PARALLEL COMPUTING WITH DASK

# Building Dask Bags & Globbing

Dhavide Aruliah
Director of Training, Anaconda

# Sequences to Bags

```
In [1]: nested_containers = [
   ...:   [0, 1, 2, 3],
   ...:   {},
   ...:   [6.5, 3.14],
   ...:   'Python',
   ...:   {'version':3},
   ...:   ''
   ...: ]

In [2]: import dask.bag as db

In [3]: the_bag = db.from_sequence(nested_containers)

In [4]: the_bag.count()
Out[4]:
6

In [5]: the_bag.any(), the_bag.all()
Out[5]:
True, False
```

# Reading Text Files

```
In [6]: import dask.bag as db

In [7]: zen = db.read_text('zen')

In [8]: taken = zen.take(1)

In [9]: type(taken)
Out[9]: tuple

In [10]: taken
Out[10]: ('The Zen of Python, by Tim Peters\n',)

In [11]: zen.take(3)
Out[11]:
('The Zen of Python, by Tim Peters\n',
 '\n',
 'Beautiful is better than ugly.\n')
```

# Glob Expressions

```
In [12]: import dask.dataframe as dd

In [13]: df = dd.read_csv('taxi/*.csv', assume_missing=True)
```

- `taxi/*.csv` is a *glob expression*
- `taxi/*.csv` matches:

```
taxi/yellow_tripdata_2015-01.csv
taxi/yellow_tripdata_2015-02.csv
taxi/yellow_tripdata_2015-03.csv
taxi/yellow_tripdata_2015-04.csv
taxi/yellow_tripdata_2015-05.csv
taxi/yellow_tripdata_2015-06.csv
taxi/yellow_tripdata_2015-07.csv
taxi/yellow_tripdata_2015-08.csv
taxi/yellow_tripdata_2015-09.csv
taxi/yellow_tripdata_2015-10.csv
taxi/yellow_tripdata_2015-11.csv
taxi/yellow_tripdata_2015-12.csv
```

# Using Python's glob Module

```
In [14]: %ls
Alice   Dave    README  a02.txt a04.txt b05.txt b07.txt b09.txt b11.txt
Bob     Lisa    a01.txt a03.txt a05.txt b06.txt b08.txt b10.txt taxi

In [15]: import glob

In [16]: txt_files = glob.glob('*.txt')

In [17]: txt_files
Out[17]:
['a01.txt',
 'a02.txt',
 'a03.txt',
 'a04.txt',
 'a05.txt',
 'b05.txt',
 'b06.txt',
 'b07.txt',
 'b08.txt',
 'b09.txt',
 'b10.txt',
 'b11.txt']
```

# More Glob Patterns

```
In [18]: glob.glob('b*.txt')
Out[18]:
['b05.txt',
 'b06.txt',
 'b07.txt',
 'b08.txt',
 'b09.txt',
 'b10.txt',
 'b11.txt']

In [19]: glob.glob('b?.txt')
Out[19]: []
```

```
In [20]: glob.glob('?0[1-6].txt')
Out[20]:
['a01.txt',
 'a02.txt',
 'a03.txt',
 'a04.txt',
 'a05.txt',
 'b05.txt',
 'b06.txt']

In [21]: glob.glob('??[1-6].txt')
Out[21]:
['a01.txt',
 'a02.txt',
 'a03.txt',
 'a04.txt',
 'a05.txt',
 'b05.txt',
 'b06.txt',
 'b11.txt']
```

# Permissible glob Patterns

- Filename characters (e.g., `file-02_tmp.txt`)
- Wildcard character `*`: matches 0 or more
- Wildcard character `?`: matches exactly 1
- Character ranges (e.g., `[0-5]`, `[a-m]`, `[A-Z0-9]`)

PARALLEL COMPUTING WITH DASK

# Let's practice!

PARALLEL COMPUTING WITH DASK

# Functional Approaches using Dask Bags

Dhavide Aruliah
Director of Training, Anaconda

# Functional Programming

- Functions: *first-class* data
- **Higher-order functions**:
  - functions as *input* or *output* to functions
- Functions replacing loops with:
  - **map** operations
  - **filter** operations
  - **reduction** operations (or **aggregations**)

# Using map

```
In [1]: def squared(x):
   ...:     return x ** 2

In [2]: squares = map(squared, [1, 2, 3, 4, 5, 6])

In [3]: squares
Out[3]: <map at 0x1037a1b70>

In [4]: squares = list(squares)

In [5]: squares
Out[5]: [1, 4, 9, 16, 25, 36]
```

# Using filter

```
In [6]: def is_even(x):
   ...:     return x % 2 == 0

In [7]: evens = filter(is_even, [1, 2, 3, 4, 5, 6])

In [8]: list(evens)
Out[8]: [2, 4, 6]

In [9]: even_squares = filter(is_even, squares))

In [10]: list(even_squares)
Out[10]: [4, 16, 36]
```

# Using dask.bag.map

```
In [11]: import dask.bag as db

In [12]: numbers = db.from_sequence([1, 2, 3, 4, 5, 6])

In [13]: squares = numbers.map(squared)

In [14]: squares
Out[14]: dask.bag<map-squared, npartitions=6>

In [15]: result = squares.compute() # Must fit in memory

In [16]: result
Out[16]: [1, 4, 9, 16, 25, 36]
```

# Using dask.bag.filter

```
In [17]: numbers = db.from_sequence([1, 2, 3, 4, 5, 6])

In [18]: evens = numbers.filter(is_even)

In [19]: evens.compute()
Out[19]: [2, 4, 6]

In [20]: even_squares = numbers.map(squared).filter(is_even)

In [21]: even_squares.compute()
Out[21]: [4, 16, 36]
```

# Using .str & String Methods

```
In [22]: zen = db.read_text('zen.txt')

In [23]: uppercase = zen.str.upper()

In [24]: uppercase.take(1)
Out[24]: ('THE ZEN OF PYTHON, BY TIM PETERS\n',)

In [25]: def my_upper(string):
   ...:     return string.upper()

In [26]: my_uppercase = zen.map(my_upper)

In [27]: my_uppercase.take(1)
Out[27]: ('THE ZEN OF PYTHON, BY TIM PETERS\n',)
```

# A Bigger Example I

```
In [28]: def load(k):
    ...:     template = 'yellow_tripdata_2015-{:02d}.csv'
    ...:     return pd.read_csv(template.format(k))

In [29]: def average(df):
    ...:     return df['total_amount'].mean()

In [30]: def total(df):
    ...:     return df['total_amount'].sum()

In [31]: data = db.from_sequence(range(1, 13)).map(load)

In [32]: data
Out[32]: dask.bag<map-loa..., npartitions=12>
```

# A Bigger Example II

```
In [33]: totals = data.map(total)

In [34]: averages = data.map(average)

In [35]: totals.compute()
Out[35]:
[1175217.5200009614,
 947282.0900005419,
 956752.3400005258,
 1304602.4800011297,
 1354966.290001166,
 1251511.6500010253,
 1167936.1000008786,
 915174.880000469,
 994643.300000564,
 1273267.4800010026,
 1158279.990000822,
 1166242.130000856]
```

```
In [36]: averages.compute()
Out[36]:
[14.75051171665384,
 15.463557844570461,
 15.790076907851297,
 15.971334410669527,
 16.477159899324676,
 16.250654434978838,
 16.163639508987067,
 16.164026987891997,
 16.364647910506154,
 16.544750841370114,
 16.385807916489675,
 16.28056690958003]
```

# Reductions (Aggregations)

```
In [37]: t_sum, t_min, t_max, = totals.sum(), totals.min(), totals.max()

In [38]: t_mean, t_std, = totals.mean(), totals.std()

In [39]: stats = [t_sum, t_min, t_max, t_mean, t_std]

In [40]: %time [s.compute() for s in stats]
CPU times: user 142 ms, sys: 101 ms, total: 243 ms
Wall time: 4.57 s
Out[40]:
[13665876.250009943,
 915174.880000469,
 1354966.290001166,
 1138823.0208341617,
 144025.81874405374]

In [41]: import dask

In [42]: %time dask.compute(t_sum, t_min, t_max, t_mean, t_std)
CPU times: user 63.7 ms, sys: 29.1 ms, total: 92.7 ms
Wall time: 852 ms
Out[42]:
(13665876.250009943,
 915174.880000469,
 1354966.290001166,
 1138823.0208341617,
 144025.81874405374)
```

PARALLEL COMPUTING WITH DASK

# Let's practice!

PARALLEL COMPUTING WITH DASK

# Analyzing Congressional Legislation

Dhavide Aruliah
Director of Training, Anaconda

# JSON data files

- **J**ava**S**cript **O**bject **N**otation:
  - stored as plain text
  - common web format
  - direct mapping to Python lists & dictionaries

# Sample JSON FIle: items.json

*items.json*

```
[
  {
    "name": "item1",
    "content": ["a","b","c"]
  },
  {
    "name": "item2",
    "content": {"a": 0, "b": 1}
  }
]
```

# Using json Module

```
In [1]: import json

In [2]: with open('items.json') as f:
   ...:     items = json.load(f)

In [3]: type(items)
Out[3]: list

In [4]: items[0]
Out[4]: {'content': ['a', 'b', 'c'], 'name': 'item1'}

In [5]: items[1]
Out[5]: {'content': {'a': 0, 'b': 1}, 'name': 'item2'}

In [6]: items[1]['content']['b']
Out[6]: 1
```

# JSON Files into Dask Bags

*items-by-line.json*

```
{"name": "item1", "content": ["a", "b", "c"]}
{"name": "item2", "content": {"a": 0, "b": 1}}
```

```
In [7]: import dask.bag as db

In [8]: items = db.read_text('items-by-line.json')

In [9]: items.take(1)  # Note: tuple containing a *string*
Out[9]: ('{"name": "item1", "content": ["a", "b", "c"]}\n',)

In [10]: dict_items = items.map(json.loads) # converts strings -> other data

In [11]: dict_items.take(2) # Note: tuple containing dicts
Out[11]:
({'content': ['a', 'b', 'c'], 'name': 'item1'},
 {'content': {'a': 0, 'b': 1}, 'name': 'item2'})
```

# Plucking Values

```
In [12]: type(dict_items.take(2))
Out[12]: tuple

In [13]: dict_items.take(2)[1]['content'] # Chained indexing
Out[14]: {'a': 0, 'b': 1}

In [14]: dict_items.take(1)[0]['name']    # Chained indexing
Out[14]: 'item1'

In [15]: contents = dict_items.pluck('content')

In [16]: names = dict_items.pluck('name')

In [17]: contents
Out[17]: dask.bag<pluck-5..., npartitions=1>

In [18]: names
Out[18]: dask.bag<pluck-3..., npartitions=1>

In [19]: contents.compute()
Out[19]: [['a', 'b', 'c'], {'a': 0, 'b': 1}]

In [20]: names.compute()
Out[20]: ['item1', 'item2']
```

# Congressional Legislation Metadata

- 23 JSON files
  - metadata about congressional bills
  - up to 1500 pieces of legislation per congress.
- Load *all* into Dask Bag
  - use `current_status` to count vetoed bills
  - use date info to compute average times

# Metadata Keys

- Selected dictionary keys

```
'bill_type'
'title_without_number'
'related_bills'
'id'
'titles'
'display_number'
'major_actions'
'current_status_description'
'link'
'current_status_date'
'committee_reports'
'current_status_label'
'introduced_date'
'sponsor'
'current_status'
'title'
```

- **Warning**: Not all available for every bill

PARALLEL COMPUTING WITH DASK

# Let's practice!