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

---

Author: Amit Shukla

https://github.com/AmitXShukla

https://twitter.com/ashuklax

https://youtube.com/AmitXShukla

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 polars
```

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.

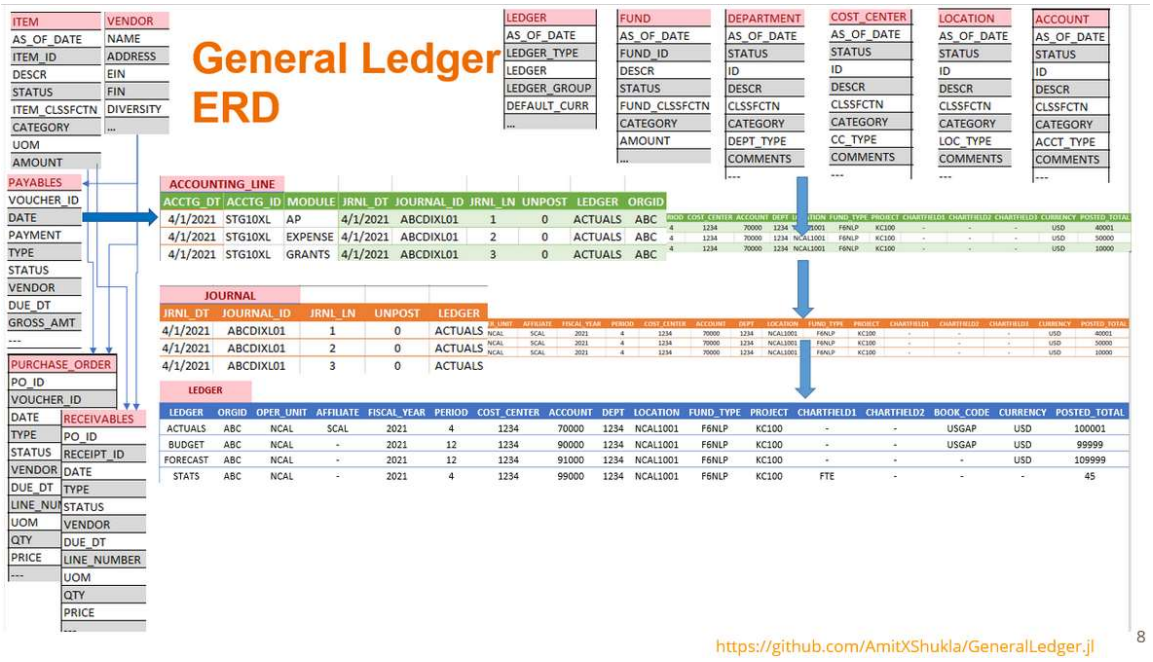
As stated above, since our objective is learn Data Science operations on Finance and Supply chain dataset, we will focus on creating few real life examples which are similar to Finance and Supply chain.

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.

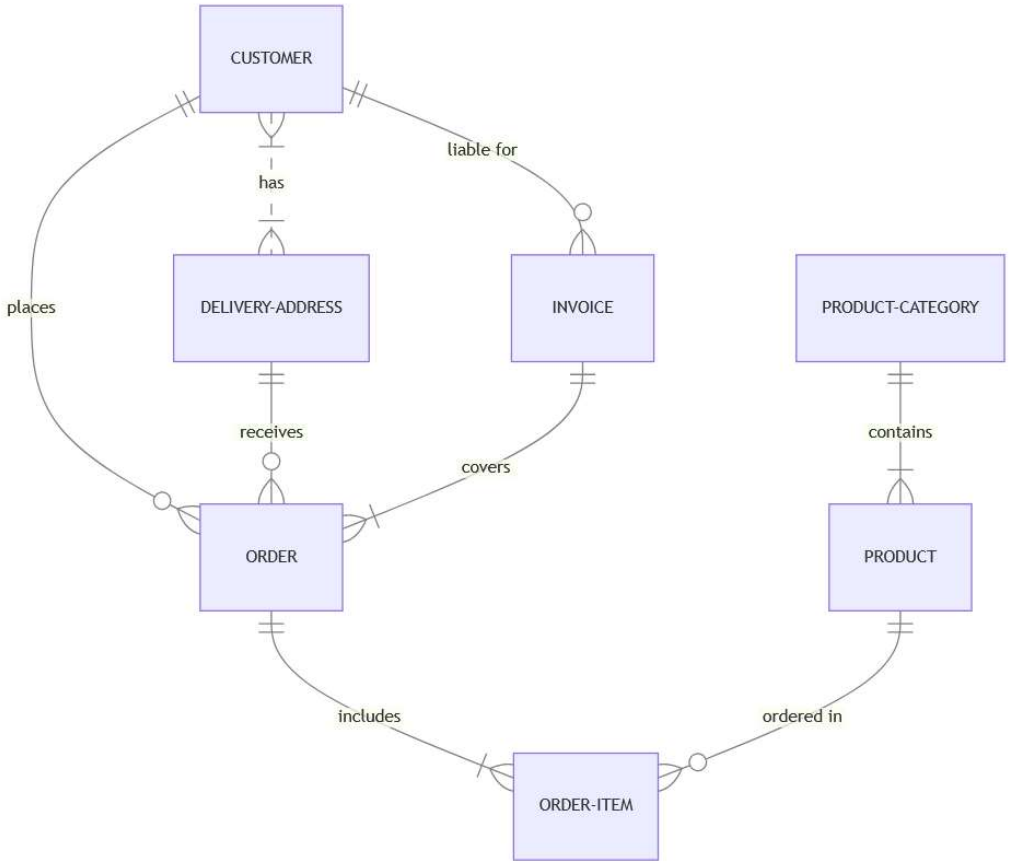
The product entity accounts for the diverse range of items being handled, from raw materials to finished goods.

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
```



Creating Polars DataFrame

Polars Data Structure

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

Finance DataSet

In [ ]:
```
#########################
## 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

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	16	2022-01-01 00:00:00	"Atlanta"	"Region B"	"Physical"	"Mfg"
1	18	2022-01-01 00:00:00	"St. Louis"	"Region D"	"Physical"	"Recv"
2	13	2022-01-01 00:00:00	"Philadelphia"	"Region C"	"Physical"	"Mfg"
3	21	2022-01-01 00:00:00	"Dallas"	"Region C"	"Physical"	"Recv"
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	CATE
u32	i64	datetime[μs]	str	str	str	str	str	
0	34000	2022-01-01 00:00:00	"Liabilities"	"Region E"	"L"	"Active"	"LIABILITIES"	"Roll
1	43000	2022-01-01 00:00:00	"Statistical Ac…	"Region D"	"S"	"Active"	"STATISTICS"	"Consu
2	23000	2022-01-01 00:00:00	"Revenue"	"Region D"	"R"	"Active"	"REVENUE"	"Re
3	26000	2022-01-01 00:00:00	"Assets"	"Region B"	"A"	"Active"	"ASSETS"	"Ac
4	19000	2022-01-01 00:00:00	"Assets"	"Region E"	"A"	"Active"	"ASSETS"	"

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	2000	2022-01-01 00:00:00	"Sales & Market…	"Region B"	"Active"	"SALES"	"S"	
1	1100	2022-01-01 00:00:00	"Human Resource…	"Region B"	"Active"	"HR"	"H"	"human
2	1700	2022-01-01 00:00:00	"Information Te…	"Region B"	"Active"	"IT"	"I"	
3	1500	2022-01-01 00:00:00	"Sales & Market…	"Region C"	"Active"	"SALES"	"S"	
4	1400	2022-01-01 00:00:00	"other temp"	"Region B"	"Active"	"OTHERS"	"O"	

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_TOTAL
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 ##
################################
```

```
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 [ ]:

```
#######################
## 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	170	2022-01-01 00:00:00	"Item 3"	"Active"	1500
1	190	2022-01-01 00:00:00	"Item 5"	"Active"	1200
2	160	2022-01-01 00:00:00	"Item 2"	"Active"	1600
3	200	2022-01-01 00:00:00	"Item 1"	"Active"	1200
4	140	2022-01-01 00:00:00	"Item 5"	"Active"	1800

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	
u32	i64	datetime[μs]	str	str	str	str	str	
0	220	2022-01-01 00:00:00	"Customer 3"	"Address 3"	"0000000003"	"3@email"	"Active"	"
1	110	2022-01-01 00:00:00	"Customer 2"	"Address 2"	"0000000002"	"2@email"	"Active"	
2	190	2022-01-01 00:00:00	"Customer 5"	"Address 5"	"0000000005"	"5@email"	"Active"	"
3	130	2022-01-01 00:00:00	"Customer 4"	"Address 4"	"0000000004"	"4@email"	"Active"	"
4	150	2022-01-01 00:00:00	"Customer 1"	"Address 1"	"0000000001"	"1@email"	"Active"	"Indiv

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	1050	2024-01-01 00:00:00	240	200	297931	537712
1	1010	2024-01-01 00:00:00	120	240	6697	575249
2	1080	2024-01-01 00:00:00	160	120	95966	920321
3	1070	2024-01-01 00:00:00	140	180	488746	504977
4	1060	2024-01-01 00:00:00	100	150	191620	714538

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	1080	"paid"
2	2024-01-01 00:00:00	1020	"shipped"
3	2024-01-01 00:00:00	1050	"shipped"
4	2024-01-01 00:00:00	1040	"cancelled"

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_TOT
AL │
│ ---     ┆ ---      ┆ ---         ┆ ---    ┆ ---     ┆ ---  ┆ ---      ┆ ---
│
│ str     ┆ str      ┆ i64         ┆ i64    ┆ i64     ┆ i64  ┆ i64      ┆ i64
│
╞═════════╪══════════╪═════════════╪════════╪═════════╪══════╪══════════╪══════════
═╡
│ BUDGET  ┆ ABC Inc. ┆ 2020        ┆ 8      ┆ 44000   ┆ 1500 ┆ 16       ┆ 785941
│
│ BUDGET  ┆ ABC Inc. ┆ 2023        ┆ 5      ┆ 10000   ┆ 1800 ┆ 17       ┆ 147279
│
│ BUDGET  ┆ ABC Inc. ┆ 2021        ┆ 11     ┆ 14000   ┆ 1700 ┆ 14       ┆ 455867
│
│ ACTUALS ┆ ABC Inc. ┆ 2021        ┆ 1      ┆ 31000   ┆ 1200 ┆ 20       ┆ 504394
│
│ ACTUALS ┆ ABC Inc. ┆ 2020        ┆ 5      ┆ 11000   ┆ 1100 ┆ 12       ┆ 827392
│
└─────────┴──────────┴─────────────┴────────┴─────────┴──────┴──────────┴──────────
┘
shape: (200_000, 9)
┌────────┬─────────────┬────────┬──────────┬───┬─────────┬──────┬──────────┬─────────
┐
│ Row #  ┆ FISCAL_YEAR ┆ PERIOD ┆ LEDGER   ┆ … ┆ ACCOUNT ┆ DEPT ┆ LOCATION ┆ in Thous
ands │
│ ---    ┆ ---         ┆ ---    ┆ ---      ┆   ┆ ---     ┆ ---  ┆ ---      ┆ ---
│
│ 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 ┆ in Thousands │
│ ---    ┆ ---     ┆ ---      ┆ ---         ┆   ┆ ---  ┆ ---      ┆ ---          ┆ ---          │
│ u32    ┆ str     ┆ str      ┆ i64         ┆   ┆ i64  ┆ i64      ┆ i64          ┆ f64          │
╞════════╪═════════╪══════════╪═════════════╪═══╪══════╪══════════╪══════════════╪══════════════╡
│ 0      ┆ ACTUALS ┆ ABC Inc. ┆ 2020        ┆ … ┆ 2400 ┆ 22       ┆ 753956       ┆ 753.956      │
│ 1      ┆ ACTUALS ┆ ABC Inc. ┆ 2020        ┆ … ┆ 1900 ┆ 14       ┆ 826906       ┆ 826.906      │
│ 2      ┆ ACTUALS ┆ ABC Inc. ┆ 2021        ┆ … ┆ 1700 ┆ 17       ┆ 454574       ┆ 454.574      │
│ 3      ┆ ACTUALS ┆ ABC Inc. ┆ 2020        ┆ … ┆ 1300 ┆ 12       ┆ 334989       ┆ 334.989      │
│ …      ┆ …       ┆ …        ┆ …           ┆ … ┆ …    ┆ …        ┆ …            ┆ …            │
│ 199996 ┆ BUDGET  ┆ ABC Inc. ┆ 2022        ┆ … ┆ 2200 ┆ 18       ┆ 65437        ┆ 65.437       │
│ 199997 ┆ BUDGET  ┆ ABC Inc. ┆ 2020        ┆ … ┆ 1300 ┆ 13       ┆ 960254       ┆ 960.254      │
│ 199998 ┆ BUDGET  ┆ ABC Inc. ┆ 2022        ┆ … ┆ 2000 ┆ 20       ┆ 41157        ┆ 41.157       │
│ 199999 ┆ BUDGET  ┆ ABC Inc. ┆ 2020        ┆ … ┆ 1900 ┆ 21       ┆ 265241       ┆ 265.241      │
└────────┴─────────┴──────────┴─────────────┴───┴──────┴──────────┴──────────────┴──────────────┘
```

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))
        ).groupby("LEDGER").agg(
                                pl.count()
                                ).with_row_count("Row #")
print(out)

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

# sort group by data by FISCAL_YEAR
out = dfLedger.groupby("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 #	LEDGER	count
u32	str	u32
0	ACTUALS	25136

shape: (8, 4)

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

shape: (8, 6)

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

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	1070	2024-01-01 00:00:00	140	180	488746	504977
1	1060	2024-01-01 00:00:00	100	150	191620	714538
2	1020	2024-01-01 00:00:00	240	190	231863	758142
3	1080	2024-01-01 00:00:00	160	120	95966	920321
4	1040	2024-01-01 00:00:00	120	190	355876	663663

```
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	160	230	97022	862.904
1	1010	2024-01-01 00:00:00	120	240	6697	575.249
2	1020	2024-01-01 00:00:00	240	190	231863	758.142
3	1030	2024-01-01 00:00:00	120	160	749266	386.456
…	…	…	…	…	…	…
6	1060	2024-01-01 00:00:00	100	150	191620	714.538
7	1070	2024-01-01 00:00:00	140	180	488746	504.977
8	1080	2024-01-01 00:00:00	160	120	95966	920.321
9	1090	2024-01-01 00:00:00	110	100	888748	611.663

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
0	1000	2024-01-01 00:00:00	160	230	97022	862.904
9	1090	2024-01-01 00:00:00	110	100	888748	611.663
2	1020	2024-01-01 00:00:00	240	190	231863	758.142
3	1030	2024-01-01 00:00:00	120	160	749266	386.456
5	1050	2024-01-01 00:00:00	240	200	297931	537.712

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	230	862.904
6	1060	2024-01-01 00:00:00	150	714.538
9	1090	2024-01-01 00:00:00	100	611.663
7	1070	2024-01-01 00:00:00	180	504.977
4	1040	2024-01-01 00:00:00	190	663.663

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

Out[ ]: shape: (5, 3)

Row #	ID	QTY
u32	i64	i64
9	1090	888748
1	1010	6697
6	1060	191620
3	1030	749266
4	1040	355876

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	97022
"2024-Jan-01"	7	1070	488746
"2024-Jan-01"	6	1060	191620
"2024-Jan-01"	3	1030	749266
"2024-Jan-01"	4	1040	355876

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
1090	888748	611.663
1010	6697	575.249
1080	95966	920.321
1030	749266	386.456
1060	191620	714.538

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 thousands  │
│ ---   ┆ ---  ┆ ---                 ┆ ---      ┆ ---  ┆ ---  ┆ ---                  │
│ u32   ┆ i64  ┆ datetime[μs]        ┆ i64      ┆ i64  ┆ i64  ┆ f64                  │
╞═══════╪══════╪═════════════════════╪══════════╪══════╪══════╪══════════════════════╡
│ 7     ┆ 1070 ┆ 2024-01-01 00:00:00 ┆ 140      ┆ 180  ┆ 488746 ┆ 504.977            │
└───────┴──────┴─────────────────────┴──────────┴──────┴──────┴──────────────────────┘
```

Out[ ]: shape: (5, 5)

Row #	ID	CUSTOMER	ITEM	QTY
u32	i64	i64	i64	i64
9	1090	110	100	888748
1	1010	120	240	6697
2	1020	240	190	231863
3	1030	120	160	749266
4	1040	120	190	355876

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 thousands │
│ ---  ┆ ---  ┆ ---                 ┆ ---      ┆ ---  ┆ ---    ┆ ---                 │
│ u32  ┆ i64  ┆ datetime[μs]        ┆ i64      ┆ i64  ┆ i64    ┆ f64                 │
╞══════╪══════╪═════════════════════╪══════════╪══════╪════════╪═════════════════════╡
│ 3    ┆ 1030 ┆ 2024-01-01 00:00:00 ┆ 120      ┆ 160  ┆ 749266 ┆ 386.456             │
└──────┴──────┴─────────────────────┴──────────┴──────┴────────┴─────────────────────┘
```

Out[ ]: shape: (5, 5)

Row #	ID	CUSTOMER	ITEM	QTY
u32	i64	i64	i64	i64
8	1080	160	120	95966
9	1090	110	100	888748
7	1070	140	180	488746
3	1030	120	160	749266
4	1040	120	190	355876

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.groupby("CUSTOMER").agg(pl.count()))

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

```
shape: (6, 2)
```

CUSTOMER	count
i64	u32
160	2
140	1
100	1
110	1
120	3
240	2

```
6
```

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
160	0
120	0
240	0
120	0
…	…
100	Preferred
140	0
160	0
110	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(selector_column_names(dfOrder, out))
```

```
True
('AS_OF_DATE',)
```

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

# 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
│
╞════════╪══════╪═════════════════════╪══════════╪══════╪════════╪══════════════════════════╡
│ 6     ┆ 1060 ┆ 2024-01-01 00:00:00 ┆ 100      ┆ 150  ┆ 191620 ┆ 714.538
│
│ 5     ┆ 1050 ┆ 2024-01-01 00:00:00 ┆ 240      ┆ 200  ┆ 297931 ┆ 537.712
│
│ 9     ┆ 1090 ┆ 2024-01-01 00:00:00 ┆ 110      ┆ 100  ┆ 888748 ┆ 611.663
│
│ 3     ┆ 1030 ┆ 2024-01-01 00:00:00 ┆ 120      ┆ 160  ┆ 749266 ┆ 386.456
│
│ 8     ┆ 1080 ┆ 2024-01-01 00:00:00 ┆ 160      ┆ 120  ┆ 95966  ┆ 920.321
│
└────────┴──────┴─────────────────────┴──────────┴──────┴────────┴──────────────────────────┘
```

```
shape: (10, 2)
┌──────────────────┬──────────────────┐
│ integers_as_floats ┆ floats_as_integers │
│ ---              ┆ ---              │
│ f32              ┆ f32              │
╞══════════════════╪══════════════════╡
│ 1000.0           ┆ 97022.0          │
│ 1010.0           ┆ 6697.0           │
│ 1020.0           ┆ 231863.0         │
│ 1030.0           ┆ 749266.0         │
│ …                ┆ …                │
│ 1060.0           ┆ 191620.0         │
│ 1070.0           ┆ 488746.0         │
│ 1080.0           ┆ 95966.0          │
│ 1090.0           ┆ 888748.0         │
└──────────────────┴──────────────────┘
```

In [ ]:
```
# 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	160	862.903992
1010	120	575.249023
1020	240	758.142029
1030	120	386.455994
…	…	…
1060	100	714.538025
1070	140	504.97699
1080	160	920.320984
1090	110	611.663025

In [ ]:
```python
# 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 value(s) [386.456, 504.977, … 920.32
1]; if you were trying to cast Utf8 to temporal dtypes, consider using `strptime`
shape: (10, 1)
```

Amount in thousands
i8
null
null
null
null
…
null
null
null
null

In [ ]:
```python
# 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 ┆ 160      ┆ 230  ┆ 97022  ┆ 862.904             ┆ 100
0 │
│ 1     ┆ 2024-01-01 00:00:00 ┆ 120      ┆ 240  ┆ 6697   ┆ 575.249             ┆ 101
0 │
│ 2     ┆ 2024-01-01 00:00:00 ┆ 240      ┆ 190  ┆ 231863 ┆ 758.142             ┆ 102
0 │
│ 3     ┆ 2024-01-01 00:00:00 ┆ 120      ┆ 160  ┆ 749266 ┆ 386.456             ┆ 103
0 │
│ …     ┆ …                   ┆ …        ┆ …    ┆ …      ┆ …                   ┆ …
  │
│ 6     ┆ 2024-01-01 00:00:00 ┆ 100      ┆ 150  ┆ 191620 ┆ 714.538             ┆ 106
0 │
│ 7     ┆ 2024-01-01 00:00:00 ┆ 140      ┆ 180  ┆ 488746 ┆ 504.977             ┆ 107
0 │
│ 8     ┆ 2024-01-01 00:00:00 ┆ 160      ┆ 120  ┆ 95966  ┆ 920.321             ┆ 108
0 │
│ 9     ┆ 2024-01-01 00:00:00 ┆ 110      ┆ 100  ┆ 888748 ┆ 611.663             ┆ 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 ┆ 160      ┆ 230  ┆ 97022  ┆ 862.904             ┆ 100
0 │
│ 1     ┆ 2024-01-01 00:00:00 ┆ 120      ┆ 240  ┆ 6697   ┆ 575.249             ┆ 101
0 │
│ 2     ┆ 2024-01-01 00:00:00 ┆ 240      ┆ 190  ┆ 231863 ┆ 758.142             ┆ 102
0 │
│ 3     ┆ 2024-01-01 00:00:00 ┆ 120      ┆ 160  ┆ 749266 ┆ 386.456             ┆ 103
0 │
│ …     ┆ …                   ┆ …        ┆ …    ┆ …      ┆ …                   ┆ …
  │
│ 6     ┆ 2024-01-01 00:00:00 ┆ 100      ┆ 150  ┆ 191620 ┆ 714.538             ┆ 106
0 │
│ 7     ┆ 2024-01-01 00:00:00 ┆ 140      ┆ 180  ┆ 488746 ┆ 504.977             ┆ 107
0 │
```

```
| 8   | 2024-01-01 00:00:00 | 160 | 120 | 95966  | 920.321 | 108
0 |
| 9   | 2024-01-01 00:00:00 | 110 | 100 | 888748 | 611.663 | 109
0 |
└─────┴─────────────────────┴─────┴─────┴────────┴─────────┴────
──┘
```

In [ ]:
```
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 value(s) ["not_a_number"]; if you w
ere trying to cast Utf8 to temporal dtypes, consider using `strptime`
```

In [ ]:
```
# 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 thousand
s │
│ ---   ┆ ---  ┆ ---                 ┆ ---      ┆ ---  ┆ ---    ┆ ---
│
│ u32   ┆ i64  ┆ datetime[μs]        ┆ i64      ┆ i64  ┆ i64    ┆ f64
│
╞════════╪══════╪═════════════════════╪══════════╪══════╪════════╪══════════════════════
═╡
│ 9     ┆ 1090 ┆ 2024-01-01 00:00:00 ┆ 110      ┆ 100  ┆ 888748 ┆ 611.663
│
│ 1     ┆ 1010 ┆ 2024-01-01 00:00:00 ┆ 120      ┆ 240  ┆ 6697   ┆ 575.249
│
│ 2     ┆ 1020 ┆ 2024-01-01 00:00:00 ┆ 240      ┆ 190  ┆ 231863 ┆ 758.142
│
│ 8     ┆ 1080 ┆ 2024-01-01 00:00:00 ┆ 160      ┆ 120  ┆ 95966  ┆ 920.321
│
└────────┴──────┴─────────────────────┴──────────┴──────┴────────┴──────────────────────
─┘
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 ┆ 160      ┆ 230  ┆ 862.904
│
│ true  ┆ true ┆ 1010 ┆ 2024-01-01 00:00:00 ┆ 120      ┆ 240  ┆ 575.249
│
│ true  ┆ true ┆ 1020 ┆ 2024-01-01 00:00:00 ┆ 240      ┆ 190  ┆ 758.142
│
│ true  ┆ true ┆ 1030 ┆ 2024-01-01 00:00:00 ┆ 120      ┆ 160  ┆ 386.456
│
│ …     ┆ …    ┆ …    ┆ …                   ┆ …        ┆ …    ┆ …
│
│ true  ┆ true ┆ 1060 ┆ 2024-01-01 00:00:00 ┆ 100      ┆ 150  ┆ 714.538
│
│ true  ┆ true ┆ 1070 ┆ 2024-01-01 00:00:00 ┆ 140      ┆ 180  ┆ 504.977
│
│ true  ┆ true ┆ 1080 ┆ 2024-01-01 00:00:00 ┆ 160      ┆ 120  ┆ 920.321
│
│ true  ┆ true ┆ 1090 ┆ 2024-01-01 00:00:00 ┆ 110      ┆ 100  ┆ 611.663
│
└────────┴──────┴──────┴─────────────────────┴──────────┴──────┴──────────────────────
┘
```

In [ ]:
```python
# 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	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

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 [ ]:
```python
# 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      │
╞═════╪══════════════╪═════════════╪═════════════╪═══╪═════════╪════════╪══════════════╪══════════╡
│ 170 ┆ 2022-01-01   ┆ Customer 3  ┆ Address 3   ┆ … ┆ 3@email ┆ Active ┆ Gov          ┆ 1200     │
│     ┆ 00:00:00     ┆             ┆             ┆   ┆         ┆        ┆              ┆          │
│ 160 ┆ 2022-01-01   ┆ Customer 2  ┆ Address 2   ┆ … ┆ 2@email ┆ Active ┆ Corp         ┆ 1500     │
│     ┆ 00:00:00     ┆             ┆             ┆   ┆         ┆        ┆              ┆          │
│ 240 ┆ 2022-01-01   ┆ Customer 5  ┆ Address 5   ┆ … ┆ 5@email ┆ Active ┆ Individual   ┆ 1500     │
│     ┆ 00:00:00     ┆             ┆             ┆   ┆         ┆        ┆              ┆          │
│ 230 ┆ 2022-01-01   ┆ Customer 4  ┆ Address 4   ┆ … ┆ 4@email ┆ Active ┆ Gov          ┆ 1600     │
│     ┆ 00:00:00     ┆             ┆             ┆   ┆         ┆        ┆              ┆          │
│ 140 ┆ 2022-01-01   ┆ Customer 5  ┆ Address 5   ┆ … ┆ 5@email ┆ Active ┆ Gov          ┆ 1900     │
│     ┆ 00:00:00     ┆             ┆             ┆   ┆         ┆        ┆              ┆          │
└─────┴──────────────┴─────────────┴─────────────┴───┴─────────┴────────┴──────────────┴──────────┘
shape: (15, 11)
┌─────┬──────────────┬─────────────┬─────────────┬───┬──────────────┬──────────┬──────────────┬──────────┐
│ ID  ┆ AS_OF_DATE   ┆ DESCRIPTIO  ┆ ADDRESS     ┆ … ┆ TYPE         ┆ CATEGORY ┆ DESCRIPTIO   ┆ ADDRESS_le │
│ --- ┆ ---          ┆ N           ┆ ---         ┆   ┆ ---          ┆ ---      ┆ N_byte_coun  ┆ tter_count │
│ i64 ┆ datetime[µs  ┆ ---         ┆ str         ┆   ┆ str          ┆ i64      ┆ t            ┆ ---        │
│     ┆ ]            ┆ str         ┆             ┆   ┆              ┆          ┆ ---          ┆ u32        │
│     ┆              ┆             ┆             ┆   ┆              ┆          ┆ u32          ┆            │
╞═════╪══════════════╪═════════════╪═════════════╪═══╪══════════════╪══════════╪══════════════╪════════════╡
│ 100 ┆ 2022-01-01   ┆ Customer 1  ┆ Address 1   ┆ … ┆ Corp         ┆ 1800     ┆ 10           ┆ 9          │
│     ┆ 00:00:00     ┆             ┆             ┆   ┆              ┆          ┆              ┆            │
│ 110 ┆ 2022-01-01   ┆ Customer 2  ┆ Address 2   ┆ … ┆ Gov          ┆ 1900     ┆ 10           ┆ 9          │
│     ┆ 00:00:00     ┆             ┆             ┆   ┆              ┆          ┆              ┆            │
```

```
| 120 | 2022-01-01 | Customer 3 | Address 3 | … | Individual | 1400 | 10
| 9          |
|     | 00:00:00   |            |           |   |            |      |
|            |
| 130 | 2022-01-01 | Customer 4 | Address 4 | … | Corp       | 1900 | 10
| 9          |
|     | 00:00:00   |            |           |   |            |      |
|            |
| …   | …          | …          | …         | … | …          | …    | …
| …          |
| 210 | 2022-01-01 | Customer 2 | Address 2 | … | Individual | 1800 | 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 | 1500 | 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	regex	literal	starts_with	ends_with
str	bool	bool	bool	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 [ ]:
```
# 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_repla ┆ text_repla │
│ --- ┆ ---         ┆ N           ┆ ---         ┆   ┆ ---         ┆ ---         ┆ ce_all      ┆ ce_all      │
│ i64 ┆ datetime[μs ┆ ---         ┆ str         ┆   ┆ str         ┆ i64         ┆ ---         ┆ ---         │
│     ┆ ]           ┆ str         ┆             ┆   ┆             ┆             ┆ str         ┆ str         │
╞═════╪═════════════╪═════════════╪═════════════╪═══╪═════════════╪═════════════╪═════════════╪═════════════╡
│ 100 ┆ 2022-01-01  ┆ Customer 1 ┆ Address 1   ┆ … ┆ Corp        ┆ 1900        ┆ ADDRESS 1   ┆ AddRESS 1   │
│     ┆ 00:00:00    ┆             ┆             ┆   ┆             ┆             ┆             ┆             │
│ 110 ┆ 2022-01-01  ┆ Customer 2 ┆ Address 2   ┆ … ┆ Gov         ┆ 1300        ┆ ADDRESS 2   ┆ AddRESS 2   │
│     ┆ 00:00:00    ┆             ┆             ┆   ┆             ┆             ┆             ┆             │
│ 120 ┆ 2022-01-01  ┆ Customer 3 ┆ Address 3   ┆ … ┆ Individual  ┆ 1300        ┆ ADDRESS 3   ┆ AddRESS 3   │
│     ┆ 00:00:00    ┆             ┆             ┆   ┆             ┆             ┆             ┆             │
│ 130 ┆ 2022-01-01  ┆ Customer 4 ┆ Address 4   ┆ … ┆ Corp        ┆ 1600        ┆ ADDRESS 4   ┆ AddRESS 4   │
│     ┆ 00:00:00    ┆             ┆             ┆   ┆             ┆             ┆             ┆             │
│ …   ┆ …           ┆ …           ┆ …           ┆ … ┆ …           ┆ …           ┆ …           ┆ …           │
│ 210 ┆ 2022-01-01  ┆ Customer 2 ┆ Address 2   ┆ … ┆ Individual  ┆ 2000        ┆ ADDRESS 2   ┆ AddRESS 2   │
│     ┆ 00:00:00    ┆             ┆             ┆   ┆             ┆             ┆             ┆             │
│ 220 ┆ 2022-01-01  ┆ Customer 3 ┆ Address 3   ┆ … ┆ Corp        ┆ 1200        ┆ ADDRESS 3   ┆ AddRESS 3   │
│     ┆ 00:00:00    ┆             ┆             ┆   ┆             ┆             ┆             ┆             │
│ 230 ┆ 2022-01-01  ┆ Customer 4 ┆ Address 4   ┆ … ┆ Gov         ┆ 2200        ┆ ADDRESS 4   ┆ AddRESS 4   │
│     ┆ 00:00:00    ┆             ┆             ┆   ┆             ┆             ┆             ┆             │
│ 240 ┆ 2022-01-01  ┆ Customer 5 ┆ Address 5   ┆ … ┆ Individual  ┆ 1600        ┆ ADDRESS 5   ┆ AddRESS 5   │
│     ┆ 00:00:00    ┆             ┆             ┆   ┆             ┆             ┆             ┆             │
└─────┴─────────────┴─────────────┴─────────────┴───┴─────────────┴─────────────┴─────────────┴─────────────┘
```

using Expression to apply Aggregation

In [ ]:
```
# change Ledger to a categorical dtype
# let's review our Ledger Data Frame we created
print(ledger.shape)
ledger.sample(5)
# url = "https://theunitedstates.io/congress-legislators/legislators-historical.csv

# dtypes = {
#     "first_name": pl.Categorical,
#     "gender": pl.Categorical,
#     "type": pl.Categorical,
#     "state": pl.Categorical,
#     "party": pl.Categorical,
# }

# dataset = pl.read_csv(url, dtypes=dtypes).with_columns(
#     pl.col("birthday").str.strptime(pl.Date, strict=False)

# basic aggregation
# Basic aggregations

# You can easily combine different aggregations by adding multiple expressions in a
# There is no upper bound on the number of aggregations you can do,
# and you can make any combination you want.
# In the snippet below we do the following aggregations:

# Per GROUP "first_name" we

#     count the number of rows in the group:
#     short form: pl.count("party")
#     full form: pl.col("party").count()
#     aggregate the gender values groups:
#     full form: pl.col("gender")
#     get the first value of column "last_name" in the group:
#     short form: pl.first("last_name") (not available in Rust)
#     full form: pl.col("last_name").first()

# Besides the aggregation, we immediately sort the result
# and limit to the top 5 so that we have a nice summary overview.
# q = (
#     dataset.lazy()
#     .groupby("first_name")
#     .agg(
#         pl.count(),
#         pl.col("gender"),
#         pl.first("last_name"),
#     )
#     .sort("count", descending=True)
#     .limit(5)
# )

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

```
(100000, 8)
```

Out[ ]: shape: (5, 8)

LEDGER	ORG	FISCAL_YEAR	PERIOD	ACCOUNT	DEPT	LOCATION	POSTED_TOTAL
str	str	i64	i64	i64	i64	i64	i64
"ACTUALS"	"ABC Inc."	2021	2	30000	1000	19	284964
"ACTUALS"	"ABC Inc."	2023	6	30000	1700	11	924142
"ACTUALS"	"ABC Inc."	2021	1	21000	2300	15	655586
"ACTUALS"	"ABC Inc."	2022	10	35000	1600	12	282449
"ACTUALS"	"ABC Inc."	2023	6	15000	2000	16	701454

In [ ]:
```python
# conditional aggregation
# It's that easy! Let's turn it up a notch.
# Let's say we want to know how many delegates of a "state" are "Pro" or "Anti" adm
# We could directly query that in the aggregation without the need of a lambda
# or grooming the DataFrame.
q = (
    dataset.lazy()
    .groupby("state")
    .agg(
        (pl.col("party") == "Anti-Administration").sum().alias("anti"),
        (pl.col("party") == "Pro-Administration").sum().alias("pro"),
    )
    .sort("pro", descending=True)
    .limit(5)
)

df = q.collect()
print(df)


q = (
    dataset.lazy()
    .groupby("state", "party")
    .agg(pl.count("party").alias("count"))
    .filter(
        (pl.col("party") == "Anti-Administration")
        | (pl.col("party") == "Pro-Administration")
    )
    .sort("count", descending=True)
    .limit(5)
)

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

In [ ]:
```python
# 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 compute_age() -> pl.Expr:
    return date(2021, 1, 1).year - pl.col("birthday").dt.year()


def avg_birthday(gender: str) -> pl.Expr:
    return (
        compute_age()
        .filter(pl.col("gender") == gender)
        .mean()
        .alias(f"avg {gender} birthday")
    )


q = (
    dataset.lazy()
    .groupby("state")
    .agg(
        avg_birthday("M"),
        avg_birthday("F"),
        (pl.col("gender") == "M").sum().alias("# male"),
        (pl.col("gender") == "F").sum().alias("# female"),
    )
    .limit(5)
)

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

In [ ]:
```python
# Sorting

# It's common to see a DataFrame being sorted for the sole purpose of
# managing the ordering during a GROUPBY operation.
# Let's say that we want to get the names of the oldest
# and youngest politicians per state. We could SORT and GROUPBY.

def get_person() -> pl.Expr:
    return pl.col("first_name") + pl.lit(" ") + pl.col("last_name")


q = (
    dataset.lazy()
    .sort("birthday", descending=True)
    .groupby("state")
    .agg(
        get_person().first().alias("youngest"),
        get_person().last().alias("oldest"),
    )
```

```
    .limit(5)
)

df = q.collect()
print(df)

# However, if we also want to sort the names alphabetically, this breaks.
# Luckily we can sort in a groupby context separate from the DataFrame.

def get_person() -> pl.Expr:
    return pl.col("first_name") + pl.lit(" ") + pl.col("last_name")


q = (
    dataset.lazy()
    .sort("birthday", descending=True)
    .groupby("state")
    .agg(
        get_person().first().alias("youngest"),
        get_person().last().alias("oldest"),
        get_person().sort().first().alias("alphabetical_first"),
    )
    .limit(5)
)

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

In [ ]:
```
# We can even sort by another column in the groupby context.
# If we want to know if the alphabetically sorted name is male or female
# we could add: pl.col("gender").sort_by("first_name").first().alias("gender")
def get_person() -> pl.Expr:
    return pl.col("first_name") + pl.lit(" ") + pl.col("last_name")


q = (
    dataset.lazy()
    .sort("birthday", descending=True)
    .groupby("state")
    .agg(
        get_person().first().alias("youngest"),
        get_person().last().alias("oldest"),
        get_person().sort().first().alias("alphabetical_first"),
        pl.col("gender").sort_by("first_name").first().alias("gender"),
    )
    .sort("state")
    .limit(5)
)

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

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

using Expression to apply Folds

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	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…
1	240	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…
2	190	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…
3	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…
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 [ ]:
```python
# 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      ┆ 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",  │
```

```
| ┆ ["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 [ ]:
```python
# 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 [ ]:
```python
# 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

############################
# 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
############################
```

using Expression to apply Structs

Numpy conversion

In [ ]:
```
# select cols including stats
```

```
# context & operations
# context & Data Type Casting, Struct Data Type, Strings
# filter context
# group by context
# SQL Context
```

Functions, User defined functions and Windows function

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

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)
```