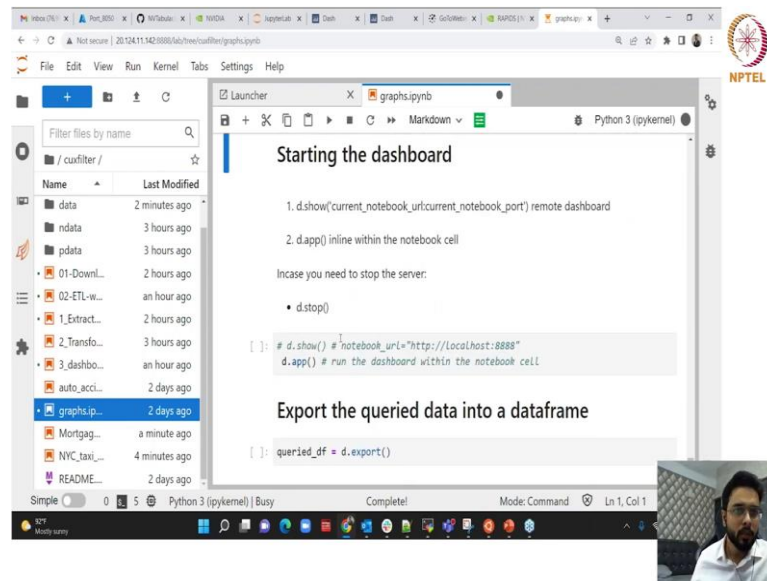
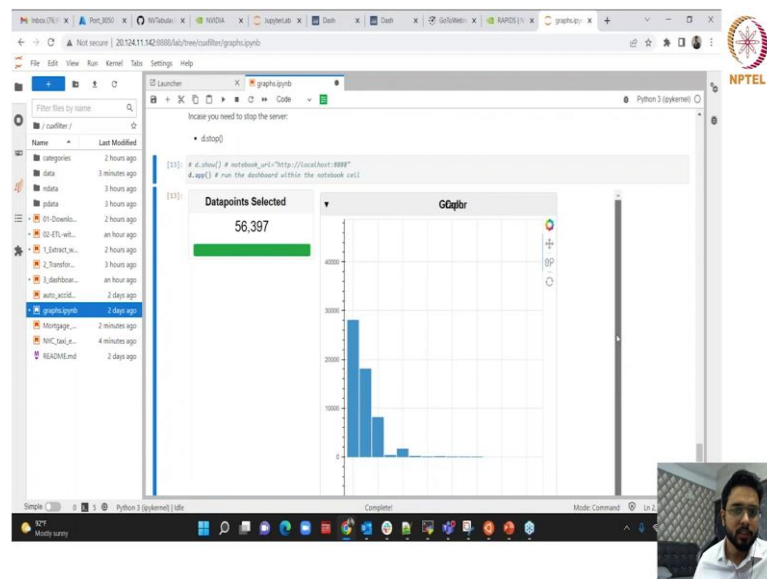


(Refer Slide Time: 01:24)



So, for this particular chart we do not need any API key. So, this should work ok.

(Refer Slide Time: 01:56)



We can see some chart here. So, let me try to show you in full screen, yeah. So, this is how it looks the data point selected, then the graph this is a simple bar chart which is being plotted. So, just ignore this one we do not need to see the preview.

(Refer Slide Time: 02:24)

The screenshot shows a JupyterLab interface with a Python 3 (ipykernel) environment. The main area displays a log of iterations with various performance metrics. Below the log, a table is shown with columns 'x', 'y', 'SYMBOL', and 'Color'. The table contains five rows of data:

x	y	SYMBOL	Color
0	1078124089	01570704	1807 5
1	4098302441	-191647502	1802 5
2	5105360449	300501020	9996 1
3	5866631348	273520027	347 1
4	-2348175208	626461823	1800 1

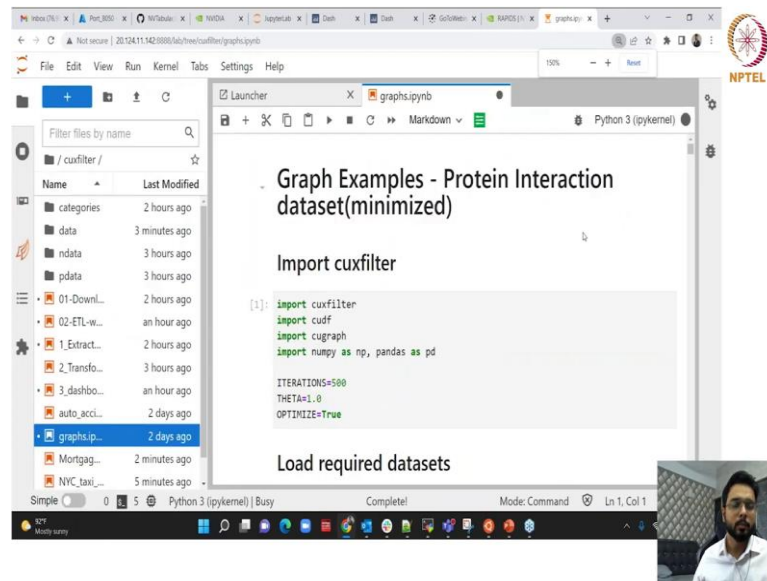
Below the table, there is a section titled 'Define charts' with a code cell containing: `cat_off = cuffilter.DataFrom.Load_graph(nodes_1, edges)`. A small video inset in the bottom right corner shows a person speaking.

(Refer Slide Time: 02:27)

The screenshot shows a JupyterLab interface with a Python 3 (ipykernel) environment. The main area displays a log of iterations with various performance metrics. The log continues from the previous slide, showing iterations 458 through 478. A small video inset in the bottom right corner shows a person speaking.

So, just try to show what we did here.

(Refer Slide Time: 02:31)



Graph Examples - Protein Interaction dataset(minimized)

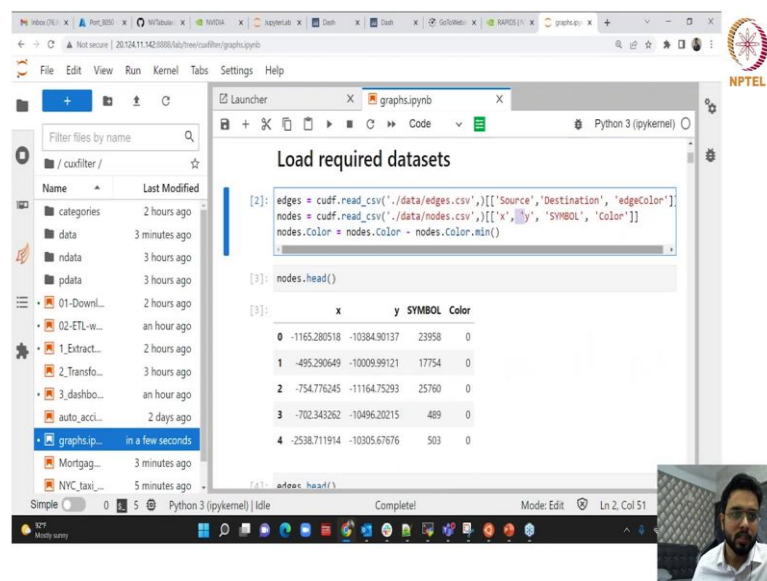
Import cuxfilter

```
[1]: import cuxfilter
import cudf
import cugraph
import numpy as np, pandas as pd

ITERATIONS=500
THETA=1.0
OPTIMIZE=True
```

Load required datasets

(Refer Slide Time: 02:35)



Load required datasets

```
[2]: edges = cudf.read_csv('./data/edges.csv')[['Source', 'Destination', 'edgeColor']]
nodes = cudf.read_csv('./data/nodes.csv')[['x', 'y', 'SYMBOL', 'Color']]
nodes.Color = nodes.Color.min()
```

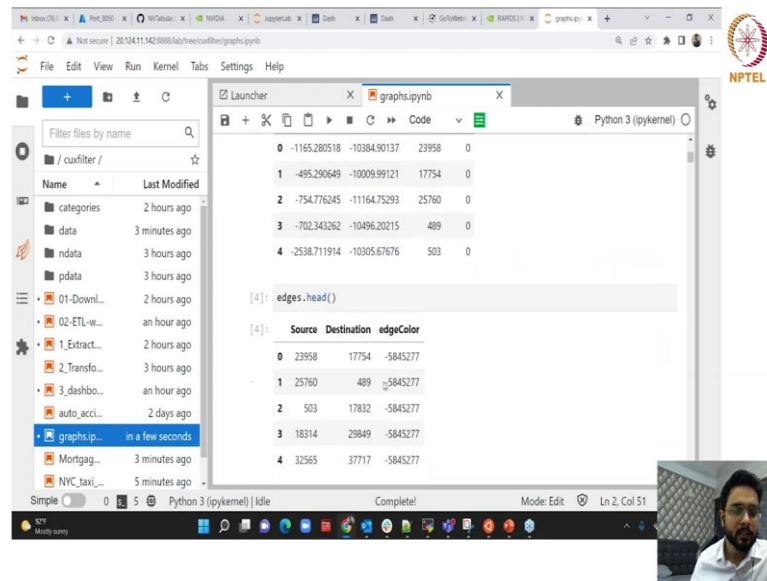
```
[3]: nodes.head()
```

```
[3]:
```

	x	y	SYMBOL	Color
0	-1165.280518	-10384.90137	23958	0
1	-495.290649	-10009.99121	17754	0
2	-754.776245	-11164.75293	25760	0
3	-702.343262	-10496.20215	489	0
4	-2538.711914	-10305.67676	503	0

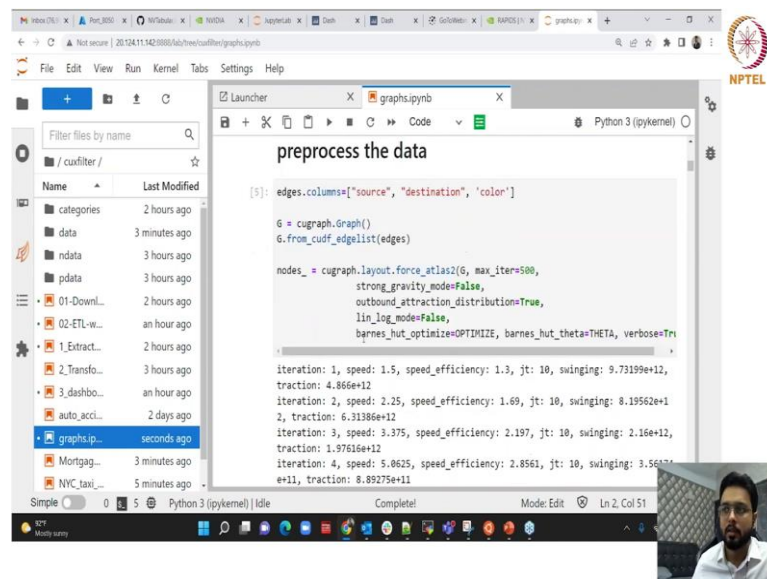
So, it is a protein interaction dataset which is the basically biological data, we have graphical format of data where we have edges, source, destination, edge color and then x, y symbol color as nodes.

(Refer Slide Time: 02:47)



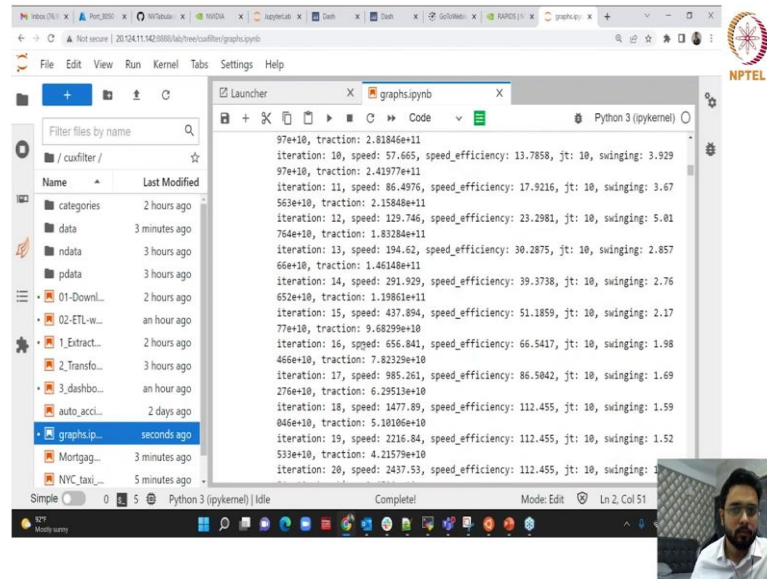
So, this is the dataset we have in hand.

(Refer Slide Time: 02:48)



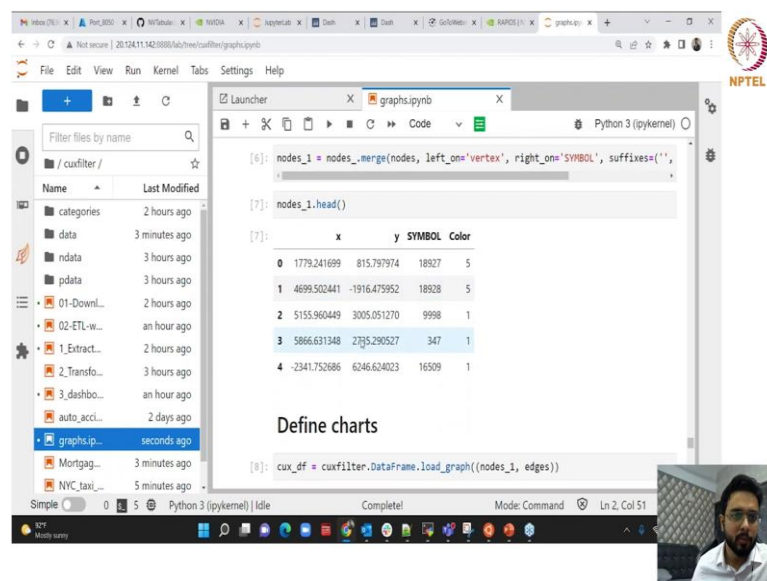
Then we pre process the data using cugraph, we do some basic processing of the data.

(Refer Slide Time: 02:57)



```
97e+10, traction: 2.81846e+11
iteration: 10, speed: 57.665, speed_efficiency: 13.7858, jt: 10, swinging: 3.929
97e+10, traction: 2.41977e+11
iteration: 11, speed: 86.4976, speed_efficiency: 17.9216, jt: 10, swinging: 3.67
563e+10, traction: 2.15848e+11
iteration: 12, speed: 129.746, speed_efficiency: 23.2981, jt: 10, swinging: 5.81
764e+10, traction: 1.83284e+11
iteration: 13, speed: 194.62, speed_efficiency: 30.2875, jt: 10, swinging: 2.857
66e+10, traction: 1.46148e+11
iteration: 14, speed: 291.929, speed_efficiency: 39.3738, jt: 10, swinging: 2.76
652e+10, traction: 1.19861e+11
iteration: 15, speed: 437.894, speed_efficiency: 51.1859, jt: 10, swinging: 2.17
77e+10, traction: 9.68299e+10
iteration: 16, speed: 656.841, speed_efficiency: 66.5417, jt: 10, swinging: 1.98
466e+10, traction: 7.82329e+10
iteration: 17, speed: 985.261, speed_efficiency: 86.5842, jt: 10, swinging: 1.69
276e+10, traction: 6.29513e+10
iteration: 18, speed: 1477.89, speed_efficiency: 112.455, jt: 10, swinging: 1.59
046e+10, traction: 5.10106e+10
iteration: 19, speed: 2216.84, speed_efficiency: 112.455, jt: 10, swinging: 1.52
533e+10, traction: 4.21579e+10
iteration: 20, speed: 2437.53, speed_efficiency: 112.455, jt: 10, swinging: 1.52
```

(Refer Slide Time: 03:05)



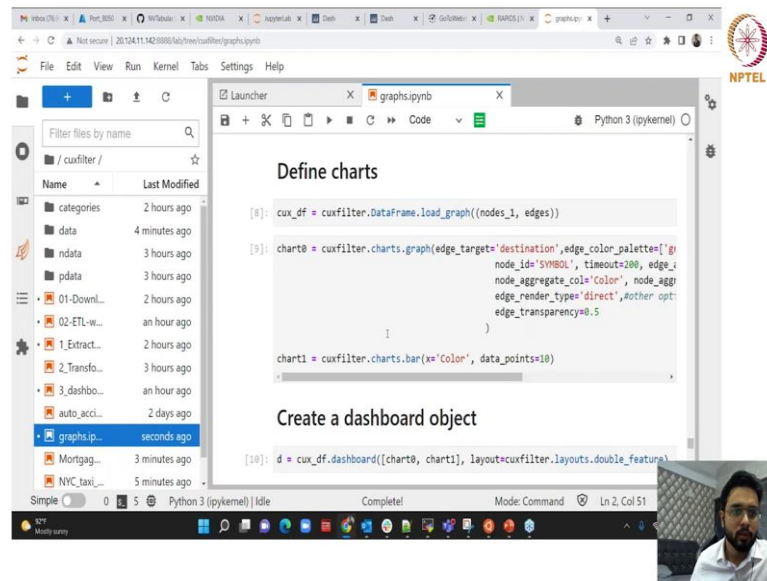
```
[6]: nodes_1 = nodes_merge(nodes, left_on='vertex', right_on='SYMBOL', suffixes=('',
[7]: nodes_1.head()
[7]:
```

	x	y	SYMBOL	Color
0	1779.241699	815.797974	18927	5
1	4699.502441	-1916.475952	18928	5
2	5155.960449	3005.051270	9998	1
3	5866.631348	2735.290527	347	1
4	-2341.752686	6246.624023	16509	1

```
Define charts
[8]: cux_df = cuxfilter.DataFrame.load_graph((nodes_1, edges))
```

And then after doing all the processing. So, it is not necessary that you use cugraph. So, it is just one of the examples, where cugraph is used. You can use normal cuDF or DASK to do that, then after processing we have this format x, y, symbol and color.

(Refer Slide Time: 03:13)



The screenshot shows a Jupyter Notebook window with the following code:

```
Define charts

[8]: cux_df = cuxfilter.DataFrame.load_graph((nodes_1, edges))

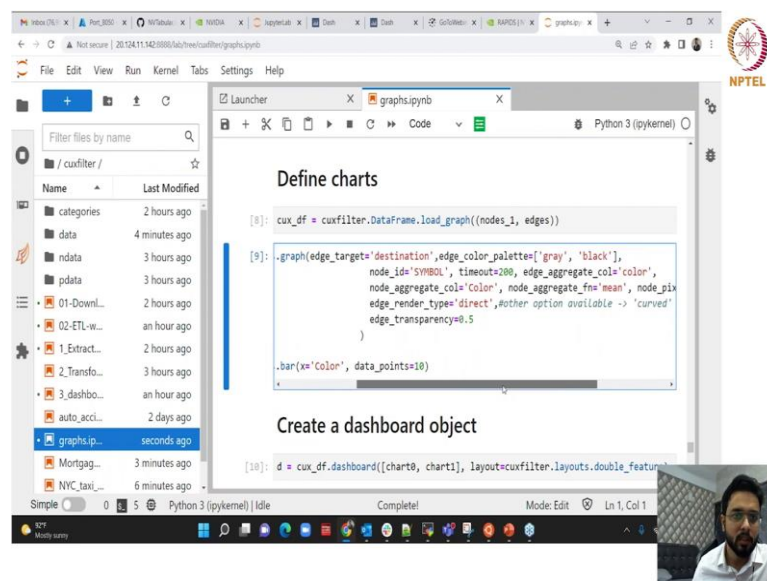
[9]: chart0 = cuxfilter.charts.graph(edge_target='destination', edge_color_palette=['gray', 'black'],
                                     node_id='SYMBOL', timeout=200, edge_aggregate_col='color', node_aggregate_col='color',
                                     edge_renderer_type='direct', other_option='curved', edge_transparency=0.5)

chart1 = cuxfilter.charts.bar(x='color', data_points=10)

Create a dashboard object

[10]: d = cux_df.dashboard([chart0, chart1], layout=cuxfilter.layouts.double_feature)
```

(Refer Slide Time: 03:20)



The screenshot shows the same Jupyter Notebook window as above, but with a blue highlight around the graph chart definition in cell [9]:

```
Define charts

[8]: cux_df = cuxfilter.DataFrame.load_graph((nodes_1, edges))

[9]: .graph(edge_target='destination', edge_color_palette=['gray', 'black'],
            node_id='SYMBOL', timeout=200, edge_aggregate_col='color',
            node_aggregate_col='color', node_aggregate_fn='mean', node_pix
            edge_renderer_type='direct', other_option available -> 'curved'
            edge_transparency=0.5)

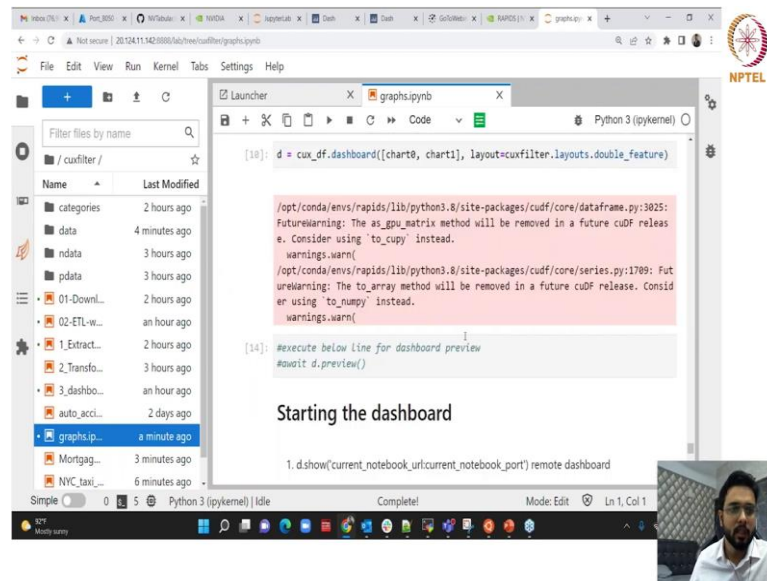
.bar(x='color', data_points=10)

Create a dashboard object

[10]: d = cux_df.dashboard([chart0, chart1], layout=cuxfilter.layouts.double_feature)
```

So, using that we plot a data using graph chart, destination is the edge target, edge color is grey, black node id SYMBOL.

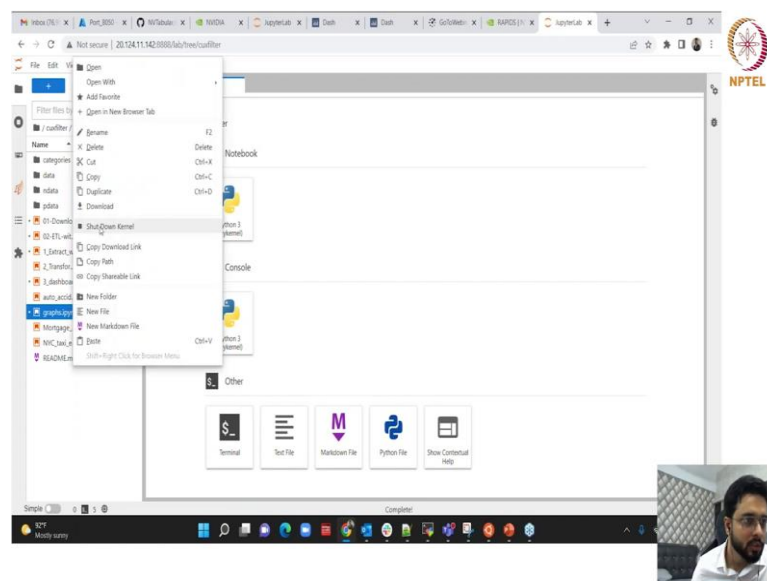
(Refer Slide Time: 03:32)



So, all these details you give to put the chart, again all this parameter values or parameter description you can refer the cuXfilter documentation to understand it in detail and then you can `d.app()` to run the dashboard within the notebook cell. So, there are 56397 data points and we have this graph created for that.

So, here you see this is the x axis, y axis respectively and I will just show one more if possible.

(Refer Slide Time: 04:06)



So, I will just close this one to all other.

(Refer Slide Time: 05:14)

```
1: 'Sunday',
2: 'Monday',
3: 'Tuesday',
4: 'Wednesday',
5: 'Thursday',
6: 'Friday',
7: 'Saturday',
8: 'Unknown'
}

gtc_demo_red_blue_palette = [ "#3182bd", "#6baed6", "#7b6d89", "#e26738", "#ff9966" ]

Uncomment the below lines and replace MAPBOX_TOKEN with mapbox
token string if you want to use mapbox map-tiles. Can be created for free
here -https://www.mapbox.com/help/define-access-token/

[7]: #from cuxfilter.assets.custom_tiles import get_provider, Vendors
#tile_provider = get_provider(Vendors.MAPBOX_LIGHT, access_token=MAPBOX_TOKEN)
```

Where we have label Sunday to Saturday.

(Refer Slide Time: 05:21)

```
[7]: chart1 = cuxfilter.charts.scatter(x='dropoff_x', y='dropoff_y', aggregate_col='Day_of_Week',
#tile_provider = get_provider(Vendors.MAPBOX_LIGHT, access_token=MAPBOX_TOKEN)
tile_provider='CartoLight',
color_palette=gtc_demo_red_blue_palette)

chart2 = cuxfilter.charts.bar("YEAR")

chart3 = cuxfilter.charts.multi_select('DAY_WEEK', label_map=label_map)

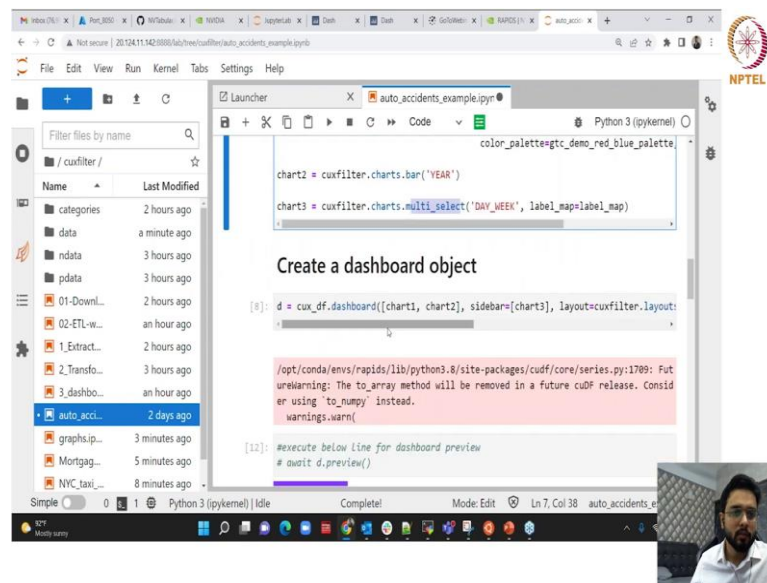
Create a dashboard object

[8]: d = cuxfilter.dashboard([chart1, chart2], sidebar=[chart3], layout=cuxfilter.layout...)

/opt/conda/envs/rapids/lib/python3.8/site-packages/cudf/core/series.py:1709: FutureWarning: The to_array method will be removed in a future cuDF release. Consider using to_numpy instead.
warnings.warn(
```

And then pilot, then there are three charts we have scatter chart, bar chart and then we have a multi select. So, we can we are now this time we are not creating only one chart, we are creating two charts and one multi select filter.

(Refer Slide Time: 05:36)



```
color_palette=gtc_demo_red_blue_palette

chart2 = cuxfilter.charts.bar("YEAR")
chart3 = cuxfilter.charts.multi_select("DAY_WEEK", label_map=label_map)

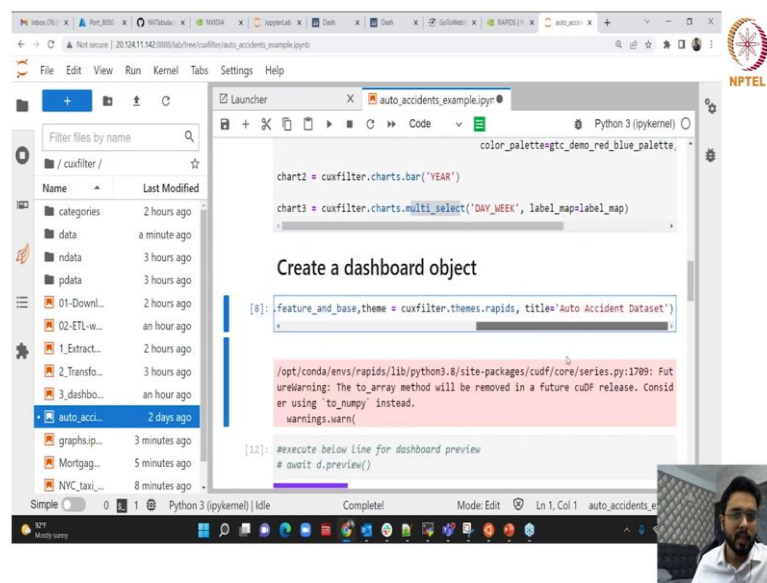
Create a dashboard object

[8]: d = cux_df.dashboard([chart1, chart2], sidebar=[chart3], layout=cuxfilter.layout_

/opt/conda/envs/rapids/lib/python3.8/site-packages/cudf/core/series.py:1789: FutureWarning: The to_array method will be removed in a future cuDF release. Consider using "to_numpy" instead.
warnings.warn(

[12]: #execute below line for dashboard preview
# await d.preview()
```

(Refer Slide Time: 05:38)



```
color_palette=gtc_demo_red_blue_palette

chart2 = cuxfilter.charts.bar("YEAR")
chart3 = cuxfilter.charts.multi_select("DAY_WEEK", label_map=label_map)

Create a dashboard object

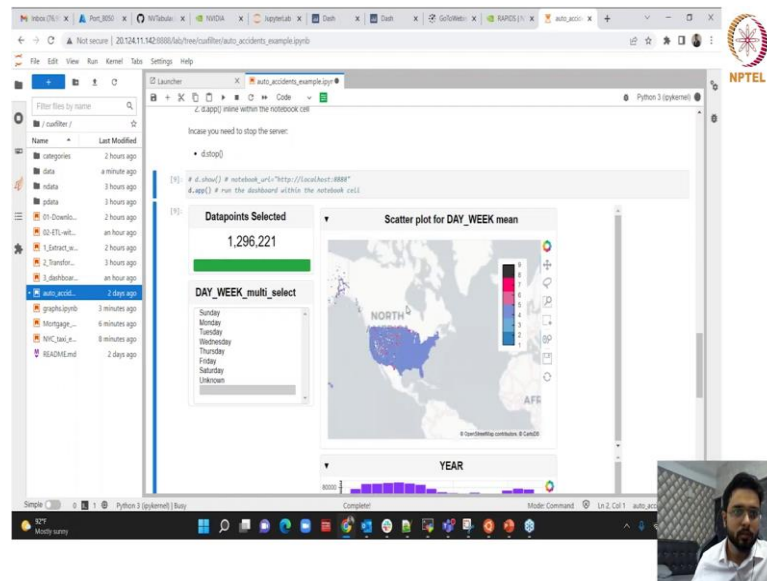
[8]: feature_and_base_theme = cuxfilter.themes.rapids, title='Auto Accident Dataset')

/opt/conda/envs/rapids/lib/python3.8/site-packages/cudf/core/series.py:1789: FutureWarning: The to_array method will be removed in a future cuDF release. Consider using "to_numpy" instead.
warnings.warn(

[12]: #execute below line for dashboard preview
# await d.preview()
```

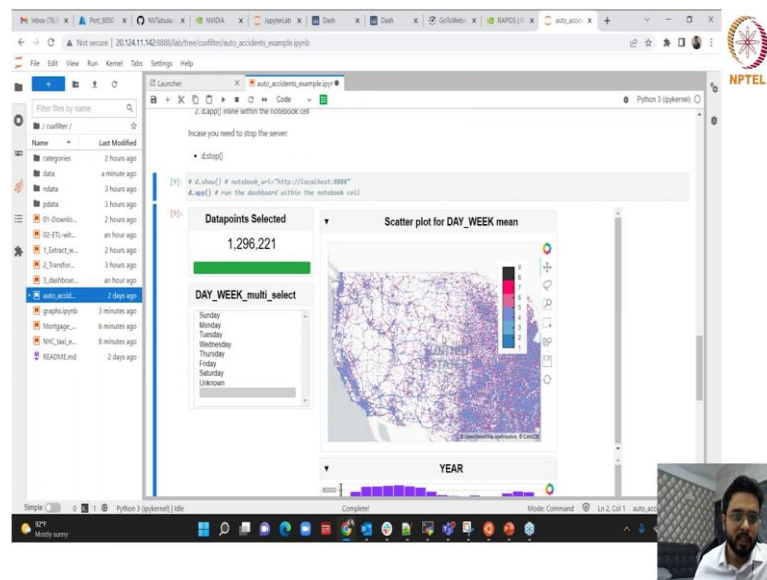
So, and then we are initializing the dashboard, dashboard is a combination of multi select filter and also various charts together.

(Refer Slide Time: 05:48)



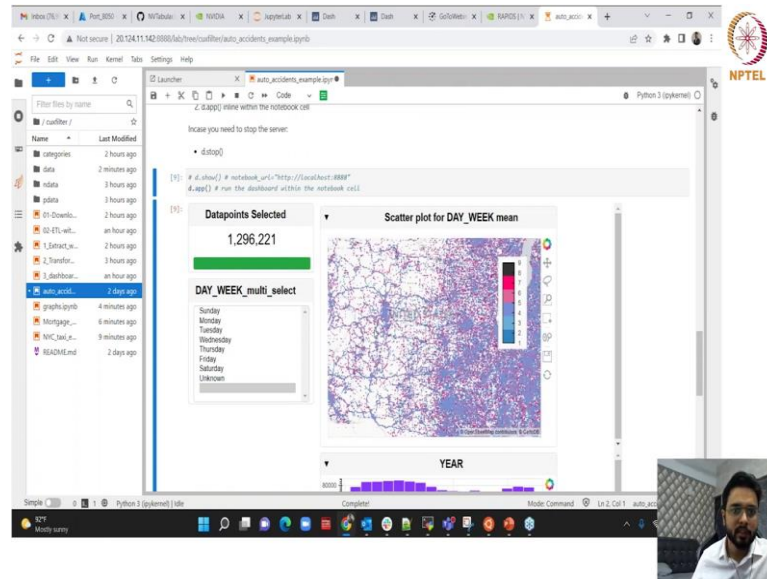
So, here is the chart, if you I will just put it in full screen. So, here is the chart which has been created Sunday, Monday up to Saturday here it is, on the right hand side if you see this is the scatter plot for the auto accident and how many auto accidents happened in which region.

(Refer Slide Time: 06:09)



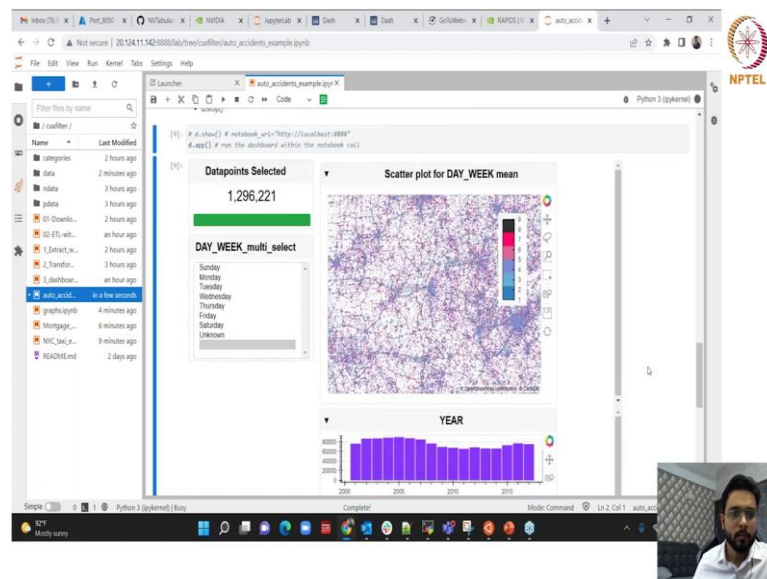
So, if you see the blue ones are the least, why there is no color; that means, no data around that and there where we have the dark color; that means, there were a lot of accidents there.

(Refer Slide Time: 06:23)



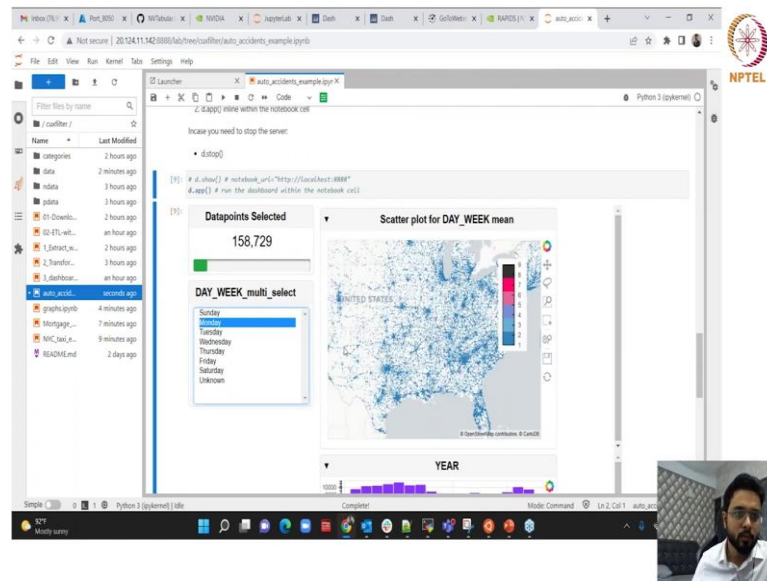
So, this is this is how fast it is seen, how fast it is filtering.

(Refer Slide Time: 06:29)



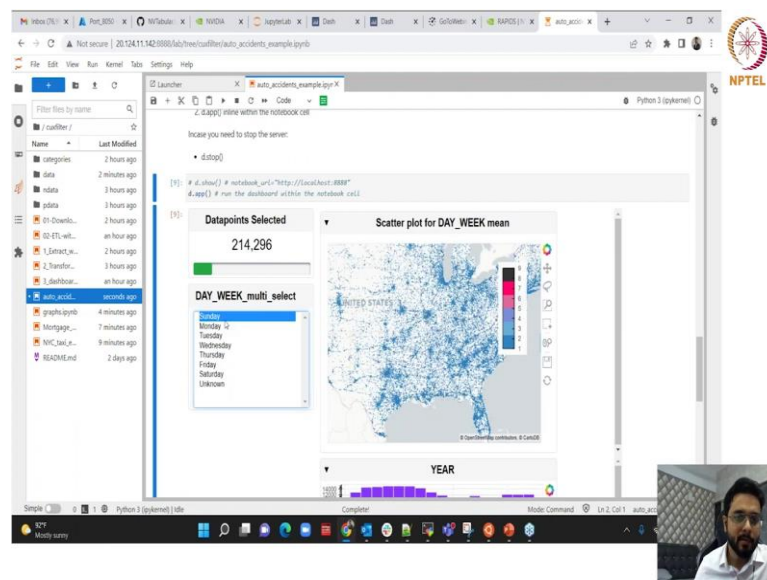
How fast I am able to do the magnification and all the processing, which is which may not be possible for large datasets on CPUs easily. And the other chart is about a simple bar graph which is showing year based how many accidents had happened per year.

(Refer Slide Time: 06:51)



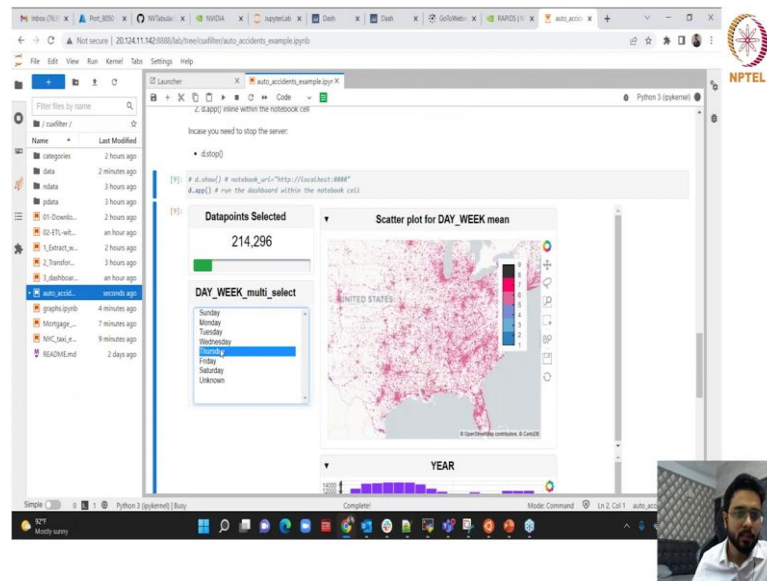
And then, if you want to see the day of the week so we just filter on Monday, the graph will change, see the graph has changed. So, Monday only few accidents happened so it is just 1.

(Refer Slide Time: 07:04)



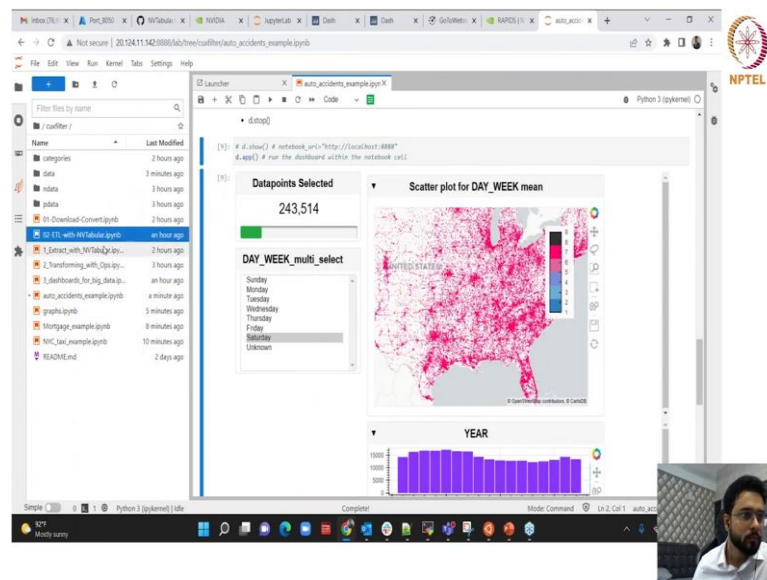
Sunday a little more.

(Refer Slide Time: 07:09)



Let us go on Saturday, Thursday, Thursday a lot of accidents. So, you can see based on the day of weeks, the number of accidents are changing it is red; that means, 6 or 7 or something around that, right. This is how good visualizations are to understand the patterns inside the data, alright.

(Refer Slide Time: 07:37)



Saturday also there are a lot of accidents ok. So, these are the simple visualizations using cuXfilter. So, if you want to do using plotly also, I will just show you how it works at a notebook level.

(Refer Slide Time: 08:10)

3_dashboards_for_big_data.ipynb

Header

Dashboards for Big Data

Now that we've scaled our data to read across all of our `.csv` for a single day, let's scale it even further by allowing users to interact with our map and select which day they wish to view.

Objects

- Learn how to arrange elements on a Plotly Dash dashboard
- Learn how to make a dashboard interactive

Planning Ahead

There are many different ways to build a dashboard. It can be easy to get lost sometimes, so before we write any code, let's sketch out the features and layout that we'd like to see. In the end, our dashboard will look something like this, but with your style as defined in the previous lab.

So, this is for using plotly, it is a bit different, you do not have to import `cuXfilter`.

(Refer Slide Time: 08:19)

3_dashboards_for_big_data.ipynb

Plotly Dash is a framework that extends Plotly so we can turn our figures into a web service that serves a dashboard. A little bit of HTML and CSS knowledge will be useful here, but for the most part, this dashboard is built using Python.

Here's how the web service will work:

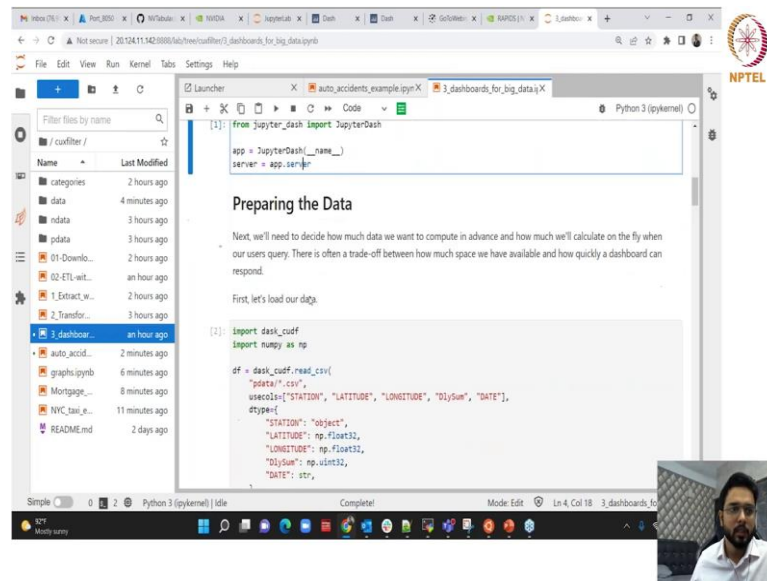
- When a user visits our site, they can select a date to view precipitation for
- Plotly has a mode to do the calculation on the client (user's computer), but not everyone has a supercomputer to breeze through data calculation
- The selected date will be sent to the server (our computer), so our GPU can handle filtering the data
- Our GPU filters the data for the date, sends it back to our host (CPU + RAM), so Plotly can generate a new graph and send it to the client.

Let's get the base of our server setup. We will be using a version of Dash built for Jupyter notebooks called `Jupyter Dash`. The below cell will define our server.

```
[1]: from jupyter_dash import JupyterDash
app = JupyterDash(__name__)
server = app.server
```

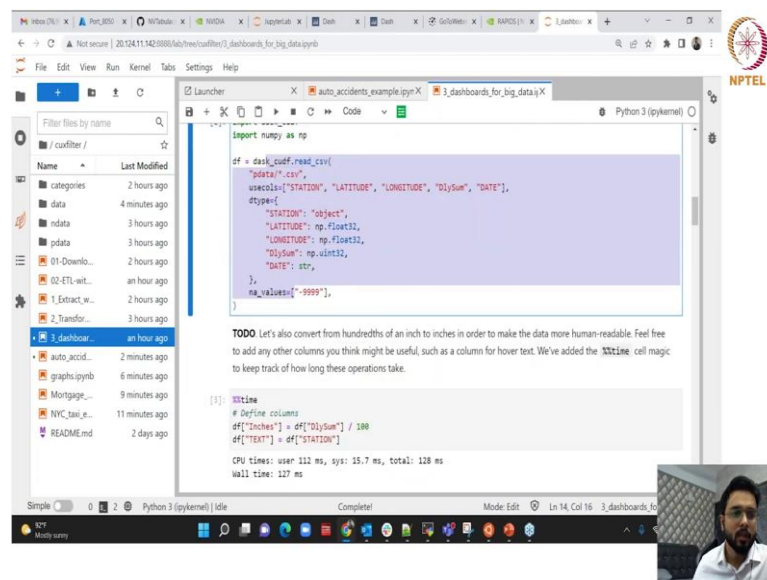
Preparing the Data

(Refer Slide Time: 08:21)



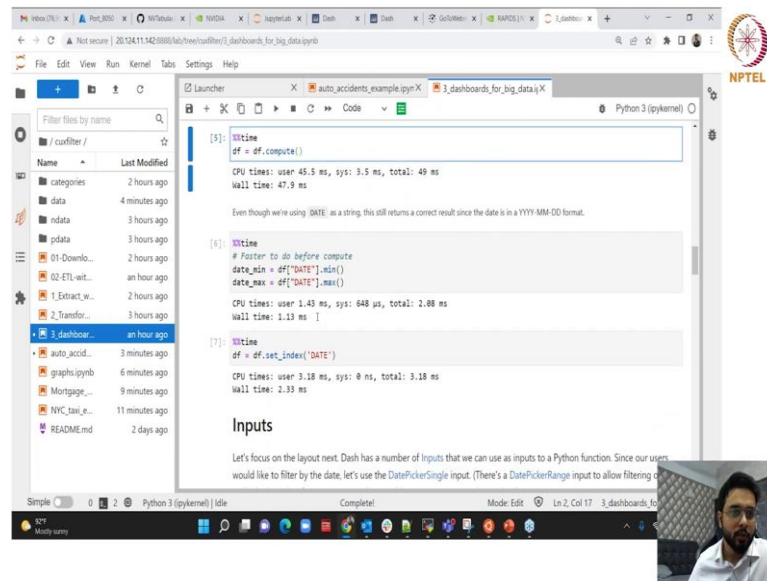
So, first you have to like import Jupyter dash. So, you have to install Jupyter dash, if you have if you do not have it in by default in NGC container docker. So, keep install Jupyter_dash and then you initialize the server.

(Refer Slide Time: 08:36)



And then we have the dashboard here, where you import the data so where your station latitude longitude. So, basically create a DASK data frame.

(Refer Slide Time: 08:49)



```
[5]: %time
df = df.compute()
CPU times: user 45.5 ms, sys: 3.5 ms, total: 49 ms
Wall time: 47.9 ms

Even though we're using DATE as a string, this still returns a correct result since the date is in a YYYY-MM-DD format.

[6]: %time
# Faster to do before compute
date_min = df['DATE'].min()
date_max = df['DATE'].max()
CPU times: user 1.43 ms, sys: 648 µs, total: 2.08 ms
Wall time: 1.13 ms

[7]: %time
df = df.set_index('DATE')
CPU times: user 3.18 ms, sys: 0 ns, total: 3.18 ms
Wall time: 2.33 ms
```

Inputs

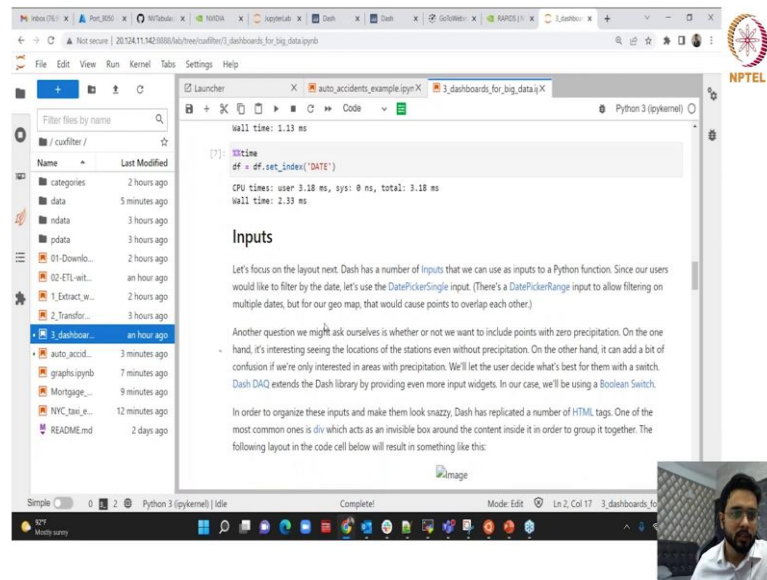
Let's focus on the layout next. Dash has a number of `Inputs` that we can use as inputs to a Python function. Since our users would like to filter by the date, let's use the `DatePickerSingle` input. (There's a `DatePickerRange` input to allow filtering on multiple dates, but for our geo map, that would cause points to overlap each other)

Another question we might ask ourselves is whether or not we want to include points with zero precipitation. On the one hand, it's interesting seeing the locations of the stations even without precipitation. On the other hand, it can add a bit of confusion if we're only interested in areas with precipitation. We'll let the user decide what's best for them with a switch. Dash `DAQ` extends the Dash library by providing even more input widgets. In our case, we'll be using a `Boolean Switch`.

In order to organize these inputs and make them look snazzy, Dash has replicated a number of HTML tags. One of the most common ones is `div` which acts as an invisible box around the content inside it in order to group it together. The following layout in the code cell below will result in something like this:

And then basic do some basic transformations and then do compute to get the data frame out of it. So, this is important. So, plotly will not take in us direct DASK data frame as an input, you have to compute it that we get a normal cuDF data frame before you pass that on to plotly, that is what I was seeing it works on the CPU.

(Refer Slide Time: 09:16)



```
[7]: %time
df = df.set_index('DATE')
CPU times: user 3.18 ms, sys: 0 ns, total: 3.18 ms
Wall time: 2.33 ms
```

Inputs

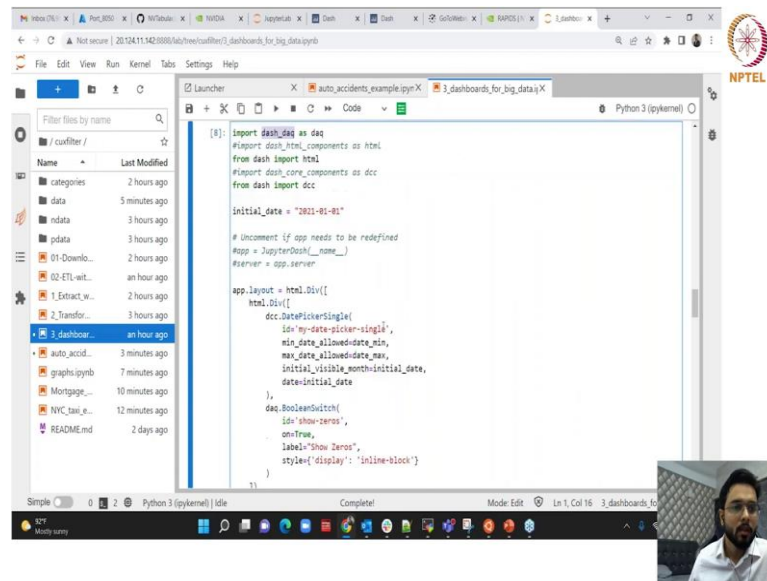
Let's focus on the layout next. Dash has a number of `Inputs` that we can use as inputs to a Python function. Since our users would like to filter by the date, let's use the `DatePickerSingle` input. (There's a `DatePickerRange` input to allow filtering on multiple dates, but for our geo map, that would cause points to overlap each other)

Another question we might ask ourselves is whether or not we want to include points with zero precipitation. On the one hand, it's interesting seeing the locations of the stations even without precipitation. On the other hand, it can add a bit of confusion if we're only interested in areas with precipitation. We'll let the user decide what's best for them with a switch. Dash `DAQ` extends the Dash library by providing even more input widgets. In our case, we'll be using a `Boolean Switch`.

In order to organize these inputs and make them look snazzy, Dash has replicated a number of HTML tags. One of the most common ones is `div` which acts as an invisible box around the content inside it in order to group it together. The following layout in the code cell below will result in something like this:

Though the pre-processing and all the back end happens on the GPU, but the plotly takes the CPU data flip.

(Refer Slide Time: 09:26)



```
[8]: import dash_extensions as daq
# Import dash_html_components as html
from dash import html
# Import dash_core_components as dcc
from dash import dcc

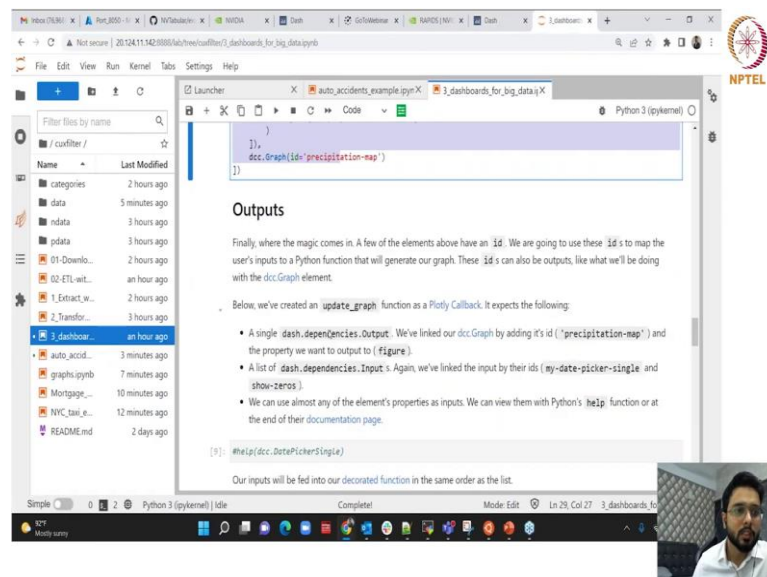
initial_date = "2021-01-01"

# Uncomment if app needs to be redefined
# app = JupyterDash(__name__)
# server = app.server

app.layout = html.Div([
    html.Div([
        dcc.DatePickerSingle(
            id='my-date-picker-single',
            min_date_allowed=date_min,
            max_date_allowed=date_max,
            initial_visible_month=initial_date,
            date=initial_date
        ),
        daq.BooleanSwitch(
            id='show-zeros',
            on=True,
            label='Show Zeros',
            style={'display': 'inline-block'}
        )
    ])
])
```

So, here if set index and all is there. So, here the I mean, the plotly dash part start again for dash you have to use dash underscore DAQ. So, Dash DAQ is the extend the Dash library for providing even more input feature. So, it is just more feature full plotly Dash library version. So, after importing you have to create that app.layout which I was explaining you. So, this is html Div, date picker, BooleanSwitch and all that stuff.

(Refer Slide Time: 09:48)



```
    ],
    dcc.Graph(id='precipitation-map')
])
```

Outputs

Finally, where the magic comes in. A few of the elements above have an `id`. We are going to use these `id`'s to map the user's inputs to a Python function that will generate our graph. These `id`'s can also be outputs, like what we'll be doing with the `dcc.Graph` element.

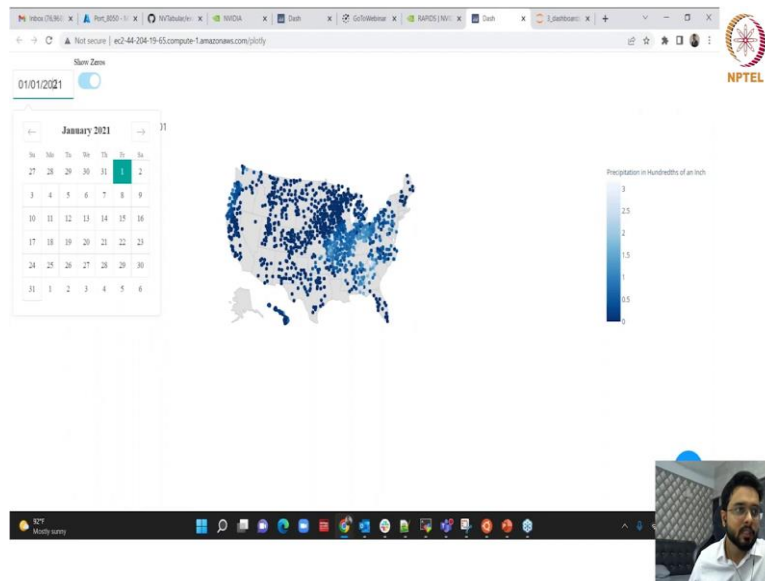
Below, we've created an `update_graph` function as a Plotly Callback. It expects the following:

- A single `dash.dependencies.Output`. We've linked our `dcc.Graph` by adding its `id` ("precipitation-map") and the property we want to output to (`figure`).
- A list of `dash.dependencies.Input`'s. Again, we've linked the input by their `ids` (`my-date-picker-single` and `show-zeros`).
- We can use almost any of the element's properties as inputs. We can view them with Python's `help` function or at the end of their documentation page.

```
[9]: #help(dcc.DatePickerSingle)

Our inputs will be fed into our decorated function in the same order as the list.
```

(Refer Slide Time: 10:03)



So, I will just show you how it works yeah.

(Refer Slide Time: 09:18)

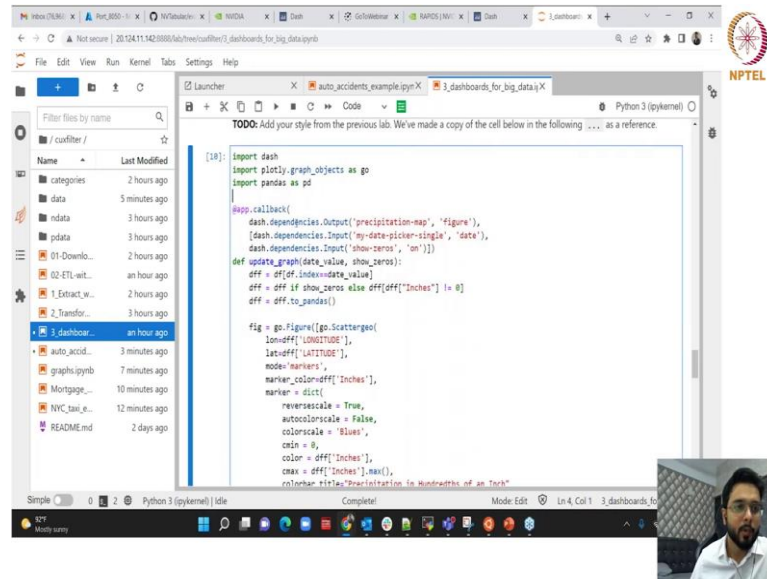
The screenshot shows a Jupyter Notebook interface. The left sidebar displays a file explorer with a list of files and folders, including 'categories', 'data', 'ndata', 'pdata', and '3_dashboard...'. The main area contains a code cell with the following Python code:

```
[18]: import dash
import plotly.graph_objects as go
import pandas as pd

@app.callback(
    dash.dependencies.Output('precipitation-map', 'figure'),
    [dash.dependencies.Input('my-date-picker-single', 'date'),
     dash.dependencies.Input('show-zeros', 'on')]
)
def update_graph(date_value, show_zeros):
    diff = diff_between_dates(date_value)
    diff = diff if show_zeros else diff[diff["Inches"] != 0]
    diff = diff.to_pandas()
```

The code cell includes comments explaining the inputs and the purpose of the function. A 'TODO' note is also present. The Jupyter interface includes a 'Launcher' window and a 'Python 3 (pykernel)' environment indicator. A small video inset in the bottom right corner shows a man speaking.

(Refer Slide Time: 10:20)



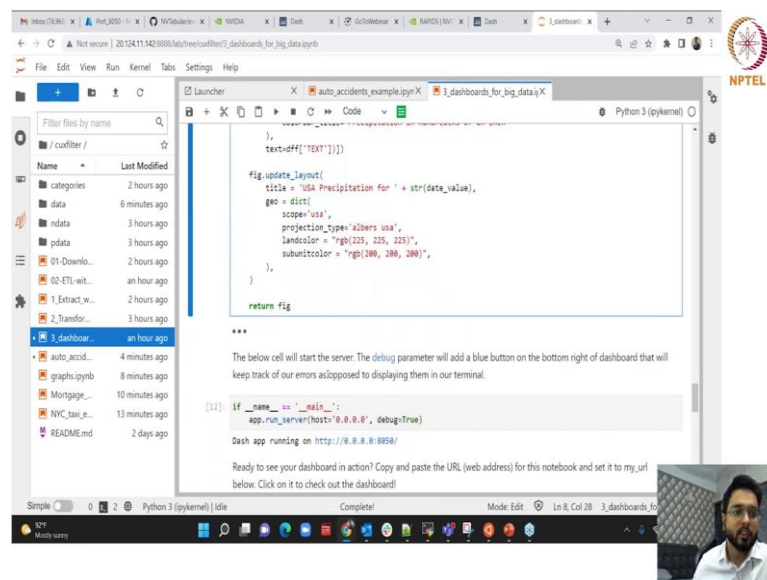
```
import dash
import plotly.graph_objects as go
import pandas as pd

@app.callback(
    dash.dependencies.Output('precipitation-map', 'figure'),
    [dash.dependencies.Input('my-date-picker-single', 'date'),
     dash.dependencies.Input('show_zeros', 'on')]
)
def update_graph(data_value, show_zeros):
    dff = df[df.index<data_value]
    dff = dff if show_zeros else dff[dff['Inches'] != 0]
    dff = dff.to_pandas()

    fig = go.Figure([go.Scattergeo(
        lon=dff['LONGITUDE'],
        lat=dff['LATITUDE'],
        mode='markers',
        marker_color=dff['Inches'],
        marker=dict(
            reversescale = True,
            autocolorscale = False,
            colorscale = 'Blues',
            cmin = 0,
            color = dff['Inches'],
            cmax = dff['Inches'].max(),
            colorbar_title='Precipitation in Hundreds of an Inch'
```

So, after you run all this then you create app.callback, which I was showing in the slides, where you pass the precipitation map, date picker and show-zeros elements of the dashboard and then you host that ok.

(Refer Slide Time: 10:29)



```
),
    text=dff['TEXT']]])

fig.update_layout(
    title = 'USA Precipitation for ' + str(data_value),
    geo = dict(
        scope='usa',
        projection_type='albers usa',
        landcolor = 'rgb(225, 225, 225)',
        subunitcolor = 'rgb(200, 200, 200)',
    ),
)

return fig

***
The below cell will start the server. The debug parameter will add a blue button on the bottom right of dashboard that will keep track of our errors as supposed to displaying them in our terminal.

[11] if __name__ == '__main__':
    app.run_server(host='0.0.0.0', debug=True)

Dash app running on http://0.0.0.0:8850/

Ready to see your dashboard in action? Copy and paste the URL (web address) for this notebook and set it to my url below. Click on it to check out the dashboard!
```

And then if you see, you want to see that how it looks, it will look like this. This is the date picker, this is the show-zeros and this is the precipitation map that ok, what is the amount of precipitation, 0 is the lowest, 3 is the highest.

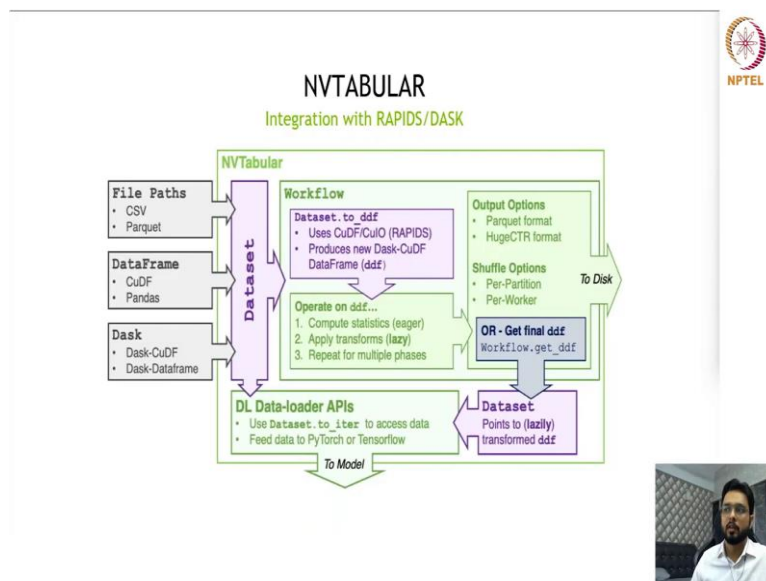
So, light we here the precipitation was a lot and the dark was, dark means its it was a relatively less. So, if you see this, this is like a new window altogether; that means, contrary to the other though cuXfilter can also be created in a new window. So, it is very similar to cuXfilter as well.

(Refer Slide Time: 11:19)



Now, going to another framework called NV TABULAR.

(Refer Slide Time: 11:35)



So, NV TABULAR is again very popular framework to do data pre-processing. So, you know that ok you have been doing data pre-processing using RAPIDS, cuDF or DASK

ah, but again using cuDF and DASK you have to write all the functions one by one like you do in pandas programming or you say sklearn programming or DASK programming. However, if you want to do pre processing in a shortcut manner by saving a lot of time.

Because we know that in a particular data science project, there are few numbers of operations, a few numbers of models which are more required ok. So, for example, if you see this NV Tabular architecture. So, what it exports is like file paths can be their CSV based path or parquet file path, super CSV and parquet are the data formats data file formats. It can have cuDF as input it can take pandas data frame as input or DASK, cuDF as input or a DASK data frame as an input.

And then we have all these inputs can be created to create something called data set NVTabular data set. And using this NVTabular data set we create something called workflow which is used to define the desired data transformation pipeline. So, workflow is something which defines the, what kind of transformations you will be doing, what kind of processing you will be doing and all that stuff.

And then we have the data loader part, where we used to feed into a tabular data source to a deep learning based model. So, no matter it can be a Keras based you can create some neural network model or LSTM or something like that. So, you can create using the load the data or convert the DASK data frame or a normal data frame into a TensorFlow data or a PyTorch data or something like that using the deep learning data loader API of NVTabular.

And then we have the other things like we have output options, if you just want to load the data into a deep learning model just you want to make it as output, you can use our output as a parquet format or a HugeCTR format. So, HugeCTR format is a format which accepts which you can do recommendation system.

So, if you want to create a recommender system using HugeCTR model, you want to train it then the data can be exported directly to a HugeCTR compatible format. Then we have like we have various operations on the data frame possible. Like for example, if you want to do compute stats like what are the maximum minimum value, aggregate values and all that stuff you want to apply transformations.

Like maybe bucketing, maybe normalization and all that stuff you can do that or using very shortcut, like shortcut methods of NVTabular. So, basically NVTabular accelerates further your development process of data science pipeline.



(Refer Slide Time: 14:48)

NVTABULAR KEY FEATURES

Faster and Easier GPU-based ETL

- GPU-accelerated, eliminating CPU bottlenecks.
- Out-of-core execution. No GPU memory limits and reduced I/O through lazy execution.
- PyTorch, TensorFlow and HugeCTR compatible.
- Filtering outliers or missing values.
- Inputting and filling in missing data.
- Discretization or bucketing of continuous features.
- Creating features by splitting or combining existing features.
- Normalizing numerical features to have zero mean and unit variance.
- Encoding discrete features using one-hot vectors or converting them to continuous integer indices.
- **More to come** 😊

	NVTabular	pandas
Dataset size limitation	Unlimited	CPU Memory
Code complexity	Simple	Moderate
Lines of code	10 - 20	100 - 1000
Flexibility	Domain specific	General
Data loading Transforms	Yes	No
Inference Transforms	Yes	No

Key features are its completely GPU accelerated, it supports out of core execution; that means, the data volume is more then also it will not fail then the GPU memory, then it supports PyTorch, TensorFlow, HugeCTR. It filters the outliers or missing values, it helps to do the transformations.

So, all these transformations are possible, filtering out layers missing value removal, then we have input the and, filling the missing data discretization of bucketing, creating features by splitting and exist combining the existing features. So, merging and all that stuff. Normalizing numerical features to have zero mean and unit variance, then encoding discrete features using one hot encoding or converting them to continuous integer indices.


And there are more and more which are coming in every release and there is a whole list of operations which are possible, which you can see in the documentation again open source. Comparison from NVTabular to pandas, even cuDF data size limitation it is based on CPU memory, but here it is unlimited, code complexity is very simple. So, this is one of the biggest thing that only 10 to 20 lines of code will be needed as per 100 to 1000 lines of code in pandas.

Flexibility is domain specific; that means, if you want to do in the retail based recommender system or if you want to do some other forecasting model, it will be different set of operations it is domain specific transformations are available. Data loading is possible to deep learning model which is not possible in pandas. Then inferencing; that means, if you want to input the data while predicting the data and do some real time pre-processing and then do the prediction then also it is possible using NVTabular.


(Refer Slide Time: 16:42)

NVTabular vs Pandas code

100x fewer lines of code required



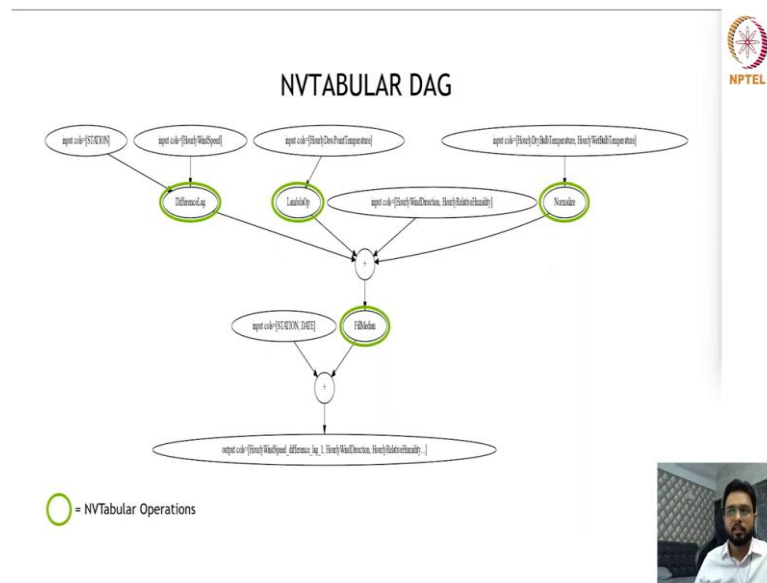
<pre>import glob import nvtabular as nvt # Create datasets from input files train_files = glob.glob("./dataset/train/*.parquet") valid_files = glob.glob("./dataset/valid/*.parquet") train_ds = nvt.Dataset(train_files, gpu_memory_frac=0.1) valid_ds = nvt.Dataset(valid_files, gpu_memory_frac=0.1)</pre>	Import libraries. Create training and validation datasets.
<pre># Initialise workflow cat_names = ["C" + str(x) for x in range(1, 27)] # Specify categorical feature names cont_names = ["I" + str(x) for x in range(1, 14)] # Specify continuous feature names label_name = ["label"] # Specify target feature proc = nvt.Workflow(cat_names=cat_names, cont_names=cont_names, label_name=label_name)</pre>	Initialise workflow specifying categorical and continuous data.
<pre># Add feature engineering and pre-processing ops to workflow proc.add_cont_feature([nvt.ops.ZeroFill(), nvt.ops.LogOp()]) proc.add_cont_preprocess([nvt.ops.Normalize()]) proc.add_cat_preprocess([nvt.ops.Categorify(use_frequency=True, freq_threshold=15)])</pre>	Zero fill any nulls, log transform and normalize continuous variables. Encode categorical data.
<pre># Compute statistics, transform data, and export to disk proc.apply(train_dataset, shuffle=True, output_path="./processed_data/train", num_out_files=len(train_files)) proc.apply(valid_dataset, shuffle=False, output_path="./processed_data/valid", num_out_files=len(valid_files))</pre>	Apply the ops creating new shuffled training and valid datasets.



So, if you see let us understand the code example that how easy it is to do it. So, on the left-hand side if you see this is the NVTabular code, where we import nvtabular import the files then create the nvtabular data set, which I was explaining. Then we create the category names so category features, then create the workflow and in the workflow we do some feature engineering in pre-processing like continuous features we do the zero filling or the log operations.

We do the for the pre-processing we do the normalization. For categorification; that means, if you want to change numerical values into categories you can do that. So, using categorify and then apply all this on the training and validation set and then pass it to the model. So, this is how easy it is. So, entire code is fitted into this PPT, hence you can understand how easy it is to do the data science workflow using NVTabular.

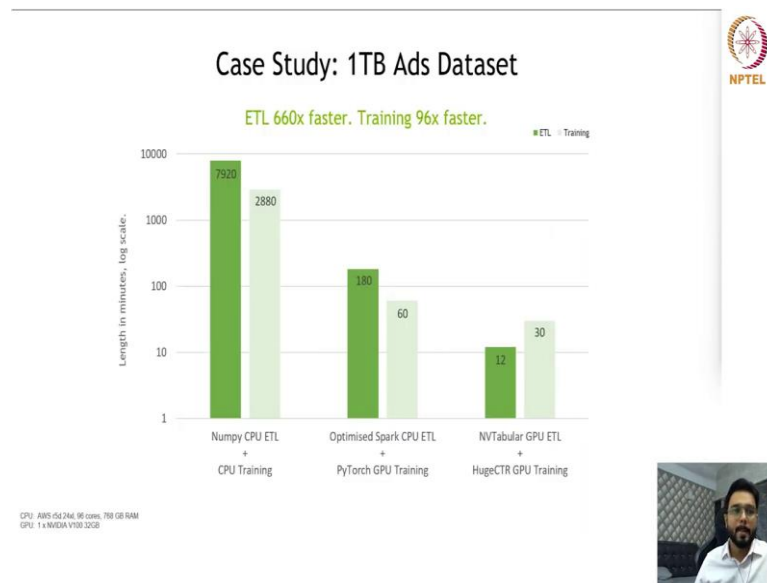
(Refer Slide Time: 17:52)



So, it creates a DAG like this. So, which is the cyclic graph; that means, you input the column, station, you input the column hourly wind speed. For these two columns if you want to do one kind of difference logging operations, for another column you have to do the lambda operation lambda operation means that any custom function you can write, like the lambda function of Python. And then if you want to include some other columns and you want to normalize for that, so all that can be done.

And then finally, you can get the output columns. So, these are the NVTabular operations, difference logging, lambda operations normalization fill missing values with median values. So, all this the, NVTabular has a graphing tool as well to you can see the DAG that is called Graphviz, that I will show you in the hands on ok.

(Refer Slide Time: 18:44)

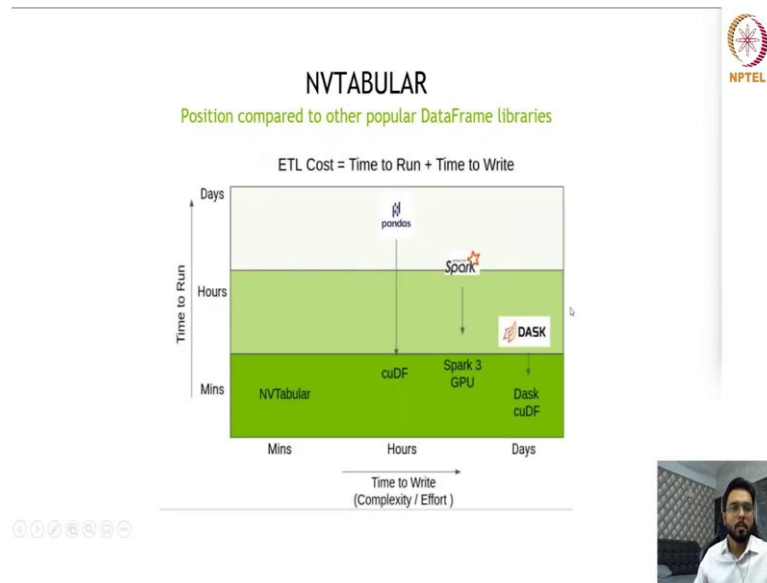


So, if you see like this particular slide, where we have done some benchmarking. So, apart from the ease of writing the code, how fast it is so the another biggest advantage is it is 660x faster based on the 1 TB Ads dataset case study we did.

So, we took the open source Criteo data set and we did ETL and then model training using HugeCTR. So, we compared with Numpy CPU ETL versus CPU based training and then we also use frameworks like Spark, which is very scalable and multi node framework. And then PyTorch, PyTorch GPU training and then we compared with NVTabular GPU. And we use similar like from the cost perspective, similar costing machines.

It is not that we use a very costly GPU machine but we use a very cheap CPU machine not like that it was very comparable still we got this kind of better performance out of it. So, just the pre processing run for 12 minutes versus 180 minutes in a Spark, 13 minutes versus 16 minutes in a Spark the model training.

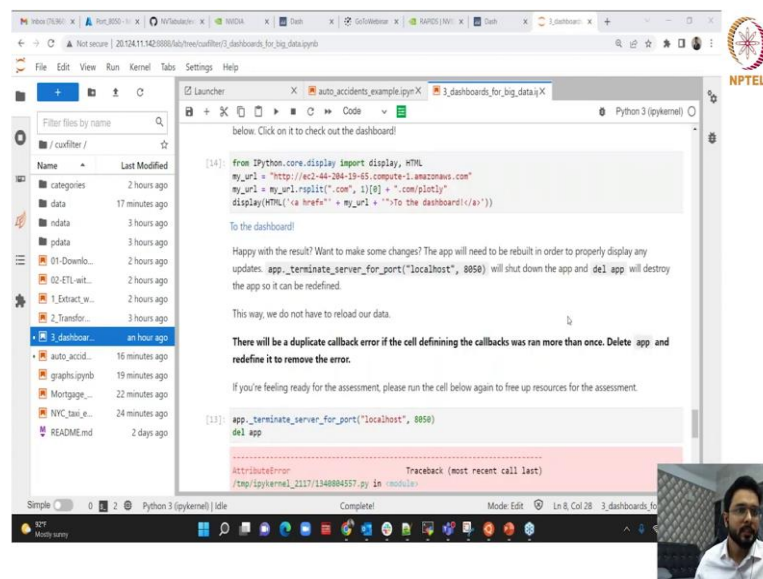
(Refer Slide Time: 20:04)



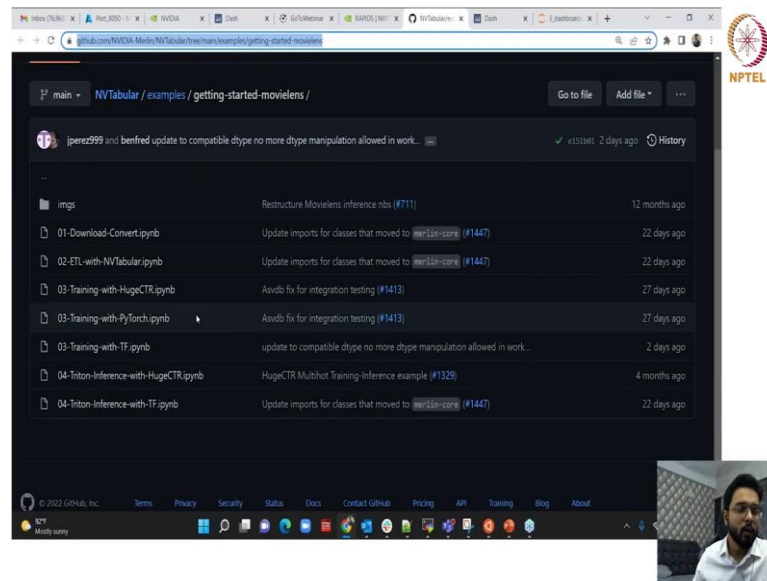
So, this is the how the performance compares. So, NVTabular is one of the fastest has comparable to DASK, cuDF. Then second fastest is that Spark GPU and then we have the cuDF and finally, the slowest is pandas.

So, this is the performance aspect of it. Let me show you some hands on, before I move to the next part.

(Refer Slide Time: 20:57)

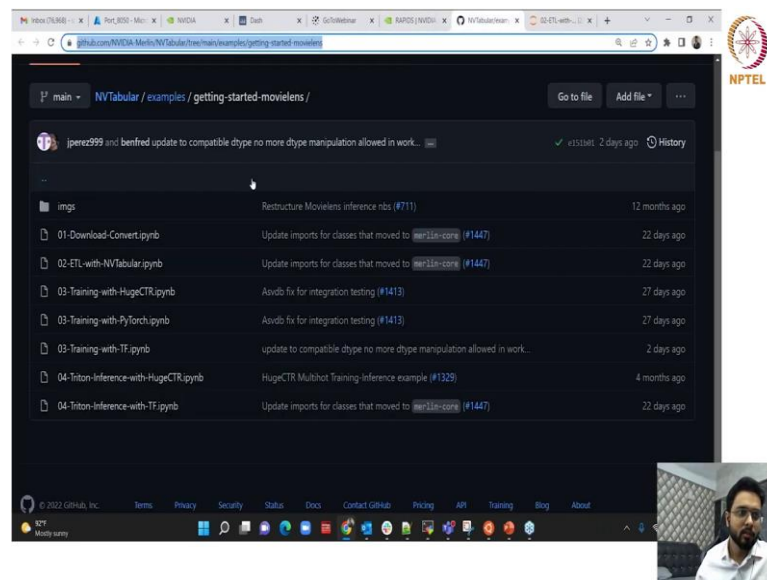


(Refer Slide Time: 20:58)



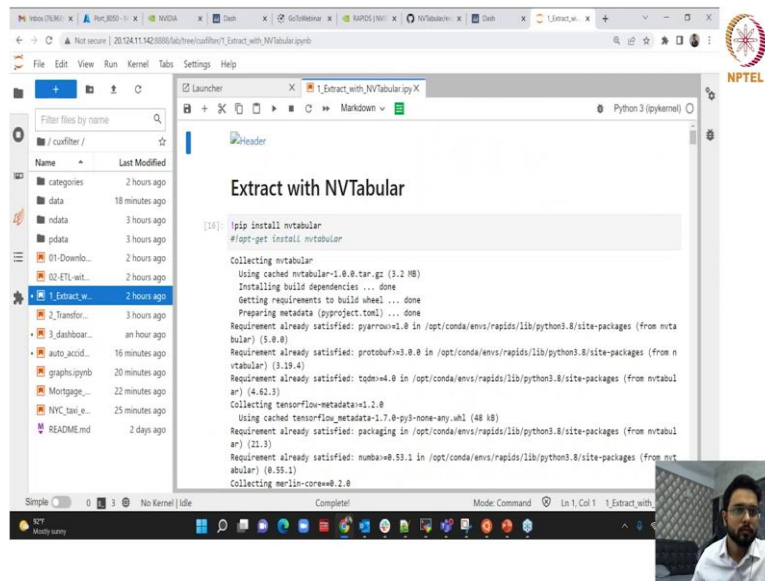
So, for this hand on, I just went to the open source NVTabular repository getting started movie lens, particular folder.

(Refer Slide Time: 21:07)



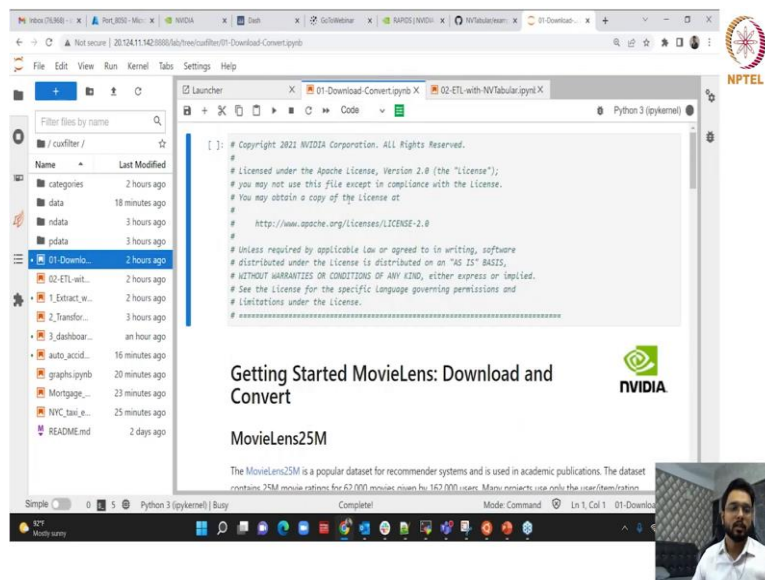
And then I downloaded these two notebooks, download convert ipynb and ETL with and NVTabular. So, again this is open source.

(Refer Slide Time: 21:18)



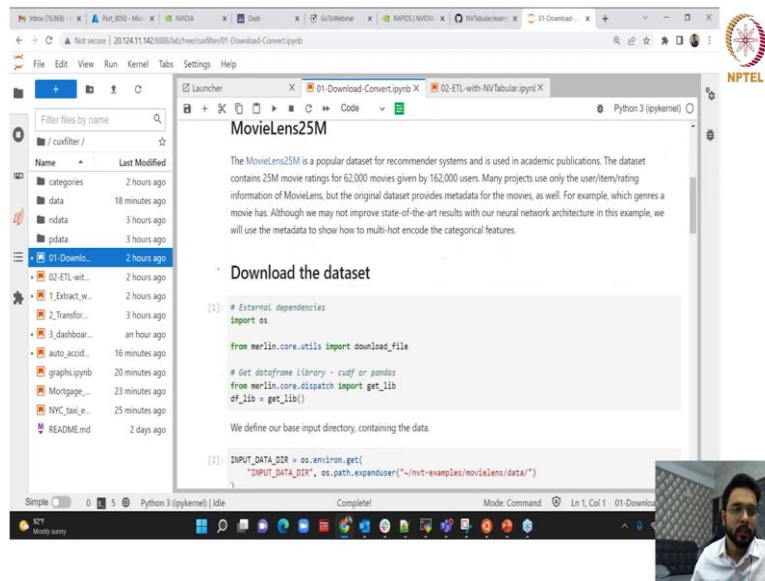
So, before that so, I will just open that folder. So, there are two 01 download convert and 02. So, I will open that 01 and 02.

(Refer Slide Time: 21:37)



So, in the 01 we will download the movie lens data set.

(Refer Slide Time: 21:40)



The screenshot shows a Jupyter Notebook titled "MovieLens25M". The notebook content includes a text block describing the dataset and a code cell with the following code:

```
[1]: # External dependencies
import os

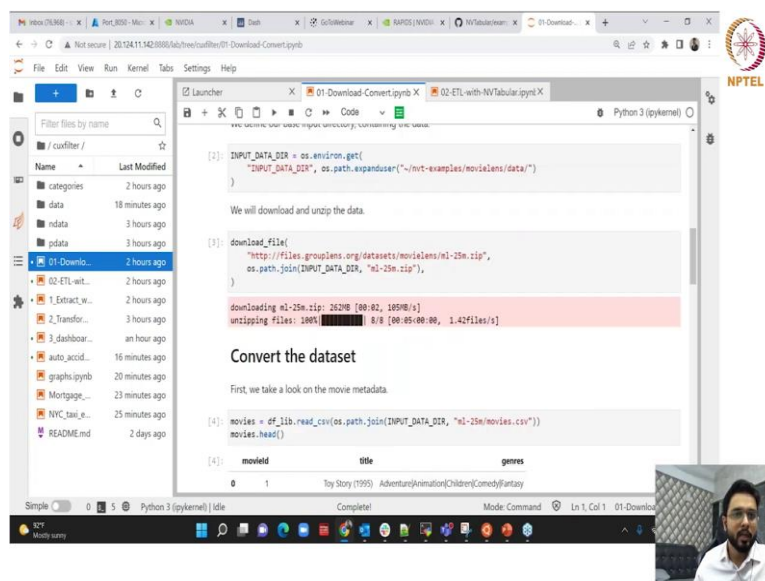
from merlin.core.utils import download_file

# Get dataframe library - cudf or pandas
from merlin.core.dispatch import get_lib
df_lib = get_lib()

We define our base input directory, containing the data.

[2]: INPUT_DATA_DIR = os.environ.get(
    "INPUT_DATA_DIR", os.path.expanduser("~/nvt-examples/movielens/data/")
```

(Refer Slide Time: 21:41)



The screenshot shows the Jupyter Notebook continuing with the following code:

```
[3]: INPUT_DATA_DIR = os.environ.get(
    "INPUT_DATA_DIR", os.path.expanduser("~/nvt-examples/movielens/data/")
)

We will download and unzip the data.

[3]: download_file(
    "http://files.grouplens.org/datasets/movielens/ml-25m.zip",
    os.path.join(INPUT_DATA_DIR, "ml-25m.zip"),
)

downloading ml-25m.zip: 262MB [00:02, 105MB/s]
unzipping files: 100% [00:08, 8/8 [00:05<00:00, 1.42Files/s]]

Convert the dataset

First, we take a look on the movie metadata.

[4]: movies = df_lib.read_csv(os.path.join(INPUT_DATA_DIR, "ml-25m/movies.csv"))
movies.head()
```

movieid	title	genres	
0	1	Toy Story (1995)	Adventure Animation Children Comedy Fantasy

So, here it is. So, if you see here, we download the data set merlin core dispatch get lib.

(Refer Slide Time: 21:54)

The screenshot shows a Jupyter Notebook interface with a file explorer on the left and a code editor on the right. The code in the notebook is as follows:

```
[4]: movies = df_lib.read_csv(os.path.join(INPUT_DATA_DIR, "nl-15e/movies.csv"))
movies.head()
```

	movieId	title	genres
0	1	Toy Story (1995)	Adventure Animation Children Comedy Fantasy
1	2	Jumanji (1995)	Adventure Children Fantasy
2	3	Grumpier Old Men (1995)	Comedy Romance
3	4	Waiting to Exhale (1995)	Comedy Drama Romance
4	5	Father of the Bride Part II (1995)	Comedy

Below the table, there is a text explanation: "We can see, that genres are a multi-hot categorical features with different number of genres per movie. Currently, genres is a String and we want split the String into a list of Strings. In addition, we drop the title."

```
[5]: movies['genres'] = movies['genres'].str.split("|")
movies = movies.drop("title", axis=1)
movies.head()
```

	movieId	genres
0	1	[Adventure, Animation, Children, Comedy, Fantasy]
1	2	[Adventure, Children, Fantasy]

The interface also shows a small video feed of the presenter in the bottom right corner.

And then input the data directory, download the file, convert the data set.

(Refer Slide Time: 22:00)

The screenshot shows the same Jupyter Notebook interface as before, but with updated code and output:

```
[5]: movies['genres'] = movies['genres'].str.split("|")
movies = movies.drop("title", axis=1)
movies.head()
```

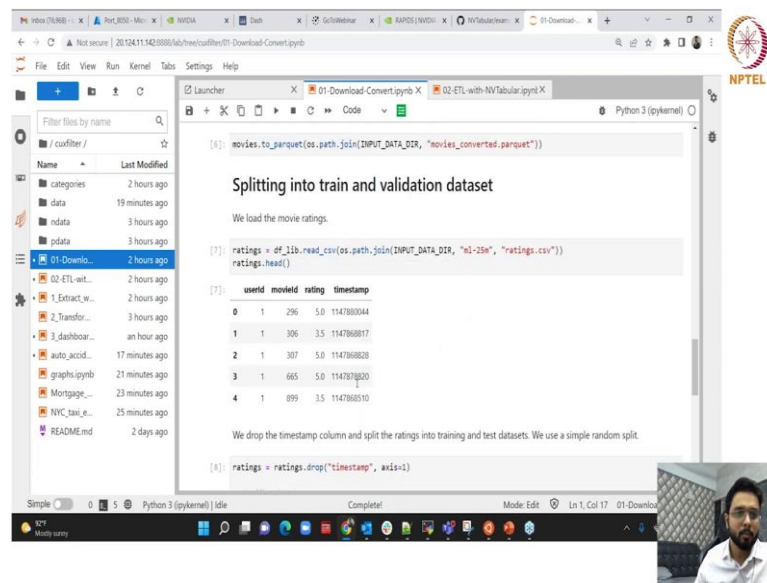
	movieId	genres
0	1	[Adventure, Animation, Children, Comedy, Fantasy]
1	2	[Adventure, Children, Fantasy]
2	3	[Comedy, Romance]
3	4	[Comedy, Drama, Romance]
4	5	[Comedy]

Below the table, there is a text explanation: "We save movies genres in parquet format, so that they can be used by NVTabular in the next notebook."

The interface also shows a small video feed of the presenter in the bottom right corner.

Then put the genres and all that stuff to basic pre-processing.

(Refer Slide Time: 22:05)



```
[6]: movies.to_parquet(os.path.join(INPUT_DATA_DIR, "movies_converted.parquet"))
```

Splitting into train and validation dataset

We load the movie ratings.

```
[7]: ratings = df_lib.read_csv(os.path.join(INPUT_DATA_DIR, "ml-25m", "ratings.csv"))
ratings.head()
```

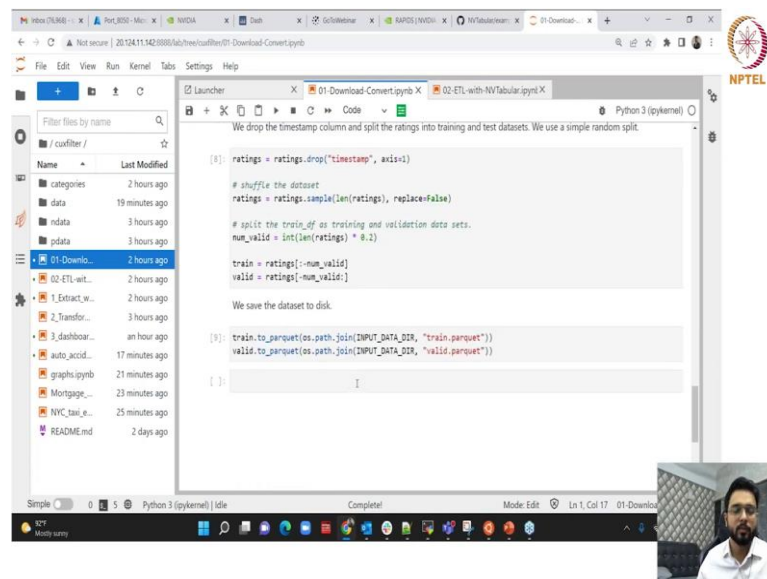
userid	movieid	rating	timestamp
0	1	296	5.0 1147800044
1	1	306	3.5 1147868817
2	1	307	5.0 1147868828
3	1	665	5.0 1147878820
4	1	899	3.5 1147868510

We drop the timestamp column and split the ratings into training and test datasets. We use a simple random split.

```
[8]: ratings = ratings.drop("timestamp", axis=1)
```

And then convert it to parquet format and write it back to the DASK.

(Refer Slide Time: 22:10)



```
[8]: ratings = ratings.drop("timestamp", axis=1)

# shuffle the dataset
ratings = ratings.sample(len(ratings), replace=False)

# split the train_of as training and validation data sets.
num_valid = int(len(ratings) * 0.2)

train = ratings[:(len(ratings) - num_valid)]
valid = ratings[(len(ratings) - num_valid):]

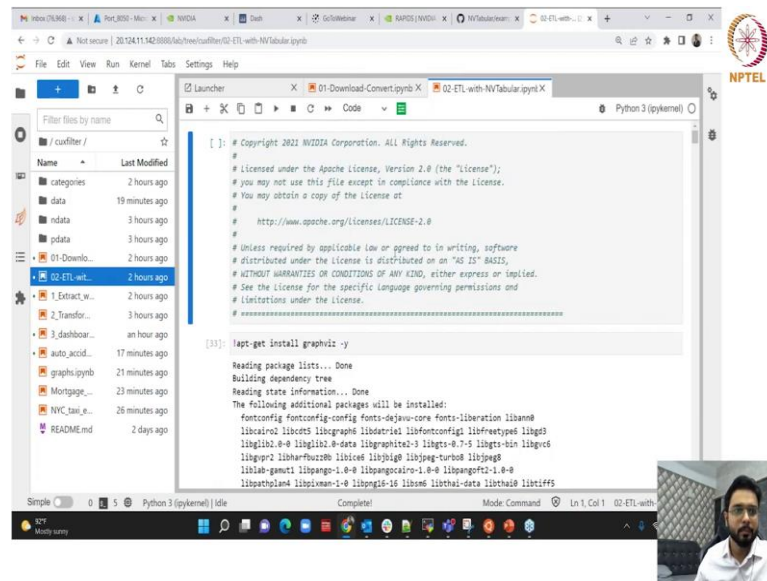
We save the dataset to disk.

[9]: train.to_parquet(os.path.join(INPUT_DATA_DIR, "train.parquet"))
valid.to_parquet(os.path.join(INPUT_DATA_DIR, "valid.parquet"))

[ ]:
```

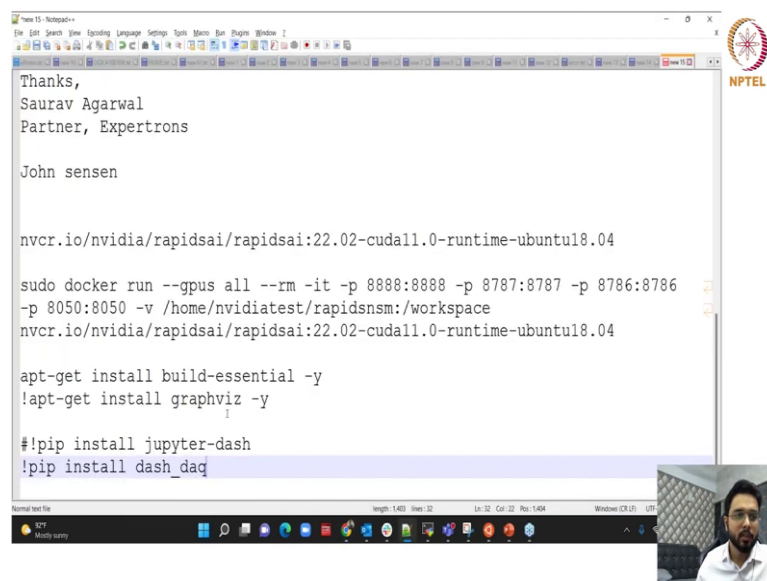
So, this is normal, this is now nowhere we are using NVTabular here it's just pandas. So, just to prepare the data, I mean download the data.

(Refer Slide Time: 22:17)



```
[ ]: # Copyright 2021 NVIDIA Corporation. All Rights Reserved.  
# Licensed under the Apache License, Version 2.0 (the "License");  
# you may not use this file except in compliance with the License.  
# You may obtain a copy of the License at  
#  
# http://www.apache.org/licenses/LICENSE-2.0  
#  
# Unless required by applicable law or agreed to in writing, software  
# distributed under the license is distributed on an "AS IS" BASIS,  
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.  
# See the License for the specific language governing permissions and  
# limitations under the license.  
# =====  
  
[33]: !apt-get install graphviz -y  
Reading package lists... Done  
Building dependency tree  
Reading state information... Done  
The following additional packages will be installed:  
fontconfig fontconfig-config fonts-dejavu-core fonts-liberation libasf0  
libcairo2 libcurl5 libgraphviz libidn12 libfontconfig libfontconfig1 libfontconfig2 libgpg2  
libglib2.0-0 libglib2.0-data libgraphviz3 libgts-0.7-5 libgts-bin libgvc6  
libgvpr2 libharfbuzz0 libice6 libjbig0 libjpeg-turbo8 libjpeg8  
liblab-gamut1 libpango-1.0-0 libpangocairo-1.0-0 libpangoft2-1.0-0  
libpathplan4 libpixman-1-0 libpng16-16 libsharpe4 libthai-data libthai0 libtiff5
```

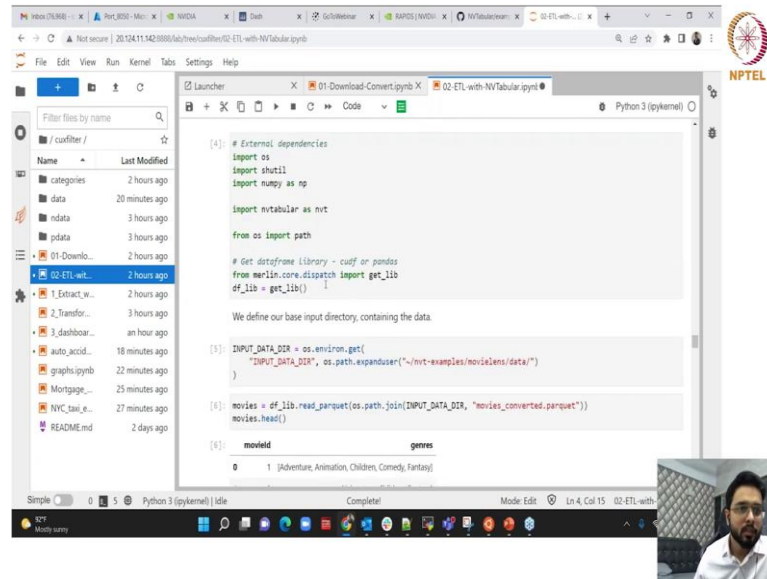
(Refer Slide Time: 22:23)



```
Thanks,  
Saurav Agarwal  
Partner, Expertrons  
  
John sensen  
  
nvcr.io/nvidia/rapidsai/rapidsai:22.02-cuda11.0-runtime-ubuntu18.04  
  
sudo docker run --gpus all --rm -it -p 8888:8888 -p 8787:8787 -p 8786:8786  
-p 8050:8050 -v /home/nvidiatest/rapidsnm:/workspace  
nvcr.io/nvidia/rapidsai/rapidsai:22.02-cuda11.0-runtime-ubuntu18.04  
  
apt-get install build-essential -y  
!apt-get install graphviz -y  
  
#!pip install jupyter-dash  
!pip install dash_daq
```

So, after that what we did is I installed NVTabular. So, using two formats, one is apt-get install build-essential -y.

(Refer Slide Time: 23:31)



```
[4]: # External dependencies
import os
import shutil
import numpy as np

import nvtabular as nvt

from os import path

# Get dataframe library - cudf or pandas
from merlin.core.dispatch import get_lib
df_lib = get_lib()

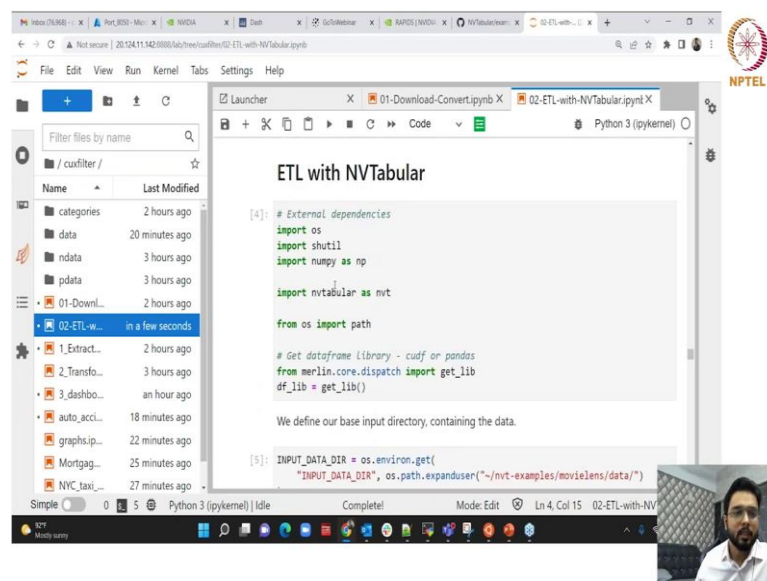
We define our base input directory, containing the data.

[5]: INPUT_DATA_DIR = os.environ.get(
    "INPUT_DATA_DIR", os.path.expanduser("~/nvt-examples/movielens/data/")
)

[6]: movies = df_lib.read_parquet(os.path.join(INPUT_DATA_DIR, "movies_converted.parquet"))
movies.head()
```

	movieid	genres
0	1	(Adventure, Animation, Children, Comedy, Fantasy)

(Refer Slide Time: 23:38)



```
ETL with NVTabular

[4]: # External dependencies
import os
import shutil
import numpy as np

import nvtabular as nvt

from os import path

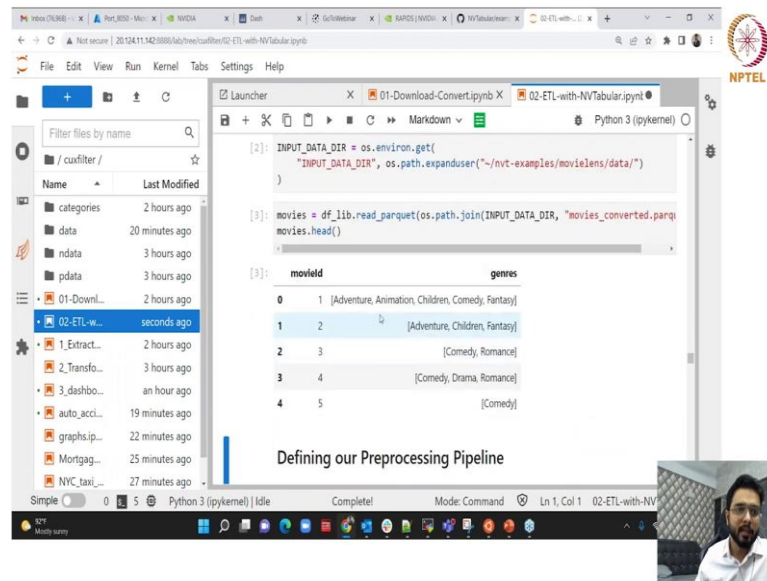
# Get dataframe library - cudf or pandas
from merlin.core.dispatch import get_lib
df_lib = get_lib()

We define our base input directory, containing the data.

[5]: INPUT_DATA_DIR = os.environ.get(
    "INPUT_DATA_DIR", os.path.expanduser("~/nvt-examples/movielens/data/")
)
```

So, I will just magnify it so that is visible clearly, detail with NVTabular, import os import shutil, numpy nvtabular, imported everything.

(Refer Slide Time: 23:47)



```
[2]: INPUT_DATA_DIR = os.environ.get(
      "INPUT_DATA_DIR", os.path.expanduser("~/nvt-examples/movielens/data/")
    )

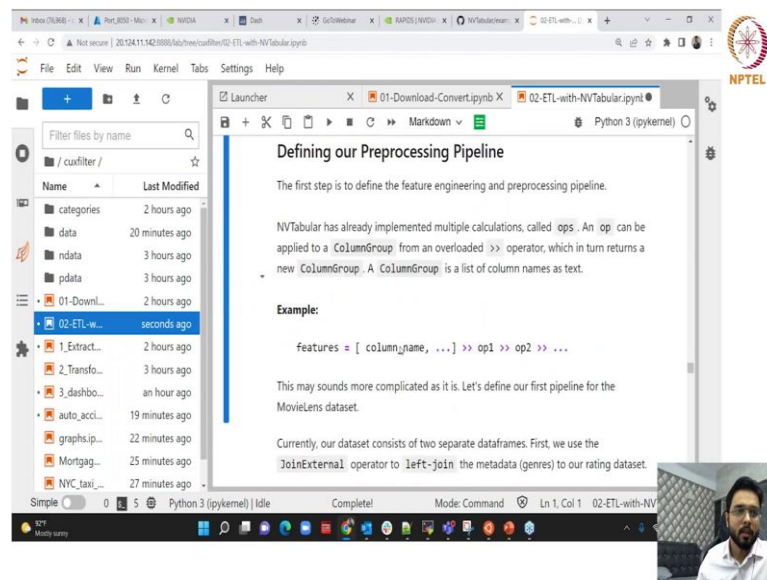
[3]: movies = df_lib.read_parquet(os.path.join(INPUT_DATA_DIR, "movies_converted.parq"))
      movies.head()
```

	movieid	genres
0	1	[Adventure, Animation, Children, Comedy, Fantasy]
1	2	[Adventure, Children, Fantasy]
2	3	[Comedy, Romance]
3	4	[Comedy, Drama, Romance]
4	5	[Comedy]

Defining our Preprocessing Pipeline

Then set up the input data directory, then set up the movies data frame, using df lib parquet. So, this movie id and genres are the columns here.

(Refer Slide Time: 24:01)



Defining our Preprocessing Pipeline

The first step is to define the feature engineering and preprocessing pipeline.

NVTabular has already implemented multiple calculations, called ops. An op can be applied to a ColumnGroup from an overloaded >> operator, which in turn returns a new ColumnGroup. A ColumnGroup is a list of column names as text.

Example:

```
features = [ column_name, ... ] >> op1 >> op2 >> ...
```

This may sound more complicated as it is. Let's define our first pipeline for the MovieLens dataset.

Currently, our dataset consists of two separate dataframes. First, we use the `JoinExternal` operator to `left-join` the metadata (genres) to our rating dataset.

(Refer Slide Time: 24:06)

```
MovieLens dataset.

Currently, our dataset consists of two separate dataframes. First, we use the
JoinExternal operator to left-join the metadata (genres) to our rating dataset.

[4]: CATEGORICAL_COLUMNS = ["userId", "movieId"]
     LABEL_COLUMNS = ["rating"]

[5]: joined = ["userId", "movieId"] >> nvt.ops.JoinExternal(movies, on=["movieId"])

Data pipelines are Directed Acyclic Graphs (DAGs). We can visualize them with
graphviz.

[34]: joined_graph

[34]:
```

[userId, movieId]

And then defining our pre-processing pipeline. Categorical columns are user Id and movie Id; label columns; that means, the target columns are rating and then joined.

(Refer Slide Time: 24:32)

```
[34]:
```

[userId, movieId]

[userId, movieId]

SelectionOp

JoinExternal

output cols=[userId, movieId]

So, how we joined two columns, user Id and movie Id together. So, and we opt so this is the transformation which I was telling the shortcut transformation nvt ops JoinExternal movies with movie Id. So, now, it is joined.

You want to see how the how it worked. So, you can just run `joined.graph`. So, you will see that user Id and movie Id was there, then selected operation, then join operation, then output columns, user Id movie Id came.

(Refer Slide Time: 24:45)

The screenshot shows a Python IDE with a file explorer on the left and a code editor on the right. The code editor contains the following code:

```
[6]: cat_features = joined >> nvt.ops.Categorify()

The ratings are on a scale between 1-5. We want to predict a binary target with 1 for
ratings >3 and 0 for ratings <=3. We use the LambdaOp for it.

[7]: ratings = nvt.ColumnGroup(["rating"]) >> nvt.ops.LambdaOp(lambda col: (col > 3).i

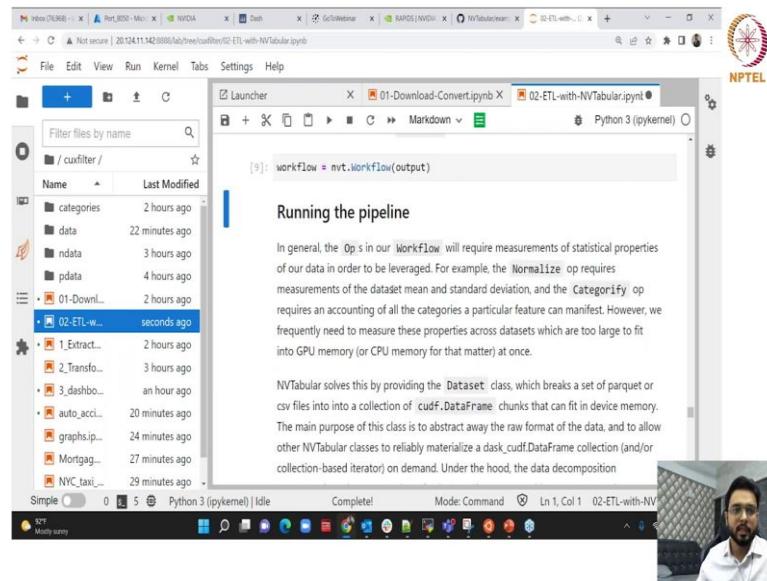
[35]: output = cat_features + ratings
(output).graph
```

Below the code, a diagram illustrates the `SelectionOp` node. It shows an input node labeled `[userId, 'movieId']` with an arrow pointing to a `SelectionOp` node. The output of the `SelectionOp` node is another node labeled `[userId, 'movieId']`.

And then if you want to categorify, because these are numerical values if you want to create categorical values, just uncatagorify you will create categorical values in that column. And then if you want to do some lambda operations so; that means, custom operation. So, col greater than 3, if it is values greater than 3 it becomes a. So, ratings are on scale of 1 to 5, we want to predict a binary target with 1 ratings greater than 3 and 0 for ratings less than 3.

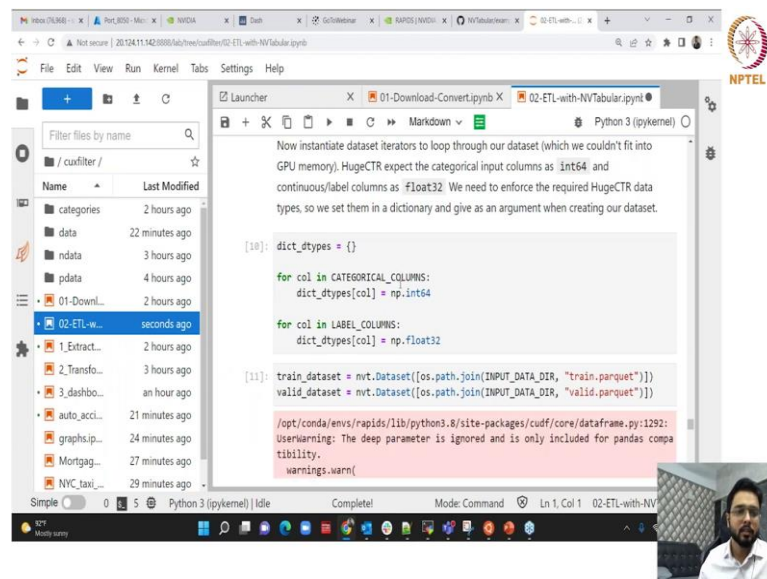
So, again we are again creating type of two categories for 5 values ok. So, based on this you have to create lambda functions. So, simple function to converts 1 to 5 to 1 or 0, based on less than 3 or greater than 3. So, here it is then create workflow `nvt.workflow` (output).

(Refer Slide Time: 25:41)



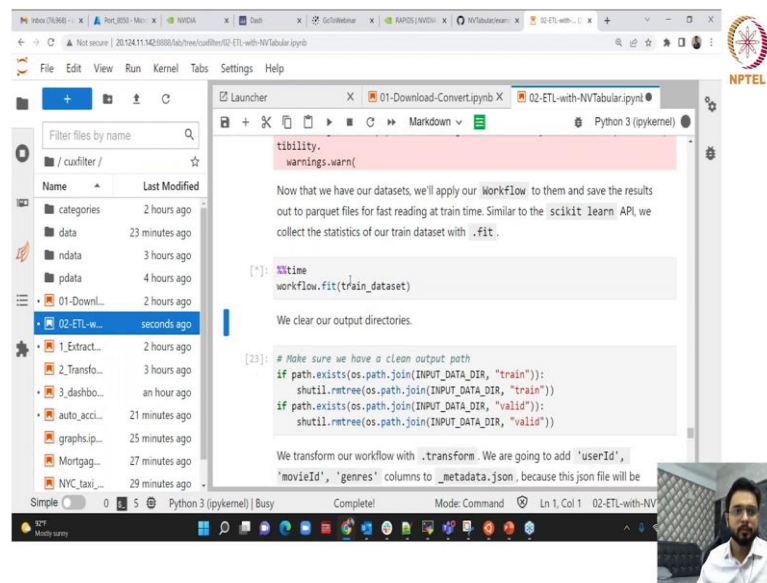
So, this workflow is created and these are all lazy operations again.

(Refer Slide Time: 25:46)



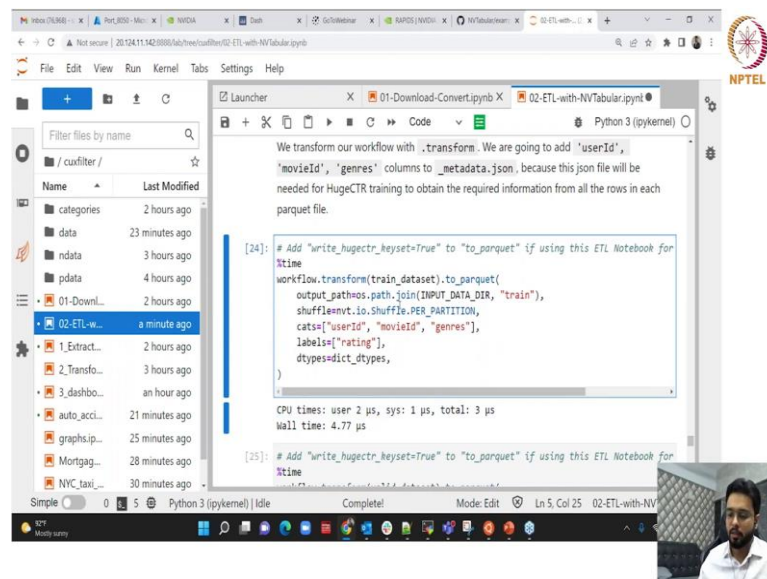
So, the pipeline will run unless until we create the data set and do the fit. So, for column of categorical columns, integer, label columns, float32, then creating the training data set, creating the validation data set.

(Refer Slide Time: 26:04)



I just ignore the warning, then workflow dot fit.

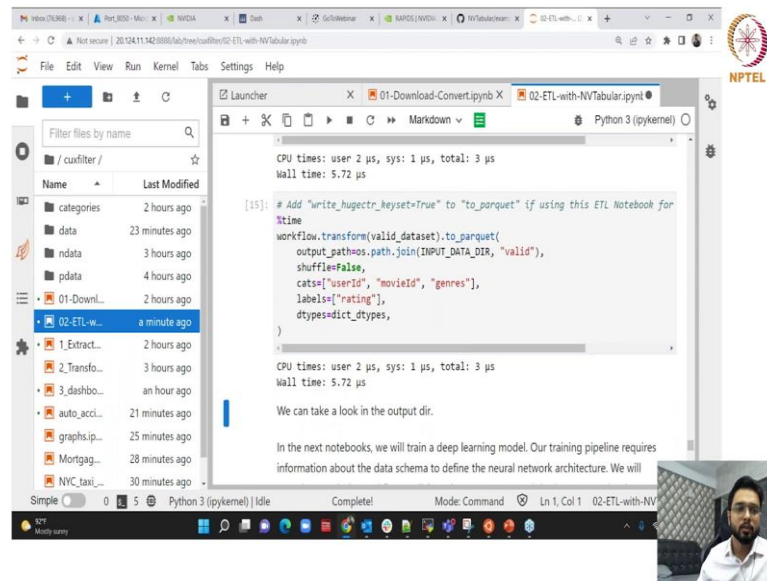
(Refer Slide Time: 26:09)



If path exists then you can create basic clean output path in the local and then add right HugeCTR key set true, to parquet, if this ETL notebook is there for training with HugeCTR.

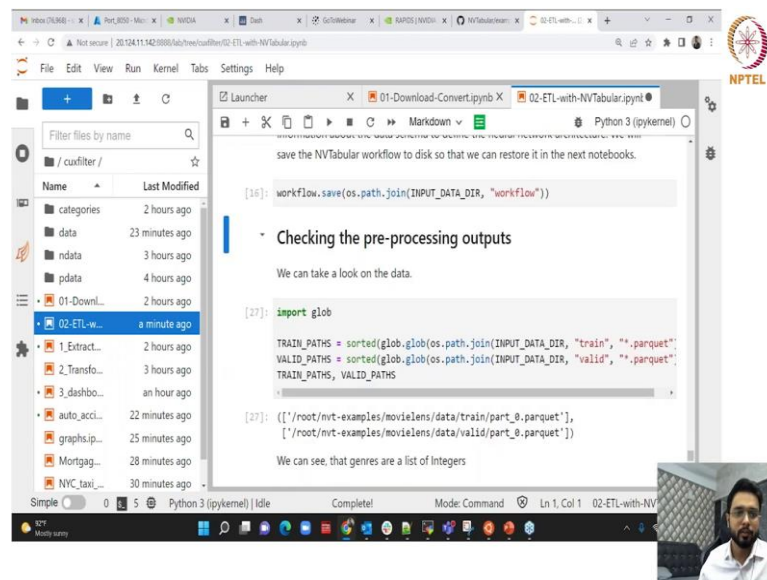
So, this is basically preparing the model for HugeCTR model, though it is not required today because we will mostly focus on the pre-processing part of it using NVTabular.

(Refer Slide Time: 26:31)



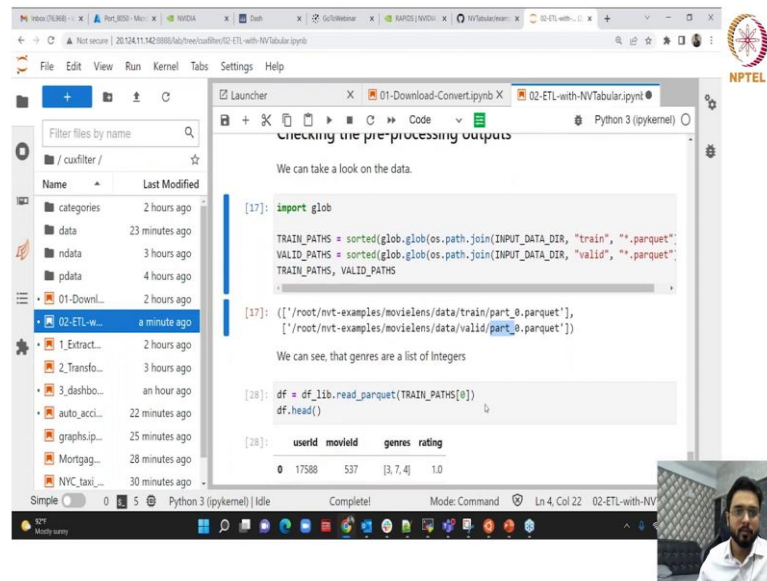
And then finally, transform to parquet output path, the directory you can give, categorical columns, label columns and done.

(Refer Slide Time: 26:47)



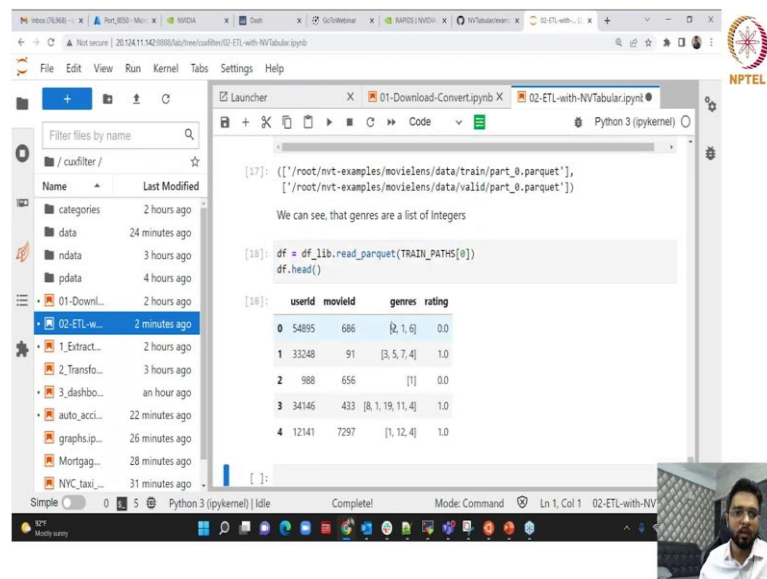
So, entire thing got done in 5.2 microsecond. You can save the workflow, if you want to use it later for different data sets.

(Refer Slide Time: 26:28)



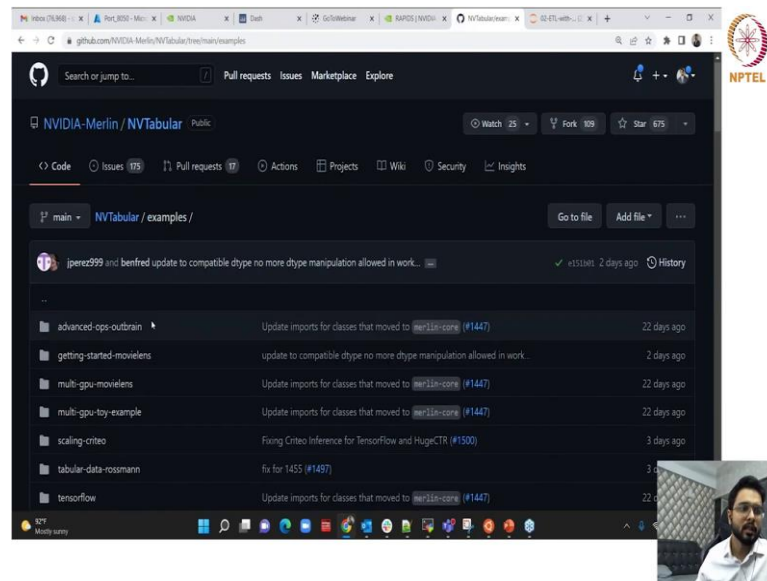
These are the files, if you see parquet part 0 dot parquet being created.

(Refer Slide Time: 27:05)

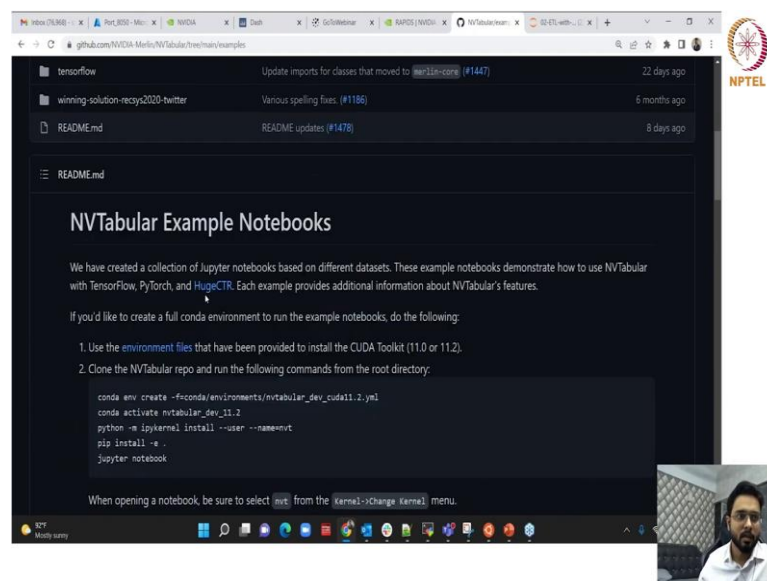


And if you want to see the output so this is the output. User id, movie id, genres, 0 and so on and so forth. Basically, this is the shortcut pre-processing done using NVTabular on GPUs.

(Refer Slide Time: 27:31)

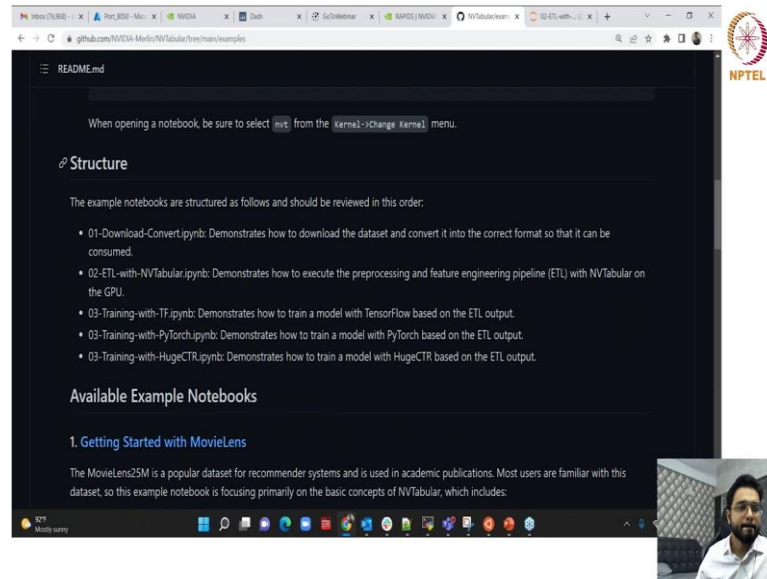


(Refer Slide Time: 27:32)



So, there are some more notebooks you can try in the example repositories. So, there is movie lens, apart of movie lens there is advance operations outbrain, outbrain data, rossmann data, criteo data you can also scale using DASK. So, the DASK intrigrated versions are also there.

(Refer Slide Time: 27:45)



The screenshot shows a web browser displaying the README for NVTabular examples. The page is titled "README.md" and contains the following text:

When opening a notebook, be sure to select `nvx` from the `Kernel->Change Kernel` menu.

Structure

The example notebooks are structured as follows and should be reviewed in this order:

- 01-Download-Convert.ipynb: Demonstrates how to download the dataset and convert it into the correct format so that it can be consumed.
- 02-ETL-with-NVTabular.ipynb: Demonstrates how to execute the preprocessing and feature engineering pipeline (ETL) with NVTabular on the GPU.
- 03-Training-with-TF.ipynb: Demonstrates how to train a model with TensorFlow based on the ETL output.
- 03-Training-with-PyTorch.ipynb: Demonstrates how to train a model with PyTorch based on the ETL output.
- 03-Training-with-HugeCTR.ipynb: Demonstrates how to train a model with HugeCTR based on the ETL output.

Available Example Notebooks

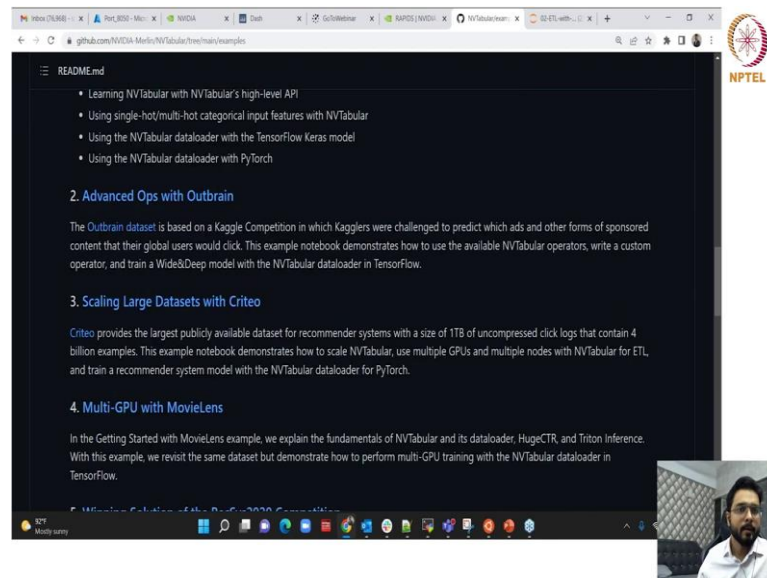
1. Getting Started with MovieLens

The MovieLens25M is a popular dataset for recommender systems and is used in academic publications. Most users are familiar with this dataset, so this example notebook is focusing primarily on the basic concepts of NVTabular, which includes:

The screenshot also shows a small video feed of a man in the bottom right corner and the NPTEL logo in the top right corner.

So, right now I just showed you the simple one node cuDF based, but it can be also integrated with DASK or multi node as well ok.

(Refer Slide Time: 27:57)



The screenshot shows a web browser displaying the README for NVTabular advanced examples. The page is titled "README.md" and contains the following text:

- Learning NVTabular with NVTabular's high-level API
- Using single-hot/multi-hot categorical input features with NVTabular
- Using the NVTabular dataloader with the TensorFlow Keras model
- Using the NVTabular dataloader with PyTorch

2. Advanced Ops with Outbrain

The Outbrain dataset is based on a Kaggle Competition in which Kagglers were challenged to predict which ads and other forms of sponsored content that their global users would click. This example notebook demonstrates how to use the available NVTabular operators, write a custom operator, and train a Wide&Deep model with the NVTabular dataloader in TensorFlow.

3. Scaling Large Datasets with Criteo

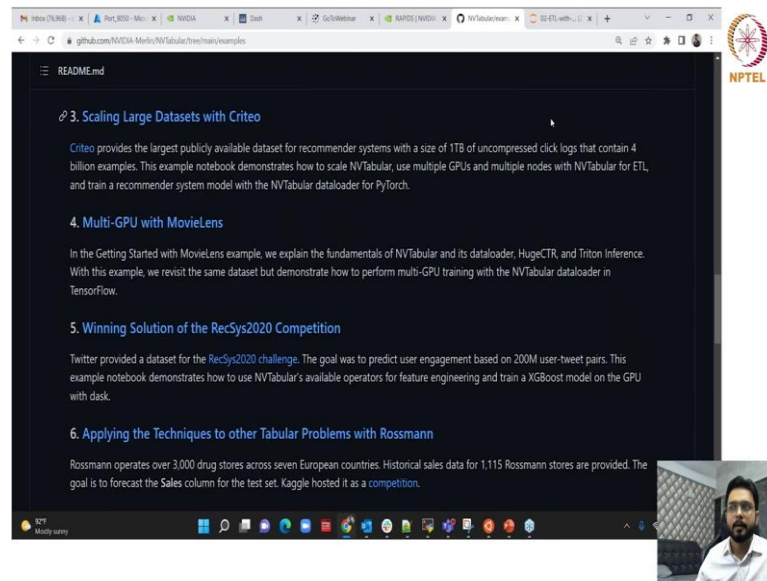
Criteo provides the largest publicly available dataset for recommender systems with a size of 1TB of uncompressed click logs that contain 4 billion examples. This example notebook demonstrates how to scale NVTabular, use multiple GPUs and multiple nodes with NVTabular for ETL, and train a recommender system model with the NVTabular dataloader for PyTorch.

4. Multi-GPU with MovieLens

In the Getting Started with MovieLens example, we explain the fundamentals of NVTabular and its dataloader, HugeCTR, and Triton Inference. With this example, we revisit the same dataset but demonstrate how to perform multi-GPU training with the NVTabular dataloader in TensorFlow.

The screenshot also shows a small video feed of a man in the bottom right corner and the NPTEL logo in the top right corner.

(Refer Slide Time: 27:58)



So, yeah I think this is how NVTabular works and this is the, this is how fast it is as compared to pandas or as compared to Spark or even it is how fast it is to develop code of NVTabular as compared to even DASK on GPU.

(Refer Slide Time: 28:32)



Coming to the learning next steps and learning paths. So, how you can get hands on and you can do something on what you learn today.

So, if there is RAPIDS.AI website you can go and understand in detail, there is Google Colab or NGC links have given in the slides or you can use Conda as well to install. The

Conda commands are also available in rapid AI website which the link to which I have given here.

Then you can go to rapids notebooks, some default notebooks are there. So, I will particularly link the NVTabular GitHub repository here as well and some of the blogs which you can refer, there are a lot of blogs published every day in day out you can refer to that and you can get help from rapids documentation for any syntax or issue help.

And you can also file GitHub report bugs or issue if you see some there are some bugs or issue with our team works day in, day out to resolve those. Spark is something which we will cover in the next session.

(Refer Slide Time: 29:34)

The rapids advantage
How RAPIDS delivers Data Science value

Maximized Productivity	Top Model Accuracy	Lowest TCO	Low Learning Curve	Targeted Acceleration
Massive speedups using RAPIDS with XGBoost	Huge savings and low error rates	Best value for money, low infrastructure costs	Easy integration with familiar APIs	Acceleration of most commonly-used programming languages
Example: 215X speedups at Oak Ridge National Labs	Example: \$1B potential savings and 4% error rate reduction at Global Retail Giant	Example: \$1.5B infrastructure cost savings at Streaming Media Company	Example: Easily replaces popularly-used libraries such as like Pandas and scikit-learn	Example: RAPIDS accelerates Python and SQL workflows which have increasing market share

So, just summarizing that ok what is the advantage you can maximize your data scientists productivity, you can get top model accuracy because you spent more time on optimizing the model rather than pre-processing.

You had a very low total cost of ownership your learning curve was very less, because you do not have to learn anything new, you just used python to learn like your existing knowledge of pandas or sklearn or Python based interfaces. And then using Python or SQL based workflows you did all the things.