IS 6493

Homework Exercise 4 : Parallel and Distributed computing with Dask

(Total: 25 points)

In this exercise, we will learn about parallel and distributed computing in Python. We will specifically spend much time on one such library – Dask.

Submission instructions

Please create a new notebook called **Hw4_LastName_FirstName.ipynb**

- Part 1 (sections A, B) and Part 2 (sections A, B, C) are only for learning, no coding
- There are coding tasks in Part 2 (sections D and E).
    - In the first cell, enter a ***L1 heading*** – 'Task 0' and a ***L2 sub-heading*** 'Exercise 4 completed by *YourName*'
    - Use ***L1 headings*** to clearly indicate the two sections – 'section D' and 'section E', and use ***L2 numbered*** headings for the tasks (Task 1, Task 2, …) within these sections. Use ***L3 heading*** for the sub-tasks - Task 9a, 9b … in separate cells.
- Ensure that you have run all commands and cells should have visible results (before you submit). We cannot re-run all your cells while grading, and will grade the visible output. (3 point penalty if we have to run your (big data) tasks)
- Please submit your completed notebook (Graded tasks are highlighted). Please complete the entire exercise (all graded and ungraded tasks) in your submission.

PART 1

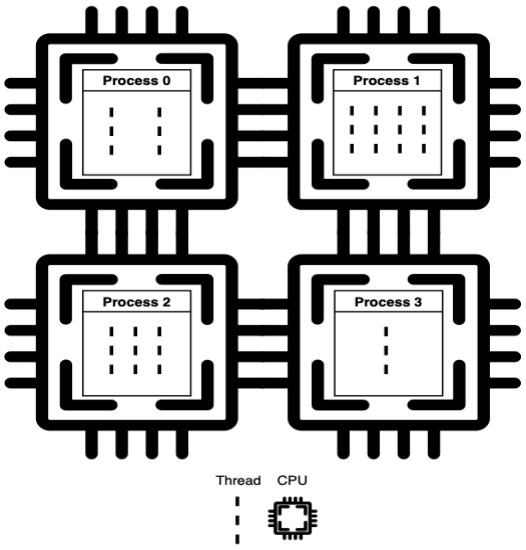
The common python data science libraries (numpy, pandas, sklearn) were built to work on a **single CPU**. When it comes to working with large datasets using these python libraries, the run time can become very high due to memory constraints. These libraries usually work well if the dataset fits into the existing RAM. But if we are given a large dataset to analyze (like 8/16/32 GB or beyond), it would be difficult to process and model it. Unfortunately, these popular libraries were not designed to scale beyond a single machine (without additional help).

Often, we need to use certain tricks to enable computation of larger-than-memory datasets: parallel execution or delayed/background execution.

A. UNDERSTANDING PARALLELISM

There are two broad techniques for working on tasks in parallel. They are `multiprocessing` and `multithreading`. While they may sound similar, they are quite different. The key differences are discussed below:

- A **process** is an independent instance of a program executed in a processor core (run on separate processing nodes aka CPU/GPU cores). A **thread** is a component of a process that can be scheduled for execution. It runs concurrently (inside that process), and is the smallest unit of processing that can be performed in an OS (Operating System).
- Processes do not share the same memory space, while threads inside a process do.
    - It is easier, faster, and safer to share data among threads.
- Threads are lighter and cause less overhead than processes.
- Running multiple processes is called **multiprocessing** and running multiple threads is called **multithreading**.
- `multithreading` implements concurrency, `multiprocessing` implements parallelism.
        - In **concurrent** execution, two or more tasks are progressing at the same time; but they are not being run at the same time.
        - This is typically used for I/O intensive tasks (IO-bound) where the bottlenecks are read/write operations (making a web request, or database operations, or reading/writing to a file). In such tasks, we usually spend a lot of time waiting for external inputs from users or other programs.
        - Here we want to assign one thread to wait on (and read) such inputs, while another thread can continue other tasks (formatting, error checking, calculating etc.)
    - In **parallel** execution, two or more jobs are being executed simultaneously.
        - Typically used for CPU intensive tasks where the bottleneck is time and resources (CPU bound). Such computing-bound tasks, if they can be divided into independent chunks, will benefit by using multiple cores for processing.
        - Here, we want to be able to run tasks at the same time with the goal of parallel computations to save time, and increase the speed of operations.
        - The gain is not proportional to the num of cores, because there will be more overhead involved as more cores are used (passing information and coordinating among them). This is what makes processes heavier than threads.



- In Python, true parallelism can ONLY be achieved using `multiprocessing`.
    - That is because in Python, only one thread can be executed at a given time inside a given process. This is assured by Python's *global interpreter lock (GIL)*
        - In a simple, single-core CPU, multithreading is achieved using frequent switching between threads. This is termed as **context switching**. In context switching, the state of a thread is saved and state of another thread is loaded whenever any interrupt (due to I/O or manually set) takes place. Context switching takes place so frequently that all the threads appear to be running pseudo-parallelly (and they appear to be multi-tasking).

Summary: use multithreading for I/O bound tasks and multiprocessing for CPU bound tasks.

B. IMPLEMENTATION IN PYTHON

In Python a class called **ThreadPoolExecutor** found in **concurrent.futures** module is used to efficiently manage and create threads. ThreadPoolExecutor class exposes a few methods to execute threads asynchronously. Similarly **ProcessPoolExecutor** in **concurrent.futures** module gives us the ability to work with multiple processes.

You can write code to use these classes directly to work with threads and processes. OR we can use the DASK library, which makes it much easier to parallelize data science tasks that use pandas and scikit-learn.

PART 2. DASK

A. UNDERSTANDING DASK

Dask is a 'parallel computing' python library that can handle parallel processing of large datasets by using multiple CPU cores of machines (**parallel computing**) or on a cluster of machines (**distributed computing**). Additionally, it also offers custom Dask data structures that ***mimic*** the ones in the single-core libraries (e.g., NumPy, Pandas, Scikit-Learn) using a similar API (methods/functions). Dask does this by providing its own implementation of Arrays and DataFrames.

Caveat: However Dask data structures are not full substitutes (can't do everything that NumPy arrays and Pandas DataFrames support; not all methods and functions are implemented.

A benefit of using Dask is that you can simply import Dask Arrays and DataFrames and write code like you would using NumPy and Pandas, with minimum changes in your code.

Dask provides the following user interfaces to parallelize (and scale) NumPy, Pandas and Scikit-learn, respectively.

- **Dask Arrays** (NumPy – vectors and matrices)
- **Dask DataFrames** (Pandas – labeled tabular data)
- **Dask ML** (Scikit-learn)
- and also, **Dask Bag** (for non-tabular big data or unordered collection of objects)
    - Dask bags are used to parallelize computations on unstructured or semi-structured data like text data, log files, JSON records, or user defined Python objects.
    - Dask Bag implements operations like map, reduce, filter, fold, and groupby on collections of generic Python objects. It does this in parallel with a small memory footprint using Python iterators. It is similar to a Pythonic version of the *PySpark RDD* (which works with unstructured data; we will learn about Spark later).

Execution on Dask data structures provide the following benefits (similar to other Big Data frameworks such as Hadoop and Spark):

- Allows processing of data that doesn't fit into memory by breaking it into blocks and specifying task chains that *iteratively* and *lazily* execute. Minimizes the memory footprint of your computation, effectively streaming data from disk.
- Parallelize execution of tasks across cores (can use all cores on the computer) and even nodes of a cluster (across machines)
- Blocked Algorithms- perform large computations by performing many smaller computations across blocks (or contiguous chunks) of data
- Moves computation (smaller program files) to the data rather than the other way around, to minimize communication overhead from moving big data around

B. DASK DATAFRAMES

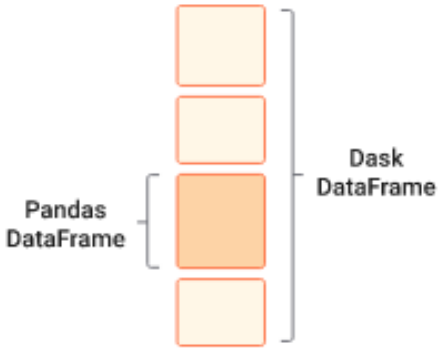
Pandas is great for tabular datasets that fit in memory. Dask becomes useful when the dataset you want to analyze is larger than your machine's RAM. The demo dataset we're working with is only about 250MB, so that you can download it in a reasonable time, but `dask.dataframe` will scale to datasets much larger than memory.

Dask DataFrame is used in situations where Pandas is commonly needed, but when Pandas fails due to data size or computation speed, such as:

1. Manipulating large datasets, even when those datasets don't fit in memory
2. Accelerating long computations by using many cores
3. Distributed computing on large datasets with standard Pandas operations like groupby, join, and time series computations

The `dask.dataframe` module implements a blocked parallel `DataFrame` object that mimics a large subset of the Pandas `DataFrame` API. One Dask `DataFrame` is comprised of many in-memory pandas `DataFrames` separated row-wise along the index. One operation on a Dask

`DataFrame` triggers many Pandas operations on the constituent pandas `DataFrame`s in a way that is mindful of potential parallelism and memory constraints.



Dask DataFrame may not be the best choice for Dask data structure in the following situations:

- If your dataset fits comfortably into RAM on your laptop, then you may be better off just using Pandas. There may be simpler ways to improve performance than through parallelism.
- If your dataset doesn't fit neatly into the Pandas tabular model, then you might find more use in **Dask Bag** or **Dask Array**
- If you need functions that are not implemented in Dask DataFrame, then you might want to look at **Dask delayed**
    - This is a low-level data structure that you can write custom functions with

In this exercise, we will mainly focus on tabular Dask DataFrames.

C. DASK CLIENT (DISTRIBUTED SCHEDULER)

Task Graphs

Normally, we humans write programs and then compilers/interpreters interpret them. Sometimes we humans disagree with how these compilers/interpreters choose to interpret and execute their programs. In these cases, we often bring the analysis, optimization, and execution of code into the code itself. This is common when we want parallel execution in Dask.

A common approach to parallel execution is *task scheduling*. In task scheduling we break our program into many medium-sized tasks or units of computation, often a function call on a non-trivial amount of data. We represent these tasks as nodes in a graph with edges between nodes if one task depends on data produced by another.

Once we have set up a task graph, we need a *task scheduler* to execute it on parallel hardware. The *task scheduler* will execute this graph in a way that respects these data dependencies and leverages parallelism where possible, so multiple independent tasks can be run simultaneously.

Different task schedulers exist, and each will consume a task graph and compute the same result, but with different performance characteristics.

Task Schedulers

Dask architecture: Dask has a ***central task scheduler*** and a set of ***workers***. The scheduler assigns tasks to the workers. Each worker is assigned a number of cores on which it can perform computations and can also serve results to other workers on demand.

Dask has two families of task schedulers:

- **Single-machine scheduler**: This scheduler provides basic features on a local process or thread pool. This scheduler was made first and is the default. It is simple and cheap to use, although it can only be used on a single machine and does not scale.

    - ***The threaded scheduler (default for Dask Array and DataFrame)***

    executes computations with a local `concurrent.futures.ThreadPoolExecutor`. It is lightweight and requires no setup. It introduces very little task overhead (around 50µs per task) and, because everything occurs in the same process, it incurs no costs to transfer data between tasks (as threads can share memory).

    However, recall that at any one time, only one thread can operate when working with standard Python code, and this is achieved in Python using Python's Global Interpreter Lock (GIL). Thus, this scheduler only provides parallelism when your computation is dominated by non-standard Python code, as is primarily the case when operating on numeric data in NumPy arrays, Pandas DataFrames, Scikit-Learn or using any of the other C/C++/Cython based projects.

    NOTE: Scientific Python libraries such as NumPy, Scipy, Pandas and Scikit-learn often release the GIL in performance critical code paths (allowing multiple threads per core) – something we can take advantage of and will in see in the exercise.

    The threaded scheduler is the default choice for Dask Array and Dask DataFrame. However, if your computation is dominated by processing pure Python objects like strings, dicts, or lists, then you may want to try a process-based scheduler (to avoid GIL limitations).

    Furthermore, even if only using a single machine, it is now recommended to use the distributed scheduler (even on a local machine).

    - ***The multiprocessing scheduler (default for Dask Bag)***

    executes computations with a local `concurrent.futures.ProcessPoolExecutor`. It is lightweight to use within Dask and requires no setup. Every task and all of its dependencies are shipped to a local process, executed, and then their result is shipped

back to the main process. This means that it is able to bypass issues with the GIL and provide parallelism even on computations that are dominated by pure Python code, such as those that process strings, dicts, and lists.

However, moving data to remote processes and back can introduce performance penalties, particularly when the data being transferred between processes is large. The multiprocessing scheduler is therefore an excellent choice when workflows are relatively linear, and so does not involve significant inter-task data transfer as well as when inputs and outputs are both small, like filenames and counts.

- **Distributed scheduler**: This scheduler is more sophisticated, offers more features, but also requires a bit more effort to set up. It can run locally on one machine or distributed across a cluster.

  For more complex workloads, where large intermediate results may be depended upon by multiple downstream tasks, it is generally recommended to use the distributed scheduler on a local machine. The distributed scheduler is more intelligent about moving around large intermediate results.

  The distributed scheduler is beneficial even on a single machine:

  - It provides access to asynchronous API (module `concurrent.futures`)
  - It provides a diagnostic dashboard that can provide valuable insight on performance and progress
  - It often can be more efficient than the multiprocessing scheduler on workloads that require multiple processes

Now, let's try out Dask in code to see some of the abovementioned concepts in action.

D. DASK DATAFRAMES IN CODE – NYC AREA FLIGHTS DATA (19 points)

In this section, we will use Dask DataFrames on a dataset comprised of many .csv files about flights from/ to NYC-area airports from 1990 – 1999.

1. Install and Imports

To use Dask, we have to make sure it (and some dependent libraries) is installed. At the Anaconda prompt (Windows) or Terminal (Mac/Linux), you can type:

[BUT DON'T DO THIS YET UNTIL YOU HAVE READ THE FOLLOWING ABOUT SETTING UP NEW VIRTUAL ENVs]

```
conda install dask
conda install graphviz
conda install python-graphviz
conda install distributed
```

Sometimes, you may wish to create virtual environments within Anaconda to work with different projects. This is done to reduce any clashes across packages and their dependencies.

`base` is the default environment given to us by Anaconda. You can see this when you start Anaconda prompt.

```
(base) PS C:\Users\u0625859>
```

And when you create and activate a new virtual env called `daskpy` (instructions follow below), the prompt will change to look like the below.

```
(daskpy) C:\Users\u0625859>
```

Next let us learn how to create, activate and use a new environment to use Dask and its dependencies, namely.

- **graphviz** (Graphviz is an open source graph visualization software. Graph visualization is a way of representing structural information as diagrams of abstract graphs and networks.)
- **python-graphviz** (This package facilitates the creation and rendering of graph descriptions in the DOT language of the Graphviz graph drawing software from Python.)
- **pygraphviz** (PyGraphviz is a Python interface to the Graphviz graph layout and visualization package. With PyGraphviz you can create, edit, read, write, and draw graphs using Python to access the Graphviz graph data structure and layout algorithms. More powerful, but not needed for our use now.
- **distributed** (to help with distributing, scheduling and managing the work)

Type the following commands in either the Anaconda prompt (Windows) or the Terminal (Mac/Linux). [… depicts the path to your Anaconda installation].

1. List all existing virtual environments for Python available on your machine
```
(base) … > conda info -e
```

2. Create a new virtual env and specify the latest Python version to use
```
(base) … > conda create -n daskpy python=3.9 anaconda
```

3. Activate the new venv
```
(base) … > conda activate daskpy
```

4. Install dask
```
(daskpy) … > conda install dask
```

5. Install graphviz
```
(daskpy) … > conda install graphviz
```

6. Install python-graphviz
```
(daskpy) … > conda install python-graphviz
```

7. Install distributed
```
(daskpy) … > conda install distributed
```

8. Check that new venv was added
```
(daskpy) … > conda info -e
```

9. Open jupyter notebook from this new venv
```
(daskpy) … > jupyter notebook
```

10. when done, close the new venv and go back to base **
```
(daskpy) … > conda deactivate
(base) … >
```

Note: Mac users used to have some trouble installing graphviz and python-graphviz due to the path setup on your OS. However, this seem to be taken care of now. Please check the end of this file, for some tips if you still encounter issues (you will need to run pip from within your Notebook to install graphviz). You are welcome to do a Google search to see what other solutions exist. If nothing works, you can skip those two steps (you will only not be able to perform task 10 in Part 2 section D).

Running the notebook in the future

Now, if you want to run a notebook that uses Dask, you will need to first activate the new venv we created above (daskpy) using just the following command:
conda activate daskpy

```
(base) PS C:\Users\u0625859> conda activate daskpy
(daskpy) PS C:\Users\u0625859> jupyter notebook
```

Returning to base venv

```
(daskpy) PS C:\Users\u0625859> conda deactivate
(base) PS C:\Users\u0625859>
```

Using daskpy as your venv, after opening up the jupyter Notebook dashboard in your browser, create a new Python Notebook in Jupyter (name it **Hw4_LastName_FirstName.ipynb**.

Import the below:

```
import os
import dask
from dask.distributed import Client
import dask.dataframe as dd
import pandas as pd
import glob
```

2. *Get the data and read it in*

We will use a dataset consisting of only domestic flights that flew from one of 3 NYC-area airports (LGA, EWR and JFK as origins) to several US destinations. The destinations also include the three airports. The data is saved across multiple .csv files – for each year 1990-1999. The data tracks information on several flight statistics.

Save the TAR data file (an archive file, like zip) to a local folder (that contains this notebook file), and extract the files into a sub-folder called 'nycflights' (right click on tar file and choose your zip utility*[1] and select to extract files to a folder called nycflights). This new folder will

[1] You will need to already have a zip utility installed on your machine to see this option to extract files from a tar/zip. If you don't have a tool, you can get a free one such as 7zip https://www.7-zip.org/

then contain .csv files from 1990-1999. It is absolutely important that you use the same path commands as shown below (DO NOT use your own custom paths).

Create a path object using os.path.join. This creates paths that work across multiple OS.

```
filename = os.path.join('nycflights','*.csv')
filename
```

We can then use the .read_csv() function of Dask DataFrame (similar to the Pandas function). We will call our Dask DataFrame object as ddf.

```
ddf = dd.read_csv(filename)
```

dd.read_csv() parallelizes the pandas.read_csv() function by supporting the loading of many files at once using globstrings.

pandas .read_csv() cannot do this directly and would need more code to read and merge/ concatenate multiple files. Let us also create a Pandas DataFrame of the same data.

```
pdf = pd.concat(map(pd.read_csv, glob.glob(filename)))
```

In the above code, we use glob() function from the glob module, which returns a list of paths matching a pathname pattern. We use Python's map() function, which applies a function specified as its first parameter on data specified as its second parameter. Finally we use pd.concat() to merge or concatenate the multiple Pandas DataFrames.

3. *Lazy operations*

```
ddf
```

The above command does not display the contents (whereas calling pdf immediately shows the content).

Dask DataFrame Structure:

	Year	Month	DayofMonth	DayOfWeek	DepTime	CRSDepTime	ArrTime	CRSArrTime	UniqueCarrier	FlightNum	TailNum	ActualElapsedTime	CRS
npartitions=10													
	int64	int64	int64	int64	float64	int64	float64	int64	object	int64	float64	float64	
	...	...	...	...	...	...	...	...	...	...	...	...	
...	...	...	...	...	...	...	...	...	...	...	...	...	
	...	...	...	...	...	...	...	...	...	...	...	...	
	...	...	...	...	...	...	...	...	...	...	...	...	

Dask Name: read-csv, 10 tasks

This is because Dask performs lazy evaluation. Many commands will not actually compute and provide results, unless we explicitly ask it to.

```
ddf.head()
```

Calling the .head() method will show content in the output.

In other cases where we perform transformation on the DataFrame, we will need to call the Dask DataFrame's .compute() method to ask it to evaluate the command.

4. Timing exercise (1 point)

Try out a timing exercise to assess the speed between the above two operations to read data in. See whether the Dask operation is faster than Pandas – it will be as the files get larger.

> Recall that **%timeit** is a line magic function, and can be used to time a single line of code. **%%timeit** is cell magic and can be used to time an entire cell (multiple lines of code)
>
> *NOTE: Any variables created within timing operations are not available in the rest of the program (as they have block scope), so you **will have to re-create** the ddf and pdf outside of the timing cells in order to have them for use later in the program. Here we already did that in step 2.*

```
%timeit  ddf = dd.read_csv(filename)
```

```
%timeit   pdf = pd.concat(map(pd.read_csv, glob.glob('nycflights/*.csv')))
```

5. DataFrame attributes

Print the column names.

```
ddf.columns
```

Print the data types

```
ddf.dtypes
```

Notice the values and data types of two columns: CRSElapsedTime and TailNum
Unlike pandas.read_csv() which reads in the entire file before inferring datatypes, dask.dataframe.read_csv() only reads in a sample from the beginning of the file (or first file if using a glob). These inferred datatypes are then enforced when reading all partitions.

In this case, the datatypes inferred in the sample are incorrect. Dask inferred the data type of CRSElapsedTime to be int64, but it has missing values (int cannot accommodate missing

values), so it should be set to float64 (float accommodates missing numeric values in Python). TailNum appears to have NaN values in the beginning (so it was inferred to be float64, but later the values are strings, so its dtype should be set to object which also allows missing text values).

If we don't correct the types, you will likely get ValueError when performing transformations later. To fix this, we can specify dtypes manually using the `dtype` keyword when reading in the data. This is the recommended solution, as it's the least error prone (better to be explicit than implicit) and also the most performant.

6. Read in the data again (2 points)

Let us fix a few things.

- We need to specify the dtype for any columns where the inference is wrong. Here CRSElapsedTime should be float and TailNum should be object. This is specified using a parameter 'dtype' which provides a dict of column names and data type to use.

- Our data has some date-related columns. By default, if there is a date column in the file, it gets read as an object data type using the default read_csv(). We don't have a single date column, but we have three columns with the components of a date – 'Year', 'Month', 'DayofMonth' at index 0, 1, 2. Let us create a new date column from these existing columns.

  To read the date column correctly, we can use the parameter 'parse_dates' to specify a list of date columns to use to construct a date. If the date is split across multiple columns - say, day, month, year, we can specify parameter 'parse_dates' to be a dict with the name of the new date column as key and the list of the column names in order (year, month, day) as the value.

```
ddf = dd.read_csv(filename,
                  parse_dates={'Date': [0, 1, 2]},
                  dtype={'CRSElapsedTime': 'float64',
                         'TailNum': 'object'})
```

Obtain the length of the Dask DataFrame.

```
len(ddf)
```

Verify that there is a new Date column added to the dask DF by calling the tail() method of the Dask DataFrame to display the last 10 rows.

```
ddf.tail()
```

Verify that the dtypes are set correctly by viewing a random sample of the data (fraction = 0.001). We have to call **.compute()** at the end to evaluate and get output.

```
ddf[['CRSElapsedTime', 'TailNum']].sample(frac=0.001).compute()
```

Note that methods of the same name may not support all the same parameter across Dask and Pandas. For example pd.DataFrame.sample() has a 'n' parameter to specify the number of random samples. Whereas dd.DataFrame.sample() does not, we have to specify the frac parameter instead.

7. Obtaining a Dask Series and indexing Dask DataFrames

Select a single column to obtain a Dask Series object.

```
ddf['TailNum'].compute()
```

We can use indexing with .loc[] to select rows and columns by label, as usual. Let us select all rows where the ArrTime is less than the mean of that column.

```
ddf.loc[ddf['ArrTime'] < ddf['ArrTime'].mean()].compute()
```

However, Dask DataFrame does not track the length of partitions, making positional indexing with .iloc inefficient for selecting rows. DataFrame.iloc[] only supports indexers where the row indexer is : (i.e. all rows)
Select all rows for column index 1 and 3.

```
ddf.iloc[:,[1, 3,]].compute()
```

But the below code produces an error (cannot use iloc to select rows with row index 1 and 3)

```
ddf.iloc[[1, 3,]].compute()
```

NOTE: Try it and then comment out this last line of code above (since it causes an error).

8. Converting from Pandas to Dask

If you have a Pandas DataFrame, we can convert it into a Dask DataFrame using .from_pandas() function of Dask 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. By default, the input DF will be sorted by the index to produce cleanly-divided partitions (with known divisions).

We can optionally specify a parameter npartitions to specify the number of partitions of the index to create. But if the data are too small, fewer partitions may be created.

```
ddf_partitions = dd.from_pandas(pdf, npartitions = 10)
ddf_partitions.partitions[1]
```

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

9. Computations on Dask DataFrames

Next let us perform some computations which are lazily evaluated on the Dask DataFrame ddf (not ddf_partitions). Remember to call .compute() to see results of transformations. You can write these commands using your knowledge of Pandas – they are the **same** in Dask (the only difference is that in Dask you have to add ***.compute()*** at the end of the command to request that the calculation take place; recall that Dask will parallelize over multiple-cores unlike Pandas).

Show all the outputs for Task 9 – don't call head(), tail(), sample() etc. Show each sub-task in a different cell, and use Markdown L3 heading (not comment) to label each sub-task.

a. Calculate the number of non-missing values in each column (1 point)

b. Calculate the counts of each of the categorical values in the Cancelled column (1 point)

c. Calculate the max value of the DepDelay column (1 point)

d. Find and print the columns for all the rows where the flights are Cancelled (1.5 points)

e. How many observations are there where the DayOfWeek column equals 1? (1 point)

f. How many non-cancelled flights were taken from each Origin airport? (1.5 points)
   BONUS: Compute the ratio of non-cancelled to total flights (for that origin) for each Origin. Need to compute numerator (Series of 3 values) and denominator (Series of 3 values) separately and then divide the two Series.

g. Calculate the average departure delay for each Origin airport by DayOfWeek (1.5 points)

h. For each of the three NYC area airports as the Destination, compute the means of all columns. You may use the .isin() method of Series (answer will have 3 rows) (1.5 points)

i. For each year, compute the maximum values of ActualElapsedTime and Distance. (2 points)

HINT for task 9i: For this last one, you will need to use the .dt accessor available on columns that contains date content. In our Dask DataFrame, we have a column called 'Date' that we specified is a date column, when we used the parse_dates parameter of dd.read_csv(). To access a component of the date, say year, we first call the .dt accessor which then allows us to select attributes (.dt.year, .dt.month, .dt.day etc). Next call DataFrame.groupby() on the year values, and select only the two columns that we need and obtain the maximum.

ASIDE on Sorting:
When performing transformations, recall that when using Pandas, we were able to call sorting methods sort_index() and sort_values(). These methods do not exist for distributed data structures – as that operation would require shuffling data across nodes of the cluster – which is very very expensive (might change in future).

NOTE:
If you compare the same operations in Pandas and Dask, and find Dask to be slower, then you may be using too many partitions of your data. Clearly summary operations such as computing the max/min/mean/var/std etc require searching within partitions and then across – the more partitions, the more overhead there is, and can be slower.

On the other hand, transformations such as taking logs of a column should be less affected by the overhead, and will benefit from parallelization.

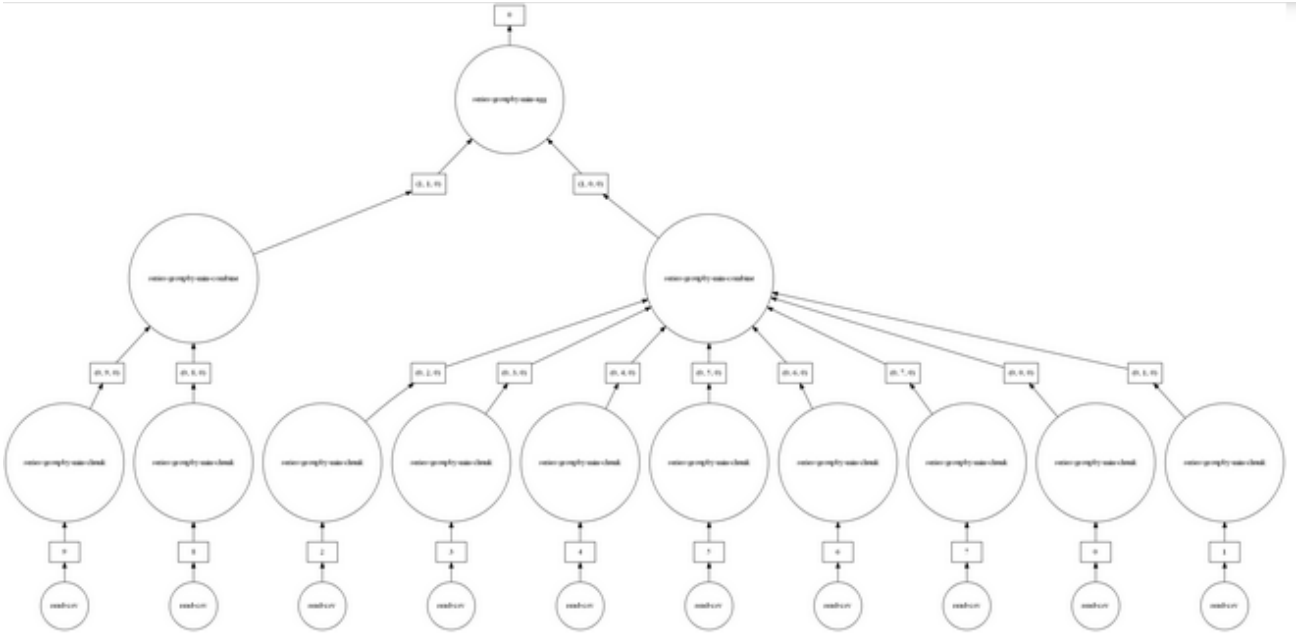
In general, parallelization will benefit more complex tasks, and for simpler tasks, we will be better off using a single-core.

10. Visualizing Dask Task Graphs

After we write commands in Dask, we can view the underlying task graph by calling .visualize().

Let us visualize the dask graph for a command to calculate the minimum of the 'ArrDelay' column by destination ('Dest' column).

```
ddf.groupby("Dest")["ArrDelay"].min().visualize()
```



Do you see the output? You need to have installed graphviz and python-graphviz to see this.

11. Use Distributed Task Scheduler: Set up a cluster and ProgressBar for diagnostics (1 point)

Set up a cluster.

Import Client class from dask.distributed module

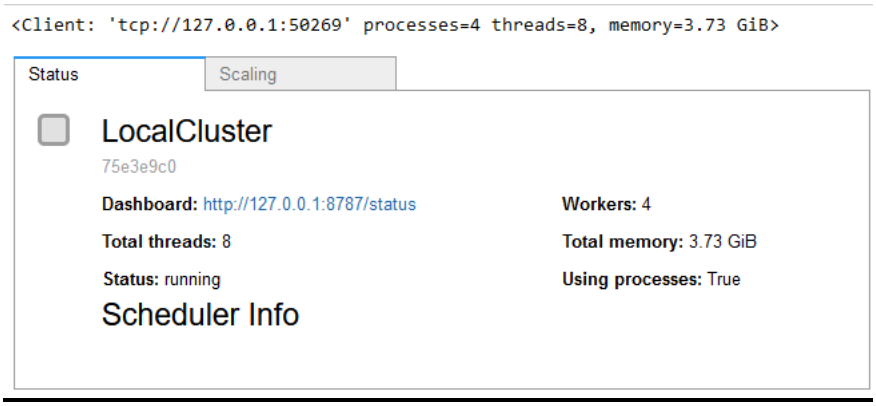
```
from dask.distributed import Client
```

Set up a new local cluster (on a single machine) with 4 workers and 2 threads per worker

```
from dask.distributed import Client

client = Client(n_workers = 4 , threads_per_worker=2, memory_limit = '1GB')
print(client)
client.cluster
```

This produces the following output (with a link to the Dask Dashboard)



OR (sometimes this output if things are not set up properly)

```
<Client: 'tcp://127.0.0.1:64332' processes=4 threads=8, memory=4.00 GB>

VBox(children=(HTML(value='<h2>LocalCluster</h2>'), HBox(children=(HTML(value='\n<div>\n
ed>\n    …
```

> A good rule of thumb for choosing number of threads per Dask worker is to choose the square root of the number of cores per worker.
>
> - In general more threads per worker are good for a program that spends most of its time in NumPy, SciPy, Numba, etc., and fewer threads per worker are better for simpler programs that spend most of their time in the standard Python interpreter (also due to GIL, which restricts to one thread per core).

Creating a new cluster also provides a dashboard for it at the URL listed (http://localhost:8787/status OR http://127.0.0.1:8787/status). But the server may be served elsewhere if this port is taken. The address of the dashboard will be displayed if you are in a Jupyter Notebook.

Click on this dashboard to view it now and again, after running a distributed task next.

In previous operations the default scheduler was used (Threaded Scheduler for Dask DataFrames), but ***once a distributed cluster is set up, it will be used by default for future computations***.

Run a previously run command again (from task 10):

```
ddf.groupby("Dest")["ArrDelay"].min().compute()
```

Then click on the dashboard URL to see how the operations are parallelized (*if you can do so while the above command is still running, you will see it live*).

Set up a ProgressBar

We can also import ProgressBar from dask.diagnostics, which will give us access to a live progress bar for our computations, which can be helpful for really long calculations.

```
from dask.diagnostics import ProgressBar
ProgressBar().register()
```

We can see the ProgressBar in action in the next several tasks (which take longer).

In summary, The Dask distributed scheduler provides live feedback in two forms:
- An interactive **dashboard** containing many plots and tables with live information
- A **progress bar** suitable for interactive use in consoles or notebook

12. Using a different scheduler for specific tasks (1 point)

If after setting up a cluster ('distributed' scheduler), you wish to run specific commands using a different scheduler – say 'threads' or 'processes', you can do so, by setting a parameter within dask.DataFrame.compute() called scheduler.

```
x = ddf.groupby("Dest")["ArrDelay"].min()

x.compute(scheduler='threads')
```

And when you want to execute a specific block of code using different configuration, such as a different scheduler, use **with**

```
with dask.config.set(scheduler='processes'):
    print(x.compute())
```

TIP:
If you obtained and registered a ProgressBar, it automatically shows you a progress rate for the calculation, and also computes how much time the operation takes.

You may have observed that the Threads scheduler is a tad bit faster. Processes scheduler is faster for Python tasks, whereas Threads is more efficient for Pandas, Numpy type tasks (because it can use multiple threads per core with less overhead for small-medium tasks).

Nowadays, it is recommended to use the distributed scheduler even if you will only be performing all tasks on a single machine (and using threads for parallelization).

Now, you understand the basics of Dask and Dask DataFrames!

E. MACHINE LEARNING WITH DASK (6 points)

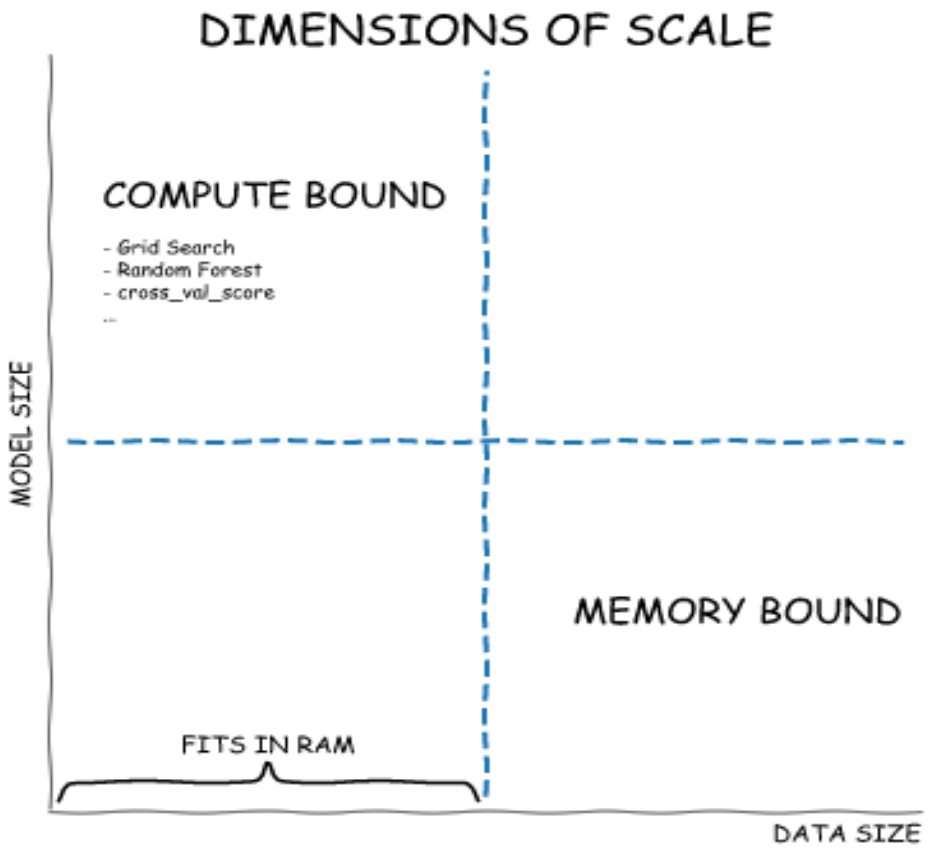
Dask provides resources for parallel and distributed Machine Learning.

Types of scaling

There are a couple of distinct scaling problems you might face in data science problems. The scaling strategy depends on which problem you're facing.

1. **CPU-bound (compute-bound)**: Data fits in RAM, but training takes too long.
    - For example – trying out many hyperparameter combinations, applying a large ensemble of many models, etc.

2. **Memory-bound**: Data is larger than RAM, and sampling isn't an option
    - This is akin to IO-bound problems as we have to move data in and out of memory (such as from disk)



- For in-memory tasks (data fits in memory))
    - just use scikit-learn (or another single-core ML library).

- For compute-bound tasks / large models (data fits in memory but computation is complex and time consuming), use `dask_ml.joblib` and your favorite scikit-learn estimator
    - The first kind of scaling challenge comes from your models growing so large or complex that it affects your workflow (shown along the vertical axis above). Under this scaling challenge tasks like model training, prediction, or evaluation steps will (eventually) complete, but they just take too long. You've become compute-bound.
        - For e.g., hyperparameter tuning using GridSearchCV and RandomizedSearchCV
    - To address these challenges you'd continue to use the collections we know (like the NumPy ndarray, Pandas DataFrame) and use a Dask Cluster to parallelize the workload on multiple cores.
    - The parallelization can occur either through
        - Dask's **Joblib** backend to parallelize Scikit-Learn directly on multiple cores of a machine (by setting the 'n_jobs' parameter on some estimators) or
        - using one of **Dask-ML**'s estimators (this is a different library of estimators similar to scikit-learn but made to operate on multicores).

- For large datasets (memory-bound issues), use `dask_ml` estimators
    - The second type of scaling challenge people face is when their datasets grow larger than RAM (shown along the horizontal axis above). Under this scaling challenge, even loading the data into NumPy or Pandas becomes impossible. Using joblib is not helpful.
    - To address these challenges, you'd use one of Dask's Array, DataFrames or Bag combined with one of Dask-ML's estimators that are designed to work with Dask collections. For example you might use Dask Array and one of the preprocessing estimators in dask_ml.preprocessing, or one of the ensemble methods in dask_ml.ensemble.
    - Most of the familiar ML classes from sklearn are also available in dask_ml under a similar model/submodule structure. More are being added continually. This makes it easy to use if you already know sklearn.

In the below tasks, we will use a classification task to demonstrate the three techniques. For these tasks, make sure to run the analyses long enough to produce results (visible outputs).

1. Generate random data

make_classification() generates random samples of data with continuous features and classification output (default = 2 classes).

```
from sklearn.datasets import make_classification

X, y = make_classification(n_samples=10000, n_features=4, random_state=0)
```

View some values from X and y (Note: we only have training data here)

```
X[:8]   y[:8]
```

NOTE: Technically, we should create separate training and testing data (as you have learned in the ML notebook, but to kep things simple in this How, we will not worry about testing data, and only use training data)

2. Fit a SupportVector Classifier (1 point)

Import model

```
from sklearn.svm import SVC
```

Create an estimator (use most of its default parameter values – please see the SVC documentation to know what they are) and set random_state=42. Then, we fit the model to data.

```
estimator = SVC(random_state=42)
estimator.fit(X, y)
```

Inspect the *learned parameters* – here, the support vectors, that are estimated from the data. Since there are four attributes/features, we will get 4 values. Let's print the first 5 rows.

```
estimator.support_vectors_[:5]
```

Check the mean classification accuracy of the model on training data (for the two classes). The .score() method provides this.

```
estimator.score(X, y)
```

3. Setting Hyperparameters

Most models have *hyperparameters*. They affect the fit, but are specified up front instead of learned during training. SVC has a few parameters- we will use 'C' (which is a regularization parameter used to prevent overfitting, default value is 1.0) and 'shrinking' (default = True).

```
estimator = SVC(C=0.00001, shrinking=False, random_state=0)
estimator.fit(X, y)
estimator.support_vectors_[:4]
```

Check the mean classification accuracy of the model on training data.

```
estimator.score(X, y)
```

We can observe that the model results are different than before – they can be worse or better since we are randomly selecting hyperparameters. But how to select the best values of hyperparameters? We can use a brute-force approach by using **GridSearchCV**, which allows us to send a list of values to try for hyperparameters and it will apply them combinatorially.

4. Applying GridSearchCV on a single core (1 point)

Import GridSearchCV

```
from sklearn.model_selection import GridSearchCV
```

Let us define an SVC estimator and set a few initial hyperparameter values using the constructor – we will not tune these. Then specify a parameter grid as a dict of hyperparameters we want to tune, with the parameter name as key and the values to try for each as a list. Notice, there is one pair of key-value for each hyperparameter that we want to tune. We will tune two, 'C' and "kernel'. You can add more, but it will take more time.

```
estimator = SVC(gamma='auto', probability=True,  random_state=42 )

param_grid = {
    'C': [0.001, 10.0],
    'kernel': ['rbf', 'poly', 'linear'],
}
```

Then we obtain an instance of GridSearchCV class. This class allows to specify a grid of values to try for the desired hyperparameters, and also applies cross-validation. It has two required parameters – the estimator such as SVC and the grid of parameters and values to try, and we can also initialize a few optional hyperparameters. Below, we specify we want 2 folds (cv=2), and set verbose=2 to ask for noisy outputs (that means, we will see intermediate information printed for each run). Lower (higher) value would reduce (increase) the amount of information.

```
grid_search = GridSearchCV(estimator, param_grid, verbose=2, cv=2)
```

After that, we can fit the model. Let us time the operation using %timeit (will run the command many times and calculate average time – this takes longer) or %time (run once) to time the cell.

```
%time   grid_search.fit(X, y)
```

The above will take a while (and possibly slow down your machine), since it is not parallelized (we are timing it so we can see the benefits of using a cluster to run the same commands next).

5. Applying GridSearchCV on multi-cores (native Scikit parallelism using Joblib) (2 points)

Scikit-Learn has nice *single-machine* parallelism, via **Joblib** module. Any scikit-learn estimator that can operate in parallel exposes an **n_jobs** keyword. This controls the number of CPU cores that will be used. Setting n_jobs = -1 will use all available cores, or you can select a number.

p.s. if you don't know how many cores you have, you can check that.

```
import os

n_cpu = os.cpu_count()
print("Number of CPUs in the system:", n_cpu)
```

NOTE: we do not have to explicitly import the Joblib module prior to using n_jobs, as scikit-learn internally manages its own dependencies (will inherit this package implicitly).

Now, let us repeat the GridSearchCV code but save it in a new variable called grid_search_MC. We will add one additional parameter n_jobs = -1 to the estimator.

```
grid_search_MC = GridSearchCV(estimator,
                              param_grid,
                              verbose=2,
                              cv=2,
                              n_jobs=-1)
```

Run the timing operation again while fitting the model.

```
%time  grid_search_MC.fit(X, y)
```

Observe the speed differences between fitting the model in this Task 5 and the previous Task 4 (starker when using %timeit - iteratively).

Both should produce the same best result for the best hyperparameters – the only difference is in the speed. Let us now view the tuned best hyperparameters, and the best score. Use the following command.

```
grid_search_MC.best_params_, grid_search_MC.best_score_
```

6. Parallelism with Dask (can use single or multi-machine) (2 points)

Many Scikit-Learn algorithms are written for parallel execution using Joblib, which natively provides thread-based and process-based parallelism (on a single machine). Joblib is what backs the n_jobs= parameter in normal use of Scikit-Learn. Dask can scale these Joblib-backed algorithms out to a cluster of machines (or multiple cores on a local machine) by providing an

alternative Joblib backend. [Similarly, you can also choose others such as Spark as a backend for Joblib.]

Dask can talk to scikit-learn (via joblib module) so that your *Dask cluster* can be used to train a model. If you run these models on a laptop, it will take quite some time, and the CPU usage may be near 100% for the duration. To run faster, we can use multi-machine distributed cluster (when available, *not done here*).

In this case, when trying to use other backends, we will need to import the joblib module.

```
import joblib
```

Set up parameter grid (if you add more params, it will take longer; you can reduce the C values).

```
param_grid = {
    'C': [0.001, 0.1, 1.0, 2.5, 5., 10.0, 100., 500., 1000.],
    # Uncomment this to perform larger Grid searches on a cluster
    #'kernel': ['rbf', 'poly', 'linear'],
    #'shrinking': [True, False],
}
```

Set up a new GridSearchCV object with n_jobs set to the number of cores to use.

```
grid_search_Dist = GridSearchCV(estimator,
                                param_grid,
                                verbose=2,
                                cv=5,
                                n_jobs=-1)
```

Next, we will set up and run a parallel job Dask and joblib.

Joblib provides three different parallel pbackends by default (without needing to install any other modules): loky (default), threading, and multiprocessing.

- 'loky': single-host, process-based parallelism (used by default),
- 'threading': single-host, thread-based parallelism,
- 'multiprocessing': legacy single-host, process-based parallelism.

NOTE: 'loky' is recommended to run functions that manipulate Python objects (which do not release the Global Interpreter Lock or GIL as we learned earlier in this file). 'threading' is a low-overhead alternative that is most efficient for functions that release the GIL: e.g. I/O-bound code or CPU-bound code, which only make few calls to native Python code that explicitly releases the GIL.

Earlier when we set n_jobs = -1 it was using "loky" backend with multiple processes by default.

If you have additional modules installed that provide alternate backends for parallel processing, you can specify them as backends as well. One such backend is Dask (others include Ray, Distributed)

When we want to specify a custom behavior for only a certain block of code, we use the **with** operator. Here, we will specify that we want to use the "dask" parallel backend only for this block of code. The other blocks outside of the with block will continue to use the default joblib backends.

Call the parallel_backend() function of the joblib module and specify "dask" as the first parameter. Then specify your scikit-learn code inside the with block.

> Once a Dask cluster is set up (local-cluster or multi-machine cluster), as we did earlier in Section E, it will be used when we call the joblib.parallel_backend("dask") command as below. Here, it will use the Local cluster that we set up earlier.

Let us also perform a cell timing operation (so we need to use %%time or %%timeit – latter will take much longer). Putting it all together, we have:

```
%%time

with joblib.parallel_backend("dask"):
    grid_search_Dist.fit(X, y)
```

Be sure to write the above in a separate cell. %% is a cell magic function, it will apply to the whole cell- so only write the code that you want to time, with %%time at the beginning of the cell.

View all the grid search results as a dataframe (because we used a cross-validation estimator, we will have as many results as the number of folds * number of hyperparameter combination to try).

```
pd.DataFrame(grid_search_Dist.cv_results_).head()
```

Obtain the best hyperparameters and the best score – i.e., learned parameters and accuracy.

```
grid_search_Dist.best_params_, grid_search_Dist.best_score_
```

We can also obtain other performance measures as we did with regression (see available metrics in the sklearn.metrics module). https://scikit-learn.org/stable/modules/model_evaluation.html.

We can obtain a summary of various measures (precision, recall, f1-score and support) using the sklearn.metrics.classification_report class. This class needs two parameters – the true y values and the predicted y values. Let us compute the predictions and save them first.

```
y_pred = grid_search_Dist.predict(X)
```

Then import and use classification_report to obtain common classification statistics such as precision, recall, f1-score for both the classes (0 and 1)

```
from sklearn.metrics import classification_report

print(classification_report(y, y_pred))
```

7. Close the Client

This will automatically happen when the program is closed, but you can also do so in the notebook whne you no longer need the client.

```
client.shutdown()
```

8. Dask-ML

Sometimes you'll want to train on a larger than memory dataset. For this we cannot apply Joblib, which can be applied to compute-bound problems. For the memory-bound problems, we can use the **dask-ml** library, which has implemented estimators that work well on dask arrays and Dataframes that may be larger than your machine's RAM. For this, **dask-ml** has estimators for many of the native scikit-learn estimators.

We will not consider this here (and is a suggestion to you for future studying in this area of big data). Next in this course, we will discuss other popular platforms for dealing with memory-bound problems: Hadoop and Spark.

DONE! Save and submit your notebook.

Bugs and warnings

Dask, being new-ish and constantly getting updates, we are likely to run into some issues. Hey, we aren't afraid of some warning and bugs, are we? So here is a collection of some "known" issues (warnings) you might run into.

Section D. Task 9h.

Dask changed recently and now may produce this warning:

```
C:\Users\u0625859\Anaconda3\envs\daskpy\lib\site-packages\dask\utils.py:1022:
FutureWarning: The default value of numeric_only in DataFrameGroupBy.sum is d
eprecated. In a future version, numeric_only will default to False. Either sp
ecify numeric_only or select only columns which should be valid for the funct
ion.
  return getattr(__obj, self.method)(*args, **kwargs)
```

It's a known bug that is being worked on by Dask. The warning suggests to set parameter numeric_only = True in DaskGroupBy.mean() - but we cannot actually do that as it does not allow that parameter yet.

Here is a fix.

```
# to suppress the FutureWarning

# use ddf.select_dtypes to select numeric columns only and get a list of numeric columns
numcols = ddf.select_dtypes(include='number').columns
numcols
# after groupby obtain just a ddf with numeric columns before calling .mean()
ddf[ddf["Dest"].isin(['JFK', 'EWR', 'LGA'])].groupby('Dest')[numcols].mean().compute()
```

Sec D. Task 10

Having trouble running task 10, "Visualizing Dask task graphs" especially on a Mac.

Try creating a new cell above imports and add this command, " pip install graphviz". This should help in running Task 10. Here is a screenshot for your assistance

```
In [40]: pip install graphviz
Collecting graphviz
  Downloading graphviz-0.20.1-py3-none-any.whl (47 kB)
     ---------------------------------------- 47.0/47.0 KB 1.5 MB/s eta 0:00:00
Installing collected packages: graphviz
Successfully installed graphviz-0.20.1
WARNING: You are using pip version 22.0.4; however, version 23.0.1 is available.
You should consider upgrading via the '/Users/saikrishna/.pyenv/versions/3.10.4/bin/python -m pip install --upgrade p
ip' command.
Note: you may need to restart the kernel to use updated packages.
```

Sec D. Task 11

a) Running the cell with the code to create a distributed client again, might produce a warning such as below if there is another active cluster (so try not to do that):

```
C:\Users\u0625859\Anaconda3\envs\daskpy\lib\site-packages\distributed\node.py:177: UserWarning:
Port 8787 is already in use.
Perhaps you already have a cluster running?
Hosting the HTTP server on port 50562 instead
  warnings.warn(
```

If you encounter this, you can run Section E, Task 7 (client.shutdown()) to manually shutdown the client() before opening a new one.

b) When setting up a cluster, you may see the below message

```
distributed.diskutils - INFO - Found stale lock file and directory '
```

This should only happen if your previous process running the cluster died unexpectedly. If a previous cluster died without giving the worker time to close gracefully (e.g. a SIGKILL) distributed will notice "stale" locks of earlier worker instances once started and is cleaning this up.

No need to worry, Dask provides a warning message and continues to take care of the problem (purging these old files).

Section D. Task 12

may produce a warning that you are using a single machine scheduler when you set scheduler =- "threads". It is okay, we are doing something not very efficient, so Dask is warning us.

Sec E. Task 6

When you run fit using joblib.parallel_backend("dask")

```
distributed.worker - WARNING - Unmanaged memory use is high. This may
indicate a memory leak or the memory may not be released to the OS; see
https://distributed.dask.org/en/latest/worker.html#memtrim for more
information. -- Unmanaged memory: 721.36 MiB -- Worker memory limit: 0.93 GiB
```

This is a bit problematic, and likely to occur on Mac and Linux, especially due to how they manage memory. One solution is to reduce the hyperparameters to try.

Other solutions (it is technical, be warned): https://distributed.dask.org/en/stable/worker-memory.html

Sec E. Task 7

When calling client.shutdown(), you may see the below

```
distributed.client - ERROR - Failed to reconnect to scheduler after 30.00
seconds, closing client
```

It is the last step. The client will be closed anyways (looks like it was already shutdown before calling shutdown() again).

Warnings about Unused memory is high

If you're getting 'unused memory is high' warnings and errors (seen usually on a Mac - due to how to Mac/ Linux OS handle memory), please see the following resources about what to do to fix.

See here for unique Linux and MacOS memory behavior:
https://distributed.dask.org/en/stable/worker-memory.html

Also here: https://www.coiled.io/blog/tackling-unmanaged-memory-with-dask

Using **jemalloc** package can help with this.

The steps to install jemalloc can be found in the first link (also noted below).

On macOS:

```
conda install jemalloc
DYLD_INSERT_LIBRARIES=$CONDA_PREFIX/lib/libjemalloc.dylib dask worker <...>
```

Alternatively on macOS, install globally with homebrew (if you have it installed; it is like conda for Mac):

```
brew install jemalloc
DYLD_INSERT_LIBRARIES=$(brew --prefix jemalloc)/lib/libjemalloc.dylib dask
worker <...>
```