Lessons learnt in optimizing a large-scale pandas application using Polars, FireDucks and cuDF: Go Smart and Save More!

Dec 09, 2025, Tuesday

Sourav Saha

Quick Introduction!

SOURAV SAHA – Senior Research Engineer @ NEC Corporation (Japan)

A software professional with 12+ years of working experience at NEC Corporation across diverse areas of **HPC, Vector Supercomputing, Distributed Programming, Big Data and Machine Learning**.

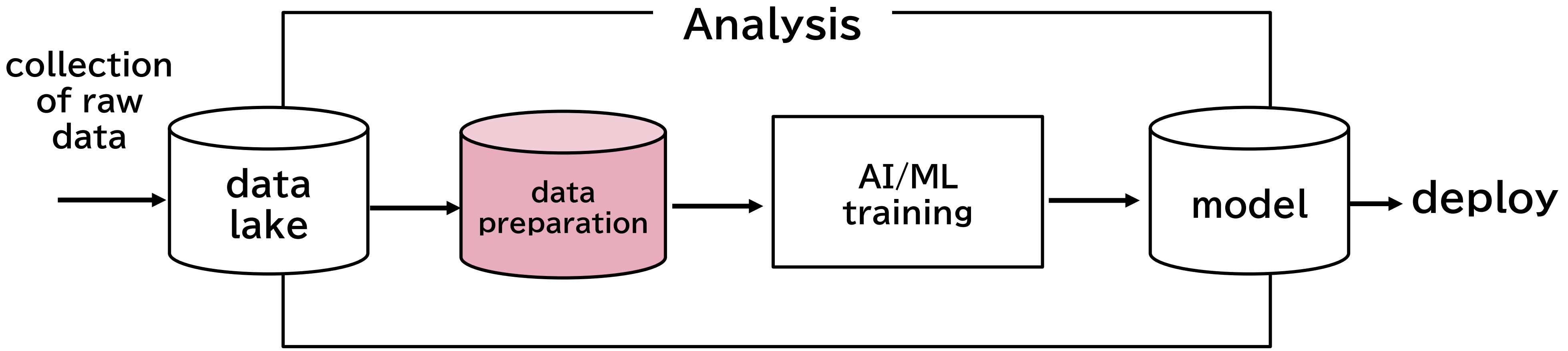
Key area of interest:
Optimization of Data Processing, AI/ML related workflow

Who is this talk for?

- You are a researcher, algorithm developer, system developer who works with data in python almost everyday.
- You are working with large-scale data in pandas.
- The data is related to Financial transactions.
- You are experiencing performance issue (probably related to cloud cost, runtime memory, etc.)
- You are looking for different high-performance pandas alternatives.
- You have already invested in GPU and want to use it for data preparation.

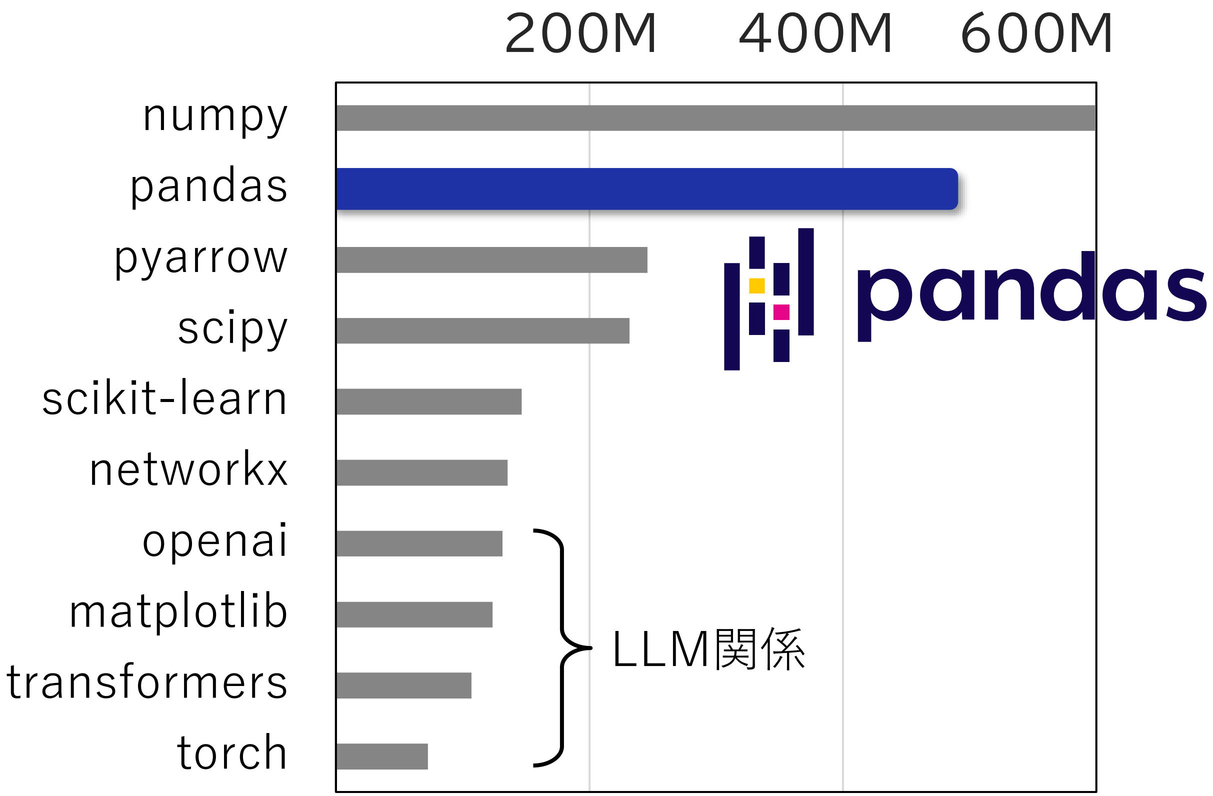
Overview of the Application

- **Application Overview:**
  - Transactional Data Analysis and Feature creation: using pandas
    - **Extremely Slow: Target of today's discussion**
    - I will be using online retail analysis dataset for the sample queries in the upcoming slides.
  - Machine Learning: Market Basket Analysis, Neural Collaborative Filtering etc.



Pandas: the de-facto toolkit for analyzing structured data in Python

The standard Python library for data analysis, downloaded over 400 million times per month



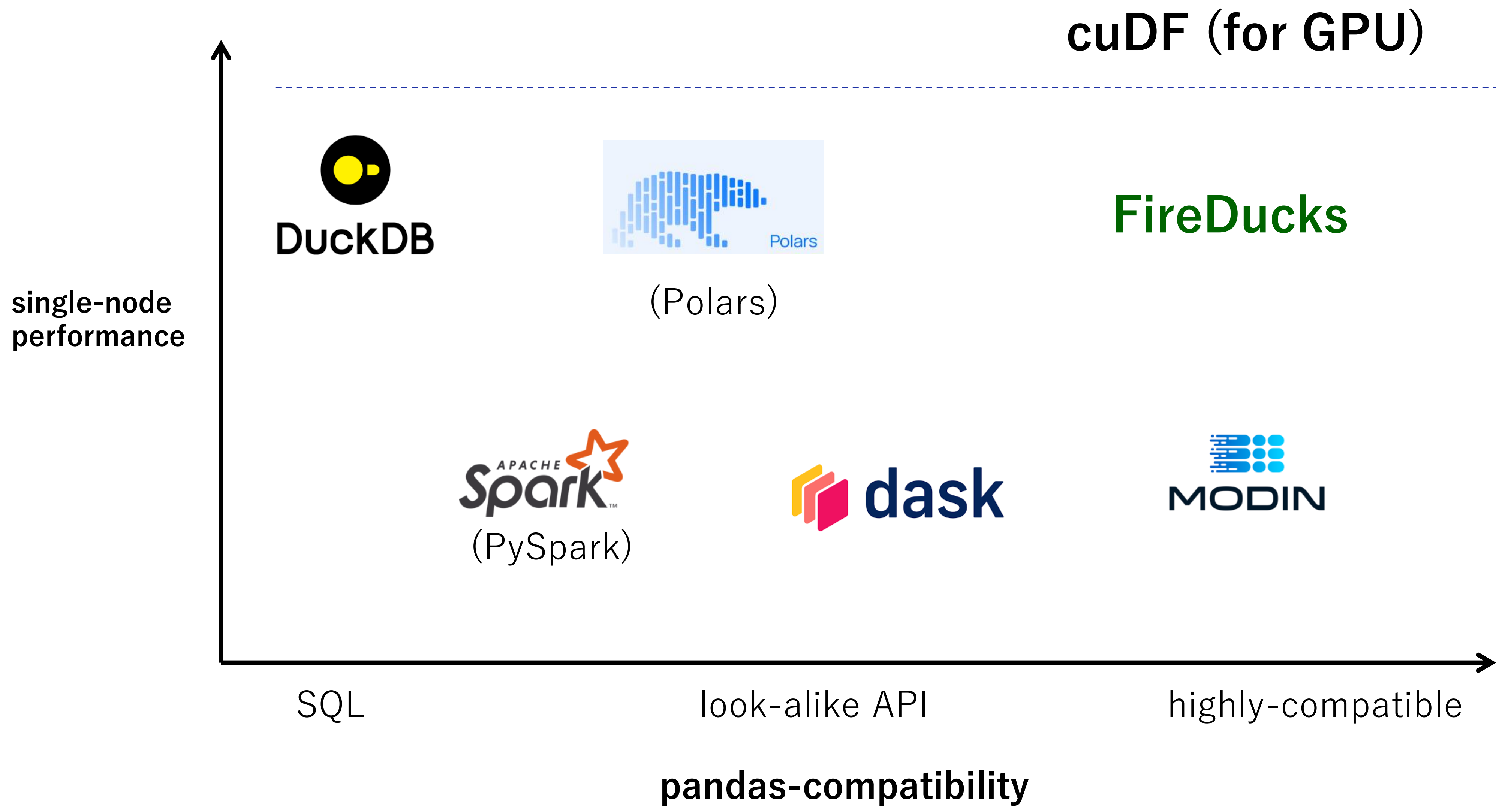
Monthly download from pypi.org（2025/11）
（Data Analytics Libraries）

https://www.amazon.co.jp/s?k=pandas+%E3%83%87%E3%83%BC%E3%82%BF&__mk_ja_JP=%E3%82%AB%E3%82%BF%E3%82%AB%E3%83%8A&crid=T44FOT0ODNPR&sprefix=pandas+%E3%83%87%E3%83%BC%E3%82%BF+%2Caps%2C165&ref=nb_sb_noss_2

https://www.udemy.com/ja/topic/pandas/

With the increase in data volume and complexity, a user starts experiencing performance issues with pandas

Exploring High-performance Pandas Alternatives

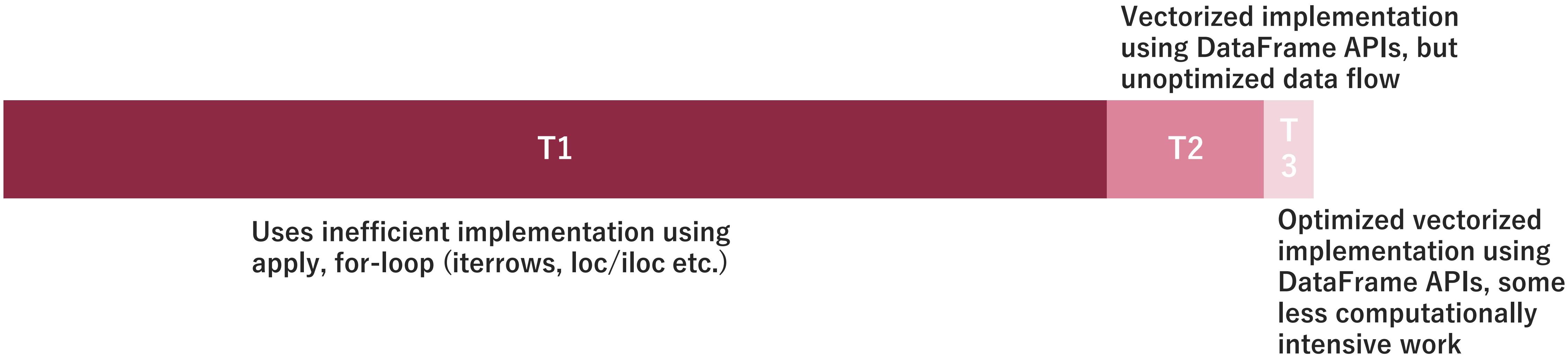


Comparison among Chosen Libraries

	Polars	FireDucks	cuDF
Primary Language	Rust (with Python API)	C++ (with Python API)	C++/CUDA (with Python API)
Execution Model	Lazy + Eager modes	Lazy + Eager modes	Eager only
Query Optimization	Yes	Yes	No
Pandas Compatibility	Not Compatible	Very High	High
Main Strength	Fast CPU analytics, low memory use, intuitive APIs	No code migration, Fast CPU analytics, Low memory use	No code migration, massive speedups for large datasets using GPU parallelism
Hardware Target	CPU (single/multi-threaded), NVIDIA GPU	CPU (single/multi-threaded), NVIDIA GPU (limited to enterprise edition)	NVIDIA GPU
Supported OS	OS Independent	Linux, macOS (Windows via WSL)	Linux Only (Windows via WSL)
License	MIT	3-Caluse BSD	Apache 2.0

Bottleneck Analysis

- During the initial investigation, we found following three bottleneck categories involve in a typical pandas application:
  - **Loop-based implementation (T1)**: Primary bottlenecks covering almost 94% of the total processing time
  - **DataFrame API-based implementation <u>without</u> optimized data flow (T2)**: Secondary bottlenecks covering almost 5% of the total processing time
  - **DataFrame API-based implementation <u>with</u> optimized data flow (T3)**: Not-critical (1%), but encounters performance issue with high-volume data



Bottleneck Analysis

Categories	pandas	FireDucks	cuDF	Polars
Loop-based (T1)	Slow	Slower	Slower	Slower
DataFrame API-based without optimized data-flow (T2)	🚀	🚀🚀🚀🚀	🚀🚀	🚀🚀🚀🚀
DataFrame API-based with optimized data-flow (T3)	🚀🚀	🚀🚀🚀	🚀🚀🚀🚀	🚀🚀🚀

🚀 = baseline speed
🚀🚀 = fast
🚀🚀🚀 = very fast
🚀🚀🚀🚀 = extremely fast

- **T1: Primary bottlenecks covering almost 94% of the total processing time**
  - Due to very inefficient implementation using iterrows, apply etc., the execution is slow in pandas
  - Such implementation hinders parallelism, query optimization etc., adds extra overheads. Therefore, further performance degradation is observed even from high-performance pandas alternatives.
- **T2: Secondary bottlenecks covering almost 5% of the total processing time**
  - Although implemented using vectorized dataframe APIs, the unoptimized data flow causes a lot of unnecessary computation overhead. Therefore, libraries with eager execution model, like pandas, cuDF are slower.
  - Whereas, libraries with query optimization support, like FireDucks, Polars excel in these areas.
- **T3: Not-critical (1%), but encounters performance issue with high-volume data**
  - Pandas works well for these cases, but experiences performance issue with high-volume data (due to its single-threaded execution model)
  - All other alternatives: cuDF, Polars, FireDucks work extremely well for such cases even with high-volume data

Exploring Type-1 Bottlenecks
(Loop-based implementation)

Query 01: Problem Statement

- **Fill missing values of “Description” column using the most frequent description of the specific “StockCode”.**

```
InvoiceNo StockCode Description  Quantity         InvoiceDate  UnitPrice  CustomerID         Country
   536414     22139        <NA>        56 2010-12-01 11:52:00        0.0         NaN  United Kingdom
   536545     21134        <NA>         1 2010-12-01 14:32:00        0.0         NaN  United Kingdom
   536546     22145        <NA>         1 2010-12-01 14:33:00        0.0         NaN  United Kingdom
   536547     37509        <NA>         1 2010-12-01 14:33:00        0.0         NaN  United Kingdom
   536549    85226A        <NA>         1 2010-12-01 14:34:00        0.0         NaN  United Kingdom
      ...       ...         ...       ...                 ...        ...         ...             ...
   581199     84581        <NA>        -2 2011-12-07 18:26:00        0.0         NaN  United Kingdom
   581203     23406        <NA>        15 2011-12-07 18:31:00        0.0         NaN  United Kingdom
   581209     21620        <NA>         6 2011-12-07 18:35:00        0.0         NaN  United Kingdom
   581234     72817        <NA>        27 2011-12-08 10:33:00        0.0         NaN  United Kingdom
   581408     85175        <NA>        20 2011-12-08 14:06:00        0.0         NaN  United Kingdom
```

```
>>> df[df["StockCode"] == "22139"]["Description"].value_counts()
Description
RETROSPOT TEA SET CERAMIC 11 PC     988
amazon                                1
Name: count, dtype: Int64
>>> df[df["StockCode"] == "21134"]["Description"].value_counts()
Series([], Name: count, dtype: Int64)
>>> df[df["StockCode"] == "22145"]["Description"].value_counts()
Description
CHRISTMAS CRAFT HEART STOCKING     1
Name: count, dtype: Int64
```

```
InvoiceNo StockCode                      Description  Quantity         InvoiceDate  UnitPrice  CustomerID         Country
   536414     22139     RETROSPOT TEA SET CERAMIC 11 PC        56 2010-12-01 11:52:00        0.0         NaN  United Kingdom
   536545     21134                                 <NA>         1 2010-12-01 14:32:00        0.0         NaN  United Kingdom
   536546     22145      CHRISTMAS CRAFT HEART STOCKING         1 2010-12-01 14:33:00        0.0         NaN  United Kingdom
   536547     37509           NEW ENGLAND MUG W GIFT BOX         1 2010-12-01 14:33:00        0.0         NaN  United Kingdom
   536549    85226A                                 <NA>         1 2010-12-01 14:34:00        0.0         NaN  United Kingdom
      ...       ...                                  ...       ...                 ...        ...         ...             ...
   581199     84581       DOG TOY WITH PINK CROCHET SKIRT        -2 2011-12-07 18:26:00        0.0         NaN  United Kingdom
   581203     23406            HOME SWEET HOME KEY HOLDER        15 2011-12-07 18:31:00        0.0         NaN  United Kingdom
   581209     21620       SET OF 4 ROSE BOTANICAL CANDLES         6 2011-12-07 18:35:00        0.0         NaN  United Kingdom
   581234     72817   SET OF 2 CHRISTMAS DECOUPAGE CANDLE        27 2011-12-08 10:33:00        0.0         NaN  United Kingdom
   581408     85175                 CACTI T-LIGHT CANDLES        20 2011-12-08 14:06:00        0.0         NaN  United Kingdom
```

Query 01: implementation using iterrows

```
df_descr_null = df[df["Description"].isnull()]

for index, row in df_descr_null.iterrows():
  stock_code = row["StockCode"]
  # Find the most frequent description for this StockCode
  most_frequent_description = df[df["StockCode"] == stock_code]["Description"].mode()
  if not most_frequent_description.empty:
     df.at[index, "Description"] = most_frequent_description[0]
```

Loop-based traditional approach

index	StockCode		Description
622	22139	get_mode(22139) →	RETROSPOT TEA SET CERAMIC 11 PC
1970	21134	get_mode(21134) →	NA
1971	22145	get_mode(22145) →	CHRISTMAS CRAFT HEART STOCKING
1972	37509	get_mode(37509) →	NEW ENGLAND MUG W GIFT BOX
1987	85226A	get_mode(85226A) →	NA
:	:		
538554	85175	get_mode(85175) →	CACTI T-LIGHT CANDLES

For a sample data with 7270 null descriptions, pandas took ~930 sec.

Query 01: modified implementation using vectorized APIs

```
# Find “StockCode” wise most-frequent element of “Description”
most_freq = (
      df[["StockCode", "Description"]]
      .value_counts()
      .reset_index()
      .groupby("StockCode")
      .head(1)
)
most_freq.columns = ["StockCode", “FreqDescription", "frequency"]

# transform the “Description” column with the most-frequent description
tmp = df.merge(most_freq, on="StockCode", how="left")

# fill nulls with the most-frequent description
df["Description"] = df["Description"].fillna(tmp[“FreqDescription"])
```

vectorized approach

group-wise count of unique descriptions

```
StockCode  Description
10002      INFLATABLE POLITICAL GLOBE      71
10080      GROOVY CACTUS INFLATABLE        22
           check                            1
10120      DOGGY RUBBER                    30
10123C     HEARTS WRAPPING TAPE             3
10124A     SPOTS ON RED BOOKCOVER TAPE      5
10124G     ARMY CAMO BOOKCOVER TAPE         4
10125      MINI FUNKY DESIGN TAPES         94
10133      COLOURING PENCILS BROWN TUBE   199
           damaged                          1
```

group-wise most frequent description

```
StockCode  Description
10002      INFLATABLE POLITICAL GLOBE      71
10080      GROOVY CACTUS INFLATABLE        22
10120      DOGGY RUBBER                    30
10123C     HEARTS WRAPPING TAPE             3
10124A     SPOTS ON RED BOOKCOVER TAPE      5
10124G     ARMY CAMO BOOKCOVER TAPE         4
10125      MINI FUNKY DESIGN TAPES         94
10133      COLOURING PENCILS BROWN TUBE   199
```

merge with most frequent description

```
StockCode Description             FreqDescription
    10002        <NA>  INFLATABLE POLITICAL GLOBE
    10002        <NA>  INFLATABLE POLITICAL GLOBE
   10123C        <NA>        HEARTS WRAPPING TAPE
    10080        <NA>    GROOVY CACTUS INFLATABLE
```

For a sample data with 7270 null descriptions

pandas	cuDF	polars	FireDucks
1.4s	2.4s	0.6s	0.5s

Query 02: Problem Statement

- **Find the number of transactions a user performed within the N days (e.g., 90) of the current transaction**

CustomerID	InvoiceNo	InvoiceDate
12347.0	537626	2010-12-07 14:57:00
12347.0	542237	2011-01-26 14:30:00
12347.0	549222	2011-04-07 10:43:00
12347.0	556201	2011-06-09 13:01:00
12347.0	562032	2011-08-02 08:48:00
12347.0	573511	2011-10-31 12:25:00
12347.0	581180	2011-12-07 15:52:00

CustomerID	InvoiceNo	InvoiceDate	count
12347.0	537626	2010-12-07 14:57:00	2
12347.0	542237	2011-01-26 14:30:00	3
12347.0	549222	2011-04-07 10:43:00	3
12347.0	556201	2011-06-09 13:01:00	3
12347.0	562032	2011-08-02 08:48:00	2
12347.0	573511	2011-10-31 12:25:00	2
12347.0	581180	2011-12-07 15:52:00	2

Query 02: implementation using row-wise apply

```
 def helper(x, offset=90):
       fdf = df[
           (df["CustomerID"] == x["CustomerID"]) &
           (abs(df["InvoiceDate"] - x["InvoiceDate"]) <= timedelta(days=offset))
       ]
       return len(fdf)

ret = df.assign(count=lambda data: data.apply(helper, axis=1))[
       ["CustomerID", “InvoiceDate”, "count"]
]
```

apply-based traditional approach

For a sample data with 110950 unique transactions made by 4372 unique customers, pandas took ~134 sec.

```
CustomerID  InvoiceNo          InvoiceDate
   12347.0     537626 2010-12-07 14:57:00
   12347.0     542237 2011-01-26 14:30:00
   12347.0     549222 2011-04-07 10:43:00
   12347.0     556201 2011-06-09 13:01:00
   12347.0     562032 2011-08-02 08:48:00
   12347.0     573511 2011-10-31 12:25:00
   12347.0     581180 2011-12-07 15:52:00
```

InvoiceDate	TimeDiff
2010-12-07 14:57:00	237 days 17:51:00
2011-01-26 14:30:00	187 days 18:18:00
2011-04-07 10:43:00	116 days 22:05:00
2011-06-09 13:01:00	**53 days 19:47:00**
2011-08-02 08:48:00	**0 days 00:00:00**
2011-10-31 12:25:00	90 days 03:37:00
2011-12-07 15:52:00	127 days 07:04:00

Count
2

Query 02: implementation using merge+filter

Vectorized approach

```
# self-join for pair-wise comparison
m = df.merge(df, on="CustomerID")
delta = timedelta(days=offset)

# filter the target pairs
m[”TimeDiff”] = (
  abs(m["InvoiceDate_x"] - m["InvoiceDate_y"])
)

# perform groupby-aggregate
ret = (
  m[m[“TimeDiff”] <= delta]
    .groupby(["CustomerID", "InvoiceDate_x"])
     ["InvoiceNo_y“].count().reset_index()
)
ret.columns = ["CustomerID", "InvoiceDate", "count"]
```

pandas	cuDF	polars	FireDucks
0.9s	3.5s	0.3s	0.4s

pair construction

```
CustomerID InvoiceNo_x         InvoiceDate_x InvoiceNo_y         InvoiceDate_y
   12347.0      537626 2010-12-07 14:57:00      537626 2010-12-07 14:57:00
   12347.0      537626 2010-12-07 14:57:00      542237 2011-01-26 14:30:00
   12347.0      537626 2010-12-07 14:57:00      549222 2011-04-07 10:43:00
   12347.0      537626 2010-12-07 14:57:00      556201 2011-06-09 13:01:00
   12347.0      537626 2010-12-07 14:57:00      562032 2011-08-02 08:48:00
   12347.0      537626 2010-12-07 14:57:00      573511 2011-10-31 12:25:00
   12347.0      537626 2010-12-07 14:57:00      581180 2011-12-07 15:52:00
   12347.0      542237 2011-01-26 14:30:00      537626 2010-12-07 14:57:00
   12347.0      542237 2011-01-26 14:30:00      542237 2011-01-26 14:30:00
   12347.0      542237 2011-01-26 14:30:00      549222 2011-04-07 10:43:00
   12347.0      542237 2011-01-26 14:30:00      556201 2011-06-09 13:01:00
   12347.0      542237 2011-01-26 14:30:00      562032 2011-08-02 08:48:00
```

pair-wise diff calculation

```
CustomerID InvoiceNo_x         InvoiceDate_x InvoiceNo_y         InvoiceDate_y          TimeDiff
   12347.0      537626 2010-12-07 14:57:00      537626 2010-12-07 14:57:00   0 days 00:00:00
   12347.0      537626 2010-12-07 14:57:00      542237 2011-01-26 14:30:00  49 days 23:33:00
   12347.0      537626 2010-12-07 14:57:00      549222 2011-04-07 10:43:00 120 days 19:46:00
   12347.0      537626 2010-12-07 14:57:00      556201 2011-06-09 13:01:00 183 days 22:04:00
   12347.0      537626 2010-12-07 14:57:00      562032 2011-08-02 08:48:00 237 days 17:51:00
   12347.0      537626 2010-12-07 14:57:00      573511 2011-10-31 12:25:00 327 days 21:28:00
   12347.0      537626 2010-12-07 14:57:00      581180 2011-12-07 15:52:00 365 days 00:55:00
   12347.0      542237 2011-01-26 14:30:00      537626 2010-12-07 14:57:00  49 days 23:33:00
   12347.0      542237 2011-01-26 14:30:00      542237 2011-01-26 14:30:00   0 days 00:00:00
   12347.0      542237 2011-01-26 14:30:00      549222 2011-04-07 10:43:00  70 days 20:13:00
   12347.0      542237 2011-01-26 14:30:00      556201 2011-06-09 13:01:00 133 days 22:31:00
   12347.0      542237 2011-01-26 14:30:00      562032 2011-08-02 08:48:00 187 days 18:18:00
```

Filtration of target pairs

```
CustomerID InvoiceNo_x         InvoiceDate_x InvoiceNo_y         InvoiceDate_y         TimeDiff
   12347.0      537626 2010-12-07 14:57:00      537626 2010-12-07 14:57:00  0 days 00:00:00
   12347.0      537626 2010-12-07 14:57:00      542237 2011-01-26 14:30:00 49 days 23:33:00
   12347.0      542237 2011-01-26 14:30:00      537626 2010-12-07 14:57:00 49 days 23:33:00
   12347.0      542237 2011-01-26 14:30:00      542237 2011-01-26 14:30:00  0 days 00:00:00
   12347.0      542237 2011-01-26 14:30:00      549222 2011-04-07 10:43:00 70 days 20:13:00
```

Query 03: Problem Statement

- **Calculate the total sales per Invoice for each Customer.**

CustomerID	InvoiceNo	StockCode	Quantity	UnitPrice
17850.0	536365	85123A	6	2.55
17850.0	536365	71053	6	3.39
17850.0	536365	84406B	8	2.75
17850.0	536365	84029G	6	3.39
17850.0	536365	84029E	6	3.39
...	...	...	...	...
12680.0	581587	22613	12	0.85
12680.0	581587	22899	6	2.10
12680.0	581587	23254	4	4.15
12680.0	581587	23255	4	4.15
12680.0	581587	22138	3	4.95

CustomerID	InvoiceNo	
12346.0	541431	385918.00
	C541433	-385918.00
12347.0	537626	3558.95
	542237	2376.95
	549222	3181.25
		...
18283.0	579673	1118.05
	580872	1040.00
18287.0	554065	3826.40
	570715	5006.60
	573167	353.40

Query 03: apply-based vs vectorized implementation

apply-based approach

```
ret = df.groupby(["CustomerID", "InvoiceNo"]).apply(
        lambda x: (x["Quantity"] * x["UnitPrice"]).sum()
)
```

For a sample data with 2709545 transactions made by 4372 unique customers, pandas took ~3.4 sec.

vectorized approach

```
ret = (
        df.assign(revenue=lambda x: x["Quantity"] * x["UnitPrice"])
        .groupby(["CustomerID", "InvoiceNo"])["revenue"]
        .sum()
)
```

pandas	cuDF	polars	FireDucks
378 ms	29 ms	80 ms	64 ms

Exploring Type-2 Bottlenecks
(Vectorized implementation without optimized data flow)

Query 04: Problem Statement

- **Find the list of customers from the "North America" (NA) region who involve in at least one refund related transaction**

	CustomerID	StockCode	InvoiceDate	Quantity	InvoiceDate_y	Quantity_y	region
	12607.0	22551	2011-10-10 16:06:00	12	2011-10-12 16:17:00	-12	NA
	12607.0	21915	2011-10-10 16:06:00	12	2011-10-12 16:17:00	-12	NA
	12607.0	22619	2011-10-10 16:06:00	4	2011-10-12 16:17:00	-4	NA
	12607.0	22138	2011-10-10 16:06:00	3	2011-10-12 16:17:00	-3	NA
	12607.0	21524	2011-10-10 16:06:00	2	2011-10-12 16:17:00	-2	NA
	...	...	...	...	...	...	...
	12558.0	84828	2011-12-02 10:41:00	24	2011-12-08 10:14:00	-24	NA
	12558.0	23158	2011-12-02 10:41:00	36	2011-12-08 10:14:00	-36	NA
	12558.0	21507	2011-12-02 10:41:00	12	2011-12-08 10:14:00	-12	NA
	12558.0	21508	2011-12-02 10:41:00	12	2011-12-08 10:14:00	-12	NA
	12558.0	22027	2011-12-02 10:41:00	12	2011-12-08 10:14:00	-12	NA

Query 04: Vectorized implementation (unoptimized data flow)

```
ret = (
  left.merge(right, on=["CustomerID", "StockCode"])
      .pipe(lambda m: m[m["Quantity"] > 0])
      .pipe(lambda m: m[m["Quantity_y"] < 0])
      .pipe(lambda m: m[m["region"] == "NA"])
      .groupby("CustomerID")["InvoiceNo"]
      .nunique()
)
```

For a sample data with 2709545 transactions made by 4372 unique customers, the execution times are as follows:

pandas	cuDF	polars	FireDucks
67 s	42 s	0.15 s	0.08 s

- Vectorized DataFrame API-based implementation
- But the flow is not optimized!!
  - It combines two large tables and filters out the unrequired rows before performing the groupby.
- An optimized flow could be:
  - Filter required rows and columns from the left and right tables
  - Perform the merge, groupby, etc.
- **Library without query optimization, like <u>pandas, cuDF</u> suffer from performance issues.**
- **Whereas, library with query optimization, like <u>FireDucks, Polars</u>, etc. outperform in such cases.**

Query 04: Vectorized implementation (optimized data flow)

```
ret = (
  left.merge(right, on=["CustomerID", "StockCode"])
      .pipe(lambda m: m[m["Quantity"] > 0])
      .pipe(lambda m: m[m["Quantity_y"] < 0])
      .pipe(lambda m: m[m["region"] == "NA"])
      .groupby("CustomerID")["InvoiceNo"]
      .nunique()
)
```

```
left = left.pipe(lambda m: m[m["region"] == "NA"]).pipe(
      lambda m: m[m["Quantity"] > 0]
)[["CustomerID", "StockCode", "InvoiceNo"]]

right = right.pipe(lambda m: m[m["Quantity_y"] < 0])[["CustomerID",
"StockCode"]]

ret = (
  left.merge(right, on=["CustomerID", "StockCode"])
      .groupby("CustomerID")["InvoiceNo"]
      .nunique()
)
```

For a sample data with 2709545 transactions made by 4372 unique customers, the execution times are as follows:

pandas	cuDF	polars	FireDucks
67 s	42 s	0.15 s	0.08 s

significant performance gain (pandas, cuDF)

Ignorable differences (polars, FireDucks)

pandas	cuDF	polars	FireDucks
0.77 s	1.02 s	0.18 s	0.11 s

Exploring Type-3 Bottlenecks
(Vectorized implementation with optimized data flow)

Query 05: Problem Statement

- **Calculate country and month wise total revenue.**

Country	InvoiceDate	Quantity	UnitPrice
United Kingdom	2010-12-01 08:26:00	6	2.55
United Kingdom	2010-12-01 08:26:00	6	3.39
United Kingdom	2010-12-01 08:26:00	8	2.75
United Kingdom	2010-12-01 08:26:00	6	3.39
United Kingdom	2010-12-01 08:26:00	6	3.39
...	...	...	...
France	2011-12-09 12:50:00	12	0.85
France	2011-12-09 12:50:00	6	2.10
France	2011-12-09 12:50:00	4	4.15
France	2011-12-09 12:50:00	4	4.15
France	2011-12-09 12:50:00	3	4.95

Country	month	revenue
Australia	1	45088.55
Australia	2	73477.10
Australia	3	86119.95
Australia	4	3858.00
Australia	5	68192.05
...	...	...
Unspecified	6	928.90
Unspecified	7	7127.05
Unspecified	8	2655.15
Unspecified	9	1434.25
Unspecified	11	4828.75

Query 05: Vectorized implementation (optimized data flow)

```
ret = (
        df.pipe(lambda x: x[x["Quantity"] > 0])
        .pipe(lambda x: x[x["UnitPrice"] > 0])
        .assign(revenue=lambda x: x["Quantity"] * x["UnitPrice"])
        .assign(month=lambda x: x["InvoiceDate"].dt.month)
        .groupby(["Country", "month"], as_index=False)["revenue"]
        .sum()
   )
```

For a sample data with 2709545 transactions made by 38 unique countries, the execution times are as follows:

pandas	cuDF	polars	FireDucks
0.74 s	0.09 s	0.18 s	0.12 s

- Efficient Query implementation!!
  - Pandas work well for this case.
  - Polars/FireDucks perform better due to multi-threaded execution
  - **cuDF performs the best** due to GPU level parallelism.

Learning Summary

Learning #1: Breaking out of the loop

- Most of the pandas applications suffers from Type-1 bottlenecks (loop-based traditional implementation)
- If such bottleneck exists, performance gain might not be expected even from high-performance pandas alternatives, like cuDF, FireDucks etc.
- Hence before migrating to different library/execution platform, it would be better to re-implement the bottlenecks using pandas specific vectorized implementation

Query	ASIS	Type-1 Optimized
Q01	929.47	1.39
Q02	133.62	0.93
Q03	3.39	0.38
Q04	66.66	67.24
Q05	0.71	0.74
Total	**1133.85**	**70.68**

Pandas itself could be made ~**16x faster** just by making efficient changes in the traditional loop-based implementation

Learning #2: Single-node processing might be enough

SF=1

Query	Type-1 Optimized			
	pandas	**cuDF**	**Polars**	**FireDucks**
Q01	1.39	2.36	0.62	0.54
Q02	0.93	3.52	0.32	0.43
Q03	0.38	0.03	0.08	0.06
Q04	67.24	42.24	0.15	0.08
Q05	0.74	0.10	0.18	0.12
Total	**70.68**	**48.25**	**1.34**	**1.24**

cuDF: ~1.5x speedup

- Only GPU parallelism, no query optimization
- Absolute zero code modification

Polars: ~50x speedup

- High migration cost: Re-writing required even for Type-2 bottlenecks (pandas -> polars)

FireDucks: ~54x speedup

- Absolute zero code modification

SF=10

Query	Type-1 Optimized			
	pandas	**cuDF**	**Polars**	**FireDucks**
Q01	14.06	23.57	3.39	3.21
Q02	67.28	mem-error	10.60	12.02
Q03	3.48	17.57	0.48	0.17
Q04	mem-error	mem-error	1.13	0.40
Q05	6.76	0.77	1.22	1.06
Total	**91.58**	**41.90**	**16.83**	**16.87**

cuDF:

- Memory error might occur due to Type-2 bottlenecks that cuDF cannot auto-optimize.

FireDucks/Polars:

- Capable of auto optimization for Type-2 bottlenecks.
- Hence, work smoothly even for high-volume data

Learning #3: FireDucks might be the one you are looking for!

- **Pandas accelerator without much code alternation**

```
$ python demo.py
Code block '[pandas] Execution time of func_1: ' took: 1.40680 s
Code block '[pandas] Execution time of func_2: ' took: 0.96898 s
Code block '[pandas] Execution time of func_3: ' took: 0.42458 s
Code block '[pandas] Execution time of func_4: ' took: 66.99391 s
Code block '[pandas] Execution time of func_5: ' took: 0.77127 s
```

```
$ python -mcudf.pandas demo.py
Code block '[cudf] Execution time of func_1: ' took: 2.32772 s
Code block '[cudf] Execution time of func_2: ' took: 3.5178 s
Code block '[cudf] Execution time of func_3: ' took: 0.02973 s
Code block '[cudf] Execution time of func_4: ' took: 42.30631 s
Code block '[cudf] Execution time of func_5: ' took: 0.09402 s
```

```
$ python -mfireducks.pandas demo.py
Code block '[fireducks] Execution time of func_1: ' took: 0.59689 s
Code block '[fireducks] Execution time of func_2: ' took: 0.40727 s
Code block '[fireducks] Execution time of func_3: ' took: 0.06636 s
Code block '[fireducks] Execution time of func_4: ' took: 0.09151 s
Code block '[fireducks] Execution time of func_5: ' took: 0.11985 s
```

- For an existing pandas based application, or the one that works with external libraries expecting pandas DataFrame as input, **FireDucks can be an excellent choice.**
- Just refrain from traditional loop-based (T1) implementation and enjoy high-performance processing speed for all your data intensive workload with FireDucks.

Thank you!

Presentation Deck

Connect me!