### Creating Database:

import pandas as pd

from sqlalchemy import create\_engine, text

# Create a SQLAlchemy engine to connect to MySQL

engine = create\_engine('mysql+pymysql://dev:dev@127.0.0.1:3307?charset=utf8mb4',isolation\_level="AUTOCOMMIT")

conn = engine.connect()

db\_name = 'python\_project'

conn.execute(text(f"CREATE DATABASE IF NOT EXISTS {db\_name}"))

print(f"Database '{db\_name}' created successfully.")

### ETL process:

import os

import logging

import pandas as pd

from sqlalchemy import create\_engine, MetaData, Table, Column, Integer, String, Text, TIMESTAMP, func

from sqlalchemy.orm import sessionmaker

from sqlalchemy.exc import SQLAlchemyError

# Set up database connection

db\_url = 'mysql+pymysql://dev:dev@127.0.0.1:3307/python\_project?charset=utf8mb4'

engine = create\_engine(db\_url, isolation\_level="AUTOCOMMIT")

metadata = MetaData()

# Define the logging table in SQLAlchemy

stocks\_data\_log = Table(

'stocks\_data\_log', metadata,

Column('ID', Integer, primary\_key=True, autoincrement=True),

Column('ETL\_PROCESS', String(255), default='stocks\_data'),

Column('Log\_time', TIMESTAMP, default=func.now()),

Column('Log\_level', String(10), default='INFO'),

Column('ERROR\_DESC', Text, nullable=True)

)

# Create the table if it doesn't exist

metadata.create\_all(engine)

# Session setup

Session = sessionmaker(bind=engine)

session = Session()

# Custom logging handler to log into the MySQL database

class DBLogHandler(logging.Handler):

def emit(self, record):

log\_message = self.format(record)

log\_entry = {

'ETL\_PROCESS': stock\_symbol,

'Log\_level': record.levelname,

'ERROR\_DESC': log\_message if record.levelname == 'ERROR' else None

}

try:

# Insert the log entry into the database

session.execute(stocks\_data\_log.insert().values(log\_entry))

session.commit()

except SQLAlchemyError as e:

print(f"Failed to log to the database: {str(e)}")

session.rollback()

# Set up logging to file and database

logging.basicConfig(filename="process\_log.log", level=logging.INFO, format="%(asctime)s - %(message)s")

logger = logging.getLogger()

# Add the custom database handler

db\_handler = DBLogHandler()

db\_handler.setLevel(logging.INFO)

logger.addHandler(db\_handler)

# Define the directory and database connection for stock data

stock\_files\_directory = "C://Users//boris//OneDrive//Documents//DataEngenier//python//python\_project//stock\_dataset//stocks"

security\_name\_file = "C://Users//boris//OneDrive//Documents//DataEngenier//python//python\_project//stock\_dataset//symbols\_valid\_meta.csv"

engine = create\_engine('mysql+pymysql://dev:dev@127.0.0.1:3307/python\_project?charset=utf8mb4', isolation\_level="AUTOCOMMIT")

# Load security names from the security name file

security\_df = pd.read\_csv(security\_name\_file)

# Get the list of stock files

stock\_files = sorted([f for f in os.listdir(stock\_files\_directory) if f.endswith('.csv')])

# Process each file

for stock\_file in stock\_files:

try:

# Extract stock symbol from the file name (assuming it's the part before '.csv')

stock\_symbol = os.path.splitext(stock\_file)[0]

# Load the stock data CSV file

stock\_data = pd.read\_csv(os.path.join(stock\_files\_directory, stock\_file))

# Add a new column 'Symbol' to the stock data

stock\_data['Symbol'] = stock\_symbol

# Merge with security names using the 'Symbol' column

merged\_data = pd.merge(stock\_data, security\_df[['Symbol', 'Security Name']], on='Symbol', how='left')

merged\_data = merged\_data.rename(columns={'Security Name': 'SecurityName'})

# Push the data into the MySQL database in chunks of 500 rows

merged\_data.to\_sql('stock\_rates', con=engine, if\_exists='append', index=False, chunksize=500)

# Log success to both the file and the database

logger.info(f"Successfully processed and inserted data for stock symbol: {stock\_symbol}")

except Exception as e:

# Log error to both the file and the database

logger.error(f"Error processing stock file {stock\_file}: {str(e)}")

### Stock details:

import json

from datetime import datetime

from sqlalchemy import create\_engine, MetaData, Table, select

from sqlalchemy.exc import SQLAlchemyError

class InvalidInputError(Exception):

"""Custom exception for invalid inputs."""

pass

class StockDataFetcher:

def \_\_init\_\_(self, db\_url: str):

"""Initialize the StockDataFetcher with the database URL."""

self.engine = create\_engine(db\_url, isolation\_level="AUTOCOMMIT")

self.metadata = MetaData()

self.stocks = Table('stock\_rates', self.metadata, autoload\_with=self.engine)

@staticmethod

def validate\_inputs(start\_date: str, end\_date: str, check\_date\_range: bool = True):

"""Validates the input parameters and ensures the maximum range of 30 days if check\_date\_range is True."""

try:

start\_date\_obj = datetime.strptime(start\_date, '%Y-%m-%d')

end\_date\_obj = datetime.strptime(end\_date, '%Y-%m-%d')

except ValueError:

raise InvalidInputError("Invalid date format. Expected format is YYYY-MM-DD.")

# Only check the date range if check\_date\_range is True

if check\_date\_range and (end\_date\_obj - start\_date\_obj).days > 30:

raise InvalidInputError("The date range cannot exceed 30 days.")

# Ensure the end\_date is not before the start\_date

if end\_date\_obj < start\_date\_obj:

raise InvalidInputError("end\_date cannot be before start\_date.")

def fetch\_stock\_data(self, stock\_symbol: str, start\_date: str, end\_date: str, file\_path: str):

"""Fetch stock data from MySQL using SQLAlchemy Core."""

# Validate the inputs

self.validate\_inputs(

start\_date, end\_date)

# Prepare the SQL query

query = select(self.stocks).where(

self.stocks.c.Symbol == stock\_symbol,

self.stocks.c.Date.between(start\_date, end\_date)

)

try:

# Execute the query

with self.engine.connect() as conn:

result = conn.execute(query)

stock\_data = [dict(zip(result.keys(), row)) for row in result]

# Convert the stock data to a JSON formatted string, each entry on a new line

stock\_data\_json = json.dumps(stock\_data, default=str, indent=4)

with open(file\_path, 'w') as f:

f.write(stock\_data\_json)

print(f"JSON data saved to {file\_path}")

return stock\_data

except SQLAlchemyError as e:

raise InvalidInputError(f"Database error: {str(e)}")

# usage

if \_\_name\_\_ == "\_\_main\_\_":

stock\_symbol = "AAPL"

start\_date = "2019-08-01"

end\_date = "2019-08-30"

current\_date = datetime.now().strftime('%Y-%m-%d')

file\_path = fr'C:\Users\boris\OneDrive\Documents\DataEngenier\python\stock\_data\_{stock\_symbol}\_{current\_date}.json'

db\_url = 'mysql+pymysql://dev:dev@127.0.0.1:3307/python\_project?charset=utf8mb4'

fetcher = StockDataFetcher(db\_url)

stock\_data = fetcher.fetch\_stock\_data(stock\_symbol, start\_date, end\_date, file\_path)

### Stock stats:

import json

from datetime import datetime

from sqlalchemy import create\_engine, Column, String, Float, Date, select, func

from sqlalchemy.orm import sessionmaker, declarative\_base

from sqlalchemy import and\_

from sqlalchemy.exc import SQLAlchemyError

import traceback

from stock\_details import StockDataFetcher

# Define the base class for models

Base = declarative\_base()

# Define the StockRate model class

class StockRate(Base):

\_\_tablename\_\_ = 'stock\_rates'

Symbol = Column(String, primary\_key=True)

SecurityName = Column(String)

Date = Column(Date, primary\_key=True)

Close = Column(Float)

High = Column(Float)

Low = Column(Float)

# Define the Stocks\_Stats class

class Stocks\_Stats:

def \_\_init\_\_(self, db\_url):

self.engine = create\_engine(db\_url)

self.Session = sessionmaker(bind=self.engine)

def fetch\_stock(self, stock\_symbol\_lst, startdate, enddate, file\_path):

session = self.Session()

StockDataFetcher.validate\_inputs(startdate, enddate, False)

try:

# Subquery 1: Calculate Yield based on startdate and enddate

start\_subquery = (

session.query(

StockRate.Symbol.label('Symbol'),

StockRate.Close.label('start\_close')

)

.filter(and\_(StockRate.Date == startdate, StockRate.Symbol.in\_(stock\_symbol\_lst)))

.subquery()

)

end\_subquery = (

session.query(

StockRate.Symbol.label('Symbol'),

StockRate.Close.label('end\_close')

)

.filter(and\_(StockRate.Date == enddate, StockRate.Symbol.in\_(stock\_symbol\_lst)))

.subquery()

)

# Calculate Yield: ((end\_close - start\_close) / start\_close) \* 100

yield\_cte = (

session.query(

start\_subquery.c.Symbol,

start\_subquery.c.start\_close,

end\_subquery.c.end\_close,

((end\_subquery.c.end\_close - start\_subquery.c.start\_close) /

start\_subquery.c.start\_close \* 100).label('Yield')

)

.join(end\_subquery, start\_subquery.c.Symbol == end\_subquery.c.Symbol)

.subquery()

)

# Subquery 2: Fetch Max\_rate, Min\_rate, AVG\_Rate for the given symbols and date range

stats\_cte = (

session.query(

StockRate.Symbol,

StockRate.SecurityName,

func.max(StockRate.High).label('Max\_rate'),

func.min(StockRate.Low).label('Min\_rate'),

func.avg(StockRate.Close).label('AVG\_Rate')

)

.filter(and\_(StockRate.Date.between(startdate, enddate), StockRate.Symbol.in\_(stock\_symbol\_lst)))

.group\_by(StockRate.Symbol, StockRate.SecurityName)

.subquery()

)

# Final query: Join Yield and stats data, order by Yield descending

final\_query = (

session.query(

yield\_cte.c.Symbol,

stats\_cte.c.SecurityName,

yield\_cte.c.start\_close,

yield\_cte.c.end\_close,

stats\_cte.c.Max\_rate,

stats\_cte.c.Min\_rate,

stats\_cte.c.AVG\_Rate,

yield\_cte.c.Yield

)

.join(stats\_cte, yield\_cte.c.Symbol == stats\_cte.c.Symbol)

.order\_by(yield\_cte.c.Yield.desc())

)

# Execute query and fetch results

results = final\_query.all()

result\_list =[

{

"Symbol": row.Symbol,

"Security\_Name": row.SecurityName,

"Close\_start\_price": row.start\_close,

"Close\_end\_price": row.end\_close,

"Max\_rate": row.Max\_rate,

"Min\_rate": row.Min\_rate,

"AVG\_Rate": row.AVG\_Rate,

"Yield": row.Yield

}

for row in results

]

stock\_stat\_json = json.dumps(result\_list, default=str, indent=4)

# Optionally save to a file

if file\_path and results:

with open(file\_path, 'w') as f:

f.write(stock\_stat\_json)

print(f"JSON data saved to {file\_path}")

# Return results

return results

except SQLAlchemyError as e:

print(f"An error occurred: {e}")

traceback.print\_exc() # Get the full error traceback for debugging

finally:

session.close()

#usage

if \_\_name\_\_ == "\_\_main\_\_":

db\_url = 'mysql+pymysql://dev:dev@127.0.0.1:3307/python\_project?charset=utf8mb4'

stocks\_stats = Stocks\_Stats(db\_url)

# Example fetch with startdate and enddate, filtering by stock\_symbol\_lst

stock\_symbol\_lst = ['BRY', 'NUAN', 'SCKT']

startdate = '1999-11-01'

enddate = '1999-11-30'

current\_date = datetime.now().strftime('%Y-%m-%d')

file\_path = fr'C:\Users\boris\OneDrive\Documents\DataEngenier\python\stock\_stat\_{current\_date}.json'

data = stocks\_stats.fetch\_stock(stock\_symbol\_lst, startdate, enddate, file\_path)

if data: # Check if stock\_data is not None

for row in data:

print(row)

else:

print("No data fetched.")