06_1_Intro_Dask

January 28, 2021

[ ]:

1 Tutorial: Feature Engineering for Recommender Systems

2 6. Scaling to Production Systems

2.1 6.1. Introduction to dask and dask_cudf

2.2 Theory

Acknowledgement: Much of the introductory material included here is borrowed from other Dask documentation and tutorials. - "Dask Video Tutorial" - YouTube link - Introduction To Dask by Richard (Rick) Zamora

Other useful Dask resources: - Dask.org - Tutorial pages - GitHub Tutorial

2.3 What is Dask

Very Short Answer: Dask is an open-source library designed to natively scale Python code.

Slightly-Longer Short Answer: Dask is a task-based library for parallel scheduling and execution. Although it is certainly possible to use the task-scheduling machinery directly to implement customized parallel workflows (we do it in NVTabular), most users only interact with Dask through a *Dask Collection API*. The most popular "collection" API's include:

- Dask DataFrame: Dask-based version of the Pandas DataFrame/Series API. Note that `dask_cudf` is just a wrapper around this collection module (`dask.dataframe`).
- Dask Array: Dask-based version of the NumPy array API
- Dask Bag: *Similar to* a Dask-based version of PyToolz or a Pythonic version of PySpark RDD

For example, Dask DataFrame provides a convenient API for decomposing large pandas (or cuDF) DataFrame/Series objects into a collection of DataFrame *partitions*. This tutorial will focus mostly on this particular Dask collection (since it is the basis for `dask_cudf`). However, instead of relying only on the established `dask.dataframe` API, we will also see how it is possible (perhaps easy) to implement a custom task graph to operate on Dask-DataFrame objects when necessary.

2.3.1 Dask Uses DAGs Internally

Before we start writing any code, it is useful to understand (on a basic level) how Dask actually works. When an application or library uses a Dask collection API (like Dask DataFrame), they are typically using that API to construct a directed acyclic graph (DAG) of tasks. Once a DAG is constructed, the **core** Dask API can be used (either directly or implicitly through the collection API) to schedule and execute the DAG on one or more threads/processes.

In other words, Dask provides various APIs to:

1. Construct a DAG of "tasks"
2. Schedule/execute those DAGs
3. (Optionally) Spin up a dedicated worker and scheduler processes to enable distributed execution

2.3.2 Important Components of the "Dask Ecosystem"

The components of the Dask ecosystem that are most critical for NVTabular (and will be discussed in this tutorial) are:

- `dask` (core Dask library): [GitHub Repo] This is the core Dask library. It also contains the Dask Dataframe API (`dask.dataframe`)
- `dask_cudf`: [GitHub Repo] This is effectively a wrapper around the `dask.dataframe` module defined in the core Dask library. Note that a `dask_cudf.DataFrame` object should be thought of as a `dask.dataframe.DataFrame` object, but with the underlying partitions being `cudf.DataFrame`'s (rather than `pandas.DataFrame`)
- `distributed`: [GitHub Repo] Distributed version of the Dask execution model (includes the necessary code for scheduling, execution and communication between distributed processes). This library does not deal with the construction of DAGs, just with the scheduling and execution of DAGs on distributed *workers*.

- `dask_cuda`: [GitHub Repo] Provides various utilities to improve deployment and management of distributed Dask *workers* on CUDA-enabled systems.

2.4 HandsOn

Before we get started, it is convenient to create a simple `dask.distributed` client. If we work with a small dataset, then it is not necessary to initialize a `dask.distributed` client. The code should run in the same way.

[1]:
```
import dask
from dask.distributed import Client, LocalCluster
import dask.dataframe as dd
```

[2]:
```
client = Client(n_workers=8,
                threads_per_worker=1,
                memory_limit='50GB',
                ip='127.0.0.1')
client
```

[2]:
```
<Client: 'tcp://127.0.0.1:40757' processes=8 threads=8, memory=132.13 GB>
```

[3]:
```
%%time

ddf_train = dd.read_parquet('./data/train.parquet', blocksize=12e3)
ddf_valid = dd.read_parquet('./data/valid.parquet', blocksize=12e3)
```

```
CPU times: user 28 ms, sys: 0 ns, total: 28 ms
Wall time: 27.4 ms
```

[4]:
```
ddf_train
```

[4]:
```
Dask DataFrame Structure:
              event_time event_type product_id   brand    price user_id
user_session target   cat_0   cat_1   cat_2   cat_3 timestamp ts_hour ts_minute
ts_weekday ts_day ts_month ts_year
npartitions=1
                  object     object       int64  object  float64   int64
object  int64  object  object  object  object    object   int64     int64
int64  int64   int64   int64
                     …        …        …      …       …      …
…    …     …     …     …     …       …     …       …
…    …      …     …
Dask Name: read-parquet, 1 tasks
```

Here we have created a `dask.dataframe.DataFrame` object called `ddf_train` and `ddf_valid`. Both are essentially a (**lazy**) collection of pandas dataframes. Dask loaded the metadata (DataFrame schema) but did not load any data in-memory. Each pandas dataframe in this collection is called a **partition**. We can access this property (the total number of partitions) using the

DataFrame.npartitions attribute.

It is absolutely critical to recognize that ddf_train and ddf_valid are *not* actually backed by *in-memory* pandas data, but instead by a DAG of tasks. This DAG (accessible via ddf.dask) specifies the exact network of operations needed to produce the underlying partitions.

[5]:
```
ddf_train._meta
```

[5]:
```
Empty DataFrame
Columns: [event_time, event_type, product_id, brand, price, user_id,
user_session, target, cat_0, cat_1, cat_2, cat_3, timestamp, ts_hour, ts_minute,
ts_weekday, ts_day, ts_month, ts_year]
Index: []
```

Let's work on some examples: Simplified Target Encoding 1. We combine two columns cat_2 and brand 2. We TargetEncode the new column cat_2_brand 3. We merge the counts back to the train and validation dataset 4. We overwrite counts with less than 20 for on cat_2_brand with global_mean

We can see that the execution time is 117ms - meaning that dask has registered the operations but hasn't executed them.

[6]:
```
%%time

ddf_train['cat_2_brand'] = ddf_train['cat_2'].astype(str) + '_' +
 ↪ddf_train['brand'].astype(str)
ddf_valid['cat_2_brand'] = ddf_valid['cat_2'].astype(str) + '_' +
 ↪ddf_valid['brand'].astype(str)

ddf_train_group = ddf_train[['cat_2_brand', 'target']].groupby(['cat_2_brand']).
 ↪agg(['count', 'mean'])
ddf_train_group = ddf_train_group.reset_index()
ddf_train_group.columns = ['cat_2_brand', 'TE_count', 'TE_mean']
ddf_train = ddf_train.merge(ddf_train_group, how='left', on='cat_2_brand')
ddf_valid = ddf_valid.merge(ddf_train_group, how='left', on='cat_2_brand')
global_mean = ddf_train['target'].mean()
ddf_train['TE_mean'] = ddf_train.TE_mean.where(ddf_train['TE_count']>20,
 ↪global_mean)
ddf_valid['TE_mean'] = ddf_valid.TE_mean.where(ddf_valid['TE_count']>20,
 ↪global_mean)
```

```
CPU times: user 60 ms, sys: 0 ns, total: 60 ms
Wall time: 56.7 ms
```

We can compute the task graph by calling .compute() or .persist()

[7]:
```
%%time

ddf_train.compute()
```

```
ddf_valid.compute()
```

```
CPU times: user 15.1 s, sys: 9.68 s, total: 24.8 s
Wall time: 3min 14s
```

[7]:

```
                     event_time event_type  product_id      brand   price  \
0       2020-03-01 00:00:59 UTC       cart     6902464     zlatek   49.91
1       2020-03-01 00:01:20 UTC       cart     1002544      apple  397.10
2       2020-03-01 00:01:52 UTC       cart     1003316      apple  823.70
3       2020-03-01 00:02:14 UTC       cart    16600067  rivertoys  422.15
4       2020-03-01 00:02:15 UTC       cart     3701428    arnica   69.24
…                             …         …           …         …       …
2461714 2020-03-31 23:57:47 UTC   purchase    24100293  cocochoco    2.65
2461715 2020-03-31 23:58:19 UTC   purchase   100049773       None  234.96
2461716 2020-03-31 23:58:20 UTC   purchase     3700689    samsung  223.92
2461717 2020-03-31 23:59:19 UTC   purchase   100077607      vitek  100.36
2461718 2020-03-31 23:59:27 UTC   purchase   100068493    samsung  319.41

           user_id                          user_session  target  \
0        531574188  48714293-b3f9-4946-8135-eb1ea05ead74       0
1        622090790  fb5b918c-f1f6-48d9-bcf4-7eb46e83fc6b       0
2        622090543  b821ee79-96fe-4979-be9d-21ee2e6777c3       0
3        616437533  aad023bc-c858-47ab-a3a7-ff4654f11b9a       0
4        516454226  ee22b80c-ed3e-3c83-d397-fb69a44d4864       0
…              …                                     …       …
2461714  513094047  d27f822c-f707-4956-a6c3-4ad8fec00cc7       1
2461715  620580925  c33fde42-a5de-4a1f-9e1c-2ac7518a7d41       1
2461716  514905289  e40783c5-7b21-429f-99af-539d2842e6d3       1
2461717  633281427  667a8535-221c-4169-aab4-a1972610f102       1
2461718  635165435  861f2378-076f-4ddd-85e3-9844923d03a9       1

               cat_0        cat_1  …           timestamp ts_hour  \
0        electronics    telephone  … 2020-03-01 00:00:59       0
1       construction        tools  … 2020-03-01 00:01:20       0
2       construction        tools  … 2020-03-01 00:01:52       0
3              sport      trainer  … 2020-03-01 00:02:14       0
4         appliances  environment  … 2020-03-01 00:02:15       0
…                 …            …   …                   …       …
2461714   appliances     personal  … 2020-03-31 23:57:47      23
2461715         None         None  … 2020-03-31 23:58:19      23
2461716   appliances  environment  … 2020-03-31 23:58:20      23
2461717   appliances  environment  … 2020-03-31 23:59:19      23
2461718 construction        tools  … 2020-03-31 23:59:27      23

        ts_minute  ts_weekday  ts_day  ts_month  ts_year      cat_2_brand  \
0               0           6       1         3     2020       nan_zlatek
1               1           6       1         3     2020      light_apple
```

```
2                1          6      1        3      2020         light_apple
3                2          6      1        3      2020       nan_rivertoys
4                2          6      1        3      2020       vacuum_arnica
…           …         …      …      …        …              …
2461714         57          1     31        3      2020  massager_cocochoco
2461715         58          1     31        3      2020             nan_nan
2461716         58          1     31        3      2020      vacuum_samsung
2461717         59          1     31        3      2020        vacuum_vitek
2461718         59          1     31        3      2020       light_samsung

          TE_count   TE_mean
0            607.0  0.258649
1        1013391.0  0.469441
2        1013391.0  0.469441
3          10564.0  0.104411
4           4450.0  0.325393
…              …       …
2461714       82.0  0.146341
2461715   521515.0  0.277106
2461716    68239.0  0.392459
2461717    11950.0  0.340000
2461718  1212393.0  0.481047

[2461719 rows x 22 columns]
```

[8]:
```
client.close()
```

About `compute`: The `compute` method is defined for all Dask collections. For Dask DataFrame, this method will (1) trigger the execution of the graph and (2) convert the Dask DataFrame into a **single** Pandas DataFrame. *This means that you should be sure the pandas equivalent of **`ddf`** will fit in memory before you use **`compute`**!*

Using `persist`

Since the `compute` method will convert your Dask DataFrame to a Pandas DataFrame, it is typically a **bad** idea to use compute on larger-than-memory (LTM) datasets. In NVTabular, we do use a `compute` method, but never on a full Dask/dask_cudf DataFrame object. Instead, we use `compute` to trigger the collection/reduction of an aggregated statistics dictionary, and/or to write out a processed dataset.

In order to execute the `ddf` DAG **without** converting it to a single pandas DataFrame, you need to use the `persist` method. This function is particularly useful when using distributed systems, because the results will be kept in distributed memory, rather than returned to the local process as with compute. It will also allow the distributed cluster to clean up data that the scheduler no longer deems necessary. For the single-machine case, the method is used less often.

Let's move on to the GPU accelerated version with dask_cudf.

We can use `nvidia-smi` command to check the usage of our GPU.

[9]:
```
!nvidia-smi
```

```
Mon Sep 21 14:10:27 2020
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 440.64.00    Driver Version: 440.64.00    CUDA Version: 10.2     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  Tesla T4            Off  | 00000000:00:1E.0 Off |                    0 |
| N/A   31C    P8     9W /  70W |      0MiB / 15109MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                       GPU Memory |
|  GPU       PID   Type   Process name                             Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
```

[10]:
```
import dask as dask, dask_cudf
from dask.distributed import Client
from dask_cuda import LocalCUDACluster
```

[11]:
```
cluster = LocalCUDACluster(ip='127.0.0.1',
                           rmm_pool_size="16GB")
client = Client(cluster)
client
```

[11]:
```
<Client: 'tcp://127.0.0.1:34691' processes=1 threads=1, memory=16.52 GB>
```

We reserve 14GB per GPU via `rmm_pool_size`.

[12]:
```
!nvidia-smi
```

```
Mon Sep 21 14:10:33 2020
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 440.64.00    Driver Version: 440.64.00    CUDA Version: 10.2     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  Tesla T4            Off  | 00000000:00:1E.0 Off |                    0 |
| N/A   33C    P0    26W /  70W |    605MiB / 15109MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                       GPU Memory |
```

```
|  GPU       PID   Type   Process name                             Usage      |
|=============================================================================|
+-----------------------------------------------------------------------------+
```

We use `dask_cudf` to read and load the data. The remaining code is exactly the same as the dask pandas version.

[13]:
```
%%time

ddf_train = dask_cudf.read_parquet('./data/train.parquet')
ddf_valid = dask_cudf.read_parquet('./data/valid.parquet')
```

```
CPU times: user 996 ms, sys: 416 ms, total: 1.41 s
Wall time: 1.41 s
```

[14]:
```
%%time

ddf_train['cat_2_brand'] = ddf_train['cat_2'].astype(str) + '_' +␣
↪ddf_train['brand'].astype(str)
ddf_valid['cat_2_brand'] = ddf_valid['cat_2'].astype(str) + '_' +␣
↪ddf_valid['brand'].astype(str)

ddf_train_group = ddf_train[['cat_2_brand', 'target']].groupby(['cat_2_brand']).
↪agg(['count', 'mean'])
ddf_train_group = ddf_train_group.reset_index()
ddf_train_group.columns = ['cat_2_brand', 'TE_count', 'TE_mean']
ddf_train = ddf_train.merge(ddf_train_group, how='left', on='cat_2_brand')
ddf_valid = ddf_valid.merge(ddf_train_group, how='left', on='cat_2_brand')
global_mean = ddf_train['target'].mean()
ddf_train['TE_mean'] = ddf_train.TE_mean.where(ddf_train['TE_count']>20,␣
↪global_mean)
ddf_valid['TE_mean'] = ddf_valid.TE_mean.where(ddf_valid['TE_count']>20,␣
↪global_mean)
```

```
CPU times: user 544 ms, sys: 0 ns, total: 544 ms
Wall time: 652 ms
```

[15]:
```
%%time

ddf_train.compute()
ddf_valid.compute()
```

```
CPU times: user 2.75 s, sys: 5.77 s, total: 8.52 s
Wall time: 14.3 s
```

[15]:
```
                       event_time event_type  product_id    brand    price  \
0         2020-03-01 08:16:04 UTC       cart     1005135    apple  1516.10
1         2020-03-01 08:16:08 UTC       cart     1005135    apple  1516.10
```

```
2        2020-03-01 08:16:09 UTC      cart    1004996  doogee    96.89
3        2020-03-01 08:16:09 UTC      cart    1005135   apple  1516.10
4        2020-03-01 08:16:13 UTC      cart    1005256  xiaomi   141.29
…                            …        …          …     …        …
2461714  2020-03-31 19:25:14 UTC  purchase   18301044    <NA>    11.04
2461715  2020-03-31 19:25:17 UTC  purchase  100058915    iqos    43.76
2461716  2020-03-31 19:25:36 UTC  purchase   32401283    <NA>    22.97
2461717  2020-03-31 19:26:18 UTC  purchase    4800282 samsung    38.59
2461718  2020-03-31 19:26:20 UTC  purchase  100058915    iqos    43.76

           user_id                          user_session  target  \
0        620967403  2f69a6e0-3a9e-4b7c-b717-ce5b8ad85ce3       0
1        620967403  2f69a6e0-3a9e-4b7c-b717-ce5b8ad85ce3       0
2        607174356  80d6850c-7f95-4978-ba1a-dedbe802e012       0
3        620967403  2f69a6e0-3a9e-4b7c-b717-ce5b8ad85ce3       0
4        571788375  da050faa-118a-405a-b9c8-63f9d730328e       0
…              …                                     …     …
2461714  572119027  172b36e9-9259-423c-bc43-5d555ff94ce4       1
2461715  620477097  47786b4a-f2c3-48fa-b714-9d05556d5b98       1
2461716  635102002  d82b8bf0-dea5-4e53-84f8-e61332eb17f1       1
2461717  622434648  4894c1b9-d00d-4418-b6c9-e8cd2f842b33       1
2461718  620477097  47786b4a-f2c3-48fa-b714-9d05556d5b98       1

                cat_0      cat_1  …            timestamp ts_hour  ts_minute  \
0        construction      tools  …  2020-03-01 08:16:04       8         16
1        construction      tools  …  2020-03-01 08:16:08       8         16
2        construction      tools  …  2020-03-01 08:16:09       8         16
3        construction      tools  …  2020-03-01 08:16:09       8         16
4        construction      tools  …  2020-03-01 08:16:13       8         16
…                   …        …  …                    …     …        …
2461714         sport        ski  …  2020-03-31 19:25:14      19         25
2461715       apparel   trousers  …  2020-03-31 19:25:17      19         25
2461716       apparel  underwear  …  2020-03-31 19:25:36      19         25
2461717         sport    bicycle  …  2020-03-31 19:26:18      19         26
2461718       apparel   trousers  …  2020-03-31 19:26:20      19         26

         ts_weekday  ts_day  ts_month  ts_year   cat_2_brand TE_count  \
0                 6       1         3     2020   light_apple  1013391
1                 6       1         3     2020   light_apple  1013391
2                 6       1         3     2020  light_doogee      769
3                 6       1         3     2020   light_apple  1013391
4                 6       1         3     2020  light_xiaomi   510657
…                 …     …       …      …             …      …
2461714           1      31         3     2020          <NA>     <NA>
2461715           1      31         3     2020          <NA>     <NA>
2461716           1      31         3     2020          <NA>     <NA>
2461717           1      31         3     2020          <NA>     <NA>
```

```
2461718           1      31          3     2020           <NA>     <NA>

          TE_mean
0        0.469441
1        0.469441
2        0.405722
3        0.469441
4        0.396346
…             …
2461714  0.366924
2461715  0.366924
2461716  0.366924
2461717  0.366924
2461718  0.366924

[2461719 rows x 22 columns]
```

[16]:
```
!nvidia-smi
```

```
Mon Sep 21 14:10:50 2020
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 440.64.00    Driver Version: 440.64.00    CUDA Version: 10.2     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  Tesla T4            Off  | 00000000:00:1E.0 Off |                    0 |
| N/A   36C    P0    33W /  70W |   1629MiB / 15109MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                       GPU Memory |
|  GPU       PID   Type   Process name                             Usage      |
|=============================================================================|
+-----------------------------------------------------------------------------+
```

[17]:
```
client.close()
```