**S&P500 Tracker**

Cheng Chen

Ruikang Liu

Puran Zhang

Wenting Zhao

**Introduction and Project Goals**

Nowadays we have seen plenty of stock trading/research apps/sites around the internet but there is no place for us to keep track of only stocks in SP500, the most important index of large cap growth stocks, and dive deep into the sectors and stocks in it. Our web app is designed for people who are interested to know the detailed composition of the SP500 index and the daily and historical movements of all these stocks, which should be an useful resource for entry level investors who are interested in this topic.

The website will offer an overview of the SP500 composition and provide insights about performance of different sectors and different stocks, you can click into each stock to review its historical performance in any time intervals in the past several years, the key metrics that demonstrate its financial performance and operational performance. There will also be some benchmarking algorithm that shows how this stock performs comparing with the overall market average and the sector average.

Our group members include Cheng Chen, Ruikang Liu, Puran Zhang, Wenting Zhao.

**Architecture**

Our project used a React as our front end framework and Node.js as our back end framework. We stored our data in AWS server, and used MySQL to build queries in the back end to retrieve data we need for calculation or display on the pages.

Data processing: Python

Frontend/Backend: React/Node.js

Database: AWS RDS MySQL

**Data**

We used Yahoo Finance API as our data source, and extracts all historical stock data of SP500 stocks for the past three years and populate to our database in AWS.

Yahoo Finance API:

Description: API that contains stock price information and company information.

Link: https://github.com/ranaroussi/yfinance

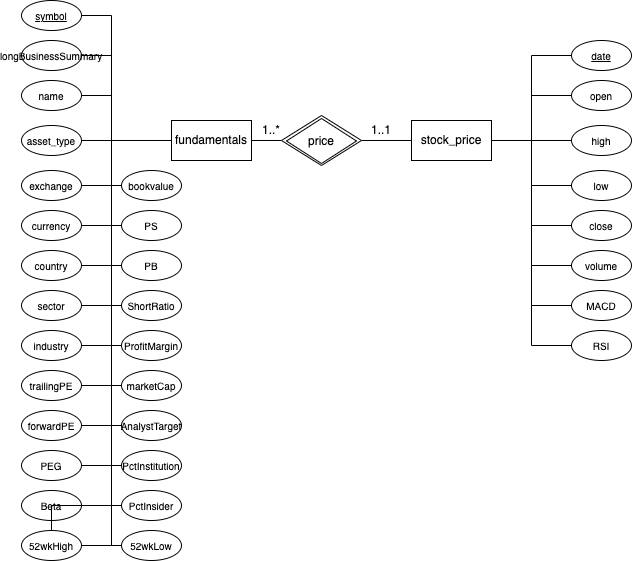
We have done careful and detailed screening on the data. This data has a few tables that can be used across the board. However, it is not clean and contains duplicated variables. As an open-sourced version, this data governance of this tool is not well implemented. So we have to evaluate the variables and do some cross check with data from other sites so that we can confirm on the availability as well as sanity. Initially we are using AlphaVantage which is more commercially and as a result, it is better in terms of the quality. We choose to move away from it because the high cost of such usage.

We also calculated some fields (technical indicators) directly in our python script as it will complicates the processes if we have to find the data source. It is a better approach as we can control the logic and not to create additional dependencies on the data quality and data availability. It is also more maintainable in the long run.

**Database**

We built two tables, one for stock price metrics, which includes daily price for all SP500 stocks over the past three years, and another one for stock fundamental data, which includes company info and financial metrics, etc. The price table contains more than 490,000 instances and the fundamental contains around 500 instances. Our tables are in normal form. For price table, the key is date and symbol, and all the rest of attributes are uniquely determined by the key and there is no functional dependencies between the non-key attributes. As for fundamental table, the key is the symbol, and although some attributes like sector and industry are correlated, but they are not used in our queries and decomposition will not reduce the number of instances, so they can be ignored or removed and all the other attributes are still uniquely determined by the key symbol. Therefore, our tables are already in 3NF.

ER Diagram is as below (attributes of each tables are a subset version):



**Queries**

For the stock detail page:

The most sophisticated query is the Stock\_Outperformance Query:

WITH target AS (

SELECT p.Symbol, f.Sector, p.Change,p.Volume, AVG(p.Change) OVER() AS avgChange, AVG(p.Volume) OVER() AS avgVolume

FROM Price p JOIN Fundamentals f ON p.Symbol = f.Symbol

WHERE Date = (

SELECT MAX(Date)

FROM Price

)

),

compute AS(

SELECT f.Sector, AVG(p.Change) AS sectorChange, AVG(p.Volume) AS sectorVolume

FROM target p JOIN Fundamentals f ON f.Symbol = p.Symbol

group by f.Sector

)

SELECT t.Change, t.Volume, t.Change - cp.sectorChange AS beatBySector, t.Volume - cp.sectorVolume AS beatVBySector, t.Change-t.avgChange AS beatByAll, t.Volume-t.avgVolume AS beatVByAll

FROM target t JOIN compute cp ON t.Sector = cp.Sector

WHERE t.Symbol = '${symbol}';

This query is used to calculate for each stock, how did it perform in price movement and in active trading volume comparing with the average of the overall market and average of stocks in the same sector it belongs to. With this query, we show these metrics in the benchmarking section in the stock detail page when user checking into a specific stock.

For the landing page:

The most advanced query is the sector performance table.

WITH a AS (  
 SELECT symbol, date, p.close, p.change,  
 ROW\_NUMBER() over (PARTITION BY symbol ORDER BY date desc) as rk  
 FROM Price AS p  
 WHERE DATEDIFF((SELECT MAX(Date) FROM Price), date) <= 20  
 ), b AS (  
 SELECT a1.symbol,  
 a1.change AS d1\_change,  
 (a1.close-a2.close)/a2.close AS d10\_change  
 FROM a AS a1  
 INNER JOIN a AS a2  
 ON a1.symbol = a2.symbol  
 AND a1.rk = 1  
 AND a2.rk = 11  
)  
SELECT f.sector,  
 CONCAT(ROUND(100 \* AVG(b.d1\_change), 3), '%') AS d1\_change,  
 CONCAT(ROUND(100 \* AVG(b.d10\_change), 3), '%') AS d10\_change  
 FROM Fundamentals AS f  
 INNER join b  
 ON f.symbol = b.symbol  
 GROUP BY f.sector;

This query is to compute the one-trading-day and ten-trading-days performance of each sector. This is used as an overview in the landing page in order to show the trend based on sector. For example, usually when oil or gas prices go up, it is very likely that energy sector will see an increase gradually. Also, when money tends to go to tech companies in recent days, that will show that technology sector is comparatively more bullish. Similarly, when fed rate goes up, the financial services sector might see a gradual jump too.

**Performance Evaluation**

In stock detail page, for the query of stock performance benchmarking, we originally split the calculation into multiple queries, for example, there was one query designed for stock performance in price movement vs market average, and another one for stock performance in price movement vs sector average, and similar two queries for volume change comparison. Each of these four queries have a complex structure of joins and temporary tables, which in turn makes the stock page loading very slowly, on average around 10 seconds to display all metrics because these four metrics requires four queries to run sequencially. Then we noticed that we can actually combine these queries into one, because for the comparisons of volume and price change, we can actually apply one query with just two more attributes to get the calculation for both at once. And for the comparisons of sector and overall market, we can still use temporary table to do the aggregation for sector but for overall market we can simply use OVER() to perform the aggregation in the original table, thus saved our time to run another query on the same page. In this way we combined the original four queries into one, and the stock page now can be loaded in less than 2 seconds on average.

In the example of 1-day and 10-day query, we do some optimization to eliminate the intermediate data that is used. Since stock doesn't simply trade on calendar days, we cannot simply do the calculation to get 1 day and 10 day directly through date. We need to order the date information and get row number of each so that we can compare. Note that there is an optimization that we do in the common table expression of "a", so that it will only do the sorting to a reduced amount of data (because we filter on the date in the where clause since regardless of what, it is guaranteed that any 10 trading days is within 20 calendar days). So by doing this the sorting cost will be significantly reduced. This small change alone help reduced 50% the total query time, from 2s 207 ms to 928 ms on average. We are just using 1000 trading days, which is a little more than 4 years historical data. However, if we want to store the entire history, and a significant proportion of stock would have more than 40 years of historical, this would mean an even larger query time improvement, considering the sorting is nlog(n) operation.

**Technical challenges**

When we are building the stock price table and charts, one of the key obstacles is formatting of the data, since most of our original data is in float, when we display the result directly in the page, it does not look professional because the numbers are either large without comma separated, or percentage shown as too many decimal places. But when if we change data format in the source or back end, it will cause a problem that the front end functions may not be working, for example, sorting in table, display of charts, etc. Therefore, we figured out that we should do the formatting in the front end, we researched a lot and finally found a solution to use toLocaleString() method to format our data in the front end so that we can display our data and charts in any format we need for the scenario without conflicting with the sorting and chart functions.

One technical challenge on the data is that we spend lots of time on collecting the data. In the beginning, we are trying to use a stock API provider which is called AlphaVantage. However, the free tier only limits to 500 requests per day and 5 requests per minute. Think about it, if we do this with a single IP, it means 100 minutes for each type of API for all S&P 500 stocks. And we just need to use 5-6 types of API. This is a great limitation for us as we need to get different data (meaning different API calls) for each of the 500 stocks. Since we are a group of four, technically we can call 4 API per stock. If we don’t try to abuse that system, which we should not do as well, we can ask our friends’ help too. But this would create a lot of dependencies and inflexibility for the project, and it is not what it should like for sure.

We try this way in the beginning, sometimes we didn’t manage the pace well so it will return empty result and we have to implement new functions, just to detect and rerun the data fetching job. We cannot simply rerun everything because we have a limitation on the speed as well the total quota. Even like that, our IP was blocked by the system.

So later on, even though we have built lots of abnormal value detection and retry functionalities, we decide to turn to open-source packages for help. That significantly reduce the workload. The downside is that we have no guarantee on the data quality because it could be considered as a free version which is not supervised by some profitable institution and the we have to do lots of screening on the data quality because it has many data variables that are overlapping with each and some of them have poor quality.