CSE 4510/5310
BIG DATA

Instructor: Fitzroy Nembhard, Ph.D.

Week 5

Dask:
Beyond Pandas

FLORIDA TECH
FLORIDA'S STEM UNIVERSITY®

Distribution

- All slides included in this class are for the exclusive use of students and instructors associated with Database Systems (CSE 4020/5260) at the Florida Institute of Technology
- Redistribution of the slides is not permitted without the written consent of the author.

Goals

- To discuss limitations of using Pandas with large datasets
- To discuss Dask & Directed Acyclic Graphs (DAGS)
- To discuss Dask DataFrames and their limitations
- To apply a data processing workflow to solve a problem in Dask
  - Problem Definition
  - Data Gathering
  - Data Cleaning
  - Exploratory Analysis
  - Hypothesis Formulation & Testing
  - Model Building & Testing
  - Model Deployment & Monitoring

5.1 BEYOND PANDAS - DASK

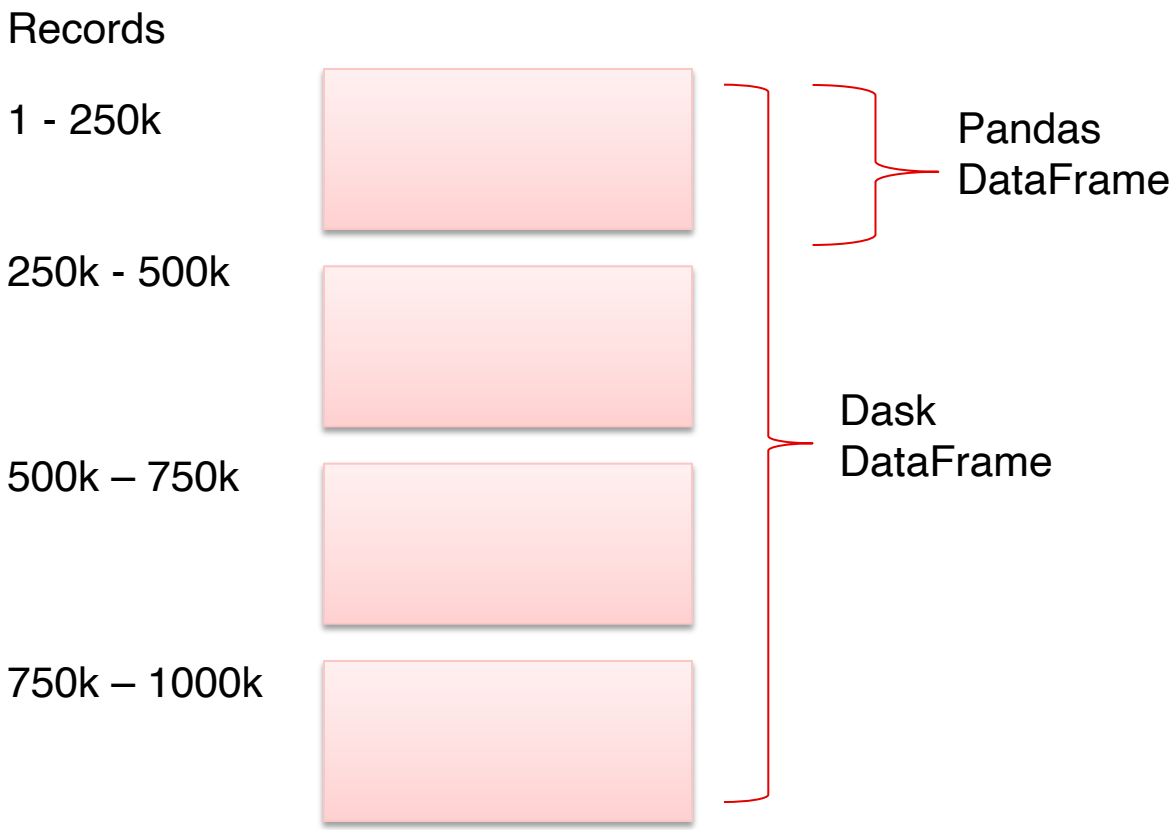
Pandas and Large Data

- Pandas is very efficient with small data (usually from 100MB up to 1GB), and performance is rarely a concern.
- It will work for relatively large data using chunks

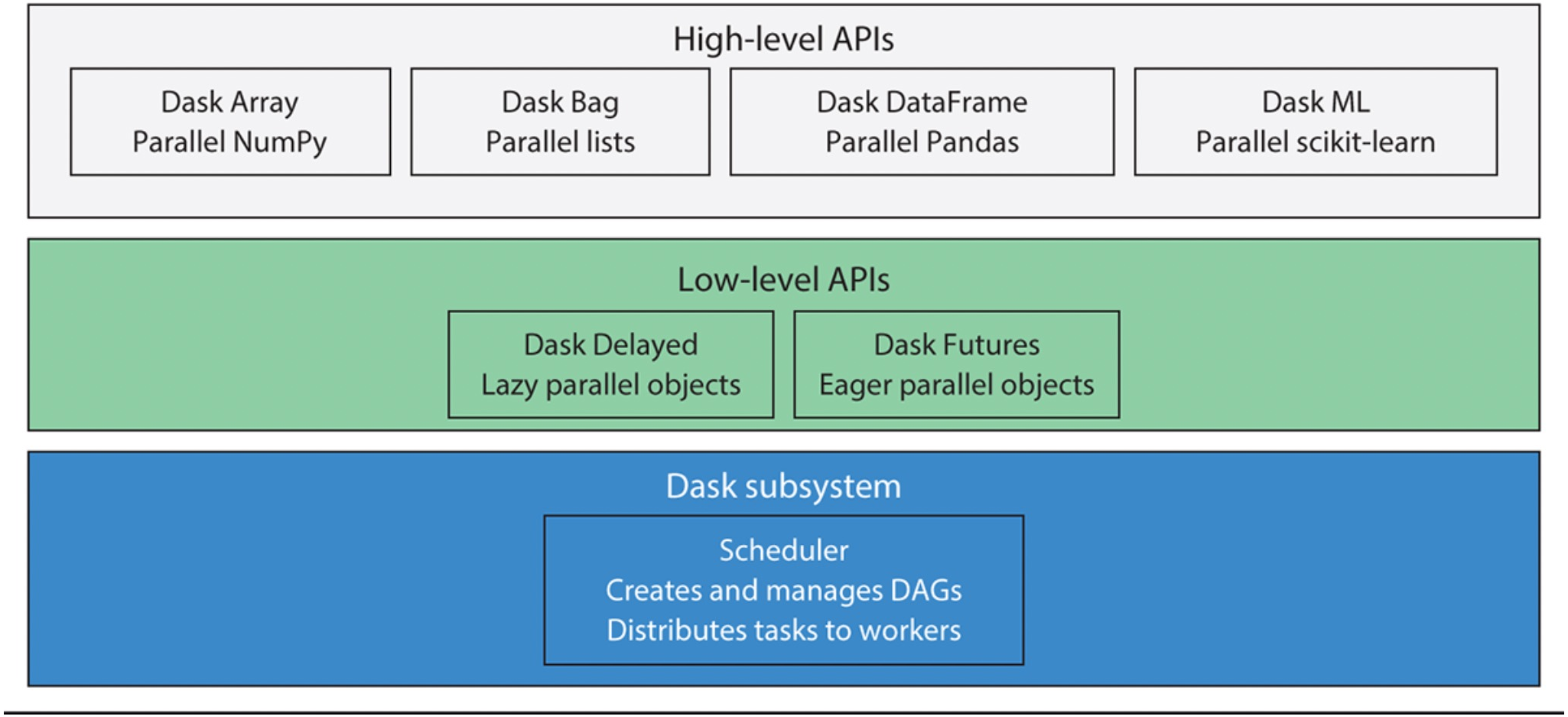
```
# read the large csv file with specified chunksize
df_chunk = pd.read_csv(r'my_dataset.csv', chunksize=1000000) # df_chunk is a TextFileReader

chunk_filter = chunk_preprocessing(chunk)

# Once the data filtering is done, append the chunk to list
chunk_list.append(chunk_filter)

# concat the list into dataframe
df_concat = pd.concat(chunk_list)

# if the dataset has many unnecessary columns, you can then select those of interest
df[['col_1','col_2', 'col_3', 'col_4', 'col_5']] =  df[['col_1','col_2', 'col_3', 'col_4', 'col_5']].astype('int32')
```

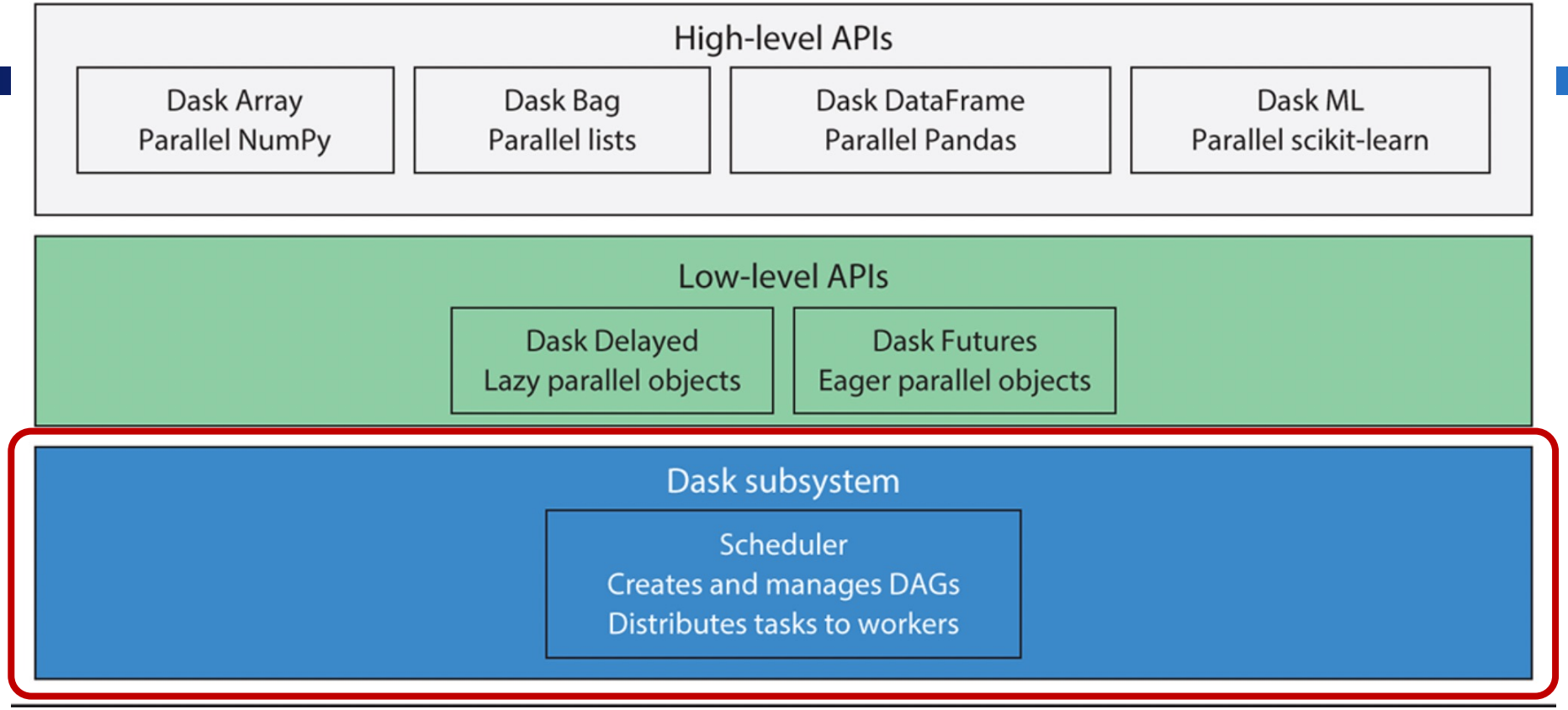
Beyond Pandas - Dask

- When you have more data that's way larger than your local RAM (say 100GB), you can either still use Pandas with chunks to a certain extent or choose a better tool
- Dask may be used to handle large datasets
- Dask is popularly known as a Python's parallel computing library
- Think of Dask as an extension of Pandas in terms of performance and scalability
- Dask will create a DataFrame that is row-wise partitioned (i.e. rows are grouped by an index value)



Dask Components and Layers



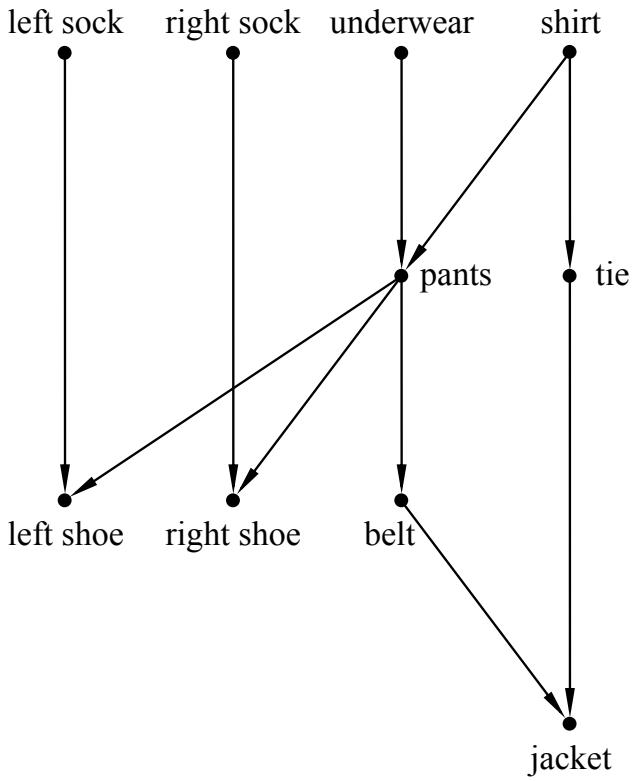


DASK & DIRECTED ACYCLIC GRAPHS (DAGS)

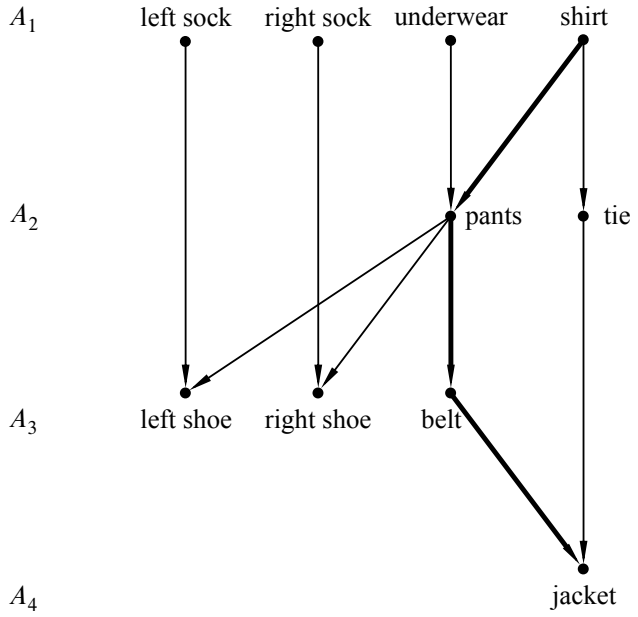
Directed Acyclic Graphs (DAGS)

- A **directed acyclic graph** (DAG) is a directed graph with no cycles.
- A **topological sort** of a finite DAG is a list of all the vertices such that each vertex v appears earlier in the list than every other vertex reachable from v.

DAG Example 1: Getting Dressed



underwear	left sock
shirt	shirt
pants	tie
belt	underwear
tie	right sock
jacket	pants
left sock	right shoe
right sock	belt
left shoe	jacket
right shoe	left shoe
(a)	(b)



a) Getting Dressed DAG

b) 2 Examples of topological sort of the DAG.

c) Example parallel schedule

DAG Example 2: Bucatini all'Amatriciana

BUCATINI ALL'AMATRICIANA

Ingredients
(Serves 4)

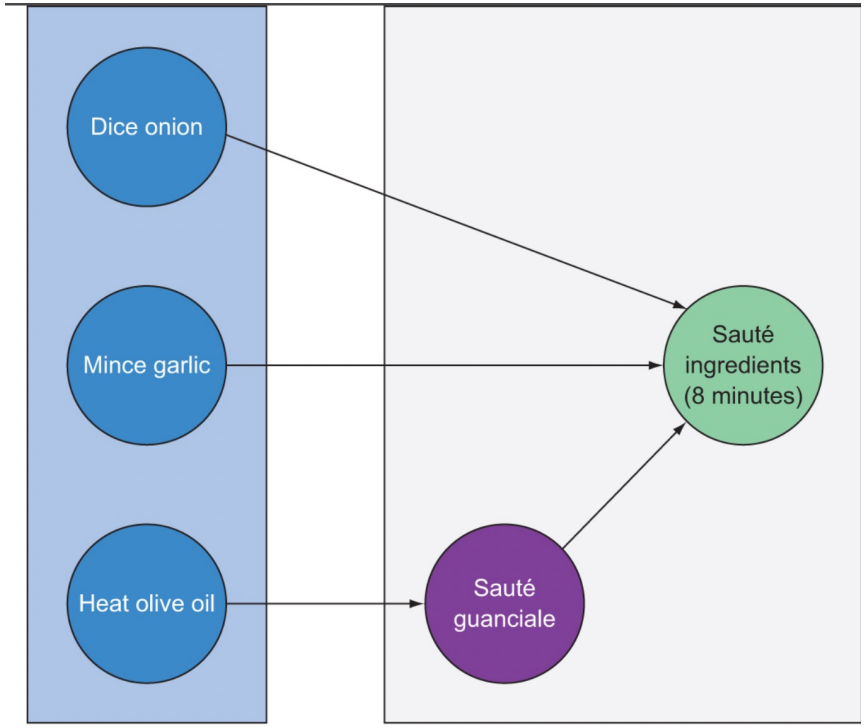
2 Tbsp. olive oil
3/4 cup diced onion
2 cloves garlic, minced
4 oz. guanciale, sliced into small pieces
1 28-oz. can San Marzano tomatoes, crushed
Kosher salt
1/2 tsp. red chili flakes
1/2 tsp. freshly ground black pepper
1 lb dried bucatini
1 oz pecorino romano, grated

Instructions

1. Heat oil in a large heavy skillet over medium heat. Add guanciale and fry until crispy (approximately 4 minutes). Add chili flakes, pepper, onion, and garlic. Cook until the onions have softened (approximately 8 minutes). Stir often to avoid burning. Add tomatoes and simmer on low heat until sauce thickens (approximately 15–20 minutes).

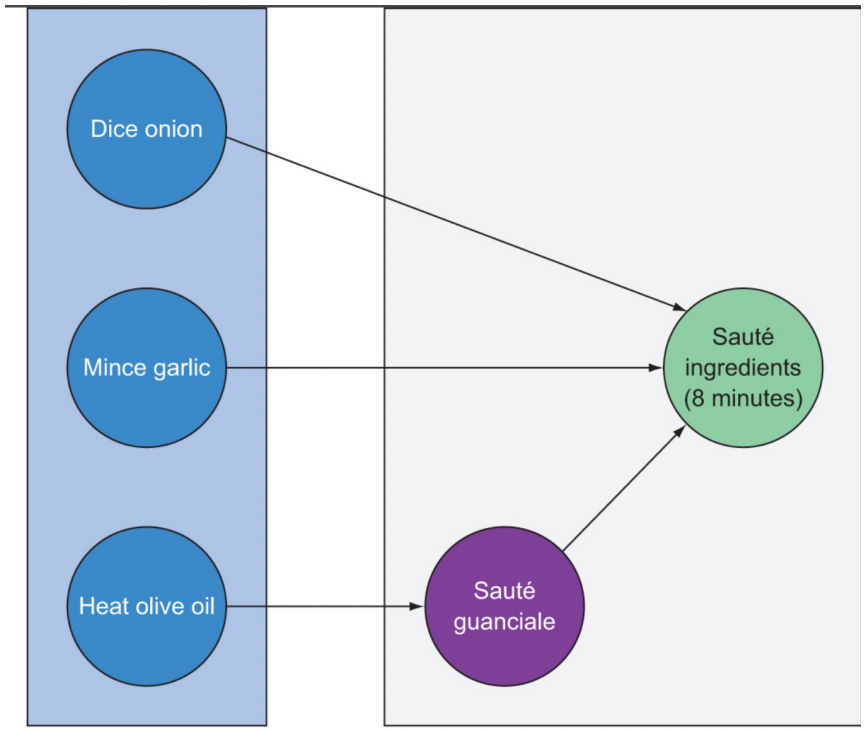
2. While the sauce is simmering, bring a large pot of salted water to the boil. Add the pasta and cook for 1–2 minutes less than the cooking time listed on the package. When finished, retain some pasta water (about 1 cup) and drain the pasta into a colander.

3. When the sauce has thickened, add the cooked pasta to the skillet and toss to combine the sauce and noodles. Add half of the retained pasta water and cook until the pasta is al dente (approximately 2 minutes). Stir in grated cheese and serve. Buon appetito!



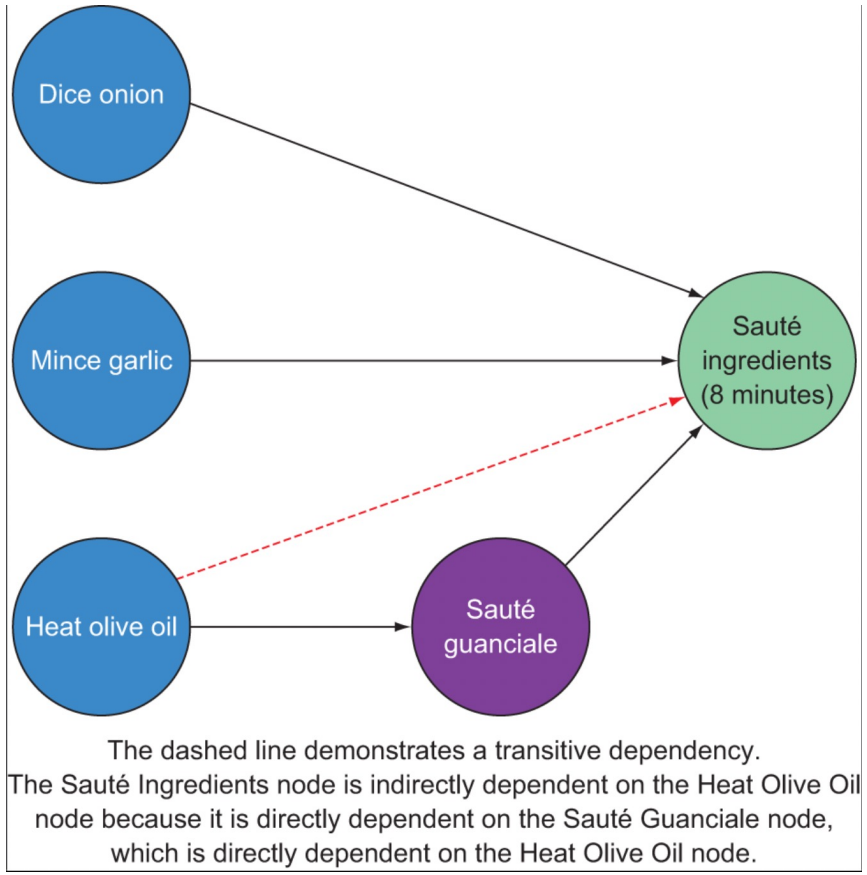
"A graph displaying nodes with dependencies"

DAG Example 2: Bucatini all'Amatriciana



No dependencies.
These tasks can be started in any order.

These tasks can only be started when all nodes connected to them have been completed.



The dashed line demonstrates a transitive dependency.
The Sauté Ingredients node is indirectly dependent on the Heat Olive Oil node because it is directly dependent on the Sauté Guanciale node, which is directly dependent on the Heat Olive Oil node.

DAG Example 2: Bucatini all'Amatriciana

BUCATINI ALL'AMATRICIANA

Ingredients
(Serves 4)

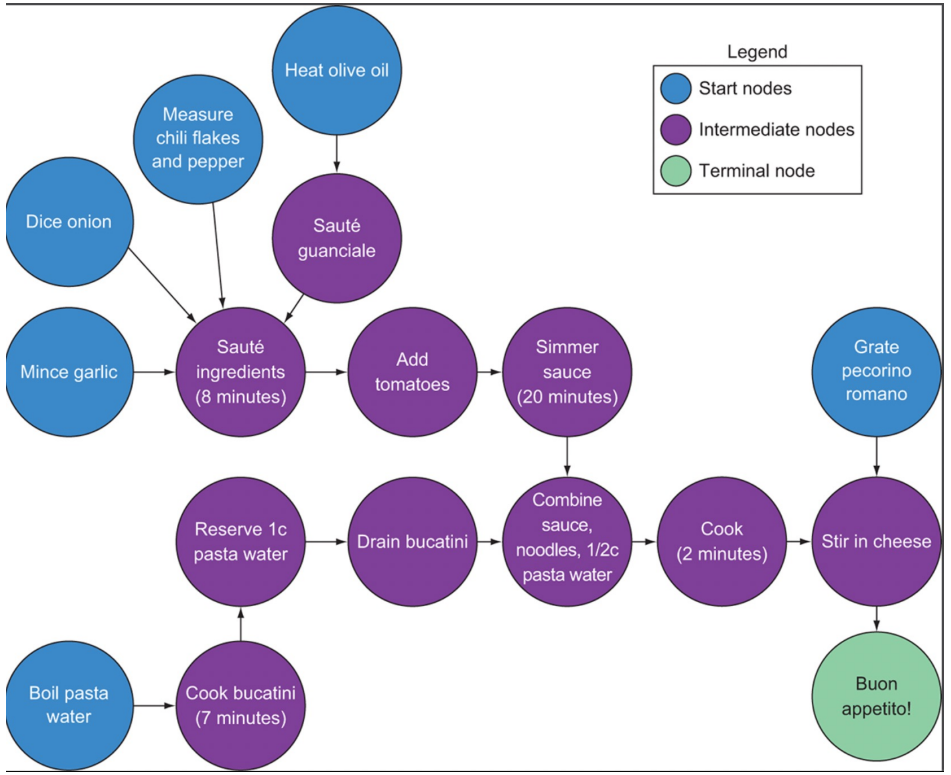
2 Tbsp. olive oil
3/4 cup diced onion
2 cloves garlic, minced
4 oz. guanciale, sliced into small pieces
1 28-oz. can San Marzano tomatoes, crushed
Kosher salt
1/2 tsp. red chili flakes
1/2 tsp. freshly ground black pepper
1 lb dried bucatini
1 oz pecorino romano, grated

Instructions

1. Heat oil in a large heavy skillet over medium heat. Add guanciale and fry until crispy (approximately 4 minutes). Add chili flakes, pepper, onion, and garlic. Cook until the onions have softened (approximately 8 minutes). Stir often to avoid burning. Add tomatoes and simmer on low heat until sauce thickens (approximately 15–20 minutes).

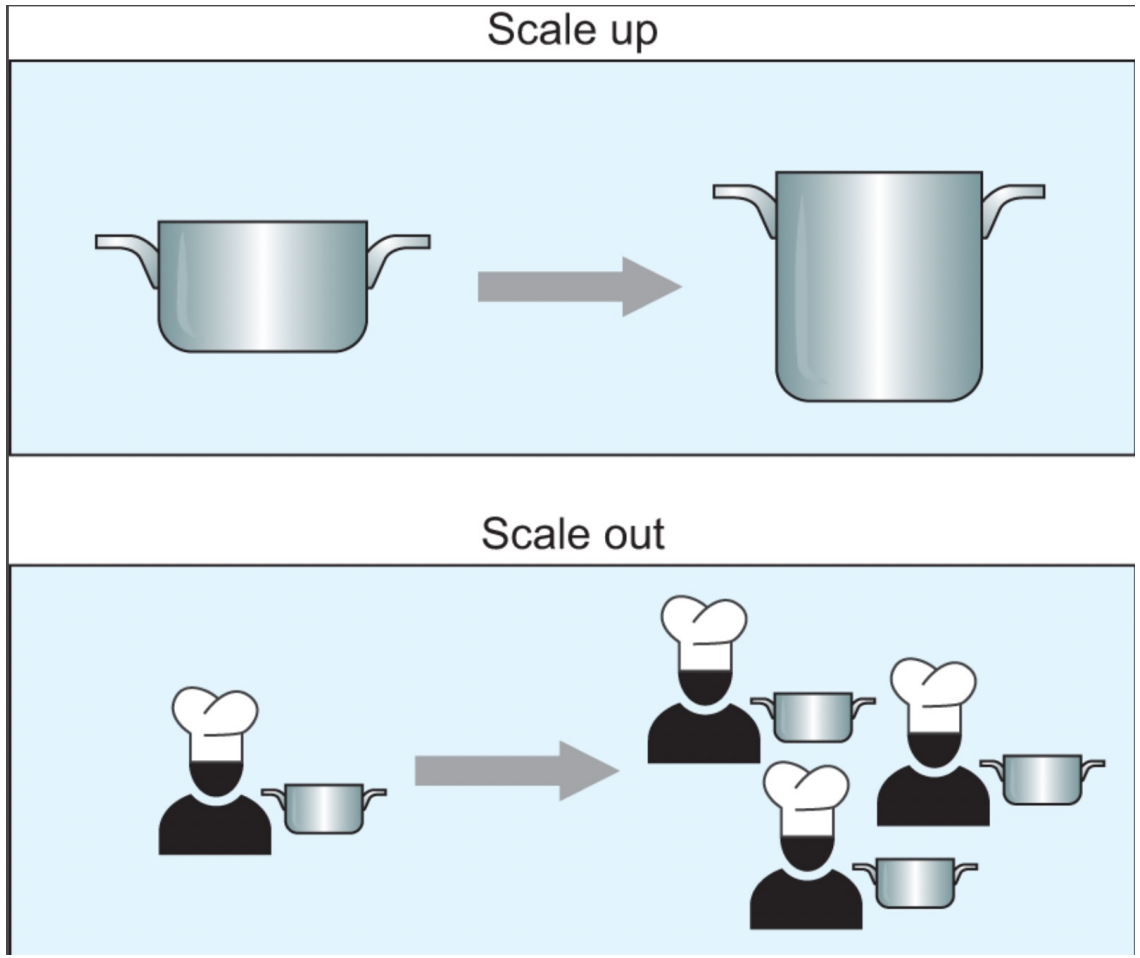
2. While the sauce is simmering, bring a large pot of salted water to the boil. Add the pasta and cook for 1–2 minutes less than the cooking time listed on the package. When finished, retain some pasta water (about 1 cup) and drain the pasta into a colander.

3. When the sauce has thickened, add the cooked pasta to the skillet and toss to combine the sauce and noodles. Add half of the retained pasta water and cook until the pasta is al dente (approximately 2 minutes). Stir in grated cheese and serve. Buon appetito!

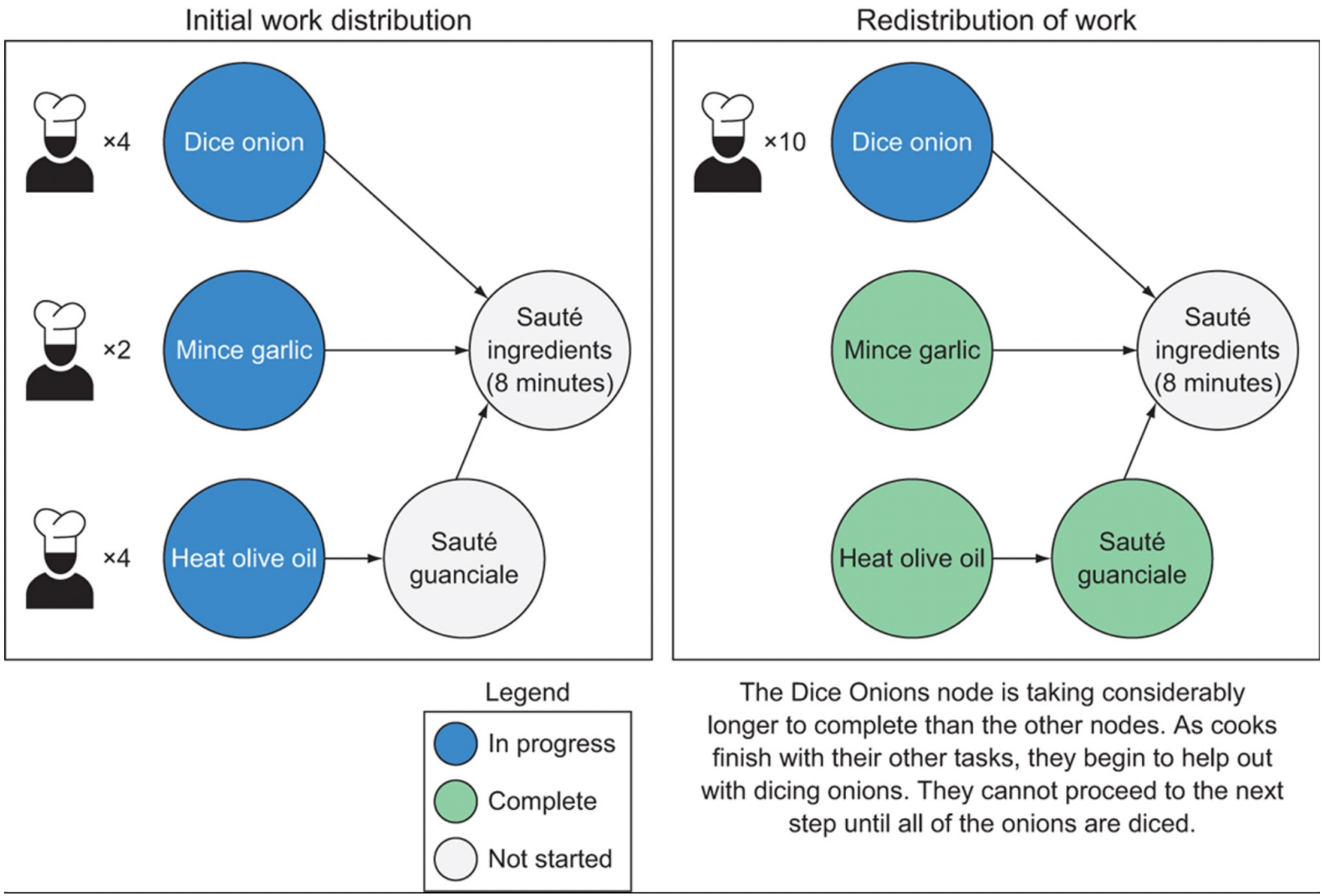


"The full directed acyclic graph representation of the bucatini all'Amatriciana recipe"

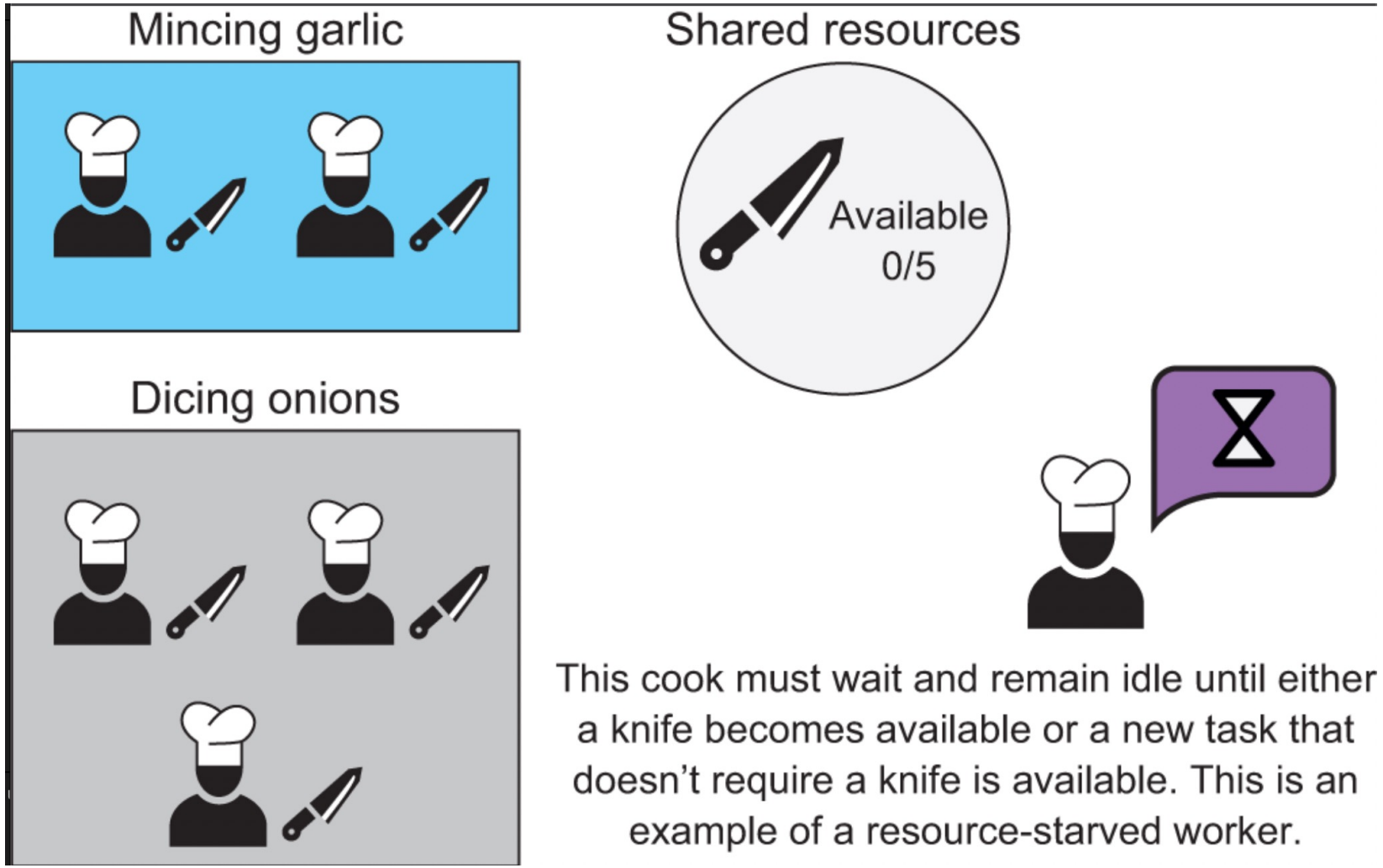
DAG Example 2: Bucatini all'Amatriciana



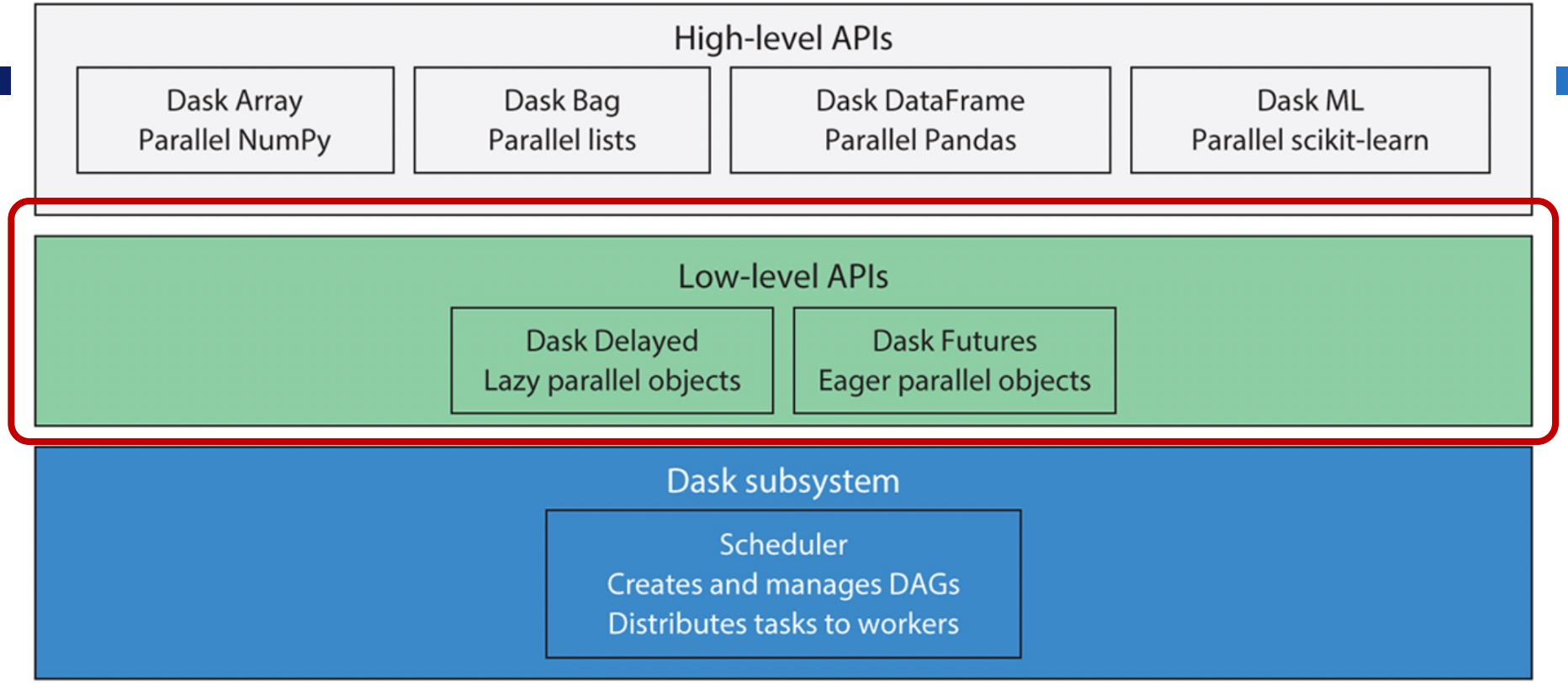
DAG Example 2: Bucatini all'Amatriciana



DAG Example 2: Bucatini all'Amatriciana



This cook must wait and remain idle until either a knife becomes available or a new task that doesn't require a knife is available. This is an example of a resource-starved worker.



DASK DAGS AND DELAYED

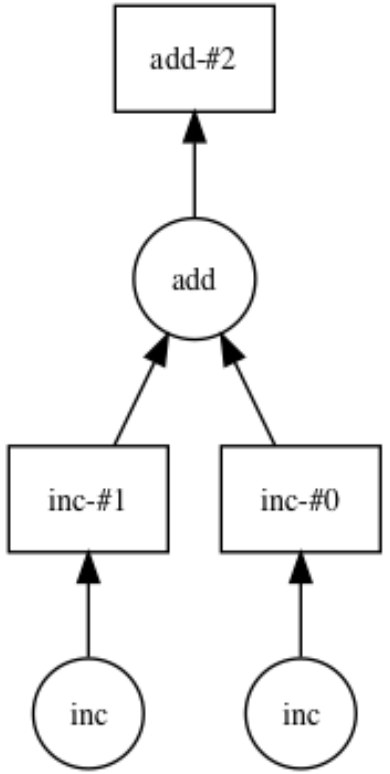
Packages to Install

- conda install -c conda-forge pyarrow
- conda install -c conda-forge dill
- conda install graphviz
- conda install python-graphviz

DAG Example 3

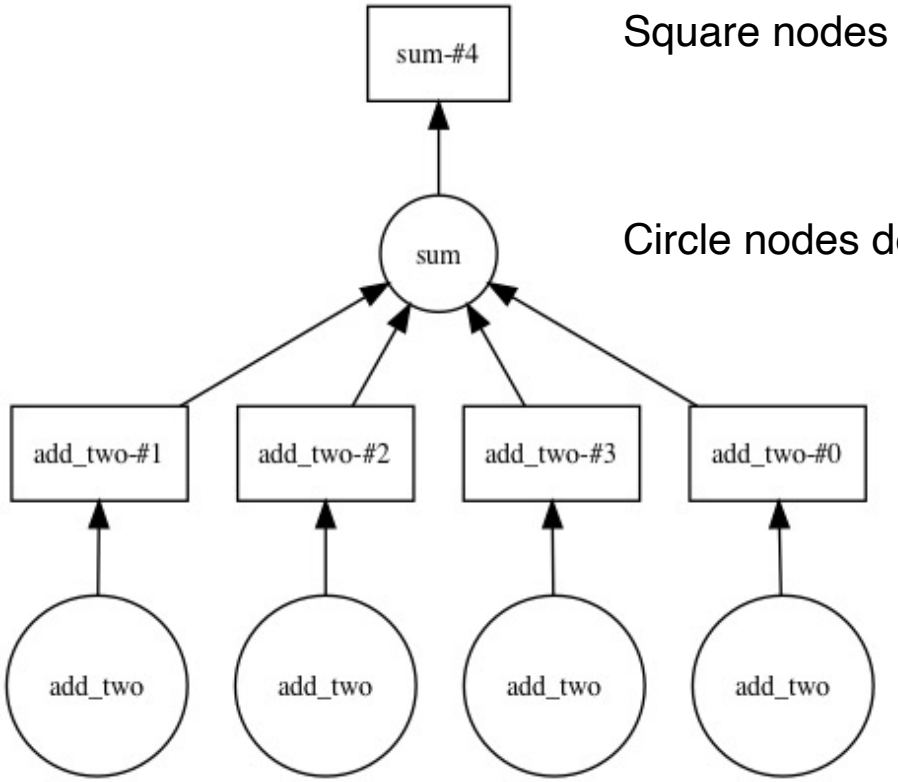
- By wrapping delayed around a function, a Dask Delayed representation of the function is produced.
- Delayed objects are equivalent to a node in a DAG.

```
import dask.delayed as delayed
from dask.diagnostics import ProgressBar

def inc(i):
    return i + 1

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

x = delayed(inc)(1)
y = delayed(inc)(2)
z = delayed(add)(x, y)
z.visualize()
```



DAG Example 4

```
def add_two(x):
    return x + 2
def sum_two_numbers(x,y):
    return x + y
def multiply_four(x):
    return x * 4
data = [1, 5, 8, 10]
step1 = [delayed(add_two)(i) for i in data]
total = delayed(sum)(step1)
total.visualize()
```

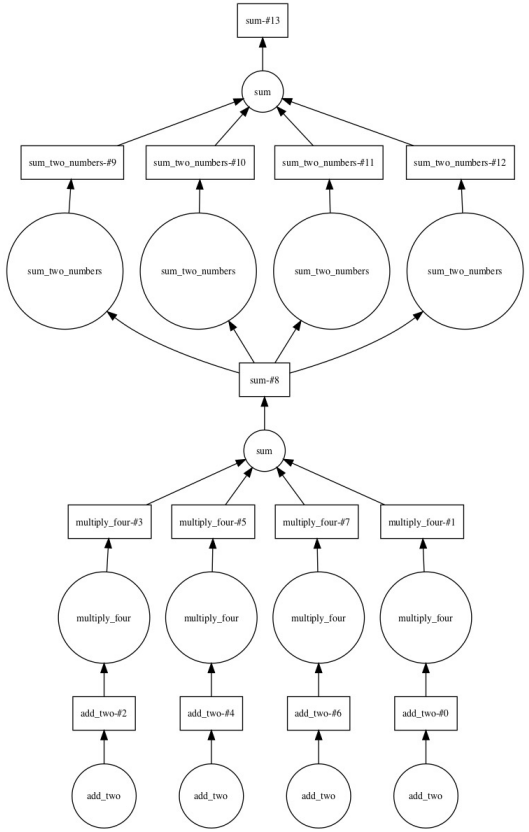


Square nodes denote intermediate results

Circle nodes denote functions/computations

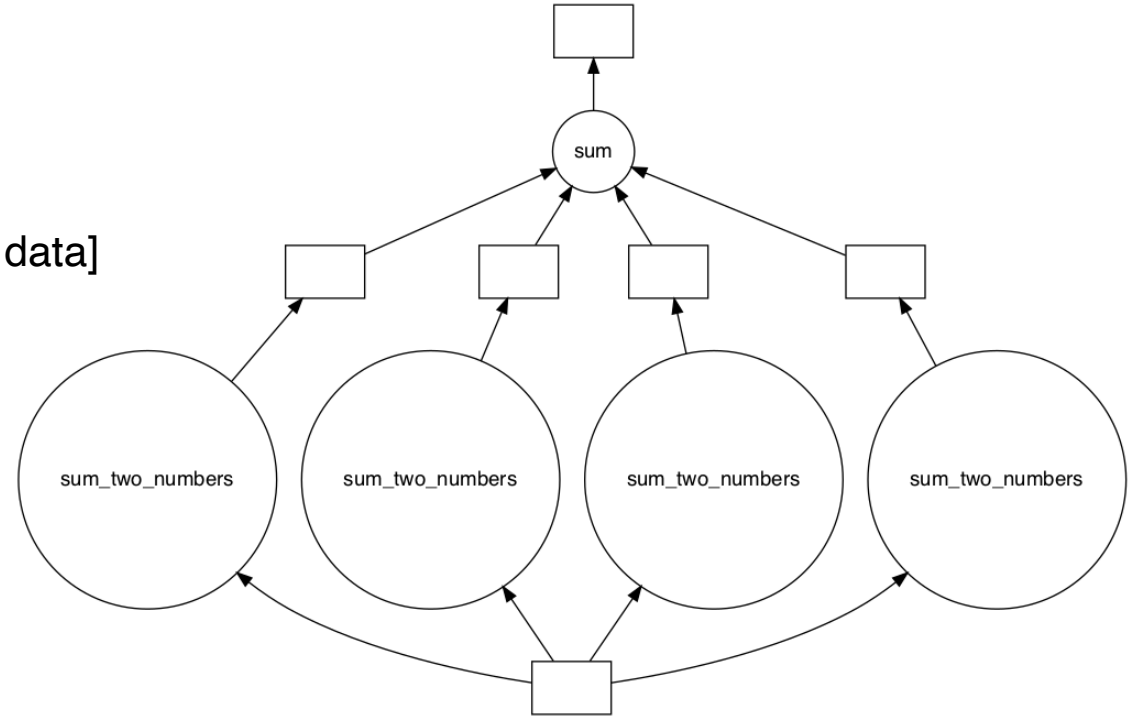
DAG Example 5

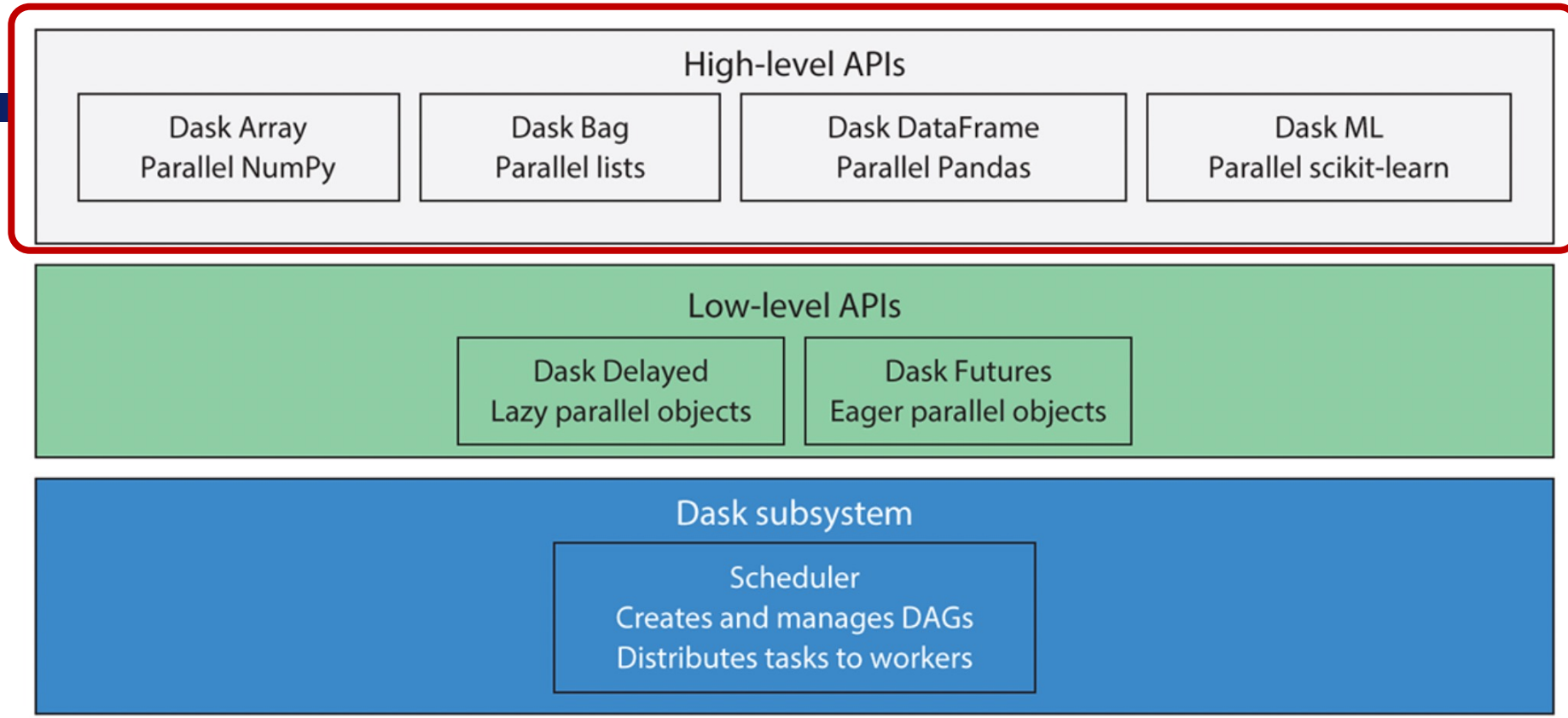
```
def add_two(x):
    return x + 2
def sum_two_numbers(x,y):
    return x + y
def multiply_four(x):
    return x * 4
data = [1, 5, 8, 10]
step1 = [delayed(add_two)(i) for i in data]
step2 = [delayed(multiply_four)(j) for j in step1]
total = delayed(sum)(step2)
total.visualize(verbose=False)
```



DAG Example 6

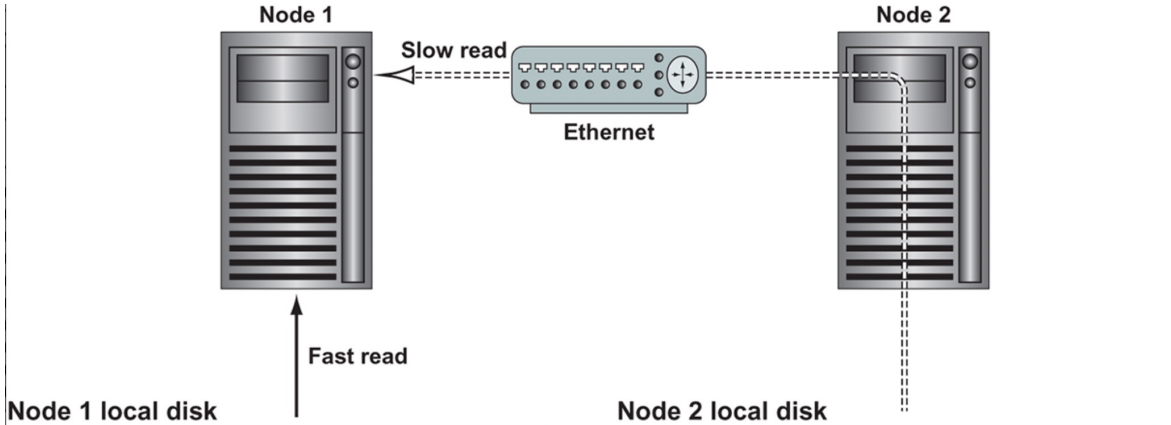
```
total_persisted = total.persist()
total_persisted.visualize(verbose=True)
data2 = [delayed(sum_two_numbers)(l, total_persisted) for l in data]
total2 = delayed(sum)(data2)
total2.visualize(optimize_graph=True)
```





DASK PARTITIONS/DATAFRAMES

Dask Partitions



Partition 1

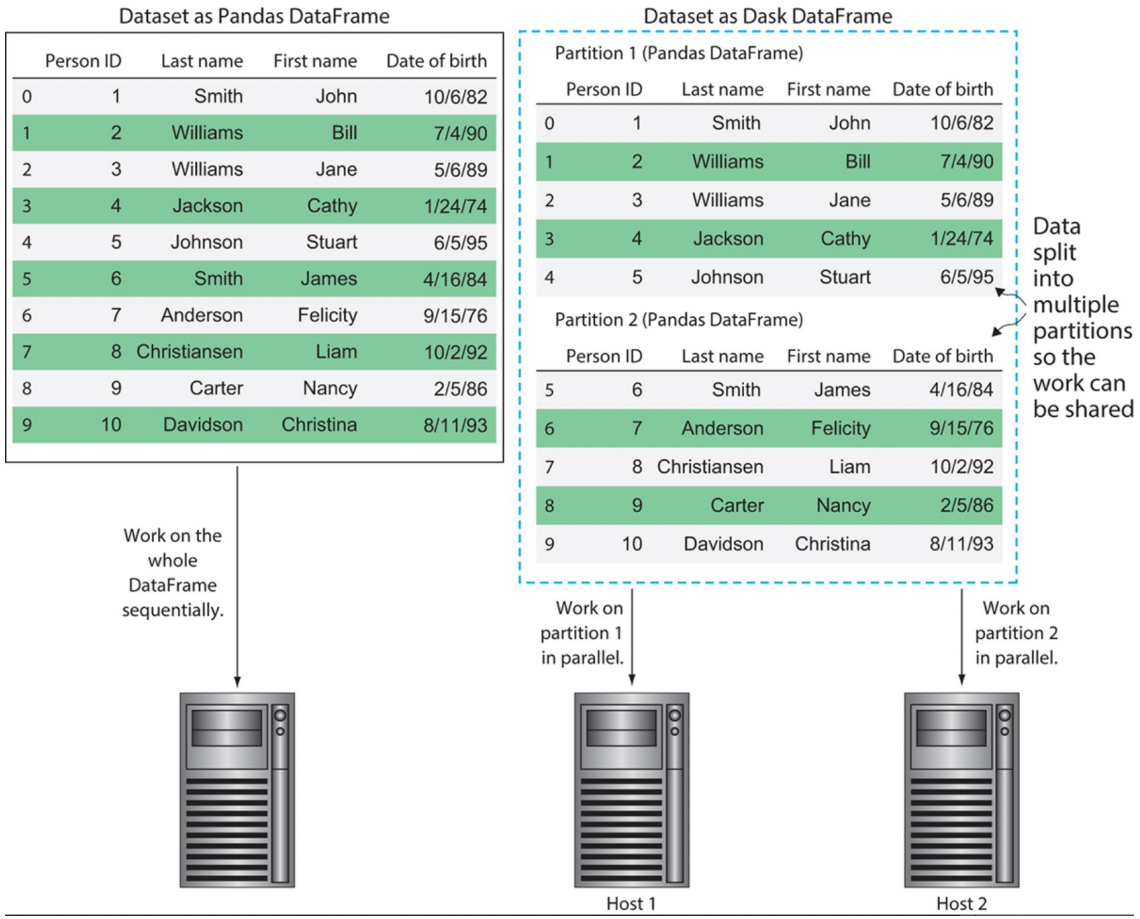
	Person ID	Last name	First name	Date of birth
0	1	Smith	John	10/6/82
1	2	Williams	Bill	7/4/90
2	3	Williams	Jane	5/6/89
3	4	Jackson	Cathy	1/24/74
4	5	Johnson	Stuart	6/5/95

Partition 1

	Person ID	Last name	First name	Date of birth
0	1	Smith	John	10/6/82
1	2	Williams	Bill	7/4/90
2	3	Williams	Jane	5/6/89
3	4	Jackson	Cathy	1/24/74
4	5	Johnson	Stuart	6/5/95

Dask Dataframes

- Dask breaks the file into smaller chunks that can be worked on independently.
- These chunks are called partitions.



Dask Dataframes

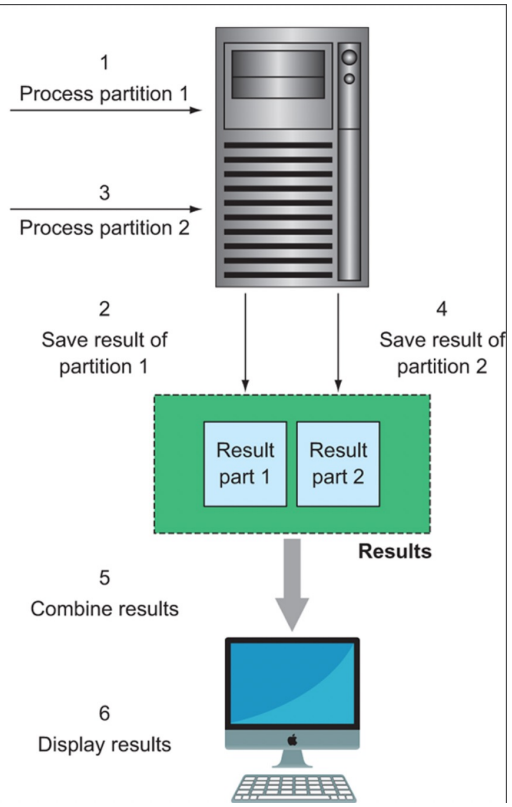
Dask DataFrame

Partition 1

	Person ID	Last name	First name	Date of birth
0	1	Smith	John	10/6/82
1	2	Williams	Bill	7/4/90
2	3	Williams	Jane	5/6/89
3	4	Jackson	Cathy	1/24/74
4	5	Johnson	Stuart	6/5/95

Partition 2

	Person ID	Last name	First name	Date of birth
5	6	Smith	James	4/16/84
6	7	Anderson	Felicity	9/15/76
7	8	Christiansen	Liam	10/2/92
8	9	Carter	Nancy	2/5/86
9	10	Davidson	Christina	8/11/93



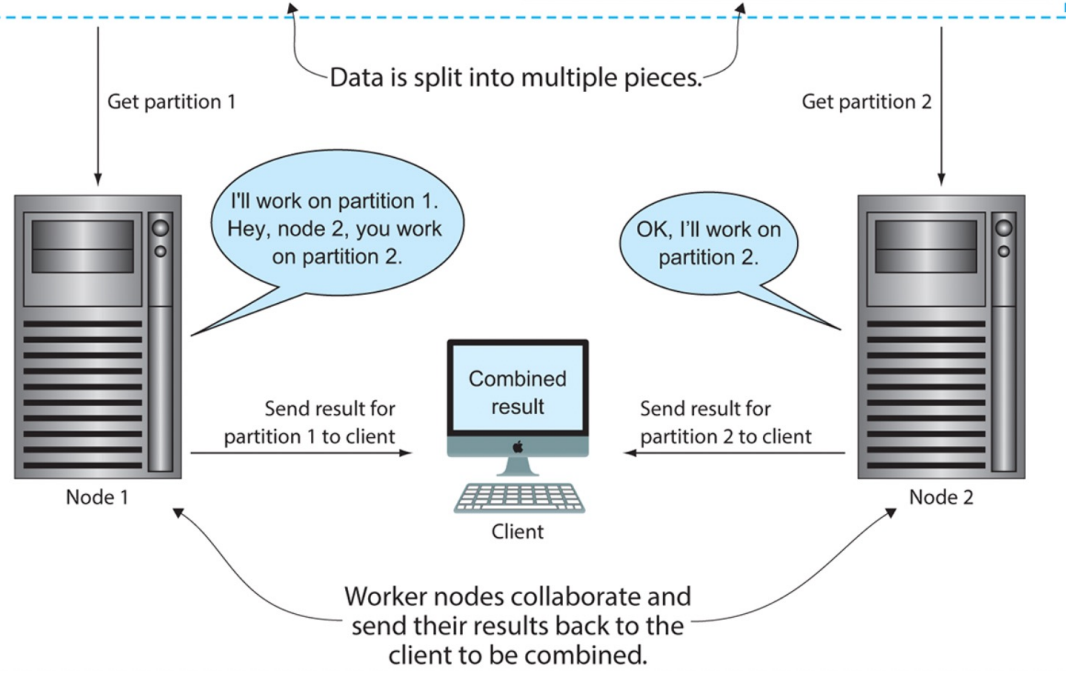
Dataset as Dask DataFrame

Partition 1 (Pandas DataFrame)

	Person ID	Last name	First name	Date of birth
0	1	Smith	John	10/6/82
1	2	Williams	Bill	7/4/90
2	3	Williams	Jane	5/6/89
3	4	Jackson	Cathy	1/24/74
4	5	Johnson	Stuart	6/5/95

Partition 2 (Pandas DataFrame)

	Person ID	Last name	First name	Date of birth
5	6	Smith	James	4/16/84
6	7	Anderson	Felicity	9/15/76
7	8	Christiansen	Liam	10/2/92
8	9	Carter	Nancy	2/5/86
9	10	Davidson	Christina	8/11/93



Dask DataFrames

```
import pandas as pd
import dask.dataframe as dd

person_ids = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
person_last_names = ['Smith', 'Williams',
                     'Williams', 'Jackson', 'Johnson', 'Smith', 'Anderson',
                     'Christiansen', 'Carter', 'Davidson']

person_first_names = ['John', 'Bill', 'Jane', 'Cathy', 'Stuart',
                      'James', 'Felicity', 'Liam', 'Nancy', 'Christina']

person_dobs = ['1982-10-06', '1990-07-04', '1989-05-06', '1974-01-24',
               '1995-06-05', '1984-04-16', '1976-09-15',
               '1992-10-02', '1986-02-05', '1993-08-11']

people_pandas_df = pd.DataFrame({'PersonID': person_ids,
                                 'Last Name': person_last_names,
                                 'First Name': person_first_names,
                                 'Date of Birth': person_dobs},
                                columns=['Person ID', 'Last Name', 'First Name',
'Date of Birth'])

people_dask_df = dd.from_pandas(people_pandas_df, npartitions=2)

print(people_dask_df.divisions)
print(people_dask_df.npartitions)
```

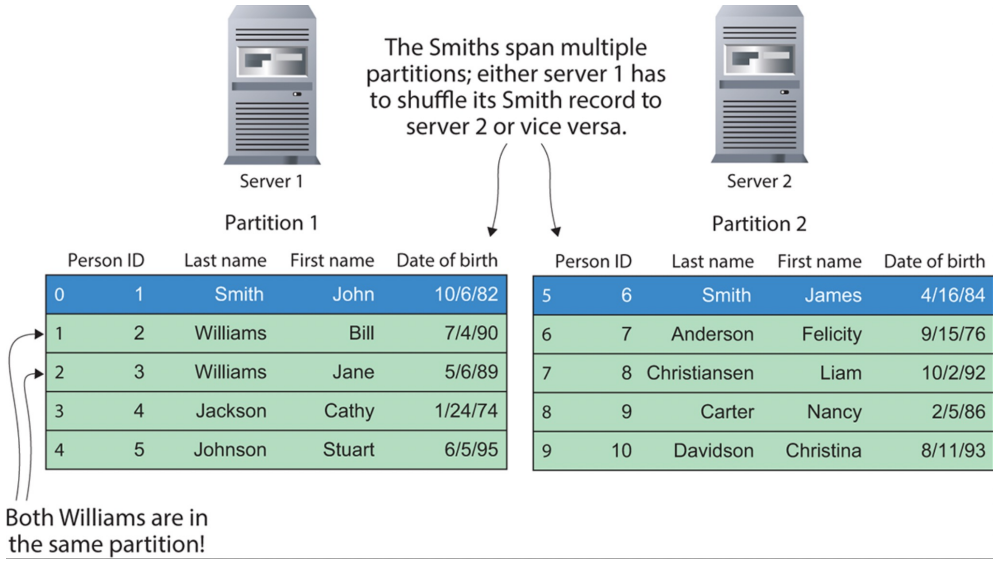
- We created a Dask DataFrame and explicitly split it into two partitions using the npartitions argument.
- Normally, Dask would have put this dataset into a single partition because it is quite small
- **divisions** shows the boundaries of the partitioning scheme; produces the output: (0, 5, 9)
- **npartitions** shows how many partitions exist in the DataFrame; produces the output: 2; partition 1 holds rows 0 to 4, partition 2 holds rows 5 to 9

Repartitioning a Dask DataFrame

```
#Inspecting the rows
people_dask_df.map_partitions(len).compute()

people_filtered = people_dask_df[people_dask_df['Last Name'] != 'Williams']

# Filters out people with a last name of Williams and recount the rows.
print(people_filtered.map_partitions(len).compute())
people_filtered_reduced = people_filtered.repartition(npartitions=1)

#Collapses the two partitions into one
print(people_filtered_reduced.map_partitions(len).compute())
```

- When computations include a substantial amount of filtering, the size of each partition can become imbalanced, which can have negative performance consequences on subsequent computations.
- The second two lines of code in the listing aim to fix the imbalance by using the repartition method on the filtered DataFrame.

Shuffling



Example: A GroupBy operation that requires a shuffle

- For the people named Smith, there's one Smith in partition 1 and one Smith in partition 2.
- Either server 1 will have to send its Smith to server 2 to make the comparison, or server 2 will have to send server 1 its Smith.
- We can do a few things to minimize the need for shuffling the data. First, ensuring that the data is stored in a presorted order will eliminate the need to sort the data with Dask.
- If possible, sorting the data in a source system, such as a relational database, can be faster and more efficient than sorting the data in a distributed system

Limitations of Dask DataFrames

- Dask DataFrames do not expose the entire Pandas API.
- Some functions that Pandas does well are simply not conducive to a distributed environment.
- For example, functions that would alter the structure of the DataFrame, such as insert and pop, are not supported because Dask DataFrames are immutable.
- Relational-type operations, such as join/merge, groupby, and rolling are likely to involve a lot of shuffling, making them performance bottlenecks.
- Indexing has a few challenges due to the distributed nature of Dask. If you wish to use a column in a DataFrame as an index in lieu of the default numeric index, it will need to be sorted.

The Index recycled back to 0 when the partition boundary was reached.

	Person ID	Last name	First name	Date of birth
0	1	Smith	John	10/6/82
1	2	Williams	Bill	7/4/90
2	3	Williams	Jane	5/6/89
3	4	Jackson	Cathy	1/24/74
4	5	Johnson	Stuart	6/5/95
0	6	Smith	James	4/16/84
1	7	Anderson	Felicity	9/15/76
2	8	Christiansen	Liam	10/2/92
3	9	Carter	Nancy	2/5/86
4	10	Davidson	Christina	8/11/93

Using Dask to Process Big Data

- There are three main types of Dask user interfaces, namely Array, Bag, and Dataframe
- This example is run on a single machine. The power of dask is realized using a cluster.
- You could run it on AWS or other cloud infrastructures such as Saturn Cloud (http://docs.saturncloud.io/en/articles/3652101-spin-up-dask-on-Saturn)

```
import dask.dataframe as dd
df = dd.read_csv('https://e-commerce-data.s3.amazonaws.com/E-commerce+Data+(1).csv', encoding = 'ISO-8859-1', blocksize=32e6)
```

```
df
```

Out[6]: **Dask DataFrame Structure:**

	InvoiceNo	StockCode	Description	Quantity	InvoiceDate	UnitPrice	CustomerID	Country
npartitions=2								
	object	object	object	int64	object	float64	float64	object
	...	...	...	...	...	...	...	...
	...	...	...	...	...	...	...	...

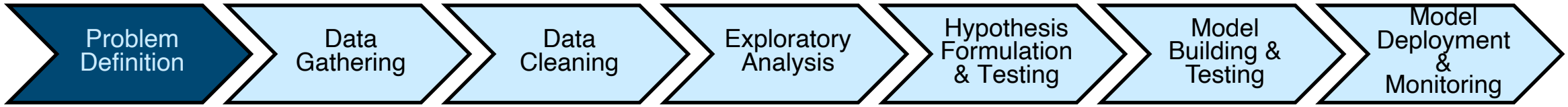
Dask Name: from-delayed, 6 tasks

Using Dask to Process Big Data (Cont'd)

- While Dask dataframe is very similar to Pandas dataframe, some differences exist.
- Most Dask user interfaces are **lazy**, in that they don't evaluate until you explicitly ask for a result using the compute method

```
df.UnitPrice.mean().compute()
df.isnull().sum().compute() # Check number of missing values for each column
df[df.quantity < 10].compute() # Filter rows based on conditions
```

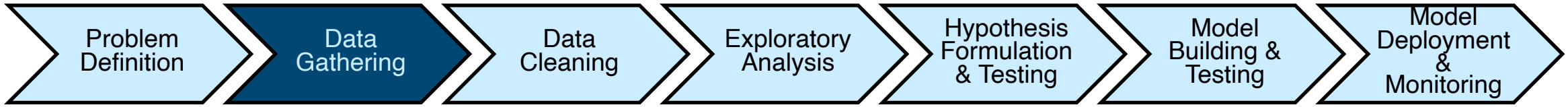
DATA PROCESSING WORKFLOW IN DASK



DEFINE THE PROBLEM

Big Data Analysis with Dask - Example

- Every third week of the month, the New York City Department of Finance records and publishes a data set of all parking citations issued throughout the fiscal year so far. The data collected by the city is quite rich, even including some interesting geographic features.
- The dataset spans from 2013 through June 2017 and is over 8 GB uncompressed.
- Locate the dataset on Canvas



GET THE DATA

Big Data Analysis with Dask - Example

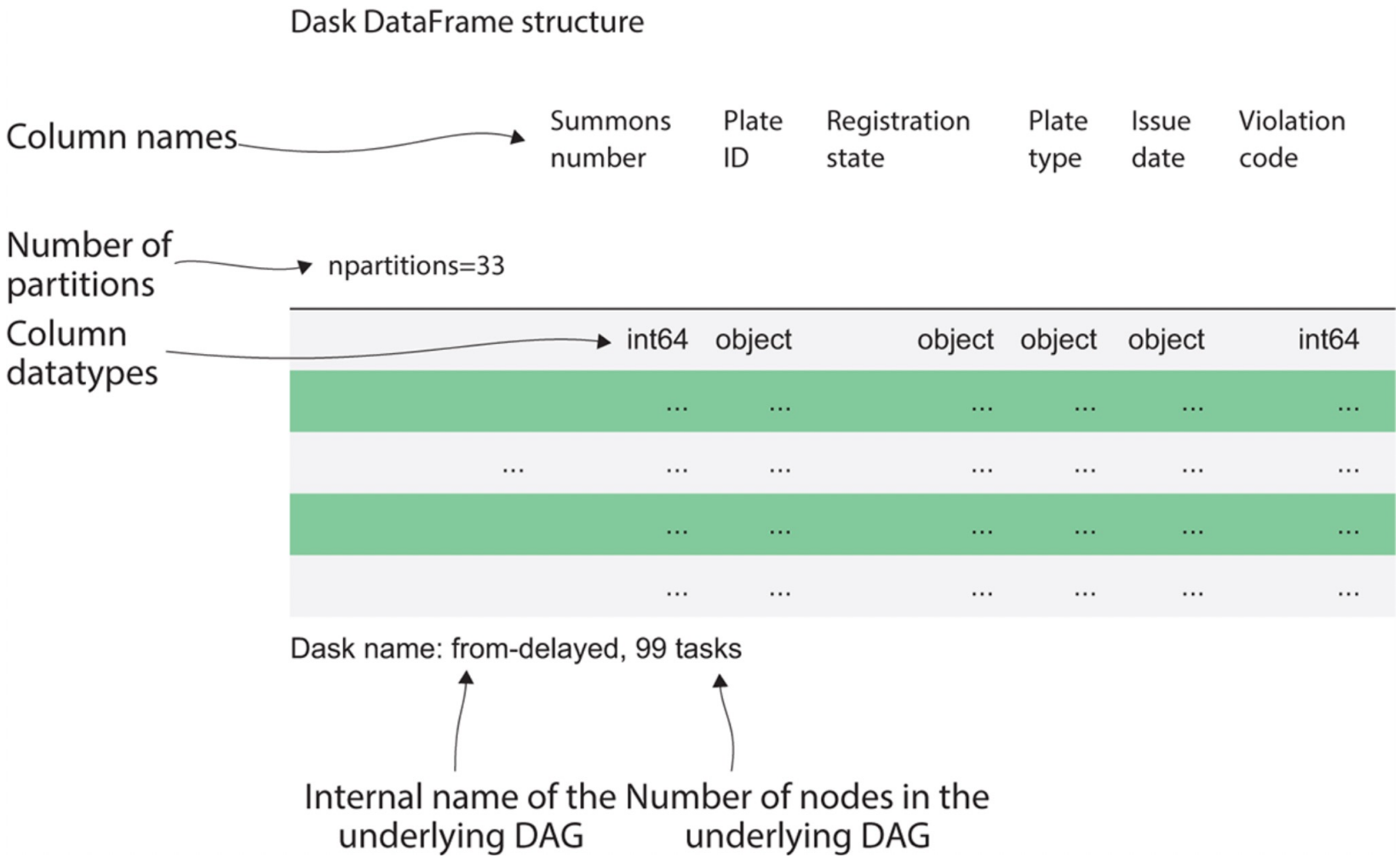
- import dask.dataframe as dd
- from dask.diagnostics import ProgressBar
- from matplotlib import pyplot as plt
- df = dd.read_csv('*2017.csv')

Dask DataFrame Structure:

	Summons Number	Plate ID	Registration State	Plate Type	Issue Date	Violation Code	Vehicle Body Type	Vehicle Make	Issuing Agency
npartitions=33									
	int64	object	object	object	object	int64	object	object	object
	...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...	...
	...	...	...	...	...	...	...	...	...
	...	...	...	...	...	...	...	...	...

Dask Name: read-csv, 33 tasks

Analyzing Parking Data With Dask



Fixing Mismatched Data Types

- Dask Data Type inferencing may fail when reading datasets

```
+----------------------+--------+----------+
| Column               | Found  | Expected |
+----------------------+--------+----------+
| House Number         | object | float64  |
| Time First Observed | object | float64  |
+----------------------+--------+----------+
The following columns also raised exceptions on conversion:
- House Number ValueError("could not convert string to float: 'N'")
- - Time First Observed ValueError("could not convert string to float:
  '1022A'")
- Usually this is due to dask's dtype inference failing, and *may* be fixed by
  specifying dtypes manually by adding:
- dtype={'House Number': 'object', 'Time First Observed': 'object'}
```

Recall Numpy Types

Basic type	Available NumPy types	Comments
Boolean	`bool`	Elements are 1 byte in size.
Integer	`int8, int16, int32, int64, int128, int`	`int` defaults to the size of `int` in C for the platform.
Unsigned integer	`uint8, uint16, uint32, uint64, uint128, uint`	`uint` defaults to the size of unsigned `int` in C for the platform.
Float	`float32, float64, float, longfloat`	`float` is always a double-precision floating-point value (64 bits). `longfloat` represents large-precision floats. Its size is platform dependent.
Complex	`complex64, complex128, complex`	The real and complex elements of a `complex64` are each represented by a single-precision (32-bit) value for a total size of 64 bits.
Strings	`str, unicode`	Unicode is always UTF32 (UCS4).
Object	`object`	Represents items in arrays as Python objects.
Records	`void`	Used for arbitrary data structures in record arrays.

Errors With Dask Inferencing

Column	Found	Expected
Issuer squad	object	int64
Unregistered vehicle?	float64	int64
Violation description	object	float64
Violation legal code	object	float64
Violation post code	object	float64

Dask inferred a text column to be a floating-point number? That doesn't look right!

- Dask treats blank records as null values when parsing files, and by default fills in missing values with NumPy's NaN (not a number) object called np.nan.
- If we use Python's built-in type function to inspect the datatype of an object, it reports that np.nan is a float type

Creating Schemas for DataFrames

Column	Found	Expected
Issuer squad	object	int64
Unregistered vehicle?	float64	int64
Violation description	object	float64
Violation legal code	object	float64
Violation post code	object	float64

Dask inferred a text column to be a floating-point number? That doesn't look right!

- Guess-and-check
  - With well-named columns, such as Product Description, Sales Amount, etc, you can try to infer what kind of data each column contains using the names
- Manually sample the data

Building a Generic Schema

```
from functools import reduce

columns = [set(fy14.columns),
           set(fy15.columns),
           set(fy16.columns),
           set(fy17.columns)]
common_columns = list(reduce(lambda a, i: a.intersection(i),
columns))
```

```
#Building a Generic Schema
import numpy as np
import pandas as pd
dtype_tuples = [(x, np.str) for x in common_columns]
dtypes = dict(dtype_tuples)
dtypes
```

```
1 fy14 = dd.read_csv('Parking_Violations_Issued_-_Fiscal_Year_2014__August_2013___June_2014_.csv',
2 dtype=dtypes)
3
```

```
1 with ProgressBar():
2     display(fy14[common_columns].head())
```

```
[########################################] | 100% Completed |  1.5s
```

	Time First Observed	Feet From Curb	Plate ID	Vehicle Year	Violation County	Violation Post Code	Violation Location	From Hours In Effect	Issuer Code	Double Parking Violation	...	Violation Legal Code	Violation Time	Street Code1	Days Parking In Effect	To Hours In Effect	Vehicle Expiration Date	I Pr
0	NaN	0	GBB9093	2013	NaN	NaN	0033	ALL	921043	NaN	...	NaN	0752A	37250	BBBBBBB	ALL	20140831	
1	NaN	0	62416MB	2012	NY	NaN	0033	ALL	921043	NaN	...	NaN	1240P	37290	BBBBBBB	ALL	20140430	
2	NaN	0	78755JZ	0	NY	NaN	0033	ALL	921043	NaN	...	NaN	1243P	37030	BBBBBBB	ALL	20140228	
3	NaN	0	63009MA	2010	NY	NaN	0033	ALL	921043	NaN	...	NaN	0232P	37270	BBBBBBB	ALL	20141031	
4	NaN	0	91648MC	2012	NY	NaN	0033	ALL	921043	NaN	...	NaN	1239P	37240	BBBBBBB	ALL	0	

5 rows × 43 columns

Reading Data into Dask from a RDBMS

```
#Reading SQL Data into Dask
username = ''
password = ''
hostname = 'localhost'
database_name = ''
odbc_driver = 'ODBC+Driver+13+for+SQL+Server'
connection_string = 'mssql+pyodbc://{0}:{1}@{2}/{3}?driver={4}'.format(username, password, hostname,
database_name, odbc_driver)

data = dd.read_sql_table('violations', connection_string, index_col='Summons Number')
```

Reading Data into Dask from a RDBMS

- If index_col='Summons Number' is a numeric or date/time datatype, Dask will automatically infer boundaries and partition the data based on a 256 MB block size (which is larger than read_csv's 64 MB block size).
- However, if the index_col is not a numeric or date/time datatype, you must either specify the number of partitions or the boundaries to partition the data by.

Reading Data into Dask from a RDBMS

Even Partitioning

```
#Even partitioning on a non-numeric or date/time index
data = dd.read_sql_table('violations', connection_string, index_col='Vehicle Color', npartitions=200)
```

Custom Partitioning

```
#Custom partitioning on a non-numeric or date/time index
partition_boundaries = sorted(['Red', 'Blue', 'White','Black', 'Silver', 'Yellow'])
data = dd.read_sql_table('violations', connection_string, index_col='Vehicle Color',
                         divisions=partition_boundaries)
```

Selecting a Subset of Columns

```
# Equivalent to: SELECT [Summons Number], [Plate ID], [Vehicle Color] FROM dbo.violations
column_filter = ['Summons Number', 'Plate ID', 'Vehicle Color']

data = dd.read_sql_table('violations', connection_string, index_col='Summons Number',
columns=column_filter)
```

Reading Data into Dask from a RDBMS

Specifying the Schema

```
# Equivalent to:
# SELECT * FROM violations
data = dd.read_sql_table('violations', connection_string, index_col='Summons Number', schema='chapterFour')
```

Reading Data into Dask from HDFS and S3

- HDFS3 must be installed on each Dask Worker so Dask can communicate with HDFS
  - pip or conda (hdfs3 is on the conda-forge channel)

```
data = dd.read_csv('hdfs://localhost/nyc-parking-tickets/*.csv', dtype=dtypes, usecols=common_columns)
```

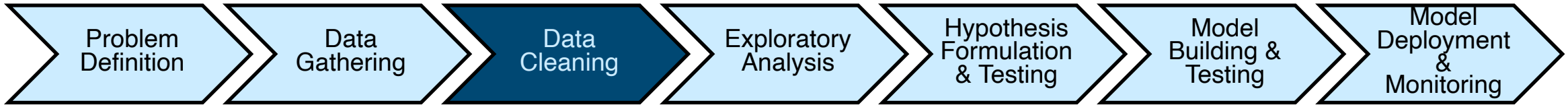
- S3FS must be installed on each Dask Worker so Dask can communicate with S3
  - pip or conda (s3fs is on the conda-forge channel)
- Each Dask worker must properly configured for authenticating with S3. s3fs uses the boto library to communicate with S3

```
data = dd.read_csv('s3://my-bucket/nyc-parking-tickets/*.csv', dtype=dtypes, usecols=common_columns)
```

Reading Data into Dask from Parquet Files

- Parquet is a high-performance columnar storage format jointly developed by Twitter and Cloudera that was designed with use on distributed filesystems in mind.
- Its design brings several key advantages to the table over text-based formats: more efficient use of IO, better compression, and strict typing

```
#Reads an entire dictionary of parquet files
data = dd.read_parquet('nyc-parking-tickets-prq')

#Reads parquet files from a distributed file system
data = dd.read_parquet('hdfs://localhost/nyc-parking-tickets-prq')
data = dd.read_parquet('s3://my-bucket/nyc-parking-tickets-prq')

#Specifying parquet read options
columms = ['Summons Number', 'Plate ID', 'Vehicle Color']
data = dd.read_parquet('nyc-parking-tickets-prq', columns=columns, index='Plate ID')
```



DATA CLEANING

Finding Percentage of Missing Values

```
missing_values = nyc_data_raw.isnull().sum()

with ProgressBar():
    percent_missing = ((missing_values / nyc_data_raw.index.size) * 100).compute()

percent_missing
```

Finding Percentage of Missing Values

Finding Percentage of Missing Values

In [12]:
```
1 missing_count = ((missing_values / df.index.size) * 100)
2 missing_count
```

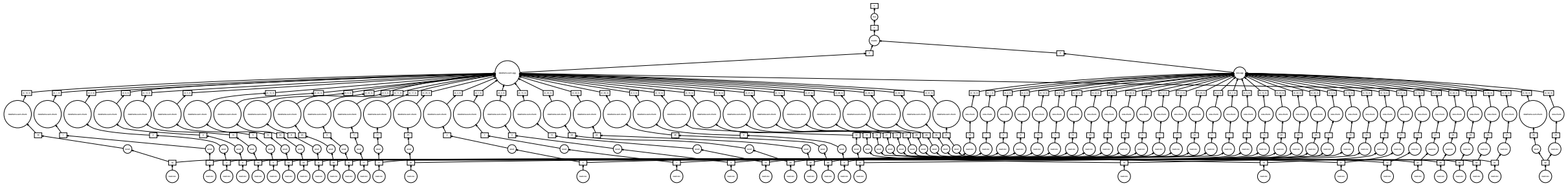
Out[12]:
```
Dask Series Structure:
npartitions=1
Date First Observed    float64
Violation Time             ...
dtype: float64
Dask Name: mul, 169 tasks
```

In [*]:
```
1 with ProgressBar():
2     missing_count_pct = missing_count.compute()
3 missing_count_pct
```

```
[                                        ] | 0% Completed |  1.6s
```

In [ ]:
```
1
```

Visualizing Missing Counts



Dropping Columns with Missing Values

```
#Drop missing values
columns_to_drop = list(percent_missing[percent_missing >= 50].index)
nyc_data_clean_stage1 = nyc_data_raw.drop(columns_to_drop, axis=1)
```

Imputing Missing Values

```
#Imputing Missing Values
with ProgressBar():
    count_of_vehicle_colors = nyc_data_clean_stage1['Vehicle Color'].value_counts().compute()
    most_common_color = count_of_vehicle_colors.sort_values(ascending=False).index[0]
    nyc_data_clean_stage2 = nyc_data_clean_stage1.fillna({'Vehicle Color': most_common_color})
```

Dropping Rows with Missing Values

```
#Drop rows with missing values
rows_to_drop = list(percent_missing[(percent_missing > 0) & (percent_missing < 5)].index)
nyc_data_clean_stage3 = nyc_data_clean_stage2.dropna(subset=rows_to_drop)
```

Imputing Multiple Columns with Missing Values

```
# Finding the datatypes of the remaining columns
remaining_columns_to_clean = list(percent_missing[(percent_missing >= 5) & (percent_missing < 50)].index)
nyc_data_raw.dtypes[remaining_columns_to_clean]

# Making a dictionary of values for fillna
unknown_default_dict = dict(map(lambda columnName: (columnName, 'Unknown'), remaining_columns_to_clean))

# Filling the DataFrame with default values
nyc_data_clean_stage4 = nyc_data_clean_stage3.fillna(unknown_default_dict)

# Checking the success of the filling/dropping operations
with ProgressBar():
    print(nyc_data_clean_stage4.isnull().sum().compute())
nyc_data_clean_stage4.persist()
```

Recoding Data with ‘Where-condition’

- Like missing values, it is not unusual to also have some instances in your dataset where the data is not missing but its validity is suspect

```
#isin returns True if the value is contained in the list of objects, otherwise false
condition = nyc_data_clean_stage4['Plate Type'].isin(['PAS', 'COM'])

#Find records where the condition is satisfied; replace all others with ‘Other’
plate_type_masked = nyc_data_clean_stage4['Plate Type'].where(condition, 'Other')

#drop the old Plate Type column completely
nyc_data_recode_stage1 = nyc_data_clean_stage4.drop('Plate Type', axis=1)

#create a new column named PlateType with the updated data
nyc_data_recode_stage2 = nyc_data_recode_stage1.assign(PlateType=plate_type_masked)

#rename this new column to the original "Plate Type"
nyc_data_recode_stage3 = nyc_data_recode_stage2.rename(columns={'PlateType': 'Plate Type'})
```

```
PAS 30452502
COM 7966914
OMT 1389341
SRF 394656
OMS 368952
...
HOU 4
JWV 3
LOC 3
HIF 2
SNO 2
Name: Plate Type, Length: 90, dtype:
int64
```

Recoding Data with a Mask

- If you have many unique values, but you want to keep only a few, using the where method is more convenient.
- Conversely, if you have many unique values but you want to get rid of only a few of them, using the mask method is more convenient.

```
single_color = list(count_of_vehicle_colors[count_of_vehicle_colors==1].index)
condition = nyc_data_clean_stage4['Vehicle Color'].isin(single_color)

vehicle_color_masked = nyc_data_clean_stage4['Vehicle Color'].mask(condition, 'Other')
nyc_data_recode_stage4 = nyc_data_recode_stage3.drop('Vehicle Color', axis=1)
nyc_data_recode_stage5 = nyc_data_recode_stage4.assign(VehicleColor=vehicle_color_masked)
nyc_data_recode_stage6 =nyc_data_recode_stage5.rename(columns={'VehicleColor':'Vehicle Color'})
```

Elementwise Operations

- Convert Date String to Date

```
from datetime import datetime
issue_date_parsed = nyc_data_recode_stage6['Issue Date'].apply(lambda x: datetime.strptime(x, "%m/%d/%Y"),
meta=datetime)
nyc_data_derived_stage1 = nyc_data_recode_stage6.drop('Issue Date', axis=1)
nyc_data_derived_stage2 = nyc_data_derived_stage1.assign(IssueDate=issue_date_parsed)
nyc_data_derived_stage3 =nyc_data_derived_stage2.rename(columns={'IssueDate': 'Issue Date'})
```

- Create a new ‘year-month’ column based on the issue-date

```
issue_date_month_year = nyc_data_derived_stage3['Issue Date'].apply(lambda dt: dt.strftime("%Y%m"), meta=int)
nyc_data_derived_stage4 = nyc_data_derived_stage3.assign(IssueMonthYear=issue_date_month_year)
nyc_data_derived_stage5 = nyc_data_derived_stage4.rename(columns={'IssueMonthYear':'Citation Issued Month Year'})
```

Filtering

- Finding all citations that occurred in October from 2013 - 2017

```
months = ['201310','201410','201510','201610','201710']
condition = nyc_data_derived_stage5['Citation Issued Month Year'].isin(months)
october_citations = nyc_data_derived_stage5[condition]

with ProgressBar():
    display(october_citations.head())
```

- Find citations that occurred after a given date

```
bound_date = '2016-4-25'
condition = nyc_data_derived_stage5['Issue Date'] > bound_date
citations_after_bound = nyc_data_derived_stage5[condition]

with ProgressBar():
    display(citations_after_bound.head())
```

ReIndexing

- Return a new DataFrame that's sorted by Year-Month, enabling us to use it for searching, filtering, and joining much more quickly

```
# Setting an index
with ProgressBar():
    condition = (nyc_data_derived_stage5['Issue Date'] > '2014-01-01')
    & (nyc_data_derived_stage5['Issue Date'] <= '2017-12-31')

nyc_data_filtered = nyc_data_derived_stage5[condition]
nyc_data_new_index = nyc_data_filtered.set_index('Citation Issued Month Year')
```

```
1 # Setting an index
2 with ProgressBar():
3     condition = (nyc_data_derived_stage5['Issue Date'] > '2014-01-01') & (nyc_data_derived_stage5['Issue Date']
4                                                                         <= '2017-12-31')
5     nyc_data_filtered = nyc_data_derived_stage5[condition]
6     nyc_data_new_index = nyc_data_filtered.set_index('Citation Issued Month Year')
7
```

```
[########################################] | 100% Completed | 38min 38.6s
```

Repartitioning the Data by Month/Year

- Define a partition scheme (**2014**01, **2014**02, **2014**03, etc).
- Next, pass the list of partitions into the repartition method to apply it to the newly reindexed DataFrame

```
# Repartitioning
years = ['2014', '2015', '2016', '2017']
months =['01','02','03','04','05','06','07','08','09','10','11','12']
divisions = [year + month for year in years for month in months]

with ProgressBar():
    nyc_data_new_index.repartition(divisions=divisions).to_parquet('nyc_data_date_index', compression='snappy')

nyc_data_new_index = dd.read_parquet('nyc_data_date_index')
```

Joining and Concatenating DataFrames

Person

Person ID	Last name	First name
1000	Daniel	Jesse
1001	Smith	John
1002	Robinson	Sarah
1003	Martinez	Amy

Pet

Pet ID	Owner ID	Name
100	1001	Norbert
101	1001	Sally
102	1000	Jack
103	1003	Fido

Two separate tables

Joined table

Person ID	Last name	First name	Pet ID	Owner ID	Name
1001	Smith	John	100	1001	Norbert
1001	Smith	John	101	1001	Sally
1000	Daniel	Jesse	102	1000	Jack
1003	Martinez	Amy	103	1003	Fido

- In a join operation, two data objects (such as tables and DataFrames) are combined into a single object by adding the columns from the left object to the columns of the right object.

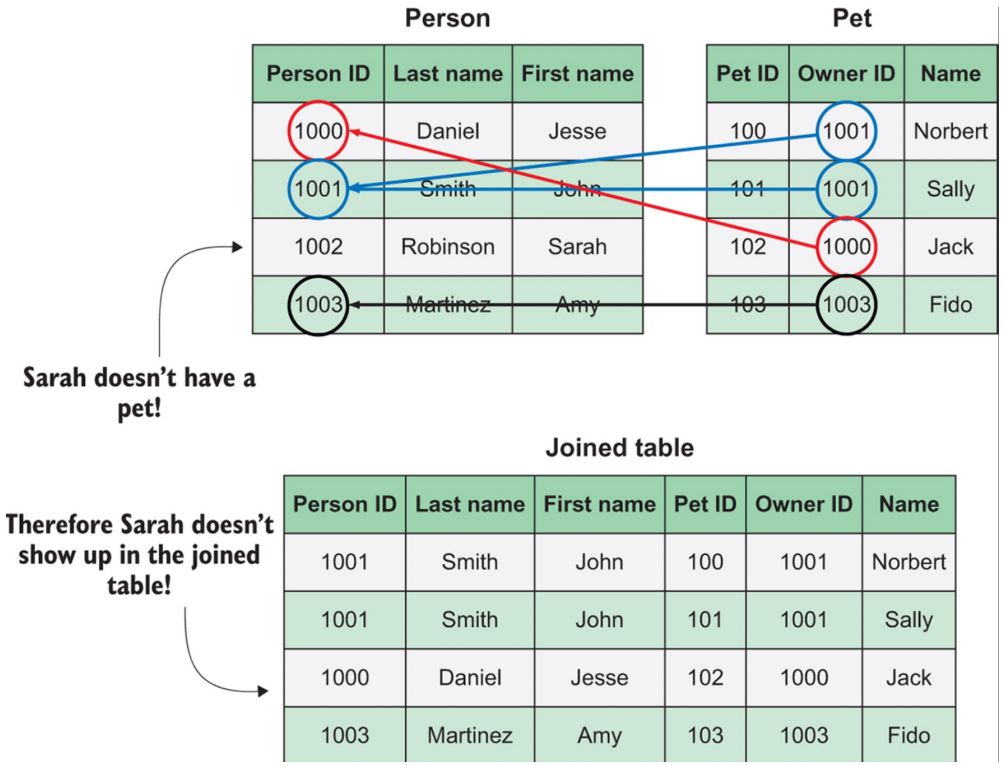
Person

Person ID	Last name	First name
1000	Daniel	Jesse
1001	Smith	John
1002	Robinson	Sarah
1003	Martinez	Amy

Pet

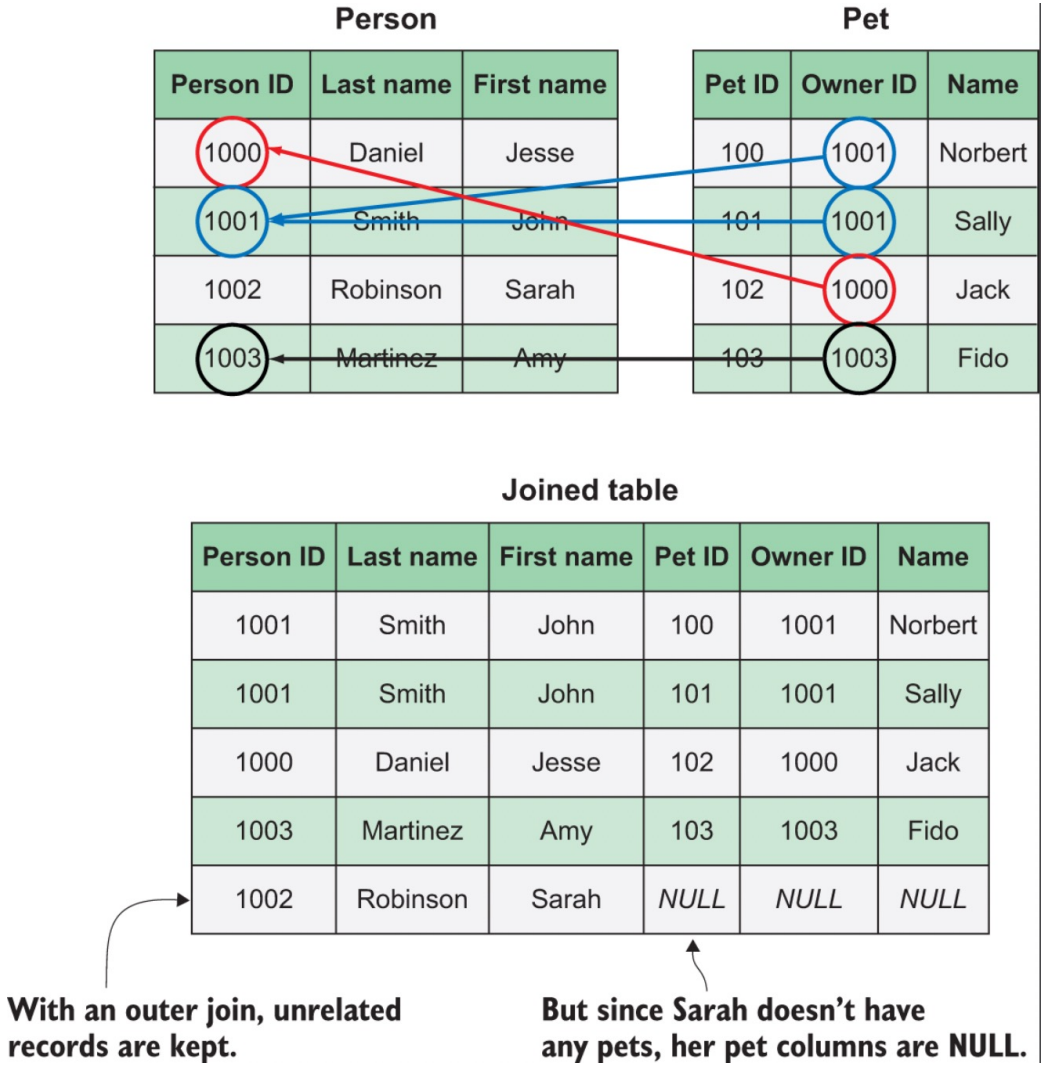
Pet ID	Owner ID	Name
100	1001	Norbert
101	1001	Sally
102	1000	Jack
103	1003	Fido

Joining and Concatenating DataFrames



- This is an inner join.
- It means that only records between the two objects that have relationships with each another are put in the combined table.
- Records that have no relationships are discarded

Joining and Concatenating DataFrames



- Since Sarah doesn't have any pets, information about her pets is NULL, which represents missing/unknown data.
- Dask's default behavior is to perform an outer join, so unless you specify otherwise, joining two tables will yield a result like this

Augmenting the NYC Parking Ticket data with NOAA weather data

```
# Augmenting the Ticket dataset with weather data
import pandas as pd
nyc_temps = pd.read_csv('data-science-python-dask-master/nyc-temp-data.csv')
nyc_temps_indexed = nyc_temps.set_index(nyc_temps.monthYear.astype(str))
nyc_data_with_temps = nyc_data_new_index.join(nyc_temps_indexed, how='inner')

with ProgressBar():
    display(nyc_data_with_temps.head(15))
```

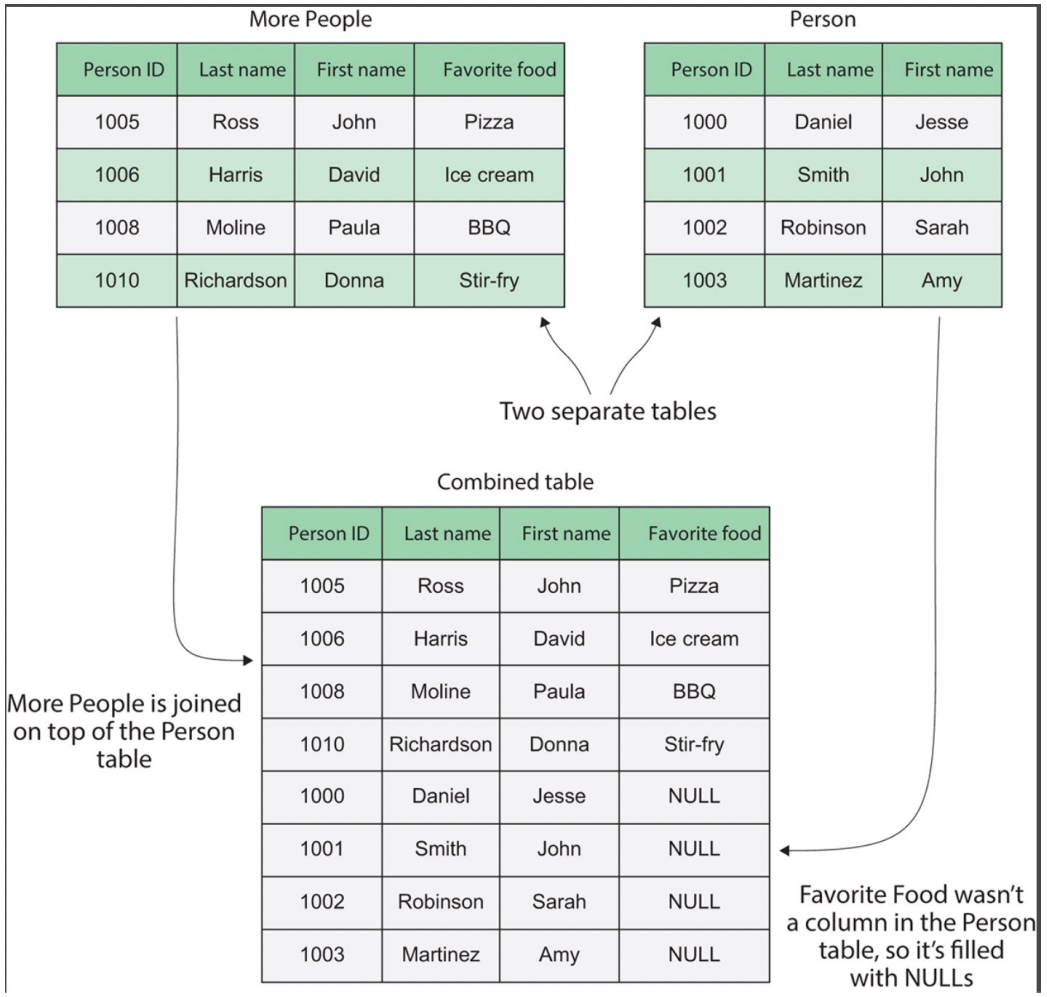
monthYear	Temp
01-2000	31.3
01-2001	33.6
01-2002	39.9
01-2003	27.5
01-2004	24.7
01-2005	31.3
01-2006	40.9
01-2007	37.5
01-2008	36.5
01-2009	27.9

Augmenting the NYC Parking Ticket data with NOAA weather data

gistration State	Violation Code	Vehicle Body Type	Vehicle Make	Issuing Agency	Street Code1	Street Code2	Street Code3	...	Vehicle Year	Feet From Curb	Violation Post Code	Violation Description	Plate Type	Vehicle Color	Issue Date	Citation Issued Month Year	Temp	monthYear
NY	46	SUBN	AUDI	P	37250	13610	21190	...	2013.0	0.0	Unknown	Unknown	PAS	GY	2013-08-04	08-2013	74.6	08-2013
NY	46	VAN	FORD	P	37290	40404	40404	...	2012.0	0.0	Unknown	Unknown	COM	WH	2013-08-04	08-2013	74.6	08-2013
NY	46	P-U	CHEVR	P	37030	31190	13610	...	0.0	0.0	Unknown	Unknown	COM	GY	2013-08-05	08-2013	74.6	08-2013
NY	46	VAN	FORD	P	37270	11710	12010	...	2010.0	0.0	Unknown	Unknown	COM	WH	2013-08-05	08-2013	74.6	08-2013
NY	41	TRLR	GMC	P	37240	12010	31190	...	2012.0	0.0	Unknown	Unknown	COM	BR	2013-08-08	08-2013	74.6	08-2013
NJ	14	P-U	DODGE	P	37250	10495	12010	...	0.0	0.0	Unknown	Unknown	PAS	RD	2013-08-11	08-2013	74.6	08-2013
NJ	24	DELV	FORD	X	63430	0	0	...	0.0	0.0	Unknown	Unknown	PAS	WHITE	2013-08-07	08-2013	74.6	08-2013
NY	24	SDN	TOYOT	X	63430	0	0	...	2001.0	0.0	Unknown	Unknown	PAS	WHITE	2013-08-07	08-2013	74.6	08-2013
NY	24	SDN	NISSA	X	23230	41330	83330	...	2012.0	0.0	Unknown	Unknown	PAS	WHITE	2013-08-12	08-2013	74.6	08-2013
NY	20	SDN	VOLKS	T	28930	27530	29830	...	2012.0	0.0	Unknown	Unknown	PAS	WHITE	2013-08-12	08-2013	74.6	08-2013
LA	17	SUBN	HONDA	T	0	0	0	...	0.0	0.0	Unknown	Unknown	PAS	TAN	2013-08-07	08-2013	74.6	08-2013
IL	40	SDN	SCIO	T	26630	40930	18630	...	0.0	6.0	Unknown	Unknown	PAS	BK	2013-08-10	08-2013	74.6	08-2013
PA	20	SDN	TOYOT	T	21130	71330	89930	...	0.0	0.0	Unknown	Unknown	PAS	GR	2013-08-06	08-2013	74.6	08-2013
NY	40	VAN	MERCU	T	23190	27290	20340	...	2003.0	0.0	Unknown	Unknown	COM	RD	2013-08-07	08-2013	74.6	08-2013
NY	51	VAN	TOYOT	X	93230	74830	67030	...	2013.0	0.0	Unknown	Unknown	PAS	GY	2013-08-06	08-2013	74.6	08-2013

monthYear	Temp
01-2000	31.3
01-2001	33.6
01-2002	39.9
01-2003	27.5
01-2004	24.7
01-2005	31.3
01-2006	40.9
01-2007	37.5
01-2008	36.5
01-2009	27.9

Unioning Two DataFrames



- Another way to combine datasets is along the row axis.
- In RDBMSs, this is called a **union** operation, but in Dask it's called **concatenating** DataFrames.

Unioning Two DataFrames

- Another way to combine datasets is along the row axis.
- In RDBMSs, this is called a **union** operation, but in Dask it's called **concatenating** DataFrames.

```
fy16 = dd.read_csv('Parking_Violations_Issued_-_Fiscal_Year_2016.csv', dtype=dtypes, usecols=dtypes.keys())
fy17 = dd.read_csv('Parking_Violations_Issued_-_Fiscal_Year_2017.csv',dtype=dtypes, usecols=dtypes.keys())

fy1617 = fy16.append(fy17)

with ProgressBar():
    print(fy16['Summons Number'].count().compute())
with ProgressBar():
    print(fy17['Summons Number'].count().compute())
with ProgressBar():
    print(fy1617['Summons Number'].count().compute())
```

Writing Datasets to File

- Dask's default behavior is to write one file per partition (ideal for distributed filesystem such as Spark or Hive)
- If we want to save a single file for be imported into another data analysis tool like Tableau or Excel, we have to fold all the data into a single partition using the repartition method before saving it.

```
import os
with ProgressBar():
    if not os.path.exists('nyc-final-csv'):
        os.makedirs('nyc-final-csv')

nyc_data_with_temps.repartition(npartitions=1).to_csv('nyc-final-csv/part*.csv')
```

Writing Datasets to File

- Dask's default behavior is to write one file per partition (ideal for distributed filesystem such as Spark or Hive)

```
with ProgressBar():
    if not os.path.exists('nyc-final-csv-compressed'):
        os.makedirs('nyc-final-csv-compressed')
    nyc_data_with_temps.to_csv(
        filename='nyc-final-csv-compressed/*',
        compression='gzip',
        sep='|',
        na_rep='NULL',
        header=False,
        index=False)
```

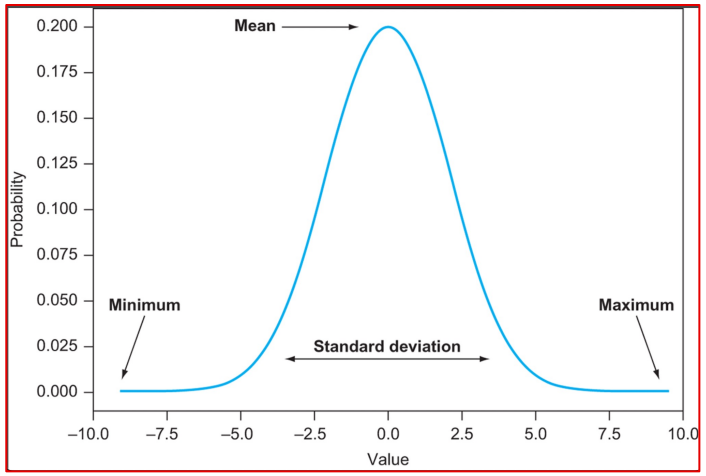
EDA WITH DASK

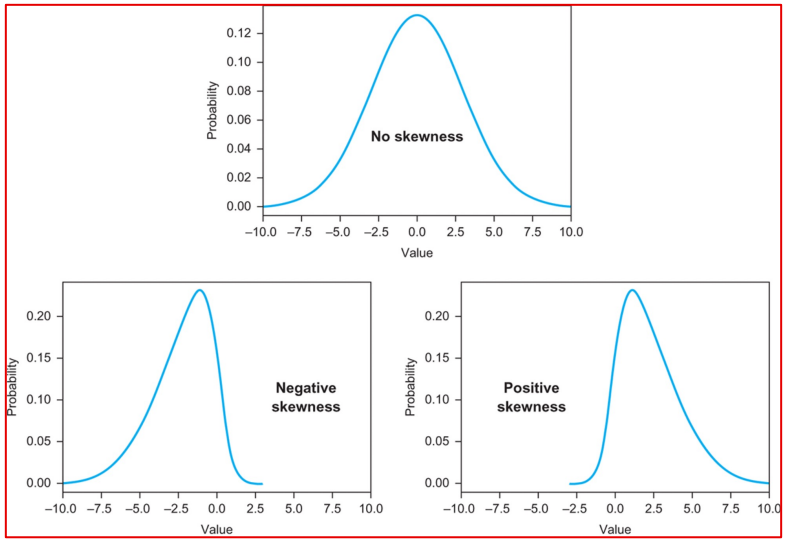
Descriptive Statistics

- In the final dataset we created and saved, we have ~41 million parking citations—that's a lot of observations!
- **Questions of Interest**
  - What is the average age of cars parked (illegally) on New York City streets?
  - Are there more newer cars than older cars?
  - How old was the oldest illegally parked car—are we talking Model T or Thunderbird?

Descriptive Statistics

- Descriptive Statistics is is typically defined by seven mathematical properties used to describe it:
  1. The smallest value (also called the **minimum**)
  2. The largest value (also called the **maximum**)
  3. The average of all data points (also called the **mean**)
  4. The middle point between the minimum and maximum value (also called the **median**)
  5. How spread out the data is from the mean (also called the **standard deviation**)
  6. The most commonly occurring observation (also called the **mode**)
  7. How balanced the number of data points to the left and right of the central point is (also called the **skewness**)





LOAD THE DATA

```
import dask.dataframe as dd
import pyarrow
from dask.diagnostics import ProgressBar

nyc_data = dd.read_parquet('nyc_final', engine='pyarrow')
```

Checking Fields for Anomalies

Check Vehicle Year for Anomalies

```
with ProgressBar():
    vehicle_age_by_year = nyc_data['Vehicle Year'].value_counts().compute()
vehicle_age_by_year
```

```
# Produces the following (abbreviated) output
0.0     8597125
2013.0  2847241
2014.0  2733114
2015.0  2423991
...
2054.0  61
2058.0  58
2041.0  56
2059.0  55
```

Filtering out Bad Data

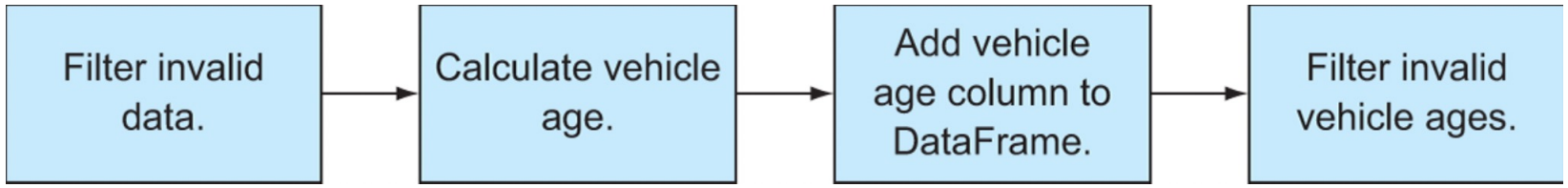
- Write a Filter to select only valid years

```
with ProgressBar():
    condition = (nyc_data['Vehicle Year'] > 0) & (nyc_data['Vehicle Year'] <= 2018)
    vehicle_age_by_year = nyc_data[condition]['Vehicle Year'].value_counts().compute().sort_index()

vehicle_age_by_year
```

```
nyc_data_filtered = nyc_data[condition]
```

Creating Derived Fields

- Calculating the vehicle age at the date of citation

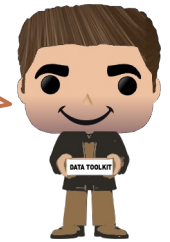



```
def age_calculation(row):
    return int(row['Issue Date'].year - row['Vehicle Year'])

vehicle_age = nyc_data_filtered.apply(age_calculation, axis=1, meta=('Vehicle Age', 'int'))

nyc_data_vehicle_age_stg1 = nyc_data_filtered.assign(VehicleAge=vehicle_age)
nyc_data_vehicle_age_stg2 = nyc_data_vehicle_age_stg1.rename(columns={'VehicleAge':'Vehicle Age'})

nyc_data_with_vehicle_age = nyc_data_vehicle_age_stg2[nyc_data_vehicle_age_stg2['Vehicle Age'] >= 0]
```

Computing Descriptive Statistics in Dask

```
from dask.array import stats as dask_stats
with ProgressBar():
    mean = nyc_data_with_vehicle_age['Vehicle Age'].mean().compute()
    stdev = nyc_data_with_vehicle_age['Vehicle Age'].std().compute()
    minimum = nyc_data_with_vehicle_age['Vehicle Age'].min().compute()
    maximum = nyc_data_with_vehicle_age['Vehicle Age'].max().compute()
    skewness = float(dask_stats.skew(nyc_data_with_vehicle_age['Vehicle Age'].values).compute())
```

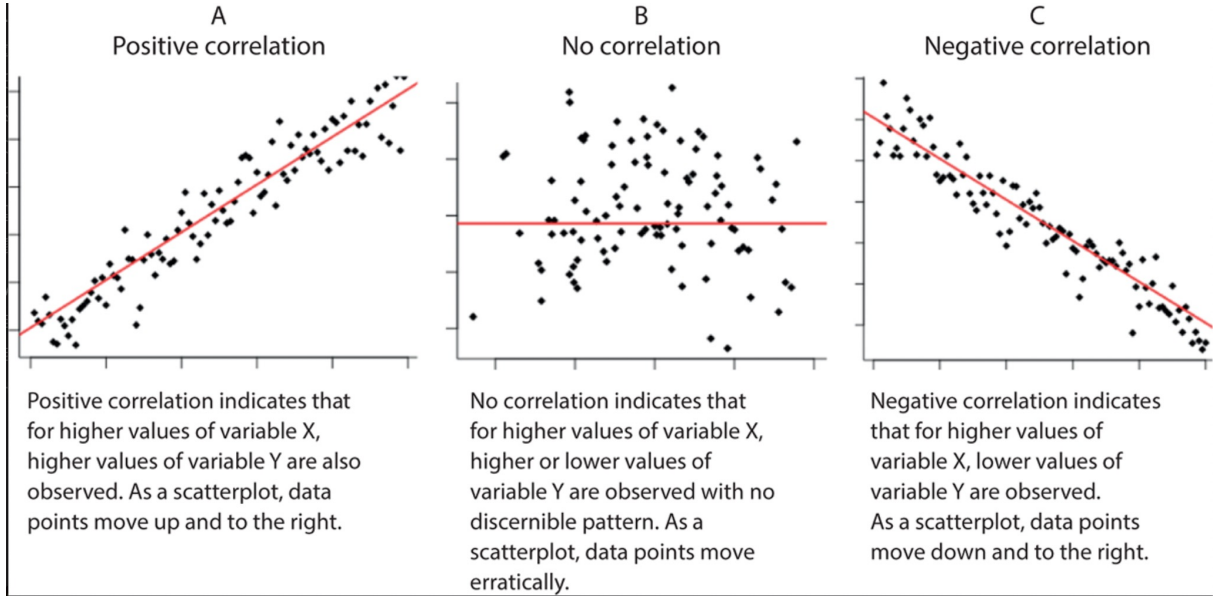
Statistic	Vehicle age
Mean	6.74
Standard deviation	5.66
Minimum	0
Maximum	47
Skewness	1.01




```
with ProgressBar():
    descriptive_stats = nyc_data_with_vehicle_age['Vehicle Age'].describe().compute()

descriptive_stats.round(2)
```

Built-In aggregate functions

- Using the NYC Parking Ticket data, how many parking citations were issued each month?
- Does the average temperature correlate with the number of citations issued?



The End