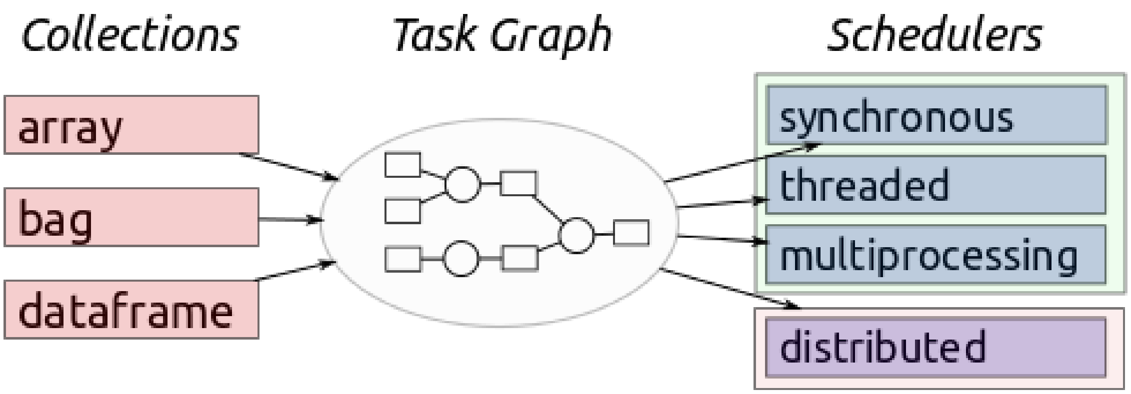
16-DaskDataframes

August 11, 2020

1 Dask Dataframes

Dask is a flexible parallel computing library for analytic computing written in Python. Dask is similar to Spark, by lazily constructing directed acyclic graph (DAG) of tasks and splitting large datasets into small portions called partitions. See the below image from Dask's web page for illustration.



It has three main interfaces:

- Array, which works like NumPy arrays;
- Bag, which is similar to RDD interface in Spark;
- DataFrame, which works like Pandas DataFrame.

While it can work on a distributed cluster, Dask works also very well on a single cpu machine.

1.1 DataFrames

Dask dataframes look and feel (mostly) like Pandas dataframes but they run on the same infrastructure that powers dask.delayed.

The `dask.dataframe` module implements a blocked parallel `DataFrame` object that mimics a large subset of the Pandas `DataFrame`. One dask `DataFrame` is comprised of many in-memory pandas `DataFrames` separated along the index. One operation on a dask `DataFrame` triggers many pandas operations on the constituent pandas `DataFrames` in a way that is mindful of potential parallelism and memory constraints.

Related Documentation

- Dask DataFrame documentation
- Pandas documentation

In this notebook, we will extracts some historical flight data for flights out of NYC between 1990 and 2000. The data is taken from here. This should only take a few seconds to run.

We will use `dask.dataframe` construct our computations for us. The `dask.dataframe.read_csv` function can take a globstring like `"data/nycflights/*.csv"` and build parallel computations on all of our data at once.

1.1.1 Prep the Data

[1]:
```python
import os
import pandas as pd
pd.set_option("max.rows", 10)
os.getcwd()
```

[1]:
```
'/home/runner/work/big-data/big-data/notebooks'
```

[2]:
```python
import os  # library to get directory and file paths
import tarfile # this module makes possible to read and write tar archives

def extract_flight():
    here = os.getcwd()
    flightdir = os.path.join(here,'data', 'nycflights')
    if not os.path.exists(flightdir):
       print("Extracting flight data")
       tar_path = os.path.join('data', 'nycflights.tar.gz')
       with tarfile.open(tar_path, mode='r:gz') as flights:
          flights.extractall('data/')

extract_flight() # this function call will extract 10 csv files in data/
 ↪nycflights
```

```
Extracting flight data
```

1.1.2 Load Data from CSVs in Dask Dataframes

[3]:
```python
import os
here = os.getcwd()
filename = os.path.join(here, 'data', 'nycflights', '*.csv')
filename
```

[3]:
```
'/home/runner/work/big-data/big-data/notebooks/data/nycflights/*.csv'
```

[4]:
```python
import dask
import dask.dataframe as dd

df = dd.read_csv(filename,
                 parse_dates={'Date': [0, 1, 2]})
```

Let's take a look to the dataframe

[5]:
```python
df
```

[5]:
```
Dask DataFrame Structure:
                          Date DayOfWeek  DepTime CRSDepTime  ArrTime CRSArrTime
UniqueCarrier FlightNum  TailNum ActualElapsedTime CRSElapsedTime  AirTime
ArrDelay DepDelay  Origin    Dest Distance   TaxiIn  TaxiOut Cancelled Diverted
npartitions=10
                datetime64[ns]     int64  float64       int64  float64       int64
object     int64  float64           float64          int64  float64  float64
float64  object  object  float64  float64  float64     int64    int64
                           …       …      …        …      …        …
…       …      …               …            …      …      …
…     …     …      …      …      …       …      …
…                        …      …      …        …      …        …
…       …      …               …            …      …      …
…     …     …      …      …      …       …      …
                           …       …      …        …      …        …
…       …      …               …            …      …      …
…     …     …      …      …      …       …      …
                           …       …      …        …      …        …
…       …      …               …            …      …      …
…     …     …      …      …      …       …      …
Dask Name: read-csv, 10 tasks
```

[6]:
```python
### Get the first 5 rows
df.head()
```

[6]:
```
        Date  DayOfWeek  DepTime  CRSDepTime  ArrTime  CRSArrTime  \
0 1990-01-01          1   1621.0        1540   1747.0        1701
1 1990-01-02          2   1547.0        1540   1700.0        1701
2 1990-01-03          3   1546.0        1540   1710.0        1701
3 1990-01-04          4   1542.0        1540   1710.0        1701
4 1990-01-05          5   1549.0        1540   1706.0        1701

  UniqueCarrier  FlightNum  TailNum  ActualElapsedTime  …  AirTime  \
0            US         33      NaN               86.0  …      NaN
1            US         33      NaN               73.0  …      NaN
2            US         33      NaN               84.0  …      NaN
3            US         33      NaN               88.0  …      NaN
```

```
4           US         33      NaN              77.0  …      NaN

   ArrDelay  DepDelay  Origin Dest Distance  TaxiIn  TaxiOut  Cancelled  \
0      46.0      41.0     EWR  PIT    319.0     NaN      NaN          0
1      -1.0       7.0     EWR  PIT    319.0     NaN      NaN          0
2       9.0       6.0     EWR  PIT    319.0     NaN      NaN          0
3       9.0       2.0     EWR  PIT    319.0     NaN      NaN          0
4       5.0       9.0     EWR  PIT    319.0     NaN      NaN          0

   Diverted
0         0
1         0
2         0
3         0
4         0

[5 rows x 21 columns]
```

[7]:

```
import traceback # we use traceback because we except an error.

try:
    df.tail() # Get the last 5 rows
except Exception:
    traceback.print_exc()
```

```
Traceback (most recent call last):
  File "<ipython-input-7-7cb27b738c02>", line 4, in <module>
    df.tail() # Get the last 5 rows
  File "/usr/share/miniconda3/envs/big-data/lib/python3.8/site-
packages/dask/dataframe/core.py", line 1053, in tail
    result = result.compute()
  File "/usr/share/miniconda3/envs/big-data/lib/python3.8/site-
packages/dask/base.py", line 167, in compute
    (result,) = compute(self, traverse=False, **kwargs)
  File "/usr/share/miniconda3/envs/big-data/lib/python3.8/site-
packages/dask/base.py", line 447, in compute
    results = schedule(dsk, keys, **kwargs)
  File "/usr/share/miniconda3/envs/big-data/lib/python3.8/site-
packages/dask/threaded.py", line 76, in get
    results = get_async(
  File "/usr/share/miniconda3/envs/big-data/lib/python3.8/site-
packages/dask/local.py", line 486, in get_async
    raise_exception(exc, tb)
  File "/usr/share/miniconda3/envs/big-data/lib/python3.8/site-
packages/dask/local.py", line 316, in reraise
    raise exc
  File "/usr/share/miniconda3/envs/big-data/lib/python3.8/site-
```

```
packages/dask/local.py", line 222, in execute_task
    result = _execute_task(task, data)
  File "/usr/share/miniconda3/envs/big-data/lib/python3.8/site-
packages/dask/core.py", line 121, in _execute_task
    return func(*(_execute_task(a, cache) for a in args))
  File "/usr/share/miniconda3/envs/big-data/lib/python3.8/site-
packages/dask/core.py", line 121, in <genexpr>
    return func(*(_execute_task(a, cache) for a in args))
  File "/usr/share/miniconda3/envs/big-data/lib/python3.8/site-
packages/dask/core.py", line 121, in _execute_task
    return func(*(_execute_task(a, cache) for a in args))
  File "/usr/share/miniconda3/envs/big-data/lib/python3.8/site-
packages/dask/dataframe/io/csv.py", line 151, in pandas_read_text
    coerce_dtypes(df, dtypes)
  File "/usr/share/miniconda3/envs/big-data/lib/python3.8/site-
packages/dask/dataframe/io/csv.py", line 255, in coerce_dtypes
    raise ValueError(msg)
ValueError: Mismatched dtypes found in `pd.read_csv`/`pd.read_table`.

+----------------+---------+----------+
| Column         | Found   | Expected |
+----------------+---------+----------+
| CRSElapsedTime | float64 | int64    |
| TailNum        | object  | float64  |
+----------------+---------+----------+

The following columns also raised exceptions on conversion:

- TailNum
  ValueError("could not convert string to float: 'N54711'")

Usually this is due to dask's dtype inference failing, and
*may* be fixed by specifying dtypes manually by adding:

dtype={'CRSElapsedTime': 'float64',
       'TailNum': 'object'}

to the call to `read_csv`/`read_table`.
```

1.1.3 What just happened?

Unlike `pandas.read_csv` which reads in the entire file before inferring datatypes, `dask.dataframe.read_csv` only reads in a sample from the beginning of the file (or first file if using a glob). These inferred datatypes are then enforced when reading all partitions.

In this case, the datatypes inferred in the sample are incorrect. The first `n` rows have no value for `CRSElapsedTime` (which pandas infers as a `float`), and later on turn out to be strings (`object` dtype). When this happens you have a few options:

- Specify dtypes directly using the `dtype` keyword. This is the recommended solution, as it's the least error prone (better to be explicit than implicit) and also the most performant.
- Increase the size of the `sample` keyword (in bytes)
- Use `assume_missing` to make `dask` assume that columns inferred to be `int` (which don't allow missing values) are actually floats (which do allow missing values). In our particular case this doesn't apply.

In our case we'll use the first option and directly specify the `dtypes` of the offending columns.

[8]:
```
df.dtypes
```

[8]:
```
Date           datetime64[ns]
DayOfWeek               int64
DepTime               float64
CRSDepTime              int64
ArrTime               float64
                    …
Distance              float64
TaxiIn                float64
TaxiOut               float64
Cancelled               int64
Diverted                int64
Length: 21, dtype: object
```

[9]:
```
df = dd.read_csv(filename,
                 parse_dates={'Date': [0, 1, 2]},
                 dtype={'TailNum': object,
                        'CRSElapsedTime': float,
                        'Cancelled': bool})
```

[10]:
```
df.tail()
```

[10]:
```
             Date  DayOfWeek  DepTime  CRSDepTime  ArrTime  CRSArrTime  \
269176 1999-12-27          1   1645.0        1645   1830.0        1901
269177 1999-12-28          2   1726.0        1645   1928.0        1901
269178 1999-12-29          3   1646.0        1645   1846.0        1901
269179 1999-12-30          4   1651.0        1645   1908.0        1901
269180 1999-12-31          5   1642.0        1645   1851.0        1901

       UniqueCarrier  FlightNum TailNum  ActualElapsedTime  …  AirTime  \
269176            UA       1753  N516UA              225.0  …    205.0
269177            UA       1753  N504UA              242.0  …    214.0
269178            UA       1753  N592UA              240.0  …    220.0
269179            UA       1753  N575UA              257.0  …    233.0
269180            UA       1753  N539UA              249.0  …    232.0
```

```
        ArrDelay  DepDelay  Origin Dest Distance  TaxiIn  TaxiOut  Cancelled  \
269176     -31.0       0.0     LGA  DEN   1619.0     7.0     13.0      False
269177      27.0      41.0     LGA  DEN   1619.0     5.0     23.0      False
269178     -15.0       1.0     LGA  DEN   1619.0     5.0     15.0      False
269179       7.0       6.0     LGA  DEN   1619.0     5.0     19.0      False
269180     -10.0      -3.0     LGA  DEN   1619.0     6.0     11.0      False

        Diverted
269176         0
269177         0
269178         0
269179         0
269180         0

[5 rows x 21 columns]
```

Let's take a look at one more example to fix ideas.

[11]:
```
len(df)
```

[11]:
```
2611892
```

1.1.4 Why df is ten times longer ?

- Dask investigated the input path and found that there are ten matching files.
- A set of jobs was intelligently created for each chunk - one per original CSV file in this case.
- Each file was loaded into a pandas dataframe, had `len()` applied to it.
- The subtotals were combined to give you the final grant total.

1.2 Computations with `dask.dataframe`

We compute the maximum of the `DepDelay` column. With `dask.delayed` we could create this computation as follows:

```
maxes = []
for fn in filenames:
    df = dask.delayed(pd.read_csv)(fn)
    maxes.append(df.DepDelay.max())

final_max = dask.delayed(max)(maxes)
final_max.compute()
```

Now we just use the normal Pandas syntax as follows:

[12]:
```
%time df.DepDelay.max().compute()
```

```
CPU times: user 4.18 s, sys: 354 ms, total: 4.53 s
Wall time: 3.07 s
```

[12]:
```
1435.0
```

This writes the delayed computation for us and then runs it. Recall that the delayed computation is a dask graph made of up of key-value pairs.

Some things to note:

1. As with `dask.delayed`, we need to call `.compute()` when we're done. Up until this point everything is lazy.
2. Dask will delete intermediate results (like the full pandas dataframe for each file) as soon as possible.
   - This lets us handle datasets that are larger than memory
   - This means that repeated computations will have to load all of the data in each time (run the code above again, is it faster or slower than you would expect?)

As with `Delayed` objects, you can view the underlying task graph using the `.visualize` method:

[13]:
```
df.DepDelay.max().visualize()
```

```
 ␣
↪---------------------------------------------------------------------------

    FileNotFoundError                         Traceback (most recent call␣
↪last)

    /usr/share/miniconda3/envs/big-data/lib/python3.8/site-packages/graphviz/
↪backend.py in run(cmd, input, capture_output, check, encoding, quiet, **kwargs)
        165     try:
    --> 166         proc = subprocess.Popen(cmd, startupinfo=get_startupinfo(),␣
↪**kwargs)
        167     except OSError as e:


    /usr/share/miniconda3/envs/big-data/lib/python3.8/subprocess.py in␣
↪__init__(self, args, bufsize, executable, stdin, stdout, stderr, preexec_fn,␣
↪close_fds, shell, cwd, env, universal_newlines, startupinfo, creationflags,␣
↪restore_signals, start_new_session, pass_fds, encoding, errors, text)
        853
    --> 854             self._execute_child(args, executable, preexec_fn,␣
↪close_fds,
        855                                 pass_fds, cwd, env,
```

```
    /usr/share/miniconda3/envs/big-data/lib/python3.8/subprocess.py in␣
↪_execute_child(self, args, executable, preexec_fn, close_fds, pass_fds, cwd,␣
↪env, startupinfo, creationflags, shell, p2cread, p2cwrite, c2pread, c2pwrite,␣
↪errread, errwrite, restore_signals, start_new_session)
   1701                         err_msg = os.strerror(errno_num)
-> 1702                     raise child_exception_type(errno_num, err_msg,␣
↪err_filename)
   1703                 raise child_exception_type(err_msg)


    FileNotFoundError: [Errno 2] No such file or directory: 'dot'


During handling of the above exception, another exception occurred:


    ExecutableNotFound                        Traceback (most recent call␣
↪last)

    <ipython-input-13-5a7336c66be3> in <module>
----> 1 df.DepDelay.max().visualize()


    /usr/share/miniconda3/envs/big-data/lib/python3.8/site-packages/dask/
↪base.py in visualize(self, filename, format, optimize_graph, **kwargs)
     91         https://docs.dask.org/en/latest/optimize.html
     92         """
---> 93         return visualize(
     94             self,
     95             filename=filename,


    /usr/share/miniconda3/envs/big-data/lib/python3.8/site-packages/dask/
↪base.py in visualize(*args, **kwargs)
    551         raise NotImplementedError("Unknown value color=%s" % color)
    552
--> 553     return dot_graph(dsk, filename=filename, **kwargs)
    554
    555


    /usr/share/miniconda3/envs/big-data/lib/python3.8/site-packages/dask/dot.
↪py in dot_graph(dsk, filename, format, **kwargs)
    270     """
    271     g = to_graphviz(dsk, **kwargs)
--> 272     return graphviz_to_file(g, filename, format)
    273
```

```
    274


    /usr/share/miniconda3/envs/big-data/lib/python3.8/site-packages/dask/dot.
↪py in graphviz_to_file(g, filename, format)
    282         format = "png"
    283
--> 284     data = g.pipe(format=format)
    285     if not data:
    286         raise RuntimeError(


    /usr/share/miniconda3/envs/big-data/lib/python3.8/site-packages/graphviz/
↪files.py in pipe(self, format, renderer, formatter, quiet)
    134         data = text_type(self.source).encode(self._encoding)
    135
--> 136         out = backend.pipe(self._engine, format, data,
    137                            renderer=renderer, formatter=formatter,
    138                            quiet=quiet)


    /usr/share/miniconda3/envs/big-data/lib/python3.8/site-packages/graphviz/
↪backend.py in pipe(engine, format, data, renderer, formatter, quiet)
    244     """
    245     cmd, _ = command(engine, format, None, renderer, formatter)
--> 246     out, _ = run(cmd, input=data, capture_output=True, check=True,␣
↪quiet=quiet)
    247     return out
    248


    /usr/share/miniconda3/envs/big-data/lib/python3.8/site-packages/graphviz/
↪backend.py in run(cmd, input, capture_output, check, encoding, quiet, **kwargs)
    167     except OSError as e:
    168         if e.errno == errno.ENOENT:
--> 169             raise ExecutableNotFound(cmd)
    170         else:
    171             raise


    ExecutableNotFound: failed to execute ['dot', '-Tpng'], make sure the␣
↪Graphviz executables are on your systems' PATH
```

If you are already familiar with the Pandas API then know how to use `dask.dataframe`. There are a couple of small changes.

As noted above, computations on dask `DataFrame` objects don't perform work, instead they build up a dask graph. We can evaluate this dask graph at any time using the `.compute()` method.

[14]:
```
result = df.DepDelay.mean()  # create the tasks graph
```

[15]:
```
%time result.compute()           # perform actual computation
```

```
CPU times: user 4.17 s, sys: 374 ms, total: 4.55 s
Wall time: 3.05 s
```

[15]:
```
9.206602541321965
```

1.3 Store Data in Apache Parquet Format

Dask encourage dataframe users to store and load data using Parquet instead. Apache Parquet is a columnar binary format that is easy to split into multiple files (easier for parallel loading) and is generally much simpler to deal with than HDF5 (from the Dask library's perspective). It is also a common format used by other big data systems like Apache Spark and Apache Impala and so is useful to interchange with other systems.

[16]:
```
df.drop("TailNum", axis=1).to_parquet("nycflights/")  # save csv files using␣
↪parquet format
```

```
/usr/share/miniconda3/envs/big-data/lib/python3.8/site-
packages/pyarrow/compat.py:24: FutureWarning: pyarrow.compat has been deprecated
and will be removed in a future release
  warnings.warn("pyarrow.compat has been deprecated and will be removed in a "
```

It is possible to specify dtypes and compression when converting. This can definitely help give you significantly greater speedups, but just using the default settings will still be a large improvement.

[17]:
```
df.size.compute()
```

[17]:
```
54849732
```

[18]:
```
import dask.dataframe as dd
df = dd.read_parquet("nycflights/")
df.head()
```

[18]:
```
         Date  DayOfWeek  DepTime  CRSDepTime  ArrTime  CRSArrTime  \
0 1990-01-01          1   1621.0        1540   1747.0        1701
1 1990-01-02          2   1547.0        1540   1700.0        1701
2 1990-01-03          3   1546.0        1540   1710.0        1701
3 1990-01-04          4   1542.0        1540   1710.0        1701
4 1990-01-05          5   1549.0        1540   1706.0        1701

  UniqueCarrier  FlightNum  ActualElapsedTime  CRSElapsedTime  AirTime  \
0            US         33               86.0            81.0      NaN
1            US         33               73.0            81.0      NaN
```

```
2           US        33                84.0              81.0       NaN
3           US        33                88.0              81.0       NaN
4           US        33                77.0              81.0       NaN

   ArrDelay  DepDelay Origin Dest  Distance  TaxiIn  TaxiOut  Cancelled  \
0      46.0      41.0    EWR  PIT     319.0     NaN      NaN      False
1      -1.0       7.0    EWR  PIT     319.0     NaN      NaN      False
2       9.0       6.0    EWR  PIT     319.0     NaN      NaN      False
3       9.0       2.0    EWR  PIT     319.0     NaN      NaN      False
4       5.0       9.0    EWR  PIT     319.0     NaN      NaN      False

   Diverted
0         0
1         0
2         0
3         0
4         0
```

[19]:
```
result = df.DepDelay.mean()
```

[20]:
```
%time result.compute()
```

```
CPU times: user 136 ms, sys: 13.6 ms, total: 150 ms
Wall time: 96 ms
```

[20]:
```
9.206602541321965
```

The computation is much faster because pulling out the DepDelay column is easy for Parquet.

1.3.1 Parquet advantages:

- Binary representation of data, allowing for speedy conversion of bytes-on-disk to bytes-in-memory
- Columnar storage, meaning that you can load in as few columns as you need without loading the entire dataset
- Row-chunked storage so that you can pull out data from a particular range without touching the others
- Per-chunk statistics so that you can find subsets quickly
- Compression

1.3.2 Exercise 15.1

If you don't remember how to use pandas. Please read pandas documentation.

- Use the `head()` method to get the first ten rows
- How many rows are in our dataset?

- Use selections `df[...]` to find how many positive (late) and negative (early) departure times there are
- In total, how many non-cancelled flights were taken? (To invert a boolean pandas Series s, use ~s).

[21]:
```
df.head(10)
```

[21]:
```
        Date  DayOfWeek  DepTime  CRSDepTime  ArrTime  CRSArrTime  \
0 1990-01-01          1   1621.0        1540   1747.0        1701
1 1990-01-02          2   1547.0        1540   1700.0        1701
2 1990-01-03          3   1546.0        1540   1710.0        1701
3 1990-01-04          4   1542.0        1540   1710.0        1701
4 1990-01-05          5   1549.0        1540   1706.0        1701
5 1990-01-06          6   1539.0        1540   1653.0        1701
6 1990-01-07          7   1553.0        1540   1713.0        1701
7 1990-01-08          1   1543.0        1540   1656.0        1701
8 1990-01-09          2   1540.0        1540   1704.0        1701
9 1990-01-10          3   1608.0        1540   1740.0        1701

  UniqueCarrier  FlightNum  ActualElapsedTime  CRSElapsedTime  AirTime  \
0            US         33               86.0            81.0      NaN
1            US         33               73.0            81.0      NaN
2            US         33               84.0            81.0      NaN
3            US         33               88.0            81.0      NaN
4            US         33               77.0            81.0      NaN
5            US         33               74.0            81.0      NaN
6            US         33               80.0            81.0      NaN
7            US         33               73.0            81.0      NaN
8            US         33               84.0            81.0      NaN
9            US         33               92.0            81.0      NaN

   ArrDelay  DepDelay Origin Dest  Distance  TaxiIn  TaxiOut  Cancelled  \
0      46.0      41.0    EWR  PIT     319.0     NaN      NaN      False
1      -1.0       7.0    EWR  PIT     319.0     NaN      NaN      False
2       9.0       6.0    EWR  PIT     319.0     NaN      NaN      False
3       9.0       2.0    EWR  PIT     319.0     NaN      NaN      False
4       5.0       9.0    EWR  PIT     319.0     NaN      NaN      False
5      -8.0      -1.0    EWR  PIT     319.0     NaN      NaN      False
6      12.0      13.0    EWR  PIT     319.0     NaN      NaN      False
7      -5.0       3.0    EWR  PIT     319.0     NaN      NaN      False
8       3.0       0.0    EWR  PIT     319.0     NaN      NaN      False
9      39.0      28.0    EWR  PIT     319.0     NaN      NaN      False

   Diverted
0         0
1         0
2         0
```

```
3          0
4          0
5          0
6          0
7          0
8          0
9          0
```

[22]:
```
len(df)
```

[22]:
```
2611892
```

[23]:
```
len(df[df.DepDelay > 0])
```

[23]:
```
1187146
```

[24]:
```
len(df[df.DepDelay < 0])
```

[24]:
```
840942
```

[25]:
```
len(df[~df.Cancelled])
```

[25]:
```
2540961
```

1.4 Divisions and the Index

The Pandas index associates a value to each record/row of your data. Operations that align with the index, like `loc` can be a bit faster as a result.

In `dask.dataframe` this index becomes even more important. Recall that one dask `DataFrame` consists of several Pandas `DataFrame`s. These dataframes are separated along the index by value. For example, when working with time series we may partition our large dataset by month.

Recall that these many partitions of our data may not all live in memory at the same time, instead they might live on disk; we simply have tasks that can materialize these pandas `DataFrames` on demand.

Partitioning your data can greatly improve efficiency. Operations like `loc`, `groupby`, and `merge/join` along the index are *much more efficient* than operations along other columns. You can see how your dataset is partitioned with the `.divisions` attribute. Note that data that comes out of simple data sources like CSV files aren't intelligently indexed by default. In these cases the values for `.divisions` will be `None`.

[26]:
```
df = dd.read_csv(filename,
                 dtype={'TailNum': str,
                        'CRSElapsedTime': float,
                        'Cancelled': bool})
df.divisions
```

[26]: (None, None, None, None, None, None, None, None, None, None, None)

However if we set the index to some new column then dask will divide our data roughly evenly along that column and create new divisions for us. Warning, `set_index` triggers immediate computation.

[27]:
```
df2 = df.set_index('Year')
df2.divisions
```

[27]: (1990, 1991, 1992, 1993, 1994, 1995, 1996, 1997, 1998, 1999, 1999)

We see here the minimum and maximum values (1990 and 1999) as well as the intermediate values that separate our data well. This dataset has ten partitions, as the final value is assumed to be the inclusive right-side for the last bin.

[28]:
```
df2.npartitions
```

[28]: 10

[29]:
```
df2.head()
```

[29]:
```
      Month  DayofMonth  DayOfWeek  DepTime  CRSDepTime  ArrTime  CRSArrTime  \
Year
1990      1           1          1   1621.0        1540   1747.0        1701
1990      1           2          2   1547.0        1540   1700.0        1701
1990      1           3          3   1546.0        1540   1710.0        1701
1990      1           4          4   1542.0        1540   1710.0        1701
1990      1           5          5   1549.0        1540   1706.0        1701

     UniqueCarrier  FlightNum TailNum  …  AirTime  ArrDelay  DepDelay  \
Year                                   …
1990            US         33     NaN  …      NaN      46.0      41.0
1990            US         33     NaN  …      NaN      -1.0       7.0
1990            US         33     NaN  …      NaN       9.0       6.0
1990            US         33     NaN  …      NaN       9.0       2.0
1990            US         33     NaN  …      NaN       5.0       9.0

      Origin  Dest Distance TaxiIn  TaxiOut  Cancelled  Diverted
Year
1990     EWR   PIT    319.0    NaN      NaN      False         0
1990     EWR   PIT    319.0    NaN      NaN      False         0
1990     EWR   PIT    319.0    NaN      NaN      False         0
1990     EWR   PIT    319.0    NaN      NaN      False         0
1990     EWR   PIT    319.0    NaN      NaN      False         0

[5 rows x 22 columns]
```

One of the benefits of this is that operations like `loc` only need to load the relevant partitions

[30]:
```
df2.loc[1991]
```

[30]:
```
Dask DataFrame Structure:
               Month DayofMonth DayOfWeek  DepTime CRSDepTime  ArrTime
CRSArrTime UniqueCarrier FlightNum TailNum ActualElapsedTime CRSElapsedTime
AirTime ArrDelay DepDelay  Origin    Dest Distance   TaxiIn  TaxiOut Cancelled
Diverted
npartitions=1
1991           int64      int64     int64  float64      int64  float64
int64         object     int64  object          float64        float64  float64
float64  float64  object  object  float64  float64  float64      bool    int64
1991             …        …       …      …        …      …
…           …       …     …                …            …      …
…      …     …     …      …      …      …       …      …
Dask Name: loc, 31 tasks
```

[31]:
```
df2.loc[1991].compute()
```

[31]:
```
      Month  DayofMonth  DayOfWeek  DepTime  CRSDepTime  ArrTime  CRSArrTime  \
Year
1991      1           8          2   1215.0        1215   1340.0        1336
1991      1           9          3   1215.0        1215   1353.0        1336
1991      1          10          4   1216.0        1215   1332.0        1336
1991      1          11          5   1303.0        1215   1439.0        1336
1991      1          12          6   1215.0        1215   1352.0        1336
…     …         …         …       …        …       …        …
1991     12          26          4   1600.0        1600   1857.0        1906
1991     12          27          5   1600.0        1600   1853.0        1906
1991     12          28          6   1600.0        1600   1856.0        1906
1991     12          29          7   1601.0        1600   1851.0        1906
1991     12          31          2   1558.0        1600   1851.0        1906

     UniqueCarrier  FlightNum TailNum  …  AirTime  ArrDelay  DepDelay  \
Year                                   …
1991            US        121     NaN  …      NaN       4.0       0.0
1991            US        121     NaN  …      NaN      17.0       0.0
1991            US        121     NaN  …      NaN      -4.0       1.0
1991            US        121     NaN  …      NaN      63.0      48.0
1991            US        121     NaN  …      NaN      16.0       0.0
…             …        …     …   …       …       …       …
1991            CO       1539     NaN  …      NaN      -9.0       0.0
1991            CO       1539     NaN  …      NaN     -13.0       0.0
1991            CO       1539     NaN  …      NaN     -10.0       0.0
1991            CO       1539     NaN  …      NaN     -15.0       1.0
1991            CO       1539     NaN  …      NaN     -15.0      -2.0

      Origin  Dest Distance TaxiIn  TaxiOut  Cancelled  Diverted
```

```
Year
1991     EWR   PIT    319.0    NaN      NaN      False          0
1991     EWR   PIT    319.0    NaN      NaN      False          0
1991     EWR   PIT    319.0    NaN      NaN      False          0
1991     EWR   PIT    319.0    NaN      NaN      False          0
1991     EWR   PIT    319.0    NaN      NaN      False          0
…      …   …      …     …      …      …      …
1991     LGA   FLL   1076.0    NaN      NaN      False          0
1991     LGA   FLL   1076.0    NaN      NaN      False          0
1991     LGA   FLL   1076.0    NaN      NaN      False          0
1991     LGA   FLL   1076.0    NaN      NaN      False          0
1991     LGA   FLL   1076.0    NaN      NaN      False          0

[258274 rows x 22 columns]
```

1.4.1 Exercises 15.2

In this section we do a few `dask.dataframe` computations. If you are comfortable with Pandas then these should be familiar. You will have to think about when to call `compute`.

- In total, how many non-cancelled flights were taken from each airport?

Hint: use `df.groupby`. `df.groupby(df.A).B.func()`.

- What was the average departure delay from each airport?

Note, this is the same computation you did in the previous notebook (is this approach faster or slower?)

- What day of the week has the worst average departure delay?

[32]:
```
df = dd.read_parquet("nycflights/")
```

[33]:
```
df[~df.Cancelled].groupby("Origin").Origin.count().compute()
```

[33]:
```
Origin
EWR    1139451
JFK     427243
LGA     974267
Name: Origin, dtype: int64
```

[34]:
```
df[~df.Cancelled].groupby("Origin").DepDelay.count().compute()
```

[34]:
```
Origin
EWR    1139451
JFK     427243
LGA     974267
Name: DepDelay, dtype: int64
```

1.5 Sharing Intermediate Results

When computing all of the above, we sometimes did the same operation more than once. For most operations, `dask.dataframe` hashes the arguments, allowing duplicate computations to be shared, and only computed once.

For example, lets compute the mean and standard deviation for departure delay of all non-cancelled flights:

[35]:
```
non_cancelled = df[~df.Cancelled]
mean_delay = non_cancelled.DepDelay.mean()
std_delay = non_cancelled.DepDelay.std()
```

Using two calls to `.compute`:

[36]:
```
%%time
mean_delay_res = mean_delay.compute()
std_delay_res = std_delay.compute()
```

```
CPU times: user 3.07 s, sys: 189 ms, total: 3.26 s
Wall time: 2.25 s
```

Using one call to `dask.compute`:

[37]:
```
%%time
mean_delay_res, std_delay_res = dask.compute(mean_delay, std_delay)
```

```
CPU times: user 1.56 s, sys: 110 ms, total: 1.67 s
Wall time: 1.16 s
```

Using `dask.compute` takes roughly 1/2 the time. This is because the task graphs for both results are merged when calling `dask.compute`, allowing shared operations to only be done once instead of twice. In particular, using `dask.compute` only does the following once:

- the calls to `read_csv`
- the filter (`df[~df.Cancelled]`)
- some of the necessary reductions (`sum`, `count`)

To see what the merged task graphs between multiple results look like (and what's shared), you can use the `dask.visualize` function (we might want to use `filename='graph.pdf'` to zoom in on the graph better):

[38]:
```
dask.visualize(mean_delay, std_delay)
```

```
␣
↪---------------------------------------------------------------------------

    FileNotFoundError                         Traceback (most recent call␣
↪last)
```

```
    /usr/share/miniconda3/envs/big-data/lib/python3.8/site-packages/graphviz/
↪backend.py in run(cmd, input, capture_output, check, encoding, quiet, **kwargs)
    165     try:
--> 166         proc = subprocess.Popen(cmd, startupinfo=get_startupinfo(),␣
↪**kwargs)
    167     except OSError as e:


    /usr/share/miniconda3/envs/big-data/lib/python3.8/subprocess.py in␣
↪__init__(self, args, bufsize, executable, stdin, stdout, stderr, preexec_fn,␣
↪close_fds, shell, cwd, env, universal_newlines, startupinfo, creationflags,␣
↪restore_signals, start_new_session, pass_fds, encoding, errors, text)
    853
--> 854             self._execute_child(args, executable, preexec_fn,␣
↪close_fds,
    855                                 pass_fds, cwd, env,


    /usr/share/miniconda3/envs/big-data/lib/python3.8/subprocess.py in␣
↪_execute_child(self, args, executable, preexec_fn, close_fds, pass_fds, cwd,␣
↪env, startupinfo, creationflags, shell, p2cread, p2cwrite, c2pread, c2pwrite,␣
↪errread, errwrite, restore_signals, start_new_session)
   1701                         err_msg = os.strerror(errno_num)
-> 1702                     raise child_exception_type(errno_num, err_msg,␣
↪err_filename)
   1703                 raise child_exception_type(err_msg)


    FileNotFoundError: [Errno 2] No such file or directory: 'dot'


During handling of the above exception, another exception occurred:


    ExecutableNotFound                        Traceback (most recent call␣
↪last)

    <ipython-input-38-547954d62040> in <module>
----> 1 dask.visualize(mean_delay, std_delay)


    /usr/share/miniconda3/envs/big-data/lib/python3.8/site-packages/dask/
↪base.py in visualize(*args, **kwargs)
    551         raise NotImplementedError("Unknown value color=%s" % color)
    552
--> 553     return dot_graph(dsk, filename=filename, **kwargs)
```

```
    554
    555

/usr/share/miniconda3/envs/big-data/lib/python3.8/site-packages/dask/dot.py in dot_graph(dsk, filename, format, **kwargs)
    270     """
    271     g = to_graphviz(dsk, **kwargs)
--> 272     return graphviz_to_file(g, filename, format)
    273
    274

/usr/share/miniconda3/envs/big-data/lib/python3.8/site-packages/dask/dot.py in graphviz_to_file(g, filename, format)
    282         format = "png"
    283
--> 284     data = g.pipe(format=format)
    285     if not data:
    286         raise RuntimeError(

/usr/share/miniconda3/envs/big-data/lib/python3.8/site-packages/graphviz/files.py in pipe(self, format, renderer, formatter, quiet)
    134         data = text_type(self.source).encode(self._encoding)
    135
--> 136         out = backend.pipe(self._engine, format, data,
    137                            renderer=renderer, formatter=formatter,
    138                            quiet=quiet)

/usr/share/miniconda3/envs/big-data/lib/python3.8/site-packages/graphviz/backend.py in pipe(engine, format, data, renderer, formatter, quiet)
    244     """
    245     cmd, _ = command(engine, format, None, renderer, formatter)
--> 246     out, _ = run(cmd, input=data, capture_output=True, check=True, quiet=quiet)
    247     return out
    248

/usr/share/miniconda3/envs/big-data/lib/python3.8/site-packages/graphviz/backend.py in run(cmd, input, capture_output, check, encoding, quiet, **kwargs)
    167     except OSError as e:
    168         if e.errno == errno.ENOENT:
--> 169             raise ExecutableNotFound(cmd)
    170         else:
```

```
    171             raise

    ExecutableNotFound: failed to execute ['dot', '-Tpng'], make sure the Graphviz executables are on your systems' PATH
```