PARALLEL COMPUTING WITH DASK

Understanding Computer Storage & Big Data

Dhavide Aruliah
Director of Training, Anaconda

What is "Big Data"?

"Data > one machine"

Storage Units: Bytes, Kilobytes, Megabytes, ...

watt	W	
Kilowatt	KW	10^3 W
Megawatt	MW	10^6 W
Gigawatt	GW	10^9 W
Terawatt	TW	10^{12} W

- Conventional units: factors of 1000
  - Kilo $\to$ Mega $\to$ Giga $\to$ Tera $\to$ $\cdots$

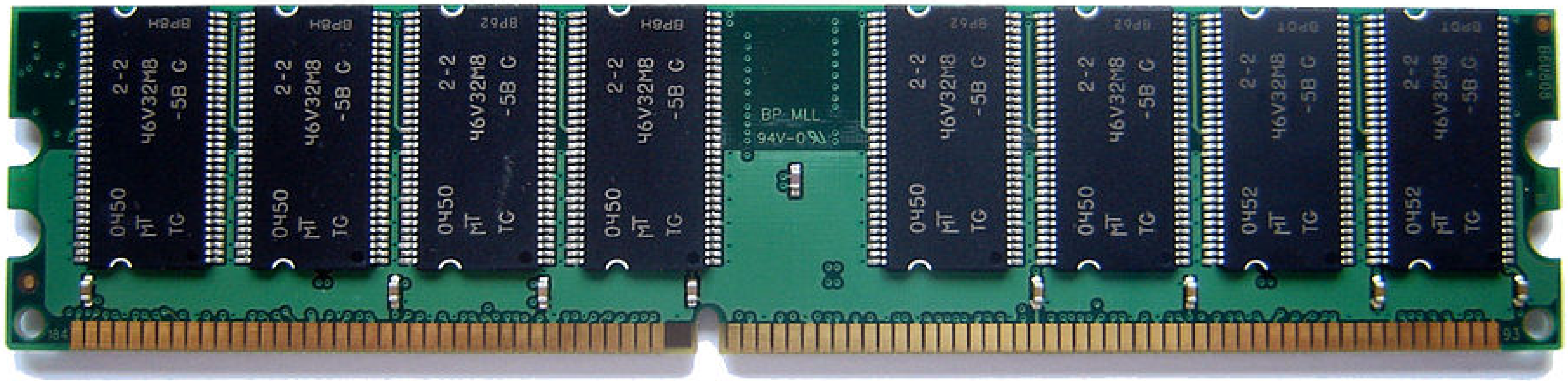
Byte	B	2^3 bits
Kilobyte	KB	2^{10} Bytes
Megabyte	MB	2^{20} Bytes
Gigabyte	GB	2^{30} Bytes
Terabyte	TB	2^{40} Bytes

- Binary computers: base 2:
  - Binary digit (bit)
  - Byte: 2^3 bits = 8 bits
  - $10^3 = 1000 \mapsto 2^{10} = 1024$

Hard disks

- Hard storage: hard disks (permanent, big, **slow**)

Random Access Memory (RAM)



- Soft storage: RAM (temporary, small, **fast**)

Time Scales of Storage Technologies

Storage medium	Access time
RAM	120 ns
Solid-state disk	50-150 µs
Rotational disk	1-10 ms
Internet (SF to NY)	40 ms

Storage medium	Rescaled
RAM	1 s
Solid-state disk	7-21 min
Rotational disk	2.5 hr - 1 day
Internet (SF to NY)	3.9 days

Big Data in Practical Terms

- RAM: **fast** (ns-μs)
- Hard disk: **slow** (μs-ms)
- I/O (input/output) is **punitive!**

Querying Python interpreter's Memory Usage

```
In [1]: import psutil, os

In [2]: def memory_footprint():
   ...:     '''Returns memory (in MB) being used by Python process'''
   ...:     mem = psutil.Process(os.getpid()).memory_info().rss
   ...:     return (mem / 1024 ** 2)
```

Allocating Memory for an Array

```
In [3]: import numpy as np

In [4]: before = memory_footprint()

In [5]: N = (1024 ** 2) // 8 # Number of floats that fill 1 MB

In [6]: x = np.random.randn(50*N) # Random array filling 50 MB

In [7]: after = memory_footprint()

In [8]: print('Memory before: {} MB'.format(before))
Memory before: 45.68359375 MB

In [9]: print('Memory after: {} MB'.format(after))
Memory after: 95.765625 MB
```

Allocating Memory for a Computation

```
In [10]: before = memory_footprint()

In [11]: x ** 2 # Computes, but doesn't bind result to a variable
Out[11]:
array([ 0.16344891,  0.05993282,  0.53595334, ...,  0.50537523,
       0.48967157,  0.06905984])

In [12]: after = memory_footprint()

In [13]: print('Extra memory obtained: {} MB'.format(after - before))
Extra memory obtained: 50.34375 MB
```

Querying Array Memory Usage

```
In [14]: x.nbytes  # memory footprint in bytes (B)
Out[14]: 52428800

In [15]: x.nbytes // (1024**2) # memory footprint in megabytes (MB)
Out[15]: 50
```

Querying DataFrame Memory Usage

```
In [16]: df = pd.DataFrame(x)

In [17]: df.memory_usage(index=False)
Out[17]:
0    52428800
dtype: int64

In [18]: df.memory_usage(index=False) // (1024**2)
Out[18]:
0    50
dtype: int64
```

PARALLEL COMPUTING WITH DASK

Let's practice!

PARALLEL COMPUTING WITH DASK

Thinking about Data in Chunks

Dhavide Aruliah
Director of Training, Anaconda

Using pd.read_csv with chunksize

```
In [3]: filename = 'NYC_taxi_2013_01.csv'

In [4]: for chunk in pd.read_csv(filename, chunksize=50000):
   ...:     print('type: %s shape %s' %
   ...:                (type(chunk), chunk.shape))
type: <class 'pandas.core.frame.DataFrame'> shape (50000, 14)
type: <class 'pandas.core.frame.DataFrame'> shape (50000, 14)
type: <class 'pandas.core.frame.DataFrame'> shape (50000, 14)
type: <class 'pandas.core.frame.DataFrame'> shape (49999, 14)
```

Examining a chunk

```
In [5]: chunk.shape
Out[5]: (49999, 14)

In [6]: chunk.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 49999 entries, 150000 to 199998
Data columns (total 14 columns):
medallion                 49999 non-null object
hack_license              49999 non-null object
vendor_id                 49999 non-null object
rate_code                 49999 non-null int64
store_and_fwd_flag        162 non-null object
pickup_datetime           49999 non-null object
dropoff_datetime          49999 non-null object
passenger_count           49999 non-null int64
trip_time_in_secs         49999 non-null int64
trip_distance             49999 non-null float64
pickup_longitude          49999 non-null float64
pickup_latitude           49999 non-null float64
dropoff_longitude         49999 non-null float64
dropoff_latitude          49999 non-null float64
dtypes: float64(5), int64(3), object(6)
memory usage: 5.3+ MB
```

Filtering a chunk

```
In [7]: is_long_trip = (chunk.trip_time_in_secs > 1200)

In [8]: chunk.loc[is_long_trip].shape
Out[8]: (5565, 14)
```

	passenger_count	trip_time_in_secs	trip_distance
167	1	300	2.1
168	3	2100	13.51
169	1	420	1.56
170	3	120	0.67
171	4	960	3.34
172	2	1140	4.13
173	5	300	2.19
174	1	1620	10.1
175	1	120	0.55
176	1	1440	10.63
177	1	120	0.47
178	1	1320	6.82
179	1	1500	5.32
180	1	420	1.71
181	3	960	4.72
182	6	1020	4.77
183	1	600	1.73
184	1	1020	7.29
185	3	1260	11.17

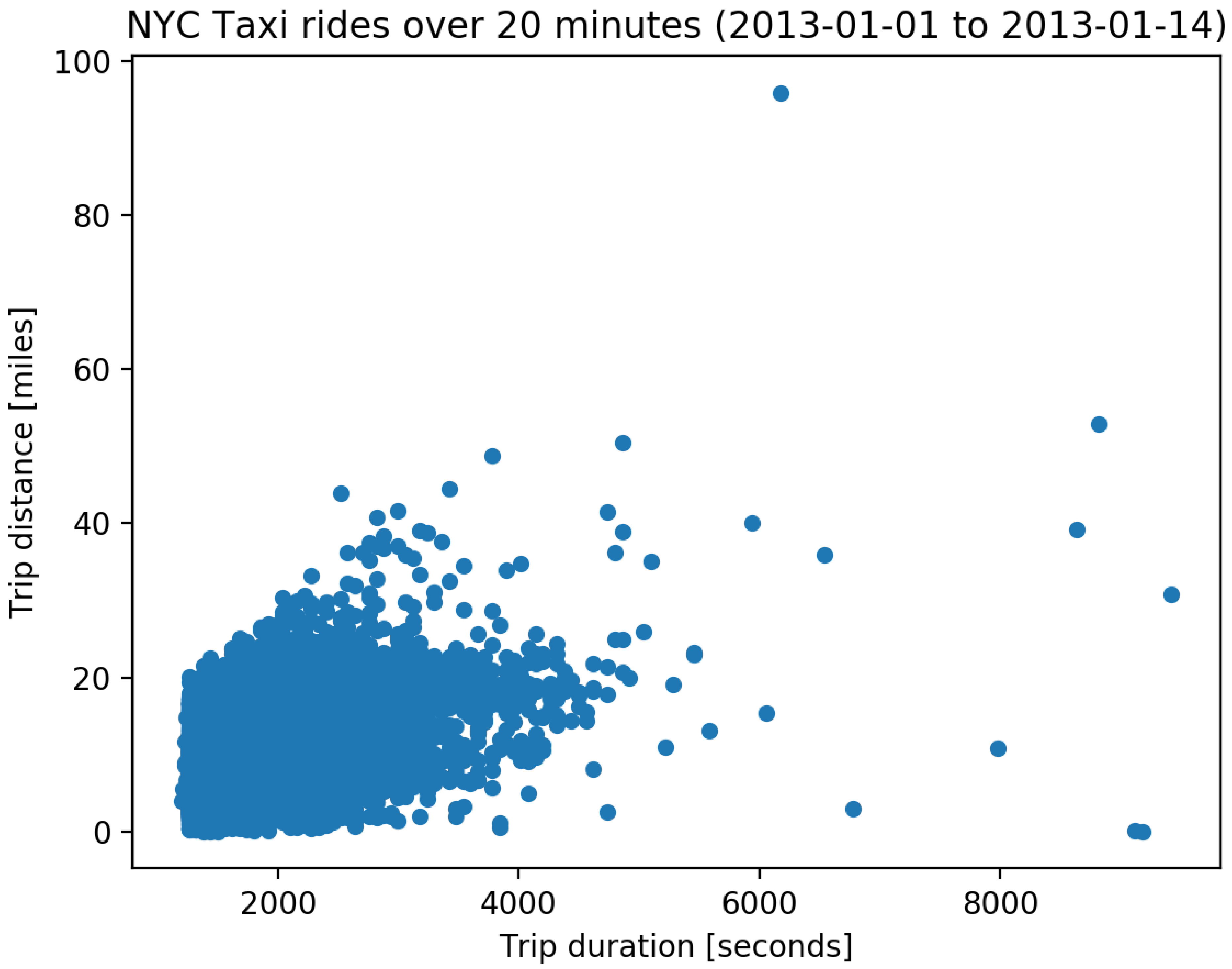
	passenger_count	trip_time_in_secs	trip_distance
168	3	2100	13.51
174	1	1620	10.1
176	1	1440	10.63
178	1	1320	6.82
179	1	1500	5.32
185	3	1260	11.17

Chunking & filtering together

```
In [9]: def filter_is_long_trip(data):
   ...:   "Returns DataFrame filtering trips longer than 20 minutes"
   ...:   is_long_trip = (data.trip_time_in_secs > 1200)
   ...:   return data.loc[is_long_trip]

In [10]: chunks = []

In [11]: for chunk in pd.read_csv(filename, chunksize=1000):
    ...:     chunks.append(filter_is_long_trip(chunk))

In [12]: chunks = [filter_is_long_trip(chunk)
    ...:           for chunk in pd.read_csv(filename,
    ...:           chunksize=1000) ]
```

Using pd.concat()

```
In [13]: len(chunks)
Out[13]: 200

In [14]: lengths = [len(chunk) for chunk in chunks]

In [15]: lengths[-5:]    # each has ~100 rows
Out[15]: [115, 147, 137, 109, 119]

In [16]: long_trips_df = pd.concat(chunks)

In [17]: long_trips_df.shape
Out[17]: (21661, 14)
```


NYC Taxi rides over 20 minutes (2013-01-01 to 2013-01-14)
Trip distance [miles]
Trip duration [seconds]
0
20
40
60
80
100
2000
4000
6000
8000

Plotting the filtered results

```
In [18]: import matplotlib.pyplot as plt

In [19]: long_trips_df.plot.scatter(x='trip_time_in_secs',
    ...:                            y='trip_distance');

In [20]: plt.xlabel('Trip duration [seconds]');

In [21]: plt.ylabel('Trip distance [miles]');

In [22]: plt.title('NYC Taxi rides over 20 minutes (2013-01-01 to 2013-01-14)');

In [23]: plt.show();
```

PARALLEL COMPUTING WITH DASK

Let's practice!

PARALLEL COMPUTING WITH DASK

Managing Data with Generators

Dhavide Aruliah
Director of Training, Anaconda

Filtering in a List Comprehension

```
In [1]: import pandas as pd

In [2]: filename = 'NYC_taxi_2013_01.csv'

In [3]: def filter_is_long_trip(data):
   ...:     "Returns DataFrame filtering trips longer than 20 minutes"
   ...:     is_long_trip = (data.trip_time_in_secs > 1200)
   ...:     return data.loc[is_long_trip]

In [4]: chunks = [filter_is_long_trip(chunk)
   ...:               for chunk in pd.read_csv(filename, chunksize=1000)]
```

Filtering & Summing with Generators

```
In [5]: chunks = (filter_is_long_trip(chunk)
   ...:                for chunk in pd.read_csv(filename, chunksize=1000))

In [6]: distances = (chunk['trip_distance'].sum() for chunk in chunks)

In [7]: sum(distances)
Out[7]: 230909.56000000003
```

Examining Consumed Generators

```
In [8]: distances
Out[8]: <generator object <genexpr> at 0x10766f9e8>

In [9]: next(distances)
StopIteration                             Traceback (most recent call last)
<ipython-input-10-9995a5373b05> in <module>()
```

Reading Many Files

```
In [10]: template = 'yellow_tripdata_2015-{:02d}.csv'

In [11]: filenames = (template.format(k) for k in range(1,13)) # generator

In [12]: for fname in filenames:
    ...:     print(fname) # Examine contents
yellow_tripdata_2015-01.csv
yellow_tripdata_2015-02.csv
yellow_tripdata_2015-03.csv
yellow_tripdata_2015-04.csv
yellow_tripdata_2015-05.csv
yellow_tripdata_2015-06.csv
yellow_tripdata_2015-07.csv
yellow_tripdata_2015-08.csv
yellow_tripdata_2015-09.csv
yellow_tripdata_2015-10.csv
yellow_tripdata_2015-11.csv
yellow_tripdata_2015-12.csv
```

Examining a Sample DataFrame

```
In [16]: df = pd.read_csv('yellow_tripdata_2015-12.csv', parse_dates=[1, 2])

In [17]: df.info()  # columns deleted from output
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 71634 entries, 0 to 71633
Data columns (total 19 columns):
VendorID                 71634 non-null int64
tpep_pickup_datetime     71634 non-null datetime64[ns]
tpep_dropoff_datetime    71634 non-null datetime64[ns]
passenger_count          71634 non-null int64
...
...
dtypes: datetime64[ns](2), float64(12), int64(4), object(1)
memory usage: 10.4+ MB

In [18]: def count_long_trips(df):
    ...:     df['duration'] = (df.tpep_dropoff_datetime -
    ...:                       df.tpep_pickup_datetime).dt.seconds
    ...:     is_long_trip = df.duration > 1200
    ...:     result_dict = {'n_long':[sum(is_long_trip)],
    ...:                    'n_total':[len(df)]}
    ...:     return pd.DataFrame(result_dict)
```

Aggregating with Generators

```
In [18]: def count_long_trips(df):
    ...:     df['duration'] = (df.tpep_dropoff_datetime -
    ...:                        df.tpep_pickup_datetime).dt.seconds
    ...:     is_long_trip = df.duration > 1200
    ...:     result_dict = {'n_long':[sum(is_long_trip)],
    ...:                    'n_total':[len(df)]}
    ...:     return pd.DataFrame(result_dict)

In [19]: filenames = [template.format(k) for k in range(1,13)] # listcomp

In [20]: dataframes = (pd.read_csv(fname, parse_dates=[1,2])
    ...:                   for fname in filenames)   # generator

In [21]: totals = (count_long_trips(df) for df in dataframes) # generator

In [22]: annual_totals = sum(totals) # Consumes generators
```

Computing the Fraction of Long Trips

```
In [23]: print(annual_totals)
   n_long  n_total
0  172617   851390

In [24]: fraction = annual_totals['n_long'] / annual_totals['n_total']

In [25]: print(fraction)
0    0.202747
dtype: float64
```

PARALLEL COMPUTING WITH DASK

Let's practice!

PARALLEL COMPUTING WITH DASK

Delaying Computation with Dask

Dhavide Aruliah
Director of Training, Anaconda

Composing functions

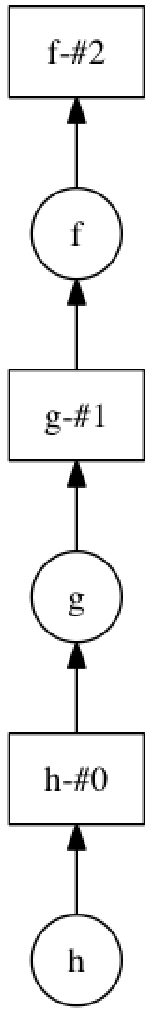
```
In [1]: from math import sqrt

In [2]: def f(z):
   ...:     return sqrt(z + 4)

In [3]: def g(y):
   ...:     return y - 3

In [4]: def h(x):
   ...:     return x ** 2

In [5]: x = 4

In [6]: y = h(x)

In [7]: z = g(y)

In [8]: w = f(z)

In [9]: print(w) # final result
4.123105625617661

In [10]: print(f(g(h(x)))) # equivalent to before
4.123105625617661
```

Deferring Computation with delayed

```
In [11]: from dask import delayed

In [12]: y = delayed(h)(x)

In [13]: z = delayed(g)(y)

In [14]: w = delayed(f)(z)

In [15]: print(w)
Delayed('f-5f9307e5-eb43-4304-877f-1df5c583c11c')

In [16]: type(w) # a dask Delayed object
Out[16]: dask.delayed.Delayed

In [17]: w.compute() # computation occurs *now*
Out[17]: 4.123105625617661
```

Visualizing a Task Graph

```
In [18]: w.visualize()
```



Renaming Decorated Functions

```
In [19]: f = delayed(f)

In [20]: g = delayed(g)

In [21]: h = delayed(h)

In [22]: w = f(g(h(4))

In [23]: type(w) # a dask Delayed object
Out[23]: dask.delayed.Delayed

In [24]: w.compute() # computation occurs *now*
Out[24]: 4.123105625617661
```

Using Decorator @-Notation

```
In [25]: def f(x):
    ...:     return sqrt(x + 4)

In [26]: f = delayed(f)

In [27]: @delayed # Equivalent to definition in above 2 cells
    ...: def f(x):
    ...:     return sqrt(x + 4)
```

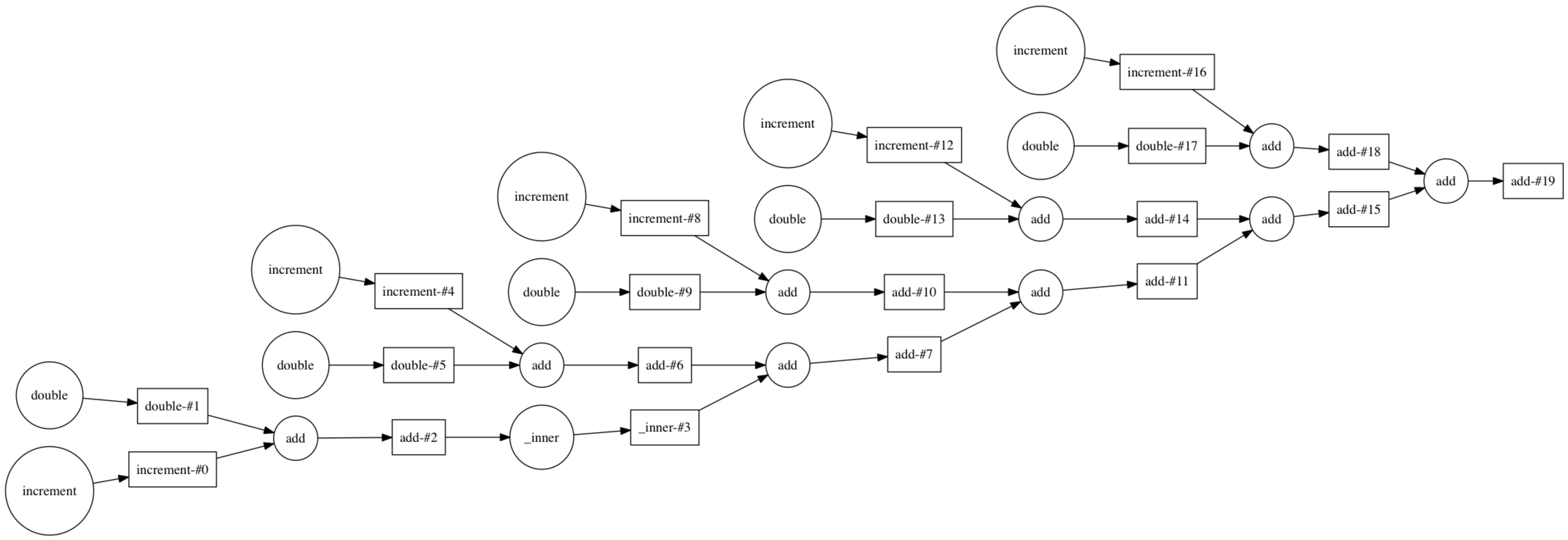
Deferring Computation with Loops

```
In [28]: @delayed
    ...: def increment(x):
    ...:     return x + 1

In [29]: @delayed
    ...: def double(x):
    ...:     return 2 * x

In [30]: @delayed
    ...: def add(x, y):
    ...:     return x + y
```

```
In [31]: data = [1, 2, 3, 4, 5]

In [32]: output = []

In [33]: for x in data:
    ...:     a = increment(x)
    ...:     b = double(x)
    ...:     c = add(a, b)
    ...:     output.append(c)

In [34]: total = sum(output)
```

```
In [35]: total
Out[35]: Delayed('add-c6803f9e890c95cec8e2e3dd3c62b384')

In [36]: output
Out[36]:
[Delayed('add-6a624d8b-8ddb-44fc-b0f0-0957064f54b7'),
 Delayed('add-9e779958-f3a0-48c7-a558-ce47fc9899f6'),
 Delayed('add-f3552c6f-b09d-4679-a770-a7372e2c278b'),
 Delayed('add-ce05d7e9-42ec-4249-9fd3-61989d9a9f7d'),
 Delayed('add-dd950ec2-c17d-4e62-a267-1dabe2101bc4')]

In [37]: total.visualize()
```

Visualizing the task graph



Aggregating with delayed Functions

```
In [35]: template = 'yellow_tripdata_2015-{:02d}.csv'

In [36]: filenames = [template.format(k) for k in range(1,13)]

In [37]: @delayed
    ...: def count_long_trips(df):
    ...:     df['duration'] = (df.tpep_dropoff_datetime -
    ...:                       df.tpep_pickup_datetime).dt.seconds
    ...:     is_long_trip = df.duration > 1200
    ...:     result_dict = {'n_long':[sum(is_long_trip)],
    ...:                    'n_total':[len(df)]}
    ...:     return pd.DataFrame(result_dict)

In [38]: @delayed
    ...: def read_file(fname):
    ...:     return pd.read_csv(fname, parse_dates=[1,2])
```

Computing Fraction of Long Trips with delayed Functions

```
In [39]: totals = [count_long_trips(read_file(fname))
    ...:                          for fname in filenames]

In [40]: annual_totals = sum(totals)

In [41]: annual_totals = annual_totals.compute()
Out[41]:
   n_long  n_total
0  172617   851390

In [42]: fraction = annual_totals['n_long'] /
    ...: annual_totals['n_total']

In [43]: print(fraction)
0    0.202747
dtype: float64
```

PARALLEL COMPUTING WITH DASK

Let's practice!