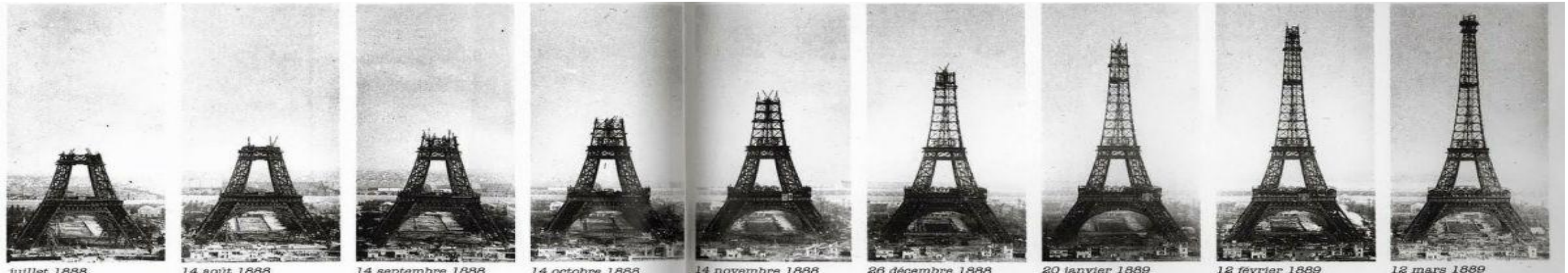
Nov. 3, 2025

Building Progressive Visual Analytics Systems with ProgressiVis

Jean-Daniel Fekete jean-daniel.fekete@inria.fr
Christian Poli christian.poli@inria.fr



Exploring Big Data Today…

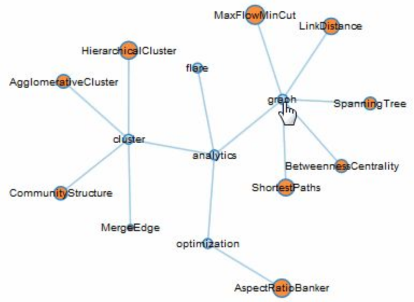
Non Progressive Loading and Visualization

This notebook shows the simplest code to download and visualize all the New York Yellow Taxi trips from January 2015. The trip data are stored in multiple CSV files, containing geolocated taxi trips. We download and visualize the pickup locations (where people have been picked up by the taxis).

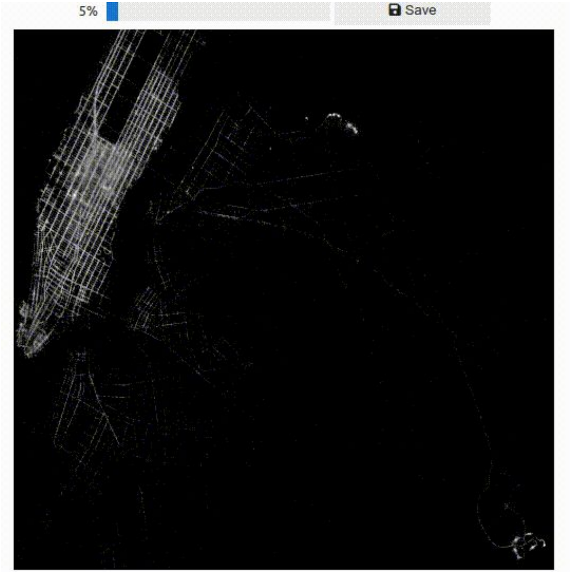
```python
import pandas as pd
import matplotlib.pyplot as plt

# Some constants we'll need: the data file to download and final image size
LARGE_TAXI_FILE = "https://www.aviz.fr/nyc-taxi/yellow_tripdata_2015-01.csv.bz2"
RESOLUTION=512

%time df = pd.read_csv(LARGE_TAXI_FILE, index_col=False, usecols=['pickup_longitude', 'pickup_latitude'])
plt.hist2d(df.pickup_longitude, df.pickup_latitude, bins=(RESOLUTION, RESOLUTION), norm="symlog", cmap=plt.cm.Greys_r)
plt.colorbar()
plt.show()
```

Examples

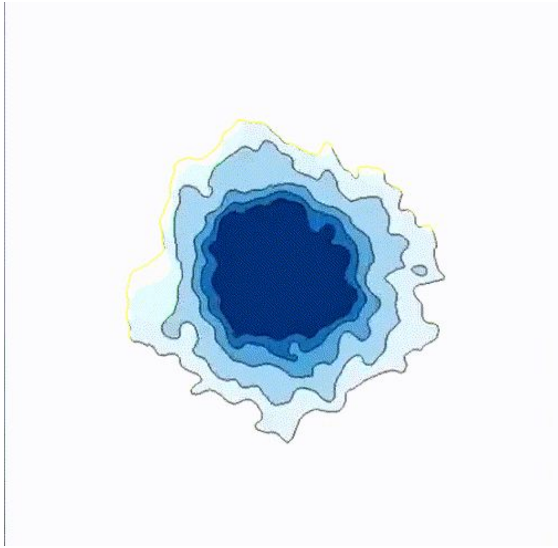
- Graph Layout algorithms are used in a progressive way for years
- New applications are coming for
  - Exploring data instantaneously
  - Looking at high-dimensional data using multidimensional projection
- Either:
  - Looking at data while it is downloaded
  - Looking at a data analysis while it is computed
  - Looking at an ML algorithms while it learns or when applied to data



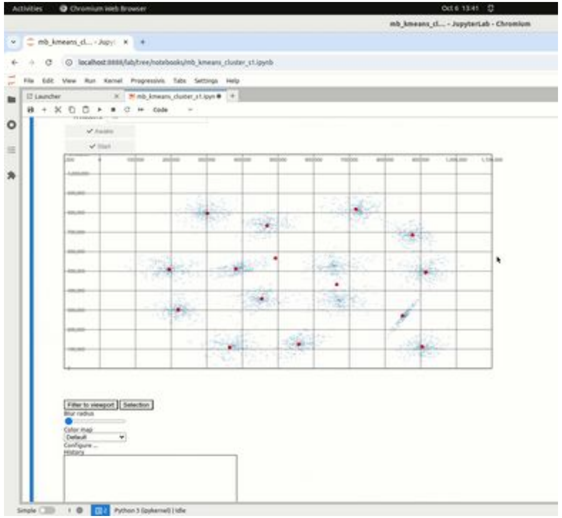
Progressive Visualization and Visual Analytics



Download and Visualization of 12.7B NYC Taxi Trips



t-SNE and Visualization of 4345 genes of 61k points [Pezzoti et al. 16]



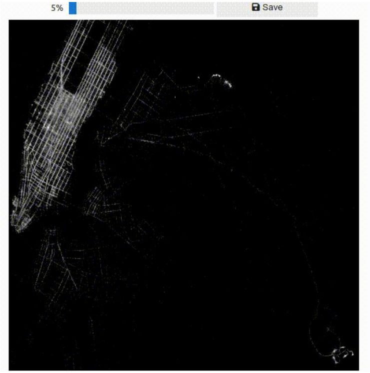
k-Means and Visualization of 105K points

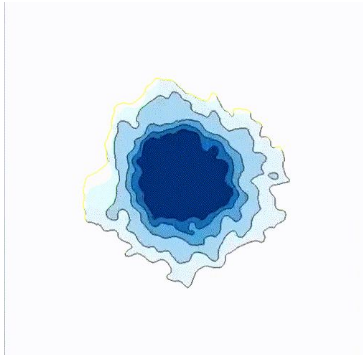
Progressive Data Analysis and Visualization: Why?

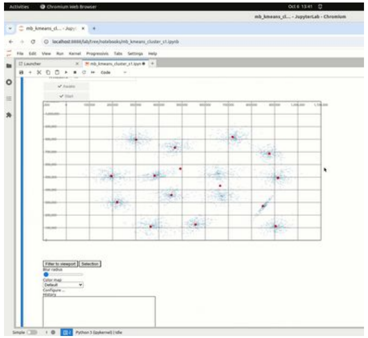
- Datasets continue to grow, taking time to load
- Algorithms that deal with them need more processing time
- But exploratory visualization needs a latency under ~5 seconds
- Faster machines are not solving the problem
  - Data grows faster
  - Algorithms still need more processing time
  - Networks have limited bandwidths
- Progressive Data Analysis and Visualization is a possible solution
  - But it comes with a few caveats

Benefits of Progressive Data Analysis and Visualization

- **Scalability for visualization** in terms of data size and download time
- **Scalability for interactive analysis** including machine learning
- **Instant data**, no need to wait for data and visualization to arrive
- **Greener computing**, processing only the required data to get a result
- **Algorithmic transparency**, monitoring algorithms while they process data





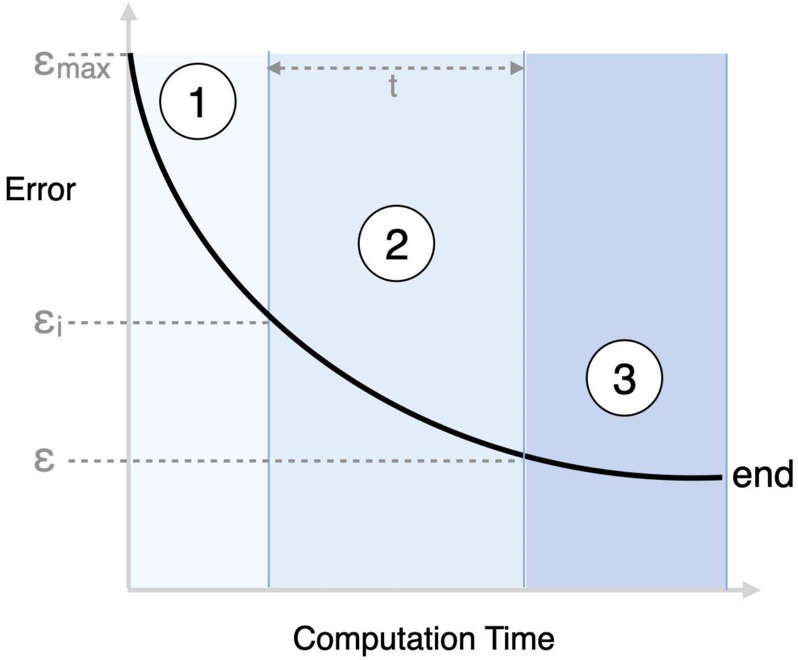


ProgressiVis

- A Toolkit for programming Progressive Data Analysis and Visualizations
- Instead of performing calculations in an unpredictable time
- ProgressiVis splits calculations in pieces
  - So each piece can provide an approximate result, every few seconds or so,
  - Improving with time
  - Until the whole calculation is finished
  - Or, at some point, the approximation is **good enough** to make a decision and stop
    - For that, calculations provide information about their **quality**
- All the calculations in ProgressiVis are done progressively!

Progressive Data Analysis **Quality**?

- PDA allows monitoring a long computation
  - The visualization is evolving over time
- When is the result is good enough?
- PDA computations have three phases:
  - **Chaotic**, no decision can be made, uncertainty high, quality low
  - **Converging**, some decisions can be made, uncertainty and quality medium
  - **Converged**, decisions can be made accurately, uncertainty is low, quality is high, waiting more is a waste of resources
- We use the word **quality** for a value that increases when the results are better
- The real meaning depends on the context.



Goal of this Tutorial

- Introduction to Progressive Data Analysis and Visualization
- Onboarding for Researchers who want to contribute to PDA-V
    - ProgressiVis is an **enabler** to facilitate research on PDA
    - Your contributions are welcome
- Understanding how to start contributing to the progressive roadmap without having to rebuild everything from scratch
    - Data I/O Management
    - Data Structures
    - Algorithms including ML, AI, Layout
    - Visualizations
    - HCI including Interaction and UI

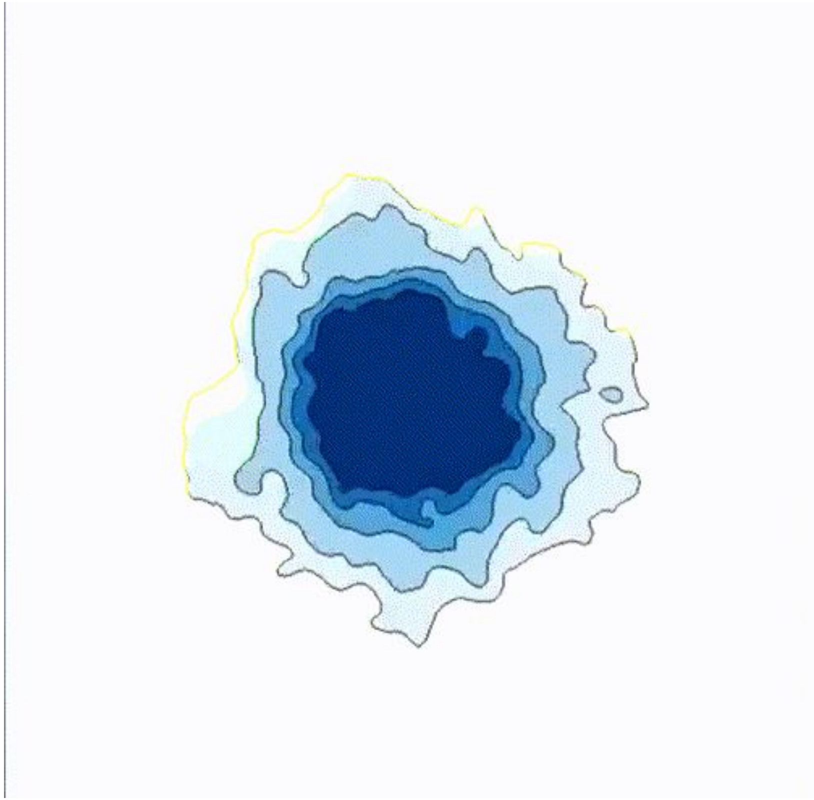
Not an introduction to a new full-fledged Python data analysis environment (yet)

Progressive Data Analysis: What is It?

- A PDA guarantees that its **latency** remains under human cognitive limits (~10 s)
- When a progressive *function* is called
  - instead of returning **one accurate** result in **unbounded** time
  - It will return **a series** of **approximate** results, improving, each within a **given latency**, along with an information on the **progression** and the **quality** of the result.
- Meanwhile, the system remains reactive and users can
  - Stop the computation if the approximation allows them to make a decision
  - Change some of the parameters of the computation to **steer** it
    The PDA system should strive to remain **as stable as possible** to facilitate the monitoring of progress
- PDA trades **time** for **quality** with **control**
- Ideally, progressive "functions" can be composed to form a progressive pipeline
- **But PDA introduces new issues:**
  - Quality assessment and visualization
  - Stability of iterative results for visualization

Multidimensional Projection

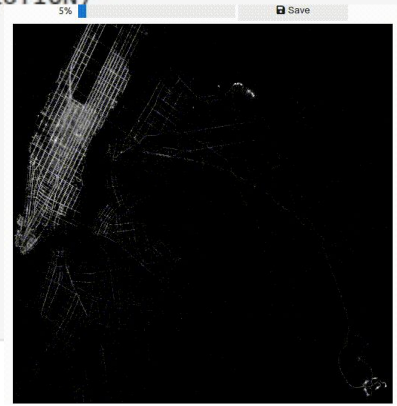
- Visualization of high-dimensional data using the t-SNE projection
- Anything can be transformed into high-dimensional data nowadays
  - Text
  - Image
  - Video
- Exploring a large and complex dataset with t-SNE is always insightful
- Loads data progressively
- Computes k-NN progressively
- Computes t-SNE progressively
- Visualizes the results progressively



Progressive Loading and Visualization

This notebook shows the simplest code to download and visualize all the New York Yellow Taxi trips from January 2015. The trip data is stored in multiple CSV files, containing geolocated taxi trips. We visualize progressively the pickup locations (where people have been picked up by the taxis).

```
from progressivis import CSVLoader, Histogram2D, Quantiles, Heatmap

# Some constants we'll need: the data file to download and final image size
LARGE_TAXI_FILE = "https://www.aviz.fr/nyc-taxi/yellow_tripdata_2015-01.csv.bz2"
RESOLUTION=512

csv = CSVLoader(LARGE_TAXI_FILE, usecols=['pickup_longitude', 'pickup_latitude'])
quantiles = Quantiles()
histogram2d = Histogram2D('pickup_longitude', 'pickup_latitude', xbins=RESOLUTION, ybins=RESOLUTION)
heatmap = Heatmap()

quantiles.input.table = csv.output.result
histogram2d.input.table = csv.output.result
histogram2d.input.min = quantiles.output.result[0.03]  # 0.03 quantile
histogram2d.input.max = quantiles.output.result[0.97]  # 0.97 quantile
heatmap.input.array = histogram2d.output.result
heatmap.display_notebook()
csv.scheduler.task_start()
```

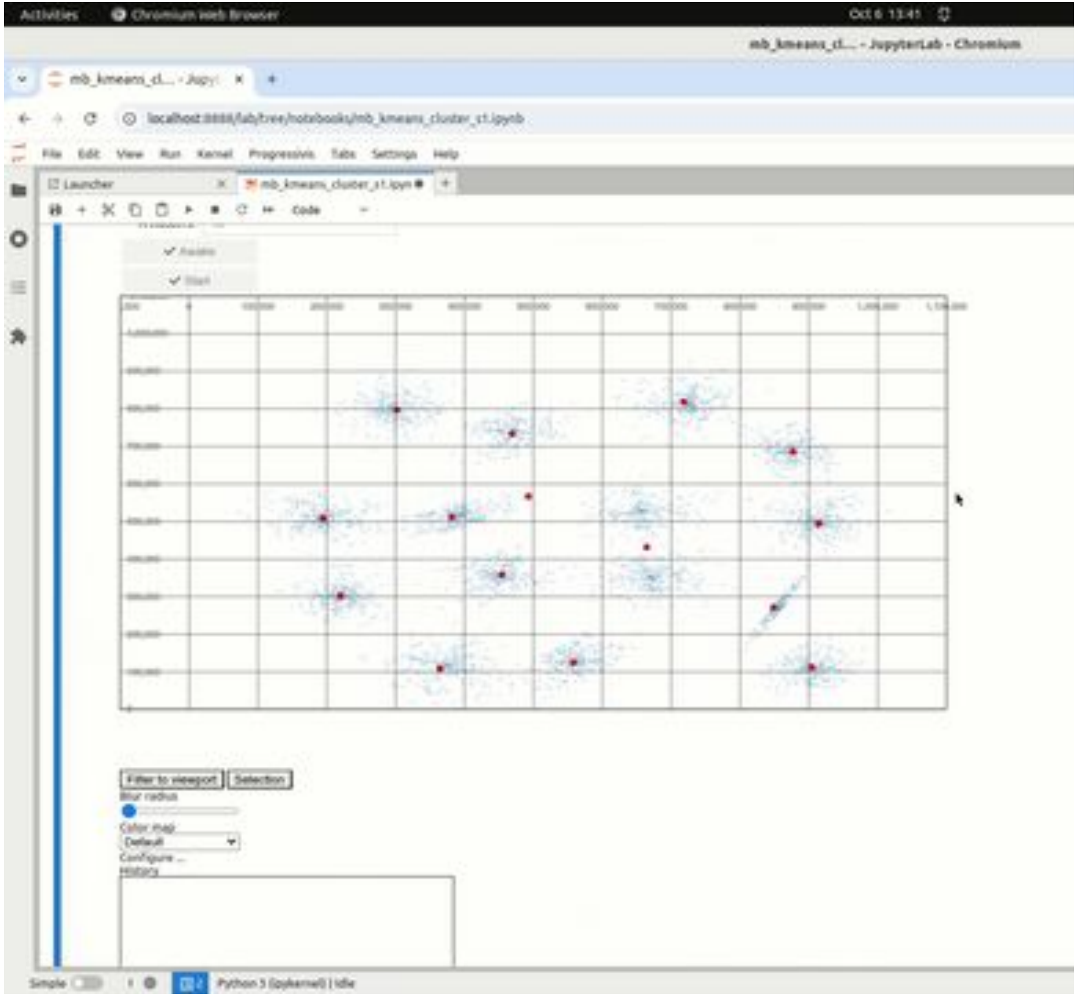


Progressive k-means

Non-deterministic

Can be trapped in local minima

See it unfold and interact to test



Outline

- Installation
- What is Progressive Data Analysis?
- A Simple Program
- Variations of the Simple Program
- Summary of components and organization
- Interaction
- Progressive Notebooks
- Visualizations in the Notebook
- Internals of ProgressiVis
- Creating a new Module
- Creating a new Visualization
- Creating a new Loader
- Wrapping Up

Installation

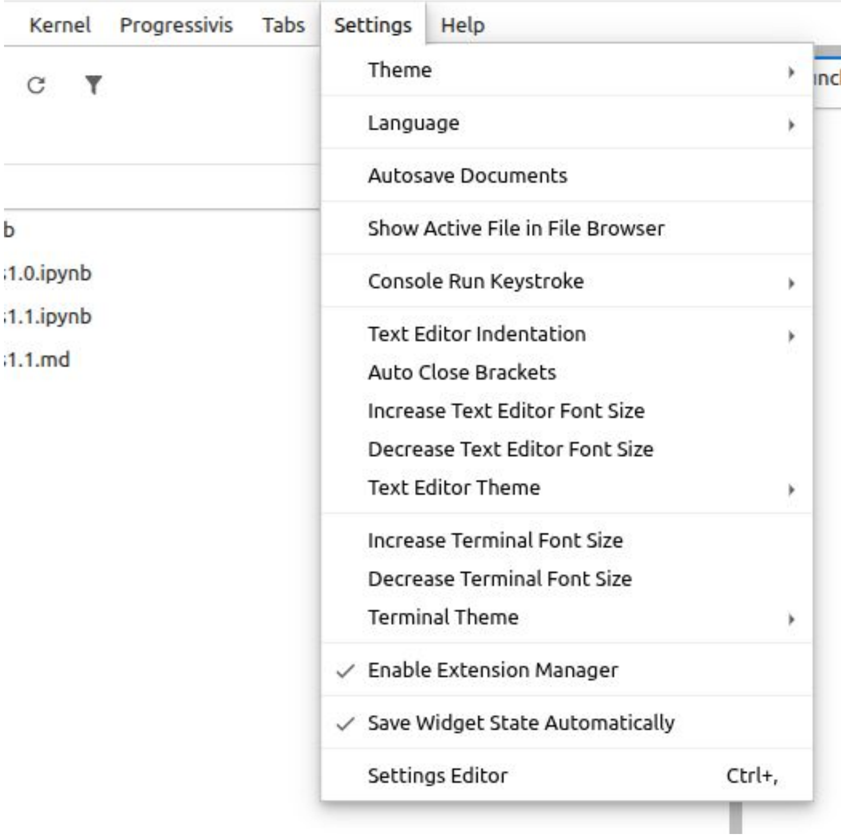
- Connect to our pre-installed MyDocker system
  - URL

See https://progressivis.readthedocs.io

- Distribution: `pip install progressivis`
  - `pip install 'progressivis[jupyter]'`
- For this tutorial, install the tutorial files:
  - `pip install jupytext matplotlib anywidget`
  - `git clone https://github.com/progressivis/progressivis-tutorial.git`
  - `cd progressivis-tutorial`
  - `sh sync.sh`

Configuration of the Jupyter Env

- Disable "Autosave Documents"
- Enable "Save Widget State Automatically"
- Start the console
- Check the tutorial examples

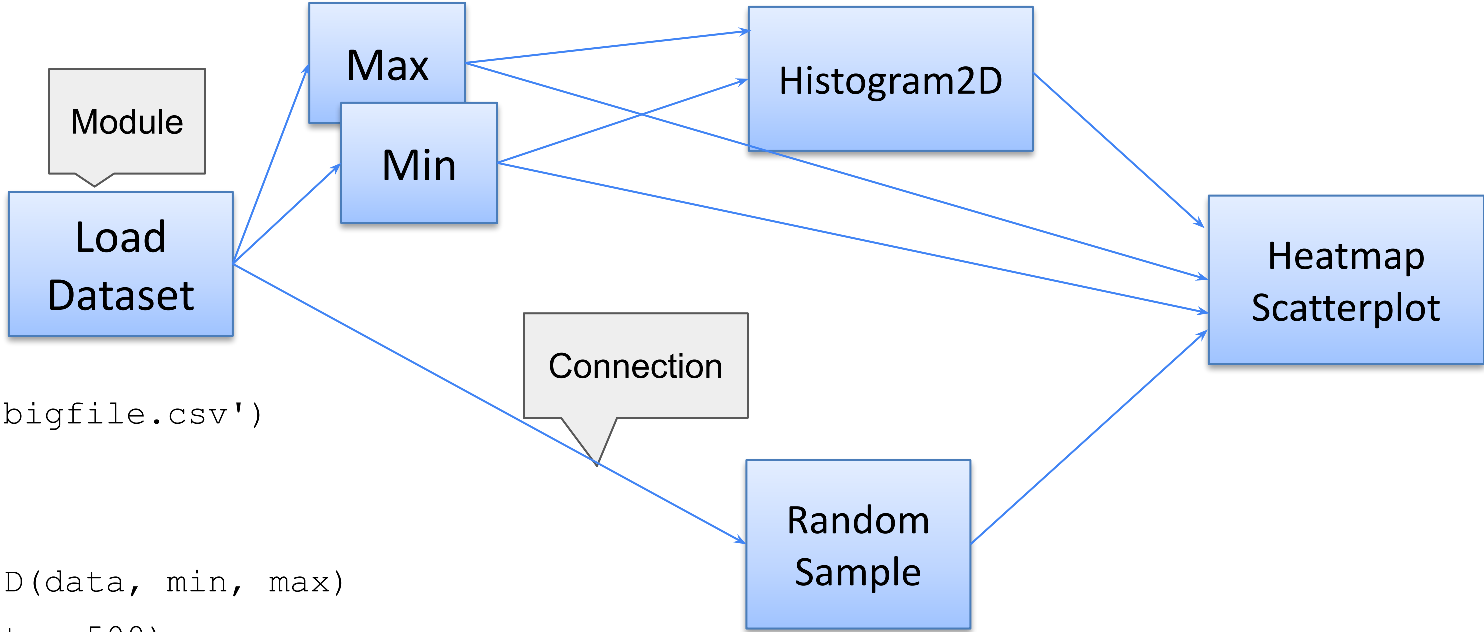


ProgressiVis: New Execution Semantics

Jean-Daniel Fekete, Christian Poli, and Romain Primet

```
data = CSVLoader('bigfile.csv')
min = Min(data)
max = Max(data)
histo = Histogram2D(data, min, max)
sample = Sample(data,  500)
plot = Scatterplot(histo, sample, min, max)
show(plot)
```

Transform the Program into a **Dataflow**



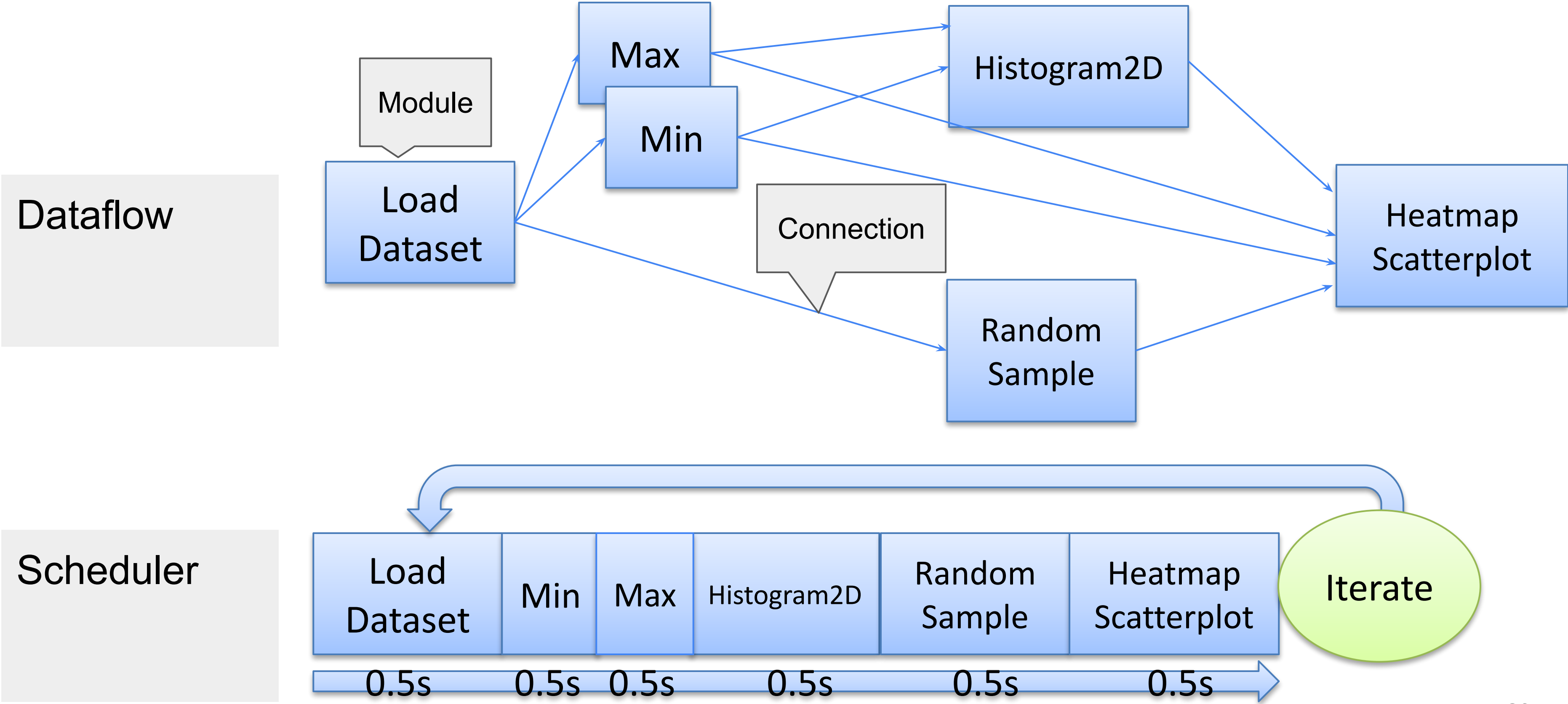

```
data = CSVLoader('bigfile.csv')
min = Min(data)
max = Max(data)
histo = Histogram2D(data, min, max)
sample = Sample(data, 500)
plot = Scatterplot(histo, sample, min, max)
show(plot)
```

ProgressiVis: Splitting the Computation in Chunks

- Each module is given a quantum to run (0.5 s)
- At the end of its quantum, it should provide a useful result, even if partial or approximate
- Modules are run in round-robin order until they reach the end of their computation

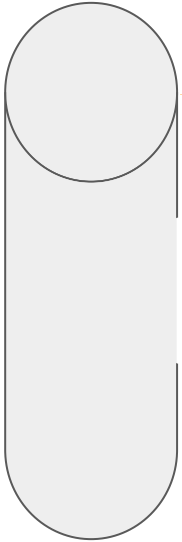
- Additionally, interaction is possible to steer and modify module parameters!

Running the Modules in **Topological Order**

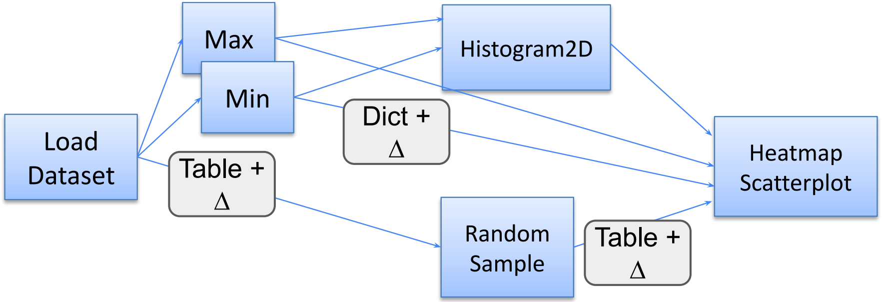


Running the Modules: What Happens with Data?

CSV File



…



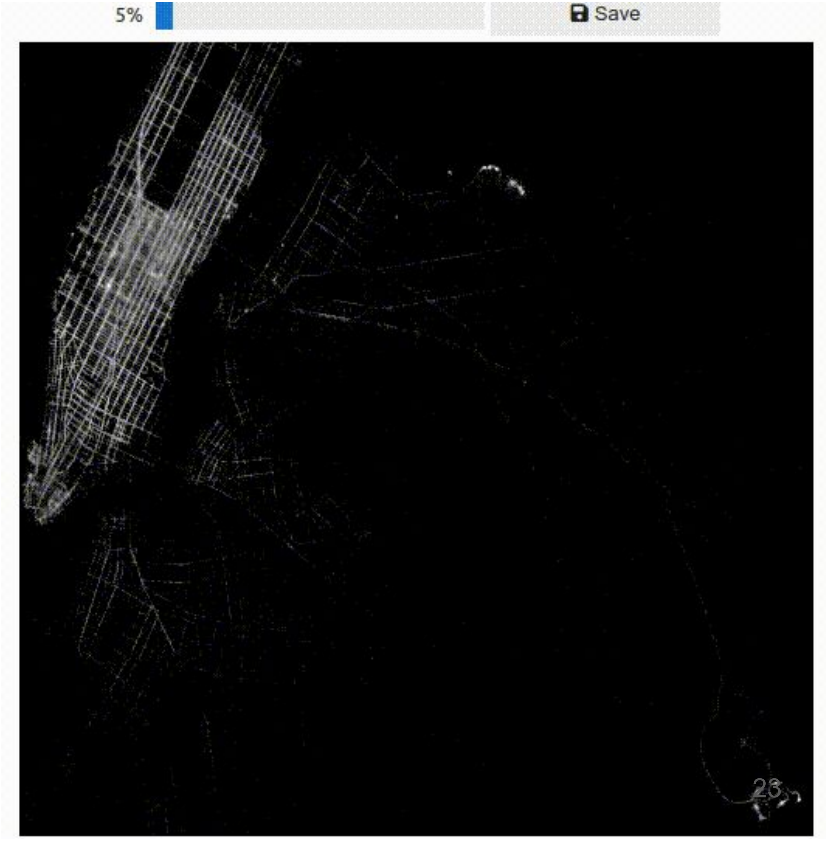
Δ = Creations
Deletions
Updates

ProgressiVis Core API

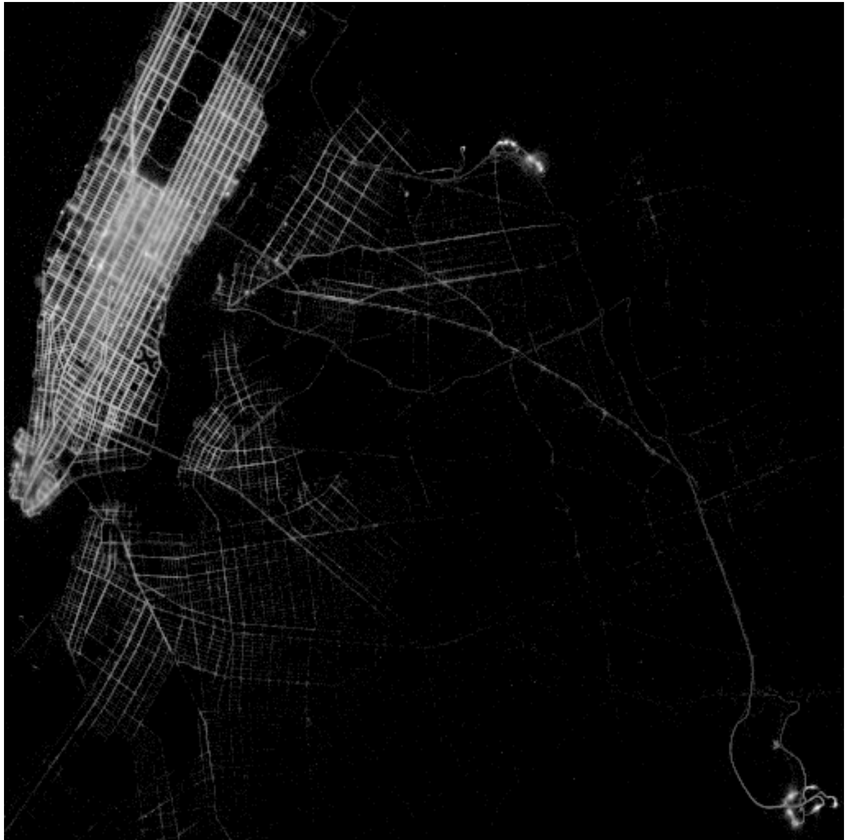
1-low-level/userguide1.0.ipynb

Simple Example using the Core API

```python
from progressivis import CSVLoader, Histogram2D, Quantiles, Heatmap

LARGE_TAXI_FILE = ("https://www.aviz.fr/nyc-taxi/"
                   "yellow_tripdata_2015-01.csv.bz2")
RESOLUTION=512

# Create the modules
csv = CSVLoader(LARGE_TAXI_FILE,
                usecols=['pickup_longitude', 'pickup_latitude'])
quantiles = Quantiles()
histogram2d = Histogram2D('pickup_longitude', 'pickup_latitude',
                          xbins=RESOLUTION, ybins=RESOLUTION)
heatmap = Heatmap()

# Connect the modules
quantiles.input.table = csv.output.result
histogram2d.input.table = quantiles.output.table
histogram2d.input.min = quantiles.output.result[0.03]
histogram2d.input.max = quantiles.output.result[0.97]
heatmap.input.array = histogram2d.output.result

# Display the Heatmap
heatmap.display_notebook()

# Run the program
csv.scheduler.task_start()
```



Simple Example using the Core API



Simple Example using the Core API

```python
1 from progressivis import CSVLoader, Histogram2D, Quantiles, Heatmap
2
3 LARGE_TAXI_FILE = ("https://www.aviz.fr/nyc-taxi/"
4                    "yellow_tripdata_2015-01.csv.bz2")
5 RESOLUTION=512
6
7 # Create the modules
8 csv = CSVLoader(LARGE_TAXI_FILE,
9                 usecols=['pickup_longitude', 'pickup_latitude'])
10 quantiles = Quantiles()
11 histogram2d = Histogram2D('pickup_longitude', 'pickup_latitude',
12                           xbins=RESOLUTION, ybins=RESOLUTION)
13 heatmap = Heatmap()
14
15 # Connect the modules
16 quantiles.input.table = csv.output.result
17 histogram2d.input.table = quantiles.output.table
18 histogram2d.input.min = quantiles.output.result[0.03]
19 histogram2d.input.max = quantiles.output.result[0.97]
20 heatmap.input.array = histogram2d.output.result
21
22 # Display the Heatmap
23 heatmap.display_notebook()
24
25 # Run the program
26 csv.scheduler.task_start()
```

- Line 8 creates the module CSVLoader
  - Same arguments than pandas.read_csv
- Line 10 creates the module Quantiles
  - Manages the quantiles of the quantitative columns
- Line 11 creates the module Histogram2D
  - Accumulates counts of two attributes on a matrix
- Line 13 creates the module Heatmap
  - Transforms the matrix into an image
- Line 16 connects the output of the CSVLoader to the input of quantiles Quantiles
- Line 17-19 connects the input of the Histogram2D to the output of the CSVLoader and Quantiles. **Note the quantiles parameters on lines 18-19.**
- Line 20 connects the output of the Histogram2D to the input of the Heatmap
- Line 23 starts the scheduler to run the program
- `csv.scheduler.task_stop()` will stop the scheduler

Summary of the Simple Example

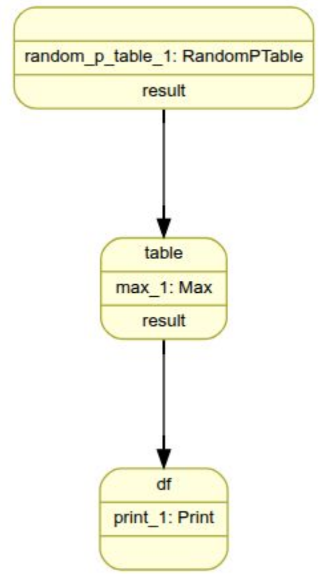
- A program is made of **modules**
- Connected to form a **dataflow**
- Underneath, a connection is done with a **Slot**
- The program is run by a **scheduler**
- What data is flowing in a connection **Slot**?
  - Tabular data: PTable
  - Dictionary data: PDict
  - IntSet: PIntSet
  - They need internal support, explained later

Look at the example on the right and identify the components, notations, and slot types

Connections can also have parameters "slot hints"

While the program runs, you can look at the module attributes live.

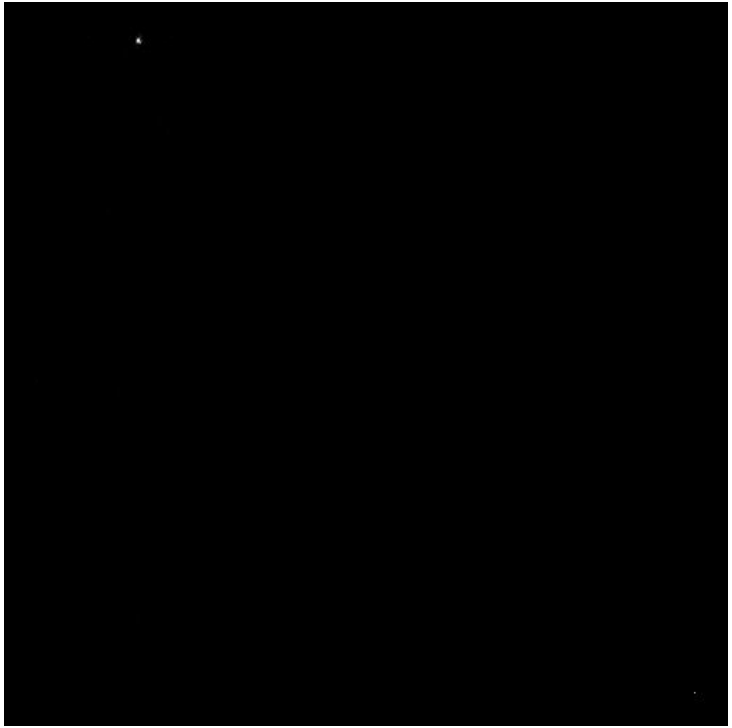
```
from progressivis import RandomTable, Max, Print

random = RandomPTable(10, rows=10000)
# produces 10 columns named _1, _2, ...
max_ = Max()
max_.input[0] = random.output.result["_1", "_2", "_3"]
# slot hints to restrict the columns to ("_1", "_2", "_3")
pr = Print(proc=self.terse)
pr.input[0] = max_.output.result
random.scheduler.task_start()
```



1-low-level/userguide1.1.ipynb

Variation 1 of the Simple Example: Simpler is not Better

```python
from progressivis import CSVLoader, Histogram2D, Min, Max, Heatmap

LARGE_TAXI_FILE = ("https://www.aviz.fr/nyc-taxi/"
                   "yellow_tripdata_2015-01.csv.bz2")
RESOLUTION=512

csv = CSVLoader(LARGE_TAXI_FILE,
                usecols=['pickup_longitude', 'pickup_latitude'])
min = Min()
max = Max()
histogram2d = Histogram2D('pickup_longitude', 'pickup_latitude',
                          xbins=RESOLUTION, ybins=RESOLUTION)
heatmap = Heatmap()

min.input.table = csv.output.result
max.input.table = csv.output.result
histogram2d.input.table = csv.output.result
histogram2d.input.min = min.output.result
histogram2d.input.max = max.output.result
heatmap.input.array = histogram2d.output.result
```



Variation 2 of the Simple Example: Extra Information Helps

```python
from progressivis import CSVLoader, Histogram2D, ConstDict, Heatmap, PDict
from dataclasses import dataclass

LARGE_TAXI_FILE = ("https://www.aviz.fr/nyc-taxi/"
                   "yellow_tripdata_2015-01.csv.bz2")
RESOLUTION=512

@dataclass
class Bounds:
    top: float = 40.92
    bottom: float = 40.49
    left: float = -74.27
    right: float = -73.68

bounds = Bounds()
col_x = "pickup_longitude"
col_y = "pickup_latitude"

csv = CSVLoader(LARGE_TAXI_FILE, usecols=[col_x, col_y])
min = ConstDict(PDict({col_x: bounds.left, col_y: bounds.bottom}))
max = ConstDict(PDict({col_x: bounds.right, col_y: bounds.top}))
histogram2d = Histogram2D(col_x, col_y,
                          xbins=RESOLUTION, ybins=RESOLUTION)
heatmap = Heatmap()

histogram2d.input.table = csv.output.result
histogram2d.input.min = min.output.result
histogram2d.input.max = max.output.result
heatmap.input.array = histogram2d.output.result
```

Return to Simple Example using the Core API

```
 1 from progressivis import CSVLoader, Histogram2D, Quantiles, Heatmap
 2
 3 LARGE_TAXI_FILE = ("https://www.aviz.fr/nyc-taxi/"
 4                    "yellow_tripdata_2015-01.csv.bz2")
 5 RESOLUTION=512
 6
 7 # Create the modules
 8 csv = CSVLoader(LARGE_TAXI_FILE,
 9                 usecols=['pickup_longitude', 'pickup_latitude'])
10 quantiles = Quantiles()
11 histogram2d = Histogram2D('pickup_longitude', 'pickup_latitude',
12                           xbins=RESOLUTION, ybins=RESOLUTION)
13 heatmap = Heatmap()
14
15 # Connect the modules
16 quantiles.input.table = csv.output.result
17 histogram2d.input.table = quantiles.output.table
18 histogram2d.input.min = quantiles.output.result[0.03]
19 histogram2d.input.max = quantiles.output.result[0.97]
20 heatmap.input.array = histogram2d.output.result
21
22 # Display the Heatmap
23 heatmap.display_notebook()
24
25 # Run the program
26 csv.scheduler.task_start()
```

Special care is needed to deal with big data:

- The Quantiles module computes min and max corresponding to the 0.03/0.97 quantiles (line 18-19) to avoid *outliers*
- Big data can always have outliers
- The syntax with brackets at the end of the connection is called a "slot hint"
    - It provides parameters to the connection
    - Here, the desired quantile in [0,1]
    - For a PTable slot, the desired columns

Continuing a Progressive Data Analysis

- Running the simplest program did not work because of the outliers
- How can we fix it, or continue the analysis with other modules?
  - Perform more computations
  - Show more visualizations
- We need to change the scheduler's Dataflow
  - But it needs to remain **valid**
- We use a Python "context manager"
- It checks before committing the changes

```
with scheduler as dataflow:
    # create and remove modules here
    # add connections
# The new dataflow is tested here
# And added to the scheduler if valid
```

Dataflow Management and Validity

To run, a ProgressiVis program should be valid.

- The input and output slots should be type-compatible
- All the modules' required slots should be connected
- There should not be any cycle in the dataflow; it should be a **directed acyclic graph (DAG)**
- ProgressiVis checks the connection types when the connection is specified
- However, when building or modifying a dataflow graph, adding modules or removing modules, the dataflow graph remains invalid until all the connections are made and dependent modules are deleted from the dataflow.
- Deleting a module can trigger a cascade of deletions
- Checking the required slots and cycles is done using a Python **context manager**

Adding, Removing, and Connecting Modules

1. Build the Program

```
from progressivis import CSVLoader, Histogram2D, Min, Max, Heatmap

LARGE_TAXI_FILE = ("https://www.aviz.fr/nyc-taxi/"
                   "yellow_tripdata_2015-01.csv.bz2")
RESOLUTION=512

csv = CSVLoader(LARGE_TAXI_FILE,
                usecols=['pickup_longitude', 'pickup_latitude'])
min = Min()
max = Max()
histogram2d = Histogram2D('pickup_longitude', 'pickup_latitude',
                          xbins=RESOLUTION, ybins=RESOLUTION)
heatmap = Heatmap()

min.input.table = csv.output.result
max.input.table = csv.output.result
histogram2d.input.table = csv.output.result
histogram2d.input.min = min.output.result
histogram2d.input.max = max.output.result
heatmap.input.array = histogram2d.output.result
```

2. Incrementally Update It

```
from progressivis import Quantiles

with csv.scheduler as dataflow:
    quant2 = Quantiles()
    hist2 = Histogram2D(col_x, col_y,
                xbins=RESOLUTION, ybins=RESOLUTION)
    heat2 = Heatmap()
    quant2.input.table = csv.output.result
    hist2.input.table = quant2.output.table
    hist2.input.min = quant2.output.result[0.03]
    hist2.input.max = quant2.output.result[0.97]
    heat2.input.array = hist2.output.result
    deps = dataflow.collateral_damage(min, max)
    print("Will remove modules:", deps)
    dataflow.delete_modules(*deps)
```

```
Will remove modules: {'max', 'heatmap_1', 'histogram2_d_1', 'min'}
```

```
heat2.display_notebook()
```

Dataflow Management and Validity

- Modules can be added and removed
- But the program should be valid:
  - Slot types should match
  - Required slots should be connected
  - The module graph should not have cycles

```
LARGE_TAXI_FILE = ...

csv = CSVLoader(
    LARGE_TAXI_FILE,
    usecols=[
        'Pickup_longitude',
        'Pickup_latitude'
    ])
m = Min(name="min")
prt = Tick(tick='.')
m.input.table = csv.output.result
prt.input.df = m.output.result
csv.scheduler.task_start()
```

To add modules:

```
with csv.scheduler as dataflow:
    M = Max(name="max")
    prt2 = Tick(tick='/')
    M.input.table = csv.output.result
    prt2.input.df = M.output.result
```

And to remove modules:

```
with csv.scheduler as dataflow:
    deps = dataflow.collateral_damage("min")
    print("The collateral damage of "
          "deleting min is:", deps)
    dataflow.delete_modules(*deps)
```

Summary of components and organization

Components so far:

- Module
- Scheduler
- Slot
- Data
- More to come:
  - Dataflow
  - Widgets

Modules so far:

- IO:
  - CSVLoader
  - Print, ConstDict
- Visualization:
  - Heatmap
- Stats:
  - Quantiles
  - Min / Max
- Table:
  - RandomPTable
- More to come:
  - core, linalg, cluster and **ipyprogressivis**

Modules API (so far)

- Core
  - Sink
  - Wait
- Io
  - CSVLoader [several]
  - ParquetLoader
  - ArrowBatchLoader
  - Every, Print, Tick
  - Variable
- Stats
  - Min, Max
  - Var, Std
  - Counter, Distinct
  - Histogram1D, Histogram2D, MCHistogram2D
  - Corr
  - Sample
  - Quantiles
  - PPCA
  - RandomPTable
  - KLLSketch
  - KernelDensity
  - TSNE
- Table
  - Constant, ConstDict
  - Aggregate
  - Index [several]
  - Binary operators
  - CategoricalQuery
  - CombineFirst
  - Filter [several]
  - GroupBy
  - Intersection
  - Join [several]
  - Merge
  - RangeQuery [several]
  - Reduce
  - Select
- Vis
  - Heatmap
  - MCScatterPlot
  - AnyVega
- Linalg
- Cluster
  - MBKMeans

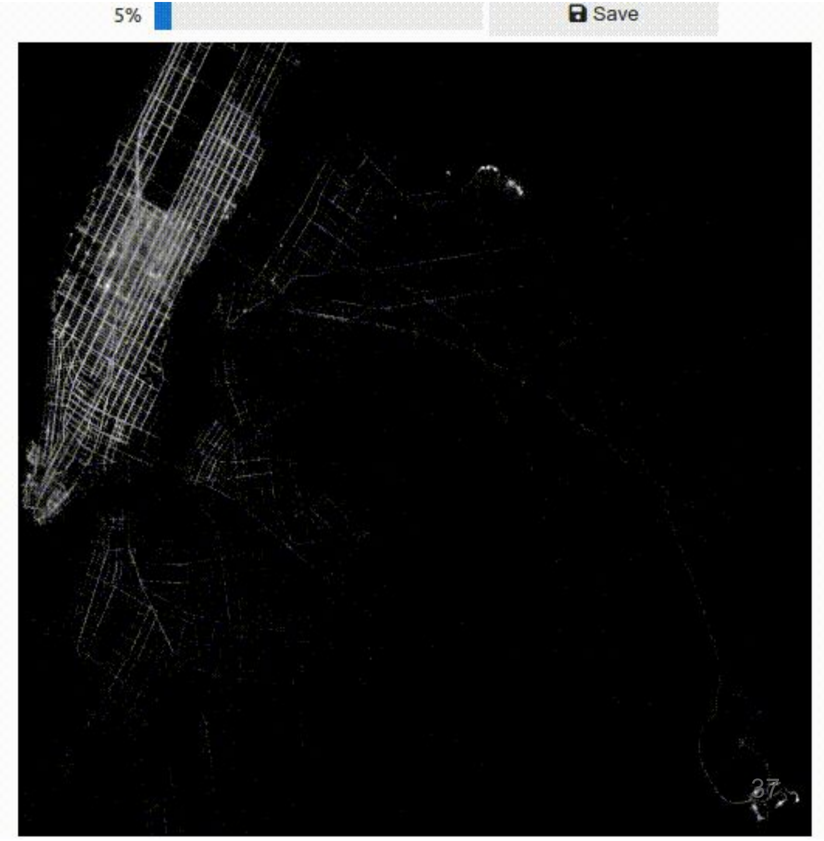
Communication with Modules

- Modules expose **attributes**:
  - **name, input, output, params, scheduler, tags, state**
- **Methods:**
  - **get_progress(), get_quality()**
  - Heatmap -> **get_image_bin**()
- They also expose **callbacks**
  - **on_start_run**
  - **on_after_run**
  - **on_ending**

See the communication with the Notebook to display the heatmap being updated.

Back to the Simple Example using the Core API

```
from progressivis import CSVLoader, Histogram2D, Quantiles, Heatmap

LARGE_TAXI_FILE = ("https://www.aviz.fr/nyc-taxi/"
                   "yellow_tripdata_2015-01.csv.bz2")
RESOLUTION=512

csv = CSVLoader(LARGE_TAXI_FILE,
                usecols=['pickup_longitude', 'pickup_latitude'])

quantiles = Quantiles()
quantiles.input.table = csv.output.result

histogram2d = Histogram2D('pickup_longitude', 'pickup_latitude',
                          xbins=RESOLUTION, ybins=RESOLUTION)
histogram2d.input.table = quantiles.output.table
histogram2d.input.min = quantiles.output.result[0.03]
histogram2d.input.max = quantiles.output.result[0.97]

heatmap = Heatmap()
heatmap.input.array = histogram2d.output.result

heatmap.display_notebook()
csv.scheduler.task_start()
```



Adding a callback to show a Heatmap

1. Create a widget `Image` with no content
2. Display the widget
3. Register a callback to the Heatmap
4. After each run
   Change the image with the new one

```
import IPython
import ipywidgets as widgets
```

```
url = "https://www.aviz.fr/wiki/uploads/Progressive/PDAV-first-page.png"
image = IPython.display.Image(url, width = 300)

widgets.Image(
    value=image.data,
    format='png',
    width=400
)
```

Adding a callback to show a Heatmap

1. Create a widget `Image` with no content
2. Display the widget
3. Register a callback to the Heatmap
4. After each run
   Change the image with the new one

In reality, the updates should be slowed down to avoid flooding the front end. Every 3-5 seconds.

```python
import ipywidgets as ipw
from IPython.display import display

# Create an empty Image widget and display it in the notebook
img = ipw.Image(value=b'\x00', width=width, height=height)
display(img)

# Define a callback that runs after the heatmap module is update
def _after_run(m: Module, run_number: int) -> None:
    assert isinstance(m, Heatmap)
    image = m.get_image_bin()  # get the image from the heatmap
    if image is not None:
        img.value = image
        # Replace the displayed image with the new one

heatmap.on_after_run(_after_run)  # Install the callback
```

`1-low-level/userguide1.4.ipynb`

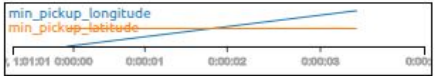
Progression and Quality

- ProgressiVis allows tracking the progression and quality of modules
- `Module.get_progress` returns a couple of value: (current, max)
    - The unit is number of items
    - Both can vary over time
- `Module.get_quality` returns a dictionary of names and floating point values. The higher, the better.
- Both can be visualized with a callback `Module.on_after_run`

```
min.get_quality()
```

```
{'min_pickup_longitude': 121.92581176757812, 'min_pickup_latitude': -0.0}
```

```
minq = display_quality(min)
minq
```




```
display_progress_bar(min)
```

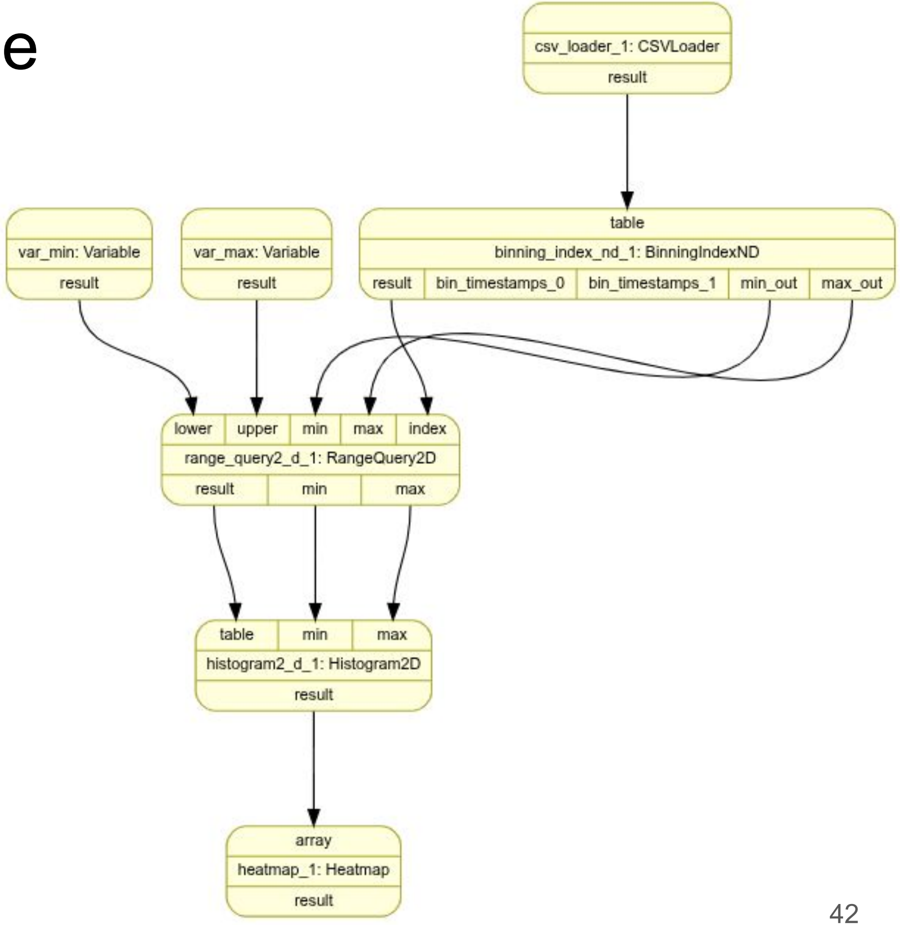
Interaction: the Variable module

- The "Variable" module simply forwards the message it receives using the `Module.from_input(msg: Dict[str, Any])` method to its output slot as a PDict
- Other modules can interpret the message in specific ways
- For interactive programs, a jupyter notebook widget can create the message in its callback
- When `from_input` is called, the scheduler assumes the interaction should be fast and optimizes temporarily the dataflow, only running a minimum set of modules

1-low-level/userguide1.3.ipynb

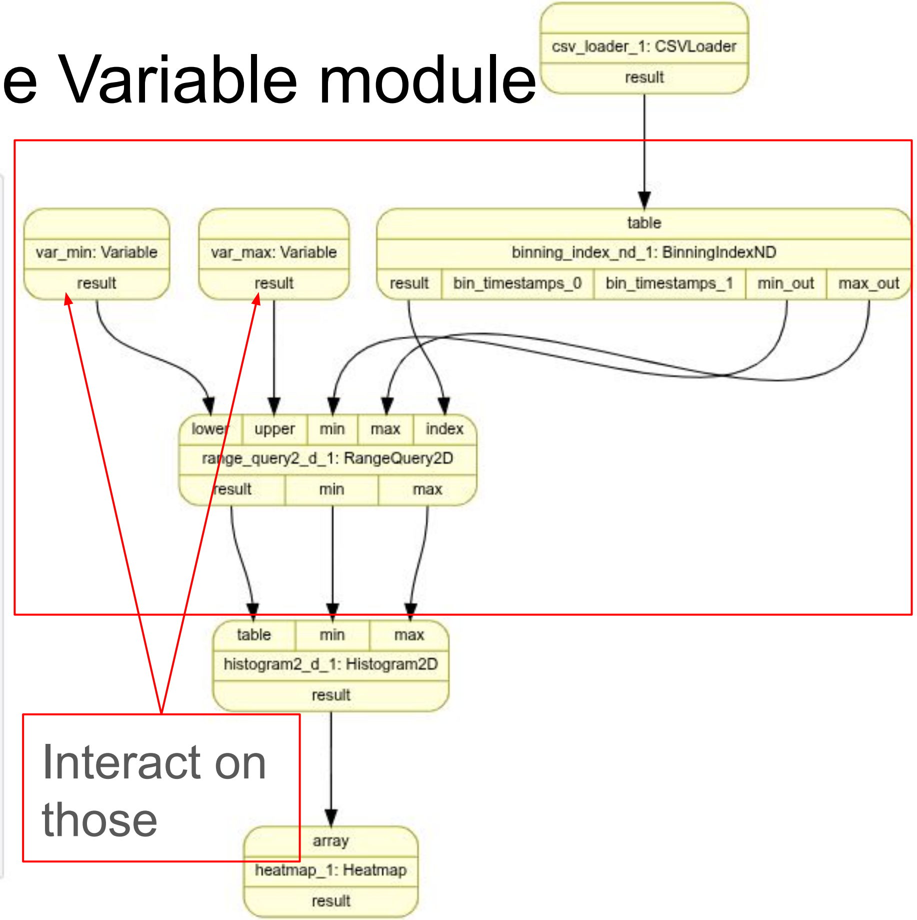
Interaction: the Variable module

```
from progressivis import (CSVLoader, Histogram2D, Heatmap, PDict,
                          BinningIndexND, RangeQuery2D, Variable)

col_x = "pickup_longitude"
col_y = "pickup_latitude"

csv = CSVLoader(LARGE_TAXI_FILE, usecols=[col_x, col_y])
index = BinningIndexND()
index.input.table = csv.output.result[col_x, col_y]
query = RangeQuery2D(column_x=col_x, column_y=col_y)
var_min = Variable(name="var_min")
var_max = Variable(name="var_max")
query.input.lower = var_min.output.result
query.input.upper = var_max.output.result
query.input.index = index.output.result
query.input.min = index.output.min_out
query.input.max = index.output.max_out

histogram2d = Histogram2D(col_x, col_y, xbins=RESOLUTION, ybins=RESOLUTION)
histogram2d.input.table = query.output.result
histogram2d.input.min = query.output.min
histogram2d.input.max = query.output.max

heatmap = Heatmap()
heatmap.input.array = histogram2d.output.result
heatmap.display_notebook()
csv.scheduler().task_start();
```



Interaction: from_input() and the Variable module

```python
from progressivis import (CSVLoader, Histogram2D, Heatmap, PDict,
                          BinningIndexND, RangeQuery2D, Variable)

col_x = "pickup_longitude"
col_y = "pickup_latitude"

csv = CSVLoader(LARGE_TAXI_FILE, usecols=[col_x, col_y])
index = BinningIndexND()
index.input.table = csv.output.result[col_x, col_y]
query = RangeQuery2D(column_x=col_x, column_y=col_y)
var_min = Variable(name="var_min")
var_max = Variable(name="var_max")
query.input.lower = var_min.output.result
query.input.upper = var_max.output.result
query.input.index = index.output.result
query.input.min = index.output.min_out
query.input.max = index.output.max_out

histogram2d = Histogram2D(col_x, col_y, xbins=RESOLUTION, ybins=RESOLUTION)
histogram2d.input.table = query.output.result
histogram2d.input.min = query.output.min
histogram2d.input.max = query.output.max

heatmap = Heatmap()
heatmap.input.array = histogram2d.output.result
heatmap.display_notebook()
csv.scheduler().task_start();
```



Control with Widgets

- Line 11-12 set the current min/max values
- Two `FloatRangeSlider` widgets are created for longitude and latitude lines 14 and 26
- An observer function is called when the widgets change and update the variables using the Module.from_input method.
- The observer is attached to the widget on line 45-46
- The widgets are displayed on line 47

See https://ipywidgets.readthedocs.io/ for informations on the Jupyter Lab widgets

```python
6 # Define the bounds for the range-slider widgets
7 bnds_min = PDict({col_x: bounds.left, col_y: bounds.bottom})
8 bnds_max = PDict({col_x: bounds.right, col_y: bounds.top})
9
10 # Assign an initial value to the min and max variables
11 await var_min.from_input(bnds_min)
12 await var_max.from_input(bnds_max);
13
14 long_slider = widgets.FloatRangeSlider(
15     value=[bnds_min[col_x], bnds_max[col_x]],
16     min=bnds_min[col_x],
17     max=bnds_max[col_x],
18     step=(bnds_max[col_x]-bnds_min[col_x])/10,
19     description='Longitude:',
20     disabled=False,
21     continuous_update=False,
22     orientation='horizontal',
23     readout=True,
24     readout_format='.1f',
25 )
26 lat_slider = widgets.FloatRangeSlider(
27     value=[bnds_min[col_y], bnds_max[col_y]],
28     min=bnds_min[col_y],
29     max=bnds_max[col_y],
30     step=(bnds_max[col_y]-bnds_min[col_y])/10,
31     description='Latitude:',
32     disabled=False,
33     continuous_update=False,
34     orientation='horizontal',
35     readout=True,
36     readout_format='.1f',
37 )
38 def observer(_):
39     async def _coro():
40         long_min, long_max = long_slider.value
41         lat_min, lat_max = lat_slider.value
42         await var_min.from_input({col_x: long_min, col_y: lat_min})
43         await var_max.from_input({col_x: long_max, col_y: lat_max})
44     aio.create_task(_coro())
45 long_slider.observe(observer, "value")
46 lat_slider.observe(observer, "value")
47 widgets.VBox([long_slider, lat_slider])
```

Adjusting the Viewport

- Create range filters for lat and long
- Specify the min/max for the histogram2d
- Connect modules to widgets

Showing a progress bar and a quality visualization is possible too, but the code becomes long!

See `1-low-level/userguide1.3.ipynb`

For simple cases, we offer a simpler interface.

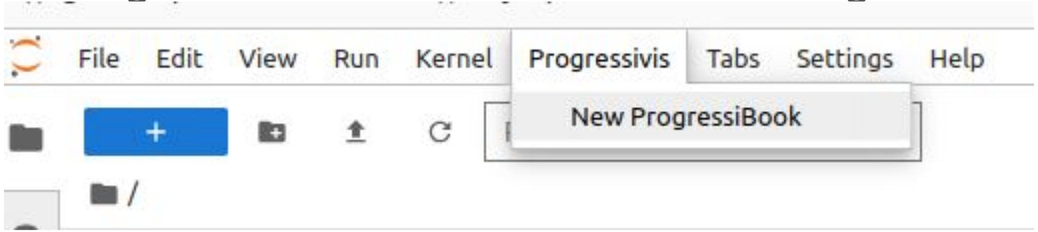
Core API Summary

- Many new concepts
  - Module
  - Connections through typed slots
  - Scheduler
  - Dataflow
  - Progression Module.get_progress()
  - Quality Module.get_quality()
  - Callbacks
- Three types of data (so far):
  - PTable -> like a DataFrame, a dictionary of typed columns
  - PDict -> a dictionary associating a key (string) to a value
  - PIntSet -> a set of integers, from 0 to 2^32-1 (4 Billions)

Progressive Notebooks

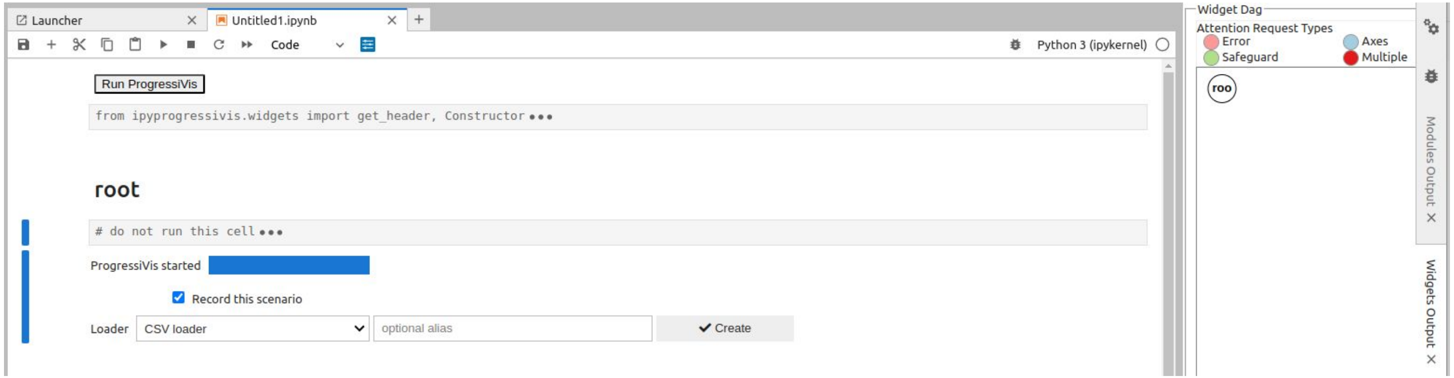
Progressive Notebooks

- The best environment to use ProgressiVis is the Jupyter lab notebook with extensions supported in the python package called `ipyprogressivis`.
- It provides:
  - Chained widgets to avoid the boilerplate code seen with the core API
    - Create progressive pipelines interactively
  - A side visualization pane to navigate between active cells (DAG widget)
  - A control panel with a graph visualization of the running modules
  - Recording progressive scenarios and replaying them
  - Improved rendering when a previously developed scenario is reopened
- A Progressive Notebook is created using the ProgressiBook extension:



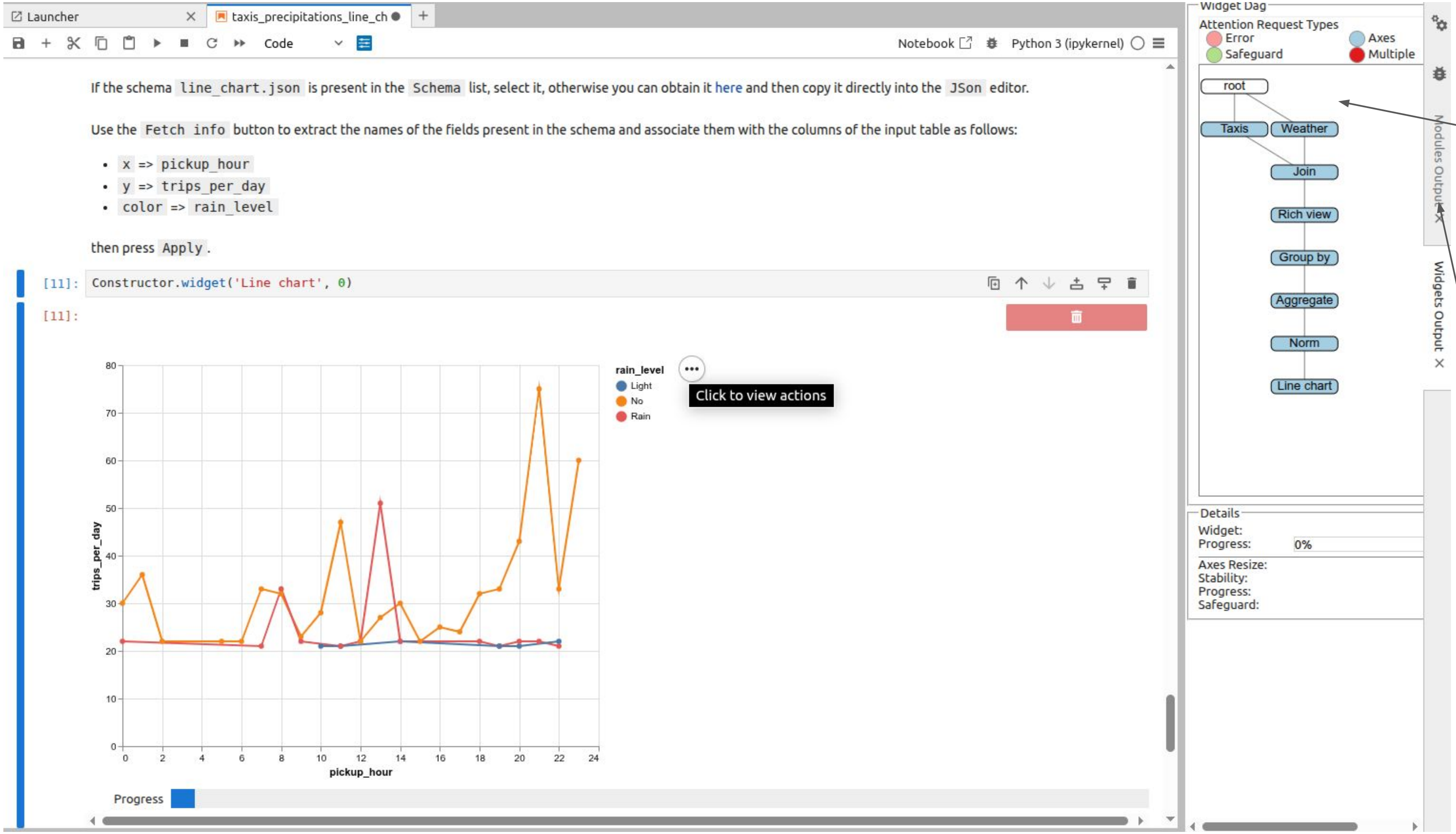
Progressive Notebooks

DAG Widget



- A starting box appears
- A scenario is usually made of a loader, with processing and visualizations
- Clicking on the button provides a guided list of possible **chaining widgets**
- It provides guidance in the creation of a scenario
- A saved scenario can be replayed

- The Simple Example using the Taxi Dataset is easy to reproduce
- It comes with a few enhancements



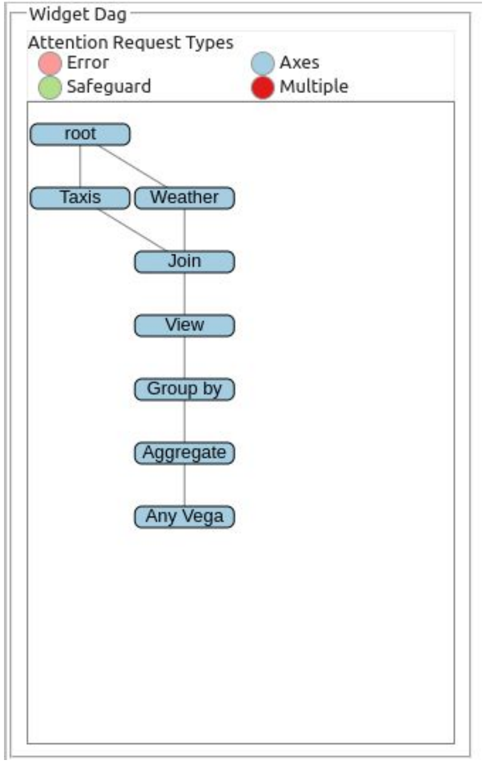
DAG Widget

Control Pane

Comments on the DAG Widget

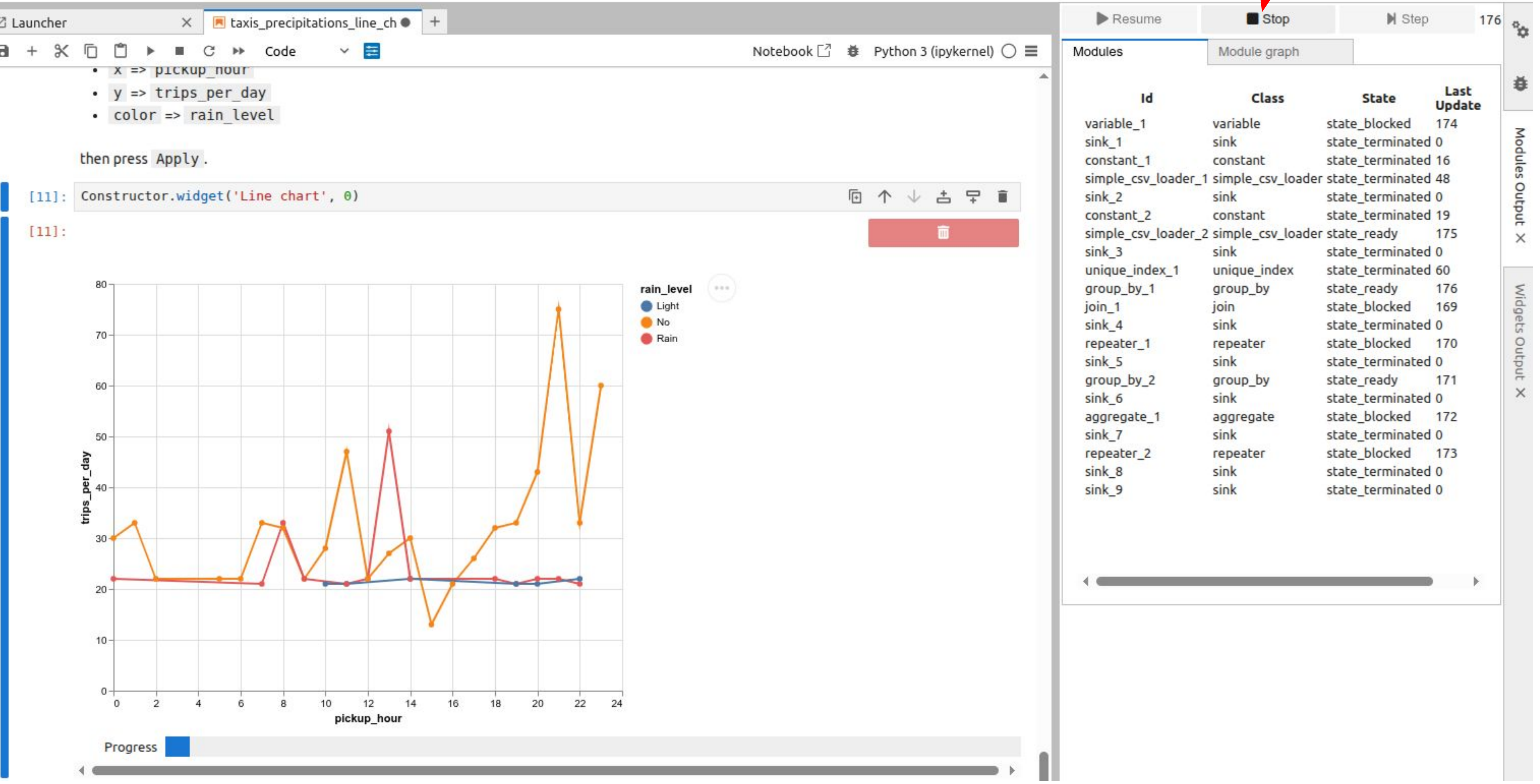
Goals and usage

- Visualizes the topology of the progressive dataflow
- Shows the state of the Chaining Widgets with colors
- Allows non-linear navigation on the notebook
  - To visit cells that continue to work
- The name can be specified at the creation of the widget
- The color indicates the state of the Chaining Widgets
  - White: Not started
  - Pink: Error during the execution
  - Blue: Running

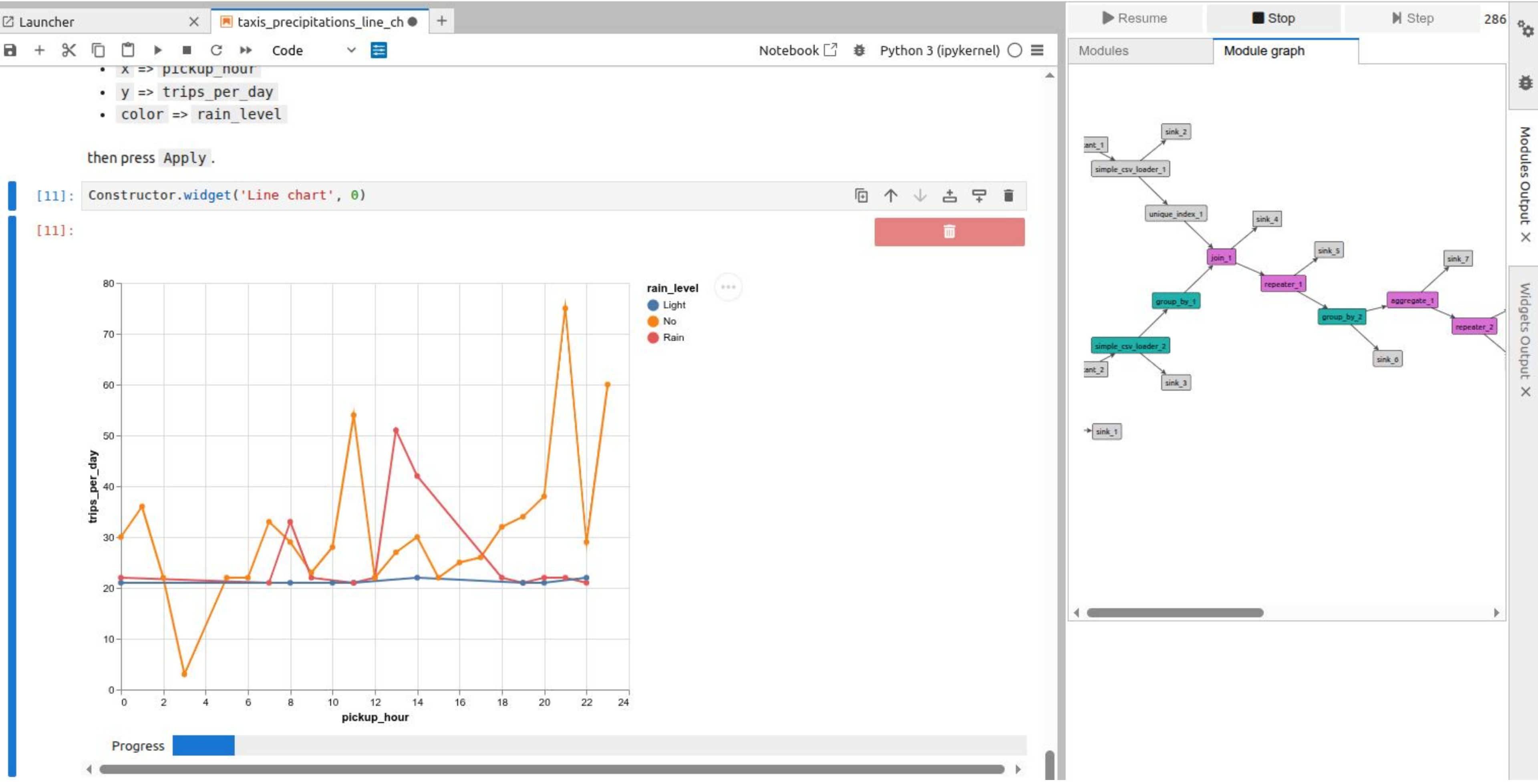


Inspecting the Running Modules

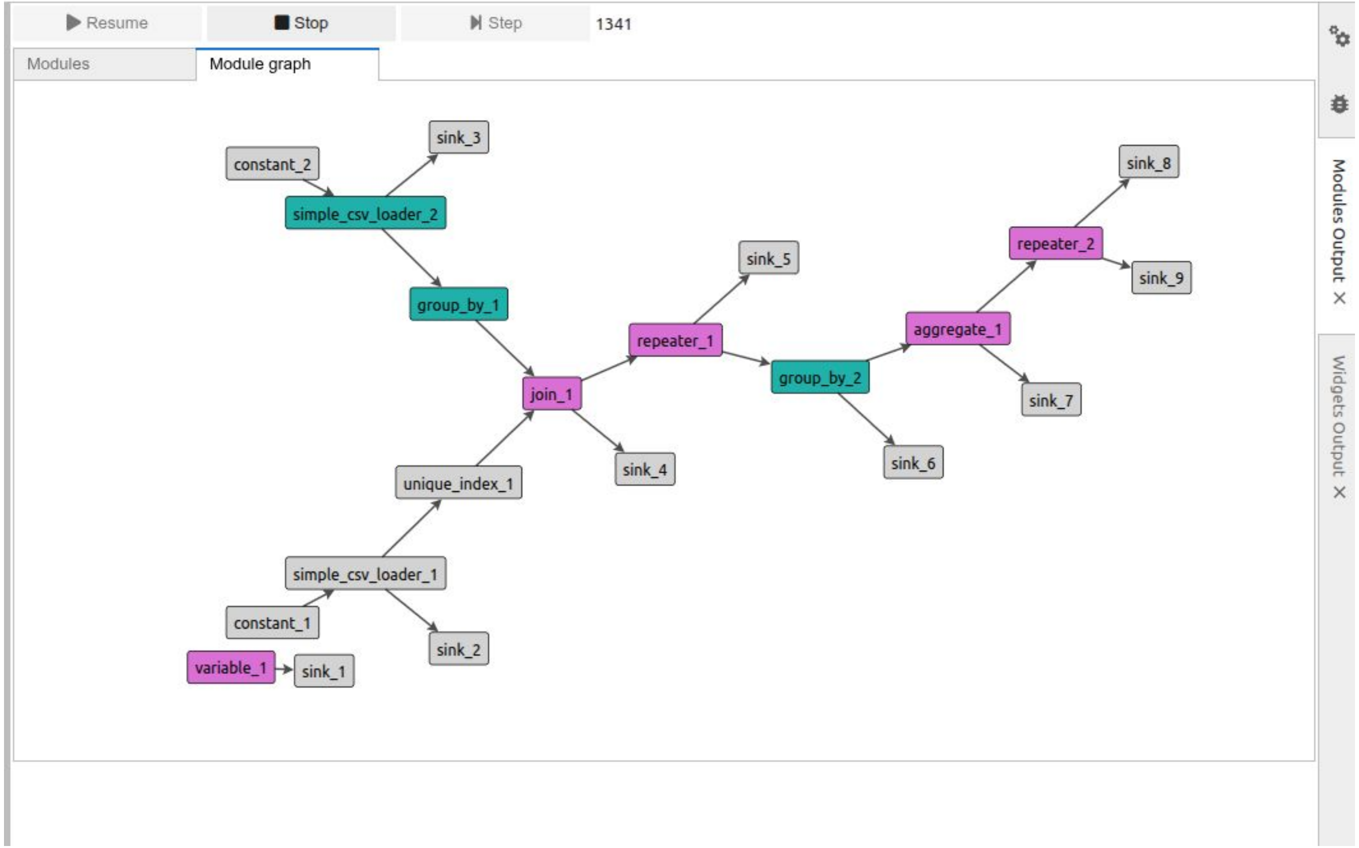
Stop button



Inspecting the Running Modules Graph



Control Panel with Module Graph



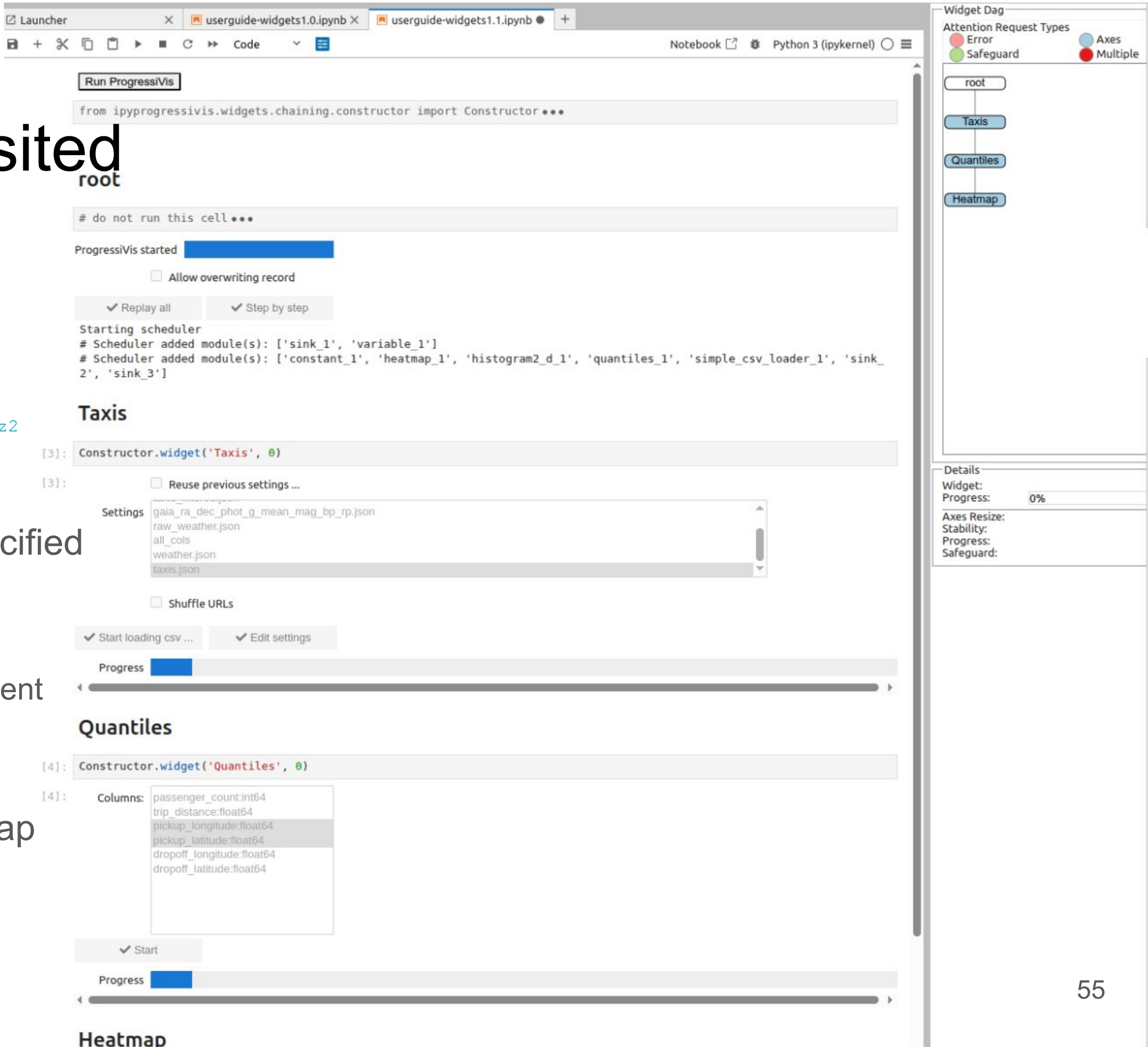
Simple Example Revisited

Let's see how to load the Taxi Dataset

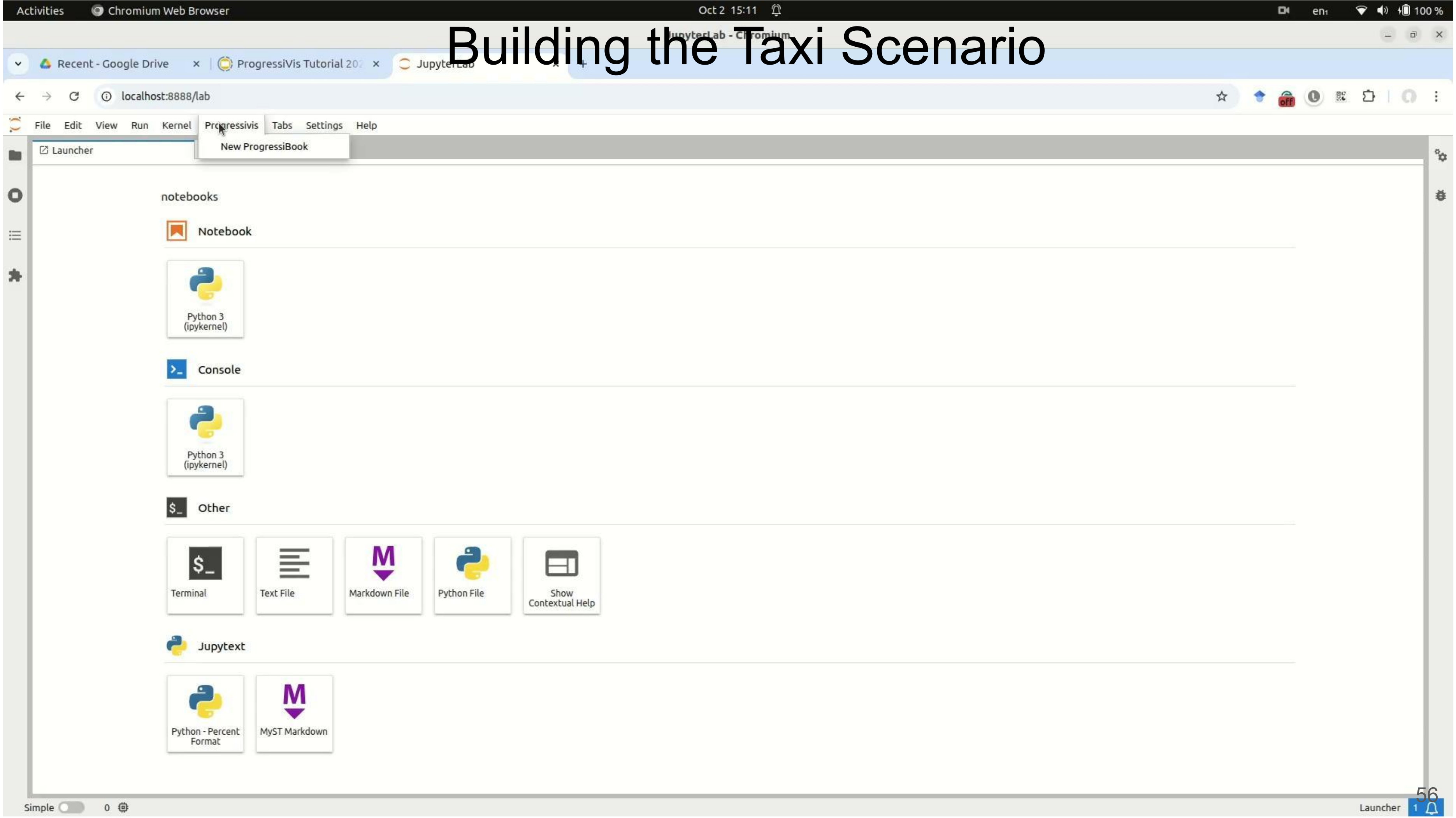
The URL is:

https://www.aviz.fr/nyc-taxi/yellow_tripdata_2015-01.csv.bz2

1. Create a new ProgressiBook
2. Create a CSVLoader with the specified URL
3. Explore the CSVSniffer
   a. Heavyweight interactive component
4. Chain the loader to the Quantiles
   a. See the parameters
5. Chain the Quantiles to the Heatmap
   a. See the parameters



Building the Taxi Scenario



Chaining Widget List

- Data Loaders: CSV with the Sniffer, Parquet, Custom
- Table operators: Group By, Aggregate, Join, View (computed columns)
- Free coding: Snippet
- Display: Dump table, Descriptive stats, Heatmap, MultiSeries, Any Vega

Free Coding: Snippets

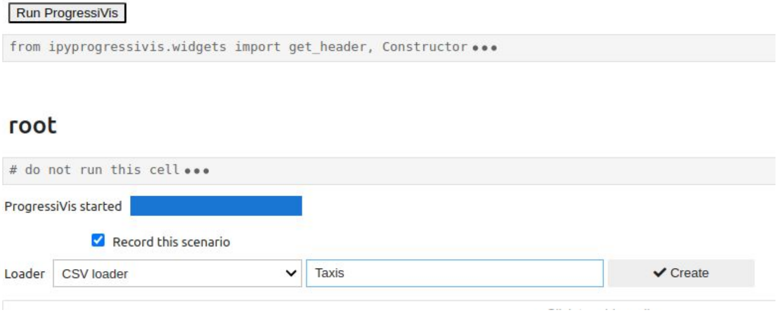
- a widget that lets you insert custom code into a Chaining Widget topology.
- the first line of the cell must begin with the comment `# progressivis-snippet`
- The user code is connected with a function with the following signature:

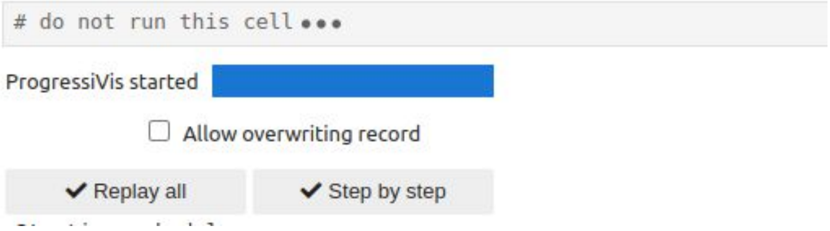
```
@register_snippet
def my_function_name(input_module: Module, input_slot: str, columns: list[str]) -> SnippetReturn | None:
    ... # the typing is optional but recommended
```

- Check the documentation for more details https://progressivis.readthedocs.io/en/latest/notebooks.html#free-coding-category
- See the two scenarios https://progressivis.readthedocs.io/en/latest/gallery.html#a-scenario-using-progressivis-snippets

Progressive Notebooks Saving and Replaying

- Recording a scenario
  - check the box when starting, and save the notebook
- Replay a scenario
  - Run it, then choose to Replay all or Step by step
- Replay in read-only or rewrite mode?
  - Step by step can be rewritten
- Unfortunately, the recorded Progressive Notebooks are not always self explanatory
- Widget states are stored internally and not exposed, this is a limitation of notebooks



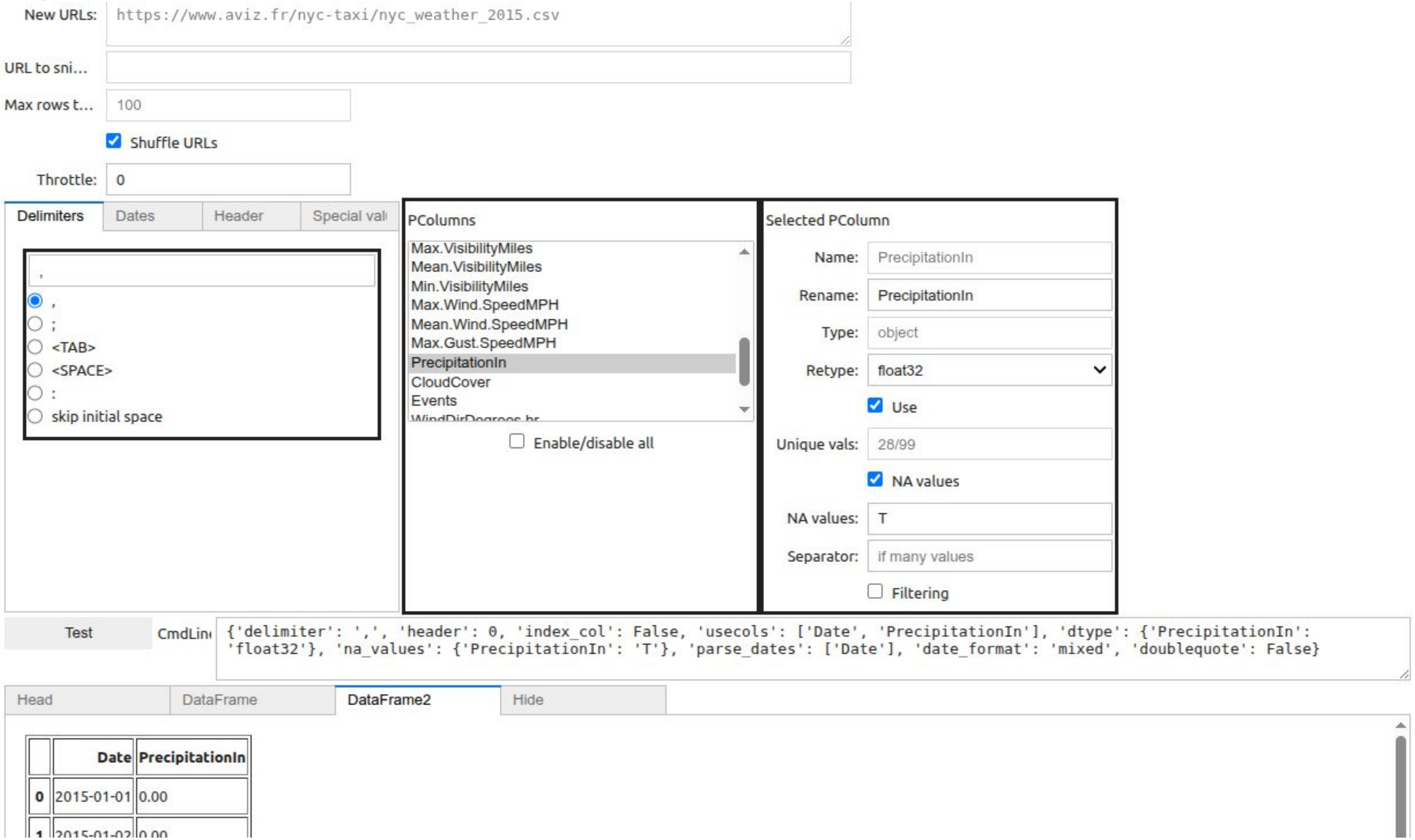


Complete Scenario: Analyzing NYC Taxis when it Rains

Let's imagine this scenario

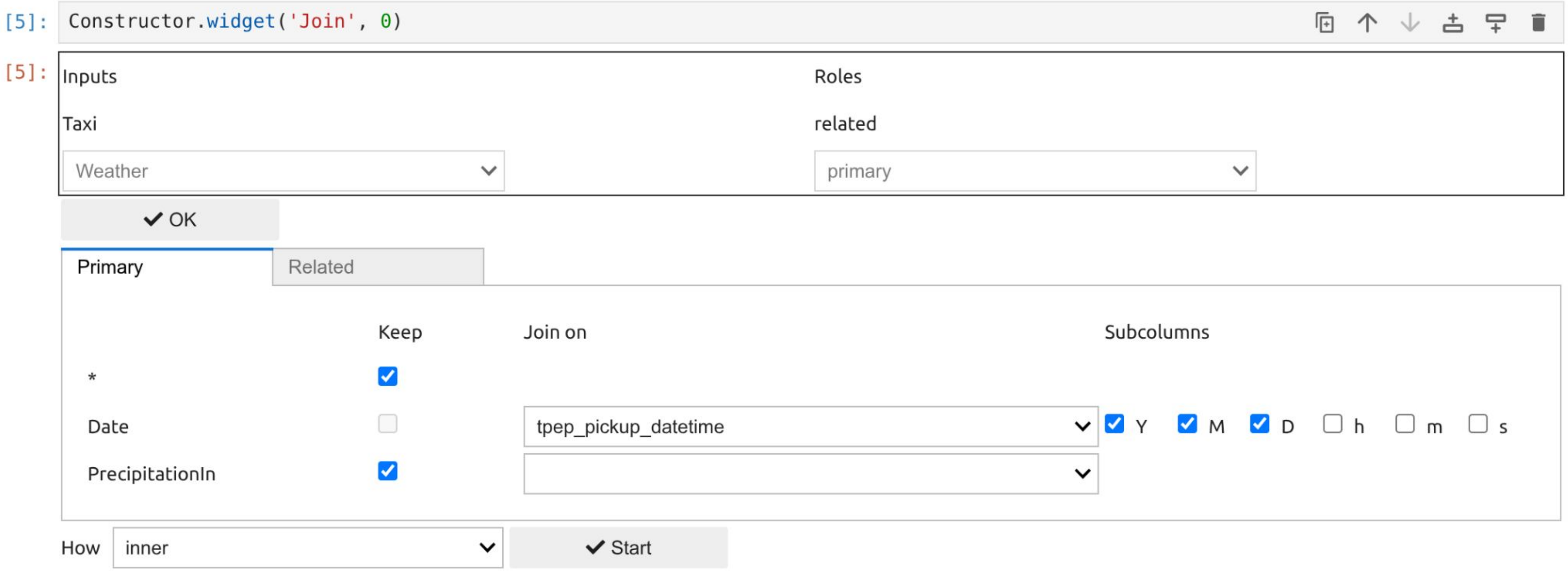
- In 2015, a New York Times journalist receives complains that there are less taxis when it rains in NYC
- She wants to check if this is true quickly
- She builds a progressive pipeline:
  - Load weather report with daily precipitation in NYC (check the weird format with the sniffer)
  - Load the taxi dataset for 2015
  - Aggregate taxis by day
  - Join them by day and aggregate the taxi trip count-> each days has a precipitation
  - Compute the correlation between precipitation and taxi count
- See notebooks/trip_rain_corr.ipynb

Loading the weather file (see https://github.com/zonination/weather-us)



Joining Tables

The smallest is usually the primary!

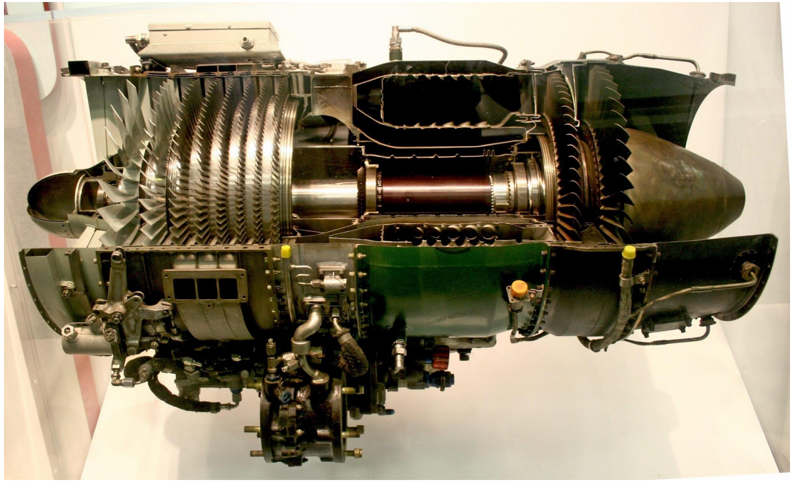


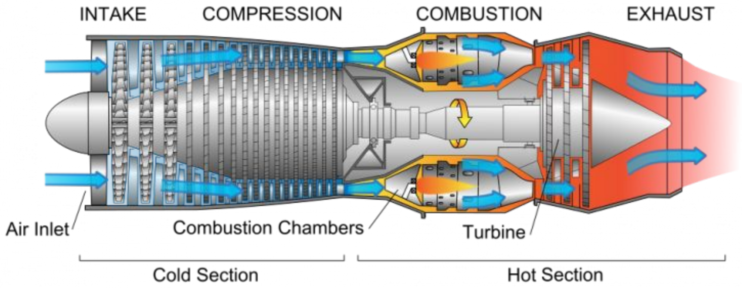
Progressive Notebooks Summary

- Simple analyses are easy to do
- Several supporting tools to monitor the program and specify parameters
- Can be saved and run again
- Not so great to understand how they work, many details are hidden in widgets
- Still a few glitches, the DAG Widget and control panels should be closed to avoid clashes…
- All the widgets can be reused in normal notebooks!

Internals of ProgressiVis

- Basic principles
  - Cooperative scheduling
  - Time predictor
  - Data updates and Reset
  - Atomic update of the Scheduler Dataflow
- Scheduler / Dataflow
- Module
- Slot
- Data structures
  - PTable, PDict, PIntSet
  - ChangeManager / Delta
- Widgets
- Visualizations





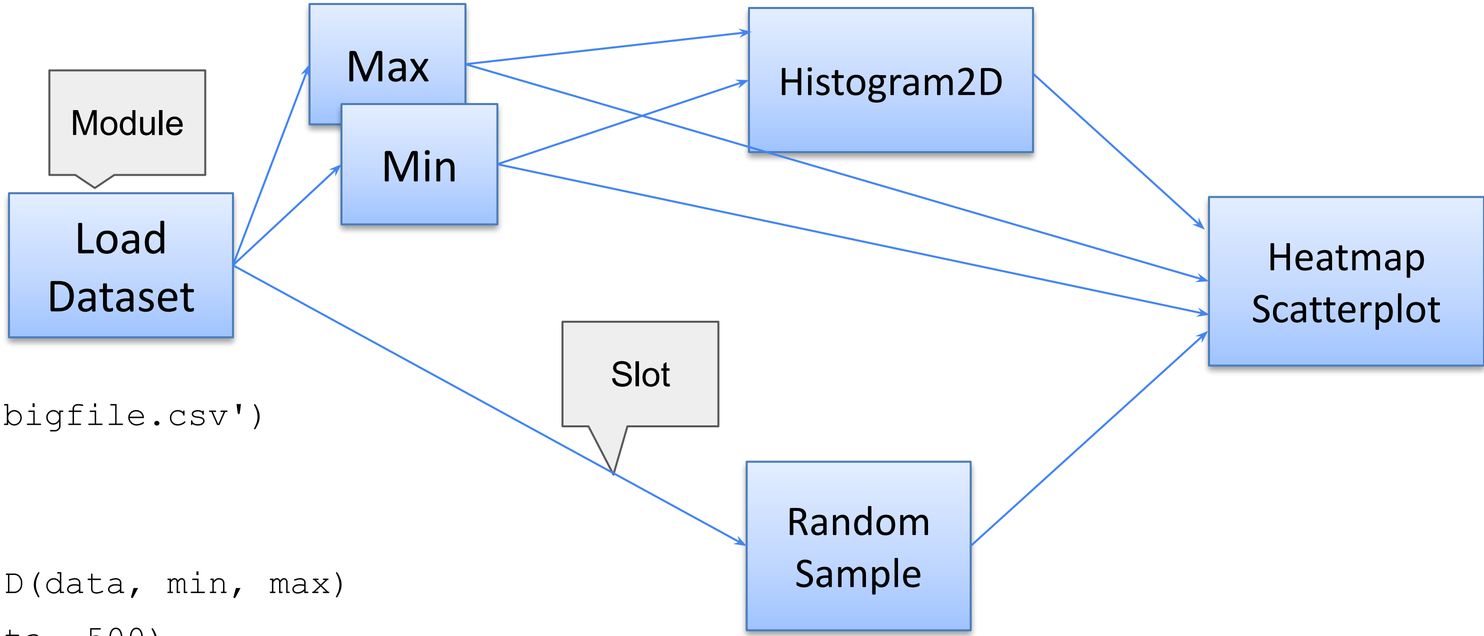
Basic Principles of ProgressiVis

- Building a Progressive Dataflow
- Cooperative scheduling
- Time predictor
- Data updates and Reset
- Atomic update of the Scheduler Dataflow

Back to the Simple Example

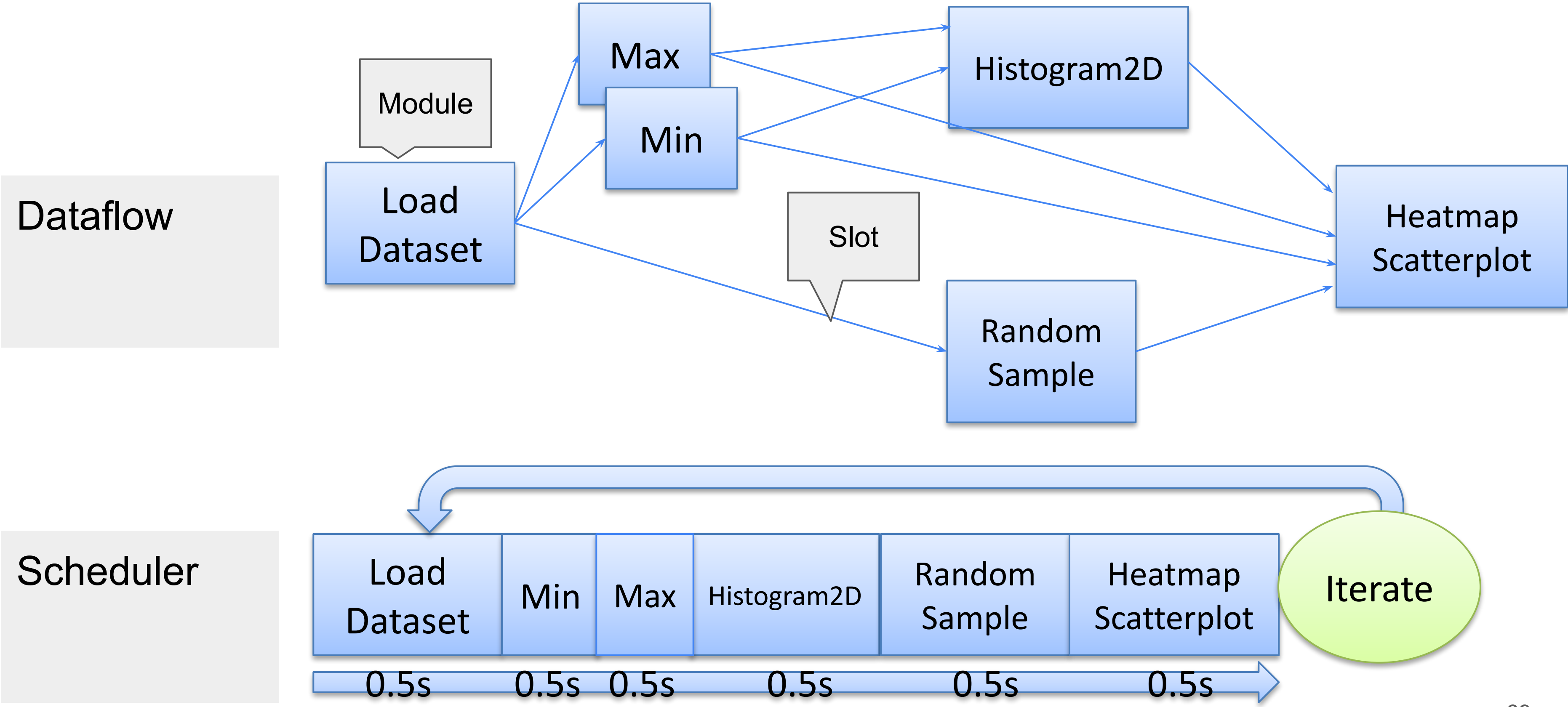
```
data = CSVLoader('bigfile.csv')
min = Min(data)
max = Max(data)
histo = Histogram2D(data, min, max)
sample = Sample(data,  500)
plot = Scatterplot(histo, sample, min, max)
show(plot)
```

Transform a Program into a **Dataflow**



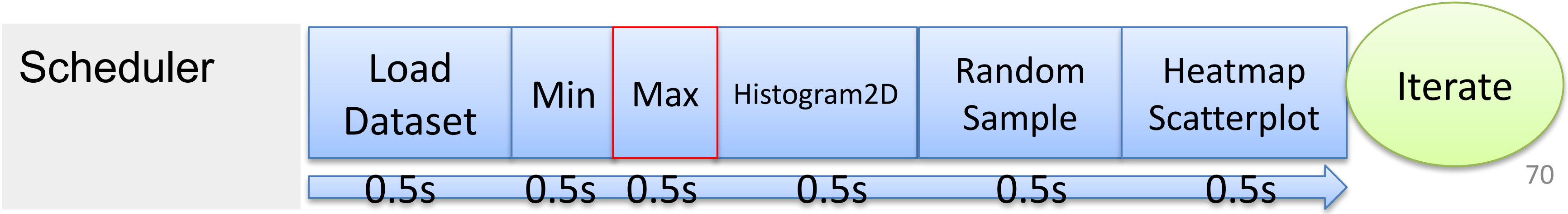

```
data = CSVLoader('bigfile.csv')
min  = Min(data)
max  = Max(data)
histo = Histogram2D(data, min, max)
sample = Sample(data, 500)
plot = Scatterplot(histo, sample, min, max)
show(plot)
```

Running the Modules in **Topological Order**



Running the Progressive Program

- Modules are run by the Scheduler in round-robin order, several times until the end of their computation
- The Scheduler tests each module with `is_ready` and, if True, calls `run` given a quantum (0.5 s by default, specified as a Module's parameter)
- At the end of its quantum, the module should provide a **useful result**, even if partial or approximate
- Cooperative scheduling means the module should return voluntarily!
- The scheduler removes terminated modules each time it reaches the end



3-module-creation/small_csv_loader-v1.py

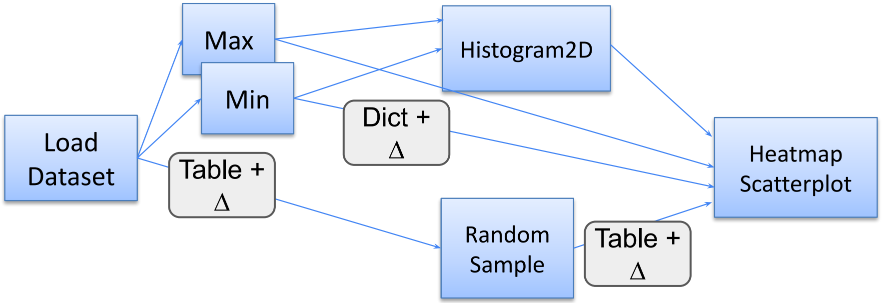
The Module SimpleMax

- Computes the max value for all the PTable columns of its input table
- Returns a PDict with the column names as keys and max value as values

```python
import numpy as np
from typing import Any
from progressivis import (Module, ReturnRunStep, PTable, PDict,
                          document, def_input, def_output)
from progressivis.core.utils import indices_len, fix_loc


@def_input("table", PTable, doc="The input PTable to process")
@def_output("result", PDict, doc=("Max value of each column"))
class SimpleMax(Module):
    def __init__(self, **kwds: Any) -> None:
        super().__init__(**kwds)
        self.default_step_size = 10000

    def reset(self) -> None:
        if self.result is not None:
            self.result.fill(-np.inf)

    def run_step(
        self, run_number: int, step_size: int, quantum: float
    ) -> ReturnRunStep:
        # Handle the changed input slots
        table_slot = self.get_input_slot("table")
        if table_slot.updated.any() or table_slot.deleted.any():
            table_slot.reset()
            table_slot.update(run_number)
            self.reset()
        # Extract the new chunk
        indices = table_slot.created.next(length=step_size)
        steps = indices_len(indices)
        chunk = table_slot.data().loc[fix_loc(indices)]
        # Apply the operation on the chunk
        op = chunk.max(keepdims=False)
        # Update the result
        if self.result is None:
            self.result = PDict(op)
        else:
            for k, v in self.result.items():
                self.result[k] = np.fmax(op[k], v)
        # Return the next state and number of steps handled
        if table_slot.has_buffered():
            next_state = Module.state_ready
        else:
            next_state = Module.state_blocked
        return self._return_run_step(next_state, steps)
```

Running the Modules: What Happens with Data?

…



Δ = Creations
Deletions
Updates

The Module Interface

```
class Module:
    name: str
    input: InputSlots
    output: OutputSlots
    params: Dict[str, Any]
    scheduler: Scheduler

    tags: Set[str]
    default_step_size: int = 100
    state: ModuleState

    def describe(self) -> None:
    def get_progress(self) -> Tuple[int, int]:
    def get_quality(self) -> Dict[str, float] | None:
    def get_data(self, name: str, hint: Any = None) -> Any:
    async def from_input(
        self, msg: D ict[str, Any], stop_iter: bool = False
    ) -> str:

    def is_ready(self) -> bool:
    def run(self, run_number: int) -> None:
    def run_step(
        self, run_number: int, step_size: int, quantum: float
    ) -> ReturnRunStep:
    def last_update(self) -> int:
    def last_time(self) -> float:

    def suspend(self) -> None:
    def resume(self) -> None:

    def on_start_run(self, proc: ModuleProc, remove:  bool = False) -> None:
    def on_after_run(self, proc: ModuleProc, remove:  bool = False) -> None:
    def on_ending(self, proc: ModuleCb, remove:  bool = False) -> None:
```

```
class ModuleState(IntEnum):
    state_created = 0
    state_ready = 1
    state_running = 2
    state_blocked = 3
    state_suspended = 4
    state_zombie = 5
    state_terminated = 6
    state_invalid = 7
```

```
class ReturnRunStep(NamedTuple):
    next_state: ModuleState
    steps_run: int
```

Defining a new Module: SimpleMax

- Define
  - Inputs
  - Outputs
  - Parameters
- Implement the method run_step
- When it makes sense, implement the method get_quality

- The SimpleMax Module defines
  - An input slot "table" that takes a PTable or derived
  - An output slot "result" that returns a PDict, for each column of the PTable, it associates the "max" value computed so far
  - No specific parameter
- The run_step maintains the max value when receiving new chunks
  - But if the table is modified, it has deleted items or updated, items, the module will reset and restart from scratch
- The get_quality is very easy to do here

Unpacking the SimpleMax Module (1)

```
7
8 @def_input("table", PTable, doc="The input PTable to process")
9 @def_output("result", PDict, doc=("Max value of each column"))
10 class SimpleMax(Module):
11     def __init__(self, **kwds: Any) -> None:
12         super().__init__(**kwds)
13         self.default_step_size = 10000
14
15     def reset(self) -> None:
16         if self.result is not None:
17             self.result.fill(-np.inf)
18
19     def run_step(
20         self, run_number: int, step_size: int, quantum: float
21     ) -> ReturnRunStep:
22         # Handle the changed input slots
23         table_slot = self.get_input_slot("table")
24         if table_slot.updated.any() or table_slot.deleted.any():
25             table_slot.reset()
26             table_slot.update(run_number)
27             self.reset()
28         # Extract the new chunk
29         indices = table_slot.created.next(length=step_size)
30         steps = indices_len(indices)
31         chunk = table_slot.data().loc[fix_loc(indices)]
32         # Apply the operation on the chunk
33         op = chunk.max(keepdims=False)
34         # Update the result
35         if self.result is None:
36             self.result = PDict(op)
37         else:
38             for k, v in self.result.items():
39                 self.result[k] = np.fmax(op[k], v)
40         # Return the next state and number of steps handled
41         if table_slot.has_buffered():
42             next_state = Module.state_ready
43         else:
44             next_state = Module.state_blocked
45         return self._return_run_step(next_state, steps)
```

- Python decorators to shorten the code
- Declaration of input slots, outputs, and parameters, with their type and doc
- The main method is "run_step" line 19
- When the method is called, the table has changed since the last run
  - With rows **created, updated, or deleted**
- If we had rows updated or deleted:
  - reset is called
  - and everything becomes "created" again
- Otherwise, there are only created items
- The chunk max is computed line 33
- The result is created on l. 36 or updated 38-39

The Module.run_step Method

```
def run_step(
        self, run_number: int, step_size: int, quantum: float
    ) -> ReturnRunStep:
```

- Parameters:
  - run_number: a unique number, increased each time the scheduler runs a module
  - step_size: the number of "steps" the method should run to stay within its quantum
  - quantum: the specified quantum for this module
- Return value: a pair of values
  - the **state** of the module after running and
  - the number of steps actually run by the module

- Structure of the method:
  a. Check for changes in the input slots, using the **delta**s returned by the slots
  b. Perform the computation abiding by the "step_size" argument
  c. Store the results in the output slots
  d. Return the new state of the module and the number of steps really done
     - Inform the scheduler if the module should die (zombie) or not
     - If it needs data from previous modules (blocked) or not (ready)
     - Of the work done to update the module speed in the Time Predictor

`3-module-creation/simple_max-v1.py`

Unpacking the Max Module (2): Time Prediction

```python
    def run_step(
        self, run_number: int, step_size: int, quantum: float
    ) -> ReturnRunStep:
        # Handle the changed input slots
        table_slot = self.get_input_slot("table")
        if table_slot.updated.any() or table_slot.deleted.any():
            table_slot.reset()
            table_slot.update(run_number)
            self.reset()
        # Extract the new chunk
        indices = table_slot.created.next(length=step_size)
        steps = indices_len(indices)
        chunk = table_slot.data().loc[fix_loc(indices)]
        # Apply the operation on the chunk
        op = chunk.max(keepdims=False)
        # Update the result
        if self.result is None:
            self.result = PDict(op)
        else:
            for k, v in self.result.items():
                self.result[k] = np.fmax(op[k], v)
        # Return the next state and number of steps handled
        if table_slot.has_buffered():
            next_state = Module.state_ready
        else:
            next_state = Module.state_blocked
        return self._return_run_step(next_state, steps)
```

- The arguments of run_step?
  - **run_number**: scheduler iteration number
  - **step_size**: number of "steps" to perform
  - **quantum**: the quantum
  - Should return a Named Pair ("next_state": …, "steps_run": …)
- Time prediction and step_size?
  - How many items can be handled in 0.5 s?
  - If 10,000 items are handled in 0.1 s …
- The Module monitors the run time and the steps run and update the module speed.
- The unit of the "step" is up to the module, but usually # of items to process

`3-module-creation/simple_max-v1.py`

Unpacking the Max Module (3): Change Management

```
19    def run_step(
20        self, run_number: int, step_size: int, quantum: float
21    ) -> ReturnRunStep:
22        # Handle the changed input slots
23        table_slot = self.get_input_slot("table")
24        if table_slot.updated.any() or table_slot.deleted.any():
25            table_slot.reset()
26            table_slot.update(run_number)
27            self.reset()
28        # Extract the new chunk
29        indices = table_slot.created.next(length=step_size)
30        steps = indices_len(indices)
31        chunk = table_slot.data().loc[fix_loc(indices)]
32        # Apply the operation on the chunk
33        op = chunk.max(keepdims=False)
34        # Update the result
35        if self.result is None:
36            self.result = PDict(op)
37        else:
38            for k, v in self.result.items():
39                self.result[k] = np.fmax(op[k], v)
40        # Return the next state and number of steps handled
41        if table_slot.has_buffered():
42            next_state = Module.state_ready
43        else:
44            next_state = Module.state_blocked
45        return self._return_run_step(next_state, steps)
```

Δ = Created
Updated
Deleted

- Progressive data structures track their changes
  - The CSV loader adds new lines
  - The table has "created" items for each new line
  - Using a PIntSet (super efficient data structure called a RoaringBitmap)
  - When filtering the table, the module will also receive "deleted" items
  - When changing the content of the table, the module will receive "updated" items
  - **Slot Delta**: 3 IntSet: created, updated, deleted, similar to a **SQL trigger**
- Line 29 reads the indices of the next "step_size" created items (a IntSet), can be a slice too
- Line 30 takes its length
- Line 31 retrieves the chunk table and Computes its max
- Line 36 builds the output PDict the first time
- Line 38-39 update it later, carefully

Unpacking the Max Module (4): State Management

```python
    def run_step(
        self, run_number: int, step_size: int, quantum: float
    ) -> ReturnRunStep:
        # Handle the changed input slots
        table_slot = self.get_input_slot("table")
        if table_slot.updated.any() or table_slot.deleted.any():
            table_slot.reset()
            table_slot.update(run_number)
            self.reset()
        # Extract the new chunk
        indices = table_slot.created.next(length=step_size)
        steps = indices_len(indices)
        chunk = table_slot.data().loc[fix_loc(indices)]
        # Apply the operation on the chunk
        op = chunk.max(keepdims=False)
        # Update the result
        if self.result is None:
            self.result = PDict(op)
        else:
            for k, v in self.result.items():
                self.result[k] = np.fmax(op[k], v)
        # Return the next state and number of steps handled
        if table_slot.has_buffered():
            next_state = Module.state_ready
        else:
            next_state = Module.state_blocked
        return self._return_run_step(next_state, steps)
```

```python
class ModuleState(IntEnum):
    state_created = 0
    state_ready = 1
    state_running = 2
    state_blocked = 3
    state_suspended = 4
    state_zombie = 5
    state_terminated = 6
    state_invalid = 7
```

- The module has a state
- Typically **ready**, **blocked**, or **terminated**
  - When **ready**, `Module.is_ready()` returns True immediately
  - When **blocked**, it checks all the input slots. If there are changes, it returns True
  - **Otherwise** it returns False
- At the end, two values are returned from `run_step`:
  - The next state
  - The number of steps actually run, used by the Time Predictor to adjust
- Here, the next step depends on the delta of the input slot
  - If there are still values left in the delta, it returns "ready"
  - Otherwise, "blocked"
- The module will terminate when its inputs are terminated and it is blocked

Improve with decorators and tailored methods

```python
import numpy as np
from typing import Any
from progressivis import (
    Module, ReturnRunStep, PTable, PDict,
    def_input, def_output
)
from progressivis.core.utils import indices_len, fix_loc


@def_input("table", PTable, doc="...")
@def_output("result", PDict, doc=("Max value of each column"))
class SimpleMax(Module):
    def __init__(self, **kwds: Any) -> None:
        super().__init__(**kwds)
        self.default_step_size = 10000

    def reset(self) -> None:
        if self.result is not None:
            self.result.fill(-np.inf)

    def run_step(
        self, run_number: int, step_size: int, quantum: float
    ) -> ReturnRunStep:
        table_slot = self.get_input_slot("table")
        if table_slot.updated.any() or table_slot.deleted.any():
            table_slot.reset()
            table_slot.update(run_number)
            self.reset()
        indices = table_slot.created.next(length=step_size)
        steps = indices_len(indices)
        chunk = table_slot.data().loc[fix_loc(indices)]
        op = chunk.max(keepdims=False)
        if self.result is None:
            self.result = PDict(op)
        else:
            for k, v in self.result.items():
                self.result[k] = np.fmax(op[k], v)
        if table_slot.has_buffered():
            next_state = Module.state_ready
        else:
            next_state = Module.state_blocked
        return self._return_run_step(next_state, steps)
```

```python
import numpy as np
from typing import Any, Sequence
from progressivis import (
    Module, ReturnRunStep, PTable, PDict,
    def_input, def_output, process_slot, run_if_any
)
from progressivis.core.utils import indices_len, fix_loc


@def_input("table", PTable, hint_type=Sequence[str], doc="...")
@def_output("result", PDict, doc=("Max value of each column"))
class Max(Module):
    def __init__(self, **kwds: Any) -> None:
        super().__init__(**kwds)
        self.default_step_size = 10000

    def reset(self) -> None:
        if self.result is not None:
            self.result.fill(-np.inf)

    @process_slot("table", reset_cb="reset")
    @run_if_any
    def run_step(
        self, run_number: int, step_size: int, quantum: float
    ) -> ReturnRunStep:
        assert self.context is not None
        with self.context as ctx:
            indices = ctx.table.created.next(length=step_size)
            steps = indices_len(indices)
            op = self.filter_slot_columns(
                ctx.table,
                fix_loc(indices)
            ).max(keepdims=False)
            if self.result is None:
                self.result = PDict(op)
            else:
                for k, v in self.result.items():
                    self.result[k] = np.fmax(op[k], v)
            return self._return_run_step(
                self.next_state(ctx.table),
                steps
            )
```

Monitoring Module Quality

- Some modules can provide a meaningful quality
  - Higher value means better quality
- Some cannot, only progress
- What is a quality?
  - Depends on the module semantics
- For mean, can be the CI?
- For Max, we use stability instead of a statistical measures
  - |prev_value - cur_value|
- Particular algorithms come with their own quality measures
  - We use them, or others (stability) for speed

- What would be the quality for max?
- The max value itself!
- If it grows, the quality improves
- When it plateau, the quality stabilizes, possibly converges

```python
def get_quality(self) -> Dict[str, float] | None:  # v3
    if self.result is None:
        return None
    for key in self.result:
        try:
            # The quality is simply the value.
            # It can only grow when improving, and
            # should stabilize eventually.
            self.quality["max_" + key] = float(self.result[key])
        except ValueError:
            pass
    return self.quality
```

Summary: Creating a Module

- The bulk of ProgressiVis processing lies in the `Module.run_step` method
- First, it checks what changed
- If it cannot deal with deleted or updated items, it resets the module
- It then computes its partial result taking into account the **number of steps** given
- It then updates the result slots, and
- Returns two values: the **next state** and **number of items processed**
- Any exception will terminate the module.

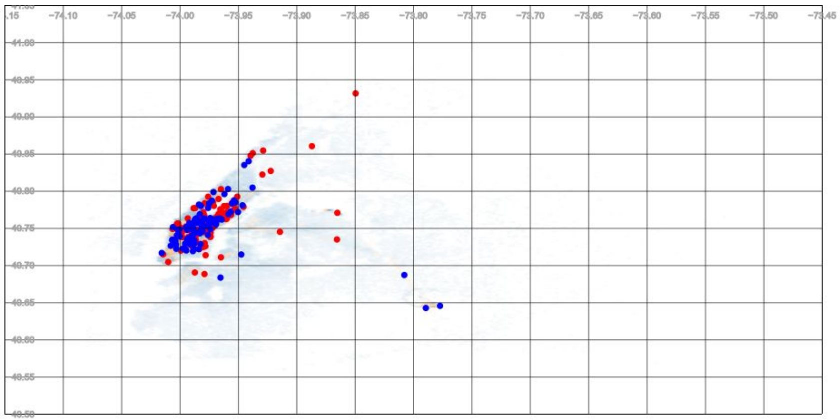
- Using decorators shorten the code and makes it more readable
- Several methods from the Module class are meant to simplify the coding
- When implementing a module, main issues are
  - Performance
  - Avoiding to reset the module if possible
  - Stability sometimes
- **Quiz**: can you think of a way to avoid the Max module from always resetting on delete/update?

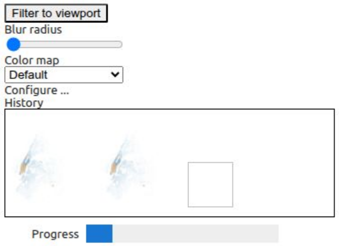
Progressive Visualizations

- Using existing visualizations
  - Heatmap
  - Multiclass Scatterplot
  - Using Vega Lite
- Creating a new visualization
  - The module
  - The Widget
    - The back end in Python
    - The front end in JavaScript
    - Sending big data (don't but…)
- Adding Interaction

Multiclass Scatterplot

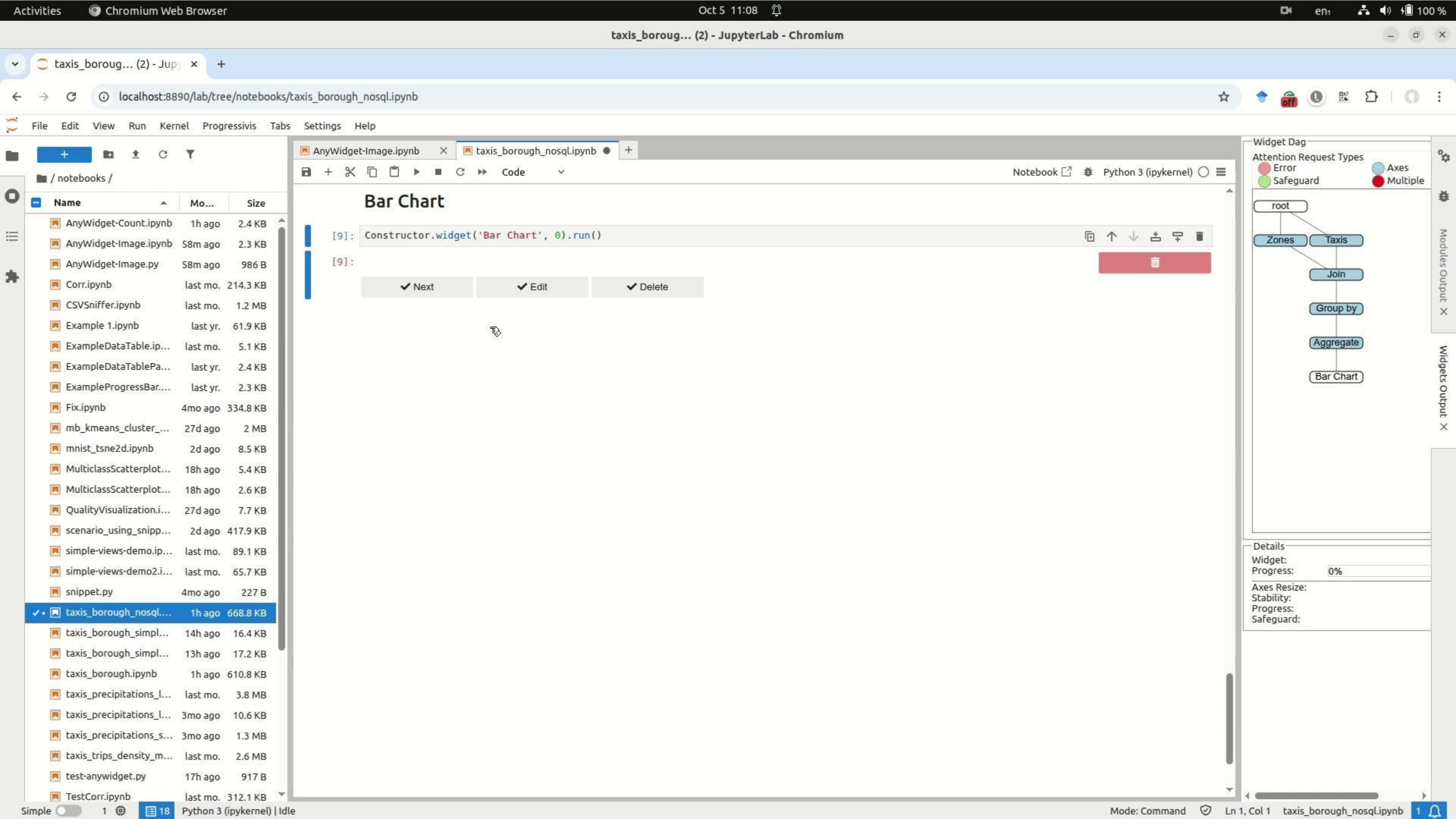
```
PREFIX = 'https://www.aviz.fr/nyc-taxi/'
SUFFIX = '.bz2'
URLS = [
    PREFIX+'yellow_tripdata_2015-01.csv'+SUFFIX,
    PREFIX+'yellow_tripdata_2015-02.csv'+SUFFIX,
    PREFIX+'yellow_tripdata_2015-03.csv'+SUFFIX,
    PREFIX+'yellow_tripdata_2015-04.csv'+SUFFIX,
    PREFIX+'yellow_tripdata_2015-05.csv'+SUFFIX,
    PREFIX+'yellow_tripdata_2015-06.csv'+SUFFIX,
]

FILENAMES = pd.DataFrame({'filename': URLS})
CST = Constant(PTable('filenames', data=FILENAMES), scheduler=s)
CSV = CSVLoader(usecols=['pickup_longitude', 'pickup_latitude',
                         'dropoff_longitude', 'dropoff_latitude'],
                filter_=_filter, scheduler=s)

CSV.input.filenames = CST.output[0]
PR = Every(scheduler=s, proc=_quiet)
PR.input.df = CSV.output[0]

MULTICLASS = MCScatterPlot(
    scheduler=s,
    classes=[
        ('pickup', 'pickup_longitude', 'pickup_latitude'),
        ('dropoff', 'dropoff_longitude', 'dropoff_latitude')
    ],
    approximate=True)
MULTICLASS.create_dependent_modules(CSV, 'result')
sc = Scatterplot()
```





The AnyVega Widget

- Uses Vega-Lite and its "streaming" mode
- Takes a table in input, with a templated Vega-Lite configuration
- Visualizes the whole table each time it is updated
    - WARNING: don't send a large table
- Useful for already aggregated data
- For example, bar chart of NYC taxis per Borough


Activities
Chromium Web Browser
Oct 5 11:08
taxis_boroug... (2) - JupyterLab - Chromium
localhost:8890/lab/tree/notebooks/taxis_borough_nosql.ipynb
File Edit View Run Kernel Progressivis Tabs Settings Help
/ notebooks /
Name Mo... Size
AnyWidget-Count.ipynb 1h ago 2.4 KB
AnyWidget-Image.ipynb 58m ago 2.3 KB
AnyWidget-Image.py 58m ago 986 B
Corr.ipynb last mo. 214.3 KB
CSVSniffer.ipynb last mo. 1.2 MB
Example 1.ipynb last yr. 61.9 KB
ExampleDataTable.ip... last mo. 5.1 KB
ExampleDataTablePa... last yr. 2.4 KB
ExampleProgressBar.... last yr. 2.3 KB
Fix.ipynb 4mo ago 334.8 KB
mb_kmeans_cluster_... 27d ago 2 MB
mnist_tsne2d.ipynb 2d ago 8.5 KB
MulticlassScatterplot... 18h ago 5.4 KB
MulticlassScatterplot... 18h ago 2.6 KB
QualityVisualization.i... 27d ago 7.7 KB
scenario_using_snipp... 2d ago 417.9 KB
simple-views-demo.ip... last mo. 89.1 KB
simple-views-demo2.i... last mo. 65.7 KB
snippet.py 4mo ago 227 B
taxis_borough_nosql.... 1h ago 668.8 KB
taxis_borough_simpl... 14h ago 16.4 KB
taxis_borough_simpl... 13h ago 17.2 KB
taxis_borough.ipynb 1h ago 610.8 KB
taxis_precipitations_l... last mo. 3.8 MB
taxis_precipitations_l... 3mo ago 10.6 KB
taxis_precipitations_s... 3mo ago 1.3 MB
taxis_trips_density_m... last mo. 2.6 MB
test-anywidget.py 17h ago 917 B
TestCorr.ipynb last mo. 312.1 KB
AnyWidget-Image.ipynb
taxis_borough_nosql.ipynb
Code
Notebook
Python 3 (ipykernel)
Bar Chart
[9]: Constructor.widget('Bar Chart', 0).run()
[9]:
Next
Edit
Delete
Widget Dag
Attention Request Types
Error
Axes
Safeguard
Multiple
root
Zones
Taxis
Join
Group by
Aggregate
Bar Chart
Details
Widget:
Progress: 0%
Axes Resize:
Stability:
Progress:
Safeguard:
Modules Output
Widgets Output
Simple
Python 3 (ipykernel) | Idle
Mode: Command
Ln 1, Col 1
taxis_borough_nosql.ipynb

Implementing a Progressive Visualization in Jupyter Lab

- Jupyter Lab connects a Python interpreter to a web browser using a shared bidirectional data stream
- There is a standard architecture to design widgets
  - One python class, two JS classes
  - Sharing data attributes automatically using the "traitlets" python package
- On top, ProgressiVis may need to implement partial updates
  - It uses a standard "send event" mechanism of jupyter widgets
- The initial mechanics is a bit heavy, but once in place, it's just JS!
- Try starting with AnyWidget: https://anywidget.dev/, launching it with

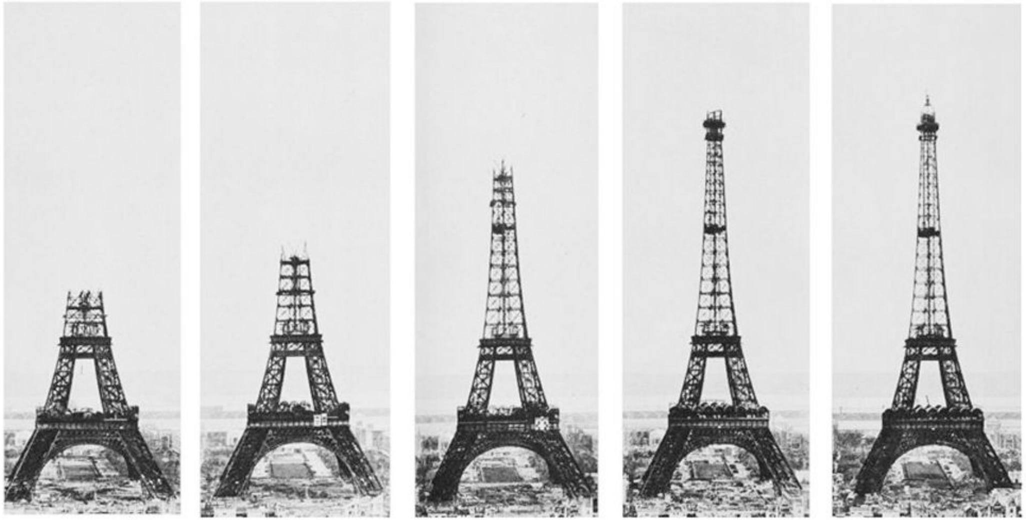
```
ANYWIDGET_HMR=1 jupyter lab
```

Implementing a Simple Image Widget

```python
import anywidget
import traitlets


class ImageWidget(anywidget.AnyWidget):
    _esm = """
    function render({ model, el }) {
      let img = document.createElement("img");
      img.src = model.get("url");
      model.on("change:url", () => {
        img.src = model.get("url");
      });
      el.classList.add("image-widget");
      el.appendChild(img);
    }
    export default { render };
    """
    _css = """
    """
    url = traitlets.Unicode("").tag(sync=True)
```

Changes consist in replacing the image

```python
ImageWidget(url="https://www.aviz.fr/wiki/uploads/Progressive/construction_tour_eiffel.jpg")
```



Improving the Simple Image Widget

```python
import anywidget
import traitlets


class ImageWidget(anywidget.AnyWidget):
    _esm = """
    function render({ model, el }) {
      let img = document.createElement("img");
      img.src = model.get("url");
      img.addEventListener("wheel", function (event) {
        event.preventDefault();
        model.set("scale", model.get("scale") + event.deltaY * -0.01);
        model.save_changes();
      });
      model.on("change:url", () => {
        img.src = model.get("url");
      });
      el.classList.add("image-widget");
      el.appendChild(img);
    }
    export default { render };
    """
    _css = """
    """
    url = traitlets.Unicode("").tag(sync=True)
    scale = traitlets.Float(1.0).tag(sync=True)
```

- The widget can also manage events from the browser to Python, with event listeners
- The Python side can add a callback to changes in the variable "scale"
- For more sophisticated operations, the widget can manage custom events by adding:

```
this.model.on("msg:custom", (ev) => {
  if (ev.type != "update") {
    return;
  }
  do_something(ev.timestamp, ev.measures);
});
```

Summary

- ProgressiVis provides a few visualizations techniques
  - Heatmap, MultiClass Density Map, AnyVega
- More can be built using IPyWidgets
  - With various levels of complexity
- More research would be useful to have a progressive Grammar of Graphics
- Meanwhile, ad-hoc visualization techniques can be implemented with reasonable efforts

Implementing a Data Loader

Implementing a Loader Module: Simple Case

- A loader module is always ready until it terminates
- CSV Loaders can take a list of file names as input but, usually, they take an URL as argument, sometimes with wildcards e.g., foo.com/*.csv
- They need to split the input in chunks and read them in as a PTable
- They also need to provide an information about their "progress"

Small CSV Loader

```python
@def_output("result", PTable, doc=RESULT_DOC)
class SmallCSVLoaderV1(Module):
    def __init__(self, filepath_or_buffer: Any, **kwds: Any) -> None:
        if "index_col" in kwds:
            raise ProgressiveError("'index_col' parameter is not supported")
        super().__init__(**kwds)
        self.default_step_size = 1000
        chunksize_ = kwds.get("chunksize")
        if isinstance(chunksize_, int):  # initial guess
            self.default_step_size = chunksize_
        if chunksize_ is None:
            kwds["chunksize"] = self.default_step_size
        else:
            kwds.setdefault("chunksize", self.default_step_size)
        # Filter out the module keywords from the csv loader keywords
        csv_kwds: Dict[str, Any] = filter_kwds(kwds, pd.read_csv)
        self.parser = pd.read_csv(filepath_or_buffer, **csv_kwds)
        self._rows_read = 0
        self.result: PTable | None  # to help mypy

    def rows_read(self) -> int:
        return self._rows_read

    def is_data_input(self) -> bool:
        return True
```

```python
def run_step(
    self, run_number: int, step_size: int, quantum: float
) -> ReturnRunStep:
    if step_size == 0:  # bug
        return self._return_run_step(self.state_ready, steps_run=0)
    try:
        df = self.parser.read(step_size)
    except StopIteration:
        return self._return_run_step(self.state_zombie, steps_run=0)
    except ValueError:
        raise
    creates = len(df)
    if creates == 0:  # should not happen
        logger.error("Received 0 elements")
        return self._return_run_step(self.state_zombie, steps_run=0)
    self._rows_read += creates
    force_valid_id_columns(df)  # fix column names
    if self.result is None:  # create the PTable
        self.result = PTable(
            name=self.generate_table_name("table"),
            dshape=dshape_from_dataframe(df),  # infer types
            data=df,
            create=True
        )
    else:
        self.result.append(df)
    return self._return_run_step(self.state_ready, steps_run=creates)
```

`5-loader-creation/small_csv_loader-v1.py`

Small CSV Loader (2)

```python
@def_output("result", PTable, doc=RESULT_DOC)
class SmallCSVLoaderV1(Module):
    def __init__(self, filepath_or_buffer: Any, **kwds: Any) -> None:
        if "index_col" in kwds:
            raise ProgressiveError("'index_col' parameter is not supported")
        super().__init__(**kwds)
        self.default_step_size = 1000
        chunksize_ = kwds.get("chunksize")
        if isinstance(chunksize_, int):  # initial guess
            self.default_step_size = chunksize_
        if chunksize_ is None:
            kwds["chunksize"] = self.default_step_size
        else:
            kwds.setdefault("chunksize", self.default_step_size)
        # Filter out the module keywords from the csv loader keywords
        csv_kwds: Dict[str, Any] = filter_kwds(kwds, pd.read_csv)
        self.parser = pd.read_csv(filepath_or_buffer, **csv_kwds)
        self._rows_read = 0
        self.result: PTable | None  # to help mypy

    def rows_read(self) -> int:
        return self._rows_read

    def is_data_input(self) -> bool:
        return True
```

- Use Pandas to read CSV efficiently
- Pandas.read_csv creates a iterative parser with the keyword "chunksize"
- We allow all the keywords of Pandas.csv_read except "index_col" and "nrows"
- The scheduler wants to know if at least one module provides data without an input with Module.is_data_input() so it can stop if everything is blocked

Small CSV Loader (3)

- Nothing fancy in run_step, no input slot
- The column names should be valid in the DataFrame -> force_valid_columns
- Our types use the "DataShape" syntax: {col-name: col-type, …}
- {name: string, age: int, height: float}
- parser.read raises a StopIteration exception when it is done

```python
def run_step(
    self, run_number: int, step_size: int, quantum: float
) -> ReturnRunStep:
    if step_size == 0:  # bug
        return self._return_run_step(self.state_ready, steps_run=0)
    try:
        df = self.parser.read(step_size)
    except StopIteration:
        return self._return_run_step(self.state_zombie, steps_run=0)
    except ValueError:
        raise
    creates = len(df)
    if creates == 0:  # should not happen
        logger.error("Received 0 elements")
        return self._return_run_step(self.state_zombie, steps_run=0)
    self._rows_read += creates
    force_valid_id_columns(df)  # fix column names
    if self.result is None:  # create the PTable
        self.result = PTable(
            name=self.generate_table_name("table"),
            dshape=dshape_from_dataframe(df),  # infer types
            data=df,
            create=True
        )
    else:
        self.result.append(df)
    return self._return_run_step(self.state_ready, steps_run=creates)
```

5-loader-creation/small_csv_loader-v1.py

Small CSV Loader (4)

- To monitor the progressive loading, we need a measure of the progress, returned as a pair (current_pos, max_pos) by `Module.get_progress()`
- This is tricky to get, we need to open the engine of `Pandas.read_csv`
- We can read the raw stream length L but it can be compressed
- We can read chunks and monitor the table length N -> current_pos
- We can read the position in the raw stream (# of bytes read) M
- We assume M/L = current_pos/max_pos

```python
def get_progress(self) -> Tuple[int, int]:
    input_size = self.parser._input._input_size
    if input_size == 0:
        return (0, 0)
    pos = self.parser._input._stream.tell()
    length = len(self.result)
    estimated_row_size = pos / length
    estimated_size = int(input_size / estimated_row_size)
    return (length, estimated_size)
```

Implementing a Loader Module: Efficient Case

- A simple loader runs for a short time (quantum) once in a while, which limits its throughput
- Use a thread to load, especially with Python >= 3.13
  - Load iteratively chunks slightly faster than the module quantum to make sure at least 1 is loaded when the module calls `run_step`
  - Add the chunks in a thread-safe Python `Queue` using `Queue.put()`
  - In `run_step`, pop all the chunks and add them to the output Table
- Use the `Module.on_ending` callback to terminate the thread if the module is deleted before loading is completed

Data Loader Summary

- Reading data progressively is not technically hard usually
- But some formats are not always appropriate
- Apache Arrow and Parquet files could be perfect or unusable
  - The are organized in columns and can be chunked or not
  - If not chunked, the whole table needs to be read, possibly blocking
- The main issue is to convince the world to split their files in chunks
- And provide a shuffled version, for better convergence in progressive algorithms

ProgressiVis Tutorial Summary

- ProgressiVis provides all the mechanisms to run progressive programs, monitoring the progress, and the quality
- It can be used as basis for development of progressive solutions:
  - Visualizations, Data structures, Loaders, Algorithms
- It could save you time and efforts to create progressive solutions
  - Scalability for visualization in terms of data size and download time
  - Scalability for interactive analysis including machine learning
  - Instant data, no need to wait for data and visualization to arrive
  - Greener computing, processing only the required data to get a result
  - Algorithmic transparency, monitoring an algorithm while it processes data
- Contact us for feedback and clarifications, we want to help the community!

ProgressiVis Needs You!

- We need to enrich ProgressiVis to become a complete environment
- There are many possible improvements:
  - Performance, it needs optimizations
    - With Python 3.14, using threads efficiently becomes possible
  - More data structures, such as graphs, trees, and 3D structures
  - More loaders and progressive databases connectors
  - More Visualizations, 2D and 3D
    - Progressive Grammar of Graphics?
  - More algorithms, possibly using Data Sketches, and Online Machine Learning
- Contact us if you want to use ProgressiVis for your research or applications
- Jean-Daniel.Fekete@inria.fr Christian.Poli@inria.fr