Python for High Performance Data Analytics

Prerequisites

- Basic proficiency in Python (variables, flow control, functions)
- Basic grasp of descriptive statistics (such as minimum, maximum, median, arithmetic mean…)
- Basic knowledge of NumPy
- Basic knowledge of some plotting package (Matplotlib, Seaborn, Holoviz…)

20 min | filename

Tabular data (aka Dataframes)

Questions

- What are series and dataframes?
- What do we mean by tidy and untidy data?
- What packages are available in Python to handle dataframes?

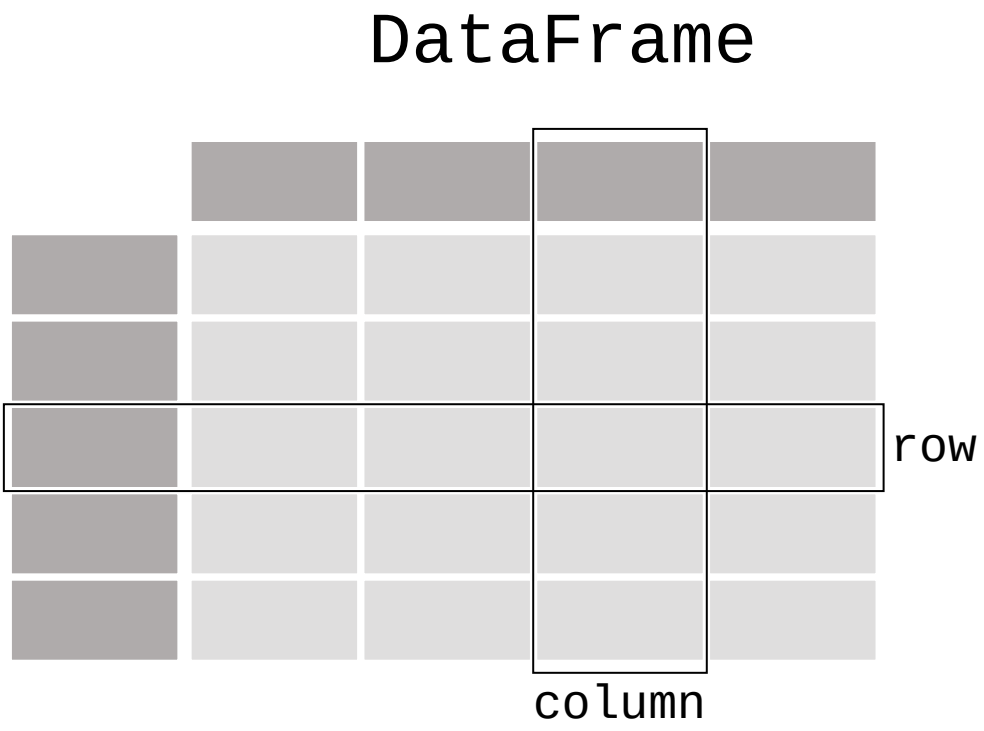
Objectives

- Learn how to manipulate dataframes in Pandas
- Lazy and eager dataframes in Polars

This episode will give an introduction to the concepts of *Series* and *DataFrame* and how they can be manipulated using different Python packages.

Series and dataframes

A collection of observations (e.g. a time series or simply a set of observations of a feature of a phenomenon) can be represented by a homogeneous vector, i.e. an array where all the elements are of the same type. This is known as a *Series* in many frameworks. Several series (of different types) can be used as columns of a tabular structure called a *Dataframe*, as depicted in the figure below



Tidy vs untidy dataframes

Let us look at the following two dataframes:

Untidy format | Tidy format

	Runner	400	800	1200	1500
0	Runner 1	64	128	192	240
1	Runner 2	80	160	240	300
2	Runner 3	96	192	288	360

Most tabular data is either in a tidy format or a untidy format (some people refer them as the long format or the wide format). The main differences are summarised below:

- In untidy (wide) format, each row represents an observation consisting of multiple variables and each variable has its own column. This is intuitive and easy for us to understand and make comparisons across different variables, calculate statistics, etc.
- In tidy (long) format , i.e. column-oriented format, each row represents only one variable of the observation, and can be considered “computer readable”. When it comes to data analysis using Pandas, the tidy format is recommended:
- Each column can be stored as a vector and this not only saves memory but also allows for vectorized calculations which are much faster.
- It’s easier to filter, group, join and aggregate the data.

Imagine, for example, that you would like to compute the speed as `speed=distance/time`. The untidy format would make this much clunkier, as:

- The distances are encoded as column names, not as data points (rows)
- The speed would have to be stored in a new dataframe since it would not fit in that data structure.

In comparison, in a tidy dataframe, this computation would be a simple operation between two columns.

> **Tip**
>
> Recovering a wide dataframe from a tidy one is commonly referred to as *pivoting*. Most dataframe libraries provide a `pivot()` or `pivot_table` function.

Pandas & Polars

Historically, Pandas has been the go-to package to handle dataframes in Python. It is based on NumPy (each column is a Numpy vector) and has been the traditional workhorse for tabular data, with a stable API and a large ecosystem built around it, including the Seaborn statistical plotting framework. More recently, Polars was introduced as a more modern and faster alternative to handle dataframes. It is written in Rust and supports out of the box out of core evaluation (i.e. does not need loading the whole dataset in memory), lazy evaluation of queries and automatically uses multiple threads. Moreover, experimental GPU support is available through cuDF. In the remainder of this episode, the NYC taxi will be used to showcase how datasets can be accessed, summarised and manipulated in both Pandas and Polars. The dataset can be

uet format from the link above (the file for the month of January was used in taset contains information about taxi trips performed in New York, such as the ID of the vendor, the total fare, pickup and drop-off time and location (expressed as an ID), type of payment, whether additional fees were charged and more.

Opening a dataset

Assuming the file is called `yellow_tripdata_2025-01.parquet`, the dataset can be opened as:

Pandas | Polars

```
import pandas as pd
df = pd.read_parquet("yellow_tripdata_2025-01.parquet")
```

Description and summarisation

We can get a first understanding of the contents of a dataframe by printing the first few lines, the "schema" (i.e. the number and type of each column) and summary statistics as follows:

Pandas | Polars

```
df.head()
```

Output

```
   VendorID tpep_pickup_datetime tpep_dropoff_datetime  ...  congestion_surcharge  Airport_fee  cbd
0         1  2025-01-01 00:18:38   2025-01-01 00:26:59  ...                   2.5          0.0
1         1  2025-01-01 00:32:40   2025-01-01 00:35:13  ...                   2.5          0.0
2         1  2025-01-01 00:44:04   2025-01-01 00:46:01  ...                   2.5          0.0
3         2  2025-01-01 00:14:27   2025-01-01 00:20:01  ...                   0.0          0.0
4         2  2025-01-01 00:21:34   2025-01-01 00:25:06  ...                   0.0          0.0
```

```
df.info()
```

Output

```
RangeIndex: 3475226 entries, 0 to 3475225
Data columns (total 20 columns):
 #   Column                 Dtype
---  ------                 -----
 0   VendorID               int32
 1   tpep_pickup_datetime   datetime64[us]
 2   tpep_dropoff_datetime  datetime64[us]
 3   passenger_count        float64
 4   trip_distance          float64
 5   RatecodeID             float64
 6   store_and_fwd_flag     object
 7   PULocationID           int32
 8   DOLocationID           int32
 9   payment_type           int64
 10  fare_amount            float64
 11  extra                  float64
 12  mta_tax                float64
 13  tip_amount             float64
 14  tolls_amount           float64
 15  improvement_surcharge  float64
 16  total_amount           float64
 17  congestion_surcharge   float64
 18  Airport_fee            float64
 19  cbd_congestion_fee     float64
dtypes: datetime64[us](2), float64(13), int32(3), int64(1), object(1)
memory usage: 490.5+ MB
```

```
df.describe()
```

Output

```
           VendorID        tpep_pickup_datetime       tpep_dropoff_datetime  ...  congestion_surcha
count  3.475226e+06                     3475226                     3475226  ...          2.935077e
mean   1.785428e+00  2025-01-17 11:02:55.910964  2025-01-17 11:17:56.997901  ...          2.225237e
min    1.000000e+00         2024-12-31 20:47:55         2024-12-18 07:52:40  ...         -2.500000e
25%    2.000000e+00         2025-01-10 07:59:01  2025-01-10 08:15:29.500000  ...          2.500000e
50%    2.000000e+00         2025-01-17 15:41:33         2025-01-17 15:59:34  ...          2.500000e
75%    2.000000e+00         2025-01-24 19:34:06         2025-01-24 19:48:31  ...          2.500000e
max    7.000000e+00         2025-02-01 00:00:44         2025-02-01 23:44:11  ...          2.500000e
std    4.263282e-01                         NaN                         NaN  ...          9.039932e

[8 rows x 19 columns]
```

Indexing

We can index data in the dataframe as follows:

Skip to content

Pandas | Polars

```
# With this we can select a column
df['VendorID'] # Could also be df.VendorID
```

Output

```
0          1
1          1
2          1
3          2
4          2
          ..
3475221    2
3475222    2
3475223    2
3475224    2
3475225    2
```

```
# Get a row
df.iloc[1000,:]
```

Output

```
VendorID                                2
tpep_pickup_datetime     2025-01-01 00:08:06
tpep_dropoff_datetime    2025-01-01 00:16:20
passenger_count                        4.0
trip_distance                         1.53
RatecodeID                             1.0
store_and_fwd_flag                       N
PULocationID                           114
DOLocationID                            90
payment_type                             1
fare_amount                           10.0
extra                                  1.0
mta_tax                                0.5
tip_amount                            2.25
tolls_amount                           0.0
improvement_surcharge                  1.0
total_amount                         17.25
congestion_surcharge                   2.5
Airport_fee                            0.0
cbd_congestion_fee                     0.0
Name: 1000, dtype: object
>>> df.iloc[1000,:]
VendorID                                2
tpep_pickup_datetime     2025-01-01 00:08:06
tpep_dropoff_datetime    2025-01-01 00:16:20
passenger_count                        4.0
trip_distance                         1.53
RatecodeID                             1.0
store_and_fwd_flag                       N
PULocationID                           114
DOLocationID                            90
payment_type                             1
fare_amount                           10.0
extra                                  1.0
mta_tax                                0.5
tip_amount                            2.25
tolls_amount                           0.0
improvement_surcharge                  1.0
total_amount                         17.25
congestion_surcharge                   2.5
Airport_fee                            0.0
cbd_congestion_fee                     0.0
```

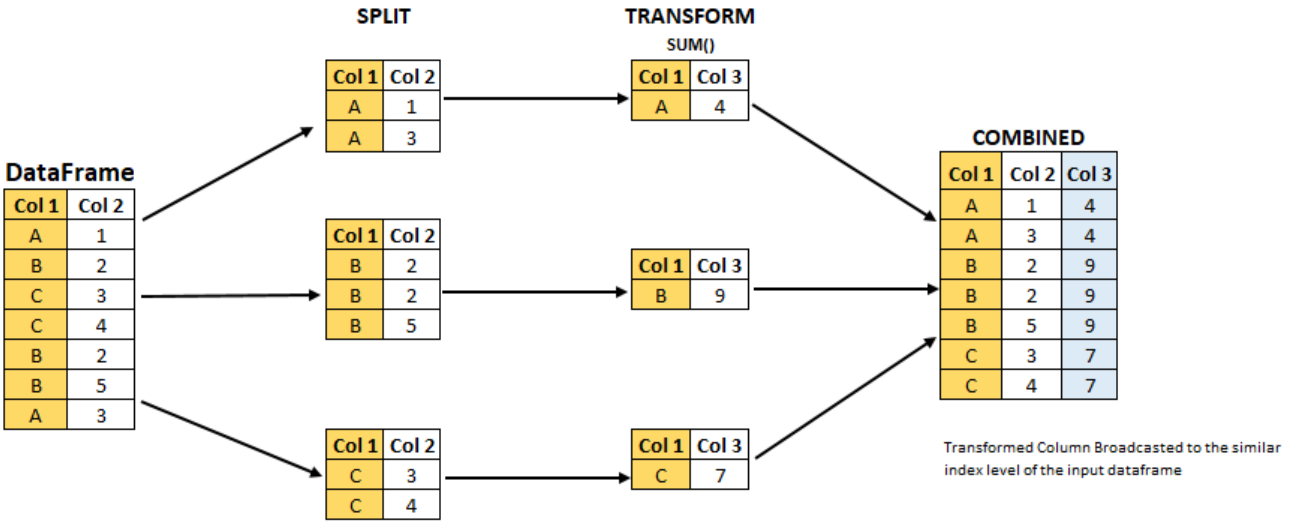
In both cases, a similar syntax can be used to do in-place modification (e.g. `df[row][column]=...`). Please note that this kind of replacement carries a big performance penalty, which is designed to do column-wide operations with minimal overhead. This is commonly achieved through the expression API, as detailed in the next section.

Common workflows

It is quite common to stratify (i.e. divide the samples into a number of groups based on categorical variables) to produce descriptive statistics (i.e. statistics that provide a summary of the samples and do not aim to predict anything regarding the population it comes from). This is commonly achieved through a `group-by` workflow, where the following happens:

- Splitting: data is partitioned into different groups based on some criterion
- Applying: applying a function/performing a calculation to each group
- Combining: assembling a dataframe (of potentially any size) with the results.

This is type of workflow is represented below.



SOURCE: EARTH AND ENVIRONMENTAL DATA SCIENCE

As an example, let us try to to compute the total fare for each hour, split by payment type.

Pandas | Polars

```
#First let us extract the hour from the tpep_pickup_datetime column
df["hour"] = df['tpep_pickup_datetime'].dt.hour

hourly_fare = (
  df.groupby(['hour', 'payment_type'], observed=False)['fare_amount']
  .sum()
  .reset_index()
  .sort_values(['hour', 'payment_type'])
```

Output

```
     hour  payment_type  fare_amount
0       0             0    352227.86
1       0             1   1088201.12
2       0             2    156546.07
3       0             3      3537.91
4       0             4      3941.24
..    ...           ...          ...
116    23             0    534063.76
117    23             1   1618143.37
118    23             2    219991.22
119    23             3      4765.54
120    23             4      3293.61
```

The `groupby` statement is used to stratify the `fare_amount` column by hour and payment type. Then the amounts per hour and type get summed and sorted according to time and payment type.

Idiomatic Polars

Polars introduces a few variations to dataset operations compared to the traditional Pandas approach. In particular, a domain-specific language (DSL) was developed, where *expressions* are written to represent dataset operations and *context*s provide the environment where they produce a result.

Expressions

Let's say that we created a `trip_duration_sec` column in our NYC cab database and, given the `trip_distance` column, we want to compute the average speed. In Polars, this can be achieved with:

```
pl.col('trip_distance') / pl.col(`trip_duration_sec`)
```

This is a lazy representation of an operation we want to perform, which can be further manipulated or just printed. For it to actually produce data, a *context* is needed.

Contexts

The same Polars expression can produce different results depending on the context where it is used. Four common contexts include:

- `select`

- `filter`
- `group_by`

Both `select` and `with_columns` can produce new columns, which may be aggregations, combinations of other columns, or literals. The difference between the two is that `select` only includes the columns contained in its input expression, whereas `with_columns` returns a new dataframe which contains all the columns from the original dataframe and the new ones created by the expression. To exemplify, using our earlier example of computing the average speed during a trip, using `select` would yield a single column, whereas `with_columns` would return the original dataframe with an additional column called `trip distance`:

```
df.select(pl.col('trip_distance')/pl.col('trip_duration_sec')*3600)
shape: (3_475_226, 1)
┌─────────────────┐
│ trip_distance   │
│ ---             │
│ f64             │
╞═════════════════╡
│ 11.497006       │
│ 11.764706       │
│ 18.461538       │
│ 5.60479         │
│ 11.207547       │
│ …               │
│ 13.68899        │
│ 19.42398        │
│ 9.879418        │
│ 9.339901        │
│ 12.781395       │
└─────────────────┘
```

```
df.with_columns((pl.col('trip_distance')/pl.col('trip_duration_sec')*3600).alias("avg_s
eed_mph"))
shape: (3_475_226, 22)
```

VendorID	tpep_pickup_datetime	tpep_dropoff_datetime	passenger_count	…	Airport_fee	cbd_congestion_fee	trip_duration_sec	avg_mp
i32	---	---	i64		f64	---	---	---
	datetime [µs]	datetime [µs]				f64	i64	f64
1	2025-01-01 00:18:38	2025-01-01 00:26:59	1	…	0.0	0.0	501	11.4
1	2025-01-01 00:32:40	2025-01-01 00:35:13	1	…	0.0	0.0	153	11.7
1	2025-01-01 00:44:04	2025-01-01 00:46:01	1	…	0.0	0.0	117	18.4
2	2025-01-01 00:14:27	2025-01-01 00:20:01	3	…	0.0	0.0	334	5.6
2	2025-01-01 00:21:34	2025-01-01 00:25:06	3	…	0.0	0.0	212	11.1
…	…	…	…	…	…	…	…	…
2	2025-01-31 23:01:48	2025-01-31 23:16:29	null	…	null	0.75	881	13.4
2	2025-01-31 23:50:29	2025-02-01 00:17:27	null	…	null	0.75	1618	19.4
2	2025-01-31 23:26:59	2025-01-31 23:43:01	null	…	null	0.75	962	9.8
2	2025-01-31 23:14:34	2025-01-31 23:34:52	null	…	null	0.75	1218	9.3
2	2025-01-31 23:56:42	2025-02-01 00:07:27	null	…	null	0.0	645	12.7

Skip to content

ext filters the rows of a dataframe based on one (or more) expressions which lean, e.g.

```
df.filter(pl.col('avg_speed_mph') < 1)
shape: (104_410, 22)
```

VendorID	tpep_pickup_datetime	tpep_dropoff_datetime	passenger_count	…	Airport_fee	cbd_congestion_fee	trip_duration_sec	avg_m
i32	datetime[μs]	datetime[μs]	i64		f64	f64	i64	f64
2	2025-01-01 00:37:43	2025-01-01 00:37:53	1	…	0.0	0.0	10	0.0
2	2025-01-01 00:57:08	2025-01-01 00:57:16	3	…	0.0	0.0	8	0.0
1	2025-01-01 00:27:40	2025-01-01 00:59:30	1	…	0.0	0.0	1910	0.0
2	2025-01-01 00:56:49	2025-01-01 00:56:54	4	…	0.0	0.0	5	0.0
1	2025-01-01 00:42:42	2025-01-01 00:42:44	0	…	0.0	0.0	2	0.0
…	…	…	…	…	…	…	…	…
1	2025-01-31 23:59:17	2025-02-01 00:03:43	null	…	null	0.75	266	0.0
1	2025-01-31 23:17:38	2025-01-31 23:35:58	null	…	null	0.75	1100	0.0
2	2025-01-31 23:39:25	2025-01-31 23:42:06	null	…	null	0.0	161	0.0
1	2025-01-31 23:30:42	2025-01-31 23:31:06	null	…	null	0.75	24	0.0
1	2025-01-31 23:10:25	2025-01-31 23:36:21	null	…	null	0.75	1556	0.0

The `group_by` context behaves like its Pandas counterpart.

Transformations

A `join` operation combines columns from one or more dataframes into a new dataframe. There are different joining strategies, which influence how columns are combined and what rows are included in the final set. A common type is the *equi* join, where rows are matched by a key expression. Let us clarify this with an example. The `df` dataframe does not include specific coordinates for each pickup and drop-off, rather only a `PULocationID` and a `DOLocationID`. There is a `taxy_zones_xy.csv` file that contains, for each `LocationID`, the latitude (X) and longitude (Y) of each location, as well as the name of zone and borough:

```
lookup_df = pl.read_csv('taxy_zones_xy.csv', has_header=True)
lookup_df.head()
```

```
┌────────────┬───────────┬──────────┬──────────────────────┬─────────────┐
│ LocationID ┆ X         ┆ Y        ┆ zone                 ┆ borough     │
│ ---        ┆ ---       ┆ ---      ┆ ---                  ┆ ---         │
│ i64        ┆ f64       ┆ f64      ┆ str                  ┆ str         │
╞════════════╪═══════════╪══════════╪══════════════════════╪═════════════╡
│ 1          ┆ -74.176786 ┆ 40.689516 ┆ Newark Airport       ┆ EWR         │
│ 2          ┆ -73.826126 ┆ 40.625724 ┆ Jamaica Bay          ┆ Queens      │
│ 3          ┆ -73.849479 ┆ 40.865888 ┆ Allerton/Pelham Gardens ┆ Bronx    │
│ 4          ┆ -73.977023 ┆ 40.724152 ┆ Alphabet City        ┆ Manhattan   │
│ 5          ┆ -74.18993  ┆ 40.55034  ┆ Arden Heights        ┆ Staten Island │
└────────────┴───────────┴──────────┴──────────────────────┴─────────────┘
```

This can be used to append these columns to the original df to have some form of geographical data as follows (e.g. for the `PULocationID`):

```
df = df.join(lookup_df, left_on='PULocationID', right_on='LocationID', how='left'
, suffix='_pickup')
```

In the line above, `left_on` is used to indicate the *key* in the original dataframe, `right_on` is used to specify the *key* in the `lookup_df` dataframe, `how=left` means that the columns from the second dataframe will be added to the first (and not the other way around) and `suffix` is what will be added to the names of the joined columns (i.e., df will contain columns called `X_pickup`, `Y_pickup`, `zone_pickup` and `borough_pickup`). More information on join operations can be found here.

Exercises

Joining geographical data

We have already seen how to add actual latitude and longitude for the pickups. Now do the same for the drop-offs!

Solution

```
df = df.join(lookup_df, left_on='DOLocationID', right_on='LocationID', how='left'
, suffix='_dropoff')
```

Feature engineering: enriching the dataset

We want to understand a bit more of the traffic in the city by creating new features (i.e. columns), in particular:

- Split the pickup datetime into hour, minute, day of the week and month to indentify daily, weekly and monthly trends
- Compute the average speed as an indicator of congestion (low speed -> traffic jam)
- Stratify the trip distance and fare by zone to identify how expensive different zones are. Below is a skeleton of the code, where some lines have been blanked out for you to fill (marked with `TODO:...`)

```
import polars as pl
raw_df = pl.read_parquet('yellow_tripdata_2025-01.parquet')
df = raw_df.with_columns([
    pl.col("tpep_pickup_datetime").dt.hour().alias("pickup_hour"),
    #TODO: do this for the minute
    pl.col("tpep_pickup_datetime").dt.day_of_week().alias("pickup_dow"),   # Mon=0 … Sun=6
    pl.col("tpep_pickup_datetime").dt.month().alias("pickup_month"),
    # Trip duration in seconds
    (pl.col("tpep_dropoff_datetime") - pl.col("tpep_pickup_datetime"))
        .dt.total_seconds()
        .alias("trip_duration_sec"),
])

df = df.with_column(
    #TODO: add expression for average velocity here
    .replace_nan(None)                        # protect against div-by-zero
    .alias("avg_speed_mph")
)

# Compute per-pickup-zone statistics once
zone_stats = (
    df.groupby("PULocationID")
      .agg([
          pl.mean("fare_amount").alias("zone_avg_fare"),
          #TODO: do the same for the trip distance here
          pl.count().alias("zone_trip_cnt"),
      ])
      .rename({"PULocationID": "pickup_zone_id"})   # avoid name clash later
)

# Join those stats back onto the original rows
df = df.join(zone_stats, left_on="PULocationID", right_on="pickup_zone_id", how="left")
```

While we haven't covered the `join` instruction earlier, its main role is to "spread" the `zone_stats` over all the rides in the original dataframe (i.e. write the `zone_avg_fare` on each ride in `df`). `join` has its roots in relational databases, where different tables can be merged based on a common column.

Skip to content

Solution

```
import polars as pl
raw_df = pl.read_parquet('yellow_tripdata_2025-01.parquet')
df = raw_df.with_columns([
    pl.col("tpep_pickup_datetime").dt.hour().alias("pickup_hour"),
    pl.col("tpep_pickup_datetime").dt.minute().alias("pickup_minute"),
    pl.col("tpep_pickup_datetime").dt.day_of_week().alias("pickup_dow"),    # Mon=0 … Sun=6
    pl.col("tpep_pickup_datetime").dt.month().alias("pickup_month"),
    # Trip duration in seconds
    (pl.col("tpep_dropoff_datetime") - pl.col("tpep_pickup_datetime"))
        .dt.seconds()
        .alias("trip_duration_sec"),
])

df = df.with_column(
    (
        pl.col("trip_distance") /
        (pl.col("trip_duration_sec") / 3600)   # seconds → hours
    )
    .replace_nan(None)                         # protect against div-by-zero
    .alias("avg_speed_mph")
)

# Compute per-pickup-zone statistics once
zone_stats = (
    df.groupby("PULocationID")
      .agg([
          pl.mean("fare_amount").alias("zone_avg_fare"),
          pl.mean("trip_distance").alias("zone_avg_dist"),
          pl.count().alias("zone_trip_cnt"),
      ])
      .rename({"PULocationID": "pickup_zone_id"})   # avoid name clash later
)

# Join those stats back onto the original rows
df = df.join(zone_stats, left_on="PULocationID", right_on="pickup_zone_id", how="left")
```

More feature engineering!

Similarly to the exercise above, define the following features in the data:

- `pickup_hour` extracted from `tpep_pickup_time`
- `is_weekend`, a Boolean value for each trip
- `avg_speed_mph`, exactly as before
- `tip_to_fare_ratio`, dividing the tip amount by the total fare. Be careful with division by 0
- `fare_per_mile`, dividing the total fare by the distance
- `dist_per_passenger`, the average distance travelled by each passenger (sum of all trip distances divided by number of trips)
- `speed_per_pickup_area`, the average velocity stratified by pickup location
- `dropoff_trip_count`, count of trips stratified per dropoff location

Skip to content

Solution

```
import polars as pl
raw_df = pl.read_parquet("yellow_tripdata_2025-01.parquet")
df = raw_df.with_columns([
    # 1. pickup_hour
    pl.col("tpep_pickup_datetime").dt.hour().alias("pickup_hour"),

    # 2. is_weekend (Sat=5, Sun=6)
    pl.col("tpep_pickup_datetime")
      .dt.day_of_week()
      .is_in([5, 6])
      .alias("is_weekend"),

    # 3. trip_duration_sec
    (pl.col("tpep_dropoff_datetime") - pl.col("tpep_pickup_datetime"))
        .dt.seconds()
        .alias("trip_duration_sec"),

    # 4. avg_speed_mph
    (
        pl.col("trip_distance") /
        (pl.col("trip_duration_sec") / 3600)
    )
    .replace_nan(None)                      # protect against div-by-zero
    .alias("avg_speed_mph"),

    # 5. tip_to_fare_ratio
    (pl.col("tip_amount") / pl.col("fare_amount"))
        .replace_inf(None)
        .replace_nan(None)
        .alias("tip_to_fare_ratio"),

    # 6. fare_per_mile
    (pl.col("fare_amount") / pl.col("trip_distance"))
        .replace_inf(None)
        .replace_nan(None)
        .alias("fare_per_mile"),

    # 7. dist_per_passenger
    (pl.col("trip_distance") / pl.col("passenger_count"))
        .replace_inf(None)
        .replace_nan(None)
        .alias("dist_per_passenger"),
])

dropoff_stats = (
    df.groupby("DOLocationID")
      .agg([
          pl.mean("avg_speed_mph").alias("dropoff_avg_speed"),
          pl.count().alias("dropoff_trip_cnt"),
      ])
      .rename({"DOLocationID": "dropoff_zone_id"})   # avoid name clash later
)

-zone stats back onto every row
ropoff_stats, left_on="DOLocationID", right_on="dropoff_zone_id", how="left")
```

Summary

We have seen how to deal with common workflows in both Pandas and Polars, starting from basic tasks like opening a dataset and inspecting it to performing split-apply-combine pipelines. We have seen how to use Polars to manipulate datasets and perform some basic feature engineering.

> Keypoints
>
> - Dataframes are combinations of series
> - Both Pandas and Polars can be used to manipulate them
> - The expression API in Polars allows to perform advanced operations with a simple DSL.

See also

There is a lot more to Polars than what we covered in this short introduction. For example, queries like the ones we introduced can be performed lazily, i.e. just declared and then run all together, giving the backend a chance to optimise them. This can dramatically improve performance in the case of complex queries. For this and a lot more, we refer you to the official documentation.

Storage & serialisation backends

> Questions
>
> - What syntax is used to make a lesson?
> - How do you structure a lesson effectively for teaching?
> - `questions` are at the top of a lesson and provide a starting point for what you might learn. It is usually a bulleted list.

> Objectives
>
> - Show a complete lesson page with all of the most common structures.
> - …
>
> This is also a holdover from the carpentries-style. It could usually be left off.

The introduction should be a high level overview of what is on the page and why it is interesting.

The lines below (only in the source) will set the default highlighting language for the entire page.

Skip to content

A section.

Discussion

Discuss the following.

- A discussion section
- Another discussion topic

Section

```
print("hello world")
# This uses the default highlighting language
```

```
print("hello world)
```

Exercises: description

Exercise Topic-1: imperative description of exercise

Exercise text here.

Solution

Solution text here

Summary

A Summary of what you learned and why it might be useful. Maybe a hint of what comes next.

See also

- Other relevant links
- Other link

Keypoints

- What the learner should take away
- point 2
- …

This is another holdover from the carpentries style. This perhaps is better done in a “summary” section.

Visualisations and dashboards

Questions

- What syntax is used to make a lesson?
- How do you structure a lesson effectively for teaching?
- `questions` are at the top of a lesson and provide a starting point for what you might learn. It is usually a bulleted list.

Objectives

- Show a complete lesson page with all of the most common structures.
- …

This is also a holdover from the carpentries-style. It could usually be left off.

The introduction should be a high level overview of what is on the page and why it is interesting.

The lines below (only in the source) will set the default highlighting language for the entire page.

Section

A section.

Discussion

Discuss the following.

- A discussion section
- Another discussion topic

Section

```
print("hello world")
# This uses the default highlighting language
```

```
print("hello world)
```

Exercises: description

Skip to content

1: imperative description of exercise

Exercise text here.

Solution

Solution text here

Summary

A Summary of what you learned and why it might be useful. Maybe a hint of what comes next.

See also

- Other relevant links
- Other link

Keypoints

- What the learner should take away
- point 2
- …

This is another holdover from the carpentries style. This perhaps is better done in a “summary” section.

Benchmarking

Questions

- What syntax is used to make a lesson?
- How do you structure a lesson effectively for teaching?
- `questions` are at the top of a lesson and provide a starting point for what you might learn. It is usually a bulleted list.

Objectives

- Show a complete lesson page with all of the most common structures.
- …

This is also a holdover from the carpentries-style. It could usually be left off.

The introduction should be a high level overview of what is on the page and why it is interesting.

The lines below (only in the source) will set the default highlighting language for the entire page.

Section

Skip to content

Discussion

Discuss the following.

- A discussion section
- Another discussion topic

Section

```
print("hello world")
# This uses the default highlighting language
```

```
print("hello world)
```

Exercises: description

Exercise Topic-1: imperative description of exercise

Exercise text here.

Solution

Solution text here

Summary

A Summary of what you learned and why it might be useful. Maybe a hint of what comes next.

See also

- Other relevant links
- Other link

Keypoints

- What the learner should take away
- point 2
- …

This is another holdover from the carpentries style. This perhaps is better done in a “summary” section.

Multithreading

Questions

- What syntax is used to make a lesson?
- How do you structure a lesson effectively for teaching?
- `questions` are at the top of a lesson and provide a starting point for what you might learn. It is usually a bulleted list.

Objectives

- Show a complete lesson page with all of the most common structures.
- …

This is also a holdover from the carpentries-style. It could usually be left off.

The introduction should be a high level overview of what is on the page and why it is interesting.

The lines below (only in the source) will set the default highlighting language for the entire page.

Section

A section.

Discussion

Discuss the following.

- A discussion section
- Another discussion topic

Section

```
print("hello world")
# This uses the default highlighting language
```

```
print("hello world)
```

Exercises: description

Skip to content

1: imperative description of exercise

Exercise text here.

Solution

Solution text here

Summary

A Summary of what you learned and why it might be useful. Maybe a hint of what comes next.

See also

- Other relevant links
- Other link

Keypoints

- What the learner should take away
- point 2
- …

This is another holdover from the carpentries style. This perhaps is better done in a “summary” section.

Dask for Scalable Analytics

Objectives

- Understand how Dask achieves parallelism
- Learn a few common workflows with Dask
- Understand lazy execution

Instructor note

- 40 min teaching/type-along
- 40 min exercises

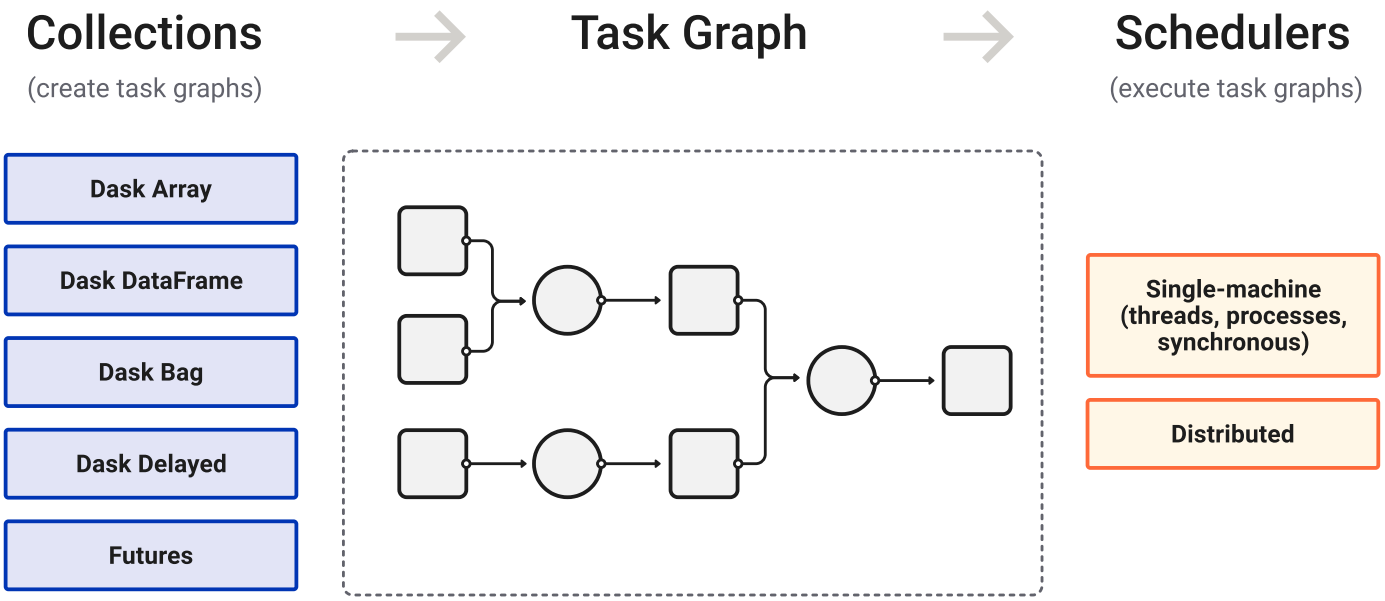
Overview

An increasingly common problem faced by researchers and data scientists today is that datasets are becoming larger and larger and modern data analysis is thus becoming more and more computationally demanding. The first difficulty to deal with is when the volume of data exceeds one's computer's RAM. Modern laptops/desktops have about 10 GB of RAM. Beyond this threshold, some special care is required to carry out data analysis. The next threshold of difficulty is when the data can not even fit on the hard drive, which is about a couple of TB on a modern laptop. In this situation, it is better to use an HPC system or a cloud-based solution, and Dask is a tool that helps us easily extend our familiar data analysis tools to work with big data. In addition, Dask can also speeds up our analysis by using multiple CPU cores which makes our work run faster on laptop, HPC and cloud platforms.

What is Dask?

Dask is composed of two parts:

- Dynamic task scheduling optimized for computation. Similar to other workflow management systems, but optimized for interactive computational workloads.
- "Big Data" collections like parallel arrays, dataframes, and lists that extend common interfaces like NumPy, Pandas, or Python iterators to larger-than-memory or distributed environments. These parallel collections run on top of dynamic task schedulers.



HIGH LEVEL COLLECTIONS ARE USED TO GENERATE TASK GRAPHS WHICH CAN BE EXECUTED BY SCHEDULERS ON A SINGLE MACHINE OR A CLUSTER. FROM THE DASK DOCUMENTATION.

Dask clusters

Dask needs computing resources in order to perform parallel computations. “Dask Clusters” have different names corresponding to different computing environments, for example:

- `LocalCluster` on laptop/desktop/cluster
- `PBSCluster` or `SLURMCluster` on HPC
- `Kubernetes` cluster in the cloud

Each cluster will be allocated with a given number of “workers” associated with CPU and RAM and the Dask scheduling system automatically maps jobs to each worker.

Dask provides four different schedulers:

Type	Multi-node	Description
`threads`	No	A single-machine scheduler backed by a thread pool
`processes`	No	A single-machine scheduler backed by a process pool
`synchronous`	No	A single-threaded scheduler, used for debugging
`distributed`	yes	A distributed scheduler for executing on multiple nodes/machines

Skip to content

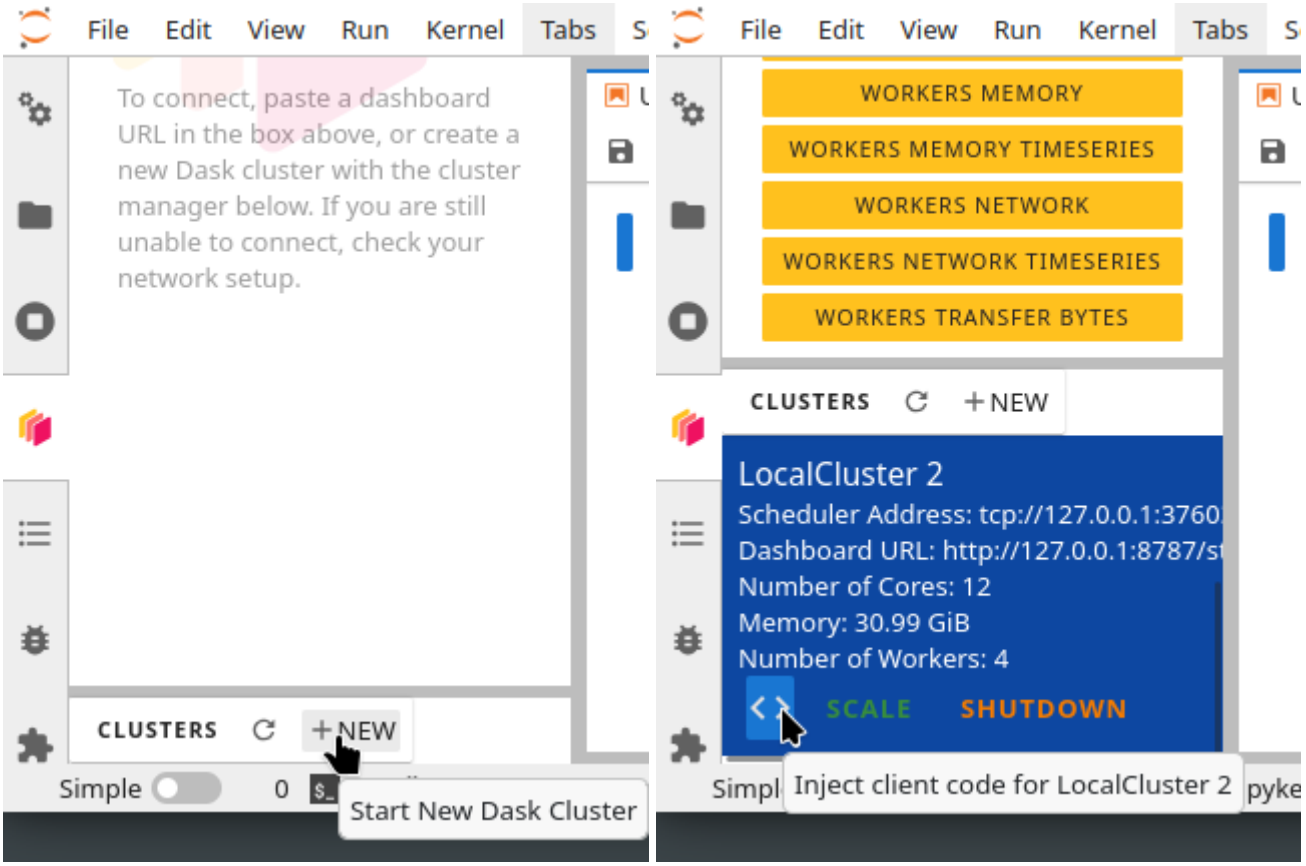
; on using a `LocalCluster`, and it is recommended to use a distributed scheduler `dask.distributed`. It is more sophisticated, offers more features, but requires minimum effort to set up. It can run locally on a laptop and scale up to a cluster.

Alternative 1: Initializing a Dask `LocalCluster` via JupyterLab

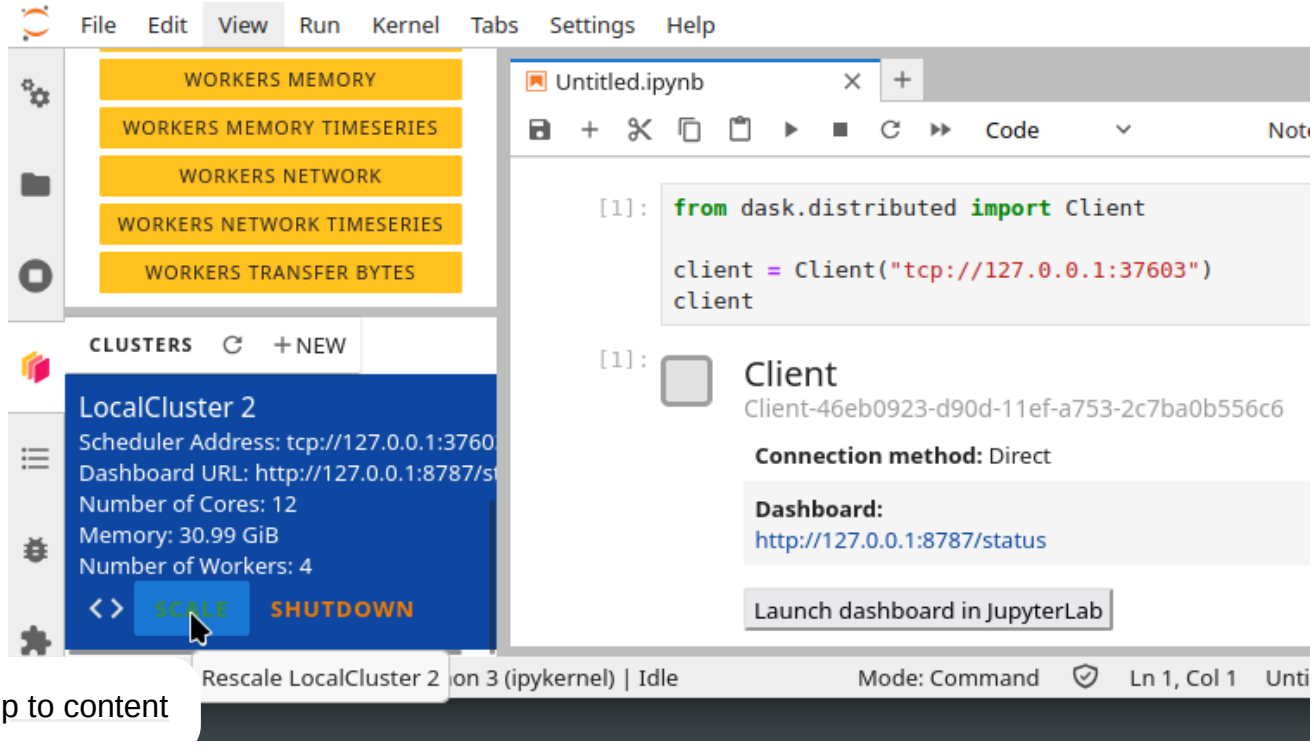
Skip to content

This makes use of the `dask-labextension` which is pre-installed in our conda environment.

1. Start New Dask Cluster from the sidebar and by clicking on `+ NEW` button.
2. Click on the `< >` button to inject the client code into a notebook cell. Execute it.



3. You can scale the cluster for more resources or launch the dashboard.



Skip to content

Alternative 2: Manual `LocalCluster`

We can also start a `LocalCluster` scheduler manually, which can use all available resources or just a subset.

All resources | Specified resources

We can use all the cores and RAM we have on the machine by:

```
from dask.distributed import Client, LocalCluster
# create a local cluster
cluster = LocalCluster()
# connect to the cluster we just created
client = Client(cluster)
client
```

Or you can simply launch a `Client()` call which is shorthand for what is described above.

```
from dask.distributed import Client
client = Client() # same as Client(processes=True)
client
```

Note

When setting up the cluster, one should consider the balance between the number of workers and threads per worker with different workloads by setting the parameter `processes`. By default `processes=True` and this is a good choice for workloads that have the GIL, thus it is better to have more workers and fewer threads per worker. Otherwise, when `processes=False`, in this case all workers run as threads within the same process as the client, and they share memory resources. This works well for large datasets.

Cluster managers also provide useful utilities: for example if a cluster manager supports scaling, you can modify the number of workers manually or automatically based on workload:

```
cluster.scale(10)  # Sets the number of workers to 10
cluster.adapt(minimum=1, maximum=10)  # Allows the cluster to auto scale to 10 when tas
```

Dask distributed scheduler also provides live feedback via its interactive dashboard. A link that redirects to the dashboard will prompt in the terminal where the scheduler is created, and it is also shown when you create a Client and connect the scheduler. By default, when starting a scheduler on your local machine the dashboard will be served at http://localhost:8787/status and can be always queried from commond line by:

```
cluster.dashboard_link
http://127.0.0.1:8787/status
# or
client.dashboard_link
```

When everything finishes, you can shut down the connected scheduler and workers by calling the `shutdown()` method:

```
client.shutdown()
```

Dask collections

Dask provides dynamic parallel task scheduling and three main high-level collections:

- `dask.array`: Parallel NumPy arrays
- `dask.dataframe`: Parallel Pandas DataFrames
- `dask.bag`: Parallel Python Lists

Dask arrays

A Dask array looks and feels a lot like a NumPy array. However, a Dask array uses the so-called “lazy” execution mode, which allows one to build up complex, large calculations symbolically before turning them over the scheduler for execution.

Lazy evaluation

Contrary to normal computation, lazy execution mode is when all the computations needed to generate results are symbolically represented, forming a queue of tasks mapped over data blocks. Nothing is actually computed until the actual numerical values are needed, e.g. plotting, to print results to the screen or write to disk. At that point, data is loaded into memory and computation proceeds in a streaming fashion, block-by-block. The actual computation is controlled by a multi-processing or thread pool, which allows Dask to take full advantage of multiple processors available on the computers.

```
import numpy as np
shape = (1000, 4000)
ones_np = np.ones(shape)
ones_np
ones_np.nbytes / 1e6
```

Skip to content ıe same array using Dask’s array interface.

```
import dask.array as da
shape = (1000, 4000)
ones = da.ones(shape)
ones
```

Although this works, it is not optimized for parallel computation. In order to use all available computing resources, we also specify the `chunks` argument with Dask, which describes how the array is split up into sub-arrays:

```
import dask.array as da
shape = (1000, 4000)
chunk_shape = (1000, 1000)
ones = da.ones(shape, chunks=chunk_shape)
ones
```

> **Note**
>
> In this course, we will use a chunk shape, but other ways to specify `chunks` size can be found here.

Let us further calculate the sum of the dask array:

```
sum_da = ones.sum()
```

So far, only a task graph of the computation is prepared. We can visualize the task graph by calling `visualize()`:

```
dask.visualize(sum_da)
# or
sum_da.visualize()
```

One way to trigger the computation is to call `compute()`:

```
dask.compute(sum_da)
# or
sum_da.compute()
```

You can find additional details and examples here.

Dask dataframe

Dask dataframes split a dataframe into partitions along an index and can be used in situations where one would normally use Pandas, but this fails due to data size or insufficient computational efficiency. Specifically, you can use Dask dataframes to:

- manipulate large datasets, even when these don't fit in memory
- accelerate long computations by using many cores
- perform distributed computing on large datasets with standard Pandas operations like groupby, join, and time series computations.

Let us revisit the dataset containing the Titanic passenger list, and now transform it to a Dask dataframe:

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

url = "https://raw.githubusercontent.com/pandas-dev/pandas/master/doc/data/titanic.csv"

df = pd.read_csv(url, index_col="Name")
# read a Dask Dataframe from a Pandas Dataframe
ddf = dd.from_pandas(df, npartitions=10)
```

Alternatively you can directly read into a Dask dataframe, whilst also modifying how the dataframe is partitioned in terms of `blocksize`:

```
# blocksize=None which means a single chunk is used
df = dd.read_csv(url,blocksize=None).set_index('Name')
ddf= df.repartition(npartitions=10)

# blocksize="4MB" or blocksize=4e6
ddf = dd.read_csv(url,blocksize="4MB").set_index('Name')
ddf.npartitions

# blocksize="default" means the chunk is computed based on
# available memory and cores with a maximum of 64MB
ddf = dd.read_csv(url,blocksize="default").set_index('Name')
ddf.npartitions
```

Dask dataframes do not support the entire interface of Pandas dataframes, but the most commonly used methods are available. For a full listing refer to the dask dataframe API.

ple perform the group-by operation we did earlier, but this time in parallel:

```
# add a column
ddf["Child"] = ddf["Age"] < 12
ddf.groupby(["Sex", "Child"])["Survived"].mean().compute()
```

However, for a small dataframe like this the overhead of parallelisation will far outweigh the benefit.

You can find additional details and examples here https://examples.dask.org/dataframe.html.

Dask bag

A Dask bag enables processing data that can be represented as a sequence of arbitrary inputs (“messy data”), like in a Python list. Dask Bags are often used to for preprocessing log files, JSON records, or other user defined Python objects.

We will content ourselves with implementing a dask version of the word-count problem, specifically the step where we count words in a text.

Demo: Dask version of word-count

If you have not already cloned or downloaded `word-count-hpda` repository, get it from here. Then, navigate to the `word-count-hpda` directory. The serial version (wrapped in multiple functions in the `source/wordcount.py` code) looks like this:

```
filename = './data/pg10.txt'
DELIMITERS = ". , ; : ? $ @ ^ < > # % ` ! * - = ( ) [ ] { } / \" '".split()

with open(filename, "r") as input_fd:
    lines = input_fd.read().splitlines()

counts = {}
for line in lines:
    for purge in DELIMITERS:
        line = line.replace(purge, " ")
    words = line.split()
    for word in words:
        word = word.lower().strip()
        if word in counts:
            counts[word] += 1
        else:
            counts[word] = 1

sorted_counts = sorted(
    list(counts.items()),
    key=lambda key_value: key_value[1],
    reverse=True
)

sorted_counts[:10]
```

A very compact `dask.bag` version of this code is as follows:

```
import dask.bag as db
filename = './data/pg10.txt'
DELIMITERS = ". , ; : ? $ @ ^ < > # % ` ! * - = ( ) [ ] { } / \" '".split()

text = db.read_text(filename, blocksize='1MiB')
sorted_counts = (
    text
    .filter(lambda word: word not in DELIMITERS)
    .str.lower()
    .str.strip()
    .str.split()
    .flatten()
    .frequencies().topk(10,key=1)
    .compute()
)
```

Skip to content

The last two steps of the pipeline could also have been done with a dataframe:

```
:emphasize-lines: 9-10

filtered = (
    text
    .filter(lambda word: word not in DELIMITERS)
    .str.lower()
    .str.strip()
    .str.split()
    .flatten()
)
ddf = filtered.to_dataframe(columns=['words'])
ddf['words'].value_counts().compute()[:10]
```

When to use Dask

There is no benefit from using Dask on small datasets. But imagine we were analysing a very large text file (all tweets in a year? a genome?). Dask provides both parallelisation and the ability to utilize RAM on multiple machines.

Exercise set 1

Choose an exercise with the data structure that you are most interested in: 1.1. Using dask.array, 1.2. Using dask.dataframe or 1.3. Using dask.bag.

1.1. Using dask.array

Chunk size

The following example calculate the mean value of a random generated array. Run the example and see the performance improvement by using dask.

NumPy | Dask

```
import numpy as np

%%time
x = np.random.random((20000, 20000))
y = x.mean(axis=0)
```

But what happens if we use different chunk sizes? Try out with different chunk sizes:

- What happens if the dask chunks=(20000,20000)
- What happens if the dask chunks=(250,250)

Choice of chunk size

The choice is problem dependent, but here are a few things to consider:

Each chunk of data should be small enough so that it fits comforably in each worker's available memory. Chunk sizes between 10MB-1GB are common, depending on the availability of RAM. Dask will likely manipulate as many chunks in parallel on one machine as you have cores on that machine. So if you have a machine with 10 cores and you choose chunks in the 1GB range, Dask is likely to use at least 10 GB of memory. Additionally, there should be enough chunks available so that each worker always has something to work on.

On the otherhand, you also want to avoid chunk sizes that are too small as we see in the exercise. Every task comes with some overhead which is somewhere between 200us and 1ms. Very large graphs with millions of tasks will lead to overhead being in the range from minutes to hours which is not recommended.

1.2. Using dask.dataframe

Benchmarking DataFrame.apply()

Recall the :ref:`word count <word-count-problem>` project that we encountered earlier and the :func:`scipy.optimize.curve_fit` function. The :download:`results.csv <data/results.csv>` file contains word counts of the 10 most frequent words in different texts, and we want to fit a power law to the individual distributions in each row.

Here are our fitting functions:

```
from scipy.optimize import curve_fit

def powerlaw(x, A, s):
    return A * np.power(x, s)

def fit_powerlaw(row):
    X = np.arange(row.shape[0]) + 1.0
    params, cov = curve_fit(f=powerlaw, xdata=X, ydata=row, p0=[100, -1], bounds=(-np.inf, np.inf))
    return params[1]
```

Compare the performance of :meth:`dask.dataframe.DataFrame.apply` with :meth:`pandas.DataFrame.apply` for the this example. You will probably see a slowdown due to the parallelisation overhead. But what if you add a `time.sleep(0.01)` inside :meth:`fit_powerlaw` to emulate a time-consuming calculation?

Hints

- You will need to call :meth:`apply` on the dataframe starting from column 1: `dataframe.iloc[:,1:].apply()`
- Remember that both Pandas and Dask have the :meth:`read_csv` function.
- Try repartitioning the dataframe into 4 partitions with `ddf4=ddf.repartition(npartitions=4)`.
- You will probably get a warning in your Dask version that `You did not provide metadata`. To remove the warning, add the `meta=(None, "float64")` flag to :meth:`apply`. For the current data, this does not affect the performance.

More hints with Pandas code

You need to reimplement the highlighted part which creates the dataframe and applies the :func:`fit_powerlaw` function.

```
import numpy as np
import pandas as pd
from scipy.optimize import curve_fit
import time

def powerlaw(x, A, s):
    return A * np.power(x, s)

def fit_powerlaw(row):
    X = np.arange(row.shape[0]) + 1.0
    params, cov = curve_fit(f=powerlaw, xdata=X, ydata=row, p0=[100, -1], bounds=(-np.inf, np.inf))
    time.sleep(0.01)
    return params[1]


df = pd.read_csv("https://raw.githubusercontent.com/ENCCS/hpda-python/main/content/data/results.csv")
%timeit results = df.iloc[:,1:].apply(fit_powerlaw, axis=1)
```

Solution

```
import numpy as np
import dask.dataframe as dd
from scipy.optimize import curve_fit
import time

def powerlaw(x, A, s):
    return A * np.power(x, s)

def fit_powerlaw(row):
    X = np.arange(row.shape[0]) + 1.0
    params, cov = curve_fit(f=powerlaw, xdata=X, ydata=row, p0=[100, -1], bounds=(-np.inf, np.inf))
    time.sleep(0.01)
    return params[1]

ddf = dd.read_csv("https://raw.githubusercontent.com/ENCCS/hpda-python/main/content/data/results.csv"
ddf4=ddf.repartition(npartitions=4)

# Note the optional argument ``meta`` which is recommended for dask dataframes.
# It should contain an empty ``pandas.DataFrame`` or ``pandas.Series``
# that matches the dtypes and column names of the output,
# or a dict of ``{name: dtype}`` or iterable of ``(name, dtype)``.

results = ddf4.iloc[:,1:].apply(fit_powerlaw, axis=1, meta=(None, "float64"))
%timeit results.compute()
results.visualize()
```

Skip to content

1.3. Using dask.bag

> Break down the dask.bag computational pipeline
>
> Revisit the :ref:`word count problem <word-count-problem>` and the implementation with a `dask.bag` that we saw above.
>
> - To get a feeling for the computational pipeline, break down the computation into separate steps and investigate intermediate results using :meth:`.compute`.
> - Benchmark the serial and `dask.bag` versions. Do you see any speedup? What if you have a larger textfile? You can for example concatenate all texts into a single file: `cat data/*.txt > data/all.txt`.

Low level interface: delayed

Sometimes problems don't fit into one of the collections like `dask.array` or `dask.dataframe`, they are not as simple as just a big array or dataframe. In these cases, `dask.delayed` may be the right choice. If the problem is paralellisable, we can use `dask.delayed` which allows users to make function calls lazy and thus can be put into a task graph with dependencies.

Consider the following example. The functions are very simple, and they *sleep* for a prescribed time to simulate real work:

```
import time

def inc(x):
    time.sleep(0.5)
    return x + 1

def dec(x):
    time.sleep(0.3)
    return x - 1

def add(x, y):
    time.sleep(0.1)
    return x + y
```

Let us run the example first, one after the other in sequence:

```
%%timeit
x = inc(1)
y = dec(2)
z = add(x, y)
# 902 ms ± 367 µs per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Skip to content

Note that the first two functions `inc` and `dec` don't depend on each other, we could have called them in parallel. We can call `dask.delayed` on these functions to make them lazy and tasks into a graph which we will run later on parallel hardware.

```
import dask
inc_delay = dask.delayed(inc)
dec_delay = dask.delayed(dec)
add_delay = dask.delayed(add)
```

```
%%timeit
x = inc_delay(1)
y = dec_delay(2)
z = add_delay(x, y)
# 59.6 µs ± 356 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
```

```
%%timeit
x = inc_delay(1)
y = dec_delay(2)
z = add_delay(x, y)
z.compute()
# 603 ms ± 181 µs per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Default scheduler for dask collections

`dask.array` and `dask.dataframe` use the `threads` scheduler

`dask.bag` uses the `processes` scheduler

In case to change the default scheduler, using `dask.config.set` is recommended:

```
# To set globally
dask.config.set(scheduler='processes')
x.compute()

# To set it as a context manager
with dask.config.set(scheduler='threads'):
    x.compute()
```

Comparison to Spark

Dask has much in common with the Apache Spark. Here are some differences between the two frameworks:

- Dask is smaller and more lightweight but is used together with other packages in the Python ecosystem. Spark is an all-in-one project with its own ecosystem.
- Spark is written in Scala, with some support for Python and R, while Dask is in Python.
- Spark is more focused on business intelligence (SQL, lightweight machine learning) while Dask is more general and is used more in scientific applications.
- Both Dask and Spark can scale from one to thousands of nodes.
- Dask supports the NumPy model for multidimensional arrays which Spark doesn't.
- Spark generally expects users to compose computations out of high-level primitives (map, reduce, groupby, join, etc.), while Dask allows to specify arbitrary task graphs for more complex and custom systems.

Exercise set 2

Dask delay

We extend the previous example a little bit more by applying the function on a data array using for loop and adding an *if* condition:

```
import time
import dask

def inc(x):
    time.sleep(0.5)
    return x + 1

def dec(x):
    time.sleep(0.3)
    return x - 1

def add(x, y):
    time.sleep(0.1)
    return x + y


data = [1, 2, 3, 4, 5]
output = []
for x in data:
    if x % 2:
        a = inc(x)
        b = dec(x)
        c = add(a, b)
    else:
        c = 10
    output.append(c)

total = sum(output)
```

Please add `dask.delayed` to parallelize the program as much as possible and check graph visualizations.

Solution

```
import time
import dask

def inc(x):
    time.sleep(0.5)
    return x + 1

def dec(x):
    time.sleep(0.3)
    return x - 1

def add(x, y):
    time.sleep(0.1)
    return x + y


data = [1, 2, 3, 4, 5]
output = []
for x in data:
    if x % 2:
        a = dask.delayed(inc)(x)
        b = dask.delayed(dec)(x)
        c = dask.delayed(add)(a, b)
    else:
        c = dask.delayed(10)
    output.append(c)

total = dask.delayed(sum)(output)
```

Climate simulation data using Xarray and Dask

This exercise is working with NetCDF files using Xarray. The files contain monthly global 2m air temperature for 10 years. Xarray is chosen due to its ability to seamlessly integrate with Dask to support parallel computations on datasets.

We will first read data with Dask and Xarray. See https://xarray.pydata.org/en/stable/dask.html#reading-and-writing-data for more details.

Note that the NetCDF files are here https://github.com/ENCCS/hpda-python/tree/main/content/data , you need to `git clone` the repository or download the files to your laptop first. Then depending on where you put the files, you may need to adapt the path to the data folder in the Python code.

```
import dask
import xarray as xr
import matplotlib.pyplot as plt
%matplotlib inline
ds=xr.open_mfdataset('./data/tas*.nc', parallel=True,use_cftime=True)
```

:func:`xarray.open_mfdataset` is for reading multiple files and will chunk each file into a single Dask array by default. One could supply the chunks keyword argument to control the size of the resulting Dask arrays. Passing the keyword argument `parallel=True` to :func:`xarray.open_mfdataset` will speed up the reading of large multi-file datasets by executing those read tasks in parallel using `dask.delayed`.

Explore the following operations line-by-line:

```
ds
ds.tas
#dsnew = ds.chunk({"time": 1,"lat": 80,"lon":80})   # you can further rechunk the data
#dask.visualize(ds.tas) # do not visualize, the graph is too big
ds['tas'] = ds['tas'] - 273.15     # convert from Kelvin to degree Celsius
mean_tas=ds.tas.mean("time")  # lazy compuation
mean_tas.plot(cmap=plt.cm.RdBu_r,vmin=-50,vmax=50) # plotting triggers computation
tas_ann=ds.tas.groupby('time.year').mean() # lazy compuation
tas_sto=tas_ann.sel(lon=18.07, lat=59.33,method='nearest')  # slicing is lazy as well
plt.plot(tas_sto.year,tas_sto)  # plotting trigers computation
```

Keypoints

- Dask uses lazy execution
- Dask can parallelize and perform out-of-memory computation. That is, handle data that would not fit in the memory if loaded at once.
- Only use Dask for processing very large amount of data

Skip to content

Dask (II)

Testing different schedulers

We will test different schedulers and compare the performance on a simple task calculating the mean of a random generated array.

Here is the code using NumPy:

```
import dask
import time
import numpy as np

def calc_mean(i, n):
    data = np.mean(np.random.normal(size = n))
    return(data)
```

Here we run the same code using different schedulers from Dask:

Serial | Threads | Processes | Distributed

```
n = 100000
%%timeit
rs=[calc_mean(i, n) for i in range(100)]
#352 ms ± 925 µs per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Skip to content

Testing different schedulers

Comparing profiling from mt_1, mt_2 and mt_4: Using `threads` scheduler is limited by the GIL on pure Python code. In our case, although it is not a pure Python function, it is still limited by GIL, therefore no multi-core speedup

Comparing profiling from mt_1, mp_1 and dis_1: Except for `threads`, the other two schedulers copy data between processes and this can introduce performance penalties, particularly when the data being transferred between processes is large.

Comparing profiling from serial, mt_1, mp_1 and dis_1: Creating and destroying threads and processes have overheads, `processes` have even more overhead than `threads`

Comparing profiling from mp_1, mp_2 and mp_4: Running multiple processes is only effective when there is enough computational work to do i.e. CPU-bound tasks. In this very example, most of the time is actually spent on transferring the data rather than computing the mean

Comparing profiling from `processes` and `distributed`: Using `distributed` scheduler has advantages over `processes`, this is related to better handling of data copying, i.e. `processes` scheduler copies data for every task, while `distributed` scheduler copies data for each worker.

SVD with large skinny matrix using `distributed` scheduler

We can use dask to compute SVD of a large matrix which does not fit into the memory of a normal laptop/desktop. While it is computing, you should switch to the Dask dashboard and watch column “Workers” and “Graph”, so you must run this using `distributed` scheduler

```
import dask
import dask.array as da
X = da.random.random((2000000, 100), chunks=(10000, 100))
X
u, s, v = da.linalg.svd(X)
dask.visualize(u, s, v)
s.compute()
```

SVD is only supported for arrays with chunking in one dimension, which requires that the matrix is either *tall-and-skinny* or *short-and-fat*. If chunking in both dimensions is needed, one should use approximate algorithm.

```
import dask
import dask.array as da
X = da.random.random((10000, 10000), chunks=(2000, 2000))
u, s, v = da.linalg.svd_compressed(X, k=5)
dask.visualize(u, s, v)
s.compute()
```

Memory management

You may observe that there are different memory categories showing on the dashboard:

- process: Overall memory used by the worker process, as measured by the OS
- managed: Size of data that Dask holds in RAM, but most probably inaccurate, excluding spilled data.
- unmanaged: Memory that Dask is not directly aware of, this can be e.g. Python modules, temporary arrays, memory leasks, memory not yet free()'d by the Python memory manager to the OS
- unmanaged recent: Unmanaged memory that has appeared within the last 30 seconds whch is not included in the "unmanaged" memory measure
- spilled: Memory spilled to disk

The sum of managed + unmanaged + unmanaged recent is equal by definition to the process memory.

When the managed memory exceeds 60% of the memory limit (target threshold), the worker will begin to dump the least recently used data to disk. Above 70% of the target memory usage based on process memory measurment (spill threshold), the worker will start dumping unused data to disk.

At 80% process memory load, currently executing tasks continue to run, but no additional tasks in the worker's queue will be started.

At 95% process memory load (terminate threshold), all workers will be terminated. Tasks will be cancelled as well and data on the worker will be lost and need to be recomputed.

Quick Reference

Instructor's guide

Why we teach this lesson

Intended learning outcomes

Timing

Preparing exercises

e.g. what to do the day before to set up common repositories.

Other practical aspects

Interesting questions you might get

Typical pitfalls

What to expect from this course

> Discussion
>
> How large are the datasets you are working with?

Both for classical machine/deep learning and (generative) AI, the amount of data needed to train ever-growing models is becoming bigger and bigger. Moreover, great strides in both hardware and software development for high performance computing (HPC) applications allow for large scale computations that were not possible before. This course focuses on high performance data analytics (HPDA). The data can come from simulations or experiments (or just generally available datasets), and the goal is to pre-process, analyse and visualise it. The lesson introduces some of the modern Python stack for data analytics, dealing with packages such as Pandas, Polars, multithreading and Dask, as well as Streamlit for large-scale data visualisations.

Learning outcomes

This lesson provides a broad overview of methods to work with large datasets using tools and libraries from the Python ecosystem. Since this field is fairly extensive, we will try to expose just enough details on each topic for you to get a good idea of the picture and an understanding of what combination of tools and libraries will work well for your particular use case.

Specifically, this lesson covers:

- Tools for efficiently storing data and writing/reading it to/from disk
- Interfacing with databases and object storage solutions
- Main libraries to work with arrays and tabular data
- Performance monitoring and benchmarking
- Workload parallelisation: threads and Dask

See also

Credit

Don’t forget to check out additional course materials from the Data carpentry, such as:

- Data Analysis and Visualization in Python for Ecologists
- 10 minutes to pandas
- Modern Pandas (blog series by Tom Augspurger)

Moreover, the Polars documentation and Awesome data science with Python are valuable resources, as well as PythonSpeed.

License

CC BY-SA for media and pedagogical material

Skip to content

MIT for source code and code snippets