**Final Project Group 24**

May 2023

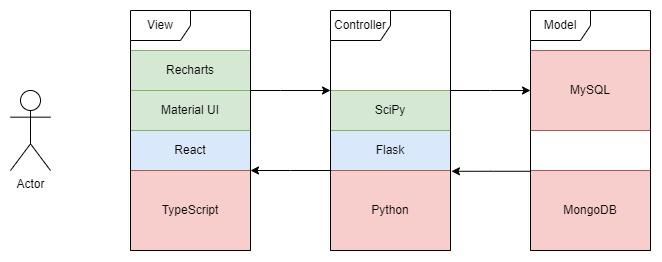
Name of Application: “Stocker”

| [Mikaela Spaventa](mailto:miksp@seas.upenn.edu) | [Natalie Gilbert](mailto:natgil@seas.upenn.edu) | [Grace Benner](mailto:gebenner@seas.upenn.edu) | [Eric Jackson](mailto:ebj29@sas.upenn.edu) |
| --- | --- | --- | --- |

# **Introduction**

We designed and built an application allowing users to evaluate how stock portfolios would have performed on historical data. Drawing inspiration from Yahoo! Finance and Robinhood, our app incorporates news headlines into each page. Our app features pages dedicated to each stock in the S&P 500, and a portfolio page where users can visualize their portfolio’s performance from 2010 to 2023 and view a day-by-day breakdown of what happened.

# **Architecture**

In this section, we briefly describe our application’s architecture. The diagram below summarizes this information.

## MySQL

The majority of our application’s data is stored in a MySQL database. As described later, this database includes tables for companies, historical market data, and historical news headlines.

## MongoDB

Our application stores user portfolios—a collection of ticker-quantity pairs. There are many stocks a user might own, of which they likely own only a small subset. Given these conditions, we determined that MongoDB’s flexible structure would be better suited for storing this information since portfolios could be easily stored as an array of documents. Thus, we decided to store user data in a MongoDB instance.

## React

We used React and Typescript to build our application’s user interface.

## Flask

The back-end of our application is built in python using the Flask package. We made this somewhat unusual decision because we wanted to use the SciPy optimizer to handle one of the routes.

# **Data**

**DB Name:** Stocks\_And\_News

**Number of Tables:** 5

**Note:** \* indicates primary key

\*\* indicates foreign key

**Daily Stocks**

**Table Name:** Daily\_Stocks

**Source:** <https://finance.yahoo.com/>(we used the yfinance library in Python to obtain this data)

**Description:** Contains detailed historical information on the stocks in the S&P500

**Date Range:** 1/2/1962 - 4/3/2023

**Cardinality:** 4,151,306

**Degree:** 10

**Schema:**

| **Column Name** | **Data Type** | **Description** | **Number of Distinct Values** | **% Missing Values** | **Mean** | **Range** |
| --- | --- | --- | --- | --- | --- | --- |
| date\* | *date* | Historical date | 15,419 | 0 | N/A | 1/2/1962 - 4/3/2023 |
| open | *float* | Opening share price | 393,288 | 0 | 49.90 | **MIN:** 0  **MAX:** 5977.61 |
| high | *float* | Highest price the share reached that day | 412,903 | 0 | 50.82 | **MIN:** 0.00651  **MAX:** 5982.45 |
| low | *float* | Lowest price the share reached that day | 410,256 | 0 | 49.62 | **MIN:** 0.006185  **MAX:** 5884.06 |
| close | *float* | Closing share price | 420,598 | 0 | 50.24 | **MIN:** 0.00651  **MAX:** 5959.33 |
| adj\_close | *float* | Closing price after adjustments for all splits and dividend distributions | 2,672,027 | 0 | 42.73 | **MIN:** 0.00178319  **MAX:** 5959.33 |
| volume | *int* | Number of shares traded | 786,063 | 0 | 5,197,209.80 | **MIN:** 0  **MAX:** 214,748,3647 |
| dividends | *float* | Distribution of company’s earnings to shareholders | 3,330 | 0 | 0.003 | **MIN:** 0  **MAX:** 103.75 |
| stock\_splits | *int* | Increase in number of shares | 9 | 0 | 0.0007 | **MIN:** 0  **MAX:** 20 |
| ticker\*,\*\*  (FK references Companies(ticker)) | *varchar(5)* | Abbreviation identifying a stock | 500 | 0 | N/A | N/A |

**Companies**

**Table Name:** Companies

**Source:** <https://en.wikipedia.org/wiki/List_of_S%26P_500_companies>

(see project\_data.ipynb in the Github too see how we webscraped this)

**Description:** Provides descriptive information on the companies that are public

**Cardinality:** 503

**Degree:** 7

**Schema:**

| **Column Name** | **Data Type** | **Description** | **Number of Distinct Values** | **% Missing Values** | **Mean** | **Range** |
| --- | --- | --- | --- | --- | --- | --- |
| ticker\* | *varchar(6)* | Abbreviation identifying a stock | 503 | 0 | N/A | N/A |
| avg\_volume | *int* | Number of shares | 503 | 0 | 4,908,673.40 | **MIN:** 20725  **MAX:** 155,914,942 |
| competitors | *text* | Stock’s competitors | 478 | 0 | N/A | N/A |
| security | *text* | Name of company | 503 | 0 | N/A | N/A |
| headquarters | *text* | Where company is based | 252 | 0 | N/A | N/A |
| industry | *text* | What type of industry company is in | 123 | 0 | N/A | N/A |
| date\_added | *date* | When became part of S&P 500 | 364 | 0 | N/A | 1957-03-04  -  2023-03-15 |

**Current News**

**Table Name:** Current\_News

**Source:** <https://newsapi.org/> (see Git for how we scraped the data)

**Description:** Contains info regarding relatively recent headlines associated with different stocks

**Date Range:** 3/2/2023-4/1/2023

**Cardinality:** 151

**Degree:** 6

**Schema:**

| **Column Name** | **Data Type** | **Description** | **Number of Distinct Values** | **% Missing Values** | **Mean** | **Range** |
| --- | --- | --- | --- | --- | --- | --- |
| ticker\*\* (FK references Companies(ticker)) | *varchar(5)* | An abbreviation identifying a stock | 116 | 0 | N/A | N/A |
| source | *varchar(32)* | Where the headline was featured | 151 | 0 | N/A | N/A |
| author | *text* | The name of the person who wrote the article | 135 | 0 | N/A | N/A |
| title\* | *text* | The name of the headline | 151 | 0 | N/A | N/A |
| url | *text* | A link to the article | 151 | 0 | N/A | N/A |
| date\* | *datetime* | The date the article was published | 151 | 0 | N/A | 3/2/2023-4/1/2023 |

**Historic News**

**Table Name:** Historic\_News

**Source:** <https://www.kaggle.com/datasets/mynameiscorn/news-headlines-youtube-metadata>

**Description:** Contains info regarding historical headlines associated with different stocks

**Date Range:** 1/28/2017 - 11/19/2021

**Cardinality:** 78,195

**Degree:** 3

**Schema:**

| **Column Name** | **Data Type** | **Description** | **Number of Distinct Values** | **% Missing Values** | **Mean** | **Range** |
| --- | --- | --- | --- | --- | --- | --- |
| source\* | *varchar(4)* | Where the headline was featured | 4 | 0 | N/A | N/A |
| date\* | *date* | The date the headline was published | 1757 | 0 | N/A | 1/28/2017 - 11/19/2021 |
| title\* | *varchar(128)* | The name of the headline | 77,028 | 0 | N/A | N/A |

**User**

**Table Name:** User

**Source:** Created by the user in the web app’s homepage and received via an API call.

**Description:** Stores all usernames and passwords of the web app’s users

**Cardinality:** 2

**Degree:** 2

**Schema:**

| **Column Name** | **Data Type** | **Description** | **Number of Distinct Values** | **% Missing Values** | **Mean** | **Range** |
| --- | --- | --- | --- | --- | --- | --- |
| username\* | *varchar(32)* | User’s username | 2 | 0 | N/A | N/A |
| password | *varchar(32)* | User’s password | 2 | 0 | N/A | N/A |

# **Database**

We used a MySQL database to store our Stocks\_and\_News database. In terms of entity resolution efforts, we did not have to do much with either of the news relations since it was already separate from the stock relation. However, we did make sure that in our relation Daily\_Stocks only had attributes that were uniquely related to the primary key of (ticker, date). To make this possible, we had attributes that were unique to each ticker only placed into a separate relation Companies. The attributes in Companies only rely on the primary key of ticker, and not on any date, hence we needed this additional table to get the DB in BCNF.

## Statistics

|  | **Daily\_Stocks** | **Companies** | **Current\_News** | **Historic\_News** |
| --- | --- | --- | --- | --- |
| Number of Instances | 4,151,306 | 503 | 2,456 | 78,195 |

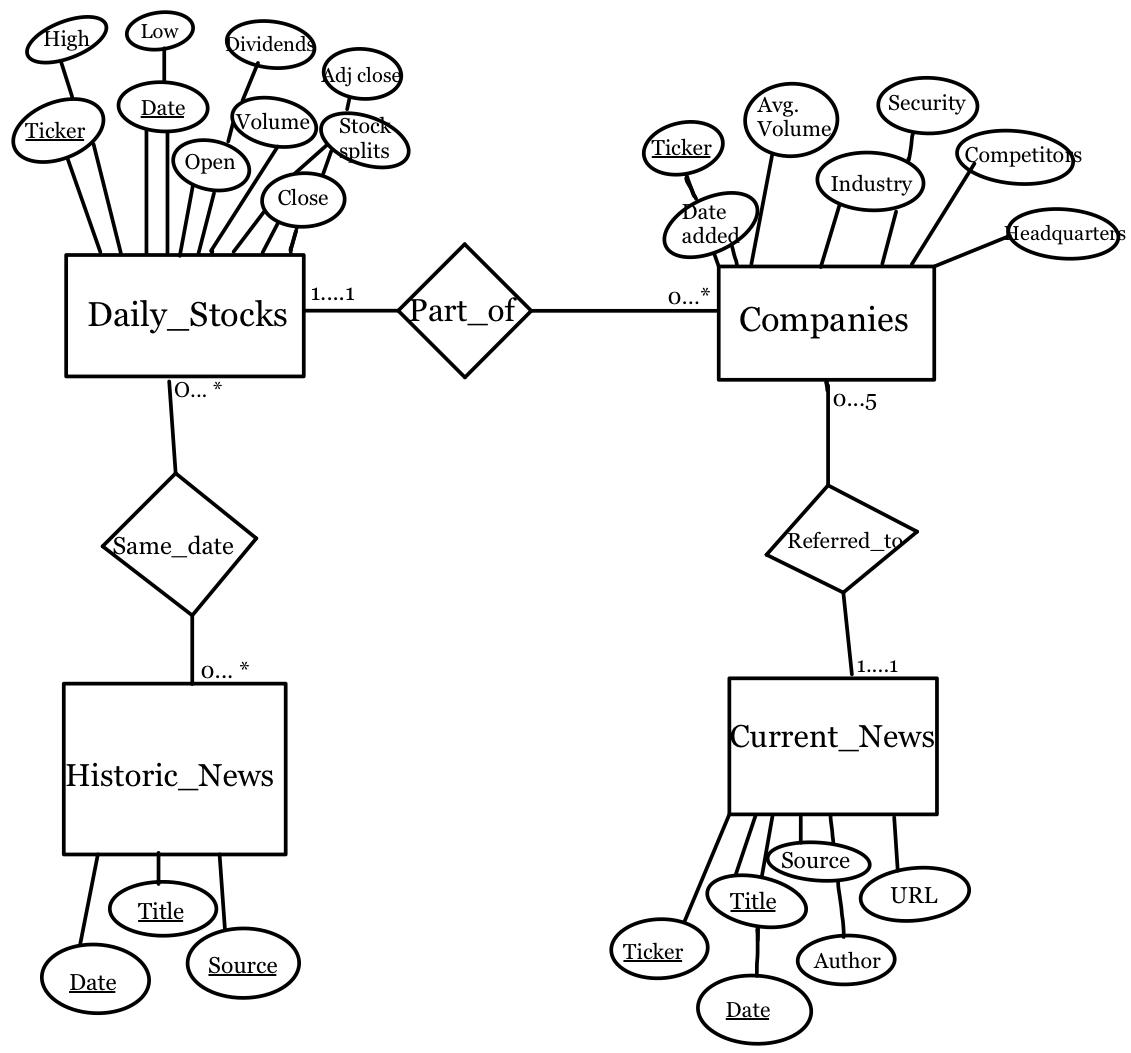
## Normal Form

In our project, we have the functional dependencies per relation as:

* FDaily\_Stocks = {Ticker, Date → High, Low, Dividends, Adj\_Close, Stock\_Splits, Volume, Open Close}
* FCompanies = {Ticker → Avg\_Volume, Date\_Added, Security, Competitors, Industry, Headquarters}
* FHistoric\_News = {Title, Date, Source → Title, Date, Source} (trivial)
* FCurrent\_News = {Ticker, Title, Date → Source, URL, Author}

Given this breakdown, we can justify that our database uses BCNF. Since, for every X → A that holds in any given relation Ri we see that X is a superkey for Ri.

## ER Diagram



# **Web Application**

In this section, we describe the five routes handled by our application.

## /portfolio

This page is the heart of our application. It displays a chart visualizing the performance of a user’s portfolio and a table showing the performance of each security within the portfolio. This page also includes a list of recommended stocks, and relevant news information. Finally, users can also update the contents of their portfolio on this page.

## /security/:ticker

These pages show the performance of a particular security and display relevant company information, including industry and competitors. These pages also display headlines trending on days of interest for this security (days with the greatest gains or losses and day with maximum price).

## /login

This page allows users to login to our application so they can view their portfolio.

## /register

This page allows a user to create an account so they can log in.

## /password-reset

This page allows a user to reset their password in case they forget it.

# **API Specification**

To see more details on the routes in our backend API that our app executes, click [here](https://docs.google.com/document/d/1quyxcLnj9dMvSmFJMmkSWZXuJMHO8CzFtLSyz5Pfvow/edit?usp=sharing) or see Appendix.

# **Queries**

In this section, we describe five of the queries used in our application.

## Query 1

As users view their portfolio’s performance on a given day, they may wonder how well the best stock’s performed on that particular day. The aim of this query is to answer that question by returning the ticker and percent change of the five stocks experiencing the greatest percent change on a given day.

WITH T AS (SELECT *DENSE\_RANK*() OVER (ORDER BY date) AS n, ticker, close, date

FROM Daily\_Stocks

WHERE date >= '<date> - 1 week'

AND date <= '<date> + 1 week')

SELECT d2.ticker, (d2.close - d1.close) / d1.close AS percent\_change

FROM T d1

JOIN T d2 ON (d1.ticker = d2.ticker AND d1.n + 1 = d2.n)

WHERE d2.date = '<date>'

ORDER BY percent\_change DESC

LIMIT 5;

## Query 2 (Complex)

As users view their portfolio, they may wonder how to best allocate their funds among the stocks they have selected. In order to answer this question, we supply the allocation that maximizes their portfolio’s Sharpe ratio. One of the inputs to this calculation is the average daily return of each stock in the portfolio. This query provides that information.

In our application, this query is constructed based on the contents of a user’s portfolio. For concreteness, the code below assumes there are two stocks in the portfolio: AMZN and F. In general if there are stocks in a user’s portfolio, this query will involve joins.

WITH T AS (SELECT *ROW\_NUMBER*() OVER () AS n, amzn, f

FROM (SELECT date, adj\_close AS amzn FROM Daily\_Stocks WHERE ticker = 'AMZN') AS amzn

NATURAL JOIN

(SELECT date, adj\_close AS f FROM Daily\_Stocks WHERE ticker = 'F') AS f

WHERE date >= '<date1>'

AND date <= '<date2>')

SELECT *AVG*((t2.amzn - t1.amzn) / t1.amzn) AS amzn, *AVG*((t2.f - t1.f) / t1.f) AS f

FROM T t1 JOIN T t2 ON t1.n + 1 = t2.n;

## Query 3 (Complex)

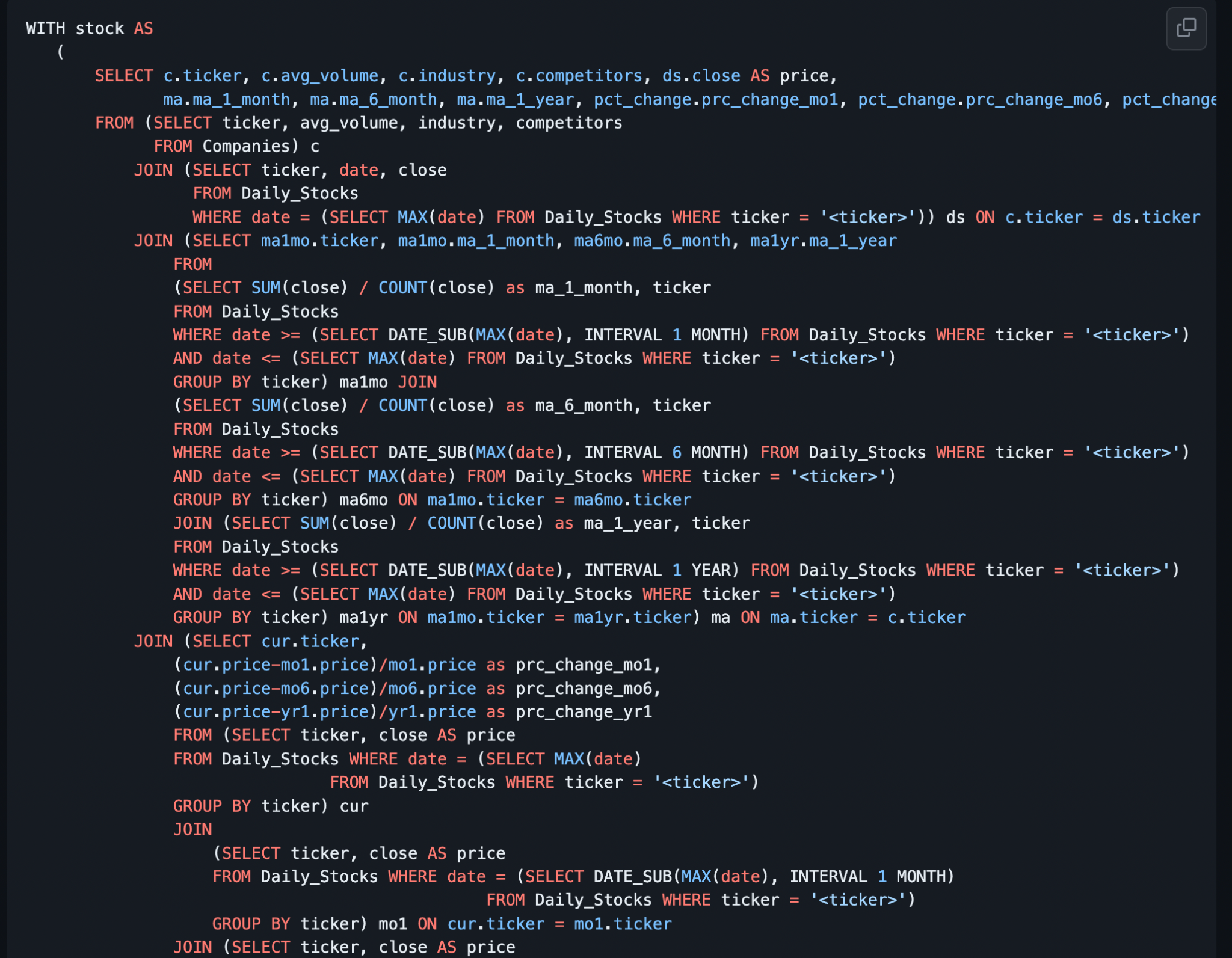
As users manage their portfolio, they may be interested in knowing what other stocks might be a good addition to their portfolio. We’ve therefore created a feature that displays a list of the top recommended stocks to invest in based on a stock that they currently invest in. Therefore, if a user is happy with the performance of one stock, they can check out the 5 most similar stocks to it.

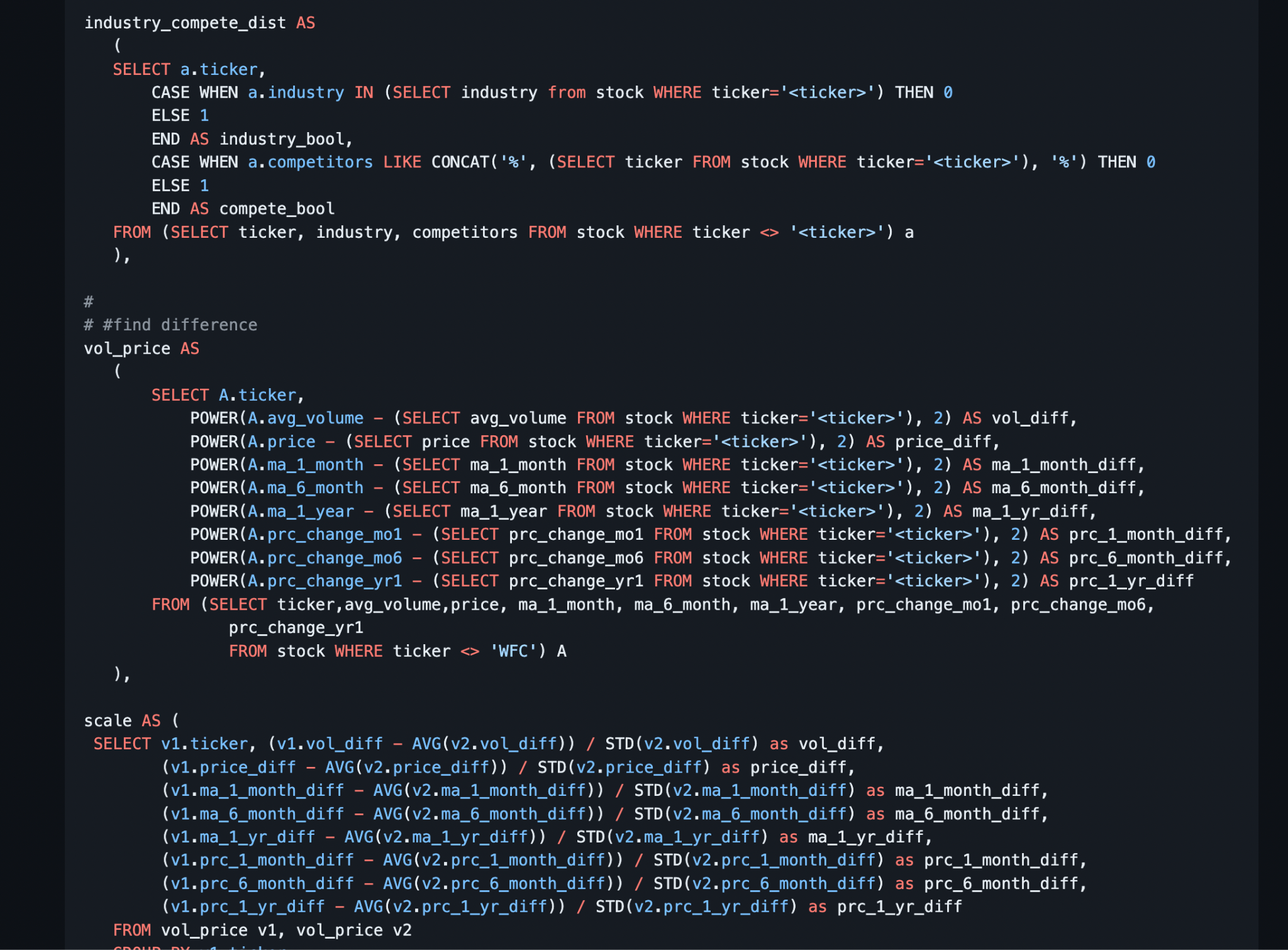
The recommendations are made based on the following parameters:

1. Whether or not a company is in the same industry as another (if a given stock and another stock are in the same industry as each other, return 0 and 1 otherwise)
2. Whether or not a company is a competitor of another (if a given stock and another stock are competitors of one another, return 0 and 1 otherwise)
3. The squared difference between the latest prices are of companies
4. The squared difference between the average volumes of the companies (as of last quarter, this data was web scraped)
5. The squared difference between the moving averages for 1 month, 6 months, and 1 year (this smooths out the price instead of looking at short term fluctuations)
6. The squared difference price percent changes are for 1 month, 6 months, and 1 year (this allows us to look at companies with similar growth rates over the time periods specified)

From here, the data is normalized by column and then these values are added together to get the total distance between a particular stock in a user’s portfolio and all the other stocks in the database. The other stocks in the database are sorted in ascending order and the 5 most similar are returns

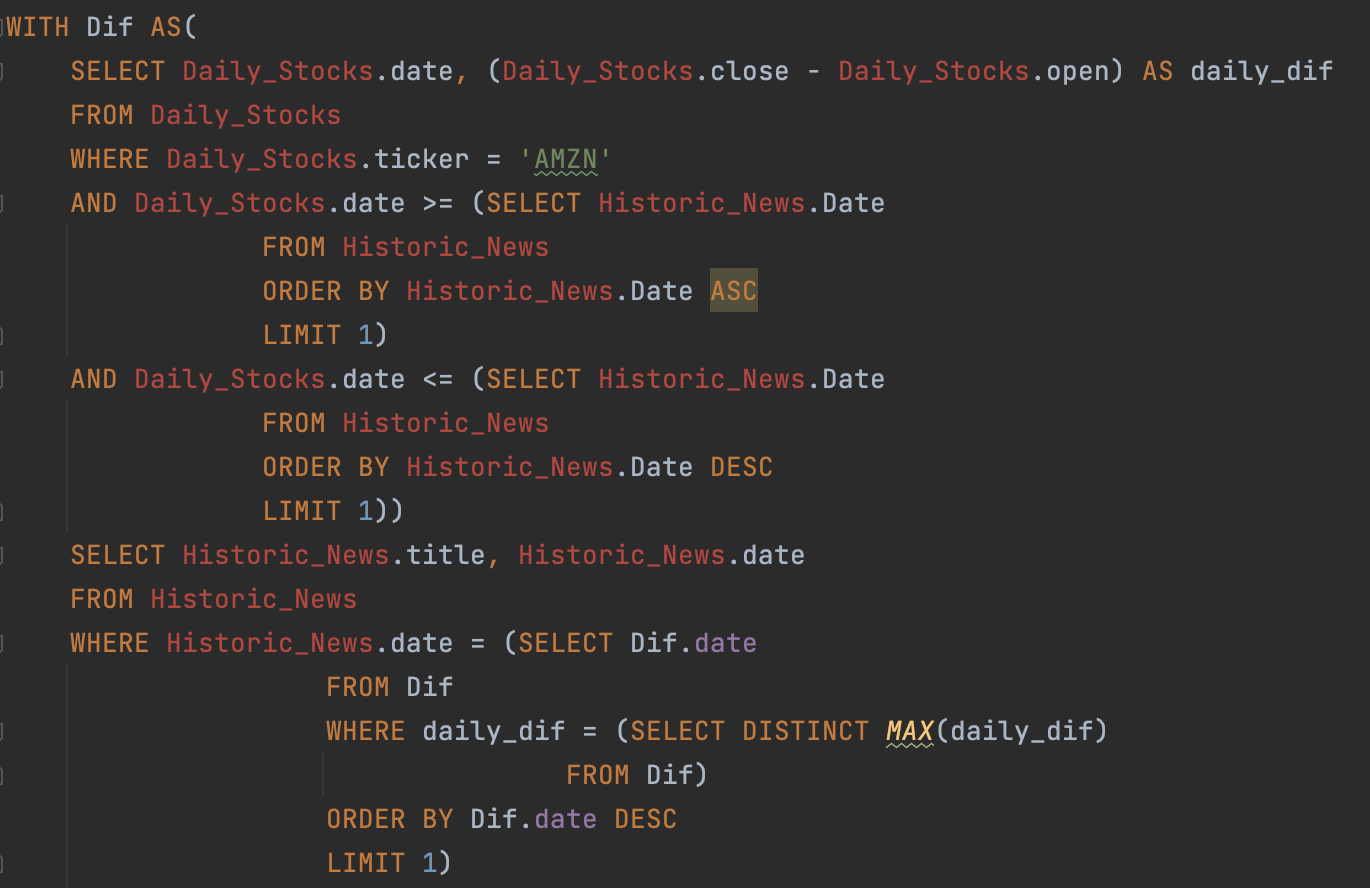
This is query #1 in queries.md of the Github. Please check Git to see the entire query. This is just part of it.





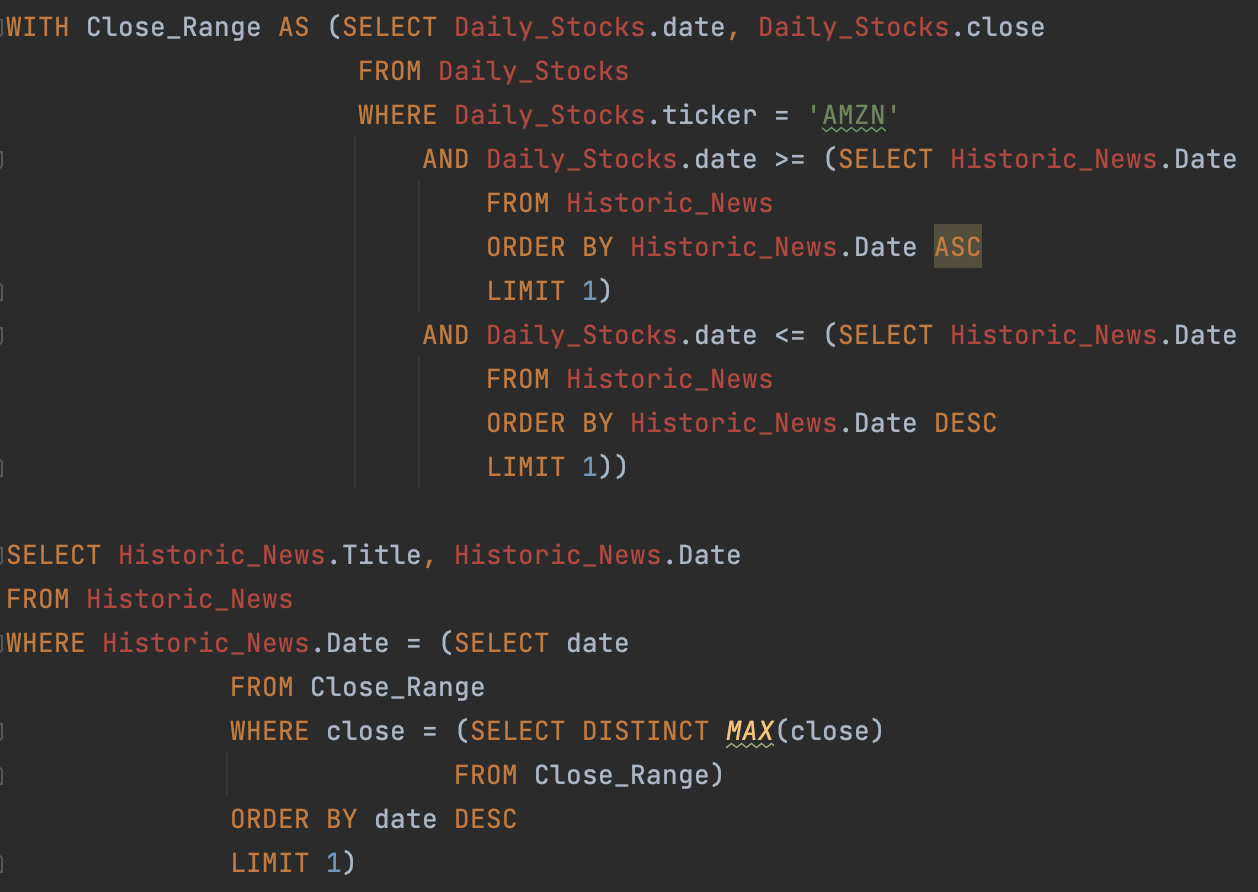
## Query 4 (Complex)

As users explore different companies that they’re invested in, they may want some real-world context to understand why a particular company encountered their biggest gain or loss. One way that we thought that we could provide this was by displaying historical news articles related to the day with the largest positive (MAX) or negative difference (MIN) in the opening and closing values, and if there are multiple days with the same difference, then the system would report the most recent.



## Query 5

The final query we decided to highlight was another news query. This one investigates news on the date that a given stock had the largest (MAX) closing value (was also used to find the lowest closing value using MIN). This is filtered on the range of our historical news data, since it does not cover the entire history of our daily stock values. It is fascinating for a user to see what events may have impacted their stock in such a positive and negative way. We return both the title of the headline and the date.



# **Performance Evaluation**

In this section, we describe how we optimized some of the queries described above. We also compare the original performance of these queries with their post-optimization performance.

## Optimization 1

The tricky part with this query (Query 1 above) is calculating the percent change—particularly because there are not entries for every date (there is no trading on weekends). Originally, we solved this problem by joining the Daily\_Stocks table with itself using the ticker as the join key. Then, we checked for each pair x, y whether there was a z in Daily\_Stocks such that x.date < z.date < y.date. While this strategy nicely captures the notion of x and y being adjacent in a strict linear order, it is also very inefficient because of the Exists clause.

As the table below indicates, the original query was unusable—no user would wait that long for this information to load. To optimize this query, we focused on eliminating the cross products. To do this, we used a SQL function called DENSE\_RANK which we did not discuss in class. This function basically allows us to assign each date in our dataset a consecutive integer value. Using this, we could enforce that percent change is computed between consecutive days inside the join condition.

One additional optimization we made was by restricting early on in the calculation the date range we are considering. Since our database contains data for many years and we only need data for a few days, doing this early on allows the query to run even faster.

### Time Comparison

This chart compares the execution time of the original query with the optimized query when 2009-01-02 is assigned to date.

|  | Execution Time |
| --- | --- |
| Before Optimization | > 20 min |
| After Optimization | 120 ms |

## Optimization 2

To optimize Query 4 above, we used many of the techniques taught in class. The main change that helped a lot was pushing many of the selections and projections down into the CTE, so we did not have to read Daily\_Stocks twice, which was our largest relation. We were also able to then not have any joins or cross products at all, since all the selections were done in the CTE, which dramatically improved performance.

### Time Comparison

This chart compares the execution time of the original query with the optimized query when ‘AMZN’ is assigned to ticker and ‘MAX’ is assigned after calculating the difference between the opening and closing values for each day.

|  | Execution Time |
| --- | --- |
| Before Optimization | 20 s |
| After Optimization | 800 ms |

## Optimization 3

To optimize Query 3 (the optimized version is shown above), we removed the single\_stock table that was used to compare all other stocks against it. Instead, we did the calculations once in a table called stock and used subqueries to filter a particular stock from the rest of the table when finding the distance between it and the other stocks. Then, we also optimized the scale table by reading vol\_price twice instead of each time we needed to find the average and standard deviation of a column.

### Time Comparison

This chart compares the execution time of the original query with the optimized query.

|  | Execution Time |
| --- | --- |
| Before Optimization | 25-40s+ |
| After Optimization | 5-7s |

## Optimization 4

One of the queries we attempted to optimize was query #8 in queries.md in our Github. Although this query was already fairly short, we wanted to see if we could shorten it even more. The first thing we did was get rid of the NATURAL JOIN and instead use a JOIN … ON to get a more distinctive mapping between columns in Companies and Daily\_Stocks. Also, we tried filtering the tuples we needed by pushing down the SELECT and WHERE clauses in Companies and Daily\_Stocks prior to joining them to minimize the amount of tuples used in the rest of the query. Daily\_Stocks has more than a million rows so we thought this would make a difference. However, we hardly saw a decrease in execution time.

### Time Comparison

This chart compares the execution time of the original query with the optimized query when we pushed down our filtering on the date and ticker.

|  | Execution Time |
| --- | --- |
| Before Optimization | 350ms |
| After Optimization | 286ms |

# **Technical Challenges**

In this final section, we describe a few technical challenges we encountered, discuss how we overcame them, and comment on what we learned from them.

## User Portfolio’s

We did not set out to use NoSQL in this project. In fact, our original idea was to store everything in a relational database. As we thought about how to model a user’s portfolio, however, it soon became clear that doing this in a relational database would be rather cumbersome. To overcome this limitation, we turned to MongoDB which allowed us to model this relation in a more natural way. In doing so, we learned firsthand about the factors that lead engineers to choose one database system over another.

## Web Scraping

We had some issues finding good data sources for the news. Many APIs to scrape news data cost a lot of money or were very limited in scope. We thought we had found a really excellent way to scrape the news data, however we soon realized it did not have a way to obtain the date of the article, a key attribute we needed for the queries we wanted. For the newsapi.org that we did end up using, we had to create multiple keys to use and could only get data for the past month. In terms of getting historical data, we had to settle on a smaller time frame than desired, and we also went with news YouTube headlines, rather than online articles. Ideally, with more money and computational resources, we would have one large relation of news data spanning from current date back many years, rather than two different news datasets.

## MySQL Data Type Constraints

One unforeseen challenge that we encountered when creating some of our complex queries were when it came to processing columns that were strings of arrays (e.g. ['QCOM', 'MSI', 'NOK']). We were unaware that MySQL is one of the few databases that doesn’t support list/array datatypes, so we had to use heavy string comprehension in order to isolate and use each element of the array. In the future, it might be better to consider using a database that is more flexible with the datatypes that it supports (e.g. Postgres).

# **Extra Credit**

## NoSQL

## We used NoSQL (MongoDB) to store a user’s portfolio.

## >80% Unit Testing on Frontend

We did unit testing on the entire frontend of our application. Tests are in Git.

## User Login Experience

We were able to add the functionality for a user to login. This was also a way for us to save their portfolio, so they could log back in and not have to add their stocks each time.

# 

# **Appendix**

Full API Specification

| **Number** | **Route** | **Route Parameters** | **Return Type** | **Description** |
| --- | --- | --- | --- | --- |
| **Authentication Routes** | | | | |
| 1 | POST /login | None | token (string) | Returns a JWT if there is a user in the database with this username and password |
| 2 | POST /register | None | None | Adds a user to the database if there is currently no user with the given username. |
| 3 | PUT /reset-password | None | None | Updates the password of the user with the given username (assuming such a user exists). |
| **Security Routes** | | | | |
| 4 | GET /security/:ticker | ticker (string) | {  ticker: string,  avg\_volume: number,  competitors: string,  security: string,  headquarters: string,  industry: string,  Date\_added: string  } (JSON Object) | Returns company information. |
| 5 | GET /history/:ticker | ticker (string) | {  date: string,  adj\_close: number  }[] (JSON Array) | Returns an array of adjusted close values ranging from January 2010 (or the stock’s inception date if its IPO was later than this) to April 2023. |
| 6 | GET /headlines/:ticker | ticker (string) | {  date: string,  headline: string  }[] (JSON Array) | Returns an array of up to five headlines from the day(s) the stock experienced the greatest change in price. |
| 7 | GET /headlines/close/high/:ticker | ticker (string) | {  date: string,  headline: string  }[] (JSON Array) | Returns an array of up to five headlines from the day(s) the stock experienced the highest closing value. |
| 8 | GET /headlines/close/low/:ticker | ticker (string) | {  date: string,  headline: string  }[] (JSON Array) | Returns an array of up to five headlines from the day(s) the stock experienced the lowest closing value. |
| 9 | GET /headlines/new/:ticker | ticker (string) | {  date: string,  headline: string  }[] (JSON Array) | Returns an array of up to three headlines recently associated with the given stock. |
| **Portfolio Routes** | | | | |
| 10 | GET /portfolio | None | {  ticker: string,  quantity: number  }[] (JSON Array) | Returns a user’s portfolio—a list of stocks they own. (This will be used as part of the “update portfolio” flow) |
| 11 | PUT /portfolio | None | None | Updates a user’s portfolio. (This is also part of the “update portfolio” flow) |
| 12 | GET /portfolio/history | None | {  date: string,  value: number  }[] (JSON Array) | Returns the historical performance of a user’s portfolio as an array of date-value pairs. |
| 13 | GET /portfolio/information | None | {  ticker: string,  quantity: number,  allocation: number,  recommended: number,  original\_price: number,  current\_price: number,  percent\_change: number  }[] (JSON Array) | Returns portfolio information for a user’s portfolio. |
| 14 | GET /recommended\_stocks/:ticker | Ticker (string) | {  ticker: string  }[] (JSON Array) | Returns a list of recommended stocks based on a user’s portfolio. |
| 15 | GET /greatest\_gain\_loss | None | {  ticker: string  percent\_change: number  }[] (JSON Array) | Returns a list of five stocks experiencing the greatest magnitude percent change on a given day. |
| 16 | GET /forecasted\_price/:ticker | Ticker (string) | {  price: number  } | Returns the price forecast of a stock |
| 17 | GET /portfolio/headlines/:tickers | Tickers (array of strings) | {  date: string,  ticker: string,  headline: string  }[] (JSON Array) | Returns an array of up to five headlines recently associated with the stock(s) in the user’s portfolio |