PyArrow Demo

August 17, 2021

1 A short demo of PyArrow & Neo4j

1.1 Our Dependencies

Nothing special really…the usual cast of characters with the addition of `pyarrow`

[1]:
```
%pip install pyarrow pandas scikit-learn matplotlib seaborn
%matplotlib inline
```

```
Requirement already satisfied: pyarrow in ./venv/lib/python3.9/site-packages
(5.0.0)
Requirement already satisfied: pandas in ./venv/lib/python3.9/site-packages
(1.3.2)
Requirement already satisfied: scikit-learn in ./venv/lib/python3.9/site-
packages (0.24.2)
Requirement already satisfied: matplotlib in ./venv/lib/python3.9/site-packages
(3.4.3)
Requirement already satisfied: seaborn in ./venv/lib/python3.9/site-packages
(0.11.2)
Requirement already satisfied: numpy>=1.16.6 in ./venv/lib/python3.9/site-
packages (from pyarrow) (1.21.2)
Requirement already satisfied: python-dateutil>=2.7.3 in
./venv/lib/python3.9/site-packages (from pandas) (2.8.2)
Requirement already satisfied: pytz>=2017.3 in ./venv/lib/python3.9/site-
packages (from pandas) (2021.1)
Requirement already satisfied: scipy>=0.19.1 in ./venv/lib/python3.9/site-
packages (from scikit-learn) (1.7.1)
Requirement already satisfied: threadpoolctl>=2.0.0 in
./venv/lib/python3.9/site-packages (from scikit-learn) (2.2.0)
Requirement already satisfied: joblib>=0.11 in ./venv/lib/python3.9/site-
packages (from scikit-learn) (1.0.1)
Requirement already satisfied: pyparsing>=2.2.1 in ./venv/lib/python3.9/site-
packages (from matplotlib) (2.4.7)
Requirement already satisfied: pillow>=6.2.0 in ./venv/lib/python3.9/site-
packages (from matplotlib) (8.3.1)
Requirement already satisfied: kiwisolver>=1.0.1 in ./venv/lib/python3.9/site-
packages (from matplotlib) (1.3.1)
Requirement already satisfied: cycler>=0.10 in ./venv/lib/python3.9/site-
packages (from matplotlib) (0.10.0)
```

```
Requirement already satisfied: six in ./venv/lib/python3.9/site-packages (from
cycler>=0.10->matplotlib) (1.16.0)
Note: you may need to restart the kernel to use updated packages.
```

1.1.1 Imports and our Integration

We'll set up our imports next.

One special import is neo4j_arrow, the client wrapper to simplify talking to the server-side `Neo4j-Arrow` service. It's like 100 lines of Python and uses the `PyArrow` framework…no Neo4j code!

> Server-side it exists as a database plugin. If you have access, the code is here: https://github.com/neo4j-field/neo4j-arrow

[2]:
```python
# Get our DS imports ready!
from sklearn.decomposition import PCA
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import seaborn as sns

# To time stuff...
import time

# And our neo4j integration!
import neo4j_arrow
```

1.2 Connecting our Neo4jArrow Client

Very simple. We provide access credentials (username, password) and then provide the location of the server in a tuple of `(host, port)`.

`Neo4jArrow` uses Neo4j's built-in authorization framework. All our calls to the server are authenticated like any other Neo4j client.

[3]:
```python
client = neo4j_arrow.Neo4jArrow('neo4j', 'password', ('voutila-arrow-test',␣
↪9999))
```

1.2.1 Discovering Available Actions

Arrow Flight uses an RPC concept. In short, clients can perform *Actions* sending optional payload data to the server with each action. Clients can also consume or put *streams* to/from the server.

Let's discover our available actions!

[4]:
```python
actions = client.list_actions()

for action in actions:
```

```
print(action)
```

```
ActionType(type='cypherRead', description='Submit a new Cypher-based read job')
ActionType(type='cypherWrite', description='Submit a new Cypher-based write
job')
ActionType(type='jobStatus', description='Check the status of a Job')
ActionType(type='gdsNodeProperties', description='Stream node properties from a
GDS Graph')
ActionType(type='gdsRelProperties', description='Stream relationship properties
from a GDS Graph')
```

Each of these actions can be called by an Apache Arrow client, regardless if it's PyArrow or the Arrow R package or Arrow for Rust!

1.3 Working with Cypher Jobs

The way I've architected `Neo4jArrow` is designed around submitting "jobs" that construct streams. Let's submit some Cypher!

[5]:
```
cypher = """
  UNWIND range(1, $i) AS n
  RETURN n, [_ IN range(1, $j) | rand()] AS embedding
"""
params = {
    "i": 1_000_000,
    "j": 128
}

print(f"Submitting cypher with params:\n{cypher}\n{params}")

ticket = client.cypher(cypher, params=params)
print(ticket)
```

```
Submitting cypher with params:

  UNWIND range(1, $i) AS n
  RETURN n, [_ IN range(1, $j) | rand()] AS embedding

{'i': 1000000, 'j': 128}
<Ticket b'c89a704a-58eb-4c16-b002-ee86cdfda4d2'>
```

1.3.1 Waiting for our Results

Each job results in a *ticket*. Clients use the ticket to check on job status or request a stream of the rsults.

Let's wait until our Cypher is producing results and our stream is ready for consumption. This little helper function just polls the `jobStatus` *Action* waiting for our job to be in a "producing" state.

[6]:
```python
print(f'Polling for status on ticket {ticket}...')
ready = client.wait_for_job(ticket, timeout=5)
if not ready:
    raise Exception('something is wrong...did you submit a job?')
else:
    print('...Stream is Ready!')
```

```
Polling for status on ticket <Ticket b'c89a704a-58eb-4c16-b002-ee86cdfda4d2'>…
…Stream is Ready!
```

1.3.2 Consuming our Results

Clients consume streams by presenting their ticket. They bet back a PyArrow stream reader and have some options to how they consume the stream:

1. They can iterate over batches in the stream and process them incrementally.
2. They can consume the entire stream into a PyArrow Table
3. They can consume the entire stream immediately into a Pandas data frame

[7]:
```python
# Let's get a dataframe!
print('>> Reading the result of our Cypher job into a dataframe. Please wait...
 ↪')

start = time.time()
table = client.stream(ticket).read_all()
delta = round(time.time() - start, 1)

print(f'>> Read our stream entirely into a PyArrow table in {delta} seconds!')
print(table)
megs = table['embedding'].to_pandas().memory_usage() / (1024 * 1024)
print(f"How big is our data? It's about {round(megs, 2):,} MiB.")
```

```
>> Reading the result of our Cypher job into a dataframe. Please wait…
>> Read our stream entirely into a PyArrow table in 11.7 seconds!
pyarrow.Table
embedding: list<embedding: double>
  child 0, embedding: double
n: int64
How big is our data? It's about 7.63 MiB.
```

Let's Work with Pandas and Scikit-Learn!

[8]:
```python
# We'll convert our series of arrays into a NumPy matrix
df = table.select(['embedding'])[0].to_pandas()
m = np.matrix(df.tolist())

# Then let's do dimensional reduction so we can plot our vectors
pca = PCA(n_components=2)
```

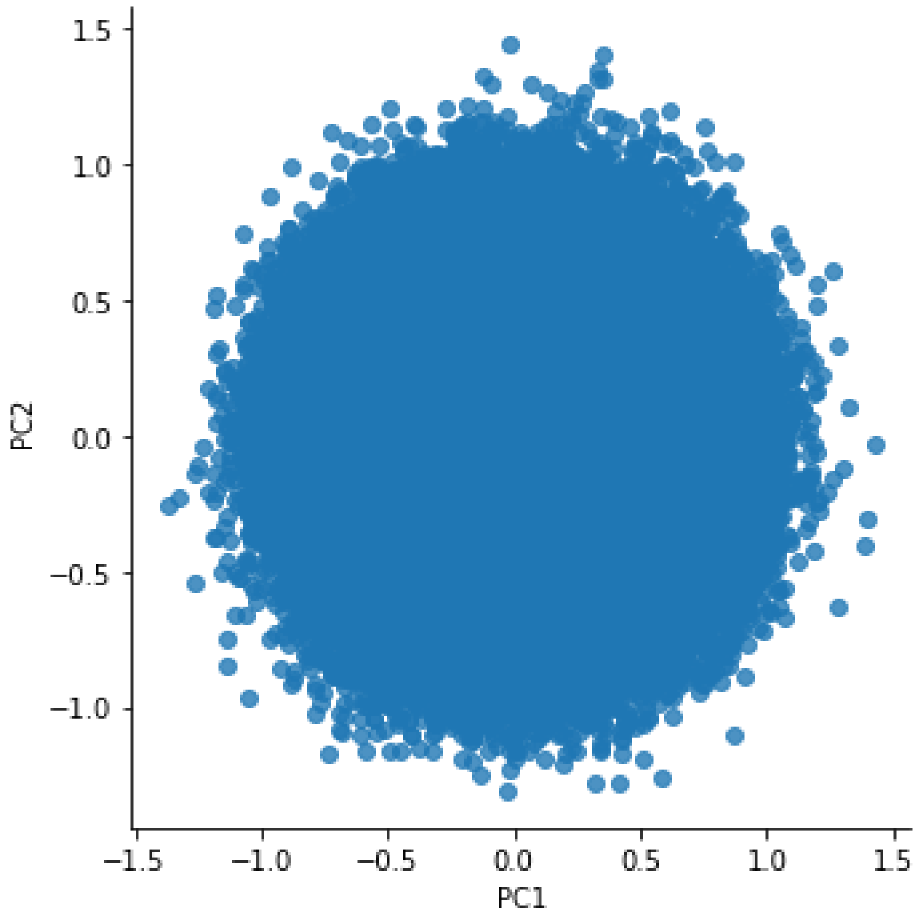
```
pc = pca.fit_transform(m)

print('Before, our data looked like:')
print(table)

print('Now we have a matrix like:')
print(pc)
```

```
Before, our data looked like:
pyarrow.Table
embedding: list<embedding: double>
  child 0, embedding: double
n: int64
Now we have a matrix like:
[[ 0.08728453  0.1223428 ]
 [ 0.04351369 -0.05593023]
 [-0.04542323 -0.14810567]
 …
 [-0.298385    0.18681565]
 [ 0.15782937 -0.12891416]
 [ 0.1579672  -0.15567771]]
```

Let's plot!

[9]:
```
pc_df = pd.DataFrame(data=pc, columns=['PC1', 'PC2'])
sns.lmplot( x="PC1", y="PC2",
  data=pc_df,
  fit_reg=False)
```

[9]:
```
<seaborn.axisgrid.FacetGrid at 0x7fc44cfe6550>
```



1.4 Now for the fun stuff: Direct GDS Integration!

We just played with Cypher, which is fine…but what about working with *even more data* from things like GDS?

Our traditional methods using the Python driver would absolutely choke here…but with PyArrow, I can stay in my comfy little Python world and still get data *fast*.

1.4.1 Submitting a GDS Job

We'll submit a GDS job that reads directly from the in-memory graph. In this case, it's analagous to something like:

```
CALL gds.graph.streamNodeProperties('mygraph', ['n'])
```

Suppose we already have this graph populated via something like:

```
CALL gds.graph.create('mygraph', 'Node', { EDGE: { orientation: 'UNDIRECTED'} });
CALL gds.fastRP.mutate('mygraph', {
```

```
    embeddingDimension: 256,
    mutateProperty: 'n'
});
```

Note that this follows the same general flow as before: submit job, get ticket, get stream

[10]:
```python
# Time for something completely different...
# Submit our GDS job to retrieve some node embeddings from a graph projection
ticket = client.gds_nodes('mygraph', ['n'])
client.wait_for_job(ticket, timeout=5)

print('''Reading the result of our GDS job into a dataframe.
Please be patient...this takes a minute or two (quite literally)...''')

# Retrieve and consume the stream directly into a Pandas DataFrame (and time it)
start = time.time()
df = client.stream(ticket).read_pandas()
delta = round(time.time() - start, 1)

# Voila!
print(f'Read our stream entirely into a dataframe in {delta} seconds!')
print(df)
print(f'Our dataframe has {len(df):,} vectors')
print('Memory usage looks like (in MiB):')
df.memory_usage(deep=True) / (1024 * 1024)
```

```
Reading the result of our GDS job into a dataframe.
Please be patient…this takes a minute or two (quite literally)…
Read our stream entirely into a dataframe in 77.3 seconds!
          nodeId                                                 n
0              0  [0.130765, 0.05128011, -0.075689286, -0.216142…
1              1  [0.12185934, 0.04503689, 0.08228396, -0.180549…
2              2  [0.11455238, -0.05738143, 0.12366827, -0.06347…
3              3  [-0.042987622, 0.12796022, -0.06732363, 0.0147…
4              4  [0.095874734, 0.08834075, -0.00030730665, -0.1…
…            …                                                 …
9999995  9999995  [-0.17068014, -0.20275281, -0.02609725, 0.0159…
9999996  9999996  [0.08360965, 0.007767517, 0.12981923, 0.180894…
9999997  9999997  [-0.05578732, -0.033202175, -0.059255105, -0.0…
9999998  9999998  [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, …
9999999  9999999  [0.0005706702, -0.123747975, 0.021083286, -0.1…

[10000000 rows x 2 columns]
Our dataframe has 10,000,000 vectors
Memory usage looks like (in MiB):
```

[10]:
```
Index        0.000122
nodeId      76.293945
```

```
n         1068.115234
dtype: float64
```

[ ]:
```
# Time for something completely different...
ticket = client.gds_nodes('mygraph', ['n'])
client.wait_for_job(ticket, timeout=5)

print('''
Reading the result of our GDS job directly into a Pandas dataframe.
Please wait...this takes a minute or 2 (quite literally!)...
''')

start = time.time()
df = client.stream(ticket).read_pandas()
delta = round(time.time() - start, 1)

print(f'Read our stream entirely into a dataframe in {delta} seconds!')
print(df)
print(f'Our dataframe has {len(df):,} vectors and is about {df["n"].
 ↪memory_usage(deep=True) / (1024 * 1024):,} MiB!')
```

Now we'll do our little data conversion and PCA…

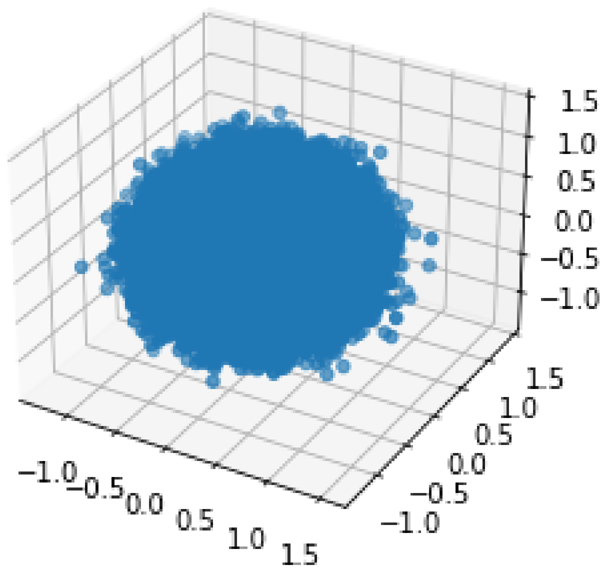
[11]:
```
# Select out just our embedding vectors and convert it to a numpy matrix
df = table.select(['embedding'])[0].to_pandas()
m = np.matrix(df.tolist())

# Then let's do dimensional reduction so we can plot our vectors in a lower␣
 ↪dimension
pca = PCA(n_components=3)
pc = pca.fit_transform(m)
print('Our new 3-dimensional vectors look like:')
print(pc)
```

```
Our new 3-dimensional vectors look like:
[[ 0.18794921 -0.16087489  0.4958695 ]
 [ 0.35928982  0.39344963 -0.01773353]
 [ 0.65337169  0.15338067  0.7251118 ]
 …
 [-0.19737952  0.15365793  0.46963622]
 [ 0.0545209   0.14824339 -0.28427196]
 [ 0.85537911  0.15831911  0.30382167]]
```

And plot!

[12]:
```
pc_df = pd.DataFrame(data=pc, columns=['PC1', 'PC2', 'PC3'])
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.scatter(pc_df['PC1'], pc_df['PC2'], pc_df['PC3'])
```

```
plt.show()
```



1.5 Towards the Future

Other languages supported by Apache Arrow: * R * Matlab * Julia * and more!

Some ponderings: * can we do better bulk updates/writes by moving data via Arrow to the server? * how about replicating an entire in-memory graph to another Neo4j system? * if we ditch the whole 2-step process (get ticket, get stream) how simple can our DX be?

```
[ ]:
```