Project Step 3

Siyuan Qi
Yaodan Zhang
Jiajun Lin

May 21, 2024

1 Descriptions

Conceptual Database Design

This is a conceptual database design for a trading system, detailing the relationships between stocks, ETFs, indices, users, and trading sessions. In this system, users engage in trading sessions dealing with a variety of financial instruments. A user's portfolio is managed over time, with asset allocation adjusted as needed. The Sharpe Ratio is used to assess the risk-adjusted return of the investments. It includes the following specifications:

- Stocks are identified by a stock ID (sId) and each stock record includes its ticker symbol, open price, high price, low price, close price, and the date of these prices.
- ETFs are identified by a ETF ID (fId) and each ETF record includes its ticker symbol, open price, high price, low price, close price, and the date.
- Indices are identified by an index ID (iId) and each index record includes its ticker symbol, open price, high price, low price, close price, and the date.
- Stocks, ETFs, and indices are interconnected through the "contain" relationships indicating their composition.
- User trading sessions are identified by a session ID and include the start date, end date, the amount of trade, the position (long/short), and an underlying ID that connects to stocks, ETFs, or indices.
- Users are identified by a user ID (uId) and records include the first name, last name, email, phone, and trading preferences.
- The trading result is recorded with risk and return metrics to calculate the Sharpe Ratio.
- Each user manages a portfolio, identified by the user ID, which records the start time, end time, and asset allocation.
- User trading sessions are managed by users and connected to the assets being traded.

2 ER Diagram

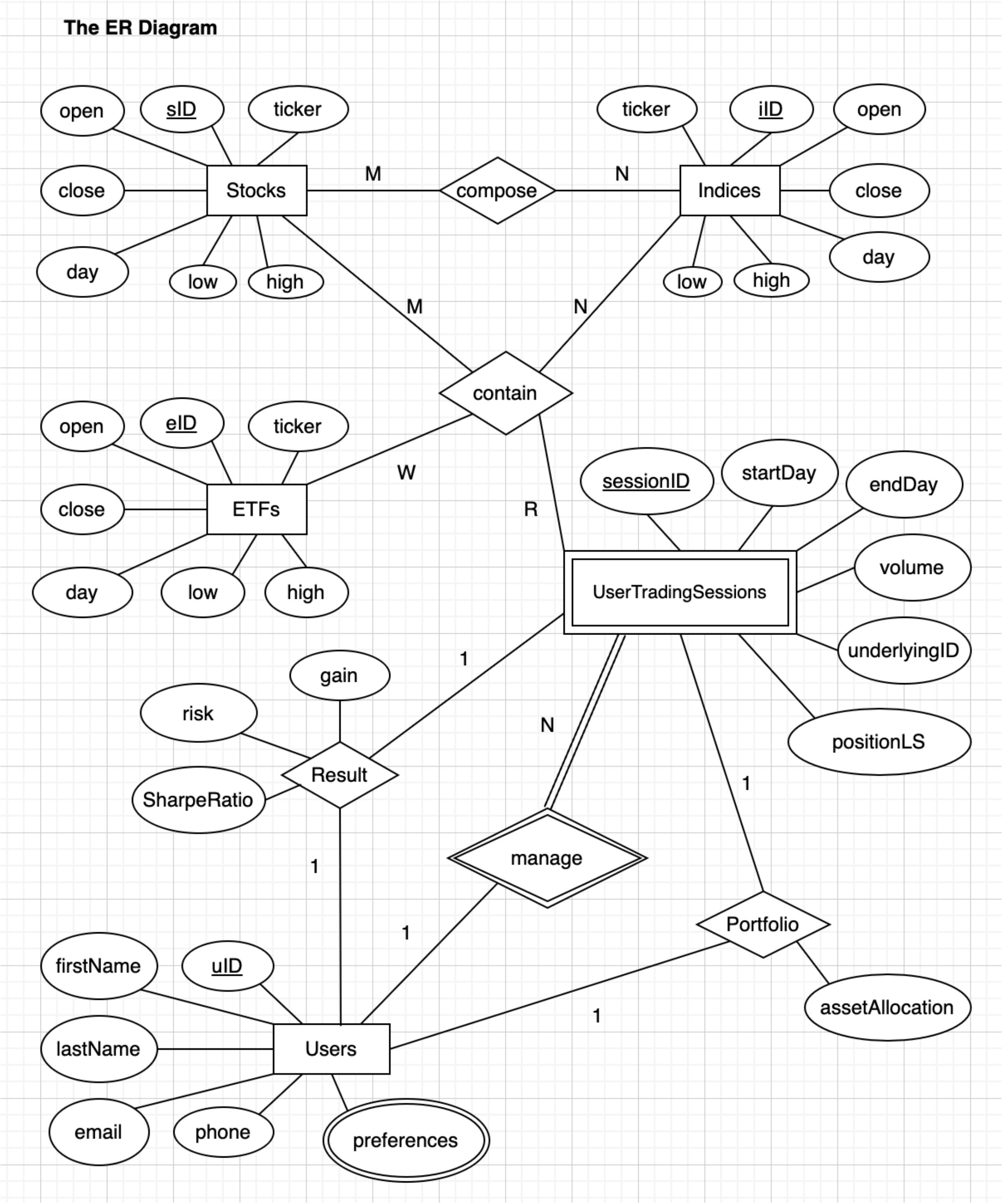
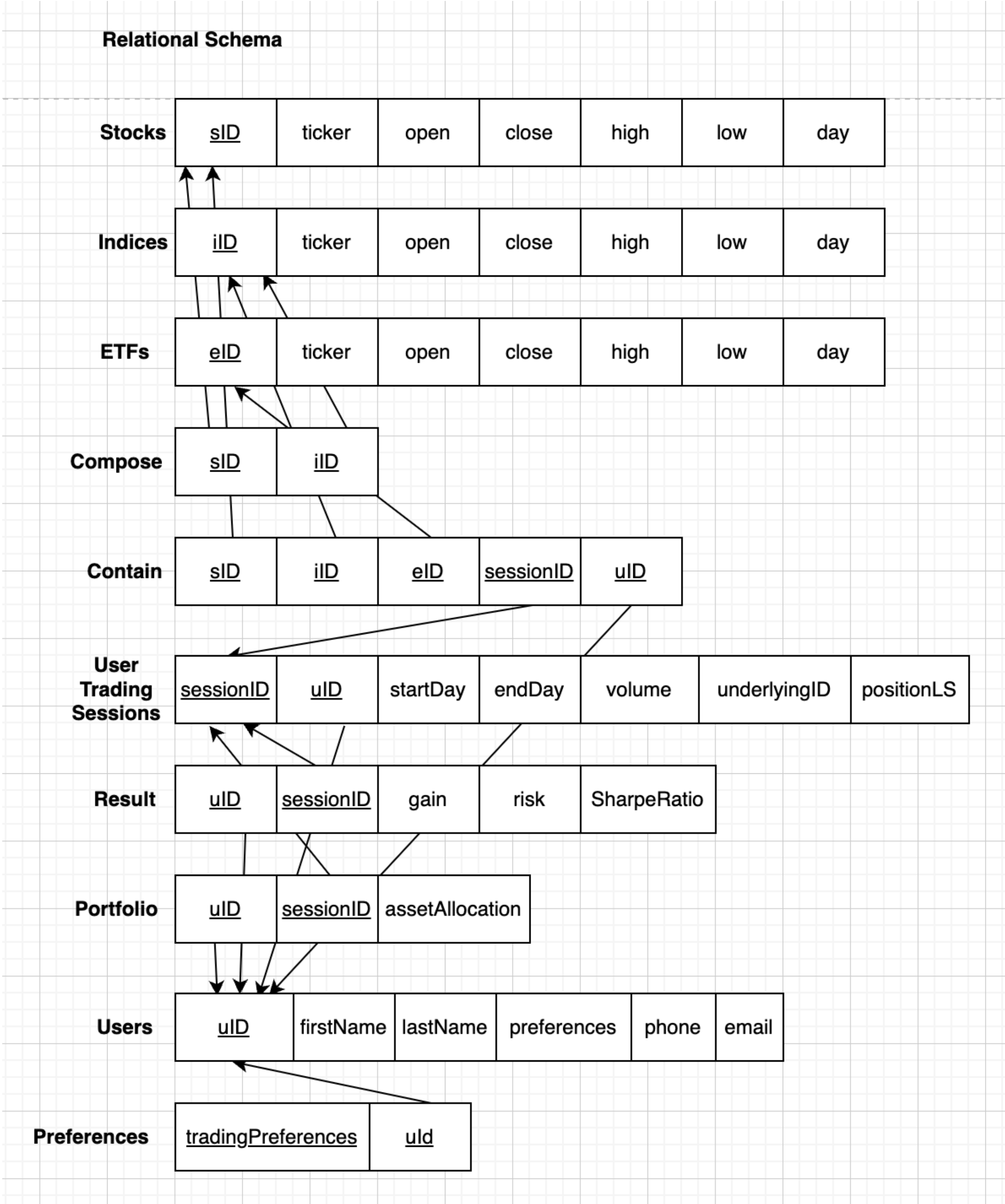


Figure 1: ER Diagram

3 Relational Schema



Stocks	sID	ticker	open	close	high	low	day
Indices	iID	ticker	open	close	high	low	day
ETFs	eID	ticker	open	close	high	low	day
Compose	sID	iID					
Contain	sID	iID	eID	sessionID	uID		
User Trading Sessions	sessionID	uID	startDay	endDay	volume	underlyingID	positionLS
Result	uID	sessionID	gain	risk	SharpeRatio		
Portfolio	uID	sessionID	assetAllocation				
Users	uID	firstName	lastName	preferences	phone	email	
Preferences	tradingPreferences	uId					

Figure 2: Relational Schemas

4 Prepare Datasets

The following details the procedure for retrieving real-world financial data to populate the trading system's database. The data acquisition will be performed using the Python ‘yfinance’ module, which allows us to fetch historical market data from Yahoo Finance. Additionally, for the purpose of testing and demonstration, we will craft mock data for non-financial entities, simulating realistic trading scenarios. This allows us to assume the role of traders and create a dynamic dataset that reflects various user interactions and trading behaviors within the system.

Retrieving Financial Data

The following Python code utilizes the ‘yfinance’ library to download historical data for stocks. The ‘get_stock_data’ function retrieves data points such as ‘Open’, ‘Close’, ‘High’, ‘Low’, and ‘Date’ for a specified stock ticker. These data points are in line with the attributes defined in the Relational Schema for the ‘Stock’ entity. Similar operations can be performed for other entities.

```python
import yfinance as yf

def get_stock_data(ticker):

    # Downloads the stock's historical data using yfinance.
    data = yf.download(ticker, period="max")

    # Filters the dataframe to include only the required columns.
    data = data[["Open", "Close", "High", "Low"]]

    # Resets the index to convert the "Date" index into a column.
    data.reset_index(inplace=True)

    # Renames the columns to adhere to the relational schemas.
    data.columns = ["date", "open", "close", "high", "low"]
    return data

# Demonstrates the function usage with Apple's stock ticker ‘AAPL’.
ticker = "AOS"
stock_data = get_stock_data(ticker)
print(stock_data.head())
```

Listing 1: An Example to Retrieve Data

Integration with the Database

The output of the ‘get_stock_data’ function is a pandas DataFrame, which can be easily exported to various formats, such as CSV, to integrate with a database.

Sourcing Additional Data

For the non-financial entities, we will generate these data assuming the role of a mock user. Essentially, we can simulate the trading activity as if we were the trader. This approach allows us to create realistic yet fictitious data sets that can be used to test and demonstrate the functionality of the trading system database.

5 How the App Works

Here is an overview of our web applciation, which is built using the Flask framework and interfaces with a MySQL database.

Database Integration

- Extensive use of MySQL for executing queries to fetch, insert, or update database records reflecting user activities, account management, and asset price data.

Trading Sessions and Portfolio Management

- **Home Page:** Automatically redirects users to the main home page.
- **Sign Up and Login:** Users can register a new account or log into an existing one. The application handles user data input and verification against the database records.
- **Trading Sessions:** Users have the ability to create new trading sessions, view results of past sessions, or delete them. The application facilitates the selection and handling of different asset types including stocks, indices, and ETFs.
- **Portfolio Page:** Displays details and weights of each trading session within a user's portfolio.

Asset Management and Alpha Visualization

- **Asset Pages:** Dedicated pages for stocks, indices, and ETFs, where specific tickers and their price data can be viewed.
- **Price Data:** Displays detailed historical price data for each asset type, including open, high, low, and close prices over selected periods.
- **Alpha Pages:** Provides functionality for analyzing asset performance using indicators such as Money Flow Multiplier and Relative Strength Index. The results are visualized through graphs plotted using matplotlib.

Utility Functions

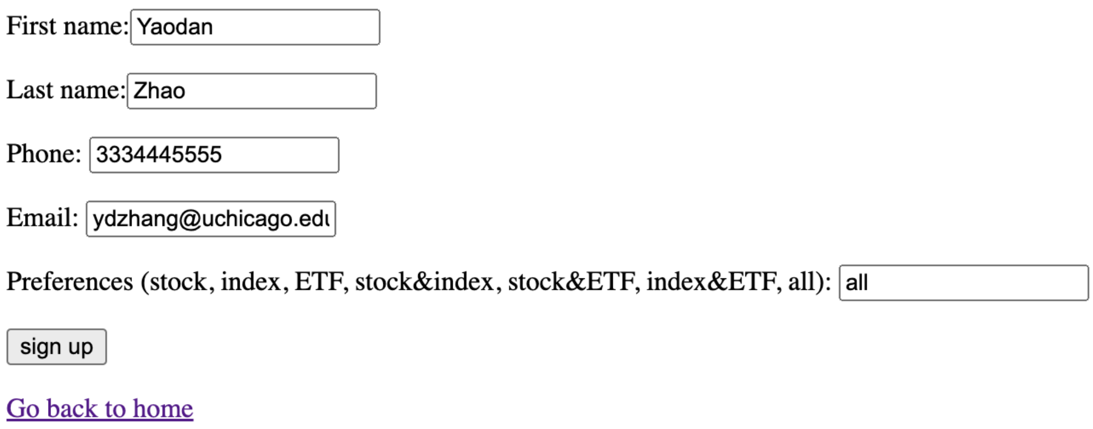
- The `utils.py` module is utilized for data processing and plotting, essential for rendering financial indicators and integrating data analysis into the user interface.

6 Queries and Results

This section details selected queries run within the application, including those that result in graphical outputs. The queries demonstrate how the system retrieves and processes data to provide valuable insights into trading activities and asset performance. Our code (Python with SQL integrated) is attached at the end of document.

6.1 Query 1: Trading Sign Up

Description: This query displays the sign-up page for trading sessions and manages user profiles using Python and SQL.



First name: Yaodan

Last name: Zhao

Phone: 3334445555

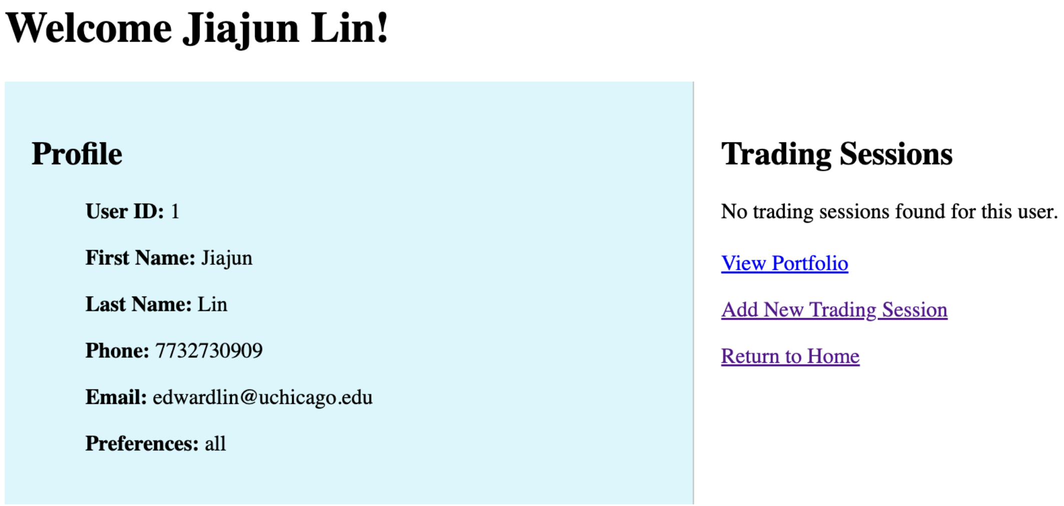
Email: ydzhang@uchicago.edu

Preferences (stock, index, ETF, stock&index, stock&ETF, index&ETF, all): all

Figure 3: Trading Sign Up

6.2 Query 2: User Detail and Home Page

Description: Retrieves current user details from the database, including basic information and trading sessions using Python and SQL.



Welcome Jiajun Lin!

Profile

User ID: 1

First Name: Jiajun

Last Name: Lin

Phone: 7732730909

Email: edwardlin@uchicago.edu

Preferences: all

Trading Sessions

No trading sessions found for this user.

View Portfolio

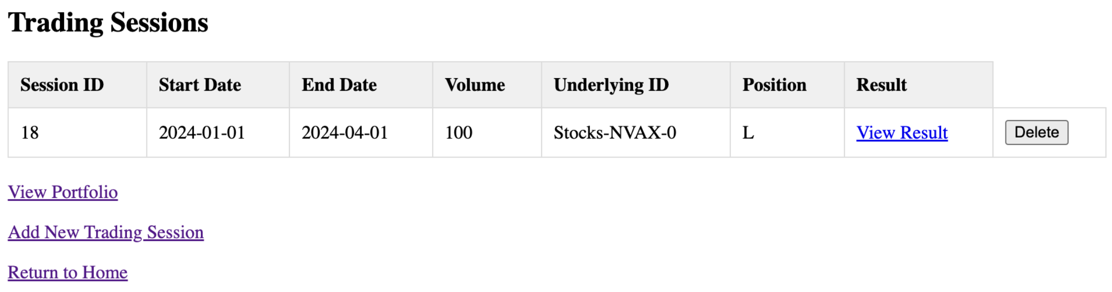
Add New Trading Session

Return to Home

Figure 4: User Detail and Home Page

6.3 Query 3: Trading Sessions and Portfolios

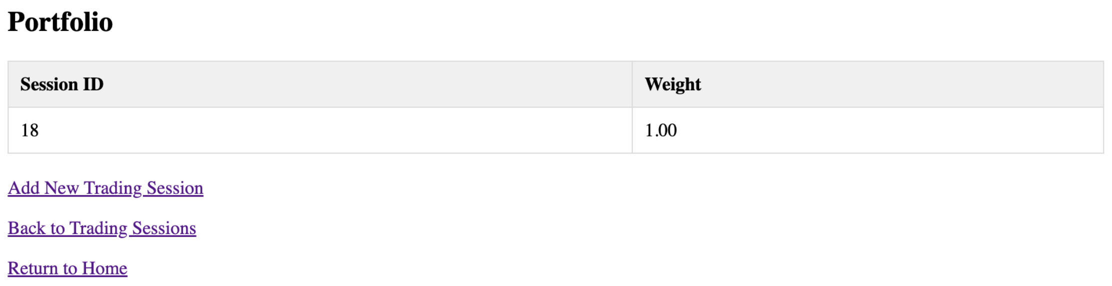
Description: Displays current trading sessions and portfolios using Python and SQL.



Trading Sessions

Session ID	Start Date	End Date	Volume	Underlying ID	Position	Result	
18	2024-01-01	2024-04-01	100	Stocks-NVAX-0	L	View Result	Delete

Figure 5: Trading Sessions



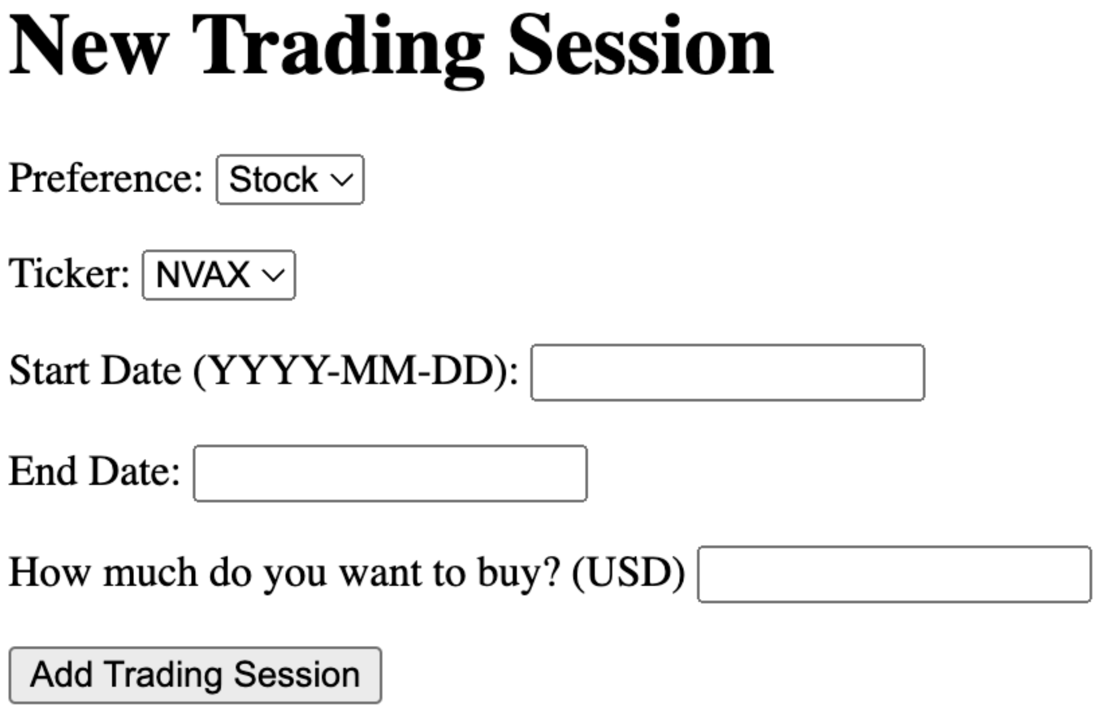
Portfolio

Session ID	Weight
18	1.00

Figure 6: Portfolios

6.4 Query 4: New Trading Sessions

Description: Creates a new trading session using Python and SQL.



New Trading Session

Preference: Stock

Ticker: NVAX

Start Date (YYYY-MM-DD):

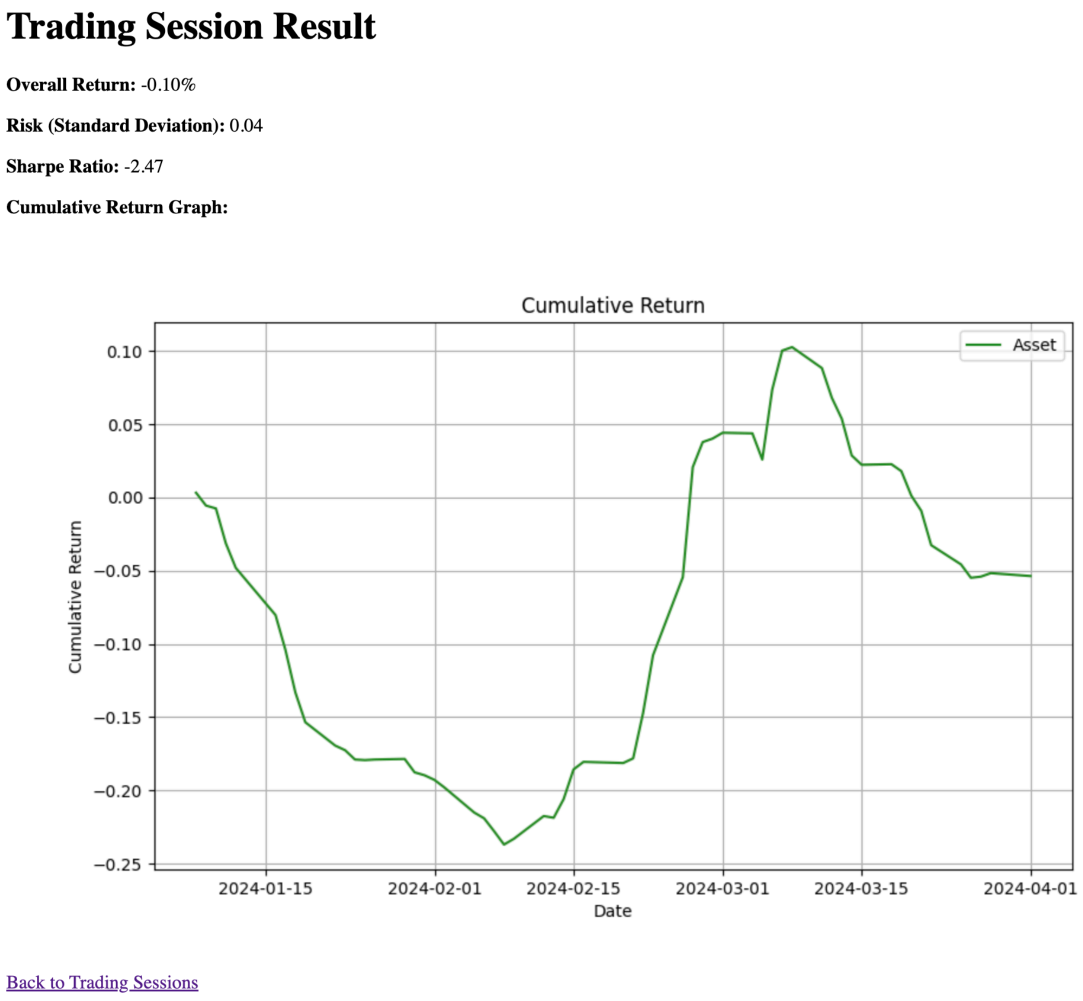
End Date:

How much do you want to buy? (USD)

Figure 7: New Trading Session

6.5 Query 5: Trading Results

Description: Fetches trading results from the database using Python and SQL.



Trading Session Result

Overall Return: -0.10%

Risk (Standard Deviation): 0.04

Sharpe Ratio: -2.47

Cumulative Return Graph:

Back to Trading Sessions

Figure 8: Trading Results

6.6 Query 6: Deletes Trading Sessions

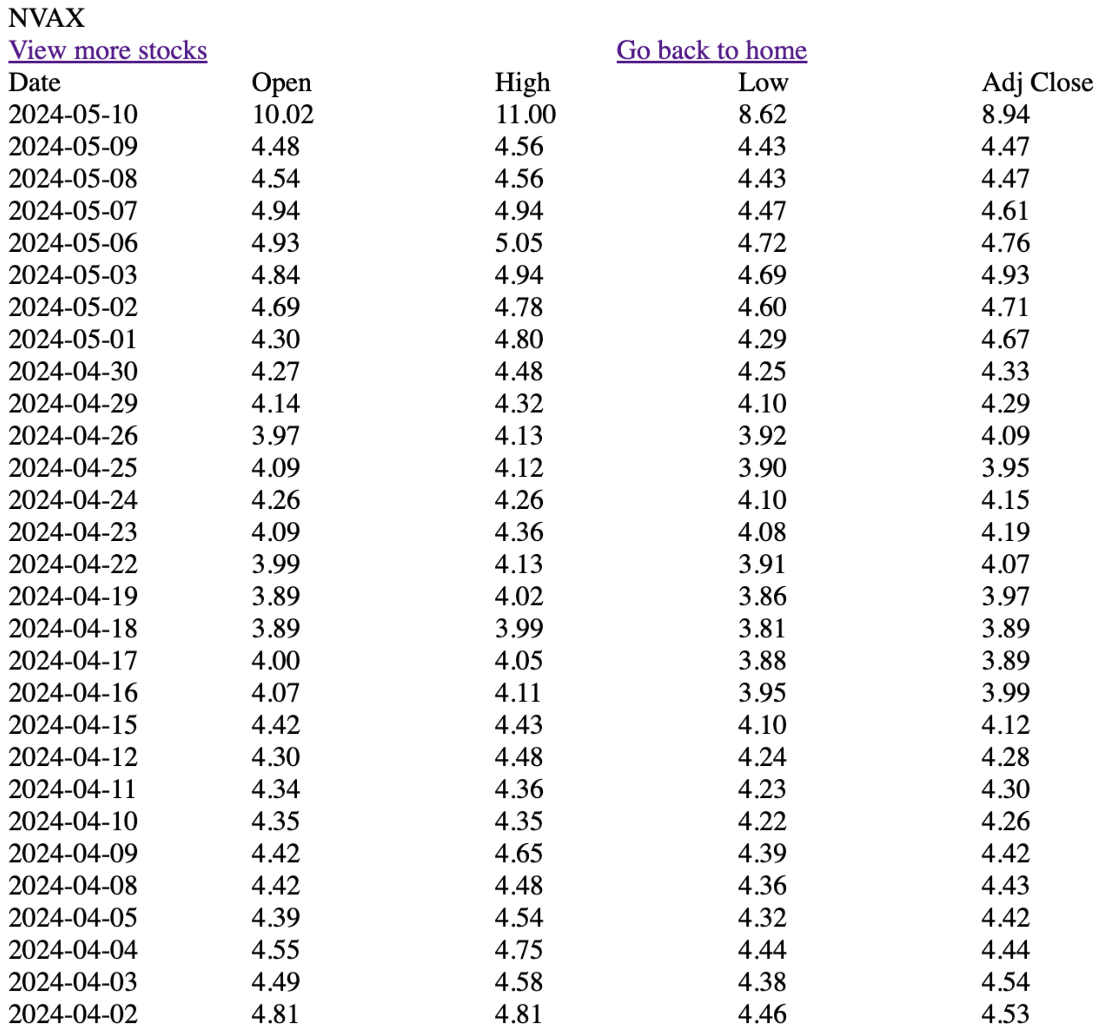
Description: Deletes a current trading session using Python and SQL.

```
@app.route("/ts/delete_trading_session/<session_id>/<user_id>", methods=["POST"])
def delete_trading_session(session_id, user_id):
    delete_id = session_id
    cursorObject.execute("DELETE FROM portfolio WHERE sessionID = %s", (delete_id,))
    cursorObject.execute("DELETE FROM Result WHERE sessionID = %s", (delete_id,))
    cursorObject.execute(
        "DELETE FROM usertradingsessions WHERE sessionID = %s", (delete_id,)
    )
    DB_Connection.commit()
    return redirect(url_for("user", user_id=user_id))
```

Listing 2: Python code for deleting a trading session

6.7 Query 7: Fetches Financial Data

Description: Retrieves financial data from a specific time range using Python and SQL.



NVAX

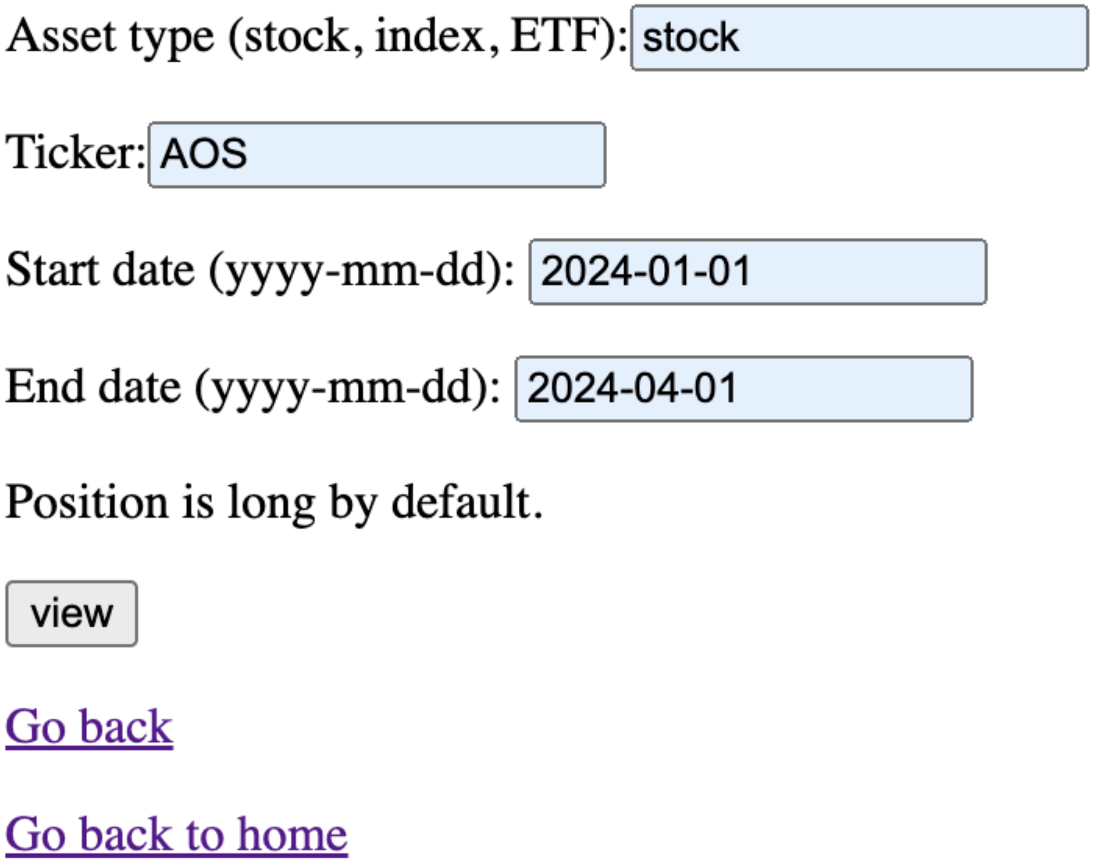
View more stocks   Go back to home

Date	Open	High	Low	Adj Close
2024-05-10	10.02	11.00	8.62	8.94
2024-05-09	4.48	4.56	4.43	4.47
2024-05-08	4.54	4.56	4.43	4.47
2024-05-07	4.94	4.94	4.47	4.61
2024-05-06	4.93	5.05	4.72	4.76
2024-05-03	4.84	4.94	4.69	4.93
2024-05-02	4.69	4.78	4.60	4.71
2024-05-01	4.30	4.80	4.29	4.67
2024-04-30	4.27	4.48	4.25	4.33
2024-04-29	4.14	4.32	4.10	4.29
2024-04-26	3.97	4.13	3.92	4.09
2024-04-25	4.09	4.12	3.90	3.95
2024-04-24	4.26	4.26	4.10	4.15
2024-04-23	4.09	4.36	4.08	4.19
2024-04-22	3.99	4.13	3.91	4.07
2024-04-19	3.89	4.02	3.86	3.97
2024-04-18	3.89	3.99	3.81	3.89
2024-04-17	4.00	4.05	3.88	3.89
2024-04-16	4.07	4.11	3.95	3.99
2024-04-15	4.42	4.43	4.10	4.12
2024-04-12	4.30	4.48	4.24	4.28
2024-04-11	4.34	4.36	4.23	4.30
2024-04-10	4.35	4.35	4.22	4.26
2024-04-09	4.42	4.65	4.39	4.42
2024-04-08	4.42	4.48	4.36	4.43
2024-04-05	4.39	4.54	4.32	4.42
2024-04-04	4.55	4.75	4.44	4.44
2024-04-03	4.49	4.58	4.38	4.54
2024-04-02	4.81	4.81	4.46	4.53

Figure 9: Fetches Financial Data

6.8 Query 8: Creates New Alpha Sessions

Description: Initiates a session to store necessary data with SQL for calculating the alpha of financial instruments using Python and SQL.



Asset type (stock, index, ETF): stock

Ticker: AOS

Start date (yyyy-mm-dd): 2024-01-01

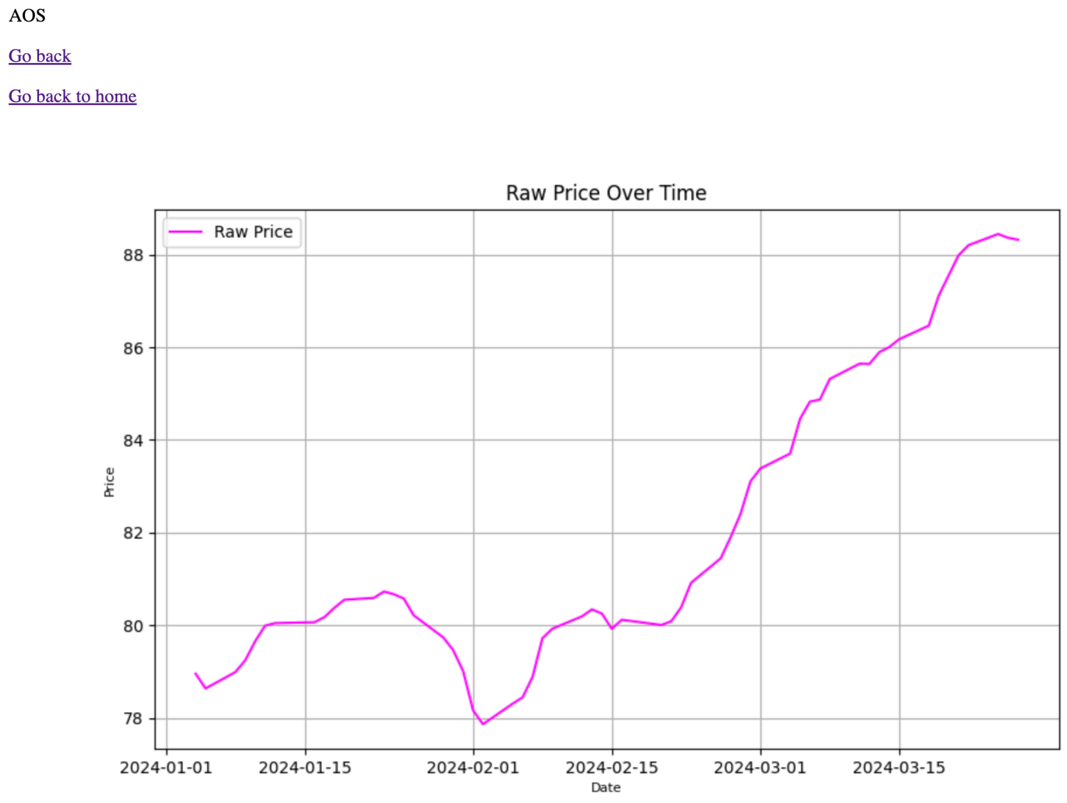
End date (yyyy-mm-dd): 2024-04-01

Position is long by default.

Figure 10: Creates New Alpha Session

6.9 Query 9: Displays Raw Data

Description: Displays raw data graph using Python and SQL.



AOS

Go back

Go back to home

Figure 11: Displays Raw Data

6.10 Query 10: Displays Related Alphas

Description: Displays useful alphas using Python and SQL.

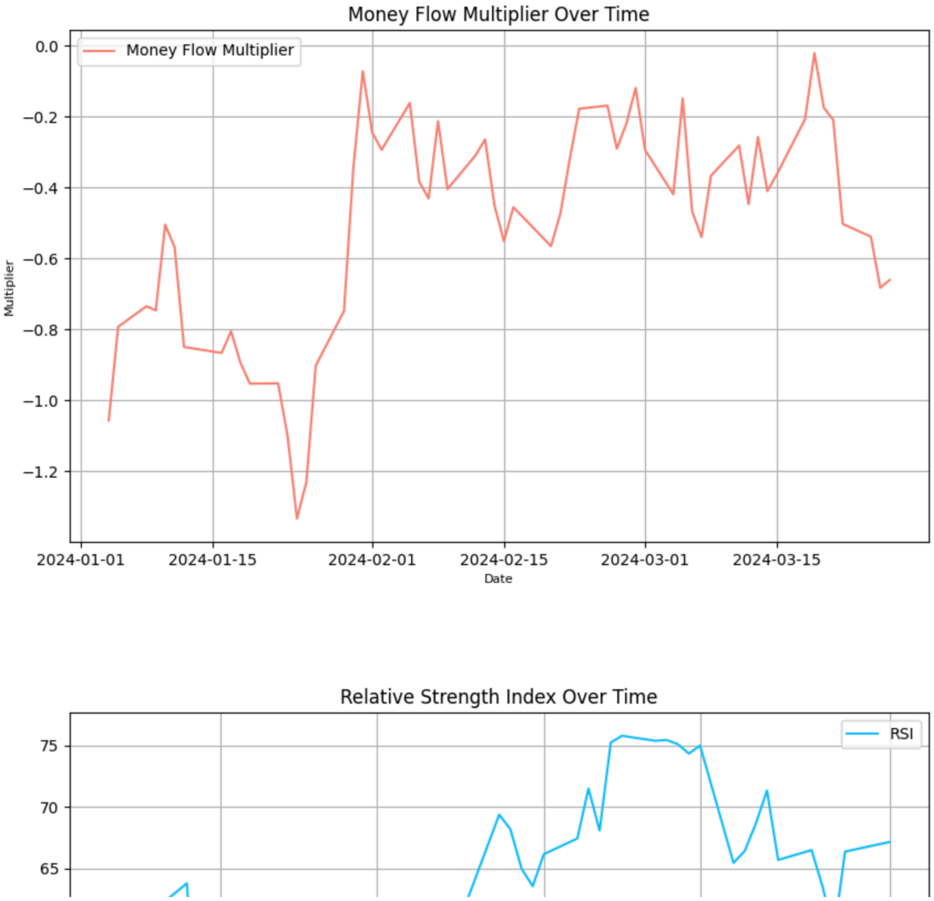


Figure 12: Displays Related Alphas

7 Code

Here is the original Python code (with SQL integrated) for the project.

```python
# app.py
#
# This file contains the main application code for the Trading System.
# It contains the main routes for the application, including the following:
#     1. Home page;
#     2. User sign up;
#     3. User log in;
#     4. User home page;
#     5. Trading session result;
#     6. New trading session;
#     7. Delete trading session;
#     8. Portfolio page;
#     9. Asset home page;
#     10. Asset price main page;
#     11. Stock asset price page;
#     12. Index asset price page;
#     13. ETF asset price page;
#     14. Asset alpha main page;
#     15. Stock asset alpha page;
#     16. Index asset alpha page;
#     17. ETF asset alpha page;
#     18. Portfolio page.
#
# The application is run using the Flask framework and connects to a MySQL database to retrieve and store data.
# The application also uses the utils.py file to process and plot data.


import matplotlib
import mysql.connector
import utils


from flask import Flask
from flask import redirect, render_template, request, url_for
from utils import process_raw_price, plot_price_with_alphas


matplotlib.use("Agg")


# Modifies the parameters to match your local machine.
"""
DB_Connection = mysql.connector.connect(
    host="127.0.0.1",
    user="teammate",
    password="mpcs53001",
    database="TradingSystem"
)
"""


DB_Connection = mysql.connector.connect(
    host="localhost",
    user="root",
    password="mpcs53001",
    database="TradingSystem"
)
cursorObject = DB_Connection.cursor()



app = Flask(__name__)


@app.route("/")
def index():
    return redirect(url_for("home"))


# ------------------------------------------------------------------------------
# This is the home page of the application.
# ------------------------------------------------------------------------------

@app.route("/ts/home/")
def home():
    return render_template("home.html")


# ------------------------------------------------------------------------------
# This is the user sign up page.
# ------------------------------------------------------------------------------

@app.route("/ts/signup", methods=["GET", "POST"])
def signup():
    if request.method == "POST":
        # Extracts the user's information from the form and inserts it into the database.
        fname = request.form["fname"]
        lname = request.form["lname"]
        phone = request.form["phone"]
        email = request.form["email"]
        preferences = request.form["preferences"]

        # Checks if the email already exists in the database.
        cursorObject.execute("SELECT * FROM users WHERE email = %s", (email,))
        existing_user = cursorObject.fetchone()
        if existing_user:
            return "Email already exists. Please use a different email."

        # Inserts the user's information into the database.
        cursorObject.execute(
            "INSERT INTO users (firstName, lastName, phone, email, preferences) "
            "VALUES (%s, %s, %s, %s, %s)",
            (fname, lname, phone, email, preferences),
        )
        DB_Connection.commit()

        return redirect(url_for("signup_success"))
    return render_template("signup.html")


# ------------------------------------------------------------------------------
# This is the user sign up success page.
# ----------------------------------------------------------------------------

@app.route("/ts/signup_success")
def signup_success():
    return render_template("signup_success.html")


# ----------------------------------------------------------------------------
# This is the user login page.
# ----------------------------------------------------------------------------

@app.route("/ts/login", methods=["GET", "POST"])
def login():
    if request.method == "POST":
        # Gets the user’s email from the form and checks if it exists in the database.
        user_email = request.form["email"]
        cursorObject.execute(
            "SELECT * FROM users WHERE email = %s",
            (user_email,)
        )
        user = cursorObject.fetchone()

        #  If the email does not exist, an error message is displayed.
        if not user:
            return render_template(
                "error.html",
                message="Email does not exists. Please sign up for a email."
            )

        # Redirects to user’s home page after login.
        uID = user[0]
        return redirect(url_for("user", user_id=uID))

    return render_template("login.html")


"""
    Extracts the user’s information from the database and displays it on the user’s home page.

    PART I
    1. The trading_sessions.html page will redirect to another route that displays the result of the trading session.
    2. Creates a new route for this which accept a variable endpoint i.e.<xx>, with the variable being the primary key of the Result Table.
    3. Extracts result from Result Table using the primary key and render an html template to display the extracted record.
    4. The result record page contains the following:
        (a) The overal return of the trading session.
        (b) The risk of the trading session (measured by asset price’s standard deviation during that trading period).
        (c) The Sharpe Ratio of the trading session (return divided by std, i.e., step 1 divided by step 2).
        (d) A cumulative return graph plotted against the cumulative return graph of a benchmark.

    PART II
```

```python
    1. The trading_session.html page have a button to accept new trading session created by the user.
    2. The button directs the user to a new route that is similar to the user sign up page which render an html to accept user inputs for the new session.
    3. After user submits, we adds the record into the database table - UserTradingSessions.
    4. Calculates the result of this session and add it to the Result Table.
    5. The asset can be assumed to be an individual asset with long position.
"""

# ------------------------------------------------------------------------------
# This is the user's home page after login, displaying all trading sessions.
# ------------------------------------------------------------------------------

@app.route("/ts/user/<user_id>")
def user(user_id):
    cursorObject.execute("SELECT * FROM users WHERE uID = %s", (user_id,))
    user = cursorObject.fetchone()

    if user:
        cursorObject.execute(
            "SELECT * FROM UserTradingSessions WHERE uID = %s", (user[0],)
        )
        trading_sessions = cursorObject.fetchall()

        return render_template(
            "trading_sessions.html",
            userid=user_id,
            user=user,
            trading_sessions=trading_sessions,
            message=f"Welcome {user[1]} {user[2]}! ",
        )

    return "User not found."


# ------------------------------------------------------------------------------
# This is the trading session result page.
# ------------------------------------------------------------------------------

@app.route("/ts/user/<user_id>/trading_session_result/<session_id>")
def trading_session_result(session_id, user_id):
    cursorObject.execute(
        "SELECT * FROM Result WHERE sessionID = %s AND uID = %s",
        (session_id, user_id)
    )
    result = cursorObject.fetchone()
    if result:
        return render_template(
            "trading_session_result.html",
            result=result,
            overall_return=result[2],
            risk=result[3],
            sharpe_ratio=result[4]
        )

    return "Trading session result not found."


```

```python
# ----------------------------------------------------------------------------
# This is the new trading session page.
# ----------------------------------------------------------------------------

@app.route("/ts/user/<user_id>/new_trading_session", methods=["GET", "POST"])
def new_trading_session(user_id):
    if request.method == "POST":
        # Extracts new trading session information from the form and inserts it into the database.
        preference = request.form["preference"].lower()
        ticker = request.form["ticker"]

        # Uses the preference and ticker to find the underlying ID.
        # Determines the table and column based on preference.
        if preference == "stock":
            table = "Stocks"
            id_column = "sID"
        elif preference == "etf":
            table = "ETFs"
            id_column = "eID"
        elif preference == "index":
            table = "Indices"
            id_column = "iID"
        else:
            return "Invalid preference. Please choose ’stock’, ’etf’, or ’index’."

        # Queries the underlying ID based on ticker.
        cursorObject.execute(
            f"SELECT DISTINCT {id_column} FROM {table} WHERE ticker = %s", (ticker,)
        )
        underlying = cursorObject.fetchone()
        if not underlying:
            return f"No underlying asset found with ticker {ticker} in {preference}."
        underlyingID = f"{table}-{ticker}-{underlying[0]}"

        start_date = request.form["start_date"]
        end_date = request.form["end_date"]
        volume = int(request.form["volume"])

        cursorObject.execute(
            "INSERT INTO UserTradingSessions (uID, startDay, endDay, volume, underlyingID, positionLS) "
            "VALUES (%s, %s,%s,%s,%s,%s)",
            (user_id, start_date, end_date, volume, underlyingID, "L"),
        )
        DB_Connection.commit()

        # Retrieves all trading sessions for the user.
        cursorObject.execute(
            "SELECT sessionID, volume FROM UserTradingSessions WHERE uID = %s",
            (user_id,),
        )
        sessions = cursorObject.fetchall()

        # Calculates the total volume.
        total_volume = sum(session[1] for session in sessions)
        cursorObject.execute(
            "INSERT INTO Portfolio (uID, sessionID, weight) VALUES (%s,%s,%s)",
            (user_id, sessions[-1][0], 0),
        )

        # Updates the weight for each session based on the new total volume.
        for session in sessions:
            session_id = session[0]
            session_volume = session[1]
            weight = session_volume / total_volume

            cursorObject.execute(
                "UPDATE Portfolio SET weight = %s WHERE sessionID = %s",
                (weight, session_id),
            )
        DB_Connection.commit()

        # Calculates results for the new session.
        cursorObject.execute(
            f"SELECT * FROM {table} "
            f"WHERE sID = {underlying[0]} AND day BETWEEN '{start_date}' AND '{end_date}'"
        )
        result = cursorObject.fetchall()

        # ----------------------------------------------------------------------
        # Calculates the overall return, risk, and Sharpe ratio.
        # ----------------------------------------------------------------------

        overall_return = -float(utils.calculate_overall_return(result))
        risk = float(utils.calculate_risk(result))
        sharpe_ratio = (
            round(overall_return / risk, 2) if risk != 0 else 0
        )
        utils.plot_cumulative_return(result)

        cursorObject.execute(
            "INSERT INTO Result (uID, sessionID, gain, risk, SharpeRatio) "
            "VALUES (%s, %s, %s, %s, %s)",
            (user_id, session_id, overall_return, risk, sharpe_ratio),
        )
        DB_Connection.commit()

        return redirect(url_for("user", user_id=user_id))

    stock_tickers = get_unique_tickers("Stocks")
    index_tickers = get_unique_tickers("Indices")
    etf_tickers = get_unique_tickers("ETFs")

    return render_template(
        "new_trading_session.html",
        user_id=user_id,
        stock_tickers=stock_tickers,
        index_tickers=index_tickers,
        etf_tickers=etf_tickers,
        image_names="cumulative_returns.png"
    )


def get_unique_tickers(table_name):
    query = f"SELECT DISTINCT ticker FROM {table_name}"
    cursorObject.execute(query)

    # Fetches all the tickers from the table.
    tickers = [row[0] for row in cursorObject.fetchall()]
    return tickers


# ------------------------------------------------------------------------------
# This is the delete trading session page.
# ------------------------------------------------------------------------------

@app.route("/ts/delete_trading_session/<session_id>/<user_id>", methods=["POST"])
def delete_trading_session(session_id, user_id):
    # Delete the trading session
    delete_id = session_id
    cursorObject.execute("DELETE FROM portfolio WHERE sessionID = %s", (delete_id,))
    cursorObject.execute("DELETE FROM Result WHERE sessionID = %s", (delete_id,))
    cursorObject.execute(
        "DELETE FROM usertradingsessions WHERE sessionID = %s", (delete_id,)
    )
    DB_Connection.commit()

    # Retrieves all trading sessions for the user.
    cursorObject.execute(
        "SELECT sessionID, volume FROM UserTradingSessions WHERE uID = %s", (user_id,)
    )
    sessions = cursorObject.fetchall()

    # Calculates the total volume.
    total_volume = sum(session[1] for session in sessions)

    # Updates the weight for each session based on the new total volume.
    for session in sessions:
        session_id = session[0]
        session_volume = session[1]
        weight = session_volume / total_volume if total_volume > 0 else 0
        cursorObject.execute(
            "UPDATE Portfolio SET weight = %s WHERE sessionID = %s",
            (weight, session_id),
        )
    DB_Connection.commit()

    return redirect(url_for("user", user_id=user_id))


# ------------------------------------------------------------------------------
# This is the portfolio page.
# ------------------------------------------------------------------------------

@app.route("/portfolio/<int:user_id>")
def portfolio(user_id):
    cursorObject.execute("SELECT * FROM Users WHERE uID = %s", (user_id,))
    user = cursorObject.fetchone()

    cursorObject.execute(
        "SELECT sessionID, weight FROM Portfolio WHERE uID = %s", (user_id,)
    )
    portfolio = cursorObject.fetchall()

    return render_template(
        "portfolio.html", user=user, user_id=user_id, portfolio=portfolio
    )


# -----------------------------------------------------------------------------
# This is the asset home page.
# -----------------------------------------------------------------------------

@app.route("/ts/asset")
def asset():
    return render_template("asset.html")


# -----------------------------------------------------------------------------
# This is the asset price main page.
# -----------------------------------------------------------------------------

@app.route("/ts/asset/price")
def price():
    return render_template("price.html")


# -----------------------------------------------------------------------------
# This is the stock asset price page.
# -----------------------------------------------------------------------------

@app.route("/ts/asset/price/stocks")
def stocks():
    # Extracts all stock tickers from database to display.
    cursorObject.execute("SELECT DISTINCT ticker FROM Stocks;")
    stock_tickers = [item[0] for item in cursorObject.fetchall()]

    return render_template("stocks.html", tickers=stock_tickers)


# -----------------------------------------------------------------------------
# This is the individual stock’s all price data page.
# -----------------------------------------------------------------------------

@app.route("/ts/asset/price/stocks/<ticker>")
def stock_ticker_data(ticker):
    # Extracts all price data of this stock ticker from database and display it.
    query = (
        "SELECT day, open, high, low, close FROM Stocks WHERE ticker=’"
        + ticker
        + "’ ORDER BY day DESC;"
    )
    cursorObject.execute(query)

    price_data_raw = cursorObject.fetchall()
    price_data = [process_raw_price(item) for item in price_data_raw]

    return render_template("stock_ticker.html", tick=ticker, data=price_data)

```

```python
# -----------------------------------------------------------------------------
# This is the index asset price page.
# -----------------------------------------------------------------------------

@app.route("/ts/asset/price/indices")
def indices():
    # Extracts all index tickers from database to display.
    cursorObject.execute("SELECT DISTINCT ticker FROM Indices;")
    index_tickers = [item[0] for item in cursorObject.fetchall()]

    return render_template("indices.html", tickers=index_tickers)


# -----------------------------------------------------------------------------
# This is the individual index's all price data page.
# -----------------------------------------------------------------------------

@app.route("/ts/asset/price/indices/<ticker>")
def index_ticker_data(ticker):
    query = (
        "SELECT day, open, high, low, close FROM Indices WHERE ticker='"
        + ticker
        + "' ORDER BY day DESC;"
    )
    cursorObject.execute(query)

    price_data_raw = cursorObject.fetchall()
    price_data = [process_raw_price(item) for item in price_data_raw]

    return render_template("index_ticker.html", tick=ticker, data=price_data)

# -----------------------------------------------------------------------------
# This is the ETF asset price page.
# -----------------------------------------------------------------------------

@app.route("/ts/asset/price/etfs")
def etfs():
    # Extracts all ETF tickers from database to display.
    cursorObject.execute("SELECT DISTINCT ticker FROM ETFs;")
    etf_tickers = [item[0] for item in cursorObject.fetchall()]

    return render_template("etfs.html", tickers=etf_tickers)


# -----------------------------------------------------------------------------
# This is the individual ETF's all price data page.
# -----------------------------------------------------------------------------

@app.route("/ts/asset/price/etfs/<ticker>")
def etf_ticker_data(ticker):
    # Extracts all price data of this ETF ticker from database and display it.
    query = (
        "SELECT day, open, high, low, close FROM ETFs WHERE ticker='"
        + ticker
        + "' ORDER BY day DESC;"
    )
    cursorObject.execute(query)

    price_data_raw = cursorObject.fetchall()
    price_data = [process_raw_price(item) for item in price_data_raw]

    return render_template("etf_ticker.html", tick=ticker, data=price_data)


# -------------------------------------------------------------------------------
# This is the asset alpha main page.
# -------------------------------------------------------------------------------

@app.route("/ts/asset/alpha", methods=["GET", "POST"])
def alpha():
    if request.method == "POST":
        # Extracts asset info from the html request to plot alphas on.
        asset_type = request.form["assettype"]
        ticker = request.form["ticker"]
        start_date = request.form["startdate"]
        end_date = request.form["enddate"]

        # Queries the database to get asset price data.
        if asset_type == "stock":
            DB_Table = "Stocks"
        elif asset_type == "index":
            DB_Table = "Indices"
        elif asset_type == "ETF":
            DB_Table = "ETFs"
        else:
            # Error handling and redirects to alpha.html.
            render_template("alpha.html")

        # The error handling for the ticker.
        ticker_query = "SELECT DISTINCT ticker FROM " + DB_Table + ";"
        cursorObject.execute(ticker_query)
        tickers = [item[0] for item in cursorObject.fetchall()]
        if ticker not in tickers:
            # Error handling and redirects to alpha.html.
            render_template("alpha.html")

        # Gets price data.
        data_query = (
            "SELECT day, open, high, low, close FROM "
            + DB_Table
            + " WHERE ticker='"
            + ticker
            + "' AND day>='"
            + start_date
            + "' AND day<='"
            + end_date
            + "' ORDER BY day ASC;"
        )
        cursorObject.execute(data_query)
        price_data = [process_raw_price(row) for row in cursorObject.fetchall()]

        # Plots alphas using asset price, plot asset price as well.
        image_names = plot_price_with_alphas(price_data)

        return render_template(
            "view_with_alphas.html", tick=ticker, image_names=image_names
        )

    return render_template("alpha.html")


app.run(host="0.0.0.0", port=5001, debug=True)
```

Listing 3: app.py