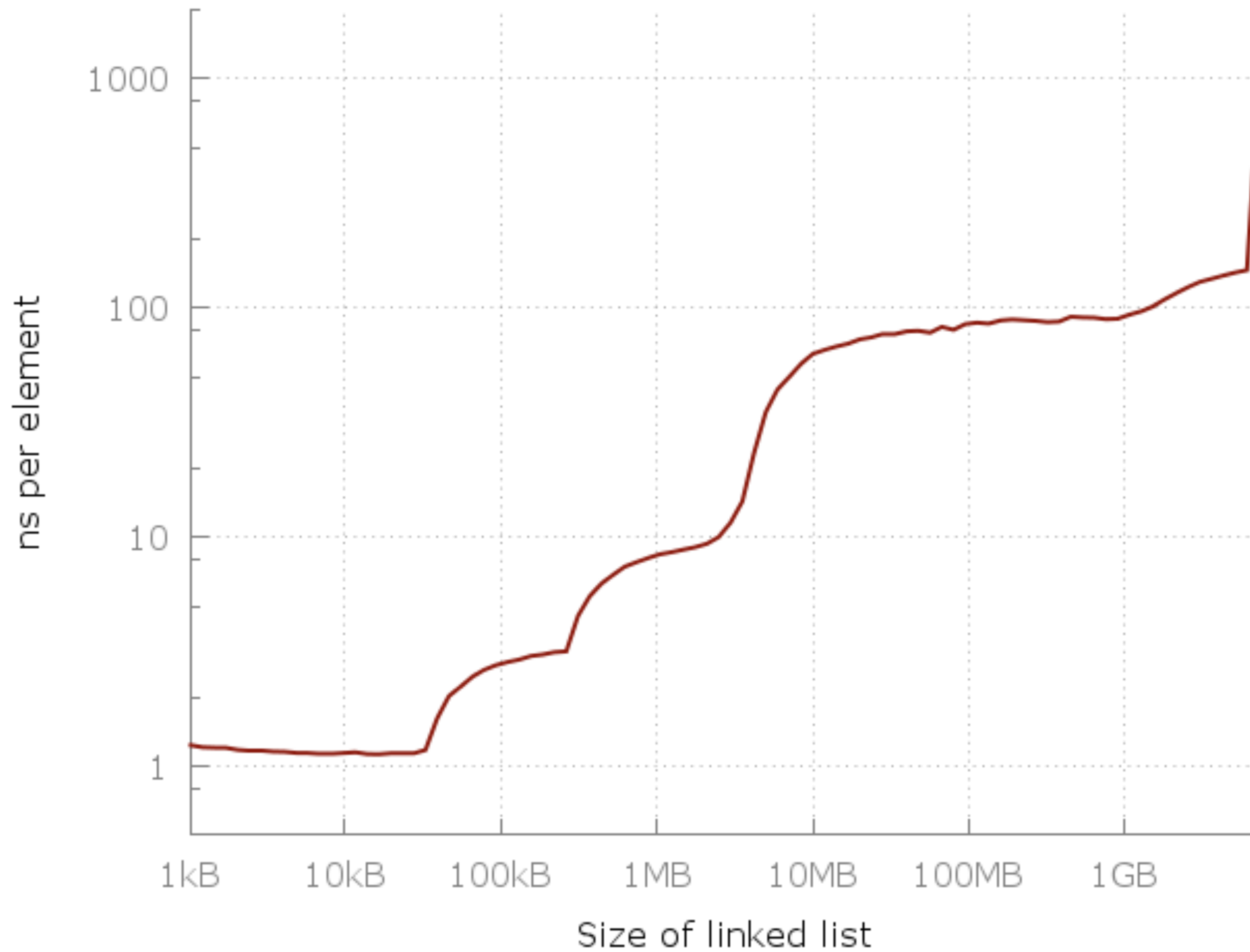
CS 696 Intro to Big Data: Tools and Methods
Spring Semester, 2021
Doc 10 Dask
Feb 18, 2021

Copyright ©, All rights reserved. 2021 SDSU & Roger Whitney, 5500 Campanile Drive, San Diego, CA 92182-7700 USA. OpenContent (http://www.opencontent.org/openpub/) license defines the copyright on this document.

How to Scale Python/Numpy/Panda Code?

Memory



How to Scale Python/Numpy/Panda Code?

Python & Parallelism

Modern hardware

- Multi-core
- Multi-processor

Python

- Can only run one thread at a time!
- GIL - Global Interpreter Lock

Protecting Memory from Read-Write conflicts

Protecting mutable memory from read-write conflicts

Fine-grain locks

- Java - synchronized methods create lock on a single object
- Complicated to implement

Global Lock

- Python
- Only one thread runs at a time

- Panda's and other libraries C code not thread safe

Dask - Python Library

Runs Python code in parallel

High level

- Dask Array
- Dask DataFrame
- Dask Bag
- Dask ML

Low level

- delayed
- futures

Works on

- Clusters
- Single machine

Dask Schedulers

Local Threads

```
import dask
dask.config.set(scheduler='threads')
```

- Single machine
- Uses thread pool
- GIL limits Python code to one thread
- Can use multiple threads for
  - Numpy arrays
  - Pandas Dataframes
  - Dask array, Dask DataFrame, Dask Delayed

Single Thread

```
import dask
dask.config.set(scheduler='synchronous')
```

- No parallelism
- Useful for debugging

Dask Schedulers

Dask Distributed (local)

```
from dask.distributed import Client
client = Client()
# or
client = Client(processes=False)
```

One machine

Provides access to asynchronous API - Futures

Provides diagnostic dashboard

Handles data locality better

Dask Distributed (Cluster)

For running on clusters

dask.delayed

Creates a proxy for function or object

Proxied functions can be scheduled to run later elsewhere

```
from time import sleep

def inc(x):
    sleep(1)
    return x + 1

def add(x, y):
    sleep(1)
    return x + y
```

```
%%time
x = inc(1)
y = inc(2)
z = add(x, y)
```

Wall time: 3.01 s

```
from dask import delayed

%%time
x = delayed(inc)(1)
y = delayed(inc)(2)
z = delayed(add)(x, y)
```

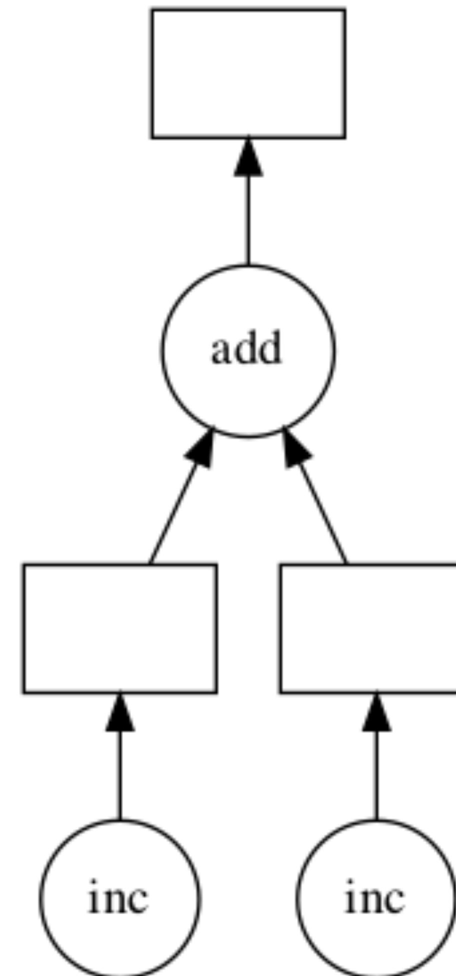
Wall time: 1.86 ms

dask.delayed

```
from dask import delayed

%%time
x = delayed(inc)(1)
y = delayed(inc)(2)
z = delayed(add)(x, y)

%%time
z.compute()
```

Wall time: 2.03 s

```
z.visualize()
```

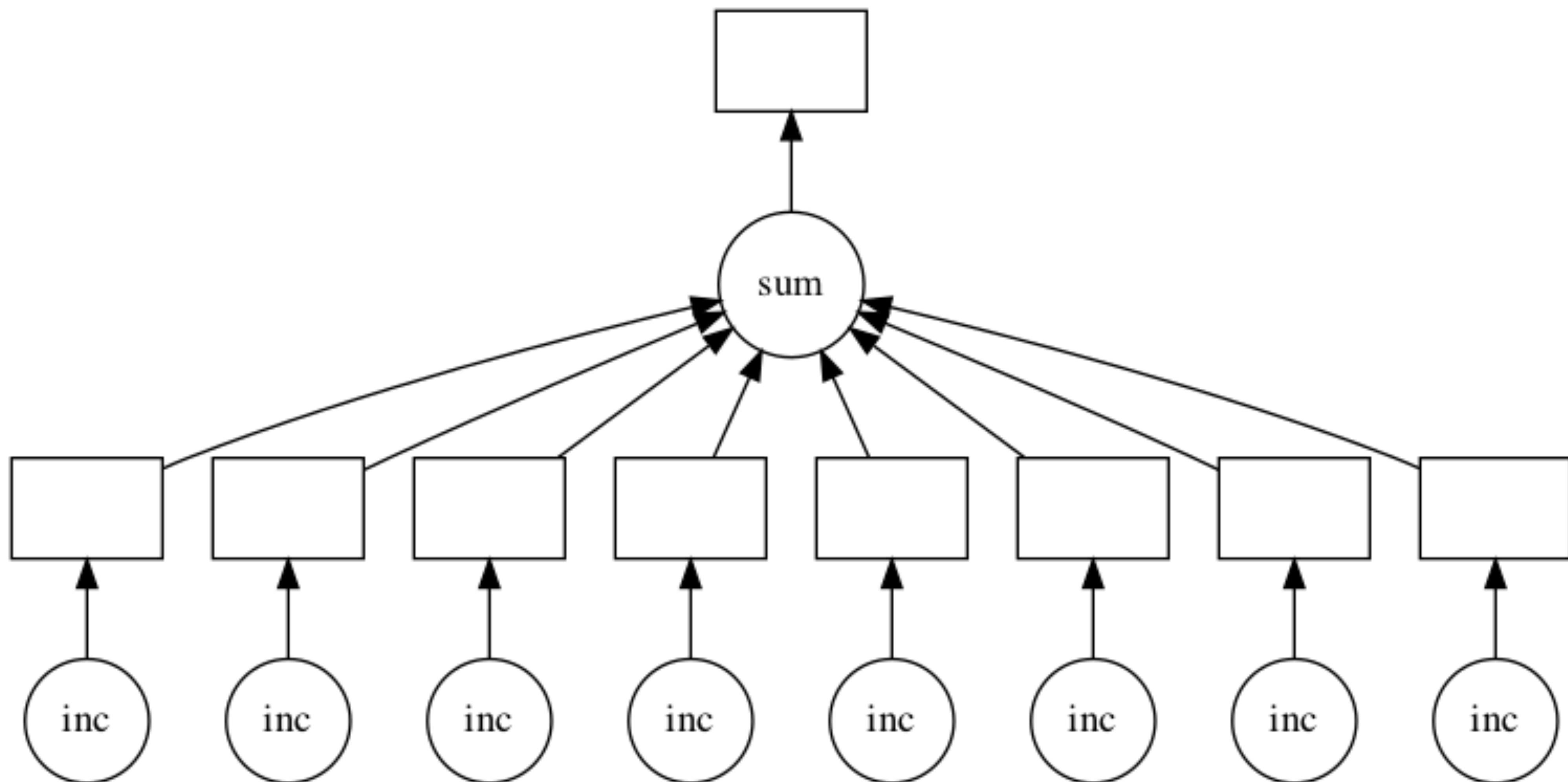


```
data = [1, 2, 3, 4, 5, 6, 7, 8]

results = []
for x in data:
    y = inc(x)
    results.append(y)

total = sum(results)
```

```
%%time

results = []

for x in data:
    y = delayed(inc)(x)
    results.append(y)

total = delayed(sum)(results)
result = total.compute()
```

Wall time: 1.03 s

```
results = []
data = [1, 2, 3, 4, 5, 6, 7, 8]
for x in data:
    y = delayed(inc)(x)
    results.append(y)

total = delayed(sum)(results)
total.visualize()
```



Some Delayed Best Practices

dask.delayed(f(x, y))

dask.delayed(f)(x, y)

Compute on lots of computation at once

    Do a lot in each compute call

Break up computations into many pieces

    A compute call should have many delayed

NYC Flight Data

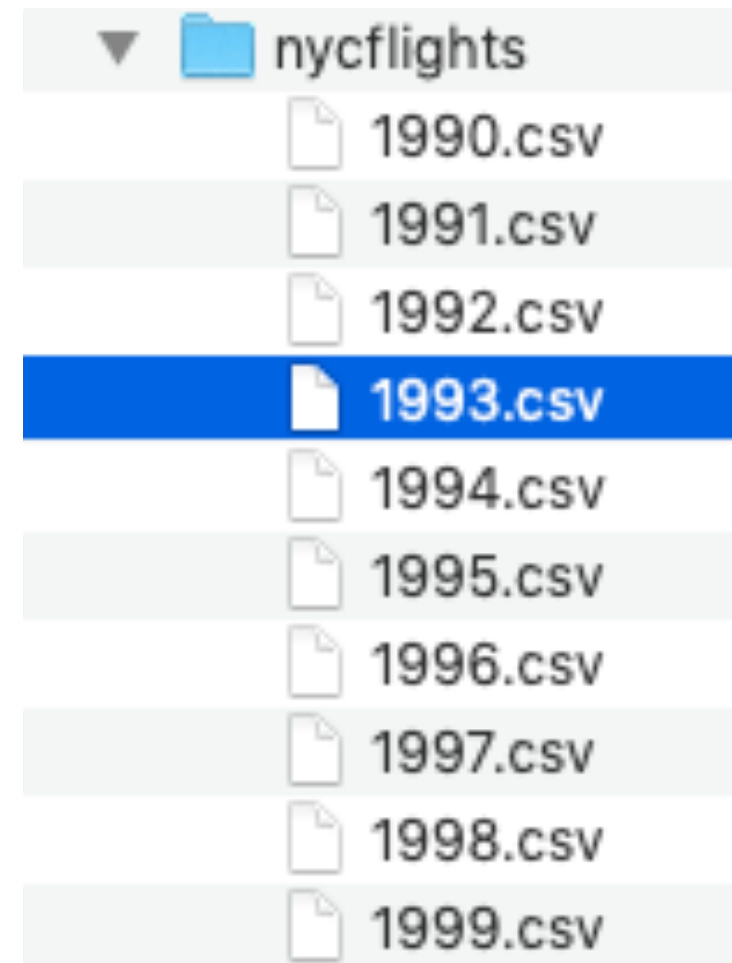
Data on all flights leaving all 3 NYC airports in 1990's

10 files
- 1 per year
- 1990, 1991, ... 1999

  Each file ~25MB

Relavent columns
- Origin - Which airport
  - EWR
  - LGA
  - JFK

- DepDelay
  - Minutes flight delayed in taking off



Compute Mean Departure Delay per Airport

```
import pandas as pd
from glob import glob

# list of filenames
filenames = sorted(glob(os.path.join('data', 'nycflights', '*.csv')))

sums = []
counts = []
for fn in filenames:
    df = pd.read_csv(fn)
    by_origin = df.groupby('Origin')
    total = by_origin.DepDelay.sum()
    count = by_origin.DepDelay.count()
    sums.append(total)
    counts.append(count)


total_delays = sum(sums)
n_flights = sum(counts)
mean = total_delays / n_flights
```

```
CPU times: user 4.93 s,
           sys: 817 ms,
           total: 5.74 s
Wall time: 5.41 s
```

Compute Mean Departure Delay - Dask Version

```
import pandas as pd
from glob import glob

# list of filenames
filenames = sorted(glob(os.path.join('data', 'nycflights', '*.csv')))

sums = []
counts = []
for fn in filenames:
   df = delayed(pd.read_csv)(fn)
   by_origin = df.groupby('Origin')
   total = by_origin.DepDelay.sum()
   count = by_origin.DepDelay.count()
   sums.append(total)
   counts.append(count)

sums, counts = compute(sums, counts)
total_delays = sum(sums)
n_flights = sum(counts)
mean = total_delays / n_flights
```

CPU times: user 150 ms,
sys: 26.2 ms,
total: 177 ms
Wall time: 1.82 s

Using 4 workers
2-5 workers - about same time

Delayed is Wrapper

```
df = delayed(pd.read_csv)(fn)

by_origin = df.groupby('Origin')

total = by_origin.DepDelay.sum()
```

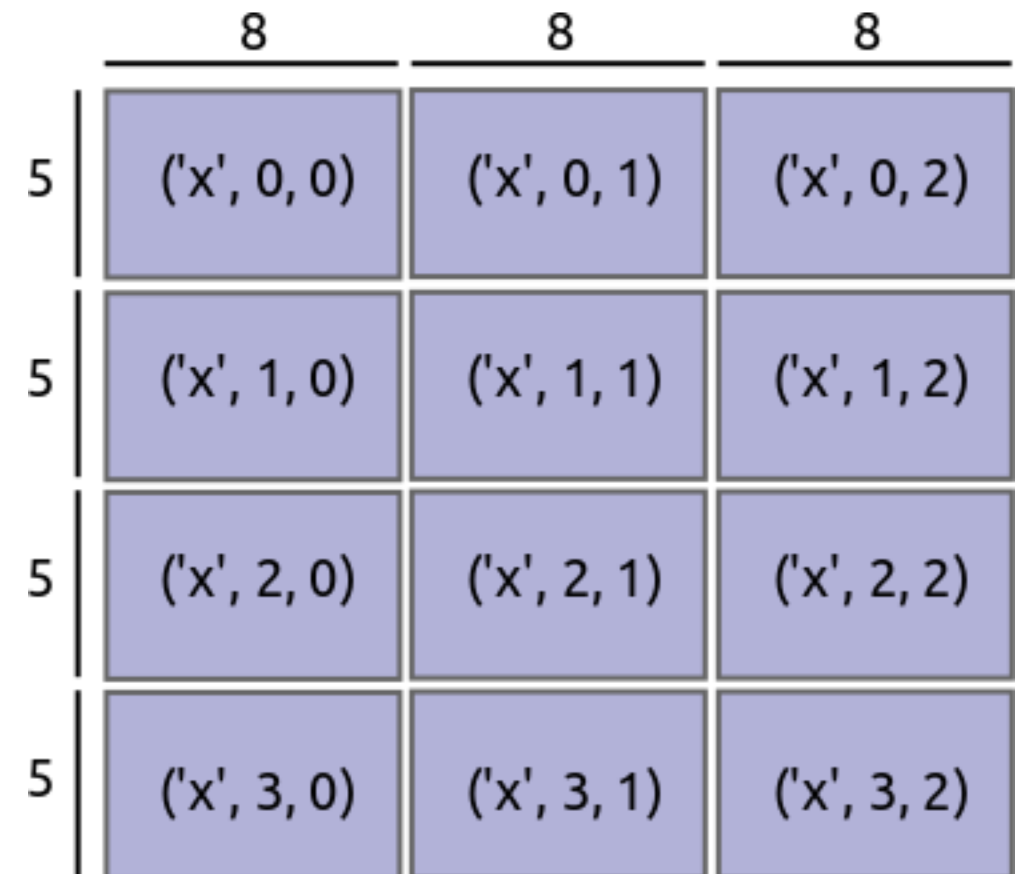
You can call most DF methods of delayed DF

They are also delayed

Dask Arrays

Parallel, larger-than-memory, n-dimensional array using blocked algorithms

- Divide large array into blocks
  - Blocks are numpy arrays
  - Blocks can be operated in parallel
  - Not all blocks need be in memory at once



```
import h5py
import os
f = h5py.File(os.path.join('data', 'random.hdf5'), mode='r')
dset = f['/x']
```

Reference to 4G dataset

```
import dask.array as da
x = da.from_array(dset, chunks=(1000000,))
```

	Array	Chunk
Bytes	4.00 GB	4.00 MB
Shape	(1000000000,)	(1000000,)
Count	1001 Tasks	1000 Chunks
Type	float32	numpy.ndarray

```
result = x.sum()
result.compute()
```

Dask DataFrames

Large parallel DataFrame composed of many smaller Pandas DataFrames

Uses subset of Panda DataFrame interface

Manipulating large datasets, even when those datasets don’t fit in memory

Accelerating long computations by using many cores

Distributed computing on large datasets

Reading the NYC Flight Data

```
import os
import dask
import dask.dataframe as dd
df = dd.read_csv(os.path.join('data', 'nycflights', '*.csv'),
                 parse_dates={'Date': [0, 1, 2]})

df
```

```
Dask DataFrame Structure:
Date      DayOfWeek   DepTime CRSDepTime ArrTime   etc.
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 Lazy Operations

dd.read_csv

Does not read entire file

Reads a sample to infer data type for each column

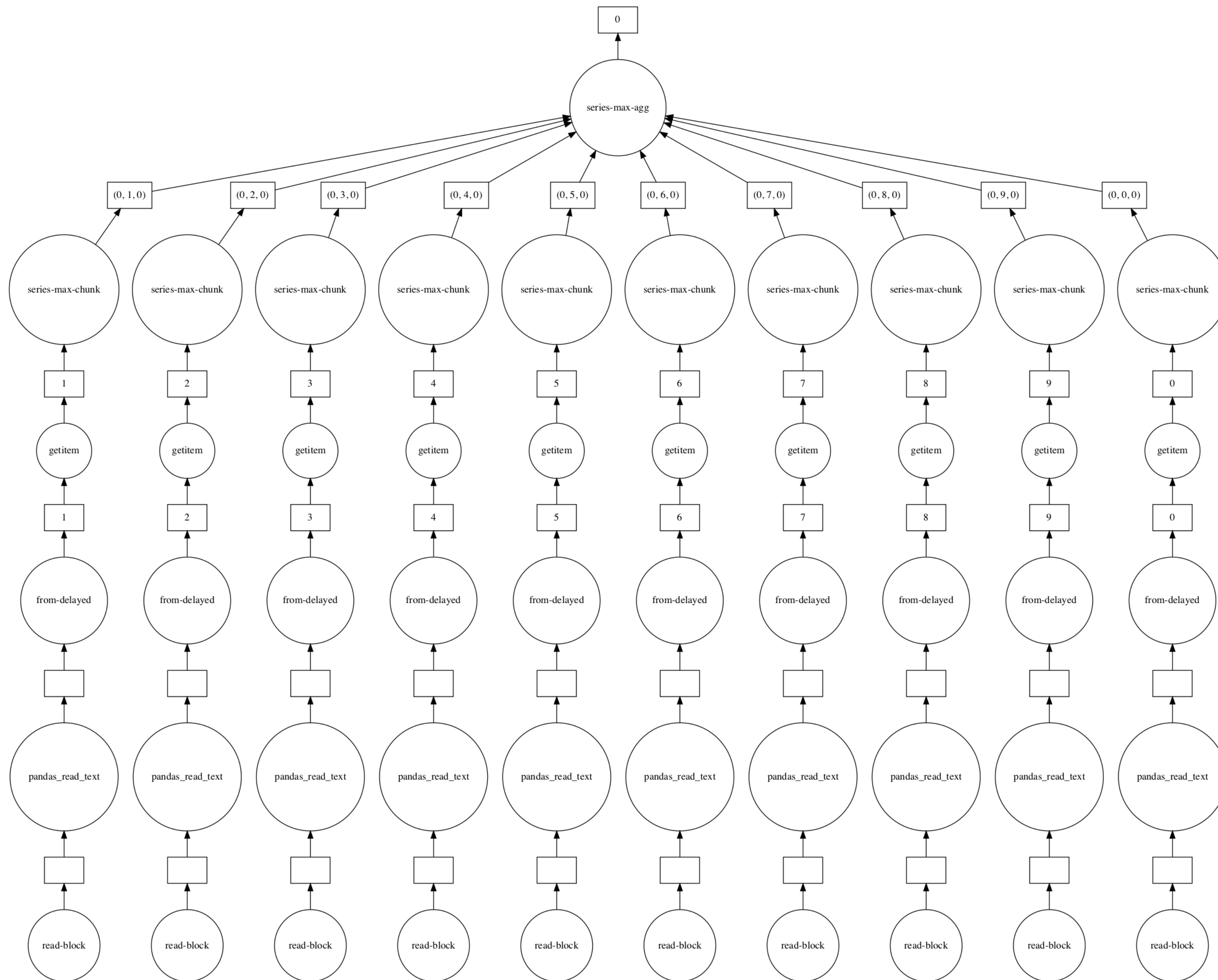
Does not always infer data types correctly

```
df = dd.read_csv(os.path.join('data', 'nycflights', '*.csv'),
                 parse_dates={'Date': [0, 1, 2]},
                 dtype={'TailNum': str,
                        'CRSElapsedTime': float,
                        'Cancelled': bool})
```

```
%time df.DepDelay.max().compute()
```

```
CPU times: user 265 ms,
           sys: 41.3 ms,
           total: 306 ms
Wall time: 2.65 s
```

df.DepDelay.max().visualize()

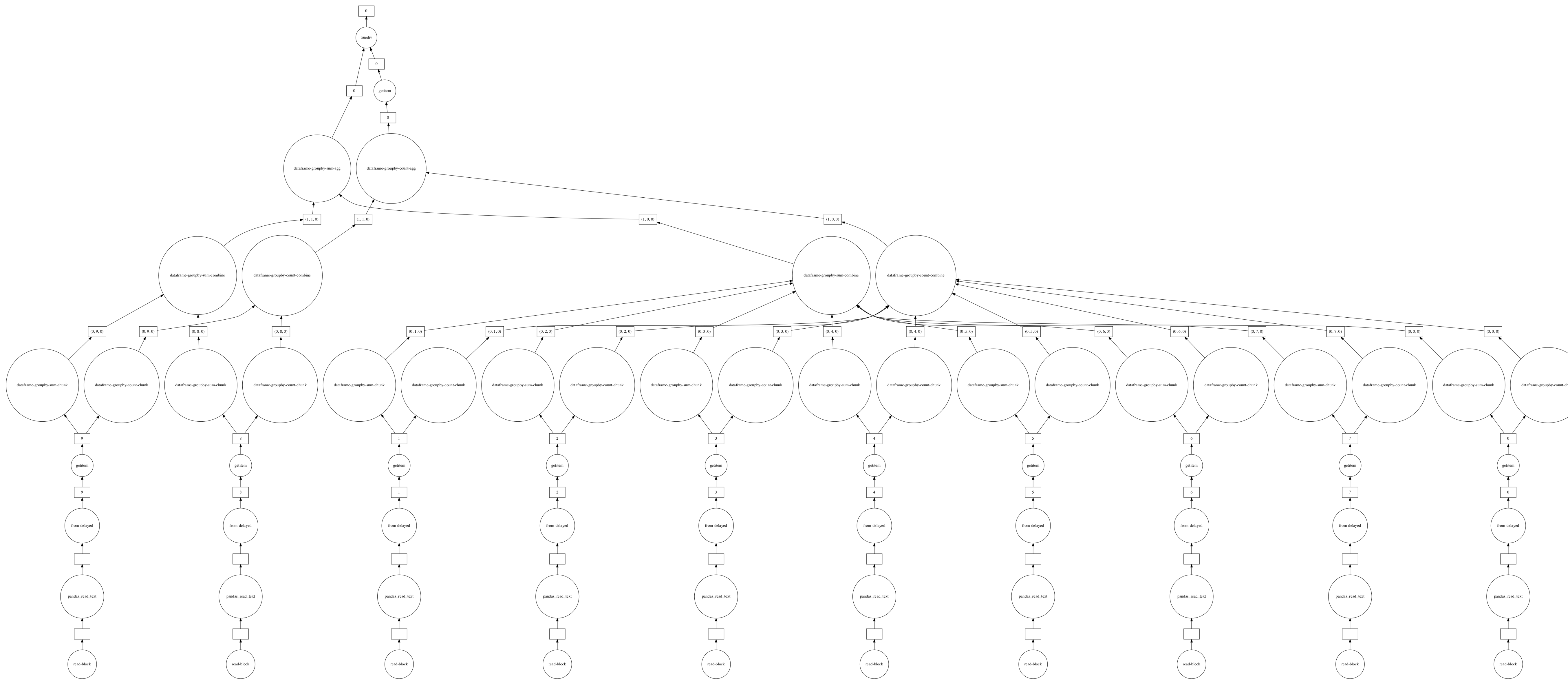


Computing mean Depart Delay

```
%time df[['Origin','DepDelay']].groupby('Origin').mean().compute()
```

```
CPU times: user 273 ms,
           sys: 29.3 ms,
           total: 303 ms
Wall time: 2.12 s
```

df[['Origin','DepDelay']].groupby('Origin').mean().visualize()



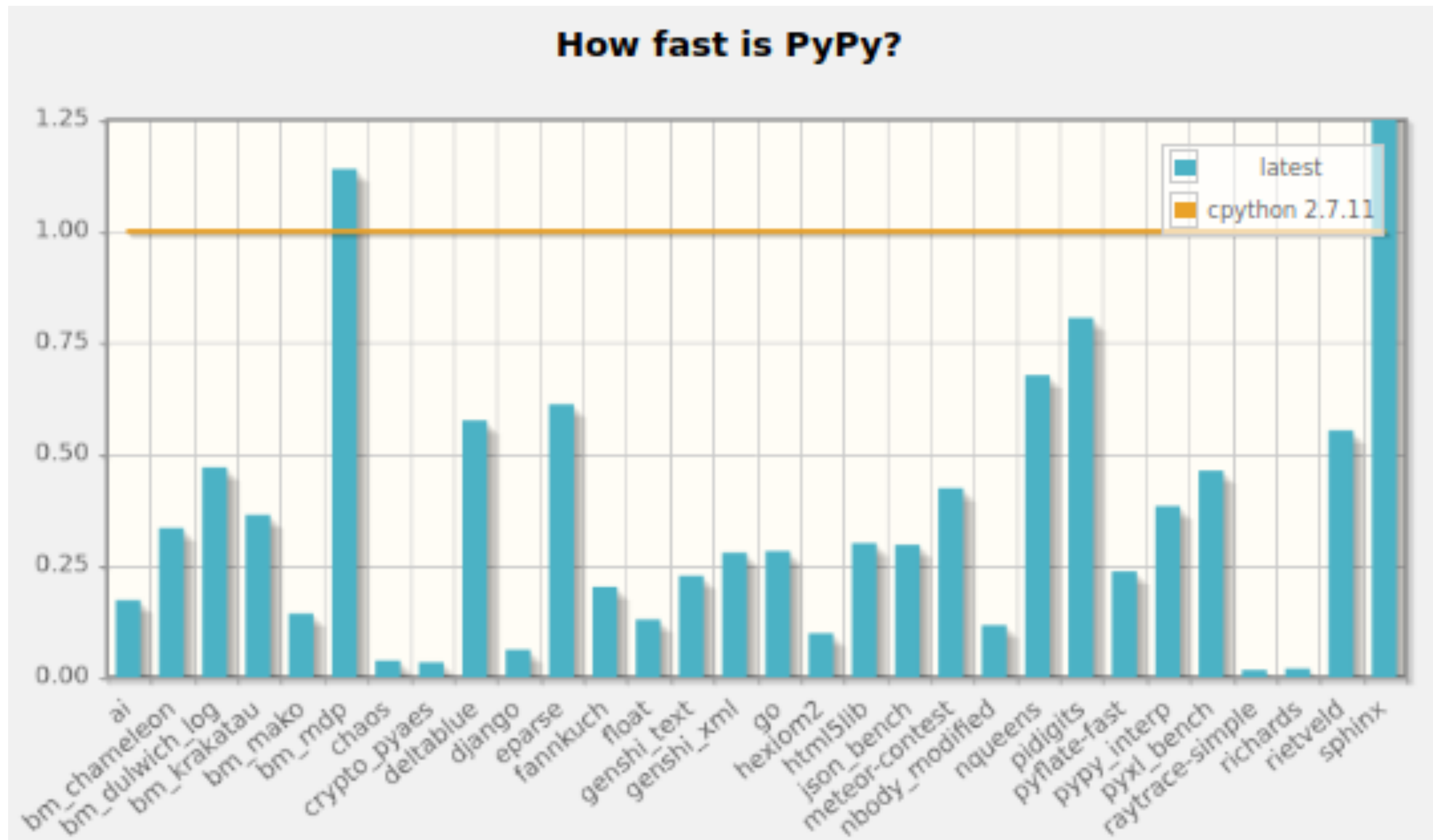
Time Computing mean Delay of Flights

System	Time in Seconds
Panda Dataframe	0.72
Dask	0.30
Julia Dataframe	0.25
Julia DB	0.12

PyPy

https://www.pypy.org

JIT compiler for Python



Overhead for Parallelizing

Dask Dataframe parallel version

```
%time df[['Origin','DepDelay']].groupby('Origin').mean().compute()
```

CPU times: user 273 ms,
sys: 29.3 ms,
total: 303 ms
Wall time: 2.12 s

Panda Dataframe version

```
pandaDF = df.compute()                    # Convert to normal Panda Dataframe
%time pandaDF[['Origin','DepDelay']].groupby('Origin').mean()
```

CPU times: user 492 ms,
sys: 223 ms,
total: 715 ms
Wall time: 679 ms

2.12/0.679 = 3.1

Julia DataFrame

```
using Lazy
using DataFramesMeta
using Statistics
using DataFrames

function csvdir_to_df(dir::String)
    @as x readdir(dir) begin
        map(i -> dir * i, x) #path to files
        map(readtable, x)
        reduce(vcat, x)     #combine into one dataframe
    end
end

function origin_depdelay(data::DataFrame)
   clean_depdely = DataFrames.dropmissing!(data,:DepDelay)
   by(clean_depdely, :Origin, :DepDelay => mean)
end
```

```
flightdf = csvdir_to_df(flightdir)
@time origin_depdelay((flightdf))

    0.246635 seconds
      (193 allocations:
        90.775 MiB,
        10.15% gc time)
```

JuliaDB

```
using JuliaDB

flightdir = "/Users/whitney/Courses/696/Spring20/Notebooks/dask-tutorial-master/data/nycflight

flighttable = loadtable(flightdir)

@time JuliaDB.groupby(mean ∘ skipmissing, flighttable, :Origin, select = :DepDelay)
```

0.119038 seconds (226 allocations: 44.846 MiB, 8.65% gc time)