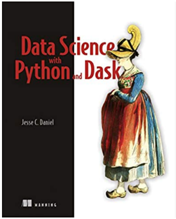
UNIVERSITY OF MARYLAND

UMD DATA605 - Big Data Systems

# 9.3: Python Dask

- **Instructor**: Dr. GP Saggese, gsaggese@umd.edu
- **Resources**
  - Web resources:
    - Dask project
    - Dask examples
  - Tutorial
    - Dask_tutorial
    - Dask_advanced_tutorial
  - Class project
  - Mastery
    - Data science with Python and Dask, 2019



# Dataset Size Issues

- **Small datasets ($< 1$ GB)**
  - Fits into RAM
  - No disk paging needed
- **Medium dataset ($< 1$TB)**
  - Doesn't fit into RAM
  - Fits into local disk
    - Performance penalty with local disk
  - Need multiple CPU cores
    - Difficult to leverage parallelism with Python/Pandas
- **Large dataset ($> 1$TB)**
  - Doesn't fit into RAM
  - Doesn't fit into local disk
  - Need multiple servers
    - Python/Pandas not built for distributed datasets
    - Use frameworks for massive datasets
    - E.g., Hadoop, Spark, Dask, Ray

# Dataset Size Issues

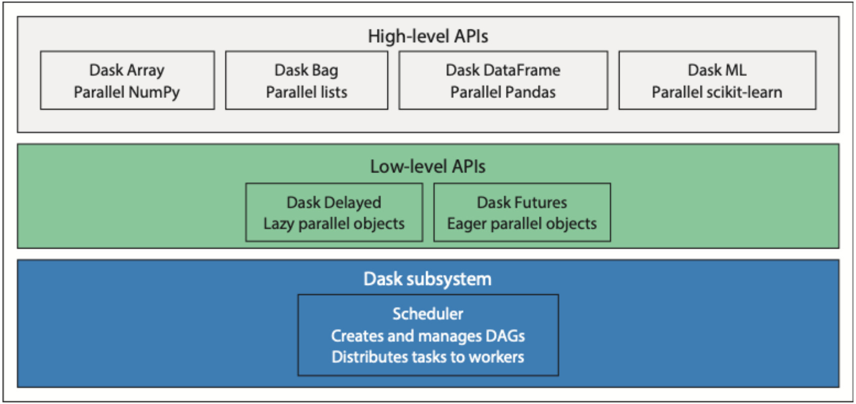
| Category | Size |
|---|---|
| Small datasets | $< 1$ GB |
| Medium datasets | $< 1$ TB |
| Large datasets | $> 1$ TB |

- **The thresholds are fuzzy and changing over time**
  - Scale computer 10x to get 10x bigger datasets
- **Problem with scaling datasets**
  - Long run times
  - Rewriting code for different dataset sizes
  - Plan what and how to do efficiently
  - Cumbersome framework (Pandas easy, Hadoop difficult)

# Dask

- **Dask is written in Python**
  - Scales Numpy, Pandas, sklearn
  - Dask objects wrap library objects (e.g., Pandas DataFrame, numpy array)
  - Parallel parts are “chunks” or “partitions”
    - Queued for work
    - Shipped between machines
    - Worked locally
- **Pros**
  - Use familiar interfaces
  - Write code optimized for parallelism
    - Dask handles heavy lifting
- **Scaling Dask is easy**
  - Prototype on local machine, use cluster when needed
  - No code refactoring needed
  - No cluster-specific issues
    - E.g., resource management, data recovery, data movement
  - Runs on multi-core
  - Uses cluster managers
    - E.g., Yarn, Mesos, Kubernetes, AWS ECS

# Dask Layers



High-level APIs
Dask Array
Parallel NumPy
Dask Bag
Parallel lists
Dask DataFrame
Parallel Pandas
Dask ML
Parallel scikit-learn
Low-level APIs
Dask Delayed
Lazy parallel objects
Dask Futures
Eager parallel objects
Dask subsystem
Scheduler
Creates and manages DAGs
Distributes tasks to workers

# Scaling Up vs Scaling Out

- **Scaling up**
  - Replace equipment with larger, faster options
    - E.g., buy a larger pot, replace knife with food processor
  - **Pros**
    - Better hardware, no code changes needed
  - **Cons**
    - Exceed current machine capacity eventually
    - Cost: more powerful machines are expensive


- **Scaling out**
  - Divide work between many workers in parallel
    - E.g., buy more pots and hire more cooks
  - **Pros**
    - Task scheduler organizes computation, assigns workers to tasks
    - Cost-effective, no specialized hardware needed
  - **Cons**
    - Write code to expose parallelism
    - Maintain cluster costs

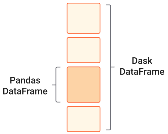
# Dask: Computation

- **Lazy computations**
  - Define transformations on data
  - Define next computation without waiting
  - Operate in chunks to avoid loading entire data in memory
  - E.g.,
    - Split 2GB file into 32 64MB chunks
    - Operate on 8 chunks per server
    - Max memory use: 512MB = (8 x 64MB)
  - Track object dimensions and data types
    - No code execution
- `compute()`
  - Run computation (materialize)

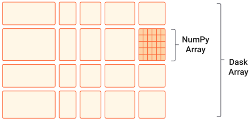
    `missing_count_pct = missing_count.compute()`
- `persist()`
  - Discard intermediate work to minimize memory
  - Re-run graph for additional computation on intermediate nodes
  - Keep intermediate result in memory
  - Speed up large, complex DAGs for reuse

# Dask: Data Structures

- **Dask DataFrame**
  - Implements Pandas DataFrame
  - Tabular/relational data




- **Dask Array**
  - Implements numpy ndarray
  - Multidimensional array



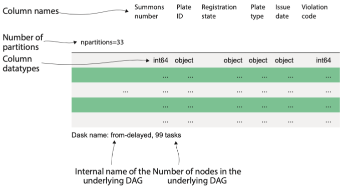

- **Dask Bag**
  - Coordinates Python lists of objects
  - Parallelize computations on unstructured/semi-structured data

```
[1, 2, 3, 4, 5]
[1, 2, 3] [4, 5]
```

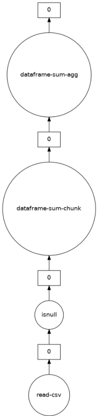
# Dask Reading Data

- Consider:

```python
import dask.dataframe as dd
df = dd.read_csv('nyc-parking-tickets-2017.csv')
missing_values = df.isnull().sum()
missing_values
```



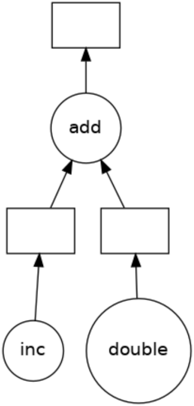

- `dask.dataframe.read_csv()`
  - Doesn't load data in memory
  - Infers column types
    - Samples data
    - Set data types
    - Use Parquet for data and types together
- Partitions = independent data chunks
  - E.g., 33 partitions
  - Graph = 99 tasks
  - Each partition reads, splits data, initializes df object



# Low Level APIs: Delayed

- Handle computations that don't fit in native Dask data structures
  - E.g., Dask DataFrame
- In the example below there is parallelism that can be exploited

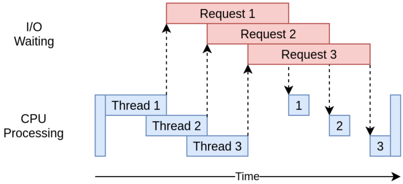
```python
def inc(x):
    return x + 1

def double(x):
    return x * 2

def add(x, y):
    return x + y

data = [1, 2, 3, 4, 5]

output = []
for x in data:
    # (x + 1) + (x * 2) = 3x + 1
    a = inc(x)
    b = double(x)
    c = add(a, b)
    # 1 -> 4
    # 2 -> 7
    # 3 -> 10
    # 4 -> 13
    # 5 -> 16
    output.append(c)

# 4 + 7 + 10 + 13 + 16 = 20 + 20 + 10 = 50
total = sum(output)
print(total)
```




SCIENCE ACADEMY

# Low Level APIs: Futures

- In parallel programming, a "future" encapsulates asynchronous execution, representing the eventual result
- Python `concurrent.futures`
  - High-level interface for asynchronous execution
  - Thread pool or Process pool (`Executor` interface)
- Dask extends `concurrent.futures`
  - Express everything as futures
  - Specify blocking and non-blocking




```python
def inc(x):
    return x + 1

def add(x, y):
    return x + y

a = client.submit(inc, 10)
b = client.submit(inc, 20)
```

```
>>> a
<Future: status: pending, key: inc-b8aaf26b99466a7a
```

```
>>> a
<Future: status: finished, type: int, key: inc-b8aaf26b9946

>>> a.result()  # blocks until task completes and data arri
11
```

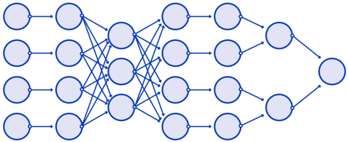
SCIENCE ACADEMY

# Different Types of Parallel Workload

- Break program in medium-size tasks of computation
  - E.g., a function call

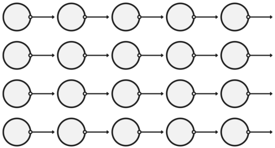
**MapReduce**

Hadoop/Spark/Dask



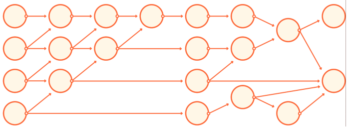
**Embarrassingly Parallel**

Hadoop/Spark/Dask/Airflow/Prefect



**Full Task Scheduling**

Dask/Airflow/Prefect



# Encoding Task Graph

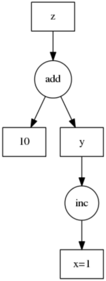
- Dask encodes tasks in terms of Python dicts and functions

```python
def inc(i):
    return i + 1

def add(a, b):
    return a + b

x = 1
y = inc(x)
z = add(y, 10)
```

→

```python
d = {'x': 1,
     'y': (inc, 'x'),
     'z': (add, 'y', 10)}
```

→



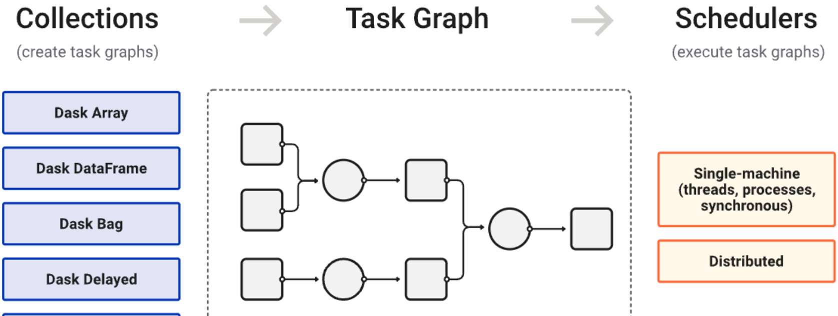

```python
import dask.dataframe as dd

df = dd.read_csv('myfile.*.csv')
df = df + 100
df = df[df.name == 'Alice']
```

→

```python
{
 # From the dask.dataframe.read_csv call
 ('read-csv', 0): (pandas.read_csv, 'myfile.0.csv'),
 ('read-csv', 1): (pandas.read_csv, 'myfile.1.csv'),
 ('read-csv', 2): (pandas.read_csv, 'myfile.2.csv'),
 ('read-csv', 3): (pandas.read_csv, 'myfile.3.csv'),

 # From the df + 100 call
 ('add', 0): (operator.add, ('read-csv', 0), 100),
 ('add', 1): (operator.add, ('read-csv', 1), 100),
 ('add', 2): (operator.add, ('read-csv', 2), 100),
 ('add', 3): (operator.add, ('read-csv', 3), 100),

 # From the df[df.name == 'Alice'] call
 ('filter', 0): (lambda part: part[part.name == 'Alice'], ('add', 0)),
 ('filter', 1): (lambda part: part[part.name == 'Alice'], ('add', 1)),
 ('filter', 2): (lambda part: part[part.name == 'Alice'], ('add', 2)),
 ('filter', 3): (lambda part: part[part.name == 'Alice'], ('add', 3)),
}
```

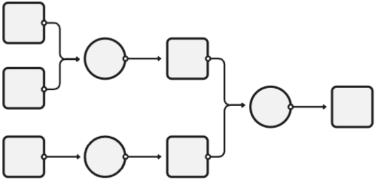
# Task Scheduling

- Data collections (Bags, Arrays, DataFrame) and operations create task graphs
  - Nodes: Python functions
  - Edges: Dependencies (output from one task used as input in another)
- Schedule task graphs for execution
  - Single-machine scheduler
    - Use local process or thread pool
    - Runs on a single machine
  - Distributed scheduler
    - Runs locally or across a cluster



# Task Scheduling

- **Dask task scheduler orchestrates work dynamically**
  - Not static scheduling like a relational DB
  - During computation, Dask dynamically assesses:
    - Completed tasks
    - Remaining tasks
    - Free resources (CPUs)
    - Data location
- **Dynamic approach handles various issues:**
  - Worker failure
    - Re-run tasks
  - Workers completing at different speeds due to:
    - Different computation
    - Different hardware
    - Varying server workloads
    - Slower data access
  - Network unreliability
    - Re-run or remove isolated nodes



# Dask vs Spark

- **Pros**
  - Popular framework for large datasets
  - In-memory alternative to MapReduce/Hadoop
- **Cons**
  - Java library, supports Python via PySpark API
    - Python code runs on JVM
    - Debugging is difficult as execution is outside Python
  - Different DataFrame API than Pandas
    - Learn "the Spark way"
    - May need to implement twice for exploratory analysis and production
  - Optimized for MapReduce operations
  - Difficult to set up and configure