02_data_engineering

June 21, 2025

[ ]:
```
%load_ext dotenv
%dotenv
```

1 What are we doing?

1.1 Objectives

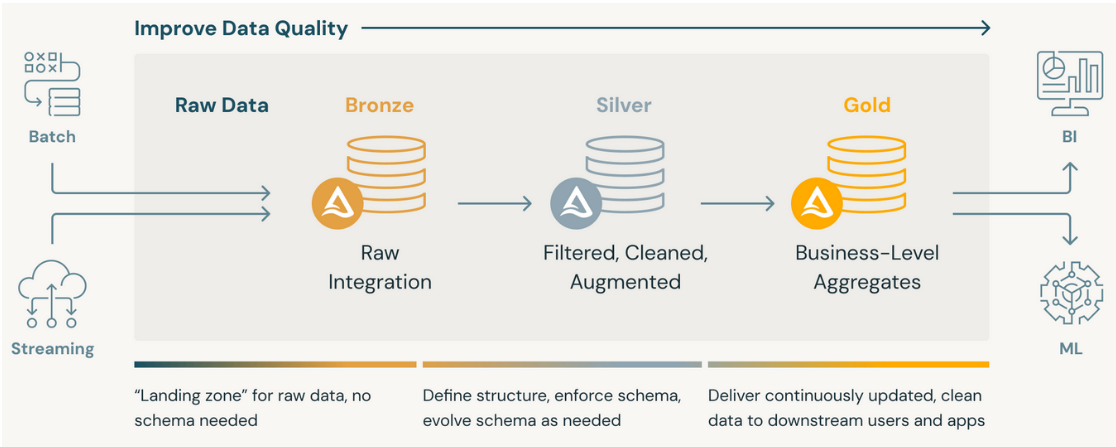
- Build a data pipeline that downloads price data from the internet, stores it locally, transforms it into return data, and stores the feature set.
  - Getting the data.
  - Schemas and index in dask.
- Explore the parquet format.
  - Reading and writing parquet files.
  - Read datasets that are stored in distributed files.
  - Discuss dask vs pandas as a small example of big vs small data.
- Discuss the use of environment variables for settings.
- Discuss how to use Jupyter notebooks and source code concurrently.
- Logging and using a standard logger.

1.2 About the Data

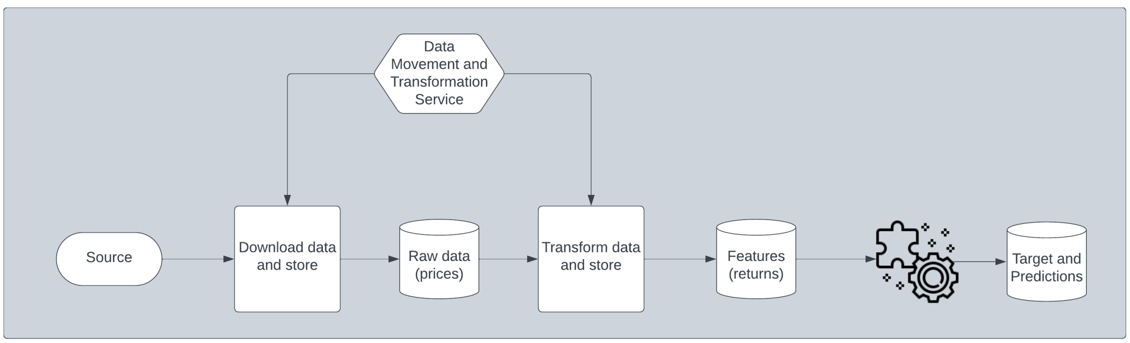
- We will download the prices for a list of stocks.
- The source is Yahoo Finance and we will use the API provided by the library yfinance.

1.3 Medallion Architecture

- The architecture that we are thinking about is called Medallion by DataBricks. It is an ELT type of thinking, although our data is well-structured.



- In our case, we would like to optimize the number of times that we download data from the internet.
- Ultimately, we will build a pipeline manager class that will help us control the process of obtaining and transforming our data.



2 Download Data

Download the Stock Market Dataset from Kaggle. Note that you may be required to register for a free account.

Extract all files into the directory: `./05_src/data/prices_csv/`

Your folder structure should include the following paths:

- `05_src/data/prices_csv/etfs`
- `05_src/data/prices_csv/stocks`

[ ]:
```
import pandas as pd
import os
import sys
from glob import glob

sys.path.append(os.getenv('SRC_DIR'))

from utils.logger import get_logger
```

```
_logs = get_logger(__name__)
```

A few things to notice in the code chunk above:

- Libraries are ordered from high-level to low-level libraries from the package manager (pip in this case, but could be conda, poetry, etc.)
- The command `sys.path.append("../05_src/)` will add the `../05_src/` directory to the path in the Notebook's kernel. This way, we can use our modules as part of the notebook.
- Local modules are imported at the end.
- The function `get_logger()` is called with `__name__` as recommended by the documentation.

Now, to load the historical price data for stocks and ETFs, we could use:

```
[ ]: import random

stock_files = glob(os.path.join(os.getenv('SRC_DIR'), "data/prices_csv/stocks/*.
 ↪csv"))

random.seed(42)
stock_files = random.sample(stock_files, 60)

dt_list = []
for s_file in stock_files:
    _logs.info(f"Reading file: {s_file}")
    dt = pd.read_csv(s_file).assign(
        source = os.path.basename(s_file),
        ticker = os.path.basename(s_file).replace('.csv', ''),
        Date = lambda x: pd.to_datetime(x['Date'])
    )
    dt_list.append(dt)
stock_prices = pd.concat(dt_list, axis = 0, ignore_index = True)
```

Verify the structure of the `stock_prices` data:

```
[ ]: stock_prices.info()
```

We can subset our ticker data set using standard indexing techniques. A good reference for this type of data manipulation is Panda's Documentation and Cookbook.

From the subset data frame, select one column and convert to list.

```
[ ]: select_tickers = stock_prices['ticker'].unique().tolist()
select_tickers
```

3 Storing Data in CSV

- We have some data. How do we store it?
- We can compare two options, CSV and Parqruet, by measuring their performance:

- Time to save.
- Space required.

[ ]:
```
def get_dir_size(path='.'):
    '''Returns the total size of files contained in path.'''
    total = 0
    with os.scandir(path) as it:
        for entry in it:
            if entry.is_file():
                total += entry.stat().st_size
            elif entry.is_dir():
                total += get_dir_size(entry.path)
    return total
```

[ ]:
```
import time
import shutil
```

[ ]:
```
temp = os.getenv("TEMP_DATA")
csv_dir = os.path.join(temp, "csv")
shutil.rmtree(csv_dir, ignore_errors=True)
stock_csv = os.path.join(csv_dir, "stock_px.csv")
os.makedirs(csv_dir, exist_ok=True)
```

[ ]:
```
start = time.time()
stock_prices.to_csv(stock_csv, index = False)
end = time.time()

_logs.info(f'Writing data ({stock_prices.shape}) to csv took {end - start}␣
 ↪seconds.')
_logs.info(f'CSV file size { os.path.getsize(stock_csv)*1e-6 } MB')
```

3.1 Save Data to Parquet

3.1.1 Dask

We can work with with large data sets and parquet files. In fact, recent versions of pandas support pyarrow data types and future versions will require a pyarrow backend. The pyarrow library is an interface between Python and the Appache Arrow project. The parquet data format and Arrow are projects of the Apache Foundation.

However, Dask is much more than an interface to Arrow: Dask provides parallel and distributed computing on pandas-like dataframes. It is also relatively easy to use, bridging a gap between pandas and Spark.

[ ]:
```
import dask.dataframe as dd

parquet_dir = os.path.join(temp, "parquet")
shutil.rmtree(parquet_dir, ignore_errors=True)
```

```
os.makedirs(parquet_dir, exist_ok=True)
```

[ ]:
```
px_dd = dd.from_pandas(stock_prices, npartitions = len(select_tickers))

start = time.time()
px_dd.to_parquet(parquet_dir, engine = "pyarrow")
end = time.time()

_logs.info(f'Writing dd ({stock_prices.shape}) to parquet took {end - start}␣
↪seconds.')
_logs.info(f'Parquet file size { get_dir_size(parquet_dir)*1e-6 } MB')
```

3.1.2 Parquet files and Dask Dataframes

- Parquet files are immutable: once written, they cannot be modified.
- Dask DataFrames are a useful implementation to manipulate data stored in parquets.
- Parquet and Dask are not the same: parquet is a file format that can be accessed by many applications and programming languages (Python, R, PowerBI, etc.), while Dask is a package in Python to work with large datasets using distributed computation.
- **Dask is not for everything** (see Dask DataFrames Best Practices).
  - Consider cases suchas small to large joins, where the small dataframe fits in memory, but the large one does not.
  - If possible, use pandas: reduce, then use pandas.
  - Pandas performance tips apply to Dask.
  - Use the index: it is beneficial to have a well-defined index in Dask DataFrames, as it may speed up searching (filtering) the data. A one-dimensional index is allowed.
  - Avoid (or minimize) full-data shuffling: indexing is an expensive operations.
  - Some joins are more expensive than others.
    - Not expensive:
      - Join a Dask DataFrame with a pandas DataFrame.
      - Join a Dask DataFrame with another Dask DataFrame of a single partition.
      - Join Dask DataFrames along their indexes.
    - Expensive:
      - Join Dask DataFrames along columns that are not their index.

4 How do we store prices?

- We can store our data as a single blob. This can be difficult to maintain, especially because parquet files are immutable.
- Strategy: organize data files by ticker and date. Update only latest month.

```
[ ]: # CLean up before start
PRICE_DATA = os.getenv("PRICE_DATA")
import shutil
if os.path.exists(PRICE_DATA):
    shutil.rmtree(PRICE_DATA)
```

```
[ ]: stock_prices.columns
```

```
[ ]: for ticker in stock_prices['ticker'].unique():
    ticker_dt = stock_prices[stock_prices['ticker'] == ticker]
    ticker_dt = ticker_dt.assign(Year = ticker_dt.Date.dt.year)
    for yr in ticker_dt['Year'].unique():
        yr_dd = dd.from_pandas(ticker_dt[ticker_dt['Year'] == yr],2)
        yr_path = os.path.join(PRICE_DATA, ticker, f"{ticker}_{yr}")
        os.makedirs(os.path.dirname(yr_path), exist_ok=True)
        yr_dd.to_parquet(yr_path, engine = "pyarrow")
```

Why would we want to store data this way?

- Easier to maintain. We do not update old data, only recent data.
- We can also access all files as follows.

5 Load, Transform and Save

5.1 Load

- Parquet files can be read individually or as a collection.
- `dd.read_parquet()` can take a list (collection) of files as input.
- Use `glob` to get the collection of files.

```
[ ]: from glob import glob

parquet_files = glob(os.path.join(PRICE_DATA, "**/*.parquet"), recursive = True)
dd_px = dd.read_parquet(parquet_files).set_index("ticker")
```

5.2 Transform

- This transformation step will create a *Features* data set. In our case, features will be stock returns (we obtained prices).
- Dask dataframes work like pandas dataframes: in particular, we can perform groupby and apply operations.
- Notice the use of an anonymous (lambda) function in the apply statement.

```
[ ]: dd_shift = dd_px.groupby('ticker', group_keys=False).apply(
    lambda x: x.assign(Close_lag_1 = x['Close'].shift(1))
)
```

```python
dd_rets = dd_shift.assign(
    Returns = lambda x: x['Close']/x['Close_lag_1'] - 1
)
```

5.3 Lazy Exection

What does `dd_rets` contain?

```python
dd_rets
```

- Dask is a lazy execution framework: commands will not execute until they are required.
- To trigger an execution in dask use `.compute()`.

```python
dd_rets.compute()
```

5.4 Save

- Apply transformations to calculate daily returns
- Store the enriched data, the silver dataset, in a new directory.
- Should we keep the same namespace? All columns?

```python
# CLean up before save
FEATURES_DATA = os.getenv("FEATURES_DATA")
if os.path.exists(FEATURES_DATA):
    shutil.rmtree(FEATURES_DATA)
dd_rets.to_parquet(FEATURES_DATA, overwrite = True)
```

6 Optional: from Jupyter to Command Line

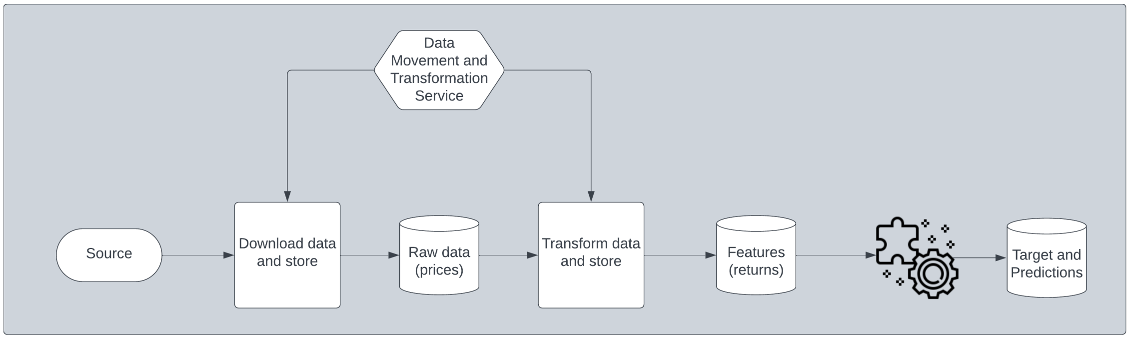
- We have drafted our code in a Jupyter Notebook.
- Finalized code should be written in Python modules.

6.1 Object Oriented vs Functional Programming

- We can use classes to keep parameters and functions together.
- We *could* use Object Oriented Programming, but parallelization of data manipulation and modelling tasks benefit from *Functional Programming*.
- An Idea:
  - Data Oriented Programming.
  - Use the class to bundle together parameters and functions.
  - Use stateless operations and treat all data objects as immutable (we do not modify them, we overwrite them).
  - Take advantage of `@staticmethod`.

The code is in `./05_src/stock_prices/data_manager.py`.

Our original design was:



[ ]:
```
from stock_prices.data_manager import DataManager
dm = DataManager()
```

Download all prices.

[ ]:
```
dm.process_sample_files()
```

Finally, add features to the data set and save to a *feature store*.

[ ]:
```
dm.featurize()
```