

“Big Pandas” - Dask from the Inside

PyCon UK tutorial, 27 October 2017

Stephen Simmons

mail@stevesimmons.com

stephen.e.simmons@jpmorgan.com

<https://github.com/stevesimmons/pyconuk-2017-pandas-and-dask>

Goals for today...

Let's try a much bigger data set – “BTS OTP” (172m records)

Try some simple analysis

- DataFrames from dask rather than pandas
- Efficient data storage
- Calculating the dask dependency graph

(Note: Examples here all designed to run on a local machine)

American “on-time performance” flight data

Monthly zipped csv files

Each file has
450,000 rows x 109 cols

220MB unzipped
22MB zipped
12MB with LZMA (.xz)

https://www.transtats.bts.gov/acTableInfo.asp?Table_ID=236

United States Department of Transportation

Ask a

Bureau of Transportation Statistics

Explore Topics and Geography Browse Statistical Products and Data Learn About BTS and Our Work

TranStats
Search this site:
[Advanced Search](#)

Resources

- Database Directory
- Glossary
- Upcoming Releases
- Data Release History

Data Tools

- Analysis
- Table Contents
- Carrier Release Status
- Download
- Data Tables
- Database Profile

On-Time : On-Time Performance

[Database Profile](#) [Data Tables](#) [Table Contents](#)

Property	Description
Name	On-Time Performance
Description	This table contains on-time arrival data for non-stop domestic flights by major air carriers, and provides such additional items as departure and arrival delays, origin and destination airports, flight numbers, scheduled and actual departure and arrival times, cancelled or diverted flights, taxi-out and taxi-in times, air time, and non-stop distance.
Records	172,695,702
Fields	109
First Year	1987
Last Year	2017
Frequency	Monthly
Latest Available Data	January, 2017

Terms	Definitions
Actual Arrival Times	Gate arrival time is the instance when the pilot sets the aircraft parking brake after arriving at the airport gate or passenger unloading area. If the parking brake is not set, record the time for the opening of the passenger door. Also, carriers using a Docking Guidance System (DGS) may record the official gate-arrival time when the aircraft is stopped at the appropriate parking mark.

THE DATA

Twenty years of data (120 million observations) on commercial domestic flights in the United States.

Variables

- Dates: day of week, date, month, year
- Arrival and departure times: actual and scheduled
- Flight times: actual and scheduled
- Origin and destination: airport code, latitude, longitude
- Carrier: American, Aloha Air, ..., United, US Air

Data are from the Research and Innovative Technology Administration's air traffic control data, U.S. Department of Transportation research programs.

GOALS

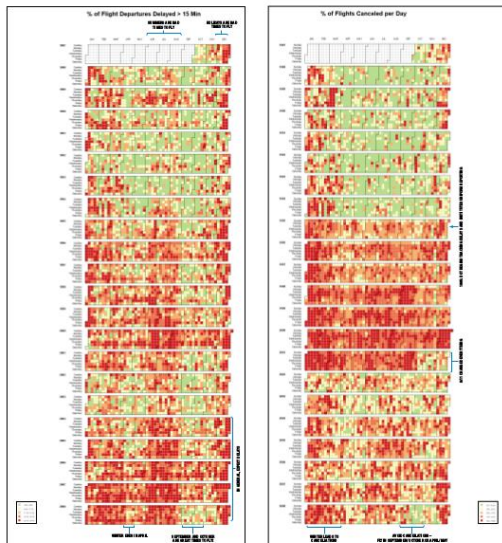
- Summarize data by time periods, airport, and carrier
- Temporal effects
 - Are some time periods more prone to delays than others?
 - Relationships between delays and seasonal factors: winter, summer, holidays
 - Weather factors: blizzards and severe weather
 - Daily factors: time of day of week
- Spatial effects
 - Are some airports more prone to delays than others?
 - Are there differences between flying into an airport and flying out?
- Carrier effects
 - Are some carriers more prone to delays than others?

LESSONS LEARNED: TIPS FOR TRAVELERS

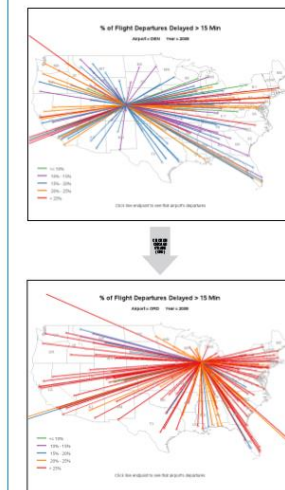
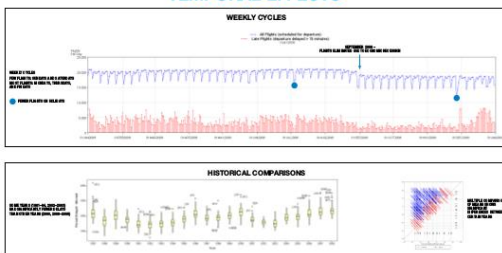


- Avoid flying during holidays and summer
- Fly in April, May, and September
- Watch the weather
- Avoid airports (Newark, JFK, Chicago,...) with consistent delays
- Use carriers (Aloha, Hawaiian, Southwest,...) with superior on-time performance
- Fly early in the day
- Avoid flights that depart between 5 and 7 p.m.

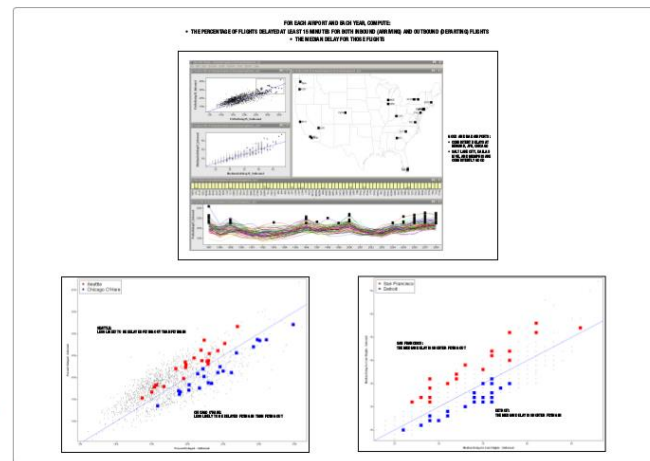
OVERVIEW



TEMPORAL EFFECTS



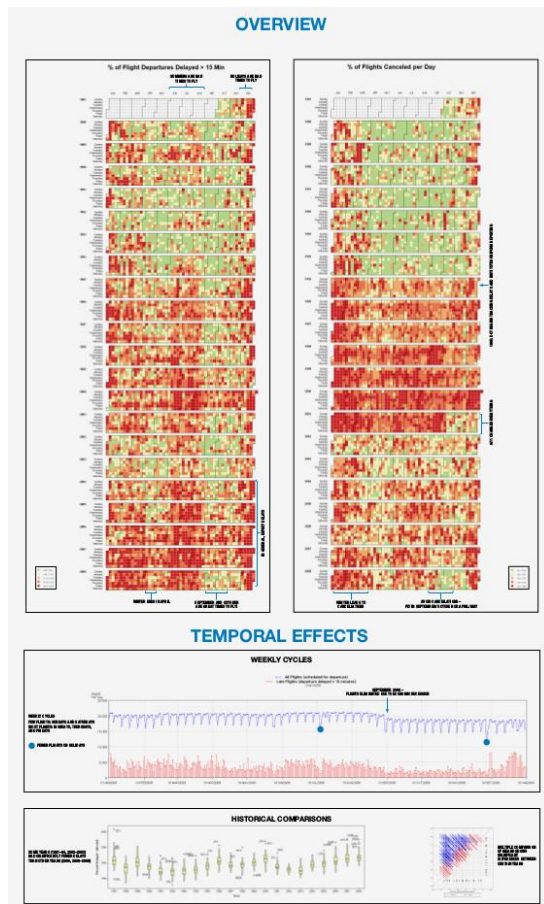
SPATIAL EFFECTS



CARRIER EFFECTS



Our analysis goals...



Calculate % of flights cancelled per day

- by date / date range
- by origin / destination
- by carrier
- by state

```
df = pd.read_csv('flights-2016-01.xz', nrows=4, dialect="excel")
```

```
> df.T
```

Year	2016	2016	2016	2016
Quarter	1	1	1	1
Month	1	1	1	1
DayofMonth	6	7	8	9
DayOfWeek	3	4	5	6
FlightDate	2016-01-06	2016-01-07	2016-01-08	2016-01-09
UniqueCarrier	AA	AA	AA	AA
AirlineID	19805	19805	19805	19805
Carrier	AA	AA	AA	AA
TailNum	N4YBAA	N434AA	N541AA	N489AA
FlightNum	43	43	43	43
...				
Origin	DFW	DFW	DFW	DFW
...				
Dest	DTW	DTW	DTW	DTW
...				
CRSDepTime	1100	1100	1100	1100
DepTime	1057	1056	1055	1102
DepDelay	-3	-4	-5	2
...				
TaxiOut	15	14	21	13
WheelsOff	1112	1110	1116	1115
WheelsOn	1424	1416	1431	1424
TaxiIn	8	10	14	9
CRSArrTime	1438	1438	1438	1438
ArrTime	1432	1426	1445	1433
ArrDelay	-6	-12	7	-5
...				
Cancelled	0	0	0	0
CancellationCode	NaN	NaN	NaN	NaN
Diverted	0	0	0	0
CRSElapsedTime	158	158	158	158
ActualElapsedTime	155	150	170	151
AirTime	132	126	135	129
Flights	1	1	1	1
Distance	986	986	986	986
...				

[110 rows x 4 columns]

Pandas DataFrames don't scale well

```
> df = pd.read_csv('flights-2016-01.xz',  
                  dialect="excel")
```

```
DtypeWarning: Columns (77) have mixed types.  
Specify dtype option on import or set low_memory=False.
```

```
CPU times: user 5.48 s, sys: 1.36 s, total: 6.83 s  
Wall time: 6.83 s
```

```
> df.info()  
<class 'pandas.core.frame.DataFrame'>  
RangeIndex: 445827 entries, 0 to 445826  
Columns: 110 entries, Year to Unnamed: 109  
dtypes: float64(71), int64(21), object(18)  
memory usage: 374.2+ MB
```

```
> df.memory_usage(deep=True).sum() / 2**20  
745.2
```

Pandas DataFrames don't scale well

```
> df = pd.read_csv('flights-2016-01.xz',  
                  dialect="excel")
```

DtypeWarning: Columns (77) have mixed types.
Specify dtype option on import or set low_memory=False.

CPU times: user 5.48 s, sys: 1.36 s, total: 6.83 s
Wall time: 6.83 s

```
> df.info()  
<class 'pandas.core.frame.DataFrame'>  
RangeIndex: 445827 entries, 0 to 445826  
Columns: 110 entries, Year to Unnamed: 109  
dtypes: float64(71), int64(21), object(18)  
memory usage: 374.2+ MB
```

```
> df.memory_usage(deep=True).sum() / 2**20  
745.2
```

```
> df = pd.concat([  
    pd.read_csv('flights-%s.xz' % m,  
                dialect="excel")  
    for m in ['2015-12', '2016-01', '2016-02']  
])
```

DtypeWarning: Columns (48,76,77,84,85) have mixed types.
DtypeWarning: Columns (77) have mixed types.
DtypeWarning: Columns (77,84) have mixed types.
Specify dtype option on import or set low_memory=False.

CPU times: user 18.4 s, sys: 6.66 s, total: 25.1 s
Wall time: 1min 26s

```
> df.info()  
<class 'pandas.core.frame.DataFrame'>  
Int64Index: 1348946 entries, 0 to 423888  
Columns: 110 entries, Year to Unnamed: 109  
dtypes: float64(69), int64(21), object(20)  
memory usage: 1.1+ GB
```

```
> df.memory_usage(deep=True).sum() / 2**20  
2326.9
```


Need to scale up in multiple places...

Data storage format

Load data in parallel

Parallelize intermediate calculations

Run on multiples core/machines

Effort required to write/optimize correct parallel code

And we still want to use Python and Pandas!

Dask is a 'drop-in' replacement for pandas*

```
> import dask.dataframe as dd

> paths = [ 'flights-%s.xz' % dt.strftime('%Y-%m')
            for dt in pd.date_range('2015-12', '2016-03', freq='M') ]

> cols = ['FlightDate', 'Origin', 'Dest', 'OriginState', 'DestState',
          'Carrier', 'FlightNum', 'TailNum', 'CRSDepTime', 'CRSArrTime',
          'DepDelay', 'ArrDelay', 'Flights', 'Cancelled', 'Diverted', ]

> ddf = dd.read_csv(
    paths,
    dialect="excel", encoding='latin-1',
    header=0, usecols=cols,
    compression='xz', blocksize=None,
    parse_dates=['FlightDate'],
    dtype={'FlightNum': str}, )
```

Dask is a 'drop-in' replacement for pandas*

```
> import dask.dataframe as dd

> paths = [ 'flights-%s.xz' % dt.strftime('%Y-%m')
            for dt in pd.date_range('2015-12', '2016-03', freq='M') ]

> cols = ['FlightDate', 'Origin', 'Dest', 'OriginState', 'DestState',
          'Carrier', 'FlightNum', 'TailNum', 'CRSDepTime', 'CRSArrTime',
          'DepDelay', 'ArrDelay', 'Flights', 'Cancelled', 'Diverted', ]

> ddf = dd.read_csv(
    paths,
    dialect="excel", encoding='latin-1',
    header=0, usecols=cols,
    compression='xz', blocksize=None,
    parse_dates=['FlightDate'],
    dtype={'FlightNum': str}, )

> ddf[['Carrier', 'Flights', 'Cancelled']].groupby('Carrier').sum()
CPU times: user 16 ms, sys: 0 ns, total: 16 ms
Wall time: 18.3 ms
```

* ... for a subset of pandas DataFrame operations

Dask is a 'drop-in' replacement for pandas*

```
> import dask.dataframe as dd

> paths = [ 'flights-%s.xz' % dt.strftime('%Y-%m')
            for dt in pd.date_range('2015-12', '2016-03', freq='M') ]

> cols = ['FlightDate', 'Origin', 'Dest', 'OriginState', 'DestState',
          'Carrier', 'FlightNum', 'TailNum', 'CRSDepTime', 'CRSArrTime',
          'DepDelay', 'ArrDelay', 'Flights', 'Cancelled', 'Diverted', ]

> ddf = dd.read_csv(
    paths,
    dialect="excel", encoding='latin-1',
    header=0, usecols=cols,
    compression='xz', blocksize=None,
    parse_dates=['FlightDate'],
    dtype={'FlightNum': str}, )

> ddf[['Carrier', 'Flights', 'Cancelled']].groupby('Carrier').sum()
CPU times: user 16 ms, sys: 0 ns, total: 16 ms
Wall time: 18.3 ms

> _.compute()
CPU times: user 13.9 s, sys: 2.22 s, total: 16.1 s
Wall time: 9.62 s
```

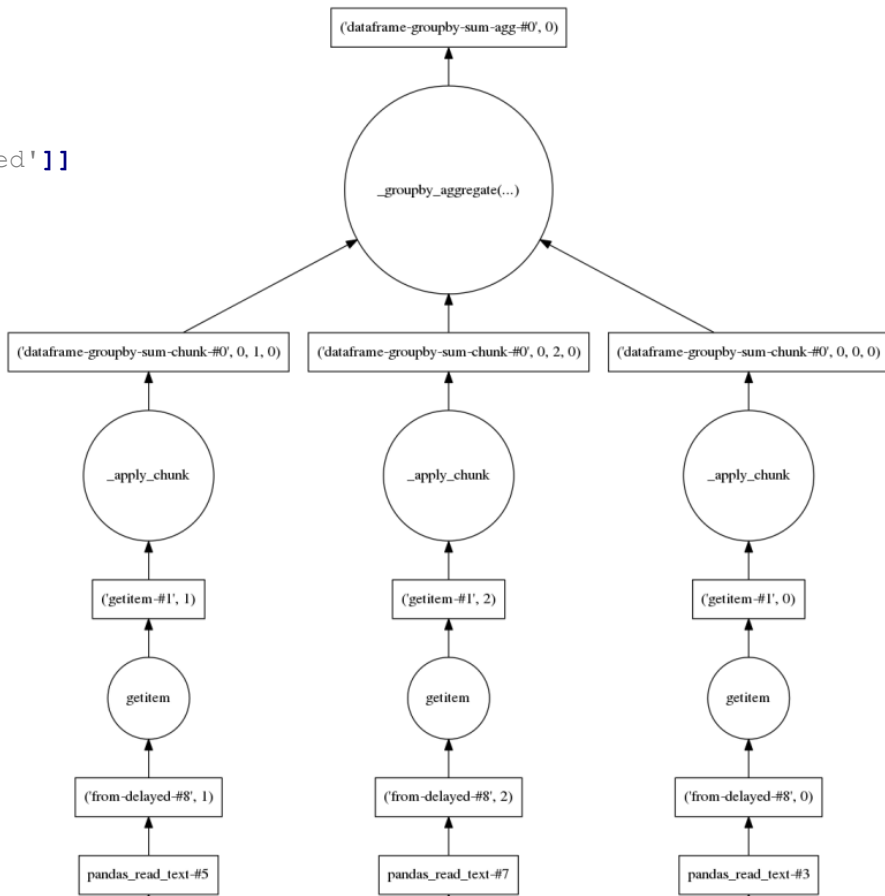
Carrier	Flights	Cancelled
AA	223582.0	5001.0
AS	42063.0	311.0
B6	68137.0	1522.0
DL	208121.0	1472.0
EV	126036.0	4409.0
F9	21865.0	266.0
HA	18390.0	12.0
MQ	20993.0	805.0
NK	32072.0	894.0
OO	140825.0	3471.0
UA	122148.0	2470.0
VX	15859.0	241.0
WN	308855.0	5677.0

Expressions build a dependency graph...

```
> task = ddf[['Carrier', 'Flights', 'Cancelled']]  
      .groupby('Carrier')  
      .sum()
```

```
> task.compute()  
CPU times: user 13.9 s,  
sys: 2.22 s, total: 16.1 s  
Wall time: 9.62 s
```

```
> task.visualize()
```

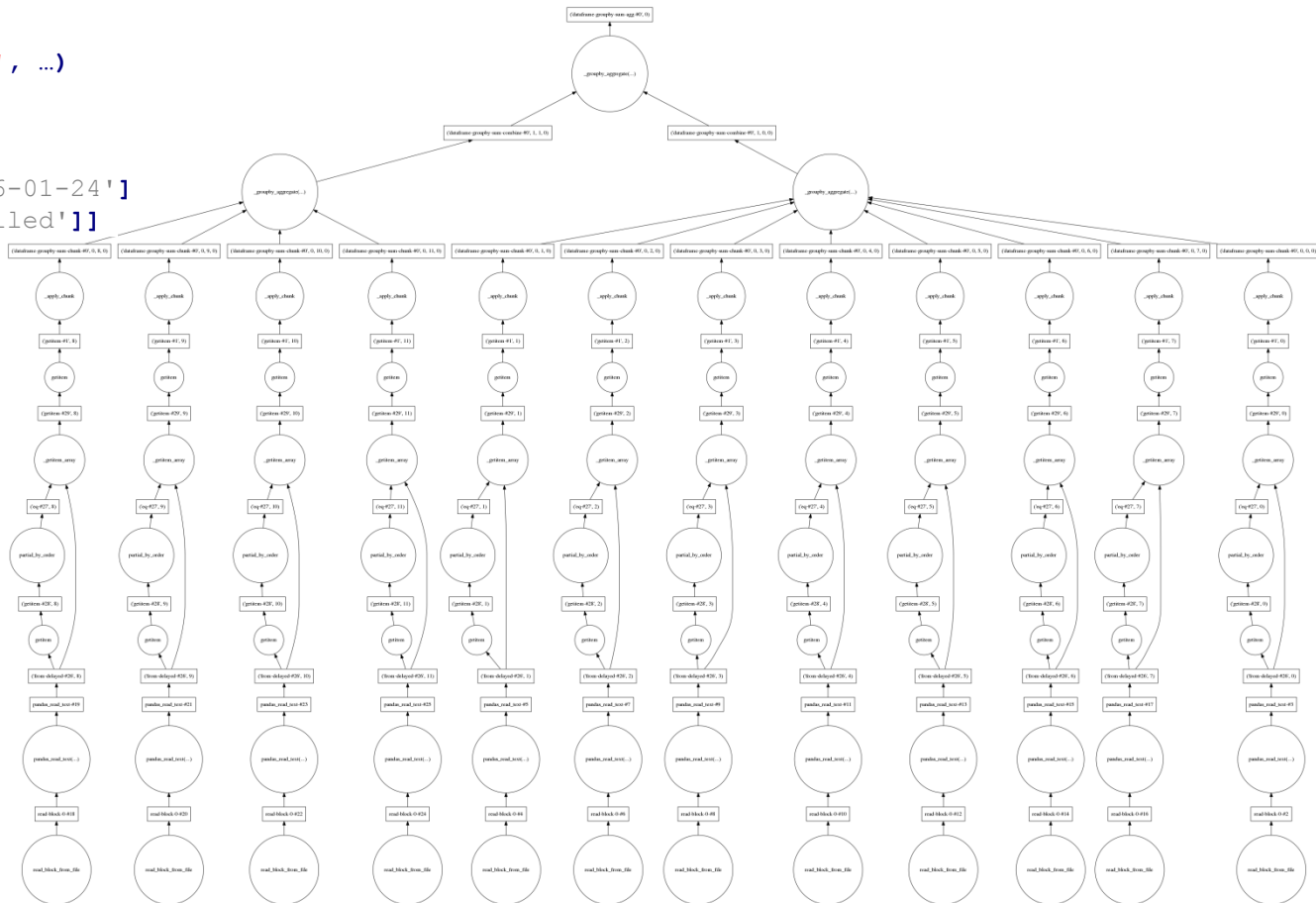


... that can get arbitrarily complex

```
> ddf = dd.read_csv('flights-*.xz', ...)
```

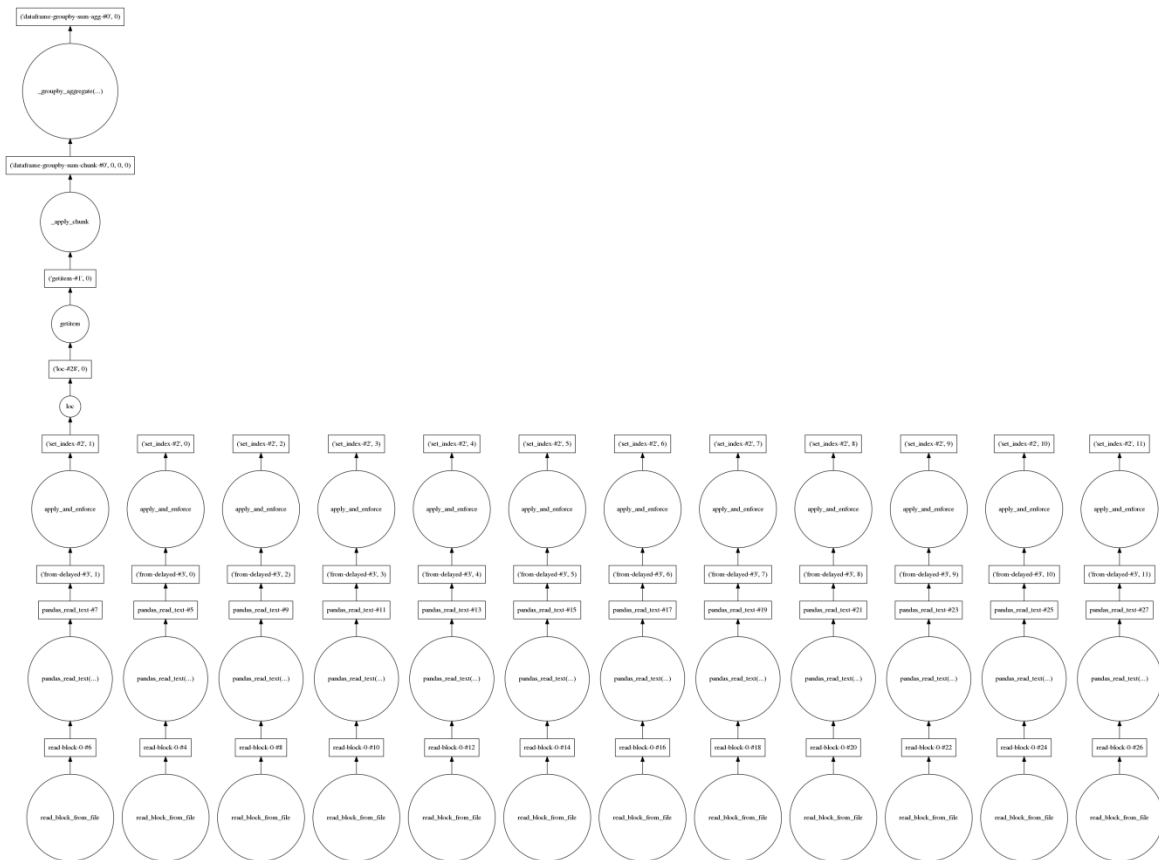
```
> task = ddf[ddf.FlightDate=='2016-01-24']  
  [['Carrier', 'Flights', 'Cancelled']]  
  .groupby('Carrier')  
  .sum()
```

```
> task.visualize()
```



Dask prunes the graph where it can

```
> ddf = dd.read_csv('flights-*.xz', ...)
> ddf = ddf.set_index('FlightDate')
> task = ddf['2016-01-24']
    [['Carrier', 'Flights', 'Cancelled']]
    .groupby('Carrier')
    .sum()
> task.visualize()
```



Storage formats like parquet are faster than csv

```
ddf = load_data(start='2016-01', end='2017-02')

def sort_partition(df):
    return df.set_index(df.FlightDate).sort_index()

task = ddf.map_partitions(func=sort_partition)

task.to_parquet('flights.parq',
                compression='SNAPPY')

$ ls -hs flights.parq/
```

```
total 100M
4.0K _common_metadata
20K _metadata
8.5M part.0.parquet
8.0M part.1.parquet
7.5M part.2.parquet
...
8.2M part.11.parquet
8.2M part.12.parquet
```


Storage formats like parquet are faster than csv

```
ddf = load_data(start='2016-01', end='2017-02')

def sort_partition(df):
    return df.set_index(df.FlightDate).sort_index()

task = ddf.map_partitions(func=sort_partition)

task.to_parquet('flights.parq',
                compression='SNAPPY')
```

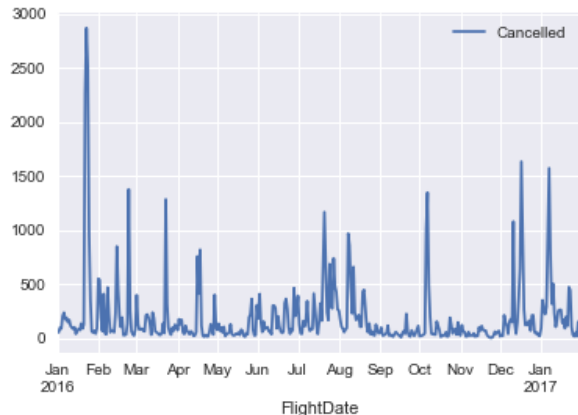
```
$ ls -hs flights.parq/
```

```
total 100M
4.0K _common_metadata
20K _metadata
8.5M part.0.parquet
8.0M part.1.parquet
7.5M part.2.parquet
...
8.2M part.11.parquet
8.2M part.12.parquet
```

Parquet example 1 – whole dataset

```
ddf = dd.read_parquet('flights.parq',
                      columns=['Cancelled']) # 50 ms

task = ddf.groupby(ddf.index).sum()
out = task.compute() # 313 ms
out.plot()
```



Parquet example 2: read a subset of the partitions

[illegible]

```
x = (ddf.loc['2016-01-18':'2016-01-28']
      .reset_index() # Move FlightDate from index to a column
      .groupby(by=['Carrier', 'FlightDate'])
      .sum()
    )
```

```
y = x['Cancelled'] / x['Flights'] * 100
```

```
out = y.compute()
```

220 ms to read data, build graph and calculate

```
> out
```

Carrier	FlightDate	
AA	2016-01-19	0.339992
	2016-01-20	0.160064
	2016-01-21	1.327606
	2016-01-22	30.712339
	2016-01-23	36.307838
	2016-01-24	25.588114
	2016-01-25	8.853119
		...
WN	2016-01-21	1.133787
	2016-01-22	14.436219
	2016-01-23	21.369961
	2016-01-24	17.904074
	2016-01-25	4.630682
	2016-01-26	1.394422
	2016-01-27	0.910643
	2016-01-28	0.597610

```
dtype: float64
```

Parquet example 2: read a subset of the partitions

```
ddf = dd.read_parquet('flights.parq',
                      columns=['Carrier', 'Flights', 'Cancelled'])
```

```
x = (ddf.loc['2016-01-18':'2016-01-28']
     .reset_index() # Move FlightDate from index to a column
     .groupby(by=['Carrier', 'FlightDate'])
     .sum())
```

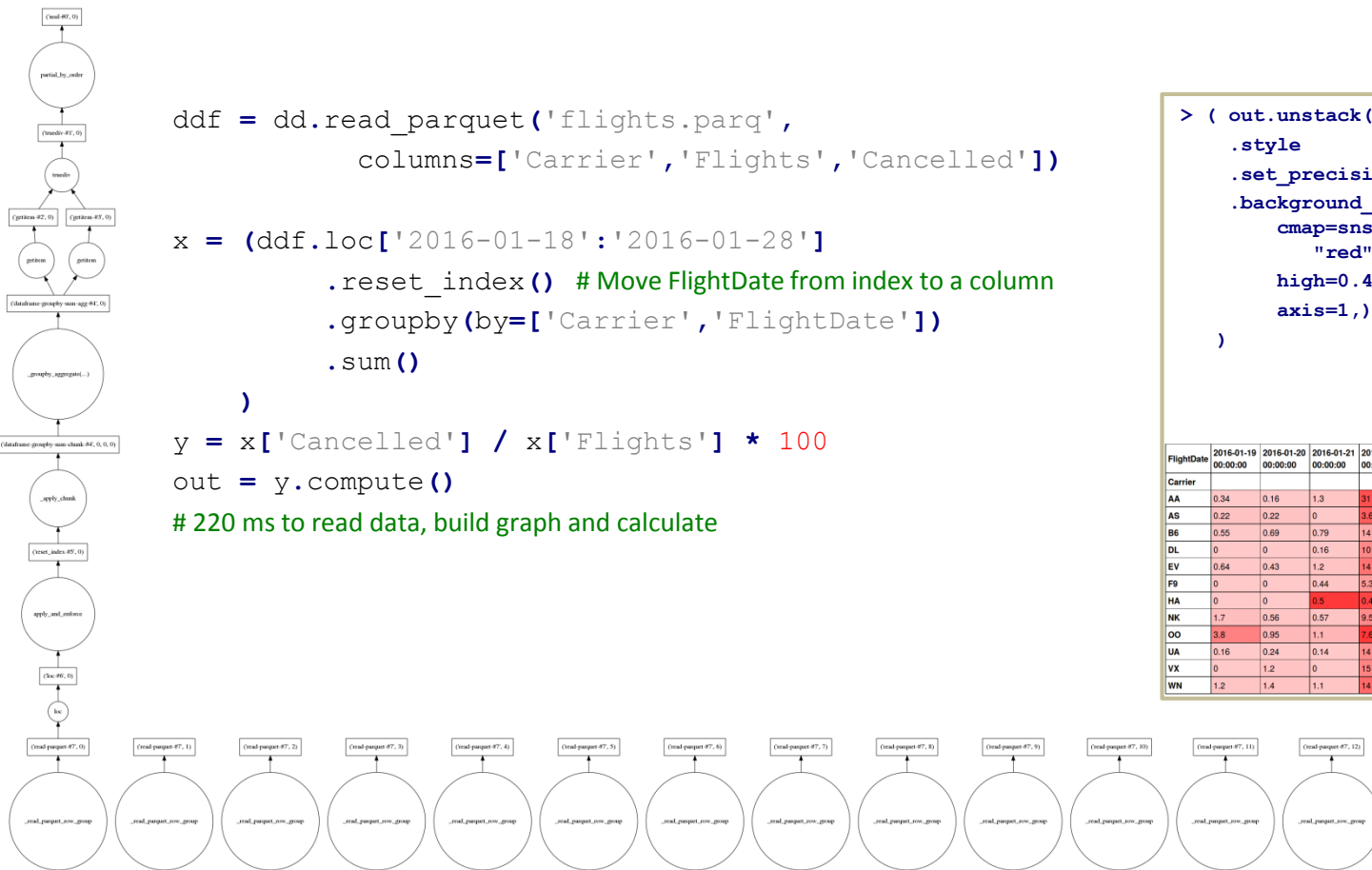
```
y = x['Cancelled'] / x['Flights'] * 100
```

```
out = y.compute()
```

220 ms to read data, build graph and calculate

```
> ( out.unstack('FlightDate')
    .style
    .set_precision(2)
    .background_gradient(
      cmap=sns.light_palette(
        "red", as_cmap=True),
      high=0.4, low=0.2,
      axis=1,
    )
  )
```

FlightDate	2016-01-19 00:00:00	2016-01-20 00:00:00	2016-01-21 00:00:00	2016-01-22 00:00:00	2016-01-23 00:00:00	2016-01-24 00:00:00	2016-01-25 00:00:00	2016-01-26 00:00:00	2016-01-27 00:00:00	2016-01-28 00:00:00
Carrier										
AA	0.34	0.16	1.3	31	36	26	8.9	0.68	0.36	0.59
AS	0.22	0.22	0	3.6	4.9	4.1	1.3	0.22	0.22	0.21
B6	0.55	0.69	0.79	14	63	33	2.5	0	0.14	0.13
DL	0	0	0.16	10	22	12	0.88	0	0	0.16
EV	0.64	0.43	1.2	14	23	20	15	12	0.5	0.27
F9	0	0	0.44	5.3	16	8	0.45	0	0	0
HA	0	0	0.5	0.47	0.5	0	0	0	0	0
NK	1.7	0.56	0.57	9.5	22	13	2.5	0	0	0.28
OO	3.8	0.95	1.1	7.6	3.4	2.5	1.3	0.38	0.38	0.062
UA	0.16	0.24	0.14	14	29	32	19	8.4	0.47	0.14
VX	0	1.2	0	15	30	27	6.1	0	0.58	0
WN	1.2	1.4	1.1	14	21	18	4.6	1.4	0.91	0.6



Monitoring progress of a Dask calculation

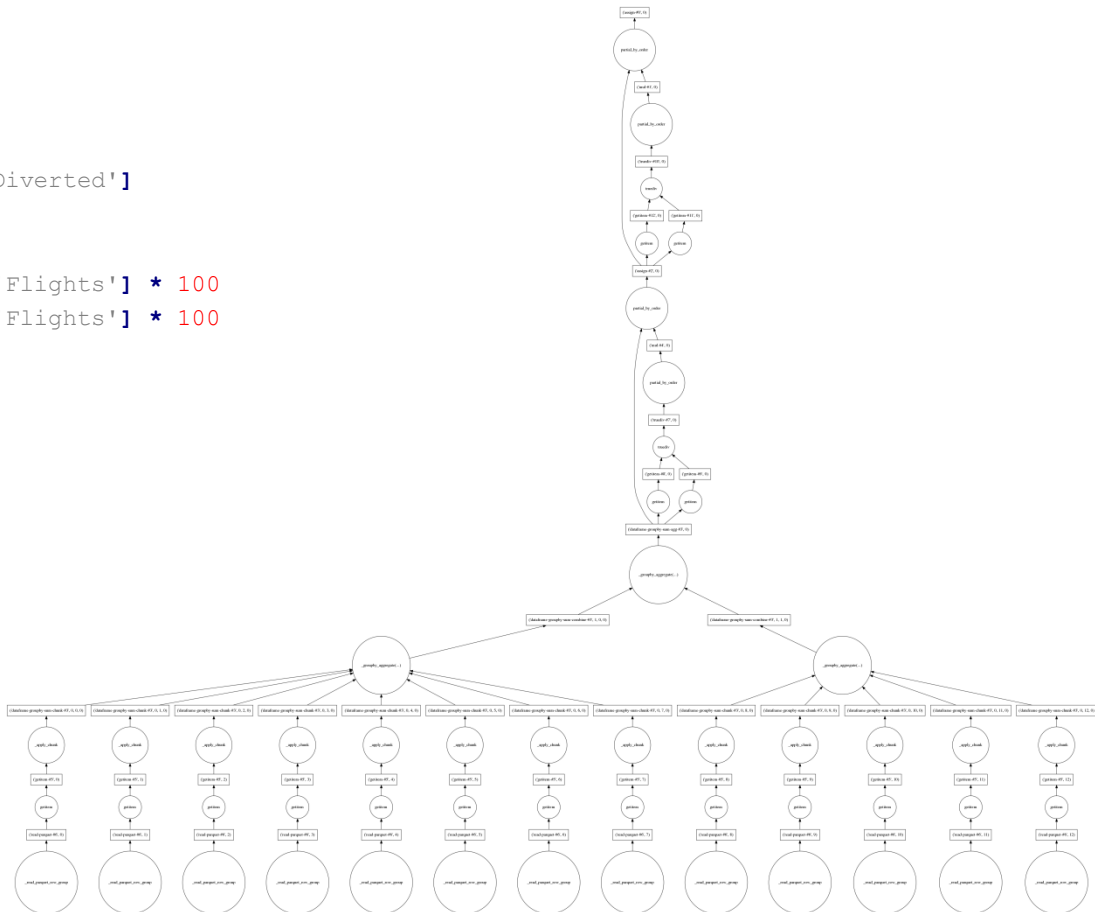
```
ddf = dd.read_parquet('flights.parq')
```

```
sum_cols = ['Carrier', 'Flights', 'Cancelled', 'Diverted']
```

```
task = ddf[sum_cols].groupby('Carrier').sum()
```

```
task['CancelledPct'] = task['Cancelled'] / task['Flights'] * 100
```

```
task['DivertedPct'] = task['Diverted'] / task['Flights'] * 100
```



Monitoring progress of a Dask calculation

```
ddf = dd.read_parquet('flights.parq')

sum_cols = ['Carrier', 'Flights', 'Cancelled', 'Diverted']
task = ddf[sum_cols].groupby('Carrier').sum()

task['CancelledPct'] = task['Cancelled'] / task['Flights'] * 100
task['DivertedPct' ] = task['Diverted' ] / task['Flights'] * 100

from dask.diagnostics import ProgressBar

with ProgressBar():
    out = task.compute()
```

```
[ ] | 0% Completed | 0.0s
```

>

Monitoring progress of a Dask calculation

```
ddf = dd.read_parquet('flights.parq')

sum_cols = ['Carrier', 'Flights', 'Cancelled', 'Diverted']
task = ddf[sum_cols].groupby('Carrier').sum()

task['CancelledPct'] = task['Cancelled'] / task['Flights'] * 100
task['DivertedPct' ] = task['Diverted' ] / task['Flights'] * 100

from dask.diagnostics import ProgressBar

with ProgressBar():
    out = task.compute()
```

```
[#####          ] | 87% Completed | 2.4s
```

>

Monitoring progress of a Dask calculation

```
ddf = dd.read_parquet('flights.parq')

sum_cols = ['Carrier', 'Flights', 'Cancelled', 'Diverted']
task = ddf[sum_cols].groupby('Carrier').sum()

task['CancelledPct'] = task['Cancelled'] / task['Flights'] * 100
task['DivertedPct'] = task['Diverted'] / task['Flights'] * 100

from dask.diagnostics import ProgressBar

with ProgressBar():
    out = task.compute()
```

```
[#####] |100% Completed | 2.6s
```

```
> print(out)
```

	Flights	Cancelled	Diverted	CancelledPct	DivertedPct
Carrier					
AA	987627.0	11847.0	2421.0	1.199542	0.245133
AS	191991.0	1072.0	520.0	0.558360	0.270846
B6	307075.0	4322.0	774.0	1.407474	0.252056
DL	992559.0	4898.0	1923.0	0.493472	0.193742
EV	526027.0	13048.0	1723.0	2.480481	0.327550
F9	102881.0	1341.0	169.0	1.303448	0.164267
HA	83065.0	136.0	91.0	0.163727	0.109553
NK	150769.0	3070.0	218.0	2.036228	0.144592
OO	656079.0	10326.0	2181.0	1.573896	0.332429
UA	587470.0	5702.0	1522.0	0.970603	0.259077
VX	74903.0	806.0	272.0	1.076058	0.363136
WN	1407229.0	18179.0	3324.0	1.291830	0.236209

Profiling a Dask calculation

```
ddf = dd.read_parquet('flights.parq')

sum_cols = ['Carrier', 'Flights', 'Cancelled', 'Diverted']
task = ddf[sum_cols].groupby('Carrier').sum()

task['CancelledPct'] = task['Cancelled'] / task['Flights'] * 100
task['DivertedPct' ] = task['Diverted' ] / task['Flights'] * 100

from dask.diagnostics import Profiler, ResourceProfiler, CacheProfiler
from cachey import nbytes

with (Profiler() as prof, ResourceProfiler(dt=0.25) as rprof,
      CacheProfiler(metric=nbytes) as cprof):
    df = task.compute()
```


Profiling a Dask calculation

```
ddf = dd.read_parquet('flights.parq')

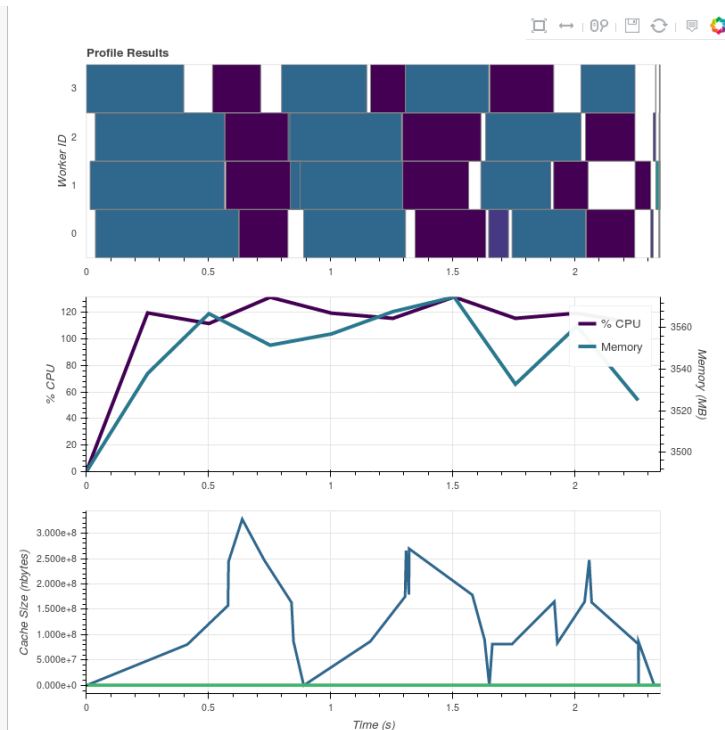
sum_cols = ['Carrier', 'Flights', 'Cancelled', 'Diverted']
task = ddf[sum_cols].groupby('Carrier').sum()

task['CancelledPct'] = task['Cancelled'] / task['Flights'] * 100
task['DivertedPct'] = task['Diverted'] / task['Flights'] * 100

from dask.diagnostics import Profiler, ResourceProfiler, CacheProfiler
from cachey import nbytes

with (Profiler() as prof, ResourceProfiler(dt=0.25) as rprof,
      CacheProfiler(metric=nbytes) as cprof):
    df = task.compute()

dask.diagnostics.visualize([prof, rprof, cprof], save=False, show=True)
```



So what exactly is a Dask DataFrame?

```
> print(dd.DataFrame.__doc__)
```

Implements out-of-core DataFrame as a sequence of pandas DataFrames

Parameters

dask: dict

The dask graph to compute this DataFrame

name: str

The key prefix that specifies which keys in the dask
comprise this particular DataFrame

meta: pandas.DataFrame

An empty ``pandas.DataFrame`` with names, dtypes, and
index matching the expected output.

divisions: tuple of index values

Values along which we partition our blocks on the index

```
def from_pandas(data, npartitions=None, chunksize=None, sort=True, name=None):
```

"""

Construct a Dask DataFrame from a Pandas DataFrame

This splits an in-memory Pandas dataframe into several parts and constructs
a dask.dataframe from those parts on which Dask.dataframe can operate in
parallel.

Note that, despite parallelism, Dask.dataframe may not always be faster
than Pandas. We recommend that you stay with Pandas for as long as
possible before switching to Dask.dataframe.

Parameters

data : pandas.DataFrame or pandas.Series

The DataFrame/Series with which to construct a Dask DataFrame/Series

npartitions : int, optional

The number of partitions of the index to create. Note that depending on
the size and index of the dataframe, the output may have fewer
partitions than requested.

chunksize : int, optional

The size of the partitions of the index.

sort: bool

Sort input first to obtain cleanly divided partitions or don't sort and
don't get cleanly divided partitions

name: string, optional

An optional keyname for the dataframe. Defaults to hashing the input

Returns

dask.DataFrame or dask.Series

A dask DataFrame/Series partitioned along the index

The simplest Dask DataFrame

```
>>> df = pd.DataFrame([[1,2,3],[4,5,6],[7,8,9],  
                        [10,11,12],[13,14,15]], columns=['a','b','c'])
```

	a	b	c
0	1	2	3
1	4	5	6
2	7	8	9
3	10	11	12
4	13	14	15

```
>>> ddf = dd.from_pandas(df, npartitions=1)
```

npartitions=1	a	b	c
	int64	int64	int64
0
4

Dask Name: from_pandas, 2 tasks

```
>>> ddf.divisions  
(0, 4)
```

```
>>> ddf._meta  
Empty DataFrame  
Columns: [a, b, c]  
Index: []
```

```
> print(dd.DataFrame.__doc__)  
dask: dict  
    The dask graph to compute this DataFrame  
name: str  
    The key prefix that specifies which keys in the dask  
    comprise this particular DataFrame  
meta: pandas.DataFrame  
    An empty ``pandas.DataFrame`` with names, dtypes, and  
    index matching the expected output.  
divisions: tuple of index values  
    Values along which we partition our blocks on the index
```

```
>>> ddf._name  
'from_pandas-b71f6a90'
```

```
>>> ddf.dask  
{  
    ('from_pandas-b71f6a90', 0): df,  
}
```

```
>>> ddf.visualize()
```

```
('from_pandas-#0', 0)
```

The next simplest Dask DataFrame

```
>>> df = pd.DataFrame([[1,2,3],[4,5,6],[7,8,9],  
                        [10,11,12],[13,14,15]], columns=['a','b','c'])
```

	a	b	c
0	1	2	3
1	4	5	6
2	7	8	9
3	10	11	12
4	13	14	15

```
>>> ddf = dd.from_pandas(df, npartitions=2)
```

npartitions=1	a	b	c
	int64	int64	int64
0
3
4

Dask Name: from_pandas, 2 tasks

```
>>> ddf.divisions  
(0, 3, 4)
```

```
>>> ddf._meta  
Empty DataFrame  
Columns: [a, b, c]  
Index: []
```

```
> print(dd.DataFrame.__doc__)  
dask: dict  
    The dask graph to compute this DataFrame  
name: str  
    The key prefix that specifies which keys in the dask  
    comprise this particular DataFrame  
meta: pandas.DataFrame  
    An empty ``pandas.DataFrame`` with names, dtypes, and  
    index matching the expected output.  
divisions: tuple of index values  
    Values along which we partition our blocks on the index
```

```
>>> ddf._name  
'from_pandas-de36e0f9'
```

```
>>> ddf.dask  
{  
    ('from_pandas-de36e0f9', 0): df[0:3],  
    ('from_pandas-de36e0f9', 1): df[3:],  
}
```

```
>>> ddf.visualize()
```

('from_pandas-#0', 0)

('from_pandas-#0', 1)

dd.DataFrame.dask

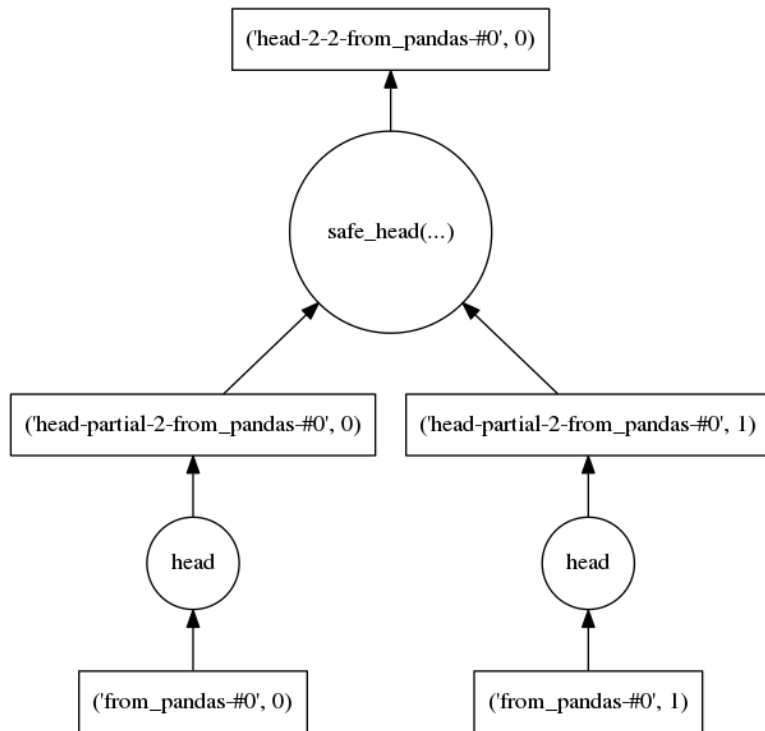
```
>>> ddf = dd.from_pandas(df, npartitions=2)
      .head(n=2, npartitions=2, compute=False)

>>> ddf._name
'head-2-2-from_pandas-de36e0f9b'

>>> ddf.dask
{
('head-2-2-from_pandas-de36e0f9b', 0):
  ( dask.dataframe.core.safe_head,
    ( function dask.dataframe.core._concat,
      [ ('head-partial-2-from_pandas-de36e0f9b', 0),
        ('head-partial-2-from_pandas-de36e0f9b', 1) ]
    ), 2
  ),
('head-partial-2-from_pandas-de36e0f9b', 0):
  (<methodcaller: head>, ('from_pandas-de36e0f9b', 0), 2),
('head-partial-2-from_pandas-de36e0f9b', 1):
  (<methodcaller: head>, ('from_pandas-de36e0f9b', 1), 2),
('from_pandas-de36e0f9b', 0):    df[0:3],
('from_pandas-de36e0f9b', 1):    df[3:],
}

>>> ddf._keys()
[('head-2-2-from_pandas-de36e0f9b', 0)]

>>> ddf.compute()
```



Reimplement pandas methods lazily

```
>>> ddf = dd.from_pandas(df, npartitions=2)
      .head(n=2, npartitions=2, compute=False)

>>> ddf._name
'head-2-2-from_pandas-de36e0f9b'

>>> ddf.dask
{
  ('head-2-2-from_pandas-de36e0f9b', 0):
    ( dask.dataframe.core.safe_head,
      ( function dask.dataframe.core._concat,
        [ ('head-partial-2-from_pandas-de36e0f9b', 0),
          ('head-partial-2-from_pandas-de36e0f9b', 1) ]
        ), 2
      ),
  ('head-partial-2-from_pandas-de36e0f9b', 0):
    (<methodcaller: head>, ('from_pandas-de36e0f9b', 0), 2),
  ('head-partial-2-from_pandas-de36e0f9b', 1):
    (<methodcaller: head>, ('from_pandas-de36e0f9b', 1), 2),
  ('from_pandas-de36e0f9b', 0):      df[0:3],
  ('from_pandas-de36e0f9b', 1):      df[3:],
}

>>> ddf._keys()
[('head-2-2-from_pandas-de36e0f9b', 0)]

>>> ddf.compute()
```

```
dask.dataframe.core._Frame (L758+):

def head(self, n=5, npartitions=1, compute=True):
    """ First n rows of the dataset"""
    if npartitions <= -1:
        npartitions = self.npartitions

    name = 'head-%d-%d-%s' % (npartitions, n, self._name)

    if npartitions > 1:
        name_p = 'head-partial-%d-%s' % (n, self._name)
        dsk = {}
        for i in range(npartitions):
            dsk[(name_p, i)] = (M.head, (self._name, i), n)
        concat = (_concat, [(name_p, i) for i in range(npartitions)])
        dsk[(name, 0)] = (safe_head, concat, n)
    else:
        dsk = {(name, 0): (safe_head, (self._name, 0), n)}

    result = new_dd_object(merge(self.dask, dsk), name, self._meta,
                            [self.divisions[0], self.divisions[npartitions]])

    if compute:
        result = result.compute()
    return result
```

Now we execute the graph with 'compute'

1. Optimize

- Cull – remove unnecessary tasks
- Fuse tasks – make parallelization less granular
- Inline cheap functions

2. Get graphs keys to evaluate

3. Execute in parallel with scheduler

- 'get' function
- Sort nodes
- Balance work between threads, processes, cores, over a cluster

4. Optionally cache intermediate results

References in dask source code:

- optimize.py, order.py and async.py

```
dask.dataframe.base (L139+):
```

```
def compute(*args, **kwargs):
    """Compute several dask collections at once.
    args      : Any dask objects are computed and the result is returned.
    traverse   : Set to False to not look for dask objects in Python collections.
    get        : An optional alternative scheduler ``get`` function to use.
    optimize_graph : If True [default], optimize the graph before computation.
                   Otherwise run as is. This can be useful for debugging.
    kwargs     : Extra keywords to forward to the scheduler ``get`` function.
    """
    from dask.delayed import delayed
    traverse = kwargs.pop('traverse', True)
    if traverse:
        args = tuple(delayed(a) if isinstance(a,
                                              (list, set, tuple, dict, Iterator))
                      else a for a in args)

    optimize_graph = kwargs.pop('optimize_graph', True)
    variables = [a for a in args if isinstance(a, Base)]
    if not variables:
        return args

    get = kwargs.pop('get', None) or _globals['get']
    dsk = collections_to_dsk(variables, optimize_graph, **kwargs)
    keys = [var._keys() for var in variables]
    results = get(dsk, keys, **kwargs)

    results_iter = iter(results)
    return tuple(a if not isinstance(a, Base)
                 else a._finalize(next(results_iter))
                 for a in args)
```

Conclusions

Dask is neat!

Especially in combination with parquet and distributed schedulers

Rough edges where it isn't quite pandas...

Distributed operations have very different costs than in-memory pandas

I'd like more time to experiment with distributed schedulers (dask/distributed, dask/dec2), and compare storage formats (csv, Parquet, HDF5, HDFS, etc)

Conclusions

Dask is neat!

Especially in combination with parquet and distributed schedulers

Rough edges where it isn't quite pandas...

Distributed operations have very different costs than in-memory pandas

I'd like more time to experiment with distributed schedulers (dask/distributed, dask/dec2), and compare storage formats (csv, Parquet, HDF5, HDFS, etc)

... but is Dask really the future of 'big pandas'?