Graphchain Documentation

Release 1.1.0

radix.ai

May 31, 2019

Contents

1 Graphchain **3**
- 1.1 What is graphchain? . . . 3
- 1.2 Usage by example . . . 3
  - 1.2.1 Basic usage . . . 3
  - 1.2.2 Storing the graphchain cache remotely . . . 5
  - 1.2.3 Excluding keys from being cached . . . 5
  - 1.2.4 Using graphchain with dask.delayed . . . 5
- 1.3 Developed by radix.ai . . . 6

2 API **7**

CHAPTER 1

Graphchain

1.1 What is graphchain?

Graphchain is like joblib.Memory for dask graphs. Dask graph computations are cached to a local or remote location of your choice, specified by a PyFilesystem FS URL.

When you change your dask graph (by changing a computation's implementation or its inputs), graphchain will take care to only recompute the minimum number of computations necessary to fetch the result. This allows you to iterate quickly over your graph without spending time on recomputing previously computed keys.

The main difference between graphchain and joblib.Memory is that in graphchain a computation's materialised inputs are *not* serialised and hashed (which can be very expensive when the inputs are large objects such as pandas DataFrames). Instead, a chain of hashes (hence the name graphchain) of the computation object and its dependencies (which are also computation objects) is used to identify the cache file.

Additionally, the result of a computation is only cached if it is estimated that loading that computation from cache will save time compared to simply computing the computation. The decision on whether to cache depends on the characteristics of the cache location, which are different when caching to the local filesystem compared to caching to S3 for example.

1.2 Usage by example

1.2.1 Basic usage

Install graphchain with pip to get started:

```
pip install graphchain
```

To demonstrate how graphchain can save you time, let's first create a simple dask graph that (1) creates a few pandas DataFrames, (2) runs a relatively heavy operation on these DataFrames, and (3) summarises the results.

```
import dask
import graphchain
import pandas as pd

def create_dataframe(num_rows, num_cols):
    print('Creating DataFrame...')
    return pd.DataFrame(data=[range(num_cols)]*num_rows)

def complicated_computation(df, num_quantiles):
    print('Running complicated computation on DataFrame...')
    return df.quantile(q=[i / num_quantiles for i in range(num_quantiles)])

def summarise_dataframes(*dfs):
    print('Summing DataFrames...')
    return sum(df.sum().sum() for df in dfs)

dsk = {
    'df_a': (create_dataframe, 10_000, 1000),
    'df_b': (create_dataframe, 10_000, 1000),
    'df_c': (complicated_computation, 'df_a', 2048),
    'df_d': (complicated_computation, 'df_b', 2048),
    'result': (summarise_dataframes, 'df_c', 'df_d')
}
```

Using `dask.get` to fetch the `'result'` key takes about 6 seconds:

```
>>> %time dask.get(dsk, 'result')

Creating DataFrame...
Running complicated computation on DataFrame...
Creating DataFrame...
Running complicated computation on DataFrame...
Summing DataFrames...

CPU times: user 7.39 s, sys: 686 ms, total: 8.08 s
Wall time: 6.19 s
```

On the other hand, using `graphchain.get` for the first time to fetch `'result'` takes only 4 seconds:

```
>>> %time graphchain.get(dsk, 'result')

Creating DataFrame...
Running complicated computation on DataFrame...
Summing DataFrames...

CPU times: user 4.7 s, sys: 519 ms, total: 5.22 s
Wall time: 4.04 s
```

The reason `graphchain.get` is faster than `dask.get` is because it can load `df_b` and `df_d` from cache after `df_a` and `df_c` have been computed and cached. Note that graphchain will only cache the result of a computation if loading that computation from cache is estimated to be faster than simply running the computation.

Running `graphchain.get` a second time to fetch `'result'` will be almost instant since this time the result itself is also available from cache:

```
>>> %time graphchain.get(dsk, 'result')
```

(continues on next page)

(continued from previous page)

```
CPU times: user 4.79 ms, sys: 1.79 ms, total: 6.58 ms
Wall time: 5.34 ms
```

Now let's say we want to change how the result is summarised from a sum to an average:

```
def summarise_dataframes(*dfs):
    print('Averaging DataFrames...')
    return sum(df.mean().mean() for df in dfs) / len(dfs)
```

If we then ask graphchain to fetch `'result'`, it will detect that only `summarise_dataframes` has changed and therefore only recompute this function with inputs loaded from cache:

```
>>> %time graphchain.get(dsk, 'result')

Averaging DataFrames...

CPU times: user 123 ms, sys: 37.2 ms, total: 160 ms
Wall time: 86.6 ms
```

1.2.2 Storing the graphchain cache remotely

Graphchain's cache is by default `./__graphchain_cache__`, but you can ask graphchain to use a cache at any PyFilesystem FS URL such as `s3://mybucket/__graphchain_cache__`:

```
graphchain.get(dsk, 'result', location='s3://mybucket/__graphchain_cache__')
```

1.2.3 Excluding keys from being cached

In some cases you may not want a key to be cached. To avoid writing certain keys to the graphchain cache, you can use the `skip_keys` argument:

```
graphchain.get(dsk, 'result', skip_keys=['result'])
```

1.2.4 Using graphchain with dask.delayed

Alternatively, you can use graphchain together with dask.delayed for easier dask graph creation:

```
@dask.delayed
def create_dataframe(num_rows, num_cols):
    print('Creating DataFrame...')
    return pd.DataFrame(data=[range(num_cols)]*num_rows)

@dask.delayed
def complicated_computation(df, num_quantiles):
    print('Running complicated computation on DataFrame...')
    return df.quantile(q=[i / num_quantiles for i in range(num_quantiles)])

@dask.delayed
def summarise_dataframes(*dfs):
    print('Summing DataFrames...')
    return sum(df.sum().sum() for df in dfs)
```

(continues on next page)

(continued from previous page)

```
df_a = create_dataframe(num_rows=50_000, num_cols=500, seed=42)
df_b = create_dataframe(num_rows=50_000, num_cols=500, seed=42)
df_c = complicated_computation(df_a, window=3)
df_d = complicated_computation(df_b, window=3)
result = summarise_dataframes(df_c, df_d)
```

After which you can compute `result` by setting the `delayed_optimize` method to `graphchain.optimize`:

```
with dask.config.set(scheduler='sync', delayed_optimize=graphchain.optimize):
    result.compute(location='s3://mybucket/__graphchain_cache__')
```

1.3 Developed by radix.ai

At radix.ai, we invent, design and develop AI-powered software.

Here are some examples of what we do with Machine Learning, the technology behind AI:

- Help job seekers find a job. On the Belgian Public Employment Service website, we serve job recommendations based on your CV.
- Help hospitals save time. We extract diagnoses from patient discharge letters.
- Help publishers calculate their impact, by detecting copycat articles.

You can follow our adventures on medium.

CHAPTER 2

API