Dask Icon

Pandas Icon

Gotcha's from Pandas to Dask

https://github.com/sephib/dask_pyconil2019 (https://github.com/sephib/dask_pyconil2019)

This notebook highlights some key differences when transfering code from `Pandas` to run in a `Dask` environment. Most issues have a link to the Dask documentation (https://docs.dask.org/en/latest/) for additional information.

Agenda

1. Intro to `Dask` framework
2. Basic setup `Client`
3. Dask.dataframe
4. Data manipulation
5. Read/Write files
6. Advanced `groupby`
7. Debugging

Dask Icon

Dask is a flexible library for parallel computing in Python.

Dask Framework

Dask is composed of two parts:

1. *Dynamic task scheduling* optimized for computation. This is similar to Airflow, Luigi, Celery, or Make, but optimized for interactive computational workloads.
2. *"Big Data" collections* like parallel arrays, dataframes, and lists that extend common interfaces like NumPy, Pandas, or Python iterators to larger-than-memory or distributed environments. These parallel collections run on top of dynamic task schedulers.

link to documentation (https://docs.dask.org/en/latest/)

Dask emphasizes the following virtues:

- Familiar: Provides parallelized NumPy array and Pandas DataFrame objects
- Flexible: Provides a task scheduling interface for more custom workloads and integration with other projects.
- Native: Enables distributed computing in pure Python with access to the PyData stack.
- Fast: Operates with low overhead, low latency, and minimal serialization necessary for fast numerical algorithms
- Scales up: Runs resiliently on clusters with 1000s of cores
- Scales down: Trivial to set up and run on a laptop in a single process
- Responsive: Designed with interactive computing in mind, it provides rapid feedback and diagnostics to aid humans

See the dask.distributed documentation (separate website) (https://distributed.dask.org/en/latest/) for more technical information on Dask's distributed scheduler.

Why Dask? (https://docs.dask.org/en/latest/why.html)

- Scales from single comptuer out to clusters
- Familiar API
- Responsive feedback (live dashboard)
- ...

In [1]:
```
# since Dask is activly beeing developed - the current example is running with
the below version
import dask
import dask.dataframe as dd
import pandas as pd
print(f'Dask versoin: {dask.__version__}')
print(f'Pandas versoin: {pd.__version__}')
```

```
Dask versoin: 1.2.2
Pandas versoin: 0.24.2
```

Dask `Distributed` scheduler

```
import dask.dataframe as dd
from dask.distributed import Client
client = Client()
df = dd.read_csv(...) # do something
```

vs

When running code within a script use a `context manager`

```
if __name__ == '__main__':
    with Client() as client:
        df = dd.read_csv(...) # do something
```

- see question in stack overflow (https://stackoverflow.com/a/53520917/5817977)
- In order to get url dashboard use inner function (https://github.com/dask/distributed/issues/2083#issue-337057906)

Start Dask Client for Dashboard

Dask Dashboard

In [2]:
```
from dask.distributed import Client
# client = Client(n_workers=1, threads_per_worker=4, processes=False, memory_l
imit='2GB')
client = Client()
client
```

Out[2]:

Client	Cluster
• **Scheduler:** tcp://127.0.0.1:45679	• **Workers:** 4
• **Dashboard:** http://127.0.0.1:8787/status (http://127.0.0.1:8787/status)	• **Cores:** 8
	• **Memory:** 67.44 GB

Starting the Dask Client is optional. In this example we are running on a `LocalCluster` , this will also provide a dashboard which is useful to gain insight on the computation.
For additional information on Dask Client see documentation (https://docs.dask.org/en/latest/setup.html?highlight=client#setup)

The link to the dashboard will become visible when you create a client (as shown below).
When running in `Jupyter Lab` an extenstion (https://github.com/dask/dask-labextension) can be installed to be able to view the various dashboard widgets.

See documentation for addtional cluster configuration (http://distributed.dask.org/en/latest/local-cluster.html)

Create 2 DataFrames for comparison:

- `Dask framework` is **lazy**

  lazy python

In [3]:
```
ddf = dask.datasets.timeseries() #  Dask comes with builtin dataset samples, w
e will use this sample for our example.
ddf
```

Out[3]: **Dask DataFrame Structure:**

	id	name	x	y
npartitions=30				
2000-01-01	int64	object	float64	float64
2000-01-02	...	...	...	...
...	...	...	...	...
2000-01-30	...	...	...	...
2000-01-31	...	...	...	...

Dask Name: make-timeseries, 30 tasks

In order to see the result we need to run compute() (https://docs.dask.org/en/latest/dataframe-api.html#dask.dataframe.DataFrame.compute) (or `head()` which runs under the hood compute()))

In [4]:
```
ddf.compute()
```

Out[4]:

timestamp	id	name	x	y
2000-01-01 00:00:00	960	Oliver	-0.745248	-0.198965
2000-01-01 00:00:01	1000	Dan	0.184641	0.295430
2000-01-01 00:00:02	1020	Norbert	-0.641957	-0.458981
2000-01-01 00:00:03	1015	Dan	-0.631978	0.454573
2000-01-01 00:00:04	998	Edith	0.508003	0.076637
2000-01-01 00:00:05	1025	Oliver	0.633132	-0.968848
2000-01-01 00:00:06	978	Ursula	-0.588642	-0.032841
2000-01-01 00:00:07	982	Charlie	0.119313	-0.518422
2000-01-01 00:00:08	1043	Victor	-0.895856	-0.954497
2000-01-01 00:00:09	1020	Wendy	0.436936	-0.312972
2000-01-01 00:00:10	988	Michael	0.286573	-0.615041
2000-01-01 00:00:11	1036	Sarah	-0.225445	0.062726
2000-01-01 00:00:12	983	Ray	-0.087305	0.392608
2000-01-01 00:00:13	992	Ursula	-0.268082	-0.700998
2000-01-01 00:00:14	1015	Zelda	-0.541838	0.364939
2000-01-01 00:00:15	1017	Michael	-0.373376	0.504668
2000-01-01 00:00:16	1022	Hannah	0.327997	-0.289495
2000-01-01 00:00:17	1054	Victor	-0.583977	-0.654631
2000-01-01 00:00:18	967	Frank	0.199236	-0.657553
2000-01-01 00:00:19	1022	Ray	-0.509110	0.923490
2000-01-01 00:00:20	960	Charlie	-0.003294	0.038744
2000-01-01 00:00:21	980	Charlie	-0.198883	0.541580
2000-01-01 00:00:22	1052	Oliver	-0.862566	-0.976609
2000-01-01 00:00:23	936	Tim	-0.970641	0.077440
2000-01-01 00:00:24	997	Edith	0.647717	0.591489
2000-01-01 00:00:25	1063	Ray	-0.650042	0.499804
2000-01-01 00:00:26	1060	Tim	-0.932838	0.016557
2000-01-01 00:00:27	1082	Victor	0.604015	-0.646004
2000-01-01 00:00:28	1019	Bob	-0.000948	0.933974
2000-01-01 00:00:29	979	Alice	-0.255095	0.991901
...	...	...	...	...
2000-01-30 23:59:30	959	Jerry	-0.382988	-0.685075
2000-01-30 23:59:31	1015	Kevin	-0.140484	-0.162593
2000-01-30 23:59:32	992	Tim	-0.731128	-0.814783
2000-01-30 23:59:33	1005	Alice	-0.270136	0.132457
2000-01-30 23:59:34	1002	Quinn	0.537519	-0.086152
2000-01-30 23:59:35	994	Ray	0.963644	-0.561226
2000-01-30 23:59:36	1013	Jerry	0.108260	-0.756177
2000-01-30 23:59:37	1001	Edith	0.201864	-0.345191

Pandas

In order to create a `Pandas` dataframe we can use the `compute()`

In [5]:
```
pdf = ddf.compute()
print(type(pdf))
pdf.head()
```

```
<class 'pandas.core.frame.DataFrame'>
```

Out[5]:

timestamp	id	name	x	y
2000-01-01 00:00:00	960	Oliver	-0.745248	-0.198965
2000-01-01 00:00:01	1000	Dan	0.184641	0.295430
2000-01-01 00:00:02	1020	Norbert	-0.641957	-0.458981
2000-01-01 00:00:03	1015	Dan	-0.631978	0.454573
2000-01-01 00:00:04	998	Edith	0.508003	0.076637

Creating a `Dask dataframe` from `Pandas`

In order to utilize `Dask` capablities on an existing `Pandas dataframe` (pdf) we need to convert the `Pandas dataframe` into a `Dask dataframe` (ddf) with the from_pandas (https://docs.dask.org/en/latest/dataframe-api.html#dask.dataframe.from_pandas) method. You must supply the number of `partitions` or `chunksize` that will be used to generate the dask dataframe

In [7]:
```
ddf2 = dd.from_pandas(pdf, npartitions=10)
print(type(ddf2))

ddf2.head()
```

```
<class 'dask.dataframe.core.DataFrame'>
```

Out[7]:

timestamp	id	name	x	y
2000-01-01 00:00:00	960	Oliver	-0.745248	-0.198965
2000-01-01 00:00:01	1000	Dan	0.184641	0.295430
2000-01-01 00:00:02	1020	Norbert	-0.641957	-0.458981
2000-01-01 00:00:03	1015	Dan	-0.631978	0.454573
2000-01-01 00:00:04	998	Edith	0.508003	0.076637

Partitions in Dask Dataframes

Notice that when we created a `Dask dataframe` we needed to supply an argument of `npartitions`.
The number of partitions will assist `Dask` on how it's going to parallelize the computation.
Each partition is a *separate* dataframe. For additional information see [partition documentation (https://docs.dask.org/en/latest/dataframe-design.html?highlight=meta%20utils#partitions)](https://docs.dask.org/en/latest/dataframe-design.html?highlight=meta%20utils#partitions)

Using `reset_index()` method we can examin the partitions:

In [8]:
```
pdf2 = pdf.reset_index()
pdf2.loc[0] # Only 1 row
```

Out[8]:
```
timestamp    2000-01-01 00:00:00
id                           960
name                      Oliver
x                       -0.745248
y                       -0.198965
Name: 0, dtype: object
```

Now lets look at a `Dask` dataframe

In [9]:
```
ddf2 = ddf2.reset_index()
ddf2.loc[0].compute()  # each partition has an index=0
```

Out[9]:

	timestamp	id	name	x	y
0	2000-01-01	960	Oliver	-0.745248	-0.198965
0	2000-01-04	1051	Jerry	-0.810596	0.703804
0	2000-01-07	1044	Sarah	0.622235	0.849257
0	2000-01-10	983	Jerry	0.472132	-0.323176
0	2000-01-13	1027	Ray	0.770773	0.441961
0	2000-01-16	1042	Norbert	0.611781	0.139298
0	2000-01-19	1043	Xavier	0.178683	-0.507595
0	2000-01-22	1004	Tim	-0.080435	0.210997
0	2000-01-25	965	Wendy	0.724862	0.148501
0	2000-01-28	1014	Bob	-0.929208	0.858530

dataframe.shape

since `Dask` is lazy we cannot get the full shape before running `len`

In [13]:
```
print(f'Pandas shape: {pdf.shape}')
print('--------------------------')
print(f'Dask lazy shape: {ddf.shape}')
```

```
Pandas shape: (2592000, 4)
--------------------------
Dask lazy shape: (Delayed('int-fecc34b2-1c31-40bf-b068-1159d4ab4cbc'), 4)
```

In [14]:
```
print(f'Dask computed shape: {len(ddf.index):,}')  # expensive
```

```
Dask computed shape: 2,592,000
```

Now that we have a `dask` (ddf) and a `pandas` (pdf) dataframe we can start to compair the interactions with them.

Moving from Update to Insert/Delete

inplaceTrue

Dask does not update - thus there are no arguments such as `inplace=True` which exist in Pandas.
For more detials see [issue#653 on github (https://github.com/dask/dask/issues/653)]

Rename Columns

In [15]:
```
# Pandas
print(pdf.columns)
pdf.rename(columns={'id':'ID'}, inplace=True)
pdf.columns
```

```
Index(['id', 'name', 'x', 'y'], dtype='object')
```

Out[15]:
```
Index(['ID', 'name', 'x', 'y'], dtype='object')
```

In [ ]:
```
# Dask - Error
# ddf.rename(columns={'id':'ID'}, inplace=True)
# ddf.columns

'''
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-12-3e70ff3a549e> in <module>
      1 # Dask - Error
----> 2 ddf.rename(columns={'id':'ID'}, inplace=True)
      3 ddf.columns
TypeError: rename() got an unexpected keyword argument 'inplace'
'''
```

- using `inplace=True` is *not* considerd to be *best practice*.

In [16]:
```
# Dask or Pandas
print(ddf.columns)
ddf = ddf.rename(columns={'id':'ID'})
ddf.columns
```

```
Index(['id', 'name', 'x', 'y'], dtype='object')
```

Out[16]:
```
Index(['ID', 'name', 'x', 'y'], dtype='object')
```

Data manipulation

loc - Pandas

In [17]:
```
mask_cond = (pdf['x']>0.5) & (pdf['x']<0.8)
pdf.loc[mask_cond, ['y']] = pdf['y']* 100
pdf[mask_cond].head(2)
```

Out[17]:

timestamp	ID	name	x	y
2000-01-01 00:00:04	998	Edith	0.508003	7.663685
2000-01-01 00:00:05	1025	Oliver	0.633132	-96.884843

In [ ]:
```
# Error
# cond_dask = (ddf['x']>0.5) & (ddf['x']<0.8)
# ddf.loc[cond_dask, ['y']] = ddf['y']* 100

'''
> TypeError                                 Traceback (most recent call last)
> <ipython-input-16-2bbb2ae570bd> in <module>
>       2 # Error
> ----> 3 ddf.loc[cond_dask, ['y']] = ddf['y']* 100
> TypeError: '_LocIndexer' object does not support item assignment
'''
```

Dask - use mask/where

```
# Pandas
mask_cond = (pdf['x']>0.5) & (pdf['x']<0.8)
pdf.loc[mask_cond, ['y']] = pdf['y']* 100
```

In [18]:
```
mask_cond = (ddf['x']>0.5) & (ddf['x']<0.8)

ddf['y'] = ddf['y'].mask(cond=mask_cond, other=ddf['y']* 100)
ddf[mask_cond].head(2)
```

Out[18]:

timestamp	ID	name	x	y
2000-01-01 00:00:04	998	Edith	0.508003	7.663685
2000-01-01 00:00:05	1025	Oliver	0.633132	-96.884843

dask mask documentation (https://docs.dask.org/en/latest/dataframe-api.html#dask.dataframe.DataFrame.mask)

Meta argument

> `meta` is the prescription of the names/types of the computation output
> see stack overflow answer (https://stackoverflow.com/questions/44432868/dask-dataframe-apply-meta)

crystal python
Since `Dask` creates a DAG for the computation it requires to understand what are the outputs of each calculation (see meta documentation (https://docs.dask.org/en/latest/dataframe-design.html?highlight=meta%20utils#metadata))

In [19]:
```
pdf['initials'] = pdf['name'].apply(lambda x: x[0]+x[1])
pdf.head(2)
```

Out[19]:

timestamp	ID	name	x	y	initials
2000-01-01 00:00:00	960	Oliver	-0.745248	-0.198965	Ol
2000-01-01 00:00:01	1000	Dan	0.184641	0.295430	Da

In [20]:
```
ddf['initials'] = ddf['name'].apply(lambda x: x[0]+x[1])
ddf.head(2)
```

```
/home/ds/.local/share/virtualenvs/dask_pyconil2019-9doxB0Ra/lib/python3.7/site
-packages/dask/dataframe/core.py:2345: UserWarning:
You did not provide metadata, so Dask is running your function on a small data
set to guess output types. It is possible that Dask will guess incorrectly.
To provide an explicit output types or to silence this message, please provide
the `meta=` keyword, as described in the map or apply function that you are us
ing.
  Before: .apply(func)
  After:  .apply(func, meta=('name', 'object'))

  warnings.warn(meta_warning(meta))
```

Out[20]:

timestamp	ID	name	x	y	initials
2000-01-01 00:00:00	960	Oliver	-0.745248	-0.198965	Ol
2000-01-01 00:00:01	1000	Dan	0.184641	0.295430	Da

Introducing meta argument

In [21]:
```
# Describe the outcome type of the calculation
meta_cal = pd.Series(object, name='initials')
ddf['initials'] = ddf['name'].apply(lambda x: x[0]+x[1]
                                    , meta = meta_cal)
ddf.head(2)
```

Out[21]:

timestamp	ID	name	x	y	initials
2000-01-01 00:00:00	960	Oliver	-0.745248	-0.198965	Ol
2000-01-01 00:00:01	1000	Dan	0.184641	0.295430	Da

In [22]:
```
def func(row, col1, col2):
    if (row[col1]> 0):  return row[col1] * 1000
    else:         return row[col2] * -1
ddf['z'] = ddf.apply(func, args=('x', 'y'), axis=1
                     , meta=('z', 'float'))
ddf.head(2)
```

Out[22]:

timestamp	ID	name	x	y	initials	z
2000-01-01 00:00:00	960	Oliver	-0.745248	-0.198965	OI	0.198965
2000-01-01 00:00:01	1000	Dan	0.184641	0.295430	Da	184.640930

Map partitions

- We can supply an ad-hoc function to run on each partition using the map_partitions (https://dask.readthedocs.io/en/latest/dataframe-api.html#dask.dataframe.DataFrame.map_partitions) method.
  Mainly useful for functions that are not implemented in `Dask` or `Pandas` .
- Finally we can return a new `dataframe` which needs to be described in the `meta` argument
  The function could also include arguments.

In [23]:
```
import numpy as np
def func2(df, coor_x, coor_y, drop_cols):
    df['dist'] =  np.sqrt ( (df[coor_x] - df[coor_x].shift())**2
                          +  (df[coor_y] - df[coor_y].shift())**2 )
    df = df.drop(drop_cols, axis=1)
    return df

ddf2 = ddf.map_partitions(func2
                          , coor_x='x'
                          , coor_y='y'
                          , drop_cols=['initials', 'z']
                          , meta=pd.DataFrame({'ID':'i8'
                                               , 'name':str
                                               , 'x':'f8'
                                               , 'y':'f8'
                                               , 'dist':'f8'}, index=[0]))
ddf2.head()
```

Out[23]:

timestamp	ID	name	x	y	dist
2000-01-01 00:00:00	960	Oliver	-0.745248	-0.198965	NaN
2000-01-01 00:00:01	1000	Dan	0.184641	0.295430	1.053148
2000-01-01 00:00:02	1020	Norbert	-0.641957	-0.458981	1.119107
2000-01-01 00:00:03	1015	Dan	-0.631978	0.454573	0.913609
2000-01-01 00:00:04	998	Edith	0.508003	7.663685	7.298689

Convert index into DateTime column

In [24]:
```
# Only Pandas
pdf = pdf.assign(times=pd.to_datetime(pdf.index).time)
pdf.head(2)
```

Out[24]:

timestamp	ID	name	x	y	initials	times
2000-01-01 00:00:00	960	Oliver	-0.745248	-0.198965	Ol	00:00:00
2000-01-01 00:00:01	1000	Dan	0.184641	0.295430	Da	00:00:01

In [25]:
```
# ddf.assign(times= dd.to_datetime(ddf.index).dt.time)
# Dask or Pandas
ddf = ddf.assign(times=ddf.index.astype('M8[ns]'))
ddf['times'] = ddf['times'].dt.time
ddf =client.persist(ddf)
ddf.head(2)
```

Out[25]:

timestamp	ID	name	x	y	initials	z	times
2000-01-01 00:00:00	960	Oliver	-0.745248	-0.198965	Ol	0.198965	00:00:00
2000-01-01 00:00:01	1000	Dan	0.184641	0.295430	Da	184.640930	00:00:01

Drop NA on column

In [26]:
```
pdf = pdf.drop(labels=['initials'],axis=1)
ddf = ddf.drop(labels=['initials','z'],axis=1)
```

In [27]:
```
pdf = pdf.assign(colna = None)
print(f'pandas: {pdf.head(1)}')
ddf = ddf.assign(colna = None)
print(f'dask: {ddf.head(1)}')
```

```
pandas:              ID    name         x         y     times colna
timestamp
2000-01-01  960  Oliver -0.745248 -0.198965  00:00:00  None
dask:              ID    name         x         y     times colna
timestamp
2000-01-01  960  Oliver -0.745248 -0.198965  00:00:00  None
```

In odrer for `Dask` to drop a column with all `na` we need to assist the graph

In [28]:
```
pdf = pdf.dropna(axis=1, how='all')
print(f'pandas: {pdf.head(1)}')
# check if all values in column are Null - expensive
if ddf.colna.isnull().all() == True:
    ddf = ddf.drop(labels=['colna'],axis=1)
print(f'dask: {ddf.compute().head(1)}')
```

```
pandas:              ID    name         x         y     times
timestamp
2000-01-01  960  Oliver -0.745248 -0.198965  00:00:00
dask:              ID    name         x         y     times
timestamp
2000-01-01  960  Oliver -0.745248 -0.198965  00:00:00
```

Reset Index

In [29]:
```
# Pandas
pdf = pdf.reset_index(drop=True)
pdf.head(1)
```

Out[29]:

	ID	name	x	y	times
0	960	Oliver	-0.745248	-0.198965	00:00:00

In [30]:
```
# Dask
ddf = ddf.reset_index()
ddf = ddf.drop(labels=['timestamp'], axis=1 )
ddf.head(1)
```

Out[30]:

	ID	name	x	y	times
0	960	Oliver	-0.745248	-0.198965	00:00:00

Read / Save files

When working with `pandas` and `dask` preferable try and work with parquet (https://docs.dask.org/en/latest/dataframe-best-practices.html?highlight=parquet#store-data-in-apache-parquet-format).
Even so when working with `Dask` - the files can be read with multiple workers .
Most `kwargs` are applicable for reading and writing files see documentaion (https://docs.dask.org/en/latest/dataframe-api.html#dask.dataframe.DataFrame.to_csv) (including the option for output file naming).
e.g. ddf = dd.read_csv('data/pd2dd/ddf*.csv', compression='gzip', header=False).

However some are not available such as `nrows` .

Save files

In [31]:
```
%%time
# Pandas
from pathlib import Path
output_file = 'pdf_single_file.csv'
output_dir = Path('data/')
output_dir.mkdir(parents=True, exist_ok=True)
pdf.to_csv(output_dir / output_file)
```

```
CPU times: user 16 s, sys: 34 ms, total: 16.1 s
Wall time: 16 s
```

In [32]:
```
list(Path(output_dir).glob('*.csv'))
```

Out[32]:
```
[PosixPath('data/pdf_single_file.csv')]
```

`Dask` Notice the '*' to allow for multiple file renaming.

In [33]:
```
%%time
# Dask
output_dask_dir = Path('data/pd2dd/')
output_dir.mkdir(parents=True, exist_ok=True)
ddf.to_csv(f'{output_dask_dir}/ddf*.csv', index = False)
```

```
CPU times: user 884 ms, sys: 76.6 ms, total: 961 ms
Wall time: 6.96 s
```

To find the number of partitions which will determine the number of output files use dask.dataframe.npartitions (https://docs.dask.org/en/latest/dataframe-api.html#dask.dataframe.DataFrame.npartitions)

In [34]:
```
ddf.npartitions
```

Out[34]:
```
30
```

In [35]:
```
list(Path(output_dask_dir).glob('*.csv'))
```

Out[35]:
```
[PosixPath('data/pd2dd/ddf20.csv'),
 PosixPath('data/pd2dd/ddf27.csv'),
 PosixPath('data/pd2dd/ddf06.csv'),
 PosixPath('data/pd2dd/ddf22.csv'),
 PosixPath('data/pd2dd/ddf18.csv'),
 PosixPath('data/pd2dd/ddf16.csv'),
 PosixPath('data/pd2dd/ddf03.csv'),
 PosixPath('data/pd2dd/ddf07.csv'),
 PosixPath('data/pd2dd/ddf12.csv'),
 PosixPath('data/pd2dd/ddf25.csv'),
 PosixPath('data/pd2dd/ddf10.csv'),
 PosixPath('data/pd2dd/ddf29.csv'),
 PosixPath('data/pd2dd/ddf09.csv'),
 PosixPath('data/pd2dd/ddf28.csv'),
 PosixPath('data/pd2dd/ddf19.csv'),
 PosixPath('data/pd2dd/ddf23.csv'),
 PosixPath('data/pd2dd/ddf02.csv'),
 PosixPath('data/pd2dd/ddf24.csv'),
 PosixPath('data/pd2dd/ddf15.csv'),
 PosixPath('data/pd2dd/ddf21.csv'),
 PosixPath('data/pd2dd/ddf13.csv'),
 PosixPath('data/pd2dd/ddf26.csv'),
 PosixPath('data/pd2dd/ddf17.csv'),
 PosixPath('data/pd2dd/ddf14.csv'),
 PosixPath('data/pd2dd/ddf01.csv'),
 PosixPath('data/pd2dd/ddf04.csv'),
 PosixPath('data/pd2dd/ddf08.csv'),
 PosixPath('data/pd2dd/ddf00.csv'),
 PosixPath('data/pd2dd/ddf11.csv'),
 PosixPath('data/pd2dd/ddf05.csv')]
```

To change the number of output files use repartition (https://docs.dask.org/en/latest/dataframe-api.html#dask.dataframe.DataFrame.repartition) which is an expensive operation.

Read files

For `pandas` it is possible to iterate and concat the files see answer from stack overflow (https://stackoverflow.com/questions/20906474/import-multiple-csv-files-into-pandas-and-concatenate-into-one-dataframe).

In [36]:
```
%%time
# Pandas
dir_path = Path(r'data/pd2dd')
concat_df = pd.concat([pd.read_csv(f) for f in list(dir_path.glob('*.csv'))])
len(concat_df)
```

```
CPU times: user 2.34 s, sys: 131 ms, total: 2.47 s
Wall time: 2.37 s
```

In [37]:
```
%%time
# Dask
_ddf = dd.read_csv('data/pd2dd/ddf*.csv')
len(_ddf)
```

```
CPU times: user 361 ms, sys: 4.78 ms, total: 366 ms
Wall time: 1.06 s
```

Consider using Persist

Since Dask is lazy - it may run the **entire** graph/DAG (again) even if it already run part of the calculation in a previous cell. Thus use persist (https://docs.dask.org/en/latest/dataframe-best-practices.html?highlight=parquet#persist-intelligently) to keep the results in memory

```
ddf = client.persist(ddf)
```

This is different from Pandas which once a variable was created it will keep all data in memory.
Additional information can be read in this stackoverflow issue (https://stackoverflow.com/questions/45941528/how-to-efficiently-send-a-large-numpy-array-to-the-cluster-with-dask-array/45941529#45941529) or see an exampel in this post (http://matthewrocklin.com/blog/work/2017/01/12/dask-dataframes)
This concept should also be used when running a code within a script (rather then a jupyter notebook) which incoperates loops within the code.

In [39]:
```
_ddf = dd.read_csv('data/pd2dd/ddf*.csv')
# do some filter
_ddf = client.persist(_ddf)
# do some computations
_ddf.head(2)
```

Out[39]:

	ID	name	x	y	times
0	960	Oliver	-0.745248	-0.198965	00:00:00
1	1000	Dan	0.184641	0.295430	00:00:01

Group By - custom aggregations

In addition to the groupby notebook example (https://github.com/dask/dask-examples/blob/master/dataframes/02-groupby.ipynb) - this is another example how to try to eliminate the use of `groupby.apply`
In this example we are grouping by columns into unique list.

In [41]:
```
# prepare pandas dataframe
pdf = pdf.assign(time=pd.to_datetime(pdf.index).time)
pdf['seconds'] = pdf.time.astype(str).str[-2:]
cols_for_demo =['name', 'ID','seconds']
pdf[cols_for_demo].head()
```

Out[41]:

	name	ID	seconds
0	Oliver	960	00
1	Dan	1000	00
2	Norbert	1020	00
3	Dan	1015	00
4	Edith	998	00

In [42]:
```
%%time
pdf_gb = pdf.groupby(pdf.name)
gp_col = ['ID', 'seconds']
list_ser_gb = [pdf_gb[att_col_gr].apply
               (lambda x: list(set(x.to_list())))
               for att_col_gr in gp_col]
df_edge_att = pdf_gb.size().to_frame(name="Weight")
for ser in list_ser_gb:
        df_edge_att = df_edge_att.join(ser.to_frame(), how='left')
```

```
CPU times: user 1.14 s, sys: 14.9 ms, total: 1.16 s
Wall time: 1.14 s
```

In [43]:
```
df_edge_att.head(2)
```

Out[43]:

	Weight	ID	seconds
name			
Alice	99839	[1024, 1025, 1026, 1027, 1028, 1029, 1030, 103...	[51, 21, 57, 30, 26, 41, 94, 45, 07, 54, 78, 7...
Bob	99407	[1024, 1025, 1026, 1027, 1028, 1029, 1030, 103...	[51, 21, 57, 30, 26, 41, 94, 45, 07, 54, 78, 7...

In any case sometimes using Pandas is more efficiante (assuming that you can load all the data into the RAM).
In this case Pandas is faster

In [44]:
```
def set_list_att(x: dd.Series):
        return list(set([item for item in x.values]))
ddf['seconds'] = ddf.times.astype(str).str[-2:]
ddf = client.persist(ddf)
ddf[cols_for_demo].head(2)
```

Out[44]:

	name	ID	seconds
0	Oliver	960	00
1	Dan	1000	01

In [48]:
```
%%time
# Dask option1 using apply
# notice the meta argument in the apply function
df_gb = ddf.groupby(ddf.name)
gp_col = ['ID', 'seconds']
list_ser_gb = [df_gb[att_col_gr].apply(set_list_att
                ,meta=pd.Series(dtype='object', name=f'{att_col_gr}_att'))
               for att_col_gr in gp_col]
df_edge_att = df_gb.size().to_frame(name="Weight")
for ser in list_ser_gb:
    df_edge_att = df_edge_att.join(ser.to_frame(), how='left')
df_edge_att.head(2)
```

```
CPU times: user 2.22 s, sys: 102 ms, total: 2.32 s
Wall time: 6.24 s
```

Using [dask custom aggregation (https://docs.dask.org/en/latest/dataframe-api.html?highlight=dropna#dask.dataframe.groupby.Aggregation)](https://docs.dask.org/en/latest/dataframe-api.html?highlight=dropna#dask.dataframe.groupby.Aggregation) is consideribly better

In [46]:
```
# Dask
import itertools
custom_agg = dd.Aggregation(
    'custom_agg',
    lambda s: s.apply(set),
    lambda s: s.apply(lambda chunks: list(set(itertools.chain.from_iterable(ch
unks)))),)
```

In [47]:
```
%%time
# Dask option1 using apply
df_gb = ddf.groupby(ddf.name)
gp_col = ['ID', 'seconds']
list_ser_gb = [df_gb[att_col_gr].agg(custom_agg) for att_col_gr in gp_col]
df_edge_att = df_gb.size().to_frame(name="Weight")
for ser in list_ser_gb:
        df_edge_att = df_edge_att.join(ser.to_frame(), how='left')
df_edge_att.head(2)
```

```
CPU times: user 385 ms, sys: 14.2 ms, total: 399 ms
Wall time: 1.3 s
```

In [ ]:
```
df_edge_att.head()
```

Debugging (https://docs.dask.org/en/latest/debugging.html)

Debugging may be challenging...

1. Run code without client
2. Verify integraty of DAG
3. Use Dashboard profiler

Corrupted DAG

In [49]:
```
# reset dataframe
ddf = dask.datasets.timeseries()
ddf.head(1)
```

Out[49]:

timestamp	id	name	x	y
2000-01-01	1033	Michael	0.974223	-0.087096

In [50]:
```
def func_dist2(df, coor_x, coor_y):
    dist =  np.sqrt ( (df[coor_x] - df[coor_x].shift())^2
                    +  (df[coor_y] - df[coor_y].shift())^2 )
    return dist
ddf['col'] = ddf.map_partitions(func_dist2, coor_x='x', coor_y='y'
                                , meta=('float'))
```

In [51]:
```
# returns an error because of ^2
ddf.head()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-51-f15aff7b9632> in <module>
      1 # returns an error because of ^2
----> 2 ddf.head()

~/.local/share/virtualenvs/dask_pyconil2019-9doxB0Ra/lib/python3.7/site-packag
es/dask/dataframe/core.py in head(self, n, npartitions, compute)
    898
    899         if compute:
--> 900             result = result.compute()
    901         return result
    902

~/.local/share/virtualenvs/dask_pyconil2019-9doxB0Ra/lib/python3.7/site-packag
es/dask/base.py in compute(self, **kwargs)
    154         dask.base.compute
    155         """
--> 156         (result,) = compute(self, traverse=False, **kwargs)
    157         return result
    158

~/.local/share/virtualenvs/dask_pyconil2019-9doxB0Ra/lib/python3.7/site-packag
es/dask/base.py in compute(*args, **kwargs)
    396     keys = [x.__dask_keys__() for x in collections]
    397     postcomputes = [x.__dask_postcompute__() for x in collections]
--> 398     results = schedule(dsk, keys, **kwargs)
    399     return repack([f(r, *a) for r, (f, a) in zip(results, postcomputes
)])
    400

~/.local/share/virtualenvs/dask_pyconil2019-9doxB0Ra/lib/python3.7/site-packag
es/distributed/client.py in get(self, dsk, keys, restrictions, loose_restricti
ons, resources, sync, asynchronous, direct, retries, priority, fifo_timeout, a
ctors, **kwargs)
   2566                     should_rejoin = False
   2567             try:
-> 2568                 results = self.gather(packed, asynchronous=asynchronou
s, direct=direct)
   2569             finally:
   2570                 for f in futures.values():

~/.local/share/virtualenvs/dask_pyconil2019-9doxB0Ra/lib/python3.7/site-packag
es/distributed/client.py in gather(self, futures, errors, maxsize, direct, asy
nchronous)
   1820                 direct=direct,
   1821                 local_worker=local_worker,
-> 1822                 asynchronous=asynchronous,
   1823             )
   1824

~/.local/share/virtualenvs/dask_pyconil2019-9doxB0Ra/lib/python3.7/site-packag
es/distributed/client.py in sync(self, func, *args, **kwargs)
    751             return future
    752         else:
--> 753             return sync(self.loop, func, *args, **kwargs)
    754
    755     def __repr__(self):

~/.local/share/virtualenvs/dask_pyconil2019-9doxB0Ra/lib/python3.7/site-packag
es/distributed/utils.py in sync(loop, func, *args, **kwargs)
    329             e.wait(10)
    330     if error[0]:
--> 331         six.reraise(*error[0])
    332     else:
    333         return result[0]

~/.local/share/virtualenvs/dask_pyconil2019-9doxB0Ra/lib/python3.7/site-packag
```

- Even if the function is currected the DAG is corrupted

In [52]:
```
# Still results with an error
def func_dist2(df, coor_x, coor_y):
    dist =  np.sqrt ( (df[coor_x] - df[coor_x].shift())**2
                    +  (df[coor_y] - df[coor_y].shift())**2 )
    return dist
ddf['col'] = ddf.map_partitions(func_dist2, coor_x='x', coor_y='y'
                                , meta=('float'))
ddf.head(2)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-52-1134deb0a399> in <module>
      6 ddf['col'] = ddf.map_partitions(func_dist2, coor_x='x', coor_y='y'
      7                                 , meta=('float'))
----> 8 ddf.head(2)

~/.local/share/virtualenvs/dask_pyconil2019-9doxB0Ra/lib/python3.7/site-packag
es/dask/dataframe/core.py in head(self, n, npartitions, compute)
    898
    899         if compute:
--> 900             result = result.compute()
    901         return result
    902

~/.local/share/virtualenvs/dask_pyconil2019-9doxB0Ra/lib/python3.7/site-packag
es/dask/base.py in compute(self, **kwargs)
    154         dask.base.compute
    155         """
--> 156         (result,) = compute(self, traverse=False, **kwargs)
    157         return result
    158

~/.local/share/virtualenvs/dask_pyconil2019-9doxB0Ra/lib/python3.7/site-packag
es/dask/base.py in compute(*args, **kwargs)
    396     keys = [x.__dask_keys__() for x in collections]
    397     postcomputes = [x.__dask_postcompute__() for x in collections]
--> 398     results = schedule(dsk, keys, **kwargs)
    399     return repack([f(r, *a) for r, (f, a) in zip(results, postcomputes
)])
    400

~/.local/share/virtualenvs/dask_pyconil2019-9doxB0Ra/lib/python3.7/site-packag
es/distributed/client.py in get(self, dsk, keys, restrictions, loose_restricti
ons, resources, sync, asynchronous, direct, retries, priority, fifo_timeout, a
ctors, **kwargs)
   2566                     should_rejoin = False
   2567             try:
-> 2568                 results = self.gather(packed, asynchronous=asynchronou
s, direct=direct)
   2569             finally:
   2570                 for f in futures.values():

~/.local/share/virtualenvs/dask_pyconil2019-9doxB0Ra/lib/python3.7/site-packag
es/distributed/client.py in gather(self, futures, errors, maxsize, direct, asy
nchronous)
   1820                 direct=direct,
   1821                 local_worker=local_worker,
-> 1822                 asynchronous=asynchronous,
   1823             )
   1824

~/.local/share/virtualenvs/dask_pyconil2019-9doxB0Ra/lib/python3.7/site-packag
es/distributed/client.py in sync(self, func, *args, **kwargs)
    751             return future
    752         else:
--> 753             return sync(self.loop, func, *args, **kwargs)
    754
    755     def __repr__(self):

~/.local/share/virtualenvs/dask_pyconil2019-9doxB0Ra/lib/python3.7/site-packag
es/distributed/utils.py in sync(loop, func, *args, **kwargs)
    329             e.wait(10)
    330     if error[0]:
--> 331         six.reraise(*error[0])
    332     else:
    333         return result[0]
```

Need to reset the dataframe

In [53]:
```
ddf = dask.datasets.timeseries()
def func_dist2(df, coor_x, coor_y):
    dist =  np.sqrt ( (df[coor_x] - df[coor_x].shift())**2
                    +  (df[coor_y] - df[coor_y].shift())**2 )
    return dist
ddf['col'] = ddf.map_partitions(func_dist2, coor_x='x', coor_y='y'
                                , meta=('float'))
ddf.head(2)
```

Out[53]:

timestamp	id	name	x	y	col
2000-01-01 00:00:00	1026	George	-0.813055	-0.263826	NaN
2000-01-01 00:00:01	990	Wendy	-0.137142	-0.495129	0.714395

Summary

1. `Dask` is lazy but efficient (parallel computing)
2. Flexable environments - from single laptop to thousands of nodes (`client`)
3. Usefull when comming from a `Pandas` (especially with comparison to `pyspark`)
4. Bonus - dashboard
5. But beware of:
   - missing functionalities from `Pandas` API
   - currupted DAGs

js.berry@gmail.com

https://github.com/sephib/dask_pyconil2019 (https://github.com/sephib/dask_pyconil2019)