out[4]:
   C     D
A
  bar  1.7327  1.0731
  foo  2.8246 -0.5748

In [5]: df.C

Out[5]:
  0    0.35979
  1    0.37158
  2    1.84923
  3    1.88074
  4    0.68994
  5   -1.02472
  6   -1.49243
  7   -0.65068
Name: C, dtype: float64

In [6]: df.C.mean()

Out[6]: -0.856963
In [3]: df[df.B == 'one']
Out[3]:
    A   B         C         D
 0  foo  one  1.346061 -1.577585
 1  bar  one  1.511763  0.396823
 6  foo  one  0.268520 -0.080952

In [4]: df.groupby("A").agg({
    ....:  'C': 'sum',
    ....:  'D': 'mean',
    ....: })
Out[4]:
     C         D
A
bar  1.732707  1.073134
foo  2.824590 -0.574779

In [5]: df.new = df.C.abs() > df.D.abs()

In [6]: df
Out[6]:
    A   B         C         D   new
 0  foo  one  1.346061 -1.577585 False
 1  bar  one  1.511763  0.396823  True
 2  foo  two  1.627081 -0.105381  True
 3  bar  three -0.990582 -0.532532  True
 4  foo  two  0.441652  1.453749 False
 5  bar  two  1.211526  1.208843 True
 6  foo  one  0.268520 -0.080952 True
 7  foo  three  0.024580 -0.264610 False
Used interactively in Notebooks

Dataset: Stanford Open Policing Project (video)

https://openpolicing.stanford.edu/

In [3]: 
# ri stands for Rhode Island 
ri = pd.read_csv('police.csv')

In [4]: 
# what does each row represent? 
ri.head()

Out[4]:

<table>
<thead>
<tr>
<th>stop_date</th>
<th>stop_time</th>
<th>county_name</th>
<th>driver_gender</th>
<th>driver_age</th>
<th>driver_age_raw</th>
<th>violation_raw</th>
<th>violati</th>
</tr>
</thead>
<tbody>
<tr>
<td>2005-01-02 01:55</td>
<td>NaN</td>
<td>N</td>
<td>M</td>
<td>1865.0</td>
<td>20.0</td>
<td>White</td>
<td>Speeding</td>
</tr>
<tr>
<td>2005-01-18 06:15</td>
<td>NaN</td>
<td>N</td>
<td>M</td>
<td>1805.0</td>
<td>40.0</td>
<td>White</td>
<td>Speeding</td>
</tr>
<tr>
<td>2005-01-23 23:15</td>
<td>NaN</td>
<td>N</td>
<td>M</td>
<td>1972.0</td>
<td>33.0</td>
<td>White</td>
<td>Speeding</td>
</tr>
<tr>
<td>2005-02-20 17:15</td>
<td>NaN</td>
<td>N</td>
<td>M</td>
<td>1956.0</td>
<td>19.0</td>
<td>White</td>
<td>Dui for Service</td>
</tr>
<tr>
<td>2005-03-14 10:00</td>
<td>NaN</td>
<td>F</td>
<td>M</td>
<td>1964.0</td>
<td>21.0</td>
<td>White</td>
<td>Speeding</td>
</tr>
</tbody>
</table>

In [5]: 
# what do these numbers mean?

2. Do men or women speed more often? (video)

In [13]: 
# when someone is stopped for speeding, how often is it a man or woman? 
ri[ri.violation == 'Speeding'].driver_gender.value_counts(normalize=True)

Out[13]:

- M: 0.680527
- F: 0.319473

Name: driver_gender, dtype: float64

In [14]: 
# alternative 
ri.loc[ri.violation == 'Speeding', 'driver_gender'].value_counts(normalize=True)

Out[14]:

- M: 0.680527
- F: 0.319473

Name: driver_gender, dtype: float64

In [15]: 
# when a man is pulled over, how often is it for speeding? 
ri[ri.driver_gender == 'M'].violation.value_counts(normalize=True)

Out[15]:

- Speeding: 0.524356
- Moving violation: 0.297012
- Equipment: 0.125971
- Other: 0.057688
- Registration/plates: 0.038461
- Seat belt: 0.038339

Name: violation, dtype: float64

Why make a pandas compatible API?

- Efficient implementation
- Declarative, concise API
- Familiar API for Python users
1. Efficient Implementation

- Columnar memory layout
- Implemented in C
- Can be re-used to compute partial results on workers
2. pandas has a declarative, concise API

```python
import pandas as pd
df = pd.read_csv(input)
agg = df.groupby('DOLocationID').passenger_count.sum()
agg.to_csv(output)
```

```python
import apache_beam as beam
(p | beam.io.ReadFromText(input, skip_header_lines=1)
  | beam.Map(lambda line: line.split(','))
  # Parse CSV, create key-value pairs
  | beam.Map(lambda splits: (int(splits[8] or 0),  # DOLocationID
                            int(splits[3] or 0)))  # passenger_count
  # Sum values per key
  | beam.CombinePerKey(sum)
  | beam.MapTuple(lambda loc_id, pc: f'{loc_id},{pc}')
  | beam.io.WriteToText(output))
```
3. pandas has a Familiar API

Among 26.9M OSS .py files from GitHub...

- import apache_beam 14.6k
- import pandas 172k (~12x apache_beam)
3. **pandas has a *Familiar* API**

Among 253k OSS .ipynb files from GitHub...

- `import apache_beam` 306
- `import pandas` 53k *(173x apache_beam, 21% of corpus)*

query
3. pandas has a **Familiar API**

Among 253k OSS .ipynb files from GitHub...

- `import apache_beam` 306
- `import pandas` 53k (173x `apache_beam`, 21% of corpus)
- “SELECT ...” 8.48k
- `import numpy` 117k
- `import matplotlib` 89k

`query`
3. pandas has a **Familiar** API

... but it’s **in-memory only**.
Agenda

● What are pandas DataFrames? Why put them in Beam?
● Tour of the Beam DataFrame API
● How it works
● Lessons Learned and Future Work
DataframeTransform

output = input | DataframeTransform(
    lambda df: df.groupby(...).agg(...))

Any schema'd PCollection

Outputs a schema'd PCollection

A “deferred” DataFrame
Multiple Inputs

\[
\text{output} = (\text{pc1}, \text{pc2}) \mid \text{DataframeTransform}(\lambda \text{df1, df2}: \ldots)
\]

\[
\text{output} = \{\text{a: pc, \ldots}\} \mid \text{DataframeTransform}(\lambda \text{a, \ldots}: \ldots)
\]
DataFrame-PCollection Conversion

with beam.Pipeline() as p:
    pc = ... # A PCollection with a Schema

    df = to_dataframe(pc) # A Beam DeferredDataFrame
    result = df.groupby('foo').agg(...)
    result_pc = to_pcollection(result)

    result_pc | beam.WriteToText(...)

Austin, 2022
pandas IOs

from apache_beam.dataframe.io import read_parquet

with beam.Pipeline() as p:
    df = p | read_parquet("gs://bucket/*.pq")
    result = df.groupby('foo').agg(...)
    result.to_csv("gs://bucket/output.csv")

Implementations use FileIO under the hood. Gives us distributed reads, liquid sharding, support for cloud object stores (gs://, s3://).
Agenda

- What are pandas DataFrames? Why put them in Beam?
- Tour of the Beam DataFrame API
- How it works
- Lessons Learned and Future Work
All pandas objects have indexes

In [3]: df
Out[3]:
        A    B         C         D
0  foo    one  1.346061 -1.577585
1  bar    one  1.511763  0.396823
2  foo    two  1.627081 -0.105381
3  bar   three -0.990582 -0.532532
4  foo    two -0.441652  1.453749
5  bar  three -1.211526  1.208843
6  foo    one  0.268520 -0.080952
7  foo  three  0.024580 -0.264610

In [4]: df.groupby("A").sum()
Out[4]:
        C         D
A
bar  1.732707  1.073134
foo  2.824590 -0.574779

In [5]: df.C
Out[5]:
0   0.359797
1   0.371583
2  -1.849233
3  -1.880074
4  -0.689943
5  -1.024726
6  -1.492432
7  -0.650677
Name: C, dtype: float64

In [6]: df.C.mean()
Out[6]: -0.8569631996465763
All pandas objects have indexes

In [3]: df
Out[3]:
      A    B     C     D
0   foo   one  1.346061 -1.577585
1  bar    one  1.511763  0.396823
2  foo    two  1.627081 -0.105381
3  bar   three -0.990582 -0.532532
4  foo    two -0.441652  1.453749
5  bar    two  1.211526  1.208843
6  foo    one  0.268520 -0.080952
7  foo  three  0.024580 -0.264610

In [4]: df.groupby("A").sum()
Out[4]:
      C     D
A
bar  1.732707  1.073134
foo  2.824590 -0.574779

In [5]: df.C
Out[5]:
0   0.359797
1   0.371583
2  -1.849233
3  -1.880074
4  -0.689943
5  -1.024726
6  -1.492432
7  -0.650677
Name: C, dtype: float64

In [6]: df.C.mean()
Out[6]: -0.8569631996465763
Many operations use the index implicitly

In [6]: a
Out[6]:
0   1
1   2
2   3
dtype: int64

In [7]: b
Out[7]:
0   0
1   0
2   1
dtype: int64

In [8]: a*b
Out[8]:
0   0
1   0
2   3
dtype: int64

In [9]: c
Out[9]:
2   0
1   0
0   1
dtype: int64

In [10]: a*c
Out[10]:
0   1
1   0
2   0
dtype: int64

Not order sensitive! Implicitly joined on the index.
How to make a Pipeline Graph?

```python
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)
```

**Objective:** Create a Beam Pipeline sub-graph that performs the computation described by `my_function`. 
How to make a Pipeline Graph?

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

Call function with our own `DeferredDataFrame`, which has custom implementations for pandas operations.
Build an expression tree

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

DeferredDataFrame operations record and validate an Expression Tree. Note this is not a Beam pipeline graph.
Goal: A Beam Pipeline Graph

= PCollection[pd.DataFrame]

DoFn

DoFn

DoFn

GBK

GBK

...
name = "add"
args = [mul, get_col_a]
fn = lambda lhs, rhs: lhs + rhs
proxy = Series([], dtype: float64)
requires_partition_by = Index()
preserves_partition_by = Index()
At execution time, `fn` is called with `pd.Series` or `pd.DataFrame` representing a partition of the full dataset.
Expression Metadata - proxy

An empty `pd.Series` or `pd.DataFrame` with the same shape that we expect to see at execution time.

Used for:
- Validation
- Authentic error messages
- Data types, mapping to Beam Schemas

```python
name = "add"
args = [mul, get_col_a]
fn = lambda lhs, rhs: lhs + rhs
proxy = Series([], dtype=float64)
requires_partition_by = Index()
preserves_partition_by = Index()
```
Expression Metadata - partitioning

Type of partitioning this expression requires in its inputs to be computed correctly.

Type of partitioning guaranteed to be preserved in the expression’s outputs.

```python
name = "add"
args = [mul, get_col_a]
fn = lambda lhs, rhs: lhs + rhs
proxy = Series([], dtype: float64)
requires_partition_by = Index()
preserves_partition_by = Index()
```
Partitioning Requirements

- **Index()**
  - Shuffle using a hash of the index modulo N to co-locate like indexes into N partitions.

- **Singleton()**
  - Collect all data onto a single node.
  - Some operations require it.
  - Used internally if we know data volume is small.

- **Arbitrary()**
  - No partitioning guarantees whatsoever.
Partitioning Requirements

- **Index()**
  - Shuffle using a hash of the index modulo N to co-locate like indexes into N partitions.

- **Singleton()**
  - Collect all data onto a single node.
  - Some operations require it.
  - Used internally if we know data volume is small.

- **Arbitrary()**
  - No partitioning guarantees whatsoever.

```python
for k in range(N):
    yield k, df[hash(df.index) % N == k]
```

**MapTuple(**
```
lambda k, vs: pd.concat(vs))
```

**GroupByKey**
Partitioning Requirements

● **Index()**
  ○ Shuffle using a hash of the index modulo N to co-locate like indexes into N partitions.

● **Singleton()**
  ○ Collect all data onto a single node.
  ○ Some operations require it.
  ○ Used internally if we know data volume is small.

● **Arbitrary()**
  ○ No partitioning guarantees whatsoever.

```
yield None, df
```

```
GroupByKey
```

```
MapTuple(
    lambda k, vs: pd.concat(vs))
```
Partitioning Requirements

- **Index()**
  - Shuffle using a hash of the index modulo N to co-locate like indexes into N partitions.

- **Singleton()**
  - Collect all data onto a single node.
  - Some operations require it.
  - Used internally if we know data volume is small.

- **Arbitrary()**
  - No partitioning guarantees whatsoever.
Expression Tree to Pipeline Graph

The expression tree is broken up into a minimal number of stages, i.e. DoFns interleaved with partitioning shuffles.

We determine where to shuffle based on the nodes' partitioning requirements.
Expression Tree to Pipeline Graph

- Place holder
- get col B
- mul
- add
- get col A
- set col
- local group by
- partial sum
- place holder
- group by
- sum
- filter
Expression Tree to Pipeline Graph

DoFn

- Place holder
  - get col B
  - get col A
    - mul
      - add
        - set col
          - local group by
            - partial sum
              - local group by
                - partial sum

DoFn

- Place holder
  - group by
    - sum
      - filter
Expression Tree to Pipeline Graph

DoFn

Place holder

get col B

get col A

mul

add

set col

local group by

partial sum

for k in range(N):
    yield k, df[hash(df.index) % N == k]

GroupByKey

MapTuple(
    lambda k, vs: pd.concat(vs))

DoFn

place holder

group by

sum

filter
Batching and Unbatching

BatchIntoDataframe → DataframeTransform → DoFn → GBK → DoFn → Extract

for k in range(N):
    yield k, df[hash(df.C)%N==k]

MapTuple(lambda k, vs: pd.concat(vs))

Austin, 2022
Agenda

- What are pandas DataFrames? Why put them in Beam?
- Tour of the Beam DataFrame API
- How it works
- Lessons Learned and Future Work
Python needs more schema’d sources!

```python
# From apache_beam.examples.dataframe.flight_delays
p | 'read table' >> beam.io.ReadFromBigQuery(query="SELECT ...")
# Use beam.Select to make sure data has a schema
# The casts in lambdas ensure data types are properly inferred
| 'set schema' >> beam.Select(
    date=lambda x: str(x['date']),
    airline=lambda x: str(x['airline']),
    departure_airport=lambda x: str(x['departure_airport']),
    arrival_airport=lambda x: str(x['arrival_airport']),
    departure_delay=lambda x: float(x['departure_delay']),
    arrival_delay=lambda x: float(x['arrival_delay']))

beam_df = p | read_gbq("SELECT ...")
```

Follow #20810
Compliance is key

● Too many operations raise WontImplementError
● We need to close the compliance gap
● [s.apache.org/interactive-dataframe-operations](https://s.apache.org/interactive-dataframe-operations) (#21638)
  ○ Add a set of `df.interactive.*` operations that are eagerly executed.
  ○ *e.g.* `df.interactive.plot`, `df.interactive.pivot`
● [s.apache.org/order-sensitive-dataframe-operations](https://s.apache.org/order-sensitive-dataframe-operations) (#20862)
  ○ ~14% of pandas operations are order-sensitive.
  ○ Proposal to support these operations, with caveats.
  ○ *e.g.* `df.sort_values().fillna()`
Streaming can be a differentiator

# Read an unbounded source into a Beam DataFrame
beam_df = p | read_kafka(topic)

# Create a 5 minute window, perform an aggregation
beam_df.rolling('5m')['column_a'].mean()

# Write the result
beam_df.to_csv()
How you can help

- **Try** your use case in the Beam DataFrame API  
  - Let us know if doesn’t work! [File an issue](#) with label [dataframe](#).
- **Contribute** tests for operations/use-cases you care about  
  - [apache_beam.dataframe.frames_test](#)
  - `self._run_test(lambda df: df.groupby('foo').sum(), df)`
- **Add** schema support to IOs
- **Add** interactive and/or order-sensitive operations
Questions?

https://s.apache.org/beam-dataframes-2022

bhulette@apache.org
@BrianHulette
linkedin.com/in/brian-hulette
github.com/TheNeuralBit

Austin, 2022
Backup/Graveyard
Dataframe Transform - Under the Hood

Classification of Operations

- Elementwise
- Grouping
- Zipping
- Order-sensitive
Elementwise Operations

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

**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 collocate all required rows, then apply the pandas grouping operation directly.

Combining operations lifted when possible.
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.