The Ultimate Guide to Data Wrangling with Python | Rust Polars Data Frame

---

Download PDF version of this notebook

Video Tutorials


```
Author: Amit Shukla
```

https://github.com/AmitXShukla

https://x.com/ashuklax

https://youtube.com/@Amit.Shukla

The aim of this comprehensive user guide is to equip you with all the necessary knowledge and skills required to utilize Python Polars Data Frame effectively for financial and supply chain data science analytics.

It provides an in-depth overview of the most commonly used functions and capabilities of the package.

Introduction

I'm Amit Shukla, and I specialize in training neural networks for finance supply chain analysis, enabling them to identify data patterns and make accurate predictions. During the challenges posed by the COVID-19 pandemic, I successfully trained GL and Supply Chain neural networks to anticipate supply chain shortages. The valuable insights gained from this effort have significantly influenced the content of this tutorial series.

Objective:

By delving into this powerful tool, we will master the fundamental techniques of Data Wrangling. This knowledge is crucial in preparing finance and supply chain data for advanced analytics, visualization, and predictive modeling using neural networks and machine learning.

Subject

It's important to note that this particular series will concentrate solely on `Data Wrangling`. Throughout this series, our main focus will be on learning `Rust Polars Data Frame Python library`.

Following

However, in future installments, we will explore Data Analytics and delve into the realm of machine learning for predictive analytics. Thank you for joining me, and I'm excited to embark on this educational journey together.

Let's get started.

Table of content

Why another DataFrame

Despite the numerous state-of-the-art dataframe packages available in the market, the Polar dataframe, which is built on RUST, boasts the fastest execution speed, enabling it to handle complex data science operations on tabular datasets.

- Execution on larger-than-memory (RAM) data analytics
- Lazy API vs Eager execution
- Automatic Optimization
- Embarrassingly Parallel
- Easy to learn consistent, predictable API that has strict schema
- SQLs like expressions

Efficient Execution of Analytics on Large-than-Memory (RAM) Data

RAM is not a big deal these days as most computers and VMs offer inexpensive GBs of RAM. In fact, the availability of affordable RAM is the primary reason why Pandas-like DataFrames remain the go-to choice, and it is unlikely that Pandas or R Tables will become obsolete anytime soon.

However, Polars DataFrames are increasingly gaining popularity among developers due to their ability to harness the horsepower of Apache Spark, the backend support of DuckDB and Apache Arrow, and the ease-of-use of Pandas-like data frame functionalities.

Additionally, Polars comes with built-in multi-core, multi-threaded parallel processing, making it a highly preferred choice.

Lazy API vs Eager execution

Just because an API is referred to as "lazy" does not necessarily imply that there will be a delay in processing or execution, and conversely, "eager" execution doesn't necessarily mean that the programming language will process data transformations or begin execution immediately and more quickly.

In simpler terms, using a Lazy API implies that the API will first take the time to optimize the query before execution, which often results in improved performance.

To illustrate this concept, consider running SQL on an RDBMS database. If the statistics, indexes, and data partitions have been appropriately optimized and the SQL is written in an optimized manner that utilizes the available statistics, indexes, and data partitions, the results will be delivered more quickly.

Automatic Optimization

We will learn few automation techniques to efficiently optimize queries.

Embarrassingly Parallel

Easy to learn consistent, predictable API that has strict schema

SQLs like expressions

`Let's get started`

Installation

In [ ]:
```
!pip install -U numpy
```

Finance and Supply chain Data Analytics

Finance data model

A finance data model is a comprehensive and structured framework used to represent and organize financial information within an organization.

It serves as the blueprint for how financial data is collected, stored, processed, and analyzed, ensuring accuracy, consistency, and efficiency in managing financial operations.

The model defines the relationships between various financial entities such as assets, liabilities, revenues, expenses, and equity, enabling financial professionals to gain insights into the company's financial health, performance, and risk exposure.

It typically encompasses multiple dimensions, including time, currency, and geographical locations, chart of accounts, departments / cost centers, fiscal years and reporting accounting periods providing a holistic view of the organization's financial landscape.

A well-designed finance data model is critical for generating accurate financial reports, facilitating financial planning and forecasting, and supporting strategic decision-making at all levels of the business.

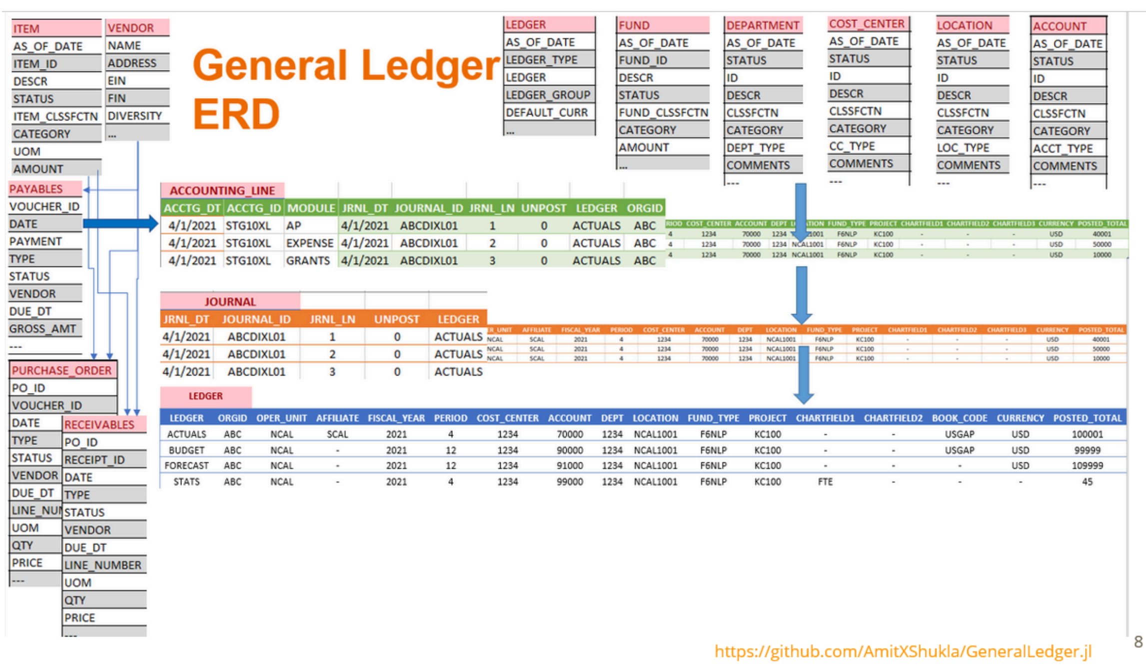
As stated above, since our objective is learn Data Science operations on Finance and Supply chain dataset, we will focus on creating few real life examples which are similar to Finance and Supply chain.

For more information, please learn more about Finance and Supply chain ERP data.

Objective of following section is to understand ERP GL like data.

A sample of data structure and ERD relationship diagram can be seen in this diagram below.

`Finance ER Diagram`



Supply chain data model

A supply chain data model is a structured representation of the various elements and interactions within a supply chain network.

It encompasses critical components such as customers, orders, receipts, products, invoices, vouchers, and ship-to locations.

Customers form the foundation of the supply chain, as they drive demand for products. Orders and receipts represent the flow of goods and services, capturing the movement of inventory throughout the supply chain.

The product entity accounts for the diverse range of items being handled, from raw materials to finished goods.

Invoices and vouchers track financial transactions, ensuring transparent and accurate billing processes.

Ship-to locations specify the destinations of goods during the distribution process.

By establishing relationships and attributes between these elements, the supply chain data model aids in optimizing inventory management, forecasting demand, enhancing order fulfillment, and ultimately, improving overall operational efficiency within the supply chain ecosystem.

`Supply Chain ER Diagram`

ERD Diagram

Creating Polars DataFrame

Polars Data Structure

The core base data structures provided by Polars are Series and DataFrames.

Finance DataSet

In [ ]:

```python
#########################
## Series and DataFrames
#########################
import polars as pl

# with a tuple
location_1 = pl.Series(["CA", "OR", "WA", "TX", "NY"])
# location_1 series when will converted to DataFrame will not have a column name
# later is used to create a DataFrame will assign a column name like column_xx
```

```python
location_2 = pl.Series("location", ["CA", "OR", "WA", "TX", "NY"])

print(f"Location Type: Series 1: ", location_1)
print(f"Location Type: Series 2: ", location_2)

location_1_df = pl.DataFrame(location_1)
location_2_df = pl.DataFrame(location_2)
print(f"Location Type: DataFrame 1: ", location_1_df)
print(f"Location Type: DataFrame 2: ", location_2_df)
# type(location_1_df["location"])
# will error out, because location_1 series didn't had column name
type(location_2_df["location"]), type(location_1), type(location_2)
```

```
Location Type: Series 1:  shape: (5,)
Series: '' [str]
[
	"CA"
	"OR"
	"WA"
	"TX"
	"NY"
]
Location Type: Series 2:  shape: (5,)
Series: 'location' [str]
[
	"CA"
	"OR"
	"WA"
	"TX"
	"NY"
]
Location Type: DataFrame 1:  shape: (5, 1)
┌──────────┐
│ column_0 │
│ ---      │
│ str      │
╞══════════╡
│ CA       │
│ OR       │
│ WA       │
│ TX       │
│ NY       │
└──────────┘
Location Type: DataFrame 2:  shape: (5, 1)
┌──────────┐
│ location │
│ ---      │
│ str      │
╞══════════╡
│ CA       │
│ OR       │
│ WA       │
│ TX       │
│ NY       │
└──────────┘
```

Out[ ]:
```
(polars.series.series.Series,
 polars.series.series.Series,
 polars.series.series.Series)
```

In [ ]:
```
# Creating DataFrame from a dict or a collection of dicts.
# let's create a more sophisticated DataFrame
# in real world, Organization maintain dozens of record structure to store
# different type of locations, like ShipTo Location, Receiving,
# Mailing, Corp. office, head office,
# field office etc. etc.

########################
## LOCATION DataFrame ##
########################
import random
from datetime import datetime

location = pl.DataFrame({
    "ID":  list(range(11, 23)),
    "AS_OF_DATE" : datetime(2022, 1, 1),
    "DESCRIPTION" : ["Boston","New York","Philadelphia","Cleveland","Richmond",
                     "Atlanta","Chicago","St. Louis","Minneapolis","Kansas City",
                     "Dallas","San Francisco"],
    "REGION": ["Region A","Region B","Region C","Region D"] * 3,
    "TYPE" : "Physical",
    "CATEGORY" : ["Ship","Recv","Mfg"] * 4
})
location.sample(5).with_row_count("Row #")
```

Out[ ]: shape: (5, 7)

Row #	ID	AS_OF_DATE	DESCRIPTION	REGION	TYPE	CATEGORY
u32	i64	datetime[μs]	str	str	str	str
0	21	2022-01-01 00:00:00	"Dallas"	"Region C"	"Physical"	"Recv"
1	12	2022-01-01 00:00:00	"New York"	"Region B"	"Physical"	"Recv"
2	13	2022-01-01 00:00:00	"Philadelphia"	"Region C"	"Physical"	"Mfg"
3	16	2022-01-01 00:00:00	"Atlanta"	"Region B"	"Physical"	"Mfg"
4	20	2022-01-01 00:00:00	"Kansas City"	"Region B"	"Physical"	"Ship"

In [ ]:
```
########################
## ACCOUNTS DataFrame ##
########################
import random
from datetime import datetime

accounts = pl.DataFrame({
    "ID":  list(range(10000, 45000, 1000)),
    "AS_OF_DATE" : datetime(2022, 1, 1),
    "DESCRIPTION" : ["Operating Expenses","Non Operating Expenses","Assets",
                     "Liabilities","Net worth accounts", "Statistical Accounts",
                     "Revenue"] * 5,
```

```
    "REGION": ["Region A","Region B","Region C","Region D", "Region E"] * 7,
    "TYPE" : ["E","E","A","L","N","S","R"] * 5,
    "STATUS" : "Active",
    "CLASSIFICATION" : ["OPERATING_EXPENSES","NON-OPERATING_EXPENSES",
                        "ASSETS","LIABILITIES","NET_WORTH","STATISTICS",
                        "REVENUE"] * 5,
    "CATEGORY" : [
                "Travel","Payroll","non-Payroll","Allowance","Cash",
                "Facility","Supply","Services","Investment","Misc.",
                "Depreciation","Gain","Service","Retired","Fault.",
                "Receipt","Accrual","Return","Credit","ROI",
                "Cash","Funds","Invest","Transfer","Roll-over",
                "FTE","Members","Non_Members","Temp","Contractors",
                "Sales","Merchant","Service","Consulting","Subscriptions"
        ],
})
accounts.sample(5).with_row_count("Row #")
```

Out[ ]: shape: (5, 9)

Row #	ID	AS_OF_DATE	DESCRIPTION	REGION	TYPE	STATUS	CLASSIFICATION	C
u32	i64	datetime[μs]	str	str	str	str	str	
0	27000	2022-01-01 00:00:00	"Liabilities"	"Region C"	"L"	"Active"	"LIABILITIES"	
1	38000	2022-01-01 00:00:00	"Operating Expe…	"Region D"	"E"	"Active"	"OPERATING_EXPE…	
2	20000	2022-01-01 00:00:00	"Liabilities"	"Region A"	"L"	"Active"	"LIABILITIES"	"De
3	40000	2022-01-01 00:00:00	"Assets"	"Region A"	"A"	"Active"	"ASSETS"	
4	18000	2022-01-01 00:00:00	"Non Operating …	"Region D"	"E"	"Active"	"NON-OPERATING_…	"Ir

In [ ]:

```
##########################
## DEPARTMENT DataFrame ##
##########################
import random
from datetime import datetime

dept = pl.DataFrame({
    "ID":  list(range(1000, 2500, 100)),
    "AS_OF_DATE" : datetime(2022, 1, 1),
    "DESCRIPTION" : ["Sales & Marketing","Human Resource",
                     "Information Technology","Business leaders","other temp"] * 3,
    "REGION": ["Region A","Region B","Region C"] * 5,
    "STATUS" : "Active",
    "CLASSIFICATION" : ["SALES","HR", "IT","BUSINESS","OTHERS"] * 3,
    "TYPE" : ["S","H","I","B","O"] * 3,
    "CATEGORY" : ["sales","human_resource","IT_Staff","business","others"] * 3,
```

```
})
dept.sample(5).with_row_count("Row #")
```

Out[ ]: shape: (5, 9)

Row #	ID	AS_OF_DATE	DESCRIPTION	REGION	STATUS	CLASSIFICATION	TYPE	
u32	i64	datetime[μs]	str	str	str	str	str	
0	2200	2022-01-01 00:00:00	"Information Te...	"Region A"	"Active"	"IT"	"I"	
1	1100	2022-01-01 00:00:00	"Human Resource...	"Region B"	"Active"	"HR"	"H"	"human
2	1500	2022-01-01 00:00:00	"Sales & Market...	"Region C"	"Active"	"SALES"	"S"	
3	2000	2022-01-01 00:00:00	"Sales & Market...	"Region B"	"Active"	"SALES"	"S"	
4	1800	2022-01-01 00:00:00	"Business leade...	"Region C"	"Active"	"BUSINESS"	"B"	

In [ ]:

```
######################
## LEDGER DataFrame ##
######################
import random
from datetime import datetime

sampleSize = 100_000
org = "ABC Inc."
ledger_type = "ACTUALS" # BUDGET, STATS are other Ledger types
fiscal_year_from = 2020
fiscal_year_to = 2023
random.seed(123)

ledger = pl.DataFrame({
        "LEDGER" : ledger_type,
        "ORG" : org,
        "FISCAL_YEAR": random.choices(list(range(fiscal_year_from,
                                      fiscal_year_to+1, 1)),k=sampleSize),
        "PERIOD": random.choices(list(range(1, 12+1, 1)),k=sampleSize),
        "ACCOUNT" : random.choices(accounts["ID"], k=sampleSize),
        "DEPT" : random.choices(dept["ID"], k=sampleSize),
        "LOCATION" : random.choices(location["ID"], k=sampleSize),
        "POSTED_TOTAL": random.sample(range(1000000), sampleSize)
})
ledger.sample(5).with_row_count("Row #")
```

Out[ ]: shape: (5, 9)

Row #	LEDGER	ORG	FISCAL_YEAR	PERIOD	ACCOUNT	DEPT	LOCATION	POSTED_TOTA
u32	str	str	i64	i64	i64	i64	i64	i6
0	"ACTUALS"	"ABC Inc."	2021	1	26000	1700	14	63525
1	"ACTUALS"	"ABC Inc."	2020	8	31000	2300	21	50213
2	"ACTUALS"	"ABC Inc."	2020	9	31000	1300	22	72944
3	"ACTUALS"	"ABC Inc."	2020	6	15000	2400	22	76932
4	"ACTUALS"	"ABC Inc."	2021	8	31000	1700	17	4946

In [ ]:
```
ledger_type = "BUDGET" # ACTUALS, STATS are other Ledger types

ledger_budg = pl.DataFrame({
        "LEDGER" : ledger_type,
        "ORG" : org,
        "FISCAL_YEAR": random.choices(list(range(fiscal_year_from, fiscal_year_to+1
                                ,k=sampleSize),
        "PERIOD": random.choices(list(range(1, 12+1, 1)),k=sampleSize),
        "ACCOUNT" : random.choices(accounts["ID"], k=sampleSize),
        "DEPT" : random.choices(dept["ID"], k=sampleSize),
        "LOCATION" : random.choices(location["ID"], k=sampleSize),
        "POSTED_TOTAL": random.sample(range(1000000), sampleSize)
})
ledger_budg.sample(5).with_row_count("Row #")
```

Out[ ]: shape: (5, 9)

Row #	LEDGER	ORG	FISCAL_YEAR	PERIOD	ACCOUNT	DEPT	LOCATION	POSTED_TOTAI
u32	str	str	i64	i64	i64	i64	i64	i6
0	"BUDGET"	"ABC Inc."	2021	11	30000	1100	11	7497
1	"BUDGET"	"ABC Inc."	2021	9	42000	2300	22	11584
2	"BUDGET"	"ABC Inc."	2022	10	17000	1300	16	15653
3	"BUDGET"	"ABC Inc."	2021	8	20000	1100	14	21206
4	"BUDGET"	"ABC Inc."	2021	5	38000	1600	21	20405

In [ ]:

```
##########################################
# combined ledger for Actuals and Budget
##########################################
dfLedger = pl.concat([ledger, ledger_budg], how="vertical")
dfLedger.sample(5).with_row_count("Row #")
```

Out[ ]: shape: (5, 9)

Row #	LEDGER	ORG	FISCAL_YEAR	PERIOD	ACCOUNT	DEPT	LOCATION	POSTED_TOTA
u32	str	str	i64	i64	i64	i64	i64	i6
0	"ACTUALS"	"ABC Inc."	2023	2	43000	2000	15	21654
1	"ACTUALS"	"ABC Inc."	2022	5	14000	2000	15	91016
2	"BUDGET"	"ABC Inc."	2020	11	23000	2000	13	86628
3	"ACTUALS"	"ABC Inc."	2021	3	31000	2400	19	60986
4	"BUDGET"	"ABC Inc."	2022	8	36000	1300	19	2461

Supply Chain DataSet

In [ ]:

```
#################################
## PRODUCT_CATEGORY DataFrame ##
#################################
```

```python
import random
from datetime import datetime

category = pl.DataFrame({
    "ID":  list(range(1000, 2500, 100)),
    "AS_OF_DATE" : datetime(2022, 1, 1),
    "DESCRIPTION" : ["Rx","Material","Consulting","Construction",
                     "un-assigned"] * 3,
    "REGION": ["Region A","Region B","Region C"] * 5,
    "STATUS" : "Active",
    "CLASSIFICATION" : ["Rx","Material", "Services","Constructions",
                        "OTHERS"] * 3,
    "TYPE" : ["R","M","S","C","O"] * 3,
})
category.sample(5).with_row_count("Row #")
```

Out[ ]: shape: (5, 8)

Row #	ID	AS_OF_DATE	DESCRIPTION	REGION	STATUS	CLASSIFICATION	TYPE
u32	i64	datetime[μs]	str	str	str	str	str
0	1000	2022-01-01 00:00:00	"Rx"	"Region A"	"Active"	"Rx"	"R"
1	1800	2022-01-01 00:00:00	"Construction"	"Region C"	"Active"	"Constructions"	"C"
2	2200	2022-01-01 00:00:00	"Consulting"	"Region A"	"Active"	"Services"	"S"
3	2100	2022-01-01 00:00:00	"Material"	"Region C"	"Active"	"Material"	"M"
4	1400	2022-01-01 00:00:00	"un-assigned"	"Region B"	"Active"	"OTHERS"	"O"

In [ ]:

```python
#######################
## PRODUCT DataFrame ##
#######################
import random
from datetime import datetime

product = pl.DataFrame({
    "ID":  list(range(100, 250, 10)),
    "AS_OF_DATE" : datetime(2022, 1, 1),
    "DESCRIPTION" : ["Item 1","Item 2","Item 3","Item 4","Item 5"] * 3,
    "STATUS" : "Active",
    "CATEGORY" : random.choices(category["ID"], k=15),
})
product.sample(5).with_row_count("Row #")
```

Out[ ]: shape: (5, 6)

Row #	ID	AS_OF_DATE	DESCRIPTION	STATUS	CATEGORY
u32	i64	datetime[μs]	str	str	i64
0	220	2022-01-01 00:00:00	"Item 3"	"Active"	1900
1	110	2022-01-01 00:00:00	"Item 2"	"Active"	1900
2	190	2022-01-01 00:00:00	"Item 5"	"Active"	1800
3	130	2022-01-01 00:00:00	"Item 4"	"Active"	1900
4	150	2022-01-01 00:00:00	"Item 1"	"Active"	1900

In [ ]:
```
########################
## CUSTOMER DataFrame ##
########################
import random
from datetime import datetime

customer = pl.DataFrame({
    "ID":  list(range(100, 250, 10)),
    "AS_OF_DATE" : datetime(2022, 1, 1),
    "DESCRIPTION" : ["Customer 1","Customer 2","Customer 3",
                     "Customer 4","Customer 5"] * 3,
    "ADDRESS" : ["Address 1","Address 2","Address 3",
                 "Address 4","Address 5"] * 3,
    "PHONE" : ["0000000001","0000000002","0000000003",
               "0000000004","0000000005"] * 3,
    "EMAIL" : ["1@email","2@email","3@email","4@email","5@email"] * 3,
    "STATUS" : "Active",
    "TYPE" : ["Corp","Gov","Individual"] * 5,
    "CATEGORY" : random.choices(category["ID"], k=15),
})
customer.sample(5).with_row_count("Row #")
```

Out[ ]: shape: (5, 10)

Row #	ID	AS_OF_DATE	DESCRIPTION	ADDRESS	PHONE	EMAIL	STATUS	TYPI
u32	i64	datetime[μs]	str	str	str	str	str	st
0	100	2022-01-01 00:00:00	"Customer 1"	"Address 1"	"0000000001"	"1@email"	"Active"	"Corp
1	170	2022-01-01 00:00:00	"Customer 3"	"Address 3"	"0000000003"	"3@email"	"Active"	"Gov
2	190	2022-01-01 00:00:00	"Customer 5"	"Address 5"	"0000000005"	"5@email"	"Active"	"Corp
3	130	2022-01-01 00:00:00	"Customer 4"	"Address 4"	"0000000004"	"4@email"	"Active"	"Corp
4	140	2022-01-01 00:00:00	"Customer 5"	"Address 5"	"0000000005"	"5@email"	"Active"	"Gov

In [ ]:

```
#####################
## ORDER DataFrame ##
#####################
import random
from datetime import datetime
sampleSize = 10

order = pl.DataFrame({
    "ID":  list(range(1000, 1100, sampleSize)),
    "AS_OF_DATE" : datetime(2024, 1, 1),
    "CUSTOMER": random.choices(customer["ID"], k=sampleSize),
    "ITEM": random.choices(product["ID"], k=sampleSize),
    "QTY": random.sample(range(1000000), sampleSize),
        "POSTED_TOTAL": random.sample(range(1000000), sampleSize)
})
order.sample(5).with_row_count("Row #")
```

Out[ ]: shape: (5, 7)

Row #	ID	AS_OF_DATE	CUSTOMER	ITEM	QTY	POSTED_TOTAL
u32	i64	datetime[μs]	i64	i64	i64	i64
0	1000	2024-01-01 00:00:00	150	180	966405	835603
1	1070	2024-01-01 00:00:00	140	120	253823	928636
2	1060	2024-01-01 00:00:00	150	160	581636	37077
3	1030	2024-01-01 00:00:00	100	240	883691	505463
4	1040	2024-01-01 00:00:00	110	230	17190	27331

In [ ]:
```
#######################
## INVOICE DataFrame ##
#######################
import random
from datetime import datetime
sampleSize = 10

invoice = pl.DataFrame({
    "AS_OF_DATE" : datetime(2024, 1, 1),
    "ORDER": random.choices(order["ID"], k=sampleSize),
    "STATUS" : ["open","paid","cancelled","shipped","hold"] * 2,
})
invoice.sample(5).with_row_count("Row #")
```

Out[ ]: shape: (5, 4)

Row #	AS_OF_DATE	ORDER	STATUS
u32	datetime[μs]	i64	str
0	2024-01-01 00:00:00	1040	"open"
1	2024-01-01 00:00:00	1010	"paid"
2	2024-01-01 00:00:00	1000	"cancelled"
3	2024-01-01 00:00:00	1010	"cancelled"
4	2024-01-01 00:00:00	1030	"paid"

Data Types

Polars is entirely based on Arrow data types and backed by Arrow memory arrays. This makes data processing cache-efficient and well-supported for Inter Process Communication.

Please read official Polars Data Type documentation for more details.

IO

In [ ]:
```
#################
### csv files ###
#################
dfLedger.write_csv("../downloads/ledger.csv")
dfLedger_c = pl.read_csv("../downloads/ledger.csv")
# Polars allows you to scan a CSV input.
# Scanning delays the actual parsing of the file
# and instead returns a lazy computation holder called a LazyFrame.
dfLedger_c = pl.scan_csv("../downloads/ledger.csv")

#####################
### parquet files ###
```

```
#####################
dfLedger.write_parquet("../downloads/ledger.parquet")
dfLedger_c = pl.read_parquet("../downloads/ledger.parquet")
# Polars allows you to scan a parquet input.
# Scanning delays the actual parsing of the file and instead
# returns a lazy computation holder called a LazyFrame.
dfLedger_c = pl.scan_parquet("../downloads/ledger.parquet")

######################################################
### json files ### ndjson: new line delimited json ###
######################################################
dfLedger.write_json("../downloads/ledger.json")
dfLedger_c = pl.scan_json("../downloads/ledger.json")
# Polars allows you to scan a json input.
# Scanning delays the actual parsing of the file and instead
# returns a lazy computation holder called a LazyFrame.
dfLedger_c = pl.scan_json("../downloads/ledger.json")

####################
## multiple files ##
####################
for i in range(5):
    dfLedger.write_csv(f"../downloads/my_many_files_{i}.csv")
pl.scan_csv("../downloads/my_many_files_*.csv").show_graph()
# see how query optimization/parallelism works
df = pl.read_csv("../downloads/my_many_files_*.csv")
print(df.shape)

###############
## databases ##
###############
import polars as pl

connection_uri = "postgres://username:password@server:port/database"
query = "SELECT * FROM foo"

pl.read_database(query=query, connection_uri=connection_uri)

# Polars doesn't manage connections and data transfer from databases by itself.
# Instead external libraries (known as engines) handle this.
#  At present Polars can use two engines to read from databases:
# ConnectorX and ADBC
# $  pip install connectorx
# $  pip install adbc-driver-sqlite

# As with reading from a database above Polars uses an engine
# to write to a database.
# The currently supported engines are:
# SQLAlchemy and
# Arrow Database Connectivity (ADBC)
# $  pip install SQLAlchemy pandas

# AWS & Big Query - API - WIP as of 07/11/23
```

Polars DataFrame Context

The two core components of the Polars DataFrame DSL (domain specific language) are Contexts and Expression.

A context, as implied by the name, refers to the context in which an expression needs to be evaluated. There are three main contexts:

```
Selection: df.select([..]), df.with_columns([..])
Filtering: df.filter()
Groupby / Aggregation: df.groupby(..).agg([..])
```

Additional Contexts

```
List, Arrays and SQL
```

In [ ]:
```
print(dfLedger.sample(5))
##################
## select context
##################
out = dfLedger.select(
    pl.col("FISCAL_YEAR").sort(),
    pl.col("PERIOD").sort(),
    pl.col(["LEDGER", "ORG", "ACCOUNT", "DEPT", "LOCATION"]),
    (pl.col("POSTED_TOTAL") / 1000).alias("in Thousands"),
).with_row_count("Row #")
print(out)
```

```
shape: (5, 8)
┌────────┬──────────┬─────────────┬────────┬─────────┬──────┬──────────┬─────────────┐
│ LEDGER ┆ ORG      ┆ FISCAL_YEAR ┆ PERIOD ┆ ACCOUNT ┆ DEPT ┆ LOCATION ┆ POSTED_TOTAL │
│ ---    ┆ ---      ┆ ---         ┆ ---    ┆ ---     ┆ ---  ┆ ---      ┆ ---          │
│ str    ┆ str      ┆ i64         ┆ i64    ┆ i64     ┆ i64  ┆ i64      ┆ i64          │
╞════════╪══════════╪═════════════╪════════╪═════════╪══════╪══════════╪══════════════╡
│ BUDGET ┆ ABC Inc. ┆ 2023        ┆ 12     ┆ 20000   ┆ 2100 ┆ 12       ┆ 739002       │
│ BUDGET ┆ ABC Inc. ┆ 2023        ┆ 9      ┆ 19000   ┆ 1400 ┆ 15       ┆ 317636       │
│ BUDGET ┆ ABC Inc. ┆ 2020        ┆ 11     ┆ 31000   ┆ 2200 ┆ 12       ┆ 226428       │
│ BUDGET ┆ ABC Inc. ┆ 2022        ┆ 12     ┆ 29000   ┆ 1500 ┆ 14       ┆ 282944       │
│ BUDGET ┆ ABC Inc. ┆ 2023        ┆ 10     ┆ 41000   ┆ 1200 ┆ 12       ┆ 712123       │
└────────┴──────────┴─────────────┴────────┴─────────┴──────┴──────────┴──────────────┘
shape: (200_000, 9)
┌────────┬─────────────┬────────┬─────────┬───┬─────────┬──────┬──────────┬──────────────┐
│ Row #  ┆ FISCAL_YEAR ┆ PERIOD ┆ LEDGER  ┆ … ┆ ACCOUNT ┆ DEPT ┆ LOCATION ┆ in Thousands │
│ ---    ┆ ---         ┆ ---    ┆ ---     ┆   ┆ ---     ┆ ---  ┆ ---      ┆ ---          │
│ u32    ┆ i64         ┆ i64    ┆ str     ┆   ┆ i64     ┆ i64  ┆ i64      ┆ f64          │
╞════════╪═════════════╪════════╪═════════╪═══╪═════════╪══════╪══════════╪══════════════╡
│ 0      ┆ 2020        ┆ 1      ┆ ACTUALS ┆ … ┆ 12000   ┆ 2400 ┆ 22       ┆ 753.956      │
│ 1      ┆ 2020        ┆ 1      ┆ ACTUALS ┆ … ┆ 10000   ┆ 1900 ┆ 14       ┆ 826.906      │
│ 2      ┆ 2020        ┆ 1      ┆ ACTUALS ┆ … ┆ 21000   ┆ 1700 ┆ 17       ┆ 454.574      │
│ 3      ┆ 2020        ┆ 1      ┆ ACTUALS ┆ … ┆ 34000   ┆ 1300 ┆ 12       ┆ 334.989      │
│ …      ┆ …           ┆ …      ┆ …       ┆ … ┆ …       ┆ …    ┆ …        ┆ …            │
│ 199996 ┆ 2023        ┆ 12     ┆ BUDGET  ┆ … ┆ 18000   ┆ 2200 ┆ 18       ┆ 65.437       │
│ 199997 ┆ 2023        ┆ 12     ┆ BUDGET  ┆ … ┆ 17000   ┆ 1300 ┆ 13       ┆ 960.254      │
│ 199998 ┆ 2023        ┆ 12     ┆ BUDGET  ┆ … ┆ 35000   ┆ 2000 ┆ 20       ┆ 41.157       │
│ 199999 ┆ 2023        ┆ 12     ┆ BUDGET  ┆ … ┆ 41000   ┆ 1900 ┆ 21       ┆ 265.241      │
└────────┴─────────────┴────────┴─────────┴───┴─────────┴──────┴──────────┴──────────────┘
```

In [ ]:
```
##########################
## with_columns context ##
##########################
out = dfLedger.with_columns(
    (pl.col("POSTED_TOTAL") / 1000).alias("in Thousands"),
).with_row_count("Row #")
print(out)
```

```
shape: (200_000, 10)
┌────────┬─────────┬──────────┬─────────────┬───┬──────┬──────────┬──────────────┬───
─────────────┐
│ Row #  ┆ LEDGER  ┆ ORG      ┆ FISCAL_YEAR ┆ … ┆ DEPT ┆ LOCATION ┆ POSTED_TOTAL ┆ i
n Thousands │
│ ---    ┆ ---     ┆ ---      ┆ ---         ┆   ┆ ---  ┆ ---      ┆ ---          ┆ -
--           │
│ u32    ┆ str     ┆ str      ┆ i64         ┆   ┆ i64  ┆ i64      ┆ i64          ┆ f
64           │
╞════════╪═════════╪══════════╪═════════════╪═══╪══════╪══════════╪══════════════╪═══
═════════════╡
│ 0      ┆ ACTUALS ┆ ABC Inc. ┆ 2020        ┆ … ┆ 2400 ┆ 22       ┆ 753956       ┆ 7
53.956       │
│ 1      ┆ ACTUALS ┆ ABC Inc. ┆ 2020        ┆ … ┆ 1900 ┆ 14       ┆ 826906       ┆ 8
26.906       │
│ 2      ┆ ACTUALS ┆ ABC Inc. ┆ 2021        ┆ … ┆ 1700 ┆ 17       ┆ 454574       ┆ 4
54.574       │
│ 3      ┆ ACTUALS ┆ ABC Inc. ┆ 2020        ┆ … ┆ 1300 ┆ 12       ┆ 334989       ┆ 3
34.989       │
│ …      ┆ …       ┆ …        ┆ …           ┆ … ┆ …    ┆ …        ┆ …            ┆ …
│
│ 199996 ┆ BUDGET  ┆ ABC Inc. ┆ 2022        ┆ … ┆ 2200 ┆ 18       ┆ 65437        ┆ 6
5.437        │
│ 199997 ┆ BUDGET  ┆ ABC Inc. ┆ 2020        ┆ … ┆ 1300 ┆ 13       ┆ 960254       ┆ 9
60.254       │
│ 199998 ┆ BUDGET  ┆ ABC Inc. ┆ 2022        ┆ … ┆ 2000 ┆ 20       ┆ 41157        ┆ 4
1.157        │
│ 199999 ┆ BUDGET  ┆ ABC Inc. ┆ 2020        ┆ … ┆ 1900 ┆ 21       ┆ 265241       ┆ 2
65.241       │
└────────┴─────────┴──────────┴─────────────┴───┴──────┴──────────┴──────────────┴───
─────────────┘
```

In [ ]:
```
##########################
## filter context ##
##########################
out = dfLedger.filter(
    ((pl.col("LEDGER") == "ACTUALS") & (pl.col("FISCAL_YEAR") == 2023))
).with_row_count("Row #")
print(out)
```

```
shape: (25_136, 9)
┌───────┬─────────┬──────────┬─────────────┬───┬─────────┬──────┬──────────┬──────────────┐
│ Row # ┆ LEDGER  ┆ ORG      ┆ FISCAL_YEAR ┆ … ┆ ACCOUNT ┆ DEPT ┆ LOCATION ┆ POSTED_TOTAL │
│ ---   ┆ ---     ┆ ---      ┆ ---         ┆   ┆ ---     ┆ ---  ┆ ---      ┆ ---          │
│ u32   ┆ str     ┆ str      ┆ i64         ┆   ┆ i64     ┆ i64  ┆ i64      ┆ i64          │
╞═══════╪═════════╪══════════╪═════════════╪═══╪═════════╪══════╪══════════╪══════════════╡
│ 0     ┆ ACTUALS ┆ ABC Inc. ┆ 2023        ┆ … ┆ 14000   ┆ 2100 ┆ 11       ┆ 290813       │
│ 1     ┆ ACTUALS ┆ ABC Inc. ┆ 2023        ┆ … ┆ 16000   ┆ 1600 ┆ 21       ┆ 28323        │
│ 2     ┆ ACTUALS ┆ ABC Inc. ┆ 2023        ┆ … ┆ 42000   ┆ 1600 ┆ 16       ┆ 106856       │
│ 3     ┆ ACTUALS ┆ ABC Inc. ┆ 2023        ┆ … ┆ 37000   ┆ 1600 ┆ 17       ┆ 145223       │
│ …     ┆ …       ┆ …        ┆ …           ┆ … ┆ …       ┆ …    ┆ …        ┆ …            │
│ 25132 ┆ ACTUALS ┆ ABC Inc. ┆ 2023        ┆ … ┆ 20000   ┆ 1400 ┆ 17       ┆ 831737       │
│ 25133 ┆ ACTUALS ┆ ABC Inc. ┆ 2023        ┆ … ┆ 32000   ┆ 1700 ┆ 16       ┆ 131723       │
│ 25134 ┆ ACTUALS ┆ ABC Inc. ┆ 2023        ┆ … ┆ 33000   ┆ 2200 ┆ 17       ┆ 187186       │
│ 25135 ┆ ACTUALS ┆ ABC Inc. ┆ 2023        ┆ … ┆ 14000   ┆ 1800 ┆ 18       ┆ 642420       │
└───────┴─────────┴──────────┴─────────────┴───┴─────────┴──────┴──────────┴──────────────┘
```

In [ ]:

```python
######################
## group by context ##
######################
out = dfLedger.filter(
    ((pl.col("LEDGER") == "ACTUALS") & (pl.col("FISCAL_YEAR") == 2023))
        ).group_by("LEDGER").agg(
                                pl.count()
                                ).with_row_count("Row #")
print(out)

out = dfLedger.group_by("LEDGER", "FISCAL_YEAR").agg(
                                pl.count()
                                ).with_row_count("Row #")
print(out)

# sort group by data by FISCAL_YEAR
out = dfLedger.group_by("LEDGER", "FISCAL_YEAR").agg(
                                pl.count(),
                                pl.sum("POSTED_TOTAL"),
                                (pl.sum("POSTED_TOTAL") / 1_000_000)
                                .alias("Posted Total in Million"),
                                ).with_row_count("Row #")
print(out)
```

shape: (1, 3)

Row # --- u32	LEDGER --- str	count --- u32
0	ACTUALS	25136

shape: (8, 4)

Row # --- u32	LEDGER --- str	FISCAL_YEAR --- i64	count --- u32
0	ACTUALS	2023	25136
1	ACTUALS	2020	24957
2	BUDGET	2022	24885
3	ACTUALS	2021	25036
4	BUDGET	2023	24815
5	BUDGET	2020	25159
6	ACTUALS	2022	24871
7	BUDGET	2021	25141

shape: (8, 6)

Row # --- u32	LEDGER --- str	FISCAL_YEAR --- i64	count --- u32	POSTED_TOTAL --- i64	Posted Total in Million --- f64
0	BUDGET	2021	25141	12639396311	12639.396311
1	ACTUALS	2020	24957	12430170980	12430.17098
2	ACTUALS	2023	25136	12526897831	12526.897831
3	BUDGET	2020	25159	12611880007	12611.880007
4	BUDGET	2023	24815	12446847043	12446.847043
5	ACTUALS	2021	25036	12506846602	12506.846602
6	BUDGET	2022	24885	12386337109	12386.337109
7	ACTUALS	2022	24871	12420814798	12420.814798

Polars DataFrame Expressions

using Expression to select columns

In [ ]:
```
######################
# selection context ##
######################
print(order.sample(5).with_row_count("Row #"))

 # using numerical operators
dfOrder = order.select(
    pl.col("ID"),
    pl.col("AS_OF_DATE"),
    pl.col("CUSTOMER"),
```

```
    pl.col("ITEM"),
    pl.col("QTY"),
    (pl.col("POSTED_TOTAL") / 1000).alias("Amount in thousands"),
).with_row_count("Row #")
print(dfOrder)
```

```
shape: (5, 7)
┌───────┬──────┬─────────────────────┬──────────┬──────┬────────┬──────────────┐
│ Row # ┆ ID   ┆ AS_OF_DATE          ┆ CUSTOMER ┆ ITEM ┆ QTY    ┆ POSTED_TOTAL │
│ ---   ┆ ---  ┆ ---                 ┆ ---      ┆ ---  ┆ ---    ┆ ---          │
│ u32   ┆ i64  ┆ datetime[μs]        ┆ i64      ┆ i64  ┆ i64    ┆ i64          │
╞═══════╪══════╪═════════════════════╪══════════╪══════╪════════╪══════════════╡
│ 0     ┆ 1050 ┆ 2024-01-01 00:00:00 ┆ 130      ┆ 230  ┆ 440813 ┆ 510808       │
│ 1     ┆ 1010 ┆ 2024-01-01 00:00:00 ┆ 180      ┆ 150  ┆ 295337 ┆ 266870       │
│ 2     ┆ 1020 ┆ 2024-01-01 00:00:00 ┆ 120      ┆ 170  ┆ 962406 ┆ 997515       │
│ 3     ┆ 1030 ┆ 2024-01-01 00:00:00 ┆ 100      ┆ 240  ┆ 883691 ┆ 505463       │
│ 4     ┆ 1070 ┆ 2024-01-01 00:00:00 ┆ 140      ┆ 120  ┆ 253823 ┆ 928636       │
└───────┴──────┴─────────────────────┴──────────┴──────┴────────┴──────────────┘
shape: (10, 7)
┌───────┬──────┬─────────────────────┬──────────┬──────┬────────┬─────────────────────┐
│ Row # ┆ ID   ┆ AS_OF_DATE          ┆ CUSTOMER ┆ ITEM ┆ QTY    ┆ Amount in thousands │
│ ---   ┆ ---  ┆ ---                 ┆ ---      ┆ ---  ┆ ---    ┆ ---                 │
│ u32   ┆ i64  ┆ datetime[μs]        ┆ i64      ┆ i64  ┆ i64    ┆ f64                 │
╞═══════╪══════╪═════════════════════╪══════════╪══════╪════════╪═════════════════════╡
│ 0     ┆ 1000 ┆ 2024-01-01 00:00:00 ┆ 150      ┆ 180  ┆ 966405 ┆ 835.603             │
│ 1     ┆ 1010 ┆ 2024-01-01 00:00:00 ┆ 180      ┆ 150  ┆ 295337 ┆ 266.87              │
│ 2     ┆ 1020 ┆ 2024-01-01 00:00:00 ┆ 120      ┆ 170  ┆ 962406 ┆ 997.515             │
│ 3     ┆ 1030 ┆ 2024-01-01 00:00:00 ┆ 100      ┆ 240  ┆ 883691 ┆ 505.463             │
│ …     ┆ …    ┆ …                   ┆ …        ┆ …    ┆ …      ┆ …                   │
│ 6     ┆ 1060 ┆ 2024-01-01 00:00:00 ┆ 150      ┆ 160  ┆ 581636 ┆ 37.077              │
│ 7     ┆ 1070 ┆ 2024-01-01 00:00:00 ┆ 140      ┆ 120  ┆ 253823 ┆ 928.636             │
│ 8     ┆ 1080 ┆ 2024-01-01 00:00:00 ┆ 160      ┆ 200  ┆ 877957 ┆ 793.188             │
│ 9     ┆ 1090 ┆ 2024-01-01 00:00:00 ┆ 220      ┆ 170  ┆ 660229 ┆ 512.122             │
└───────┴──────┴─────────────────────┴──────────┴──────┴────────┴─────────────────────┘
```

In [ ]:

```
# select context the selection applies expressions over columns.
# The expressions in this context must produce Series that are
# all the same length or have a length of 1.

# select all cols
```

```
dfOrder.select(pl.col("*")).sample(5)
# dfOrder.select(pl.all()).sample(5)
```

Out[ ]: shape: (5, 7)

Row #	ID	AS_OF_DATE	CUSTOMER	ITEM	QTY	Amount in thousands
u32	i64	datetime[μs]	i64	i64	i64	f64
9	1090	2024-01-01 00:00:00	220	170	660229	512.122
1	1010	2024-01-01 00:00:00	180	150	295337	266.87
2	1020	2024-01-01 00:00:00	120	170	962406	997.515
5	1050	2024-01-01 00:00:00	130	230	440813	510.808
4	1040	2024-01-01 00:00:00	110	230	17190	27.331

In [ ]:
```
# select all cols excluding some
dfOrder.select(pl.col("*").exclude("CUSTOMER", "QTY")).sample(5)
```

Out[ ]: shape: (5, 5)

Row #	ID	AS_OF_DATE	ITEM	Amount in thousands
u32	i64	datetime[μs]	i64	f64
0	1000	2024-01-01 00:00:00	180	835.603
8	1080	2024-01-01 00:00:00	200	793.188
6	1060	2024-01-01 00:00:00	160	37.077
3	1030	2024-01-01 00:00:00	240	505.463
5	1050	2024-01-01 00:00:00	230	510.808

In [ ]:
```
# select certain cols
dfOrder.select(pl.col("Row #", "ID", "QTY")).sample(5)
```

Out[ ]: shape: (5, 3)

Row #	ID	QTY
u32	i64	i64
0	1000	966405
9	1090	660229
7	1070	253823
3	1030	883691
4	1040	17190

In [ ]:
```
# working with date columns
dfOrder.select(
```

```
    pl.col("AS_OF_DATE").dt.to_string("%Y-%h-%d"),
    pl.col("Row #", "ID", "QTY")
    ).sample(5)
```

Out[ ]: shape: (5, 4)

AS_OF_DATE	Row #	ID	QTY
str	u32	i64	i64
"2024-Jan-01"	0	1000	966405
"2024-Jan-01"	6	1060	581636
"2024-Jan-01"	2	1020	962406
"2024-Jan-01"	3	1030	883691
"2024-Jan-01"	9	1090	660229

In [ ]:
```
# select cols by regex
dfOrder.select(pl.col("^.*(ID|QT|Amount).*$")).sample(5)
```

Out[ ]: shape: (5, 3)

ID	QTY	Amount in thousands
i64	i64	f64
1070	253823	928.636
1010	295337	266.87
1060	581636	37.077
1080	877957	793.188
1040	17190	27.331

In [ ]:
```
# select cols by data types
print(dfOrder.sample(1)) # original - take a note of dtypes
dfOrder.select(pl.col(pl.UInt32, pl.Int64)).sample(5)
```

```
shape: (1, 7)
┌───────┬──────┬─────────────────────┬──────────┬──────┬────────┬──────────────────────
──┐
│ Row # ┆ ID   ┆ AS_OF_DATE          ┆ CUSTOMER ┆ ITEM ┆ QTY    ┆ Amount in thousand
s │
│ ---   ┆ ---  ┆ ---                 ┆ ---      ┆ ---  ┆ ---    ┆ ---
│
│ u32   ┆ i64  ┆ datetime[μs]        ┆ i64      ┆ i64  ┆ i64    ┆ f64
│
╞═══════╪══════╪═════════════════════╪══════════╪══════╪════════╪══════════════════════
══╡
│ 3     ┆ 1030 ┆ 2024-01-01 00:00:00 ┆ 100      ┆ 240  ┆ 883691 ┆ 505.463
│
└───────┴──────┴─────────────────────┴──────────┴──────┴────────┴──────────────────────
──┘
```

Out[ ]: shape: (5, 5)

Row #	ID	CUSTOMER	ITEM	QTY
u32	i64	i64	i64	i64
7	1070	140	120	253823
1	1010	180	150	295337
8	1080	160	200	877957
3	1030	100	240	883691
4	1040	110	230	17190

In [ ]:
```
# select cols by data types
print(dfOrder.sample(1)) # original - take a note of dtypes
dfOrder.select(pl.col(pl.UInt32, pl.Int64)).sample(5)
```

```
shape: (1, 7)
┌───────┬─────┬─────────────────────┬──────────┬──────┬────────┬─────────────────────
──┐
│ Row # ┆ ID  ┆ AS_OF_DATE          ┆ CUSTOMER ┆ ITEM ┆ QTY    ┆ Amount in thousand
s │
│ ---   ┆ --- ┆ ---                 ┆ ---      ┆ ---  ┆ ---    ┆ ---
│
│ u32   ┆ i64 ┆ datetime[μs]        ┆ i64      ┆ i64  ┆ i64    ┆ f64
│
╞═══════╪═════╪═════════════════════╪══════════╪══════╪════════╪═════════════════════
══╡
│ 9     ┆ 1090┆ 2024-01-01 00:00:00 ┆ 220      ┆ 170  ┆ 660229 ┆ 512.122
│
└───────┴─────┴─────────────────────┴──────────┴──────┴────────┴─────────────────────
──┘
```

Out[ ]: shape: (5, 5)

Row #	ID	CUSTOMER	ITEM	QTY
u32	i64	i64	i64	i64
5	1050	130	230	440813
1	1010	180	150	295337
2	1020	120	170	962406
7	1070	140	120	253823
6	1060	150	160	581636

In [ ]:
```
# select cols to pull unique # of column values
# for example, pull # of distint | unique customers

dfOrderSample = dfOrder.select(pl.col("CUSTOMER"))

# print row count by CUSTOMER
```

```
print(dfOrderSample.group_by("CUSTOMER").agg(pl.count()))

# print unique # of rows by CUSTOMER
print(dfOrderSample.select(pl.col("CUSTOMER")).n_unique())
```

```
shape: (9, 2)
┌──────────┬───────┐
│ CUSTOMER ┆ count │
│ ---      ┆ ---   │
│ i64      ┆ u32   │
╞══════════╪═══════╡
│ 130      ┆ 1     │
│ 120      ┆ 1     │
│ 220      ┆ 1     │
│ 110      ┆ 1     │
│ 150      ┆ 2     │
│ 180      ┆ 1     │
│ 140      ┆ 1     │
│ 160      ┆ 1     │
│ 100      ┆ 1     │
└──────────┴───────┘
9
```

In [ ]:
```
# using conditional expression

df_conditional = dfOrder.select(
    pl.col("CUSTOMER"),
    pl.when(pl.col("CUSTOMER") == 100)
    .then(pl.lit("Preferred"))
    .otherwise(pl.lit(False))
    .alias("conditional"),

)
print(df_conditional)
```

```
shape: (10, 2)
┌──────────┬─────────────┐
│ CUSTOMER ┆ conditional │
│ ---      ┆ ---         │
│ i64      ┆ str         │
╞══════════╪═════════════╡
│ 150      ┆ 0           │
│ 180      ┆ 0           │
│ 120      ┆ 0           │
│ 100      ┆ Preferred   │
│ …        ┆ …           │
│ 150      ┆ 0           │
│ 140      ┆ 0           │
│ 160      ┆ 0           │
│ 220      ┆ 0           │
└──────────┴─────────────┘
```

select columns using selectors

In [ ]:
```
########################################
## select columns using selectors
```

```
## using cs selector for column selection
##########################################

import polars.selectors as cs

dfOrder.select(pl.all())
dfOrder.select(cs.integer(), cs.string()).sample(5)
# all int and string cols

dfOrder.select(cs.numeric() - cs.first())
# all cols except first col

dfOrder.select(cs.by_name("CUSTOMER") | cs.numeric())
# col=CUSTOMER or all numeric cols

dfOrder.select(~cs.numeric())
# everything else which is not numeric

dfOrder.select(cs.contains("ID"), cs.matches(".*_.*"))
# select cols by pattern

dfOrder.select(cs.temporal().as_expr().dt.to_string("%Y-%h-%d"))
# cols by converting to expressions

#####################
# debugging selectors
#####################

# is_selector
from polars.selectors import is_selector

out = cs.temporal()
print(is_selector(out))

# selector_column_names
from polars.selectors import selector_column_names

out = cs.temporal().as_expr().dt.to_string("%Y-%h-%d")
print(out.sample(5))
```

```
True
dtype_columns([Datetime(Milliseconds, None), Datetime(Microseconds, Some("*")), Date
time(Milliseconds, Some("*")), Duration(Microseconds), Datetime(Microseconds, None),
Duration(Microseconds), Duration(Nanoseconds), Time, Date, Duration(Milliseconds), D
atetime(Nanoseconds, None), Datetime(Nanoseconds, Some("*")), Datetime(Microseconds,
None)]).to_string().sample_n()
```

Data Type Casting

In [ ]:

```
#####################
## DataType Casting #
#####################
# Polar provide casting method to convert the underlying DataType
# of a column to a new one
# strict=True # means, DataType conversion will throw an error
# strict=False # means, DataType conversion will convert value to null
```

```python
# cast
print(dfOrder.sample(5))
out = dfOrder.select(
    pl.col("ID").cast(pl.Float32).alias("integers_as_floats"),
    pl.col("QTY").cast(pl.Float32).alias("floats_as_integers"),
)
print(out)
```

```
shape: (5, 7)
┌───────┬──────┬─────────────────────┬──────────┬──────┬────────┬─────────────────────
──┐
│ Row # ┆ ID   ┆ AS_OF_DATE          ┆ CUSTOMER ┆ ITEM ┆ QTY    ┆ Amount in thousand
s │
│ ---   ┆ ---  ┆ ---                 ┆ ---      ┆ ---  ┆ ---    ┆ ---
│
│ u32   ┆ i64  ┆ datetime[μs]        ┆ i64      ┆ i64  ┆ i64    ┆ f64
│
╞═══════╪══════╪═════════════════════╪══════════╪══════╪════════╪═════════════════════
══╡
│ 8     ┆ 1080 ┆ 2024-01-01 00:00:00 ┆ 160      ┆ 200  ┆ 877957 ┆ 793.188
│
│ 1     ┆ 1010 ┆ 2024-01-01 00:00:00 ┆ 180      ┆ 150  ┆ 295337 ┆ 266.87
│
│ 2     ┆ 1020 ┆ 2024-01-01 00:00:00 ┆ 120      ┆ 170  ┆ 962406 ┆ 997.515
│
│ 7     ┆ 1070 ┆ 2024-01-01 00:00:00 ┆ 140      ┆ 120  ┆ 253823 ┆ 928.636
│
│ 4     ┆ 1040 ┆ 2024-01-01 00:00:00 ┆ 110      ┆ 230  ┆ 17190  ┆ 27.331
│
└───────┴──────┴─────────────────────┴──────────┴──────┴────────┴─────────────────────
──┘
shape: (10, 2)
┌────────────────────┬────────────────────┐
│ integers_as_floats ┆ floats_as_integers │
│ ---                ┆ ---                │
│ f32                ┆ f32                │
╞════════════════════╪════════════════════╡
│ 1000.0             ┆ 966405.0           │
│ 1010.0             ┆ 295337.0           │
│ 1020.0             ┆ 962406.0           │
│ 1030.0             ┆ 883691.0           │
│ …                  ┆ …                  │
│ 1060.0             ┆ 581636.0           │
│ 1070.0             ┆ 253823.0           │
│ 1080.0             ┆ 877957.0           │
│ 1090.0             ┆ 660229.0           │
└────────────────────┴────────────────────┘
```

In [ ]:

```python
# downncast
# casting from Int64 to Int16 and from Float64 to Float32 can be
# used to lower memory usage
out = dfOrder.select(
    pl.col("ID").cast(pl.Int16).alias("integers_smallfootprint"),
    pl.col("CUSTOMER").cast(pl.Int16).alias("floats_smallfootprint"),
    pl.col("Amount in thousands").cast(pl.Float32).alias("Amount"),
```

```
)
print(out)
```

```
shape: (10, 3)
┌─────────────────────────┬──────────────────────┬────────────┐
│ integers_smallfootprint ┆ floats_smallfootprint ┆ Amount     │
│ ---                     ┆ ---                  ┆ ---        │
│ i16                     ┆ i16                  ┆ f32        │
╞═════════════════════════╪══════════════════════╪════════════╡
│ 1000                    ┆ 150                  ┆ 835.603027 │
│ 1010                    ┆ 180                  ┆ 266.869995 │
│ 1020                    ┆ 120                  ┆ 997.515015 │
│ 1030                    ┆ 100                  ┆ 505.463013 │
│ …                       ┆ …                    ┆ …          │
│ 1060                    ┆ 150                  ┆ 37.077     │
│ 1070                    ┆ 140                  ┆ 928.635986 │
│ 1080                    ┆ 160                  ┆ 793.187988 │
│ 1090                    ┆ 220                  ┆ 512.122009 │
└─────────────────────────┴──────────────────────┴────────────┘
```

In [ ]:
```
# overflow
try:
    out = dfOrder.select(pl.col("Amount in thousands").cast(pl.Int8))
    print(out)
except Exception as e:
    print(e)

# to supress above error
# run with strict=False
out = dfOrder.select(pl.col("Amount in thousands").cast(pl.Int8, strict=False))
print(out)
```

```
strict conversion from `f64` to `i8` failed for column: Amount in thousands, value
(s) [266.87, 505.463, … 997.515]; if you were trying to cast Utf8 to temporal dtype
s, consider using `strptime`
shape: (10, 1)
┌─────────────────────┐
│ Amount in thousands │
│ ---                 │
│ i8                  │
╞═════════════════════╡
│ null                │
│ null                │
│ null                │
│ null                │
│ …                   │
│ 37                  │
│ null                │
│ null                │
│ null                │
└─────────────────────┘
```

In [ ]:
```
# strings datatype casting

# changing numeric fields to string
out = dfOrder.select(
    pl.all().exclude("ID"),
```

```
    pl.col("ID").cast(pl.Utf8)
)
print(out)

# change string back to numeric
out = dfOrder.select(
    pl.all().exclude("ID"),
    pl.col("ID").cast(pl.Int16)
)
print(out)
```

```
shape: (10, 7)
┌────────┬─────────────────────┬──────────┬──────┬────────┬─────────────────────┬──────
──┐
│ Row # ┆ AS_OF_DATE          ┆ CUSTOMER ┆ ITEM ┆ QTY    ┆ Amount in thousands ┆ ID
│
│ ---   ┆ ---                 ┆ ---      ┆ ---  ┆ ---    ┆ ---                 ┆ ---
│
│ u32   ┆ datetime[μs]        ┆ i64      ┆ i64  ┆ i64    ┆ f64                 ┆ str
│
╞════════╪═════════════════════╪══════════╪══════╪════════╪═════════════════════╪══════
══╡
│ 0     ┆ 2024-01-01 00:00:00 ┆ 150      ┆ 180  ┆ 966405 ┆ 835.603             ┆ 100
0 │
│ 1     ┆ 2024-01-01 00:00:00 ┆ 180      ┆ 150  ┆ 295337 ┆ 266.87              ┆ 101
0 │
│ 2     ┆ 2024-01-01 00:00:00 ┆ 120      ┆ 170  ┆ 962406 ┆ 997.515             ┆ 102
0 │
│ 3     ┆ 2024-01-01 00:00:00 ┆ 100      ┆ 240  ┆ 883691 ┆ 505.463             ┆ 103
0 │
│ …     ┆ …                   ┆ …        ┆ …    ┆ …      ┆ …                   ┆ …
│
│ 6     ┆ 2024-01-01 00:00:00 ┆ 150      ┆ 160  ┆ 581636 ┆ 37.077              ┆ 106
0 │
│ 7     ┆ 2024-01-01 00:00:00 ┆ 140      ┆ 120  ┆ 253823 ┆ 928.636             ┆ 107
0 │
│ 8     ┆ 2024-01-01 00:00:00 ┆ 160      ┆ 200  ┆ 877957 ┆ 793.188             ┆ 108
0 │
│ 9     ┆ 2024-01-01 00:00:00 ┆ 220      ┆ 170  ┆ 660229 ┆ 512.122             ┆ 109
0 │
└────────┴─────────────────────┴──────────┴──────┴────────┴─────────────────────┴──────
──┘
shape: (10, 7)
┌────────┬─────────────────────┬──────────┬──────┬────────┬─────────────────────┬──────
──┐
│ Row # ┆ AS_OF_DATE          ┆ CUSTOMER ┆ ITEM ┆ QTY    ┆ Amount in thousands ┆ ID
│
│ ---   ┆ ---                 ┆ ---      ┆ ---  ┆ ---    ┆ ---                 ┆ ---
│
│ u32   ┆ datetime[μs]        ┆ i64      ┆ i64  ┆ i64    ┆ f64                 ┆ i16
│
╞════════╪═════════════════════╪══════════╪══════╪════════╪═════════════════════╪══════
══╡
│ 0     ┆ 2024-01-01 00:00:00 ┆ 150      ┆ 180  ┆ 966405 ┆ 835.603             ┆ 100
0 │
│ 1     ┆ 2024-01-01 00:00:00 ┆ 180      ┆ 150  ┆ 295337 ┆ 266.87              ┆ 101
0 │
│ 2     ┆ 2024-01-01 00:00:00 ┆ 120      ┆ 170  ┆ 962406 ┆ 997.515             ┆ 102
0 │
│ 3     ┆ 2024-01-01 00:00:00 ┆ 100      ┆ 240  ┆ 883691 ┆ 505.463             ┆ 103
0 │
│ …     ┆ …                   ┆ …        ┆ …    ┆ …      ┆ …                   ┆ …
│
│ 6     ┆ 2024-01-01 00:00:00 ┆ 150      ┆ 160  ┆ 581636 ┆ 37.077              ┆ 106
0 │
│ 7     ┆ 2024-01-01 00:00:00 ┆ 140      ┆ 120  ┆ 253823 ┆ 928.636             ┆ 107
0 │
```

```
│ 8      ┆ 2024-01-01 00:00:00 ┆ 160        ┆ 200    ┆ 877957 ┆ 793.188                ┆ 108
0 │
│ 9      ┆ 2024-01-01 00:00:00 ┆ 220        ┆ 170    ┆ 660229 ┆ 512.122                ┆ 109
0 │
└────────┴─────────────────────┴────────────┴────────┴────────┴────────────────────────┴─────
──┘
```

In [ ]:
```python
df = pl.DataFrame({"strings_not_float": ["4.0", "not_a_number",
                                          "6.0", "7.0", "8.0"]})
try:
    out = df.select(pl.col("strings_not_float").cast(pl.Float64))
    print(out)
except Exception as e:
    print(e)

# as per exception, it suggests to use strptime to convert string to numeric
# this is only applicable and works well when string hold datatime values
```

```
strict conversion from `str` to `f64` failed for column: strings_not_float, value(s)
["not_a_number"]; if you were trying to cast Utf8 to temporal dtypes, consider using
`strptime`
```

In [ ]:
```python
# boolean
# Boolean cast convert all non-zero numeric (int & floats) to true
# Boolean cast convert all zero numeric (int & floats) to false

print(dfOrder.sample(4))
out = dfOrder.select(
        pl.col("Row #").cast(pl.Boolean),
        pl.col("QTY").cast(pl.Boolean),
        pl.all().exclude("Row #", "QTY"),
        )
print(out)
```

shape: (4, 7)

Row #	ID	AS_OF_DATE	CUSTOMER	ITEM	QTY	Amount in thousands
---	---	---	---	---	---	---
u32	i64	datetime[μs]	i64	i64	i64	f64
8	1080	2024-01-01 00:00:00	160	200	877957	793.188
1	1010	2024-01-01 00:00:00	180	150	295337	266.87
2	1020	2024-01-01 00:00:00	120	170	962406	997.515
9	1090	2024-01-01 00:00:00	220	170	660229	512.122

shape: (10, 7)

Row #	QTY	ID	AS_OF_DATE	CUSTOMER	ITEM	Amount in thousands
---	---	---	---	---	---	---
bool	bool	i64	datetime[μs]	i64	i64	f64
false	true	1000	2024-01-01 00:00:00	150	180	835.603
true	true	1010	2024-01-01 00:00:00	180	150	266.87
true	true	1020	2024-01-01 00:00:00	120	170	997.515
true	true	1030	2024-01-01 00:00:00	100	240	505.463
…	…	…	…	…	…	…
true	true	1060	2024-01-01 00:00:00	150	160	37.077
true	true	1070	2024-01-01 00:00:00	140	120	928.636
true	true	1080	2024-01-01 00:00:00	160	200	793.188
true	true	1090	2024-01-01 00:00:00	220	170	512.122

In [ ]:
```
# dates
out = dfOrder.select(
                    pl.col("AS_OF_DATE"),
```

```
                    pl.col("AS_OF_DATE").cast(pl.Int64).alias("DATE_as_Int"),
                    pl.col("AS_OF_DATE").cast(pl.Utf8).alias("DATE_as_Str")
                    )
print(out)
```

shape: (10, 3)

AS_OF_DATE --- datetime[μs]	DATE_as_Int --- i64	DATE_as_Str --- str
2024-01-01 00:00:00	1704067200000000	2024-01-01 00:00:00.000000
2024-01-01 00:00:00	1704067200000000	2024-01-01 00:00:00.000000
2024-01-01 00:00:00	1704067200000000	2024-01-01 00:00:00.000000
2024-01-01 00:00:00	1704067200000000	2024-01-01 00:00:00.000000
…	…	…
2024-01-01 00:00:00	1704067200000000	2024-01-01 00:00:00.000000
2024-01-01 00:00:00	1704067200000000	2024-01-01 00:00:00.000000
2024-01-01 00:00:00	1704067200000000	2024-01-01 00:00:00.000000
2024-01-01 00:00:00	1704067200000000	2024-01-01 00:00:00.000000

In [ ]:

```
# To perform casting operations between strings and Dates/Datetimes, strftime and s
# Polars adopts the chrono format syntax for when formatting.
# It's worth noting that strptime features additional options that support timezone
# Refer to the API documentation for further information.

dfOrderTmp = dfOrder.select(
                    pl.col("AS_OF_DATE"),
                    pl.col("AS_OF_DATE").cast(pl.Int64).alias("DATE_as_Int"),
                    pl.col("AS_OF_DATE").cast(pl.Utf8).alias("DATE_as_Str")
                     )
print(dfOrderTmp)

out = dfOrderTmp.select(
    pl.col("AS_OF_DATE"),
    pl.col("AS_OF_DATE").dt.strftime("%Y-%m-%d")
    .alias("AS_OF_DATE_as_strftime"),
    pl.col("DATE_as_Str").str.strptime(pl.Datetime, "%Y-%m-%d", strict=False)
)
print(out)
```

```
shape: (10, 3)
┌─────────────────────┬──────────────────┬────────────────────────────┐
│ AS_OF_DATE          ┆ DATE_as_Int      ┆ DATE_as_Str                │
│ ---                 ┆ ---              ┆ ---                        │
│ datetime[μs]        ┆ i64              ┆ str                        │
╞═════════════════════╪══════════════════╪════════════════════════════╡
│ 2024-01-01 00:00:00 ┆ 1704067200000000 ┆ 2024-01-01 00:00:00.000000 │
│ 2024-01-01 00:00:00 ┆ 1704067200000000 ┆ 2024-01-01 00:00:00.000000 │
│ 2024-01-01 00:00:00 ┆ 1704067200000000 ┆ 2024-01-01 00:00:00.000000 │
│ 2024-01-01 00:00:00 ┆ 1704067200000000 ┆ 2024-01-01 00:00:00.000000 │
│ …                   ┆ …                ┆ …                          │
│ 2024-01-01 00:00:00 ┆ 1704067200000000 ┆ 2024-01-01 00:00:00.000000 │
│ 2024-01-01 00:00:00 ┆ 1704067200000000 ┆ 2024-01-01 00:00:00.000000 │
│ 2024-01-01 00:00:00 ┆ 1704067200000000 ┆ 2024-01-01 00:00:00.000000 │
│ 2024-01-01 00:00:00 ┆ 1704067200000000 ┆ 2024-01-01 00:00:00.000000 │
└─────────────────────┴──────────────────┴────────────────────────────┘
shape: (10, 3)
┌─────────────────────┬───────────────────────┬──────────────┐
│ AS_OF_DATE          ┆ AS_OF_DATE_as_strftime ┆ DATE_as_Str  │
│ ---                 ┆ ---                   ┆ ---          │
│ datetime[μs]        ┆ str                   ┆ datetime[μs] │
╞═════════════════════╪═══════════════════════╪══════════════╡
│ 2024-01-01 00:00:00 ┆ 2024-01-01            ┆ null         │
│ 2024-01-01 00:00:00 ┆ 2024-01-01            ┆ null         │
│ 2024-01-01 00:00:00 ┆ 2024-01-01            ┆ null         │
│ 2024-01-01 00:00:00 ┆ 2024-01-01            ┆ null         │
│ …                   ┆ …                     ┆ …            │
│ 2024-01-01 00:00:00 ┆ 2024-01-01            ┆ null         │
│ 2024-01-01 00:00:00 ┆ 2024-01-01            ┆ null         │
│ 2024-01-01 00:00:00 ┆ 2024-01-01            ┆ null         │
│ 2024-01-01 00:00:00 ┆ 2024-01-01            ┆ null         │
└─────────────────────┴───────────────────────┴──────────────┘
```

working with Strings

In [ ]:
```
# working with Strings
# Polars store string as Utf8 strings
# String processing functions are available in the str namespace.

print(customer.sample(5))

out = customer.select(
    pl.all(),
    pl.col("DESCRIPTION").str.lengths().alias("DESCRIPTION_byte_count"),
    pl.col("ADDRESS").str.n_chars().alias("ADDRESS_letter_count"),
)
print(out)
```

```
shape: (5, 9)
┌─────┬─────────────────────┬─────────────┬───────────┬───┬─────────┬────────┬──────┬──────────┐
│ ID  ┆ AS_OF_DATE          ┆ DESCRIPTION ┆ ADDRESS   ┆ … ┆ EMAIL   ┆ STATUS ┆ TYPE ┆ CATEGORY │
│ --- ┆ ---                 ┆ ---         ┆ ---       ┆   ┆ ---     ┆ ---    ┆ ---  ┆ ---      │
│ i64 ┆ datetime[μs]        ┆ str         ┆ str       ┆   ┆ str     ┆ str    ┆ str  ┆ i64      │
╞═════╪═════════════════════╪═════════════╪═══════════╪═══╪═════════╪════════╪══════╪══════════╡
│ 100 ┆ 2022-01-01 00:00:00 ┆ Customer 1  ┆ Address 1 ┆ … ┆ 1@email ┆ Active ┆ Corp ┆ 1600     │
│ 230 ┆ 2022-01-01 00:00:00 ┆ Customer 4  ┆ Address 4 ┆ … ┆ 4@email ┆ Active ┆ Gov  ┆ 1600     │
│ 160 ┆ 2022-01-01 00:00:00 ┆ Customer 2  ┆ Address 2 ┆ … ┆ 2@email ┆ Active ┆ Corp ┆ 1000     │
│ 130 ┆ 2022-01-01 00:00:00 ┆ Customer 4  ┆ Address 4 ┆ … ┆ 4@email ┆ Active ┆ Corp ┆ 1200     │
│ 140 ┆ 2022-01-01 00:00:00 ┆ Customer 5  ┆ Address 5 ┆ … ┆ 5@email ┆ Active ┆ Gov  ┆ 1200     │
└─────┴─────────────────────┴─────────────┴───────────┴───┴─────────┴────────┴──────┴──────────┘
shape: (15, 11)
┌─────┬──────────────┬─────────────┬───────────┬───┬────────────┬──────────┬──────────┬─────────────┐
│ ID  ┆ AS_OF_DATE   ┆ DESCRIPTIO  ┆ ADDRESS   ┆ … ┆ TYPE       ┆ CATEGORY ┆ DESCRIPTI ┆ ADDRESS_le │
│ --- ┆ ---          ┆ N           ┆ ---       ┆   ┆ ---        ┆ ---      ┆ O         ┆ tter_count │
│ i64 ┆ datetime[μs  ┆ ---         ┆ str       ┆   ┆ str        ┆ i64      ┆ N_byte_co ┆ ---        │
│     ┆ ]            ┆ str         ┆           ┆   ┆            ┆          ┆ u         ┆ u32        │
│     ┆              ┆             ┆           ┆   ┆            ┆          ┆ nt        ┆            │
│     ┆              ┆             ┆           ┆   ┆            ┆          ┆ ---       ┆            │
│     ┆              ┆             ┆           ┆   ┆            ┆          ┆ u32       ┆            │
╞═════╪══════════════╪═════════════╪═══════════╪═══╪════════════╪══════════╪═══════════╪════════════╡
│ 100 ┆ 2022-01-01   ┆ Customer 1  ┆ Address 1 ┆ … ┆ Corp       ┆ 1600     ┆ 10        ┆ 9          │
│     ┆ 00:00:00     ┆             ┆           ┆   ┆            ┆          ┆           ┆            │
│ 110 ┆ 2022-01-01   ┆ Customer 2  ┆ Address 2 ┆ … ┆ Gov        ┆ 1200     ┆ 10        ┆ 9          │
│     ┆ 00:00:00     ┆             ┆           ┆   ┆            ┆          ┆           ┆            │
│ 120 ┆ 2022-01-01   ┆ Customer 3  ┆ Address 3 ┆ … ┆ Individual ┆ 2400     ┆ 10        ┆ 9          │
│     ┆ 00:00:00     ┆             ┆           ┆   ┆            ┆          ┆           ┆            │
│ 130 ┆ 2022-01-01   ┆ Customer 4  ┆ Address 4 ┆ … ┆ Corp       ┆ 1200     ┆ 10        ┆ 9          │
│     ┆ 00:00:00     ┆             ┆           ┆   ┆            ┆          ┆           ┆            │
│ …   ┆ …            ┆ …           ┆ …         ┆ … ┆ …          ┆ …        ┆ …         ┆ …          │
```

```
│ 210 ┆ 2022-01-01 ┆ Customer 2 ┆ Address 2 ┆ … ┆ Individual ┆ 2400 ┆ 10
┆ 9 │
│ ┆ 00:00:00 ┆ ┆ ┆ ┆ ┆ ┆
┆ │
│ 220 ┆ 2022-01-01 ┆ Customer 3 ┆ Address 3 ┆ … ┆ Corp ┆ 1900 ┆ 10
┆ 9 │
│ ┆ 00:00:00 ┆ ┆ ┆ ┆ ┆ ┆
┆ │
│ 230 ┆ 2022-01-01 ┆ Customer 4 ┆ Address 4 ┆ … ┆ Gov ┆ 1600 ┆ 10
┆ 9 │
│ ┆ 00:00:00 ┆ ┆ ┆ ┆ ┆ ┆
┆ │
│ 240 ┆ 2022-01-01 ┆ Customer 5 ┆ Address 5 ┆ … ┆ Individual ┆ 1900 ┆ 10
┆ 9 │
│ ┆ 00:00:00 ┆ ┆ ┆ ┆ ┆ ┆
┆ │
└─────┴────────────┴────────────┴───────────┴───┴────────────┴──────────┴────────
──┴──────────────┘
```

In [ ]:

```python
# string parsing
out = customer.select(
    # pl.all(),
    pl.col("ADDRESS"),
    pl.col("ADDRESS").str.contains("aDD|ress").alias("regex"),
    pl.col("ADDRESS").str.contains("Add$", literal=True).alias("literal"),
    pl.col("ADDRESS").str.starts_with("Add").alias("starts_with"),
    pl.col("ADDRESS").str.ends_with("ress").alias("ends_with"),
)
print(out)
```

shape: (15, 5)

ADDRESS --- str	regex --- bool	literal --- bool	starts_with --- bool	ends_with --- bool
Address 1	true	false	true	false
Address 2	true	false	true	false
Address 3	true	false	true	false
Address 4	true	false	true	false
…	…	…	…	…
Address 2	true	false	true	false
Address 3	true	false	true	false
Address 4	true	false	true	false
Address 5	true	false	true	false

In [ ]:

```python
# extract a pettern
df = pl.DataFrame(
    {
        "a": [
            "http://vote.com/ballon_dor?candidate=messi&ref=polars",
            "http://vote.com/ballon_dor?candidat=jorginho&ref=polars",
            "http://vote.com/ballon_dor?candidate=ronaldo&ref=polars",
        ]
    }
```

```
)
out = df.select(
    pl.col("a").str.extract(r"candidate=(\w+)", group_index=1),
)
print(out)

# extract all
df = pl.DataFrame({"foo": ["123 bla 45 asd", "xyz 678 910t"]})
out = customer.select(
    pl.col("ADDRESS").str.extract_all(r"(\d+)").alias("extracted_nrs"),
)
print(out)
```

```
shape: (3, 1)
┌─────────┐
│ a       │
│ ---     │
│ str     │
╞═════════╡
│ messi   │
│ null    │
│ ronaldo │
└─────────┘
shape: (15, 1)
┌───────────────┐
│ extracted_nrs │
│ ---           │
│ list[str]     │
╞═══════════════╡
│ ["1"]         │
│ ["2"]         │
│ ["3"]         │
│ ["4"]         │
│ …             │
│ ["2"]         │
│ ["3"]         │
│ ["4"]         │
│ ["5"]         │
└───────────────┘
```

In [ ]:

```
# replace | replace all
df = pl.DataFrame({"id": [1, 2], "text": ["123abc", "abc456"]})
out = customer.with_columns(
    pl.col("ADDRESS").str.replace(r"Address", "ADDRESS")
    .alias("text_replace"),
    pl.col("ADDRESS").str.replace_all("ress", "RESS", literal=True)
    .alias("text_replace_all"),
)
print(out)
```

```
shape: (15, 11)
┌─────┬──────────────┬─────────────┬───────────┬───┬────────────┬──────────┬───────────
──┬─────────────┐
│ ID  ┆ AS_OF_DATE   ┆ DESCRIPTIO ┆ ADDRESS   ┆ … ┆ TYPE       ┆ CATEGORY ┆ text_repl
a ┆ text_repla │
│ --- ┆ ---          ┆ N          ┆ ---       ┆   ┆ ---        ┆ ---      ┆ ce
┆ ce_all     │
│ i64 ┆ datetime[μs ┆ ---         ┆ str       ┆   ┆ str        ┆ i64      ┆ ---
┆ ---        │
│     ┆ ]            ┆ str         ┆           ┆   ┆            ┆          ┆ str
┆ str        │
╞═════╪══════════════╪═════════════╪═══════════╪═══╪════════════╪══════════╪═══════════
══╪═════════════╡
│ 100 ┆ 2022-01-01   ┆ Customer 1 ┆ Address 1 ┆ … ┆ Corp       ┆ 1600     ┆ ADDRESS 1
┆ AddRESS 1  │
│     ┆ 00:00:00     ┆             ┆           ┆   ┆            ┆          ┆
┆            │
│ 110 ┆ 2022-01-01   ┆ Customer 2 ┆ Address 2 ┆ … ┆ Gov        ┆ 1200     ┆ ADDRESS 2
┆ AddRESS 2  │
│     ┆ 00:00:00     ┆             ┆           ┆   ┆            ┆          ┆
┆            │
│ 120 ┆ 2022-01-01   ┆ Customer 3 ┆ Address 3 ┆ … ┆ Individual ┆ 2400     ┆ ADDRESS 3
┆ AddRESS 3  │
│     ┆ 00:00:00     ┆             ┆           ┆   ┆            ┆          ┆
┆            │
│ 130 ┆ 2022-01-01   ┆ Customer 4 ┆ Address 4 ┆ … ┆ Corp       ┆ 1200     ┆ ADDRESS 4
┆ AddRESS 4  │
│     ┆ 00:00:00     ┆             ┆           ┆   ┆            ┆          ┆
┆            │
│ …   ┆ …            ┆ …           ┆ …         ┆ … ┆ …          ┆ …        ┆ …
┆ …          │
│ 210 ┆ 2022-01-01   ┆ Customer 2 ┆ Address 2 ┆ … ┆ Individual ┆ 2400     ┆ ADDRESS 2
┆ AddRESS 2  │
│     ┆ 00:00:00     ┆             ┆           ┆   ┆            ┆          ┆
┆            │
│ 220 ┆ 2022-01-01   ┆ Customer 3 ┆ Address 3 ┆ … ┆ Corp       ┆ 1900     ┆ ADDRESS 3
┆ AddRESS 3  │
│     ┆ 00:00:00     ┆             ┆           ┆   ┆            ┆          ┆
┆            │
│ 230 ┆ 2022-01-01   ┆ Customer 4 ┆ Address 4 ┆ … ┆ Gov        ┆ 1600     ┆ ADDRESS 4
┆ AddRESS 4  │
│     ┆ 00:00:00     ┆             ┆           ┆   ┆            ┆          ┆
┆            │
│ 240 ┆ 2022-01-01   ┆ Customer 5 ┆ Address 5 ┆ … ┆ Individual ┆ 1900     ┆ ADDRESS 5
┆ AddRESS 5  │
│     ┆ 00:00:00     ┆             ┆           ┆   ┆            ┆          ┆
┆            │
└─────┴──────────────┴─────────────┴───────────┴───┴────────────┴──────────┴───────────
──┴─────────────┘
```

using Expression to apply Aggregation

In [ ]:
```
# most used functionality in EDA is to view data
# aggregated by different criterias
# Polars Data Frame literally shines in here and is the tool of choice
```

```
# because aggregation requires speed, parallel optimized queries

# let's review our Ledger Data Frame we created

######################
## LEDGER DataFrame ##
######################
import random
from datetime import datetime

sampleSize = 100_000
org = "ABC Inc."
ledger_type = "ACTUALS" # BUDGET, STATS are other Ledger types
fiscal_year_from = 2020
fiscal_year_to = 2023
random.seed(123)

ledger = pl.DataFrame({
        "LEDGER" : ledger_type,
        "ORG" : org,
        "FISCAL_YEAR": random.choices(list(range(fiscal_year_from,
                                          fiscal_year_to+1, 1)),k=sampleSize),
        "PERIOD": random.choices(list(range(1, 12+1, 1)),k=sampleSize),
        "ACCOUNT" : random.choices(accounts["ID"], k=sampleSize),
        "DEPT" : random.choices(dept["ID"], k=sampleSize),
        "LOCATION" : random.choices(location["ID"], k=sampleSize),
        "POSTED_TOTAL": random.sample(range(1000000), sampleSize)
})

ledger_type = "BUDGET" # ACTUALS, STATS are other Ledger types
ledger_budg = pl.DataFrame({
        "LEDGER" : ledger_type,
        "ORG" : org,
        "FISCAL_YEAR": random.choices(list(range(fiscal_year_from, fiscal_year_to+1
                               ,k=sampleSize),
        "PERIOD": random.choices(list(range(1, 12+1, 1)),k=sampleSize),
        "ACCOUNT" : random.choices(accounts["ID"], k=sampleSize),
        "DEPT" : random.choices(dept["ID"], k=sampleSize),
        "LOCATION" : random.choices(location["ID"], k=sampleSize),
        "POSTED_TOTAL": random.sample(range(1000000), sampleSize)
})

#########################################
# combined Ledger for Actuals and Budget
#########################################
dfLedger = pl.concat([ledger, ledger_budg], how="vertical")
print(dfLedger.sample(5).with_row_count("Row #"))
print(dfLedger.shape)
```

```
shape: (5, 9)
┌───────┬─────────┬───────────┬─────────────┬───┬─────────┬──────┬──────────┬─────────────┐
│ Row # ┆ LEDGER  ┆ ORG       ┆ FISCAL_YEAR ┆ … ┆ ACCOUNT ┆ DEPT ┆ LOCATION ┆ POSTED_TOTAL │
│ ---   ┆ ---     ┆ ---       ┆ ---         ┆   ┆ ---     ┆ ---  ┆ ---      ┆ ---         │
│ u32   ┆ str     ┆ str       ┆ i64         ┆   ┆ i64     ┆ i64  ┆ i64      ┆ i64         │
╞═══════╪═════════╪═══════════╪═════════════╪═══╪═════════╪══════╪══════════╪═════════════╡
│ 0     ┆ BUDGET  ┆ ABC Inc.  ┆ 2021        ┆ … ┆ 20000   ┆ 1800 ┆ 12       ┆ 725180      │
│ 1     ┆ ACTUALS ┆ ABC Inc.  ┆ 2022        ┆ … ┆ 43000   ┆ 2100 ┆ 15       ┆ 873826      │
│ 2     ┆ ACTUALS ┆ ABC Inc.  ┆ 2022        ┆ … ┆ 38000   ┆ 1300 ┆ 18       ┆ 590613      │
│ 3     ┆ ACTUALS ┆ ABC Inc.  ┆ 2022        ┆ … ┆ 30000   ┆ 1800 ┆ 12       ┆ 308302      │
│ 4     ┆ BUDGET  ┆ ABC Inc.  ┆ 2020        ┆ … ┆ 13000   ┆ 1600 ┆ 22       ┆ 251211      │
└───────┴─────────┴───────────┴─────────────┴───┴─────────┴──────┴──────────┴─────────────┘
(200000, 8)
```

In [ ]:

```
# first let's change data type of few columns
# to a categorical dtype
out = dfLedger.with_columns(
    pl.col("LEDGER").cast(pl.Categorical),
    pl.col("ORG").cast(pl.Categorical),
    (pl.col("POSTED_TOTAL") / 1000).cast(pl.Float32)
)
print(out.sample(5))
```

```
shape: (5, 8)
┌─────────┬──────────┬─────────────┬────────┬─────────┬──────┬──────────┬─────────────┐
│ LEDGER  ┆ ORG      ┆ FISCAL_YEAR ┆ PERIOD ┆ ACCOUNT ┆ DEPT ┆ LOCATION ┆ POSTED_TOTAL │
│ ---     ┆ ---      ┆ ---         ┆ ---    ┆ ---     ┆ ---  ┆ ---      ┆ ---         │
│ cat     ┆ cat      ┆ i64         ┆ i64    ┆ i64     ┆ i64  ┆ i64      ┆ f32         │
╞═════════╪══════════╪═════════════╪════════╪═════════╪══════╪══════════╪═════════════╡
│ ACTUALS ┆ ABC Inc. ┆ 2022        ┆ 1      ┆ 16000   ┆ 2200 ┆ 19       ┆ 651.127991  │
│ BUDGET  ┆ ABC Inc. ┆ 2021        ┆ 7      ┆ 14000   ┆ 1900 ┆ 15       ┆ 933.062012  │
│ ACTUALS ┆ ABC Inc. ┆ 2022        ┆ 8      ┆ 23000   ┆ 1900 ┆ 19       ┆ 667.994995  │
│ ACTUALS ┆ ABC Inc. ┆ 2022        ┆ 4      ┆ 26000   ┆ 1100 ┆ 15       ┆ 70.503998   │
│ ACTUALS ┆ ABC Inc. ┆ 2022        ┆ 7      ┆ 23000   ┆ 1000 ┆ 18       ┆ 155.164993  │
└─────────┴──────────┴─────────────┴────────┴─────────┴──────┴──────────┴─────────────┘
```

In [ ]:
```python
q = (
    out.lazy()
    .group_by("LEDGER", "FISCAL_YEAR")
    .agg(
        pl.count(),
        pl.col("PERIOD").sort(), # list of different periods
        (pl.sum("POSTED_TOTAL") / 1000000).cast(pl.Float32).alias("TOTAL (in milion
    )
    .sort("count", descending=True)
    .limit(5)
)

df = q.collect()
print(df)
```

```
shape: (5, 5)
┌─────────┬─────────────┬───────┬────────────────┬────────────────────┐
│ LEDGER  ┆ FISCAL_YEAR ┆ count ┆ PERIOD         ┆ TOTAL (in milions) │
│ ---     ┆ ---         ┆ ---   ┆ ---            ┆ ---                │
│ cat     ┆ i64         ┆ u32   ┆ list[i64]      ┆ f32                │
╞═════════╪═════════════╪═══════╪════════════════╪════════════════════╡
│ BUDGET  ┆ 2023        ┆ 25260 ┆ [1, 1, … 12]   ┆ 12.646653          │
│ ACTUALS ┆ 2023        ┆ 25136 ┆ [1, 1, … 12]   ┆ 12.526934          │
│ ACTUALS ┆ 2021        ┆ 25036 ┆ [1, 1, … 12]   ┆ 12.506885          │
│ ACTUALS ┆ 2020        ┆ 24957 ┆ [1, 1, … 12]   ┆ 12.430145          │
│ BUDGET  ┆ 2020        ┆ 24944 ┆ [1, 1, … 12]   ┆ 12.488555          │
└─────────┴─────────────┴───────┴────────────────┴────────────────────┘
```

In [ ]:
```python
# conditional aggregation
# Let's say we want to know counts by Quarters.
# We could directly query that in the aggregation without the need of a lambda
# or grooming the DataFrame.
```

```
out = dfLedger.with_columns(
    pl.col("LEDGER").cast(pl.Categorical),
    pl.col("ORG").cast(pl.Categorical),
    (pl.col("POSTED_TOTAL") / 1000).cast(pl.Float32)
)
q = (
    out.lazy()
    .groupby("LEDGER", "FISCAL_YEAR")
    .agg(
        pl.count(),
        ((pl.col("PERIOD") >= 1) & (pl.col("PERIOD") <= 3)).sum().cast(pl.Int32).al
        ((pl.col("PERIOD") >= 4) & (pl.col("PERIOD") <= 6)).sum().cast(pl.Int32).al
        ((pl.col("PERIOD") >= 7) & (pl.col("PERIOD") <= 9)).sum().cast(pl.Int32).al
        ((pl.col("PERIOD") >= 10) & (pl.col("PERIOD") <= 12)).sum().cast(pl.Int32).
        (pl.sum("POSTED_TOTAL") / 1000000).cast(pl.Float32).alias("TOTAL (in milion
    )
    .sort("FISCAL_YEAR", descending=True)
    .limit(5)
)

df = q.collect()
print(df)
```

```
shape: (5, 8)
┌─────────┬─────────────┬───────┬───────────┬───────────┬───────────┬───────────┬───
─────────┐
│ LEDGER  ┆ FISCAL_YEAR ┆ count ┆ Q1 counts ┆ Q2 counts ┆ Q3 counts ┆ Q4 counts ┆ TO
TAL (in │
│ ---     ┆ ---         ┆ ---   ┆ ---       ┆ ---       ┆ ---       ┆ ---       ┆ mi
lions)  │
│ cat     ┆ i64         ┆ u32   ┆ i32       ┆ i32       ┆ i32       ┆ i32       ┆ --
-       │
│         ┆             ┆       ┆           ┆           ┆           ┆           ┆ f3
2       │
╞═════════╪═════════════╪═══════╪═══════════╪═══════════╪═══════════╪═══════════╪═══
═════════╡
│ ACTUALS ┆ 2023        ┆ 25136 ┆ 6339      ┆ 6377      ┆ 6222      ┆ 6198      ┆ 1
2.526934 │
│ BUDGET  ┆ 2023        ┆ 25260 ┆ 6221      ┆ 6298      ┆ 6406      ┆ 6335      ┆ 1
2.646653 │
│ ACTUALS ┆ 2022        ┆ 24871 ┆ 6284      ┆ 6138      ┆ 6242      ┆ 6207      ┆ 1
2.420824 │
│ BUDGET  ┆ 2022        ┆ 24877 ┆ 6228      ┆ 6228      ┆ 6248      ┆ 6173      ┆ 1
2.404573 │
│ ACTUALS ┆ 2021        ┆ 25036 ┆ 6305      ┆ 6203      ┆ 6187      ┆ 6341      ┆ 1
2.506885 │
└─────────┴─────────────┴───────┴───────────┴───────────┴───────────┴───────────┴───
─────────┘
```

In [ ]:

```
# filtering
# We can also filter the groups. Let's say we want to compute a mean per group,
# but we don't want to include all values from that group,
# and we also don't want to filter the rows from the DataFrame
# (because we need those rows for another aggregation).

# In the example below we show how this can be done.
```

```
def avg_by_qtr(fromMonth: int, toMonth: int) -> pl.Expr:
    return (
        (pl.col("POSTED_TOTAL")).filter(((pl.col("PERIOD") >= fromMonth) & (pl.col(
        .mean()
        .alias(f"avg {fromMonth}-{toMonth} Month")
    )

q = (
    out.lazy()
    .group_by("LEDGER", "FISCAL_YEAR")
    .agg(
        pl.count(),
        # ((pl.col("PERIOD") >= 1) & (pl.col("PERIOD") <= 3)).sum().cast(pl.Int32).
        # ((pl.col("PERIOD") >= 4) & (pl.col("PERIOD") <= 6)).sum().cast(pl.Int32).
        # ((pl.col("PERIOD") >= 7) & (pl.col("PERIOD") <= 9)).sum().cast(pl.Int32).
        # ((pl.col("PERIOD") >= 10) & (pl.col("PERIOD") <= 12)).sum().cast(pl.Int32
        avg_by_qtr(1,3),
        avg_by_qtr(4,6),
        avg_by_qtr(7,9),
        avg_by_qtr(10,12),
        (pl.sum("POSTED_TOTAL") / 1000000).cast(pl.Float32).alias("TOTAL (in milion
    )
    .sort("FISCAL_YEAR", descending=True)
    .limit(5)
)

df = q.collect()
print(df)
```

```
shape: (5, 8)
┌─────────┬─────────────┬───────┬────────────┬────────────┬────────────┬───────────
──┬──────────────┐
│ LEDGER  ┆ FISCAL_YEAR ┆ count ┆ avg 1-3    ┆ avg 4-6    ┆ avg 7-9    ┆ avg 10-12
┆ TOTAL (in   │
│ ---     ┆ ---         ┆ ---   ┆ Month      ┆ Month      ┆ Month      ┆ Month
┆ milions)    │
│ cat     ┆ i64         ┆ u32   ┆ ---        ┆ ---        ┆ ---        ┆ ---
┆ ---         │
│         ┆             ┆       ┆ f32        ┆ f32        ┆ f32        ┆ f32
┆ f32         │
╞═════════╪═════════════╪═══════╪════════════╪════════════╪════════════╪═══════════
══╪══════════════╡
│ ACTUALS ┆ 2023        ┆ 25136 ┆ 489.695587 ┆ 497.554016 ┆ 504.889496 ┆ 501.51684
6 ┆ 12.526934   │
│ BUDGET  ┆ 2023        ┆ 25260 ┆ 497.987671 ┆ 503.527313 ┆ 496.942291 ┆ 504.19125
4 ┆ 12.646653   │
│ ACTUALS ┆ 2022        ┆ 24871 ┆ 500.281952 ┆ 505.403595 ┆ 494.125092 ┆ 497.91458
1 ┆ 12.420824   │
│ BUDGET  ┆ 2022        ┆ 24877 ┆ 502.2229   ┆ 496.800751 ┆ 494.446503 ┆ 501.10818
5 ┆ 12.404573   │
│ ACTUALS ┆ 2021        ┆ 25036 ┆ 499.123901 ┆ 503.51593  ┆ 497.496216 ┆ 498.11737
1 ┆ 12.506885   │
└─────────┴─────────────┴───────┴────────────┴────────────┴────────────┴───────────
──┴──────────────┘
```

```
In [ ]: # Do not kill parallelization

# Python Users Only

# The following section is specific to Python,
# and doesn't apply to Rust. Within Rust, blocks and closures (lambdas) can,
# and will, be executed concurrently.

# We have all heard that Python is slow, and does "not scale."
# Besides the overhead of running "slow" bytecode, Python has to remain
# within the constraints of the Global Interpreter Lock (GIL).
# This means that if you were to use a lambda or a custom Python function to
# apply during a parallelized phase, Polars speed is capped running Python code
# preventing any multiple threads from executing the function.

# This all feels terribly limiting,
# especially because we often need those lambda functions in a .groupby() step,
# for example. This approach is still supported by Polars,
# but keeping in mind bytecode and the GIL costs have to be paid.
# It is recommended to try to solve your queries using the expression syntax
# before moving to lambdas.
# If you want to learn more about using lambdas, go to the user defined functions s

# Conclusion

# In the examples above we've seen that we can do a lot by combining expressions.
# By doing so we delay the use of custom Python functions
# that slow down the queries (by the slow nature of Python AND the GIL).

# If we are missing a type expression let us know by opening a feature request!
```

using Expression to handle missing data

```
In [ ]: # let's review two example scenarios

# 1st case: one period/month of data is missing from dataset

# another useful example, is user want to prepare budgets
# based on actuals

# 2nd use case: create datasets based on FILL strategy

# check out few samples of Ledger Data Frame
out = dfLedger.with_columns(
    pl.col("LEDGER").cast(pl.Categorical),
    pl.col("ORG").cast(pl.Categorical),
    (pl.col("POSTED_TOTAL") / 1000).cast(pl.Float32)
).sort("FISCAL_YEAR", "PERIOD")
print(out.head(5))

# count of rows for ACCOUNT = 10000, LOCATION = 14 & PERIOD = 12
print("Rows|Columns: ", out.filter((pl.col("ACCOUNT") == 10000)
                 & (pl.col("LOCATION") == 14)
                 & (pl.col("PERIOD") == 12)).shape)
```

```
outRevised = dfLedger.with_columns(
    pl.col("LEDGER").cast(pl.Categorical),
    pl.col("ORG").cast(pl.Categorical),
    (pl.col("POSTED_TOTAL") / 1000).cast(pl.Float32),
    pl.when(
        (pl.col("ACCOUNT") == 10000)
                 & (pl.col("LOCATION") == 14)
                 & (pl.col("PERIOD") == 12)
        )
    .then(None)
    .otherwise((pl.col("POSTED_TOTAL") / 1000).cast(pl.Float32))
    .alias("POSTED_TOTAL_rev")
).sort("FISCAL_YEAR", "PERIOD")

print("Rows|Columns: ", outRevised.filter((pl.col("ACCOUNT") == 10000)
                 & (pl.col("LOCATION") == 14)
                 & (pl.col("PERIOD") == 12)).sample(5))
```

```
shape: (5, 8)
┌─────────┬──────────┬─────────────┬────────┬─────────┬──────┬──────────┬────────────
───┐
│ LEDGER  ┆ ORG      ┆ FISCAL_YEAR ┆ PERIOD ┆ ACCOUNT ┆ DEPT ┆ LOCATION ┆ POSTED_TOT
AL │
│ ---     ┆ ---      ┆ ---         ┆ ---    ┆ ---     ┆ ---  ┆ ---      ┆ ---
│
│ cat     ┆ cat      ┆ i64         ┆ i64    ┆ i64     ┆ i64  ┆ i64      ┆ f32
│
╞═════════╪══════════╪═════════════╪════════╪═════════╪══════╪══════════╪════════════
═══╡
│ ACTUALS ┆ ABC Inc. ┆ 2020        ┆ 1      ┆ 10000   ┆ 1900 ┆ 14       ┆ 826.906006
│
│ ACTUALS ┆ ABC Inc. ┆ 2020        ┆ 1      ┆ 17000   ┆ 1400 ┆ 16       ┆ 725.926025
│
│ ACTUALS ┆ ABC Inc. ┆ 2020        ┆ 1      ┆ 29000   ┆ 1900 ┆ 18       ┆ 127.103996
│
│ ACTUALS ┆ ABC Inc. ┆ 2020        ┆ 1      ┆ 42000   ┆ 1400 ┆ 17       ┆ 198.975006
│
│ ACTUALS ┆ ABC Inc. ┆ 2020        ┆ 1      ┆ 39000   ┆ 1100 ┆ 22       ┆ 440.190002
│
└─────────┴──────────┴─────────────┴────────┴─────────┴──────┴──────────┴────────────
───┘
Rows|Columns:  (38, 8)
Rows|Columns:  shape: (5, 9)
┌─────────┬──────────┬─────────────┬────────┬───┬──────┬──────────┬──────────────┬──
────────────────┐
│ LEDGER  ┆ ORG      ┆ FISCAL_YEAR ┆ PERIOD ┆ … ┆ DEPT ┆ LOCATION ┆ POSTED_TOTAL ┆ P
OSTED_TOTAL_re │
│ ---     ┆ ---      ┆ ---         ┆ ---    ┆   ┆ ---  ┆ ---      ┆ ---          ┆ v
│
│ cat     ┆ cat      ┆ i64         ┆ i64    ┆   ┆ i64  ┆ i64      ┆ f32          ┆ -
--             │
│         ┆          ┆             ┆        ┆   ┆      ┆          ┆              ┆ f
32             │
╞═════════╪══════════╪═════════════╪════════╪═══╪══════╪══════════╪══════════════╪══
═══════════════╡
│ BUDGET  ┆ ABC Inc. ┆ 2022        ┆ 12     ┆ … ┆ 1700 ┆ 14       ┆ 869.401001   ┆ n
ull            │
│ ACTUALS ┆ ABC Inc. ┆ 2022        ┆ 12     ┆ … ┆ 2400 ┆ 14       ┆ 833.10498    ┆ n
ull            │
│ BUDGET  ┆ ABC Inc. ┆ 2021        ┆ 12     ┆ … ┆ 1300 ┆ 14       ┆ 963.809021   ┆ n
ull            │
│ ACTUALS ┆ ABC Inc. ┆ 2020        ┆ 12     ┆ … ┆ 1800 ┆ 14       ┆ 416.033997   ┆ n
ull            │
│ BUDGET  ┆ ABC Inc. ┆ 2022        ┆ 12     ┆ … ┆ 1700 ┆ 14       ┆ 917.286987   ┆ n
ull            │
└─────────┴──────────┴─────────────┴────────┴───┴──────┴──────────┴──────────────┴──
────────────────┘
```

In [ ]:
```
# display counts of Null rows
outRevised.null_count()
```

Out[ ]: shape: (1, 9)

LEDGER	ORG	FISCAL_YEAR	PERIOD	ACCOUNT	DEPT	LOCATION	POSTED_TOTAL	POST
u32	u32	u32	u32	u32	u32	u32	u32	
0	0	0	0	0	0	0	0	

In [ ]:
```
# Filling missing data

# Fill with specified literal value
outRevised_1 = outRevised.with_columns(
        pl.col("POSTED_TOTAL_rev").fill_null(
            pl.lit(0.0),
        ),
    )
print("Rows|Columns: ", outRevised_1.filter((pl.col("ACCOUNT") == 10000)
                 & (pl.col("LOCATION") == 14)
                 & (pl.col("PERIOD") == 12)).sample(5))
```

```
Rows|Columns:  shape: (5, 9)
┌─────────┬──────────┬─────────────┬────────┬───┬──────┬──────────┬──────────────┬──
───────────────┐
│ LEDGER  ┆ ORG      ┆ FISCAL_YEAR ┆ PERIOD ┆ … ┆ DEPT ┆ LOCATION ┆ POSTED_TOTAL ┆ P
OSTED_TOTAL_re │
│ ---     ┆ ---      ┆ ---         ┆ ---    ┆   ┆ ---  ┆ ---      ┆ ---          ┆ v
│
│ cat     ┆ cat      ┆ i64         ┆ i64    ┆   ┆ i64  ┆ i64      ┆ f32          ┆ -
--             │
│         ┆          ┆             ┆        ┆   ┆      ┆          ┆              ┆ f
32             │
╞═════════╪══════════╪═════════════╪════════╪═══╪══════╪══════════╪══════════════╪══
═══════════════╡
│ BUDGET  ┆ ABC Inc. ┆ 2023        ┆ 12     ┆ … ┆ 2400 ┆ 14       ┆ 658.724976   ┆
0.0              │
│ BUDGET  ┆ ABC Inc. ┆ 2022        ┆ 12     ┆ … ┆ 1700 ┆ 14       ┆ 855.356995   ┆
0.0              │
│ ACTUALS ┆ ABC Inc. ┆ 2021        ┆ 12     ┆ … ┆ 1400 ┆ 14       ┆ 629.336975   ┆
0.0              │
│ BUDGET  ┆ ABC Inc. ┆ 2021        ┆ 12     ┆ … ┆ 2000 ┆ 14       ┆ 78.411003    ┆
0.0              │
│ ACTUALS ┆ ABC Inc. ┆ 2022        ┆ 12     ┆ … ┆ 1000 ┆ 14       ┆ 794.187012   ┆
0.0              │
└─────────┴──────────┴─────────────┴────────┴───┴──────┴──────────┴──────────────┴──
───────────────┘
```

In [ ]:
```
# Filling missing data

# Fill with a strategy
outRevised_2 = outRevised.with_columns(
        pl.col("POSTED_TOTAL_rev").fill_null(
            strategy="forward"
        ),
    )
print("Rows|Columns: ", outRevised_2.filter((pl.col("ACCOUNT") == 10000)
```

```
                 & (pl.col("LOCATION") == 14)
                 & (pl.col("PERIOD") == 12)).sample(5))
```

```
Rows|Columns:  shape: (5, 9)
┌─────────┬──────────┬─────────────┬────────┬───┬──────┬──────────┬──────────────┬──
────────────────┐
│ LEDGER  ┆ ORG      ┆ FISCAL_YEAR ┆ PERIOD ┆ … ┆ DEPT ┆ LOCATION ┆ POSTED_TOTAL ┆ P
OSTED_TOTAL_re │
│ ---     ┆ ---      ┆ ---         ┆ ---    ┆   ┆ ---  ┆ ---      ┆ ---          ┆ v
│
│ cat     ┆ cat      ┆ i64         ┆ i64    ┆   ┆ i64  ┆ i64      ┆ f32          ┆ -
--             │
│         ┆          ┆             ┆        ┆   ┆      ┆          ┆              ┆ f
32             │
╞═════════╪══════════╪═════════════╪════════╪═══╪══════╪══════════╪══════════════╪══
════════════════╡
│ ACTUALS ┆ ABC Inc. ┆ 2022        ┆ 12     ┆ … ┆ 1100 ┆ 14       ┆ 944.950012   ┆ 1
45.623993      │
│ BUDGET  ┆ ABC Inc. ┆ 2023        ┆ 12     ┆ … ┆ 1300 ┆ 14       ┆ 515.521973   ┆ 5
3.018002       │
│ BUDGET  ┆ ABC Inc. ┆ 2021        ┆ 12     ┆ … ┆ 2200 ┆ 14       ┆ 112.551003   ┆ 8
29.671997      │
│ BUDGET  ┆ ABC Inc. ┆ 2020        ┆ 12     ┆ … ┆ 1500 ┆ 14       ┆ 142.300995   ┆ 1
19.074997      │
│ BUDGET  ┆ ABC Inc. ┆ 2021        ┆ 12     ┆ … ┆ 2000 ┆ 14       ┆ 78.411003    ┆ 9
48.60199       │
└─────────┴──────────┴─────────────┴────────┴───┴──────┴──────────┴──────────────┴──
────────────────┘
```

In [ ]:

```
# Filling missing data

# Fill with an expression
outRevised_3 = outRevised.with_columns(
        pl.col("POSTED_TOTAL_rev").fill_null(
            pl.median("POSTED_TOTAL_rev")
        ),
    )
print("Rows|Columns: ", outRevised_3.filter((pl.col("ACCOUNT") == 10000)
                 & (pl.col("LOCATION") == 14)
                 & (pl.col("PERIOD") == 12)).sample(5))
```

```
Rows|Columns:  shape: (5, 9)
┌─────────┬─────────┬────────────┬────────┬───┬──────┬──────────┬──────────────┬──
────────────────┐
│ LEDGER  ┆ ORG     ┆ FISCAL_YEAR ┆ PERIOD ┆ … ┆ DEPT ┆ LOCATION ┆ POSTED_TOTAL ┆ P
OSTED_TOTAL_re │
│ ---     ┆ ---     ┆ ---        ┆ ---    ┆   ┆ ---  ┆ ---      ┆ ---          ┆ v
│
│ cat     ┆ cat     ┆ i64        ┆ i64    ┆   ┆ i64  ┆ i64      ┆ f32          ┆ -
--             │
│         ┆         ┆            ┆        ┆   ┆      ┆          ┆              ┆ f
32             │
╞═════════╪═════════╪════════════╪════════╪═══╪══════╪══════════╪══════════════╪══
═══════════════╡
│ ACTUALS ┆ ABC Inc. ┆ 2020      ┆ 12     ┆ … ┆ 1900 ┆ 14       ┆ 368.372009   ┆ 5
01.114014      │
│ ACTUALS ┆ ABC Inc. ┆ 2020      ┆ 12     ┆ … ┆ 1100 ┆ 14       ┆ 637.629028   ┆ 5
01.114014      │
│ BUDGET  ┆ ABC Inc. ┆ 2021      ┆ 12     ┆ … ┆ 2200 ┆ 14       ┆ 112.551003   ┆ 5
01.114014      │
│ BUDGET  ┆ ABC Inc. ┆ 2022      ┆ 12     ┆ … ┆ 1700 ┆ 14       ┆ 917.286987   ┆ 5
01.114014      │
│ BUDGET  ┆ ABC Inc. ┆ 2021      ┆ 12     ┆ … ┆ 1000 ┆ 14       ┆ 79.495003    ┆ 5
01.114014      │
└─────────┴─────────┴────────────┴────────┴───┴──────┴──────────┴──────────────┴──
────────────────┘
```

In [ ]:

```python
# Filling missing data

# Fill with interpolation
outRevised_4 = outRevised.with_columns(
        pl.col("POSTED_TOTAL_rev").fill_null(
            pl.col("POSTED_TOTAL_rev").interpolate(),
        ),
    )
print("Rows|Columns: ", outRevised_4.filter((pl.col("ACCOUNT") == 10000)
                 & (pl.col("LOCATION") == 14)
                 & (pl.col("PERIOD") == 12)).sample(5))
```

```
Rows|Columns:  shape: (5, 9)
┌─────────┬──────────┬─────────────┬────────┬───┬──────┬──────────┬──────────────┬──
────────────────┐
│ LEDGER  ┆ ORG      ┆ FISCAL_YEAR ┆ PERIOD ┆ … ┆ DEPT ┆ LOCATION ┆ POSTED_TOTAL ┆ P
OSTED_TOTAL_re │
│ ---     ┆ ---      ┆ ---         ┆ ---    ┆   ┆ ---  ┆ ---      ┆ ---          ┆ v
│
│ cat     ┆ cat      ┆ i64         ┆ i64    ┆   ┆ i64  ┆ i64      ┆ f32          ┆ -
--             │
│         ┆          ┆             ┆        ┆   ┆      ┆          ┆              ┆ f
32             │
╞═════════╪══════════╪═════════════╪════════╪═══╪══════╪══════════╪══════════════╪══
═══════════════╡
│ ACTUALS ┆ ABC Inc. ┆ 2021        ┆ 12     ┆ … ┆ 2000 ┆ 14       ┆ 401.223999   ┆ 5
3.867001       │
│ BUDGET  ┆ ABC Inc. ┆ 2023        ┆ 12     ┆ … ┆ 1300 ┆ 14       ┆ 515.521973   ┆ 2
33.958008      │
│ ACTUALS ┆ ABC Inc. ┆ 2020        ┆ 12     ┆ … ┆ 1100 ┆ 14       ┆ 886.544983   ┆ 5
92.49353       │
│ BUDGET  ┆ ABC Inc. ┆ 2021        ┆ 12     ┆ … ┆ 2300 ┆ 14       ┆ 977.153015   ┆ 3
42.100006      │
│ BUDGET  ┆ ABC Inc. ┆ 2023        ┆ 12     ┆ … ┆ 1400 ┆ 14       ┆ 809.125      ┆ 4
68.296997      │
└─────────┴──────────┴─────────────┴────────┴───┴──────┴──────────┴──────────────┴──
───────────────┘
```

using Expression to apply Folds

Polars provides expressions/methods for horizontal aggregations like sum,min, mean, etc. However, when you need a more complex aggregation the default methods Polars supplies may not be sufficient. That's when folds come in handy.

The fold expression operates on columns for maximum speed. It utilizes the data layout very efficiently and often has vectorized execution.

In [ ]:
```python
# basic calculations using Polars expressions
df = pl.DataFrame(
    {
        "a": [1, 2, 3],
        "b": [10, 20, 30],
    }
)

out = df.select(
    (pl.col("a") + pl.col("b")).alias("sum"),
    pl.col("a").mean().alias("mean"),
    pl.col("a").min().alias("min"),
    pl.col("a").max().alias("max")
    )
out
```

Out[ ]: shape: (3, 4)

sum	mean	min	max
i64	f64	i64	i64
11	2.0	1	3
22	2.0	1	3
33	2.0	1	3

In [ ]:
```
# using Polars Folds to perform complex algerbic aggregations
out = df.select(
    pl.fold(acc=pl.lit(0), function=lambda acc, x: acc + x, exprs=pl.all()).alias(
        "sum"
    ),
)
out
```

Out[ ]: shape: (3, 1)

sum
i64
11
22
33

In [ ]:
```
# Let's use Polars Folds to perform a conditional Algebric operation
# below sample DataFrame produces a Balance Sheet,
# however, NETWORTH calculated is wrong


############################
## BalanceSheet DataFrame ##
############################
import random
from datetime import datetime

sampleSize = 100_000
org = "ABC Inc."
ledger_type = "ACTUALS" # BUDGET, STATS are other Ledger types
fiscal_year_from = 2020
fiscal_year_to = 2023
random.seed(123)

balanceSheet = pl.DataFrame({
        "LEDGER" : ledger_type,
        "ORG" : org,
        "FISCAL_YEAR": random.choices(list(range(fiscal_year_from,
                                          fiscal_year_to+1, 1)),k=sampleSize),
        "PERIOD": random.choices(list(range(1, 12+1, 1)),k=sampleSize),
        "ASSETS": random.sample(range(1000000), sampleSize),
```

```
    "LIABILITY": random.sample(range(1000000), sampleSize),
    "REVENUE": random.sample(range(1000000), sampleSize),
    "NETWORTH": random.sample(range(1000000), sampleSize),
})
print(balanceSheet.sample(5).with_row_count("Row #"))
```

```
shape: (5, 9)
┌───────┬─────────┬──────────┬─────────────┬───┬────────┬───────────┬─────────┬────────
──────┐
│ Row # ┆ LEDGER  ┆ ORG      ┆ FISCAL_YEAR ┆ … ┆ ASSETS ┆ LIABILITY ┆ REVENUE ┆ NETW
ORTH │
│ ---   ┆ ---     ┆ ---      ┆ ---         ┆   ┆ ---    ┆ ---       ┆ ---     ┆ ---
│
│ u32   ┆ str     ┆ str      ┆ i64         ┆   ┆ i64    ┆ i64       ┆ i64     ┆ i64
│
╞═══════╪═════════╪══════════╪═════════════╪═══╪════════╪═══════════╪═════════╪════════
══════╡
│ 0     ┆ ACTUALS ┆ ABC Inc. ┆ 2022        ┆ … ┆ 229177 ┆ 162079    ┆ 746304  ┆ 9345
36    │
│ 1     ┆ ACTUALS ┆ ABC Inc. ┆ 2022        ┆ … ┆ 58165  ┆ 645852    ┆ 795922  ┆ 3720
16    │
│ 2     ┆ ACTUALS ┆ ABC Inc. ┆ 2020        ┆ … ┆ 457983 ┆ 853620    ┆ 540738  ┆ 5135
16    │
│ 3     ┆ ACTUALS ┆ ABC Inc. ┆ 2021        ┆ … ┆ 949507 ┆ 907875    ┆ 28191   ┆ 7655
34    │
│ 4     ┆ ACTUALS ┆ ABC Inc. ┆ 2020        ┆ … ┆ 848681 ┆ 419144    ┆ 732089  ┆ 5115
68    │
└───────┴─────────┴──────────┴─────────────┴───┴────────┴───────────┴─────────┴────────
──────┘
```

In [ ]:

```
# let's use Polars Folds to
# A. calculate NetWorth_c = ASSETS + REVENUE - LIABILITY
# B. generate a unique COSTCenter String = LEDGER+ORG+FISCAL_YEAR

out = balanceSheet.with_columns(
    pl.fold(acc=pl.lit(0),
            function=lambda acc, x: acc + x,
            exprs=(pl.col("ASSETS","REVENUE"), pl.col("LIABILITY")*-1))
            .alias("NetWorth_c"),
    pl.fold(acc=pl.lit(""),
            function=lambda acc, x: acc + x,
            exprs=(pl.col("LEDGER","ORG","FISCAL_YEAR"))
            .alias("CostCenter"),
    )
)
print(out.head)
```

```
<bound method DataFrame.head of shape: (100_000, 10)
┌─────────┬──────────┬─────────────┬────────┬───┬─────────┬──────────┬────────────┬─
───────────┐
│ LEDGER  ┆ ORG      ┆ FISCAL_YEAR ┆ PERIOD ┆ … ┆ REVENUE ┆ NETWORTH ┆ NetWorth_c ┆
CostCenter │
│ ---     ┆ ---      ┆ ---         ┆ ---    ┆   ┆ ---     ┆ ---      ┆ ---        ┆
---        │
│ str     ┆ str      ┆ i64         ┆ i64    ┆   ┆ i64     ┆ i64      ┆ i64        ┆
str        │
╞═════════╪══════════╪═════════════╪════════╪═══╪═════════╪══════════╪════════════╪═
═══════════╡
│ ACTUALS ┆ ABC Inc. ┆ 2020        ┆ 10     ┆ … ┆ 825234  ┆ 630949   ┆ 358184     ┆
ACTUALSABC │
│         ┆          ┆             ┆        ┆   ┆         ┆          ┆            ┆
Inc.2020   │
│ ACTUALS ┆ ABC Inc. ┆ 2020        ┆ 1      ┆ … ┆ 401531  ┆ 307809   ┆ -289400    ┆
ACTUALSABC │
│         ┆          ┆             ┆        ┆   ┆         ┆          ┆            ┆
Inc.2020   │
│ ACTUALS ┆ ABC Inc. ┆ 2021        ┆ 12     ┆ … ┆ 777927  ┆ 536595   ┆ 597339     ┆
ACTUALSABC │
│         ┆          ┆             ┆        ┆   ┆         ┆          ┆            ┆
Inc.2021   │
│ ACTUALS ┆ ABC Inc. ┆ 2020        ┆ 3      ┆ … ┆ 445913  ┆ 400960   ┆ 398951     ┆
ACTUALSABC │
│         ┆          ┆             ┆        ┆   ┆         ┆          ┆            ┆
Inc.2020   │
│ …       ┆ …        ┆ …           ┆ …      ┆ … ┆ …       ┆ …        ┆ …          ┆
…          │
│ ACTUALS ┆ ABC Inc. ┆ 2022        ┆ 6      ┆ … ┆ 812724  ┆ 541937   ┆ 851037     ┆
ACTUALSABC │
│         ┆          ┆             ┆        ┆   ┆         ┆          ┆            ┆
Inc.2022   │
│ ACTUALS ┆ ABC Inc. ┆ 2023        ┆ 8      ┆ … ┆ 188039  ┆ 876970   ┆ 301553     ┆
ACTUALSABC │
│         ┆          ┆             ┆        ┆   ┆         ┆          ┆            ┆
Inc.2023   │
│ ACTUALS ┆ ABC Inc. ┆ 2023        ┆ 8      ┆ … ┆ 610809  ┆ 715951   ┆ -100056    ┆
ACTUALSABC │
│         ┆          ┆             ┆        ┆   ┆         ┆          ┆            ┆
Inc.2023   │
│ ACTUALS ┆ ABC Inc. ┆ 2020        ┆ 2      ┆ … ┆ 783855  ┆ 226190   ┆ 667217     ┆
ACTUALSABC │
│         ┆          ┆             ┆        ┆   ┆         ┆          ┆            ┆
Inc.2020   │
└─────────┴──────────┴─────────────┴────────┴───┴─────────┴──────────┴────────────┴─
───────────┘>
```

using Expression to apply List & Arrays

Polars has first class support for List columns. Please note that, Polars List Column is different than Python list data type.

For example, Python list data type can store data of any type, however, Polars List column contains each row as same data type.

Polars can still keep different type in List column, but stores it as `Object` data type instead of list, hence List data type manipulations is not available.

Polars also support Array data type which is analogous to numpy ndarray.

Let's see these in action, so that it become more clear.

In [ ]:
```
########################
## CUSTOMER DataFrame ##
########################

import random
from datetime import datetime

customer = pl.DataFrame({
    "ID":  list(range(100, 250, 10)),
    "AS_OF_DATE" : datetime(2022, 1, 1),
    "DESCRIPTION" : ["Customer 1","Customer 2","Customer 3","Customer 4","Customer
    "ADDRESS" : [["Address 1a, Address 1b"],["Address 2a, Address 2b"],
                 ["Address 3a, Address 3b"],["Address 4a, Address 4b"],
                 ["Address 5a, Address 5b"]] * 3,
    "PHONE1" : [[100100,100200],[200100,200200],[300100,300200],
                [400100,400200],[500100,500200]] * 3,
    "PHONE2" : ["100100 | 100200","200100 | 200200","300100 | 300200",
                "400100 | 400200","500100 | 500200"] * 3,
    "PHONE3" : [["100100 | 100200 | 100300 | 100400 | 100500"],
                ["200100 | 200200 | 200300 | 200400 | 200500"],
                ["300100 | 300200 | 300300 | 300400 | 300500"],
                ["400100 | 300200 | 400300 | 400400 | 400500"],
                ["500100 | 500200 | 500300 | 500400 | 500500"],
                ]
                * 3,
    "EMAIL" : ["1a@email 1b@email","2a@email 2b@email","3a@email 3b@email",
               "4a@email 4b@email","5a@email 5b@email donotreply@email"] * 3,
    "STATUS" : "Active",
    "TYPE" : ["Corp","Gov","Individual"] * 5,
    "CATEGORY" : random.choices(category["ID"], k=15),
})
customer.sample(5).with_row_count("Row #")
# Address, PHONE1, PHONE3, EMAIL are lists of [str, int, str, str]
# PHONE2, EMAIL contains a list, but is saved as str
```

Out[ ]: shape: (5, 12)

Row #	ID	AS_OF_DATE	DESCRIPTION	ADDRESS	PHONE1	PHONE2	PHONE3	EMAIL
u32	i64	datetime[μs]	str	list[str]	list[i64]	str	list[str]	str
0	100	2022-01-01 00:00:00	"Customer 1"	["Address 1a, Address 1b"]	[100100, 100200]	"100100 | 10020...	["100100 | 100200 | 100300 | 100400 | 100500"]	"1a@email 1b@em...
1	200	2022-01-01 00:00:00	"Customer 1"	["Address 1a, Address 1b"]	[100100, 100200]	"100100 | 10020...	["100100 | 100200 | 100300 | 100400 | 100500"]	"1a@email 1b@em...
2	210	2022-01-01 00:00:00	"Customer 2"	["Address 2a, Address 2b"]	[200100, 200200]	"200100 | 20020...	["200100 | 200200 | 200300 | 200400 | 200500"]	"2a@email 2b@em...
3	180	2022-01-01 00:00:00	"Customer 4"	["Address 4a, Address 4b"]	[400100, 400200]	"400100 | 40020...	["400100 | 300200 | 400300 | 400400 | 400500"]	"4a@email 4b@em...
4	140	2022-01-01 00:00:00	"Customer 5"	["Address 5a, Address 5b"]	[500100, 500200]	"500100 | 50020...	["500100 | 500200 | 500300 | 500400 | 500500"]	"5a@email 5b@em...

In [ ]:
```
# Polars provide powerful List manipulations methods

# creating a list

out = customer.select(
    pl.col("ID"),
    pl.col("DESCRIPTION"),
    pl.col("PHONE1"),
    pl.col("PHONE1").cast(pl.List(pl.Utf8)).alias("Phone1_Int_to-Str"),
    pl.col("PHONE2"),
    pl.col("PHONE2").str.split(" | ").alias("PHONE2_new"),
    pl.col("PHONE3"),
    pl.col("PHONE3").explode().str.split(" | ").alias("new"),
```

```
    )
print(out)
```

```
shape: (15, 8)
┌─────┬─────────────┬─────────────┬────────────┬───────────┬────────────┬────────────┬──────────
───┬─────────────┐
│ ID  ┆ DESCRIPTION ┆ PHONE1      ┆ Phone1_Int ┆ PHONE2    ┆ PHONE2_new ┆ PHONE3
┆ new         │
│ --- ┆ ---         ┆ ---         ┆ _to-Str    ┆ ---       ┆ ---        ┆ ---
┆ ---         │
│ i64 ┆ str         ┆ list[i64]   ┆ ---        ┆ str       ┆ list[str]  ┆ list[str]
┆ list[str]   │
│     ┆             ┆             ┆ list[str]  ┆           ┆            ┆
┆             │
╞═════╪═════════════╪═════════════╪════════════╪═══════════╪════════════╪════════════╪══════════
══╪═════════════╡
│ 100 ┆ Customer 1  ┆ [100100,    ┆ ["100100", ┆ 100100 |  ┆ ["100100", ┆ ["100100
| ┆ ["100100", │
│     ┆             ┆ 100200]     ┆ "100200"]  ┆ 100200    ┆ "100200"]  ┆ 100200 |
┆ "100200",   │
│     ┆             ┆             ┆            ┆           ┆            ┆ 100300 |
┆ …           │
│     ┆             ┆             ┆            ┆           ┆            ┆ 100…
┆ "100500"]   │
│ 110 ┆ Customer 2  ┆ [200100,    ┆ ["200100", ┆ 200100 |  ┆ ["200100", ┆ ["200100
| ┆ ["200100", │
│     ┆             ┆ 200200]     ┆ "200200"]  ┆ 200200    ┆ "200200"]  ┆ 200200 |
┆ "200200",   │
│     ┆             ┆             ┆            ┆           ┆            ┆ 200300 |
┆ …           │
│     ┆             ┆             ┆            ┆           ┆            ┆ 200…
┆ "200500"]   │
│ 120 ┆ Customer 3  ┆ [300100,    ┆ ["300100", ┆ 300100 |  ┆ ["300100", ┆ ["300100
| ┆ ["300100", │
│     ┆             ┆ 300200]     ┆ "300200"]  ┆ 300200    ┆ "300200"]  ┆ 300200 |
┆ "300200",   │
│     ┆             ┆             ┆            ┆           ┆            ┆ 300300 |
┆ …           │
│     ┆             ┆             ┆            ┆           ┆            ┆ 300…
┆ "300500"]   │
│ 130 ┆ Customer 4  ┆ [400100,    ┆ ["400100", ┆ 400100 |  ┆ ["400100", ┆ ["400100
| ┆ ["400100", │
│     ┆             ┆ 400200]     ┆ "400200"]  ┆ 400200    ┆ "400200"]  ┆ 300200 |
┆ "300200",   │
│     ┆             ┆             ┆            ┆           ┆            ┆ 400300 |
┆ …           │
│     ┆             ┆             ┆            ┆           ┆            ┆ 400…
┆ "400500"]   │
│ …   ┆ …           ┆ …           ┆ …          ┆ …         ┆ …          ┆ …
┆ …           │
│ 210 ┆ Customer 2  ┆ [200100,    ┆ ["200100", ┆ 200100 |  ┆ ["200100", ┆ ["200100
| ┆ ["200100", │
│     ┆             ┆ 200200]     ┆ "200200"]  ┆ 200200    ┆ "200200"]  ┆ 200200 |
┆ "200200",   │
│     ┆             ┆             ┆            ┆           ┆            ┆ 200300 |
┆ …           │
│     ┆             ┆             ┆            ┆           ┆            ┆ 200…
┆ "200500"]   │
│ 220 ┆ Customer 3  ┆ [300100,    ┆ ["300100", ┆ 300100 |  ┆ ["300100", ┆ ["300100
```

```
│ ┆ ["300100", │
│     ┆              ┆ 300200]      ┆ "300200"] ┆ 300200      ┆ "300200"] ┆ 300200 │
┆ "300200",  │
│     ┆              ┆              ┆            ┆              ┆            ┆ 300300 │
┆ …          │
│     ┆              ┆              ┆            ┆              ┆            ┆ 300…
┆ "300500"]  │
│ 230 ┆ Customer 4  ┆ [400100,     ┆ ["400100", ┆ 400100 │     ┆ ["400100", ┆ ["400100
│ ┆ ["400100", │
│     ┆              ┆ 400200]      ┆ "400200"] ┆ 400200      ┆ "400200"] ┆ 300200 │
┆ "300200",  │
│     ┆              ┆              ┆            ┆              ┆            ┆ 400300 │
┆ …          │
│     ┆              ┆              ┆            ┆              ┆            ┆ 400…
┆ "400500"]  │
│ 240 ┆ Customer 5  ┆ [500100,     ┆ ["500100", ┆ 500100 │     ┆ ["500100", ┆ ["500100
│ ┆ ["500100", │
│     ┆              ┆ 500200]      ┆ "500200"] ┆ 500200      ┆ "500200"] ┆ 500200 │
┆ "500200",  │
│     ┆              ┆              ┆            ┆              ┆            ┆ 500300 │
┆ …          │
│     ┆              ┆              ┆            ┆              ┆            ┆ 500…
┆ "500500"]  │
└─────┴──────────────┴──────────────┴────────────┴──────────────┴────────────┴───────────
──┴──────────────┘
```

In [ ]:

```
# once you have a list to work with
# use explode() to break list content by rows
# say, list all phone #s by customers from PHONE1 column

out = customer.select(
    pl.col("ID"),
    pl.col("DESCRIPTION"),
    pl.col("PHONE1").cast(pl.List(pl.Utf8)).alias("Phone1_Int_to-Str"),
    ).explode("Phone1_Int_to-Str")
print(out)

# out = customer.select(
#     pl.col("ID"),
#     pl.col("DESCRIPTION"),
#     pl.col("PHONE3"),
#     pl.col("PHONE3").explode().str.split(" | ").alias("new"),
#     ).explode("new")
# print(out)
```

```
shape: (30, 3)
┌─────┬─────────────┬───────────────────┐
│ ID  ┆ DESCRIPTION ┆ Phone1_Int_to-Str │
│ --- ┆ ---         ┆ ---               │
│ i64 ┆ str         ┆ str               │
╞═════╪═════════════╪═══════════════════╡
│ 100 ┆ Customer 1  ┆ 100100            │
│ 100 ┆ Customer 1  ┆ 100200            │
│ 110 ┆ Customer 2  ┆ 200100            │
│ 110 ┆ Customer 2  ┆ 200200            │
│ …   ┆ …           ┆ …                 │
│ 230 ┆ Customer 4  ┆ 400100            │
│ 230 ┆ Customer 4  ┆ 400200            │
│ 240 ┆ Customer 5  ┆ 500100            │
│ 240 ┆ Customer 5  ┆ 500200            │
└─────┴─────────────┴───────────────────┘
```

In [ ]:

```
# operating on list columns
# using list methods inside columns to view slice of data
# for example , PHONE3 column has 5 values in list
# use head(), tail(), slice(), length()

out = customer.select(
    pl.col("ID"),
    pl.col("DESCRIPTION"),
    # pl.col("PHONE3"),
    pl.col("PHONE3").explode().str.split(" | ").alias("new"),
    ).with_columns(
    pl.col("new").list.head(3).alias("top3"),
    pl.col("new").list.slice(-3, 3).alias("bottom_3"),
    pl.col("new").list.lengths().alias("observations"),
)
print(out)
```

```
shape: (15, 6)
┌──────┬───────────────┬──────────────┬─────────────────────────┬─────────────────────────
┬───────────────┐
│ ID   ┆ DESCRIPTION ┆ new          ┆ top3                    ┆ bottom_3
┆ observations │
│ ---  ┆ ---         ┆ ---          ┆ ---                     ┆ ---
┆ ---          │
│ i64  ┆ str         ┆ list[str]    ┆ list[str]               ┆ list[str]
┆ u32          │
╞══════╪═══════════════╪══════════════╪═════════════════════════╪═════════════════════════
╪═══════════════╡
│ 100  ┆ Customer 1  ┆ ["100100",   ┆ ["100100", "100200",    ┆ ["100300", "100400",
┆ 5            │
│      ┆             ┆ "100200", …  ┆ "100300"]               ┆ "100500"]
┆              │
│      ┆             ┆ "100500"]    ┆                         ┆
┆              │
│ 110  ┆ Customer 2  ┆ ["200100",   ┆ ["200100", "200200",    ┆ ["200300", "200400",
┆ 5            │
│      ┆             ┆ "200200", …  ┆ "200300"]               ┆ "200500"]
┆              │
│      ┆             ┆ "200500"]    ┆                         ┆
┆              │
│ 120  ┆ Customer 3  ┆ ["300100",   ┆ ["300100", "300200",    ┆ ["300300", "300400",
┆ 5            │
│      ┆             ┆ "300200", …  ┆ "300300"]               ┆ "300500"]
┆              │
│      ┆             ┆ "300500"]    ┆                         ┆
┆              │
│ 130  ┆ Customer 4  ┆ ["400100",   ┆ ["400100", "300200",    ┆ ["400300", "400400",
┆ 5            │
│      ┆             ┆ "300200", …  ┆ "400300"]               ┆ "400500"]
┆              │
│      ┆             ┆ "400500"]    ┆                         ┆
┆              │
│ …    ┆ …           ┆ …            ┆ …                       ┆ …
┆ …            │
│ 210  ┆ Customer 2  ┆ ["200100",   ┆ ["200100", "200200",    ┆ ["200300", "200400",
┆ 5            │
│      ┆             ┆ "200200", …  ┆ "200300"]               ┆ "200500"]
┆              │
│      ┆             ┆ "200500"]    ┆                         ┆
┆              │
│ 220  ┆ Customer 3  ┆ ["300100",   ┆ ["300100", "300200",    ┆ ["300300", "300400",
┆ 5            │
│      ┆             ┆ "300200", …  ┆ "300300"]               ┆ "300500"]
┆              │
│      ┆             ┆ "300500"]    ┆                         ┆
┆              │
│ 230  ┆ Customer 4  ┆ ["400100",   ┆ ["400100", "300200",    ┆ ["400300", "400400",
┆ 5            │
│      ┆             ┆ "300200", …  ┆ "400300"]               ┆ "400500"]
┆              │
│      ┆             ┆ "400500"]    ┆                         ┆
┆              │
│ 240  ┆ Customer 5  ┆ ["500100",   ┆ ["500100", "500200",    ┆ ["500300", "500400",
```

```
┆ 5              │
│     ┆              ┆ "500200", … ┆ "500300"]              ┆ "500500"]
┆                │
│     ┆              ┆ "500500"]   ┆                        ┆
┆                │
└─────┴──────────────┴─────────────┴────────────────────────┴──────────────
└────────────────┘
```

```
In [ ]: # working with List elements
# let;s say, we want to list customers with
# number of invalid emailIDs

out = customer.select(
    pl.col("ID"),
    pl.col("DESCRIPTION"),
    pl.col("EMAIL"),
)
print(out)
# for example, Customer 5 has one email ID = donotreply

out = out.with_columns(
    pl.col("EMAIL").str.split(" ").list.eval(pl.element().str.contains("donotreply"
    .list.sum().alias("InvalidEMAILs")
)
print(out)
```

```
shape: (15, 3)
┌─────┬─────────────┬─────────────────────────────────┐
│ ID  ┆ DESCRIPTION ┆ EMAIL                           │
│ --- ┆ ---         ┆ ---                             │
│ i64 ┆ str         ┆ str                             │
╞═════╪═════════════╪═════════════════════════════════╡
│ 100 ┆ Customer 1  ┆ 1a@email 1b@email               │
│ 110 ┆ Customer 2  ┆ 2a@email 2b@email               │
│ 120 ┆ Customer 3  ┆ 3a@email 3b@email               │
│ 130 ┆ Customer 4  ┆ 4a@email 4b@email               │
│ …   ┆ …           ┆ …                               │
│ 210 ┆ Customer 2  ┆ 2a@email 2b@email               │
│ 220 ┆ Customer 3  ┆ 3a@email 3b@email               │
│ 230 ┆ Customer 4  ┆ 4a@email 4b@email               │
│ 240 ┆ Customer 5  ┆ 5a@email 5b@email donotreply@ema… │
└─────┴─────────────┴─────────────────────────────────┘
shape: (15, 4)
┌─────┬─────────────┬─────────────────────────────────┬───────────────┐
│ ID  ┆ DESCRIPTION ┆ EMAIL                           ┆ InvalidEMAILs │
│ --- ┆ ---         ┆ ---                             ┆ ---           │
│ i64 ┆ str         ┆ str                             ┆ u32           │
╞═════╪═════════════╪═════════════════════════════════╪═══════════════╡
│ 100 ┆ Customer 1  ┆ 1a@email 1b@email               ┆ 0             │
│ 110 ┆ Customer 2  ┆ 2a@email 2b@email               ┆ 0             │
│ 120 ┆ Customer 3  ┆ 3a@email 3b@email               ┆ 0             │
│ 130 ┆ Customer 4  ┆ 4a@email 4b@email               ┆ 0             │
│ …   ┆ …           ┆ …                               ┆ …             │
│ 210 ┆ Customer 2  ┆ 2a@email 2b@email               ┆ 0             │
│ 220 ┆ Customer 3  ┆ 3a@email 3b@email               ┆ 0             │
│ 230 ┆ Customer 4  ┆ 4a@email 4b@email               ┆ 0             │
│ 240 ┆ Customer 5  ┆ 5a@email 5b@email donotreply@ema… ┆ 1             │
└─────┴─────────────┴─────────────────────────────────┴───────────────┘
```

In [ ]:
```
# row wise computation #

# If you ask me, this is the most important feature of Polars
# I highly recommend practicing row wise computation using Polars
# as this is very beneficial and used extensively in data preparation
# for Machine Learning models training

##########################
# FROM POLARS user guide #
# We can apply any Polars operations on the elements of the list
# with the list.eval (list().eval in Rust) expression!
# These expressions run entirely on Polars' query engine and can run in parallel,
# so will be well optimized.
# Let's say we have another set of weather data across three days, for different st
##########################

# for example calculate RANK in below list of numbers
# Number    RANK
# 7         5
# 3.5       3   # when duplicate, next RANK will skip 1 by each duplicate
# 3.5       3
# 1         1
# 2         2
```

In [ ]:
```
import polars as pl
sampleDF = pl.DataFrame(
    {
        "ID": list(range(1,5)),
        "NUMBER": [[7, 3.5, 3.5, 1, 2],
                   [6, 2.5, 3.5, 1, 4],
                   [3, 3.5, 3.5, 5, 5],
                   [2, 3.5, 3.5, 4, 3]]
    }
)
print(sampleDF)

rank_pct = (pl.element().rank(descending=True) / pl.col("*").count()).round(2)

out = sampleDF.with_columns(
    # create the list of homogeneous data
    pl.concat_list(pl.all().exclude("ID")).alias("all_numbers")
).select(
    # select all columns except the intermediate list
    pl.all().exclude("all_numbers"),
    # compute the rank by calling `list.eval`
    pl.col("all_numbers").list.eval(rank_pct, parallel=True).alias("NUMBERS_RANK"),
)

print(out)
```

```
shape: (4, 2)
┌─────┬────────────────────┐
│ ID  ┆ NUMBER             │
│ --- ┆ ---                │
│ i64 ┆ list[f64]          │
╞═════╪════════════════════╡
│ 1   ┆ [7.0, 3.5, … 2.0]  │
│ 2   ┆ [6.0, 2.5, … 4.0]  │
│ 3   ┆ [3.0, 3.5, … 5.0]  │
│ 4   ┆ [2.0, 3.5, … 3.0]  │
└─────┴────────────────────┘
shape: (4, 3)
┌─────┬────────────────────┬────────────────────┐
│ ID  ┆ NUMBER             ┆ NUMBERS_RANK       │
│ --- ┆ ---                ┆ ---                │
│ i64 ┆ list[f64]          ┆ list[f64]          │
╞═════╪════════════════════╪════════════════════╡
│ 1   ┆ [7.0, 3.5, … 2.0]  ┆ [0.2, 0.5, … 0.8]  │
│ 2   ┆ [6.0, 2.5, … 4.0]  ┆ [0.2, 0.8, … 0.4]  │
│ 3   ┆ [3.0, 3.5, … 5.0]  ┆ [1.0, 0.7, … 0.3]  │
│ 4   ┆ [2.0, 3.5, … 3.0]  ┆ [1.0, 0.5, … 0.8]  │
└─────┴────────────────────┴────────────────────┘
```

In [ ]:

```python
# applying RANK | eval functions on BALANCE SHEET data
# for example

############################
## BalanceSheet DataFrame ##
############################
import random
from datetime import datetime

sampleSize = 100_000
org = "ABC Inc."
ledger_type = "ACTUALS" # BUDGET, STATS are other Ledger types
fiscal_year_from = 2020
fiscal_year_to = 2023
random.seed(123)

balanceSheet = pl.DataFrame({
        "LEDGER" : ledger_type,
        "ORG" : org,
        "FISCAL_YEAR": random.choices(list(range(fiscal_year_from,
                                 fiscal_year_to+1, 1)),k=sampleSize),
        "PERIOD": random.choices(list(range(1, 12+1, 1)),k=sampleSize),
        "ASSETS": random.sample(range(1000000), sampleSize),
    "LIABILITY": random.sample(range(1000000), sampleSize),
    "REVENUE": random.sample(range(1000000), sampleSize),
    "NETWORTH": random.sample(range(1000000), sampleSize),
})
print(balanceSheet.sample(5).with_row_count("Row #"))

rank_pct = (pl.element().rank(descending=True) / pl.col("*").count()).round(2)

out = balanceSheet.with_columns(
    # create the list of homogeneous data
```

```
    pl.concat_list(pl.all().exclude("LEDGER","ORG","FISCAL_YEAR","PERIOD")).alias("
).select(
    # select all columns except the intermediate list
    pl.all().exclude("all_CASHFLOW"),
    # compute the rank by calling `list.eval`
    pl.col("all_CASHFLOW").list.eval(rank_pct, parallel=True).alias("cf_RANK"),
)

print(out.sample(5))
```

```
shape: (5, 9)
┌───────┬─────────┬──────────┬─────────────┬───┬────────┬───────────┬─────────┬─────
─────┐
│ Row # ┆ LEDGER  ┆ ORG      ┆ FISCAL_YEAR ┆ … ┆ ASSETS ┆ LIABILITY ┆ REVENUE ┆ NETW
ORTH │
│ ---   ┆ ---     ┆ ---      ┆ ---         ┆   ┆ ---    ┆ ---       ┆ ---     ┆ ---
│
│ u32   ┆ str     ┆ str      ┆ i64         ┆   ┆ i64    ┆ i64       ┆ i64     ┆ i64
│
╞═══════╪═════════╪══════════╪═════════════╪═══╪════════╪═══════════╪═════════╪═════
═════╡
│ 0     ┆ ACTUALS ┆ ABC Inc. ┆ 2022        ┆ … ┆ 229177 ┆ 162079    ┆ 746304  ┆ 9345
36   │
│ 1     ┆ ACTUALS ┆ ABC Inc. ┆ 2022        ┆ … ┆ 58165  ┆ 645852    ┆ 795922  ┆ 3720
16   │
│ 2     ┆ ACTUALS ┆ ABC Inc. ┆ 2020        ┆ … ┆ 457983 ┆ 853620    ┆ 540738  ┆ 5135
16   │
│ 3     ┆ ACTUALS ┆ ABC Inc. ┆ 2021        ┆ … ┆ 949507 ┆ 907875    ┆ 28191   ┆ 7655
34   │
│ 4     ┆ ACTUALS ┆ ABC Inc. ┆ 2020        ┆ … ┆ 848681 ┆ 419144    ┆ 732089  ┆ 5115
68   │
└───────┴─────────┴──────────┴─────────────┴───┴────────┴───────────┴─────────┴─────
─────┘
shape: (5, 9)
┌─────────┬──────────┬─────────────┬────────┬───┬───────────┬─────────┬──────────┬───
──────────────┐
│ LEDGER  ┆ ORG      ┆ FISCAL_YEAR ┆ PERIOD ┆ … ┆ LIABILITY ┆ REVENUE ┆ NETWORTH ┆ c
f_RANK          │
│ ---     ┆ ---      ┆ ---         ┆ ---    ┆   ┆ ---       ┆ ---     ┆ ---      ┆ -
--              │
│ str     ┆ str      ┆ i64         ┆ i64    ┆   ┆ i64       ┆ i64     ┆ i64      ┆ l
ist[f64]        │
╞═════════╪══════════╪═════════════╪════════╪═══╪═══════════╪═════════╪══════════╪═══
═══════════════╡
│ ACTUALS ┆ ABC Inc. ┆ 2021        ┆ 10     ┆ … ┆ 118334    ┆ 110193  ┆ 182505   ┆
[0.25, 0.75, …   │
│         ┆          ┆             ┆        ┆   ┆           ┆         ┆          ┆
0.5]             │
│ ACTUALS ┆ ABC Inc. ┆ 2020        ┆ 4      ┆ … ┆ 100528    ┆ 316989  ┆ 431846   ┆
[0.25, 1.0, …    │
│         ┆          ┆             ┆        ┆   ┆           ┆         ┆          ┆
0.5]             │
│ ACTUALS ┆ ABC Inc. ┆ 2021        ┆ 12     ┆ … ┆ 251313    ┆ 8225    ┆ 386441   ┆
[0.25, 0.75, …   │
│         ┆          ┆             ┆        ┆   ┆           ┆         ┆          ┆
0.5]             │
│ ACTUALS ┆ ABC Inc. ┆ 2021        ┆ 2      ┆ … ┆ 436892    ┆ 38548   ┆ 684450   ┆
[0.75, 0.5, …    │
│         ┆          ┆             ┆        ┆   ┆           ┆         ┆          ┆
0.25]            │
│ ACTUALS ┆ ABC Inc. ┆ 2020        ┆ 5      ┆ … ┆ 671804    ┆ 643440  ┆ 473534   ┆
[1.0, 0.25, …    │
│         ┆          ┆             ┆        ┆   ┆           ┆         ┆          ┆
0.75]            │
└─────────┴──────────┴─────────────┴────────┴───┴───────────┴─────────┴──────────┴───
──────────────┘
```

In [ ]:
```
# Polars Array datatype
# Arrays are a new data type that was recently introduced,
# and are still pretty nascent in features that it offers.

# The major difference between a List and an Array is that the latter
# is limited to having the same number of elements per row,
# while a List can have a variable number of elements.

# Both still require that each element's data type is the same.

array_df = pl.DataFrame(
    [
        pl.Series("Array_1", [[1, 3], [2, 5]]),
        pl.Series("Array_2", [[1, 7, 3], [8, 1, 0]]),
    ],
    schema={"Array_1": pl.Array(2, pl.Int64), "Array_2": pl.Array(3, pl.Int64)},
)
print(array_df)

out = array_df.select(
    pl.col("Array_1").arr.min().suffix("_min"),
    pl.col("Array_2").arr.sum().suffix("_sum"),
)
print(out)
```

shape: (2, 2)

Array_1 --- array[i64, 2]	Array_2 --- array[i64, 3]
[1, 3]	[1, 7, 3]
[2, 5]	[8, 1, 0]

shape: (2, 2)

Array_1_min --- i64	Array_2_sum --- i64
1	11
2	9

using Expression to apply Structs

In [ ]:
```
# like Polars List and Arrays, which allows users to work with data in sets
# Polars provide one more datatype Struct
# Struct data type is different than List and Arrays
# as Struct allows users to work with multiple columns without copying actual data

# Struct
############################
## BalanceSheet DataFrame ##
############################
```

```python
import random
from datetime import datetime

sampleSize = 100_000
org = "ABC Inc."
ledger_type = "ACTUALS" # BUDGET, STATS are other Ledger types
fiscal_year_from = 2020
fiscal_year_to = 2023
random.seed(123)

balanceSheet = pl.DataFrame({
        "LEDGER" : ledger_type,
        "ORG" : org,
        "FISCAL_YEAR": random.choices(list(range(fiscal_year_from,
                                         fiscal_year_to+1, 1)),k=sampleSize),
        "PERIOD": random.choices(list(range(1, 12+1, 1)),k=sampleSize),
        "ASSETS": random.sample(range(1000000), sampleSize),
    "LIABILITY": random.sample(range(1000000), sampleSize),
    "REVENUE": random.sample(range(1000000), sampleSize),
    "NETWORTH": random.sample(range(1000000), sampleSize),
})
print(balanceSheet.sample(5).with_row_count("Row #"))

out = balanceSheet.select(pl.col("FISCAL_YEAR").value_counts(sort=True))
print(out)
```

```
shape: (5, 9)
┌────────┬─────────┬───────────┬─────────────┬───┬────────┬───────────┬──────────┬──────────┐
│ Row #  ┆ LEDGER  ┆ ORG       ┆ FISCAL_YEAR ┆ … ┆ ASSETS ┆ LIABILITY ┆ REVENUE  ┆ NETWORTH │
│ ---    ┆ ---     ┆ ---       ┆ ---         ┆   ┆ ---    ┆ ---       ┆ ---      ┆ ---      │
│ u32    ┆ str     ┆ str       ┆ i64         ┆   ┆ i64    ┆ i64       ┆ i64      ┆ i64      │
╞════════╪═════════╪═══════════╪═════════════╪═══╪════════╪═══════════╪══════════╪══════════╡
│ 0      ┆ ACTUALS ┆ ABC Inc.  ┆ 2022        ┆ … ┆ 229177 ┆ 162079    ┆ 746304   ┆ 934536   │
│ 1      ┆ ACTUALS ┆ ABC Inc.  ┆ 2022        ┆ … ┆ 58165  ┆ 645852    ┆ 795922   ┆ 372016   │
│ 2      ┆ ACTUALS ┆ ABC Inc.  ┆ 2020        ┆ … ┆ 457983 ┆ 853620    ┆ 540738   ┆ 513516   │
│ 3      ┆ ACTUALS ┆ ABC Inc.  ┆ 2021        ┆ … ┆ 949507 ┆ 907875    ┆ 28191    ┆ 765534   │
│ 4      ┆ ACTUALS ┆ ABC Inc.  ┆ 2020        ┆ … ┆ 848681 ┆ 419144    ┆ 732089   ┆ 511568   │
└────────┴─────────┴───────────┴─────────────┴───┴────────┴───────────┴──────────┴──────────┘
shape: (4, 1)
┌───────────────┐
│ FISCAL_YEAR   │
│ ---           │
│ struct[2]     │
╞═══════════════╡
│ {2023,25136}  │
│ {2021,25036}  │
│ {2020,24957}  │
│ {2022,24871}  │
└───────────────┘
```

In [ ]:
```python
# using unnest will convert Struct to regular data frame
out = balanceSheet.select(pl.col("FISCAL_YEAR").value_counts(sort=True)).unnest("FI
print(out)
```

```
shape: (4, 2)
┌─────────────┬────────┐
│ FISCAL_YEAR ┆ counts │
│ ---         ┆ ---    │
│ i64         ┆ u32    │
╞═════════════╪════════╡
│ 2023        ┆ 25136  │
│ 2021        ┆ 25036  │
│ 2020        ┆ 24957  │
│ 2022        ┆ 24871  │
└─────────────┴────────┘
```

In [ ]:
```python
# using Struct to identify duplicates
out = balanceSheet.filter(pl.struct("LEDGER","ORG","FISCAL_YEAR","PERIOD","ASSETS")
# is_duplicated() is same as is_unique()
print(out)
```

```
shape: (0, 8)
┌────────┬─────┬─────────────┬────────┬────────┬───────────┬─────────┬──────────┐
│ LEDGER ┆ ORG ┆ FISCAL_YEAR ┆ PERIOD ┆ ASSETS ┆ LIABILITY ┆ REVENUE ┆ NETWORTH │
│ ---    ┆ --- ┆ ---         ┆ ---    ┆ ---    ┆ ---       ┆ ---     ┆ ---      │
│ str    ┆ str ┆ i64         ┆ i64    ┆ i64    ┆ i64       ┆ i64     ┆ i64      │
╞════════╪═════╪═════════════╪════════╪════════╪═══════════╪═════════╪══════════╡
└────────┴─────┴─────────────┴────────┴────────┴───────────┴─────────┴──────────┘
```

In [ ]:

```python
# using Struct for multi column Ranking
# This is another very useful feature of Polars

# consider a situation, when you are working with TimeSeries data
# and want to fill in missing data based on certain RANK

out = balanceSheet.with_columns(
    pl.struct("FISCAL_YEAR", "PERIOD")
    .rank("dense", descending=True)
    .over("LEDGER", "ORG", "ASSETS")
    .alias("Rank")
)
# .filter(pl.struct("LEDGER", "ORG", "ASSETS").is_duplicated())
print(out)
```

```
shape: (100_000, 9)
┌─────────┬──────────┬─────────────┬────────┬───┬───────────┬─────────┬──────────┬──
────┐
│ LEDGER  ┆ ORG      ┆ FISCAL_YEAR ┆ PERIOD ┆ … ┆ LIABILITY ┆ REVENUE ┆ NETWORTH ┆ R
ank │
│ ---     ┆ ---      ┆ ---         ┆ ---    ┆   ┆ ---       ┆ ---     ┆ ---      ┆ -
--  │
│ str     ┆ str      ┆ i64         ┆ i64    ┆   ┆ i64       ┆ i64     ┆ i64      ┆ u
32  │
╞═════════╪══════════╪═════════════╪════════╪═══╪═══════════╪═════════╪══════════╪══
════╡
│ ACTUALS ┆ ABC Inc. ┆ 2020        ┆ 10     ┆ … ┆ 535987    ┆ 825234  ┆ 630949   ┆ 1
│
│ ACTUALS ┆ ABC Inc. ┆ 2020        ┆ 1      ┆ … ┆ 708385    ┆ 401531  ┆ 307809   ┆ 1
│
│ ACTUALS ┆ ABC Inc. ┆ 2021        ┆ 12     ┆ … ┆ 202378    ┆ 777927  ┆ 536595   ┆ 1
│
│ ACTUALS ┆ ABC Inc. ┆ 2020        ┆ 3      ┆ … ┆ 378510    ┆ 445913  ┆ 400960   ┆ 1
│
│ …       ┆ …        ┆ …           ┆ …      ┆ … ┆ …         ┆ …       ┆ …        ┆ …
│
│ ACTUALS ┆ ABC Inc. ┆ 2022        ┆ 6      ┆ … ┆ 274014    ┆ 812724  ┆ 541937   ┆ 1
│
│ ACTUALS ┆ ABC Inc. ┆ 2023        ┆ 8      ┆ … ┆ 804535    ┆ 188039  ┆ 876970   ┆ 1
│
│ ACTUALS ┆ ABC Inc. ┆ 2023        ┆ 8      ┆ … ┆ 752404    ┆ 610809  ┆ 715951   ┆ 1
│
│ ACTUALS ┆ ABC Inc. ┆ 2020        ┆ 2      ┆ … ┆ 489051    ┆ 783855  ┆ 226190   ┆ 1
│
└─────────┴──────────┴─────────────┴────────┴───┴───────────┴─────────┴──────────┴──
────┘
```

```
In [ ]: # use Struct in multi-column apply
        # we will cover this in USD (user defined functions) section below
```

Numpy conversion

```
In [ ]: # Polars expressions support NumPy ufuncs
        # This means that if a function is not provided by Polars
        # we can use NumPy and we still have fast columnar operation through the NumPy API.

        import polars as pl
        import numpy as np

        df = pl.DataFrame({"a": [1, 2, 3], "b": [4, 5, 6]})

        out = df.select(np.log(pl.all()).suffix("_log"))
        print(out)
```

```
shape: (3, 2)
┌──────────┬──────────┐
│ a_log    ┆ b_log    │
│ ---      ┆ ---      │
│ f64      ┆ f64      │
╞══════════╪══════════╡
│ 0.0      ┆ 1.386294 │
│ 0.693147 ┆ 1.609438 │
│ 1.098612 ┆ 1.791759 │
└──────────┴──────────┘
```

Functions, User defined functions and Windows function

Polars provide large number of built-in functions through Polars Expressions. Here are few examples using built-in expressions as functions.

```
In [ ]: # basic calculations using Polars expressions
        df = pl.DataFrame(
            {
                "a": [1, 2, 3],
                "b": [10, 20, 30],
            }
        )

        out = df.select(
            (pl.col("a") + pl.col("b")).alias("sum"),
            pl.col("a").mean().alias("mean"),
            pl.col("a").min().alias("min"),
            pl.col("a").max().alias("max")
            )
        out
```

shape: (3, 4)

sum	mean	min	max
i64	f64	i64	i64
11	2.0	1	3
22	2.0	1	3
33	2.0	1	3

In [ ]:
```
# Let's use Polars expression built-in functions
# to perform a conditional Algebric operation

# below sample DataFrame produces a Balance Sheet,
# however, NETWORTH calculated is wrong

############################
## BalanceSheet DataFrame ##
############################
import random
from datetime import datetime

sampleSize = 100_000
org = "ABC Inc."
ledger_type = "ACTUALS" # BUDGET, STATS are other Ledger types
fiscal_year_from = 2020
fiscal_year_to = 2023
random.seed(123)

balanceSheet = pl.DataFrame({
        "LEDGER" : ledger_type,
        "ORG" : org,
        "FISCAL_YEAR": random.choices(list(range(fiscal_year_from,
                                         fiscal_year_to+1, 1)),k=sampleSize),
        "PERIOD": random.choices(list(range(1, 12+1, 1)),k=sampleSize),
        "ASSETS": random.sample(range(1000000), sampleSize),
    "LIABILITY": random.sample(range(1000000), sampleSize),
    "REVENUE": random.sample(range(1000000), sampleSize),
    "NETWORTH": random.sample(range(1000000), sampleSize),
})
print(balanceSheet.sample(5).with_row_count("Row #"))
```

```
shape: (5, 9)
┌───────┬─────────┬──────────┬────────────┬───┬────────┬───────────┬─────────┬──────
──────┐
│ Row # ┆ LEDGER  ┆ ORG      ┆ FISCAL_YEAR ┆ … ┆ ASSETS ┆ LIABILITY ┆ REVENUE ┆ NETW
ORTH │
│ ---   ┆ ---     ┆ ---      ┆ ---        ┆   ┆ ---    ┆ ---       ┆ ---     ┆ ---
│
│ u32   ┆ str     ┆ str      ┆ i64        ┆   ┆ i64    ┆ i64       ┆ i64     ┆ i64
│
╞═══════╪═════════╪══════════╪════════════╪═══╪════════╪═══════════╪═════════╪══════
══════╡
│ 0     ┆ ACTUALS ┆ ABC Inc. ┆ 2022       ┆ … ┆ 229177 ┆ 162079    ┆ 746304  ┆ 9345
36    │
│ 1     ┆ ACTUALS ┆ ABC Inc. ┆ 2022       ┆ … ┆ 58165  ┆ 645852    ┆ 795922  ┆ 3720
16    │
│ 2     ┆ ACTUALS ┆ ABC Inc. ┆ 2020       ┆ … ┆ 457983 ┆ 853620    ┆ 540738  ┆ 5135
16    │
│ 3     ┆ ACTUALS ┆ ABC Inc. ┆ 2021       ┆ … ┆ 949507 ┆ 907875    ┆ 28191   ┆ 7655
34    │
│ 4     ┆ ACTUALS ┆ ABC Inc. ┆ 2020       ┆ … ┆ 848681 ┆ 419144    ┆ 732089  ┆ 5115
68    │
└───────┴─────────┴──────────┴────────────┴───┴────────┴───────────┴─────────┴──────
──────┘
```

Polars Functions

In [ ]:

```
# let's use Polars expression as functions to
# A. calculate NetWorth_c = ASSETS + REVENUE - LIABILITY
# B. generate a unique COSTCenter String = LEDGER+ORG+FISCAL_YEAR

out = balanceSheet.with_columns(
        (pl.col("ASSETS") + pl.col("REVENUE") -  pl.col("LIABILITY"))
            .alias("NetWorth_c"),
        (pl.col("LEDGER") + pl.col("ORG") + pl.col("FISCAL_YEAR").cast(pl.Utf8))
            .alias("CostCenter"),
    )
print(out.head(5))
```

```
shape: (5, 10)
┌─────────┬──────────┬─────────────┬────────┬───┬─────────┬──────────┬─────────────┬─
───────────┐
│ LEDGER  ┆ ORG      ┆ FISCAL_YEAR ┆ PERIOD ┆ … ┆ REVENUE ┆ NETWORTH ┆ NetWorth_c  ┆
CostCenter │
│ ---     ┆ ---      ┆ ---         ┆ ---    ┆   ┆ ---     ┆ ---      ┆ ---         ┆
---        │
│ str     ┆ str      ┆ i64         ┆ i64    ┆   ┆ i64     ┆ i64      ┆ i64         ┆
str        │
╞═════════╪══════════╪═════════════╪════════╪═══╪═════════╪══════════╪═════════════╪═
═══════════╡
│ ACTUALS ┆ ABC Inc. ┆ 2020        ┆ 10     ┆ … ┆ 825234  ┆ 630949   ┆ 358184      ┆
ACTUALSABC │
│         ┆          ┆             ┆        ┆   ┆         ┆          ┆             ┆
Inc.2020   │
│ ACTUALS ┆ ABC Inc. ┆ 2020        ┆ 1      ┆ … ┆ 401531  ┆ 307809   ┆ -289400     ┆
ACTUALSABC │
│         ┆          ┆             ┆        ┆   ┆         ┆          ┆             ┆
Inc.2020   │
│ ACTUALS ┆ ABC Inc. ┆ 2021        ┆ 12     ┆ … ┆ 777927  ┆ 536595   ┆ 597339      ┆
ACTUALSABC │
│         ┆          ┆             ┆        ┆   ┆         ┆          ┆             ┆
Inc.2021   │
│ ACTUALS ┆ ABC Inc. ┆ 2020        ┆ 3      ┆ … ┆ 445913  ┆ 400960   ┆ 398951      ┆
ACTUALSABC │
│         ┆          ┆             ┆        ┆   ┆         ┆          ┆             ┆
Inc.2020   │
│ ACTUALS ┆ ABC Inc. ┆ 2023        ┆ 9      ┆ … ┆ 339738  ┆ 116069   ┆ -72121      ┆
ACTUALSABC │
│         ┆          ┆             ┆        ┆   ┆         ┆          ┆             ┆
Inc.2023   │
└─────────┴──────────┴─────────────┴────────┴───┴─────────┴──────────┴─────────────┴─
───────────┘
```

user defined function

let's pretend, we have a function which is more complex in that case, a user defined function is required

for example, ORG incur interest on Liability, which is calculated as following and substracted from NetWorth

interest paid on Liability Accounts

```
math
    i = P (1 + r/n)^n*T
```

In [ ]:

```
# user defined function

# let's first setup OOPs for Liability class
import random
class Liability:

    def __init__(self, P, n, r, t):
```

```
        self.principalAmount = P
        # self.rate = r/100
        self.rate = r/100
        self.compound = n # in case of simple interest, n = 1
        self.time = t/12

    def getLoanInterest(self):
    # compound rate interest deposit
        # returns a tuple of interest and Total
        return round(self.principalAmount
                     * (1 + self.rate / self.compound)**(self.compound * self.time)
                     - self.principalAmount, ndigits=2), round(self.principalAmount

def getInterest(amount):
    compound=1.0
    rate=2.875
    t=60
    d = Liability(amount, compound, rate, t)
    return d.getLoanInterest()
```

In [ ]:

```
out = balanceSheet.with_columns(
    [
        pl.col("LIABILITY").map_elements(getInterest).alias("int_on_liab")
    ]
)
print(out)
```

```
shape: (100_000, 9)
┌──────────┬───────────┬─────────────┬────────┬─────┬───────────┬─────────┬──────────┬──
───────────┐
│ LEDGER   ┆ ORG       ┆ FISCAL_YEAR ┆ PERIOD ┆ …   ┆ LIABILITY ┆ REVENUE ┆ NETWORTH ┆ i
nt_on_liab │
│ ---      ┆ ---       ┆ ---         ┆ ---    ┆     ┆ ---       ┆ ---     ┆ ---      ┆ -
--         │
│ str      ┆ str       ┆ i64         ┆ i64    ┆     ┆ i64       ┆ i64     ┆ i64      ┆ l
ist[f64]   │
╞══════════╪═══════════╪═════════════╪════════╪═════╪═══════════╪═════════╪══════════╪══
═══════════╡
│ ACTUALS  ┆ ABC Inc.  ┆ 2020        ┆ 10     ┆ …   ┆ 535987    ┆ 825234  ┆ 630949   ┆
[81607.61,  │
│          ┆           ┆             ┆        ┆     ┆           ┆         ┆          ┆ 5
35987.0]   │
│ ACTUALS  ┆ ABC Inc.  ┆ 2020        ┆ 1      ┆ …   ┆ 708385    ┆ 401531  ┆ 307809   ┆
[107856.36, │
│          ┆           ┆             ┆        ┆     ┆           ┆         ┆          ┆ 7
08385.0]   │
│ ACTUALS  ┆ ABC Inc.  ┆ 2021        ┆ 12     ┆ …   ┆ 202378    ┆ 777927  ┆ 536595   ┆
[30813.41,  │
│          ┆           ┆             ┆        ┆     ┆           ┆         ┆          ┆ 2
02378.0]   │
│ ACTUALS  ┆ ABC Inc.  ┆ 2020        ┆ 3      ┆ …   ┆ 378510    ┆ 445913  ┆ 400960   ┆
[57630.68,  │
│          ┆           ┆             ┆        ┆     ┆           ┆         ┆          ┆ 3
78510.0]   │
│ …        ┆ …         ┆ …           ┆ …      ┆ …   ┆ …         ┆ …       ┆ …        ┆ …
│
│ ACTUALS  ┆ ABC Inc.  ┆ 2022        ┆ 6      ┆ …   ┆ 274014    ┆ 812724  ┆ 541937   ┆
[41720.47,  │
│          ┆           ┆             ┆        ┆     ┆           ┆         ┆          ┆ 2
74014.0]   │
│ ACTUALS  ┆ ABC Inc.  ┆ 2023        ┆ 8      ┆ …   ┆ 804535    ┆ 188039  ┆ 876970   ┆
[122495.84, │
│          ┆           ┆             ┆        ┆     ┆           ┆         ┆          ┆ 8
04535.0]   │
│ ACTUALS  ┆ ABC Inc.  ┆ 2023        ┆ 8      ┆ …   ┆ 752404    ┆ 610809  ┆ 715951   ┆
[114558.55, │
│          ┆           ┆             ┆        ┆     ┆           ┆         ┆          ┆ 7
52404.0]   │
│ ACTUALS  ┆ ABC Inc.  ┆ 2020        ┆ 2      ┆ …   ┆ 489051    ┆ 783855  ┆ 226190   ┆
[74461.29,  │
│          ┆           ┆             ┆        ┆     ┆           ┆         ┆          ┆ 4
89051.0]   │
└──────────┴───────────┴─────────────┴────────┴─────┴───────────┴─────────┴──────────┴──
───────────┘
```

In [ ]:

```python
## ideal way of doing it, use Polars built in Expressions
# compound=1.0
# rate=2.875
# t=60

out = balanceSheet.with_columns(
    # create the list of homogeneous data
    pl.concat_list(pl.all()
```

```
                    .exclude(["LEDGER","ORG","FISCAL_YEAR",
                              "PERIOD","ASSETS","REVENUE","NETWORTH"]))
                              .alias("liab")
).select([
    # select all columns except the intermediate list
    pl.all().exclude("liab"),
    # compute the rank by calling `arr.eval`
    pl.col("LIABILITY").map_elements(lambda x:
                                    round(x
                      * (1 + 2.875 / 1.0)**(1.0 * 60)
                      - x, ndigits=2)
                              )
    .alias("liability_new_Calc")
])
print(out)
```

```
shape: (100_000, 9)
┌─────────┬──────────┬─────────────┬────────┬───┬───────────┬─────────┬──────────┬──
────────────────┐
│ LEDGER  ┆ ORG      ┆ FISCAL_YEAR ┆ PERIOD ┆ … ┆ LIABILITY ┆ REVENUE ┆ NETWORTH ┆ l
iability_new_C │
│ ---     ┆ ---      ┆ ---         ┆ ---    ┆   ┆ ---       ┆ ---     ┆ ---      ┆ a
lc             │
│ str     ┆ str      ┆ i64         ┆ i64    ┆   ┆ i64       ┆ i64     ┆ i64      ┆ -
--             │
│         ┆          ┆             ┆        ┆   ┆           ┆         ┆          ┆ f
64             │
╞═════════╪══════════╪═════════════╪════════╪═══╪═══════════╪═════════╪══════════╪══
════════════════╡
│ ACTUALS ┆ ABC Inc. ┆ 2020        ┆ 10     ┆ … ┆ 535987    ┆ 825234  ┆ 630949   ┆
1.0604e41       │
│ ACTUALS ┆ ABC Inc. ┆ 2020        ┆ 1      ┆ … ┆ 708385    ┆ 401531  ┆ 307809   ┆
1.4014e41       │
│ ACTUALS ┆ ABC Inc. ┆ 2021        ┆ 12     ┆ … ┆ 202378    ┆ 777927  ┆ 536595   ┆
4.0037e40       │
│ ACTUALS ┆ ABC Inc. ┆ 2020        ┆ 3      ┆ … ┆ 378510    ┆ 445913  ┆ 400960   ┆
7.4882e40       │
│ …       ┆ …        ┆ …           ┆ …      ┆ … ┆ …         ┆ …       ┆ …        ┆ …
│
│ ACTUALS ┆ ABC Inc. ┆ 2022        ┆ 6      ┆ … ┆ 274014    ┆ 812724  ┆ 541937   ┆
5.4209e40       │
│ ACTUALS ┆ ABC Inc. ┆ 2023        ┆ 8      ┆ … ┆ 804535    ┆ 188039  ┆ 876970   ┆
1.5916e41       │
│ ACTUALS ┆ ABC Inc. ┆ 2023        ┆ 8      ┆ … ┆ 752404    ┆ 610809  ┆ 715951   ┆
1.4885e41       │
│ ACTUALS ┆ ABC Inc. ┆ 2020        ┆ 2      ┆ … ┆ 489051    ┆ 783855  ┆ 226190   ┆
9.6751e40       │
└─────────┴──────────┴─────────────┴────────┴───┴───────────┴─────────┴──────────┴──
────────────────┘
```

In [ ]:

```
# keep in mind,
# wherver possible, use Polars Expresssion than user defined function
# as Polars Expressions outperform USD or any other methods in speed
# above USD can we re-written as this Polars Expression

# using Polars custom function
```

```
# using map or apply
# be mindful, apply or map when call Python function will be slow
```

windows function

Window functions are expressions with superpowers. They allow you to perform aggregations on groups in the select context.

use multiple group by operations in parallel, using a single query

In [ ]:
```
print(balanceSheet.sample(5).with_row_count("Row #"))
```

```
shape: (5, 9)
┌───────┬─────────┬──────────┬─────────────┬───┬────────┬───────────┬─────────┬──────
───────┐
│ Row # ┆ LEDGER  ┆ ORG      ┆ FISCAL_YEAR ┆ … ┆ ASSETS ┆ LIABILITY ┆ REVENUE ┆ NETW
ORTH │
│ ---   ┆ ---     ┆ ---      ┆ ---         ┆   ┆ ---    ┆ ---       ┆ ---     ┆ ---
│
│ u32   ┆ str     ┆ str      ┆ i64         ┆   ┆ i64    ┆ i64       ┆ i64     ┆ i64
│
╞═══════╪═════════╪══════════╪═════════════╪═══╪════════╪═══════════╪═════════╪══════
══════╡
│ 0     ┆ ACTUALS ┆ ABC Inc. ┆ 2021        ┆ … ┆ 677730 ┆ 118334    ┆ 110193  ┆ 1825
05    │
│ 1     ┆ ACTUALS ┆ ABC Inc. ┆ 2020        ┆ … ┆ 981373 ┆ 100528    ┆ 316989  ┆ 4318
46    │
│ 2     ┆ ACTUALS ┆ ABC Inc. ┆ 2021        ┆ … ┆ 431052 ┆ 251313    ┆ 8225    ┆ 3864
41    │
│ 3     ┆ ACTUALS ┆ ABC Inc. ┆ 2021        ┆ … ┆ 259722 ┆ 436892    ┆ 38548   ┆ 6844
50    │
│ 4     ┆ ACTUALS ┆ ABC Inc. ┆ 2020        ┆ … ┆ 13388  ┆ 671804    ┆ 643440  ┆ 4735
34    │
└───────┴─────────┴──────────┴─────────────┴───┴────────┴───────────┴─────────┴──────
──────┘
```

In [ ]:
```
out = balanceSheet.select(
    "LEDGER",
    "FISCAL_YEAR",
    pl.col("NETWORTH").mean().over("FISCAL_YEAR").alias("networth_by_FY"),
    pl.col("REVENUE")
    .mean()
    .over(["FISCAL_YEAR", "PERIOD"])
    .alias("avg_rev_by_FYAP"),
    pl.col("ASSETS").mean().alias("avg_assets"),
)
print(out)
```

shape: (100_000, 5)

LEDGER --- str	FISCAL_YEAR --- i64	networth_by_FY --- f64	avg_rev_by_FYAP --- f64	avg_assets --- f64
ACTUALS	2020	498528.963417	500178.734118	500706.79386
ACTUALS	2020	498528.963417	504533.509133	500706.79386
ACTUALS	2021	499910.464251	502578.376791	500706.79386
ACTUALS	2020	498528.963417	500137.570938	500706.79386
…	…	…	…	…
ACTUALS	2022	501173.793897	504434.734793	500706.79386
ACTUALS	2023	497007.493555	495342.956584	500706.79386
ACTUALS	2023	497007.493555	495342.956584	500706.79386
ACTUALS	2020	498528.963417	509464.849498	500706.79386

In [ ]:
```
filtered = balanceSheet.filter(pl.col("LEDGER") == "ACTUALS").select(
    "FISCAL_YEAR",
    "PERIOD",
    "NETWORTH",
)
print(filtered)
```

shape: (100_000, 3)

FISCAL_YEAR --- i64	PERIOD --- i64	NETWORTH --- i64
2020	10	630949
2020	1	307809
2021	12	536595
2020	3	400960
…	…	…
2022	6	541937
2023	8	876970
2023	8	715951
2020	2	226190

In [ ]:
```
out = filtered.with_columns(
    pl.col(["FISCAL_YEAR","PERIOD","NETWORTH",]).sort_by("PERIOD", descending=False
)
print(out)
```

```
shape: (100_000, 3)
┌─────────────┬────────┬──────────┐
│ FISCAL_YEAR ┆ PERIOD ┆ NETWORTH │
│ ---         ┆ ---    ┆ ---      │
│ i64         ┆ i64    ┆ i64      │
╞═════════════╪════════╪══════════╡
│ 2020        ┆ 1      ┆ 307809   │
│ 2020        ┆ 1      ┆ 272713   │
│ 2021        ┆ 1      ┆ 617923   │
│ 2020        ┆ 1      ┆ 34682    │
│ …           ┆ …      ┆ …        │
│ 2022        ┆ 12     ┆ 225945   │
│ 2023        ┆ 12     ┆ 230376   │
│ 2023        ┆ 12     ┆ 586496   │
│ 2020        ┆ 12     ┆ 858653   │
└─────────────┴────────┴──────────┘
```

In [ ]:
```
# import note about user defined funtions

# Do not kill parallelization

# Python Users Only

# The following section is specific to Python, and doesn't apply to Rust.
# Within Rust, blocks and closures (lambdas) can, and will, be executed concurrentl

# We have all heard that Python is slow, and does "not scale."
# Besides the overhead of running "slow" bytecode,
# Python has to remain within the constraints of the Global Interpreter Lock (GIL).
# This means that if you were to use a lambda or a custom Python function to
# apply during a parallelized phase, Polars speed is capped running Python code
# preventing any multiple threads from executing the function.

# This all feels terribly limiting, especially because we often need
# those lambda functions in a .groupby() step, for example.
# This approach is still supported by Polars,
# but keeping in mind bytecode and the GIL costs have to be paid.
# It is recommended to try to solve your queries using the expression syntax
# before moving to lambdas.
#If you want to learn more about using lambdas, go to the user defined functions se

# Conclusion

# In the examples above we've seen that we can do a lot by combining expressions.
# By doing so we delay the use of custom Python functions
# that slow down the queries (by the slow nature of Python AND the GIL).

# If we are missing a type expression let us know by opening a feature request!
```

Polars Data Transformation

---

joins, concat, pivot and melts

joins

Polars Dataframe allows two data sets join by one or more columns. Polars supports the following join strategies by specifying the strategy argument:

- inner join: Returns row with matching keys in both frames. Non-matching rows in either the left or right frame are discarded.
- left join: Returns all rows in the left dataframe, whether or not a match in the right-frame is found. Non-matching rows have their right columns null-filled.
- outer join: Returns all rows from both the left and right dataframe. If no match is found in one frame, columns from the other frame are null-filled.
- cross join: Returns the Cartesian product of all rows from the left frame with all rows from the right frame. Duplicates rows are retained; the table length of A cross-joined with B is always len(A) × len(B).
- asof join A left-join in which the match is performed on the nearest key rather than on equal keys.
- semi join: Returns all rows from the left frame in which the join key is also present in the right frame.
- anti join: Returns all rows from the left frame in which the join key is not present in the right frame.

In [ ]:
```
########################
## ACCOUNTS DataFrame ##
########################
import random
from datetime import datetime

accounts = pl.DataFrame({
    "ACCOUNT":  list(range(10000, 45000, 1000)),
    "AS_OF_DATE" : datetime(2022, 1, 1),
    "DESCRIPTION" : ["Operating Expenses","Non Operating Expenses","Assets",
                     "Liabilities","Net worth accounts", "Statistical Accounts",
                     "Revenue"] * 5,
    "REGION": ["Region A","Region B","Region C","Region D", "Region E"] * 7,
    "TYPE" : ["E","E","A","L","N","S","R"] * 5,
    "STATUS" : "Active",
    "CLASSIFICATION" : ["OPERATING_EXPENSES","NON-OPERATING_EXPENSES",
                        "ASSETS","LIABILITIES","NET_WORTH","STATISTICS",
                        "REVENUE"] * 5,
    "CATEGORY" : [
                "Travel","Payroll","non-Payroll","Allowance","Cash",
                "Facility","Supply","Services","Investment","Misc.",
                "Depreciation","Gain","Service","Retired","Fault.",
                "Receipt","Accrual","Return","Credit","ROI",
                "Cash","Funds","Invest","Transfer","Roll-over",
                "FTE","Members","Non_Members","Temp","Contractors",
                "Sales","Merchant","Service","Consulting","Subscriptions"
        ],
```

```
})
accounts.sample(5).with_row_count("Row #")
```

Out[ ]: shape: (5, 9)

Row #	ACCOUNT	AS_OF_DATE	DESCRIPTION	REGION	TYPE	STATUS	CLASSIFICATION
u32	i64	datetime[μs]	str	str	str	str	str
0	30000	2022-01-01 00:00:00	"Revenue"	"Region A"	"R"	"Active"	"REVENUE"
1	37000	2022-01-01 00:00:00	"Revenue"	"Region C"	"R"	"Active"	"REVENUE"
2	43000	2022-01-01 00:00:00	"Statistical Ac…	"Region D"	"S"	"Active"	"STATISTICS"
3	13000	2022-01-01 00:00:00	"Liabilities"	"Region D"	"L"	"Active"	"LIABILITIES"
4	44000	2022-01-01 00:00:00	"Revenue"	"Region E"	"R"	"Active"	"REVENUE"

In [ ]:

```
##########################
## DEPARTMENT DataFrame ##
##########################
import random
from datetime import datetime

dept = pl.DataFrame({
    "DEPT":  list(range(1000, 2500, 100)),
    "AS_OF_DATE" : datetime(2022, 1, 1),
    "DESCRIPTION" : ["Sales & Marketing","Human Resource",
                     "Information Technology","Business leaders","other temp"] * 3,
    "REGION": ["Region A","Region B","Region C"] * 5,
    "STATUS" : "Active",
    "CLASSIFICATION" : ["SALES","HR", "IT","BUSINESS","OTHERS"] * 3,
    "TYPE" : ["S","H","I","B","O"] * 3,
    "CATEGORY" : ["sales","human_resource","IT_Staff","business","others"] * 3,
})
dept.sample(5).with_row_count("Row #")
```

Out[ ]: shape: (5, 9)

Row #	DEPT	AS_OF_DATE	DESCRIPTION	REGION	STATUS	CLASSIFICATION	TYPE	CATEG
u32	i64	datetime[μs]	str	str	str	str	str	
0	1000	2022-01-01 00:00:00	"Sales & Market…	"Region A"	"Active"	"SALES"	"S"	"s
1	2200	2022-01-01 00:00:00	"Information Te…	"Region A"	"Active"	"IT"	"I"	"IT_S
2	1500	2022-01-01 00:00:00	"Sales & Market…	"Region C"	"Active"	"SALES"	"S"	"s
3	1800	2022-01-01 00:00:00	"Business leade…	"Region C"	"Active"	"BUSINESS"	"B"	"busir
4	1400	2022-01-01 00:00:00	"other temp"	"Region B"	"Active"	"OTHERS"	"O"	"oth

In [ ]:

```
########################
## LOCATION DataFrame ##
########################
import random
from datetime import datetime

location = pl.DataFrame({
    "LOCATION":  list(range(11, 23)),
    "AS_OF_DATE" : datetime(2022, 1, 1),
    "DESCRIPTION" : ["Boston","New York","Philadelphia","Cleveland","Richmond",
                     "Atlanta","Chicago","St. Louis","Minneapolis","Kansas City",
                     "Dallas","San Francisco"],
    "REGION": ["Region A","Region B","Region C","Region D"] * 3,
    "TYPE" : "Physical",
    "CATEGORY" : ["Ship","Recv","Mfg"] * 4
})
location.sample(5).with_row_count("Row #")
```

Out[ ]: shape: (5, 7)

Row #	LOCATION	AS_OF_DATE	DESCRIPTION	REGION	TYPE	CATEGORY
u32	i64	datetime[μs]	str	str	str	str
0	16	2022-01-01 00:00:00	"Atlanta"	"Region B"	"Physical"	"Mfg"
1	22	2022-01-01 00:00:00	"San Francisco"	"Region D"	"Physical"	"Mfg"
2	13	2022-01-01 00:00:00	"Philadelphia"	"Region C"	"Physical"	"Mfg"
3	20	2022-01-01 00:00:00	"Kansas City"	"Region B"	"Physical"	"Ship"
4	15	2022-01-01 00:00:00	"Richmond"	"Region A"	"Physical"	"Recv"

In [ ]:

```
######################
## LEDGER DataFrame ##
######################
import random
from datetime import datetime

sampleSize = 100_000
org = "ABC Inc."
ledger_type = "ACTUALS" # BUDGET, STATS are other Ledger types
fiscal_year_from = 2020
fiscal_year_to = 2023
random.seed(123)

ledger = pl.DataFrame({
        "LEDGER" : ledger_type,
        "ORG" : org,
        "FISCAL_YEAR": random.choices(list(range(fiscal_year_from,
                                  fiscal_year_to+1, 1)),k=sampleSize),
        "PERIOD": random.choices(list(range(1, 12+1, 1)),k=sampleSize),
        "ACCOUNT" : random.choices(accounts["ACCOUNT"], k=sampleSize),
        "DEPT" : random.choices(dept["DEPT"], k=sampleSize),
        "LOCATION" : random.choices(location["LOCATION"], k=sampleSize),
        "POSTED_TOTAL": random.sample(range(1000000), sampleSize)
})
ledger.sample(5).with_row_count("Row #")
```

Out[ ]: shape: (5, 9)

Row #	LEDGER	ORG	FISCAL_YEAR	PERIOD	ACCOUNT	DEPT	LOCATION	POSTED_TOTA
u32	str	str	i64	i64	i64	i64	i64	i6
0	"ACTUALS"	"ABC Inc."	2021	1	26000	1700	14	63525
1	"ACTUALS"	"ABC Inc."	2020	8	31000	2300	21	50213
2	"ACTUALS"	"ABC Inc."	2020	9	31000	1300	22	72944
3	"ACTUALS"	"ABC Inc."	2020	6	15000	2400	22	76932
4	"ACTUALS"	"ABC Inc."	2021	8	31000	1700	17	4946

In [ ]:

```
## using joins

df_inner_ledger_join = ledger.join(accounts, on="ACCOUNT", how="inner")
print(df_inner_ledger_join)

# df_inner_ledger_join = ledger.join(dept, on="DEPT", how="inner")
# print(df_inner_ledger_join)

# df_inner_ledger_join = ledger.join(location, on="LOCATION", how="inner")
# print(df_inner_ledger_join)
```

```
shape: (100_000, 15)
┌──────────┬──────────┬─────────────┬─────────┬───┬──────┬────────┬──────────────────────
─┬───────────────┐
│ LEDGER   ┆ ORG      ┆ FISCAL_YEAR ┆ PERIOD  ┆ … ┆ TYPE ┆ STATUS ┆ CLASSIFICATION
┆ CATEGORY      │
│ ---      ┆ ---      ┆ ---         ┆ ---     ┆   ┆ ---  ┆ ---    ┆ ---
┆ ---           │
│ str      ┆ str      ┆ i64         ┆ i64     ┆   ┆ str  ┆ str    ┆ str
┆ str           │
╞══════════╪══════════╪═════════════╪═════════╪═══╪══════╪════════╪══════════════════════
═╪═══════════════╡
│ ACTUALS ┆ ABC Inc. ┆ 2020        ┆ 10      ┆ … ┆ A    ┆ Active ┆ ASSETS
┆ non-Payroll │
│ ACTUALS ┆ ABC Inc. ┆ 2020        ┆ 1       ┆ … ┆ E    ┆ Active ┆ OPERATING_EXPENSES
┆ Travel        │
│ ACTUALS ┆ ABC Inc. ┆ 2021        ┆ 12      ┆ … ┆ N    ┆ Active ┆ NET_WORTH
┆ Gain          │
│ ACTUALS ┆ ABC Inc. ┆ 2020        ┆ 3       ┆ … ┆ L    ┆ Active ┆ LIABILITIES
┆ Roll-over   │
│ …        ┆ …        ┆ …           ┆ …       ┆ … ┆ …    ┆ …      ┆ …
┆ …             │
│ ACTUALS ┆ ABC Inc. ┆ 2022        ┆ 6       ┆ … ┆ L    ┆ Active ┆ LIABILITIES
┆ Merchant     │
│ ACTUALS ┆ ABC Inc. ┆ 2023        ┆ 8       ┆ … ┆ A    ┆ Active ┆ ASSETS
┆ Transfer     │
│ ACTUALS ┆ ABC Inc. ┆ 2023        ┆ 8       ┆ … ┆ N    ┆ Active ┆ NET_WORTH
┆ Cash          │
│ ACTUALS ┆ ABC Inc. ┆ 2020        ┆ 2       ┆ … ┆ E    ┆ Active ┆ NON-OPERATING_EXPE
┆ Payroll       │
│          ┆          ┆             ┆         ┆   ┆      ┆        ┆ NSES
┆               │
└──────────┴──────────┴─────────────┴─────────┴───┴──────┴────────┴──────────────────────
─┴───────────────┘
```

concatenation

There are a number of ways to concatenate data from separate DataFrames:

- Vertical: two dataframes with the same columns can be vertically concatenated to make a longer dataframe
- Horizontal: two dataframes with the same number of rows and non-overlapping columns can be horizontally concatenated to make a wider dataframe
- Diagonal: two dataframes with different numbers of rows and columns can be diagonally concatenated to make a dataframe which might be longer and/ or wider. Where column names overlap values will be vertically concatenated. Where column names do not overlap new rows and columns will be added. Missing values will be set as null

In [ ]:
```
accounts.shape, dept.shape, location.shape, ledger.shape
```

Out[ ]:
```
((35, 8), (15, 8), (12, 6), (100000, 8))
```

In [ ]:
```
######################
## LEDGER DataFrame ##
######################
import random
from datetime import datetime

sampleSize = 100_000
org = "ABC Inc."
ledger_type = "ACTUALS" # BUDGET, STATS are other Ledger types
fiscal_year_from = 2020
fiscal_year_to = 2023
random.seed(123)

ledger = pl.DataFrame({
        "LEDGER" : ledger_type,
        "ORG" : org,
        "FISCAL_YEAR": random.choices(list(range(fiscal_year_from,
                                          fiscal_year_to+1, 1)),k=sampleSize),
        "PERIOD": random.choices(list(range(1, 12+1, 1)),k=sampleSize),
        "ACCOUNT" : random.choices(accounts["ID"], k=sampleSize),
        "DEPT" : random.choices(dept["ID"], k=sampleSize),
        "LOCATION" : random.choices(location["ID"], k=sampleSize),
        "POSTED_TOTAL": random.sample(range(1000000), sampleSize)
})
ledger.sample(5).with_row_count("Row #")
```

Out[ ]: shape: (5, 9)

Row #	LEDGER	ORG	FISCAL_YEAR	PERIOD	ACCOUNT	DEPT	LOCATION	POSTED_TOTA
u32	str	str	i64	i64	i64	i64	i64	i6
0	"ACTUALS"	"ABC Inc."	2021	1	26000	1700	14	63525
1	"ACTUALS"	"ABC Inc."	2020	8	31000	2300	21	50213
2	"ACTUALS"	"ABC Inc."	2020	9	31000	1300	22	72944
3	"ACTUALS"	"ABC Inc."	2020	6	15000	2400	22	76932
4	"ACTUALS"	"ABC Inc."	2021	8	31000	1700	17	4946

In [ ]:
```
ledger_budg = pl.DataFrame({
        "LEDGER" : ledger_type,
        "ORG" : org,
        "FISCAL_YEAR": random.choices(list(range(fiscal_year_from, fiscal_year_to+1
                                   ,k=sampleSize),
        "PERIOD": random.choices(list(range(1, 12+1, 1)),k=sampleSize),
        "ACCOUNT" : random.choices(accounts["ID"], k=sampleSize),
```

```
        "DEPT" : random.choices(dept["ID"], k=sampleSize),
        "LOCATION" : random.choices(location["ID"], k=sampleSize),
        "POSTED_TOTAL": random.sample(range(1000000), sampleSize)
})
ledger_budg.sample(5).with_row_count("Row #")
```

Out[ ]: shape: (5, 9)

Row #	LEDGER	ORG	FISCAL_YEAR	PERIOD	ACCOUNT	DEPT	LOCATION	POSTED_TOTA
u32	str	str	i64	i64	i64	i64	i64	i6
0	"ACTUALS"	"ABC Inc."	2021	11	30000	1100	11	7497
1	"ACTUALS"	"ABC Inc."	2021	9	42000	2300	22	11584
2	"ACTUALS"	"ABC Inc."	2022	10	17000	1300	16	15653
3	"ACTUALS"	"ABC Inc."	2021	8	20000	1100	14	21206
4	"ACTUALS"	"ABC Inc."	2021	5	38000	1600	21	20405

In [ ]:

```
#########################################
# combined ledger for Actuals and Budget
#########################################
dfLedger = pl.concat([ledger, ledger_budg], how="vertical")
dfLedger.sample(5).with_row_count("Row #")
```

In [ ]:

```
dfLedger.shape
```

Out[ ]: (200000, 8)

In [ ]:

```
# Horizontal concatenation fails when dataframes have overlapping columns or a diff
# make sure, number of rows are same and column names are not same

# dfLedger = pl.concat([ledger, ledger_budg], how="horizontal")
# dfLedger.sample(5).with_row_count("Row #")
```

pivot

Pivot a column in a DataFrame and perform one of the following aggregations:

- first
- sum
- min
- max
- mean

- median

Eager Pivot

The pivot operation consists of a group by one, or multiple columns (these will be the new y-axis), the column that will be pivoted (this will be the new x-axis) and an aggregation.

Lazy Pivot

A Polars LazyFrame always need to know the schema of a computation statically (before collecting the query). As a pivot's output schema depends on the data, and it is therefore impossible to determine the schema without running the query.

Polars could have abstracted this fact for you just like Spark does, but we don't want you to shoot yourself in the foot with a shotgun. The cost should be clear upfront.

In [ ]:
```
df = pl.DataFrame(
    {
        "foo": ["A", "A", "B", "B", "C"],
        "N": [1, 2, 2, 4, 2],
        "bar": ["k", "l", "m", "n", "o"],
    }
)
print(df)

out = df.pivot(index="foo", columns="bar", values="N", aggregate_function="sum")
print(out)
```

shape: (5, 3)

foo --- str	N --- i64	bar --- str
A	1	k
A	2	l
B	2	m
B	4	n
C	2	o

shape: (3, 6)

foo --- str	k --- i64	l --- i64	m --- i64	n --- i64	o --- i64
A	1	2	null	null	null
B	null	null	2	4	null
C	null	null	null	null	2

In [ ]:
```
############################
## BalanceSheet DataFrame ##
############################
```

```
import random
from datetime import datetime

sampleSize = 100_000
org = "ABC Inc."
ledger_type = "ACTUALS" # BUDGET, STATS are other Ledger types
fiscal_year_from = 2020
fiscal_year_to = 2023
random.seed(123)

balanceSheet = pl.DataFrame({
        "LEDGER" : ledger_type,
        "ORG" : org,
        "FISCAL_YEAR": random.choices(list(range(fiscal_year_from,
                                         fiscal_year_to+1, 1)),k=sampleSize),
        "PERIOD": random.choices(list(range(1, 12+1, 1)),k=sampleSize),
        "ASSETS": random.sample(range(1000000), sampleSize),
    "LIABILITY": random.sample(range(1000000), sampleSize),
    "REVENUE": random.sample(range(1000000), sampleSize),
    "NETWORTH": random.sample(range(1000000), sampleSize),
})
print(balanceSheet.sample(5).with_row_count("Row #"))
```

```
shape: (5, 9)
┌───────┬─────────┬──────────┬─────────────┬───┬────────┬───────────┬─────────┬──────
──────┐
│ Row # ┆ LEDGER  ┆ ORG      ┆ FISCAL_YEAR ┆ … ┆ ASSETS ┆ LIABILITY ┆ REVENUE ┆ NETW
ORTH │
│ ---   ┆ ---     ┆ ---      ┆ ---         ┆   ┆ ---    ┆ ---       ┆ ---     ┆ ---
│
│ u32   ┆ str     ┆ str      ┆ i64         ┆   ┆ i64    ┆ i64       ┆ i64     ┆ i64
│
╞═══════╪═════════╪══════════╪═════════════╪═══╪════════╪═══════════╪═════════╪══════
═════╡
│ 0     ┆ ACTUALS ┆ ABC Inc. ┆ 2022        ┆ … ┆ 229177 ┆ 162079    ┆ 746304  ┆ 9345
36    │
│ 1     ┆ ACTUALS ┆ ABC Inc. ┆ 2022        ┆ … ┆ 58165  ┆ 645852    ┆ 795922  ┆ 3720
16    │
│ 2     ┆ ACTUALS ┆ ABC Inc. ┆ 2020        ┆ … ┆ 457983 ┆ 853620    ┆ 540738  ┆ 5135
16    │
│ 3     ┆ ACTUALS ┆ ABC Inc. ┆ 2021        ┆ … ┆ 949507 ┆ 907875    ┆ 28191   ┆ 7655
34    │
│ 4     ┆ ACTUALS ┆ ABC Inc. ┆ 2020        ┆ … ┆ 848681 ┆ 419144    ┆ 732089  ┆ 5115
68    │
└───────┴─────────┴──────────┴─────────────┴───┴────────┴───────────┴─────────┴──────
──────┘
```

In [ ]:

```
out = balanceSheet.pivot(index="FISCAL_YEAR", columns="PERIOD", values="NETWORTH",
print(out)
```

shape: (4, 13)

FISCAL_YEAR	10	1	12	…	4	8	2	7
---	---	---	---		---	---	---	---
i64	f64	f64	f64		f64	f64	f64	f64
2020	519604.0	512152.0	491098.5	…	487557.0	494666.0	491307.0	494794.0
2021	514325.0	493920.5	502130.5	…	483349.0	493417.0	503360.5	514359.0
2023	490111.5	500121.0	497847.5	…	484256.0	506810.0	486628.5	487821.0
2022	490166.0	503240.0	506036.0	…	509159.0	487072.5	495781.5	507723.5

In [ ]:

```python
# lazy execution
q = (balanceSheet.lazy()
     .collect()
     .pivot(index="FISCAL_YEAR", columns="PERIOD", values="NETWORTH", aggregate_fun
     .lazy()
)

out = q.collect()
print(out)
```

```
shape: (4, 13)
┌───────────┬───────────┬───────────┬───────────┬───┬───────────┬───────────┬──────────
────┬───────────┐
│ FISCAL_YE ┆ 10        ┆ 1         ┆ 12        ┆ … ┆ 4         ┆ 8         ┆ 2
┆ 7         │
│ AR        ┆ ---       ┆ ---       ┆ ---       ┆   ┆ ---       ┆ ---       ┆ ---
┆ ---       │
│ ---       ┆ i64       ┆ i64       ┆ i64       ┆   ┆ i64       ┆ i64       ┆ i64
┆ i64       │
│ i64       ┆           ┆           ┆           ┆   ┆           ┆           ┆
┆           │
╞═══════════╪═══════════╪═══════════╪═══════════╪═══╪═══════════╪═══════════╪══════════
════╪═══════════╡
│ 2020      ┆ 108224941 ┆ 108038900 ┆ 104273367 ┆ … ┆ 972685995 ┆ 101209961 ┆ 103032
396 ┆ 10381162 │
│           ┆ 5         ┆ 1         ┆ 8         ┆   ┆           ┆ 2         ┆ 6
┆ 31        │
│ 2021      ┆ 108108676 ┆ 104743057 ┆ 104492372 ┆ … ┆ 100276120 ┆ 101698492 ┆ 104962
643 ┆ 10383994 │
│           ┆ 5         ┆ 7         ┆ 9         ┆   ┆ 2         ┆ 2         ┆ 3
┆ 32        │
│ 2023      ┆ 102746210 ┆ 105297297 ┆ 103340523 ┆ … ┆ 103721463 ┆ 106394484 ┆ 104096
539 ┆ 10014053 │
│           ┆ 5         ┆ 9         ┆ 5         ┆   ┆ 5         ┆ 2         ┆ 8
┆ 61        │
│ 2022      ┆ 102879002 ┆ 103223842 ┆ 100630852 ┆ … ┆ 105154873 ┆ 105099774 ┆ 105418
639 ┆ 10246101 │
│           ┆ 9         ┆ 6         ┆ 5         ┆   ┆ 5         ┆ 6         ┆ 1
┆ 72        │
└───────────┴───────────┴───────────┴───────────┴───┴───────────┴───────────┴──────────
────┴───────────┘
```

melts

Melt operations unpivot a DataFrame from wide format to long format

In [ ]:
```
df = pl.DataFrame(
    {
        "A": ["a", "b", "a"],
        "B": [1, 3, 5],
        "C": [10, 11, 12],
        "D": [2, 4, 6],
    }
)
print(df)

out = df.melt(id_vars=["A", "B"], value_vars=["C", "D"])
print(out)
```

```
shape: (3, 4)
┌─────┬─────┬─────┬─────┐
│ A   ┆ B   ┆ C   ┆ D   │
│ --- ┆ --- ┆ --- ┆ --- │
│ str ┆ i64 ┆ i64 ┆ i64 │
╞═════╪═════╪═════╪═════╡
│ a   ┆ 1   ┆ 10  ┆ 2   │
│ b   ┆ 3   ┆ 11  ┆ 4   │
│ a   ┆ 5   ┆ 12  ┆ 6   │
└─────┴─────┴─────┴─────┘
shape: (6, 4)
┌─────┬─────┬──────────┬───────┐
│ A   ┆ B   ┆ variable ┆ value │
│ --- ┆ --- ┆ ---      ┆ ---   │
│ str ┆ i64 ┆ str      ┆ i64   │
╞═════╪═════╪══════════╪═══════╡
│ a   ┆ 1   ┆ C        ┆ 10    │
│ b   ┆ 3   ┆ C        ┆ 11    │
│ a   ┆ 5   ┆ C        ┆ 12    │
│ a   ┆ 1   ┆ D        ┆ 2     │
│ b   ┆ 3   ┆ D        ┆ 4     │
│ a   ┆ 5   ┆ D        ┆ 6     │
└─────┴─────┴──────────┴───────┘
```

In [ ]:
```
out = balanceSheet.melt(id_vars=["FISCAL_YEAR", "PERIOD"], value_vars=["NETWORTH"])
print(out)
```

```
shape: (100_000, 4)
┌─────────────┬────────┬──────────┬────────┐
│ FISCAL_YEAR ┆ PERIOD ┆ variable ┆ value  │
│ ---         ┆ ---    ┆ ---      ┆ ---    │
│ i64         ┆ i64    ┆ str      ┆ i64    │
╞═════════════╪════════╪══════════╪════════╡
│ 2020        ┆ 10     ┆ NETWORTH ┆ 630949 │
│ 2020        ┆ 1      ┆ NETWORTH ┆ 307809 │
│ 2021        ┆ 12     ┆ NETWORTH ┆ 536595 │
│ 2020        ┆ 3      ┆ NETWORTH ┆ 400960 │
│ …           ┆ …      ┆ …        ┆ …      │
│ 2022        ┆ 6      ┆ NETWORTH ┆ 541937 │
│ 2023        ┆ 8      ┆ NETWORTH ┆ 876970 │
│ 2023        ┆ 8      ┆ NETWORTH ┆ 715951 │
│ 2020        ┆ 2      ┆ NETWORTH ┆ 226190 │
└─────────────┴────────┴──────────┴────────┘
```

Polars SQL

how to use Lazy API

- the lazy API allows Polars to apply automatic query optimization with the query optimizer
- the lazy API allows you to work with larger than memory datasets using streaming

- the lazy API can catch schema errors before processing the data

In the ideal case we use the lazy API right from a file as the query optimizer may help us to reduce the amount of data we read from the file.

- scan_csv or scan_parquet or scan_xxx

```
import polars as pl

from ..paths import DATA_DIR

q1 = (
    pl.scan_csv(f"{DATA_DIR}/reddit.csv")
    .with_columns(pl.col("name").str.to_uppercase())
    .filter(pl.col("comment_karma") > 0)
)
```

If we were to run the code above on the Reddit CSV the query would not be evaluated. Instead Polars takes each line of code, adds it to the internal query graph and optimizes the query graph.

```
import polars as pl

from ..paths import DATA_DIR

q4 = (
    pl.scan_csv(f"{DATA_DIR}/reddit.csv")
    .with_columns(pl.col("name").str.to_uppercase())
    .filter(pl.col("comment_karma") > 0)
    .collect()
)
```

Execution on larger-than-memory (RAM) data analytics

If your data requires more memory than you have available Polars may be able to process the data in batches using streaming mode. To use streaming mode you simply pass the streaming=True argument to collect

```
import polars as pl

from ..paths import DATA_DIR

q5 = (
    pl.scan_csv(f"{DATA_DIR}/reddit.csv")
    .with_columns(pl.col("name").str.to_uppercase())
    .filter(pl.col("comment_karma") > 0)
    .collect(streaming=True)
)
```

Execution on a partial dataset

While you're writing, optimizing or checking your query on a large dataset, querying all available data may lead to a slow development process.

You can instead execute the query with the .fetch method. The .fetch method takes a parameter n_rows and tries to 'fetch' that number of rows at the data source. The number of rows cannot be guaranteed, however, as the lazy API does not count how many rows there are at each stage of the query.

Here we "fetch" 100 rows from the source file and apply the predicates.

```
import polars as pl

from ..paths import DATA_DIR

q9 = (
    pl.scan_csv(f"{DATA_DIR}/reddit.csv")
    .with_columns(pl.col("name").str.to_uppercase())
    .filter(pl.col("comment_karma") > 0)
    .fetch(n_rows=int(100))
)
```

- TODO: cover streaming topic
- TODO: cover sinking to a a file
- TODO: all topics from Lazy API Chapter

In [ ]:
```
#### Query Optimization
import polars as pl
q3 = pl.DataFrame({"foo": ["a", "b", "c"], "bar": [0, 1, 2]}).lazy()

print(q3.schema)

q3.describe_optimized_plan()

## query example to show schema
lazy_eager_query = (
    pl.DataFrame(
        {
            "id": ["a", "b", "c"],
            "month": ["jan", "feb", "mar"],
            "values": [0, 1, 2],
        }
    )
    .lazy()
    .with_columns((2 * pl.col("values")).alias("double_values"))
    .collect()
    .pivot(
        index="id", columns="month", values="double_values", aggregate_function="fi
    )
    .lazy()
```

```
    .filter(pl.col("mar").is_null())
    .collect()
)
print(lazy_eager_query)
q3.show_graph(optimized=False)
q3.explain(optimized=False)
```