

Financial Data Report

Database Retrieval:

Find the following GitHub repository online:

<https://github.com/jaywhitlw45/Finance-Data-Management?tab=readme-ov-file#usage>

Download all files in the “/sql_files” directory.

Open each file in a **MySQL** database. Execute the commands in each file in this exact order:

1. **Create the database (first query create db command, then query the entire file):**
/sql_files/create_db_mysql.sql
2. **Insert stocks (query the entire file):**
/sql_files/insert_company_and_stocks.sql
3. **Insert daily metrics (query the entire file):**
/sql_files/insert_daily_metrics.sql
4. **Insert the customers and accounts (query the entire file):**
/sql_files/insert_customers_and_accounts.sql
5. **Insert transactions and holdings (query the entire file):**
/sql_files/insert_transactions_and_holdings.sql
6. **Perform various queries on the database (query one command at a time):**
/sql_files/example_queries.sql

NOTE: There is a create_db_sqlite.sql file designed for an sqlite database. This is not guaranteed to work with the insertion tables

Stock Metrics Retrieval (Optional)

This option requires downloading the GitHub repository.

Along with the SQL commands, there are two Python scripts provided to download historical stock data and generate SQL commands to insert that data into the database.

- Install the required Python packages:
pip install yfinance
- Run the data download script from the project's home directory (do not navigate to /stock_data): python /stock_data/download_stock_data.py
- The script will download historical stock data for the specified stocks and save it as CSV files. There is a date option in the script to adjust the desired range of dates.
- Run the sql insertion generator script from the project's home directory (do not navigate to /stock_data): python /stock_data/sql_insertion_generator.py
- The generated sql commands are stored in /sql_files/insert_daily_metrics.sql

All information is available in the README.md in the root directory of the GitHub repository.

Description

This project aims to manage financial market data and brokerage firm operations using a relational database system.

The project's data domain encompasses financial markets and brokerage firms operating within them. A brokerage firm facilitates customers' transactions involving securities such as stocks, bonds, options, and futures. The database supports brokerage firms in managing customer accounts and transactions, as well as providing information on the securities they offer for sale.

The relationships provided below keep track of customers, customer accounts, transactions, and stock data. Each customer may have more than one account. Each account consists of transactions that involve the exchange of stock. Additionally, there is information about stocks and their daily metrics.

Data Domain

Stock, Daily Metrics and Company are all data that was gathered from the internet.

Customer, Transaction, Account and Stock Holding were all populated with artificial data. The goal is to simulate a Customer buying and selling stock. The stock is placed into a Stock Holding account.

Stock	Stores information for stocks traded on an exchange.
<i>stock_id</i>	Key for table entry.
<i>symbol</i>	The symbol that is associated with the company name.
<i>company_name</i>	The name of the company that the stock represents
<i>stock_exchange</i>	The stock exchange that the stock is publicly traded on.

- Stock is designed to hold static information about a single stock.

DailyStockMetric	Stores information about the trading day.
<i>metric_id</i>	Key for table entry.
<i>stock_id</i>	The stock associated with the metrics. Foreign key to Stock Table.
<i>date</i>	Date of the metric. Dates for an individual stock are unique.
<i>open_price</i>	Price of the stock at market open.
<i>close_price</i>	Price of the stock at market close.
<i>high_price</i>	Highest price during the trading day.
<i>low_price</i>	Lowest price during the trading day.
<i>volume</i>	Amount of shares traded during the trading day.

- DailyStockMetric keeps track of a stocks value on a given trading day.

Customer	An account holder. May hold one or more accounts.
<i>customer_id</i>	Key for table entry.
<i>first_name</i>	First name of customer.
<i>last_name</i>	Last name of customer.
<i>email</i>	Email of customer. Every email is unique.
<i>date_of_birth</i>	Date of birth of customer.
<i>ssn</i>	Social Security Number for the customer. Every ssn is unique.

- Customers own trading accounts. They can buy and sell stock with these accounts; this is called a Transaction.

Transaction	Stores every purchase or sell an account has made.
<i>transaciton_id</i>	Key for table entry.
<i>stock_id</i>	The stock being purchased or sold. Foreign key to the Stock table.
<i>account_id</i>	The account making the transaction. Foreign key to the Account table.
<i>type</i>	An account can either buy or sell stock.
<i>date</i>	Date of the transaction
<i>stock_price</i>	The price of one stock at the time of purchase.
<i>quantity</i>	Number of stocks purchased for this transaction.

- Transactions occur when a Customer buys or sells stock.

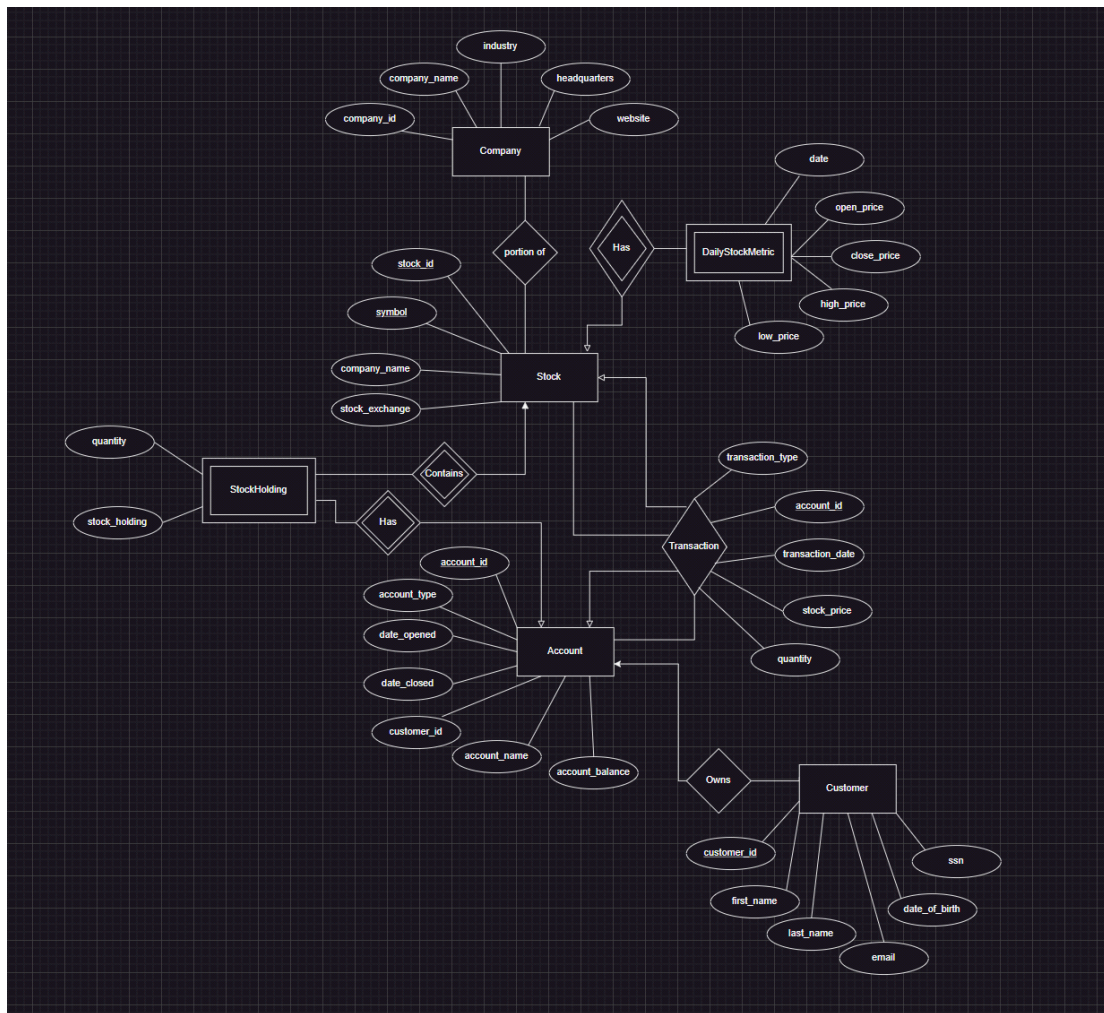
Account	Customers hold accounts that can store stock and/or currency.
<i>account_id</i>	Key for table entry.
<i>date_opened</i>	Date the account was opened.
<i>date_closed</i>	Date the account was closed. Set to NULL if the account is open.
<i>customer_id</i>	The owner of the account. Foreign key to the Customer table
<i>type</i>	There are three types of accounts, savings, checkings, and investment. All three accounts can store currency and/or stock.
<i>name</i>	The name given to the account by the Customer.
<i>balance</i>	The amount of currency in the account.

- Customers may own one or more accounts. Accounts store the amount of currency a customer has. Accounts are also contain stock via the StockHolding table.

StockHolding	The amount of one stock in one account.
<i>holding_id</i>	Key for table entry.
<i>account_id</i>	The account that holds this stock.
<i>stock_id</i>	The stock that this account holds.
<i>quantity</i>	The amount of stock in this stock holding.
<i>total_investment</i>	The total amount that has been paid for all stock in this account. In other words, the total cost of all transactions made for a specific account and specific stock.

- StockHolding stores stocks the Customer has purchased. If a Customer has 2 shares of Palantir and 5 shares of AAPL, there will exist two separate entries in StockHolding.

Entity Relation Diagram



Entities are represented by rectangles. These are typically the tables in a database.

- Company
- Stock
- DailyStockMetric
- Account
- Customer
- StockHolding

NOTE: Transaction is not considered an entity, but rather a relationship between account and stock

Weak Entity Sets are represented by the double rectangles. These are supported by strong entity relations and cannot be uniquely identified based on their own characteristics.

- **Daily Stock Metric:**
 - Daily Stock Metric depends on the Stock entity to exist. Its uniqueness is not established by its own attributes alone (such as the date and prices), but by its association with a Stock, referenced by the stock_id.

- The date attribute could act as a discriminator or partial key; however, only in combination with the stock_id from the Stock entity
- **Stockholding:**
 - Stockholding is also a weak it is dependent on both the Account and Stock entities, with foreign keys account_id and stock_id.
 - Stockholding relies on a combination of the account_id and stock_id from the strong entities Account and Stock to uniquely identify each holding.

Attributes of the entities are listed within the rectangles. These represent the columns within the tables.

- For example, the Company entity has attributes company_id, company_name, industry, headquarters, and website.

Primary Keys are attributes that uniquely identify a record within an entity and are typically underlined. In this diagram, they are the first attribute listed in each entity and are used to establish relationships between the entities.

- company_id, stock_id, account_id, and transaction_id are primary keys for their respective entities.

Unique keys in a database are constraints that ensure all values in a column are different from one another; no two rows can have the same value for the columns that are part of the unique key.

- company_name, email, ssn, and symbol are unique keys for their respective entities.

Relationships are represented by diamonds or lines connecting entities. They define how entities relate to each other.

- For example, a Stock is part of a Company, and an Account has a StockHolding.

Connectors (lines) indicate the type of relationship between entities:

- A line with a single bar at one end and an arrow at the other indicates a "one-to-many" relationship. The single bar indicates the "one" side, and the arrow points to the "many" side.
- Lines without arrows or bars might suggest a "one-to-one" or "many-to-many" relationship, depending on the context provided by the diagram.

Intersection Entities in a many-to-many relationship are represented by a rectangle with relationships to the entities it connects.

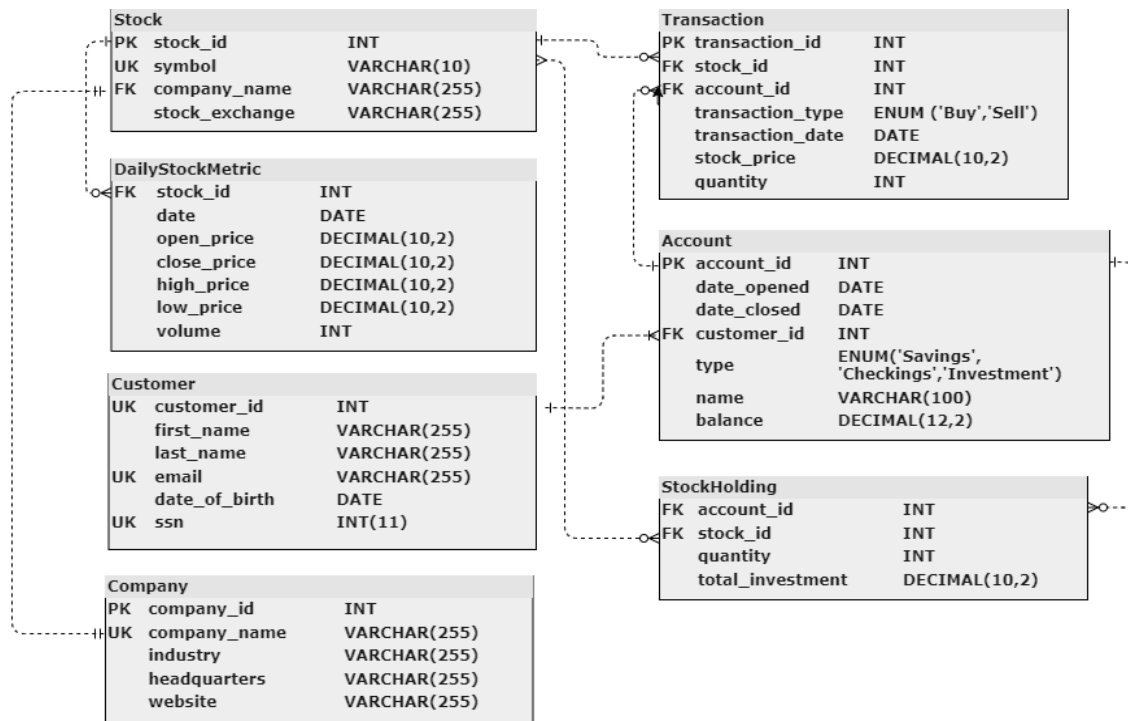
- StockHolding seems to be an intersection entity that ties Account and Stock together.

Secondary Attributes are non-primary key attributes that provide additional information about an entity.

- For example, first_name, last_name, email, date_of_birth, and ssn in the Customer entity.

Foreign Keys can be inferred based on the relationships. For instance, customer_id in Account likely references customer_id in Customer.

Database Schema



Database diagram schema

Company to Stock:

- A one-to-many relationship exists between Company and Stock. Each company can have multiple stocks, but each stock is issued by only one company. This is why there's a foreign key (**company_name**) in the Stock entity that references **company_name** in the Company entity.

Stock to DailyStockMetric:

- Another one-to-many relationship exists between Stock and DailyStockMetric. A particular stock will have multiple entries for daily metrics, one for each day, but each daily metric corresponds to only one stock, indicated by the foreign key (**stock_id**) in the DailyStockMetric entity.

Customer to Account:

- This relationship is also one-to-many. A single customer can own multiple accounts (savings, checking, investment, etc.), but each account is held by only one customer. The foreign key (**customer_id**) in the Account entity enforces this relationship.

Account to Transaction:

- Transactions are related to accounts by a one-to-many relationship. An account can have multiple transactions associated with it, while each transaction is linked to a single account, denoted by the foreign key (**account_id**) in Transaction.

Account to StockHolding:

- There is a one-to-many relationship between Account and StockHolding. An account can hold multiple types of stock, but each individual stock holding is associated with one account, shown by the foreign key (**account_id**) in the StockHolding entity.

Stock to StockHolding:

- Similarly, the relationship between Stock and StockHolding is one-to-many. A specific stock can be part of multiple holdings across different accounts, but each holding refers to one particular stock, which is why **stock_id** from the Stock entity is a foreign key in the StockHolding entity.

Referential Integrity Constraints

Referential integrity constraints are rules that enforce the consistency of relationships between entities. They are crucial for maintaining the correctness of a database by ensuring that foreign keys match primary keys in related entities.

In the entity relation diagram, the foreign keys are as follows:

Stock (company_name) references Company (company_name):

- Ensures that each stock can only be associated with an existing company.

DailyStockMetric (stock_id) references Stock (stock_id):

- Guarantees that stock metrics refer to a valid stock record.

Account (customer_id) references Customer (customer_id):

- Makes sure that every account is tied to a customer who exists in the Customer entity.

Transaction (account_id) references Account (account_id):

- Transactions must be related to an existing account.

StockHolding (account_id) references Account (account_id):

- Stock holdings must be linked to valid accounts.

StockHolding (stock_id) references Stock (stock_id):

- Each stock holding must correspond to an actual stock.

Example Query Demonstrating Referential Constraints

Create Table Query:

```
CREATE TABLE Accounts (
    account_id INT AUTO_INCREMENT PRIMARY KEY,
    date_opened DATE,
    date_closed DATE,
    customer_id INT,
    type ENUM('Savings', 'Checking', 'Investment'),
    name VARCHAR(100),
    balance DECIMAL(12, 2),
    FOREIGN KEY (customer_id) REFERENCES Customer(customer_id) on update
    cascade on delete cascade );
```

Cascade Demonstration Query:

```
Select * From Customer, Accounts;
DELETE FROM Customer WHERE ssn = '111-11-1111';
Select * From Customer, Accounts;
```


	customer_id	first_name	last_name	email	date_of_birth	ssn	account_id	date_open	date_close	customer_type	name	balance
1	1	Hudson	Darden	hudson@example.com	1991-05-14	111-11-1111	1	1996	2011	1 Checking	Hudson Checking	1000
2	1	Hudson	Darden	hudson@example.com	1991-05-14	111-11-1111	2	1998	2011	1 Savings	Hudson Savings	5000
3	1	Hudson	Darden	hudson@example.com	1991-05-14	111-11-1111	3	1996	NULL	1 Investment	Hudson Investment	10000
4	1	Hudson	Darden	hudson@example.com	1991-05-14	111-11-1111	4	1991	NULL	2 Checking	Asterios Checking	1500
5	1	Hudson	Darden	hudson@example.com	1991-05-14	111-11-1111	5	1991	NULL	2 Savings	Asterios Savings	7000
6	1	Hudson	Darden	hudson@example.com	1991-05-14	111-11-1111	6	1991	NULL	2 Investment	Asterios Investment	12000
7	1	Hudson	Darden	hudson@example.com	1991-05-14	111-11-1111	7	2011	NULL	3 Checking	Jaak Checking	2000
8	1	Hudson	Darden	hudson@example.com	1991-05-14	111-11-1111	8	2011	NULL	3 Savings	Jaak Savings	6000
9	1	Hudson	Darden	hudson@example.com	1991-05-14	111-11-1111	9	2011	NULL	3 Investment	Jaak Investment	8000
10	2	Asterios	Sante	asterios@example.com	1987-09-30	222-22-2222	1	1996	2011	1 Checking	Hudson Checking	1000
11	2	Asterios	Sante	asterios@example.com	1987-09-30	222-22-2222	2	1998	2011	1 Savings	Hudson Savings	5000
12	2	Asterios	Sante	asterios@example.com	1987-09-30	222-22-2222	3	1996	NULL	1 Investment	Hudson Investment	10000
13	2	Asterios	Sante	asterios@example.com	1987-09-30	222-22-2222	4	1991	NULL	2 Checking	Asterios Checking	1500
14	2	Asterios	Sante	asterios@example.com	1987-09-30	222-22-2222	5	1991	NULL	2 Savings	Asterios Savings	7000
15	2	Asterios	Sante	asterios@example.com	1987-09-30	222-22-2222	6	1991	NULL	2 Investment	Asterios Investment	12000
16	2	Asterios	Sante	asterios@example.com	1987-09-30	222-22-2222	7	2011	NULL	3 Checking	Jaak Checking	2000
17	2	Asterios	Sante	asterios@example.com	1987-09-30	222-22-2222	8	2011	NULL	3 Savings	Jaak Savings	6000
18	2	Asterios	Sante	asterios@example.com	1987-09-30	222-22-2222	9	2011	NULL	3 Investment	Jaak Investment	8000
19	3	Jaak	Pantelis	jaak@example.com	1996-02-10	333-33-3333	1	1996	2011	1 Checking	Hudson Checking	1000
20	3	Jaak	Pantelis	jaak@example.com	1996-02-10	333-33-3333	2	1998	2011	1 Savings	Hudson Savings	5000

Status

[14:40:40] Query finished in 0.001 second(s).

[14:40:40] SQLStudio was unable to extract metadata from the query. Results won't be editable.

[14:40:40] Query finished in 0.001 second(s).

	customer_id	first_name	last_name	email	date_of_birth	ssn	account_id	date_open	date_close	customer_type	name	balance
1	2	Asterios	Sante	asterios@example.com	1987-09-30	222-22-2222	4	1991	NULL	2 Checking	Asterios Checking	1500
2	2	Asterios	Sante	asterios@example.com	1987-09-30	222-22-2222	5	1991	NULL	2 Savings	Asterios Savings	7000
3	2	Asterios	Sante	asterios@example.com	1987-09-30	222-22-2222	6	1991	NULL	2 Investment	Asterios Investment	12000
4	2	Asterios	Sante	asterios@example.com	1987-09-30	222-22-2222	7	2011	NULL	3 Checking	Jaak Checking	2000
5	2	Asterios	Sante	asterios@example.com	1987-09-30	222-22-2222	8	2011	NULL	3 Savings	Jaak Savings	6000
6	2	Asterios	Sante	asterios@example.com	1987-09-30	222-22-2222	9	2011	NULL	3 Investment	Jaak Investment	8000
7	3	Jaak	Pantelis	jaak@example.com	1996-02-10	333-33-3333	4	1991	NULL	2 Checking	Asterios Checking	1500
8	3	Jaak	Pantelis	jaak@example.com	1996-02-10	333-33-3333	5	1991	NULL	2 Savings	Asterios Savings	7000
9	3	Jaak	Pantelis	jaak@example.com	1996-02-10	333-33-3333	6	1991	NULL	2 Investment	Asterios Investment	12000
10	3	Jaak	Pantelis	jaak@example.com	1996-02-10	333-33-3333	7	2011	NULL	3 Checking	Jaak Checking	2000
11	3	Jaak	Pantelis	jaak@example.com	1996-02-10	333-33-3333	8	2011	NULL	3 Savings	Jaak Savings	6000
12	3	Jaak	Pantelis	jaak@example.com	1996-02-10	333-33-3333	9	2011	NULL	3 Investment	Jaak Investment	8000

Status
[14:41:04] Query finished in 0.036 second(s). Rows affected: 4
[14:41:07] Query finished in 0.002 second(s).

The first picture shows the cascading relationship between Customer and Account where when a customer is deleted from the Customer relationship the corresponding customer account(s) will also delete. This also applies to all reference key relationships in the database.

Normal Forms

The database follows 3rd normal form, Boyce-Codd normal form, and 4th normal form. The following functional dependencies prove that the database follows Boyce-Codd Normal Form:

Stock (stock_id, symbol, company_name, stock_exchange)

stock_id -> symbol, company_name, stock_exchange

symbol -> stock_id, company_name, stock_exchange

From these functional dependencies it follows that both stock_id and symbol are keys.

There are no other functional dependencies which proves this relation is in BCNF.

DailyStockMetric (stock_id, date, open_price, close_price, high_price, low_price, volume)

stock_id, date -> open_price, close_price, high_price, low_price, volume

This is the only functional dependency in the relation. Stock_id and date together make a key for the relation. This relation is in BCNF.

Customer (customer_id, first_name, last_name, email, date_of_birth, ssn)

customer_id -> first_name, last_name, email, date_of_birth, ssn

email -> customer_id, first_name, last_name, date_of_birth, ssn

ssn -> customer_id, first_name, last_name, email, date_of_birth

These are the only functional dependencies of the relation. Customer_id, email, and ssn are all keys. This relation is in BCNF.

Company (company_id, company_name, industry, headquarters, website)

company_id -> company_name, industry, headquarters, website

company_name -> company_id, industry, headquarters, website

These are the only functional dependencies of the relation. Company_id and company_name are all keys, so the relation is in BCNF.

Transaction (transaction_id, stock_id, account_id, transaction_type, transaction_date, stock_price, quantity)

transaction_id -> stock_id, account_id, transaction_type, transaction_date, stock_price, quantity

This is the only functional dependency in the relation. Transaction_id is a key so the relation is in BCNF.

Account (account_id, date_opened, date_closed, customer_id, type, name, balance)

account_id -> date_opened, date_closed, customer_id, type, name, balance

This is the only functional dependency in the relation. Account_id is a key so the relation is in BCNF.

StockHolding (account_id, stock_id, quantity, total_investment)

account_id, stock_id -> quantity, total_investment

This is the only functional dependency in the relation. The combination of account_id and stock_id create a key for the relation so it is in BCNF.

When inspecting each relation in the database there are no multivalued dependencies. Each attribute of each tuple uniquely belongs to that tuple. **The database conforms to 4th normal form.**

Queries

Below is an explanation of 4 example queries. Additional queries are available in the example_queries.sql file. Throughout the database examples of subqueries, aggregation, insertion, and update queries can be found.

Query: Which stock had the largest price change between two dates?

```

101
102 -- Which stocks had the largest price change between 03-20-2024 and 03-20-2028
103 • SELECT symbol, company_name, ABS(end_metric.close_price - start_metric.open_price) AS price_change
104 FROM Stock
105 INNER JOIN DailyStockMetric AS start_metric ON Stock.stock_id = start_metric.stock_id
106 INNER JOIN DailyStockMetric AS end_metric ON Stock.stock_id = end_metric.stock_id
107 WHERE start_metric.date = '2024-03-20' AND end_metric.date = '2024-03-28'
108 ORDER BY price_change DESC;

```

Result Grid	Filter Rows:	Export:	Wrap Cell Content:
symbol	company_name	price_change	
COIN	Coinbase Global Inc	33.74	
AAPL	Apple Inc	4.24	
GOOGL	Alphabet Inc	2.93	
MSFT	Microsoft Corp	1.28	
PLTR	Palantir Technologies Inc	0.76	

In this query, our goal is to find the stock's largest price change within a specified date range. We want our output to be separated by the company symbol and name, so they are included in our SELECT statement. To ensure that we get the price change as a positive number, we use ABS when subtracting the close_price from the open_price. We set both open_price and close_price to end_metric and start_metric respectively.

We want to inner join our DSM table on both start_metric and end_metric with our Stock table to associate the stock_id from both tables. This ensures that the necessary data is pulled from the same stock. Then, we choose our specified date range using the DailyStockMetric start date and DailyStockMetric end date. Finally, we order the results by the biggest price change to the smallest.

Query: What is the highest traded stock in the technology industry?

```

78 -- What is the highest traded stock in the technology industry?
79
80 • SELECT S.symbol AS stock_symbol, S.company_name AS company_name, MAX(D.volume) AS max_volume
81 FROM Company AS C
82 JOIN Stock AS S ON C.company_name = S.company_name
83 JOIN DailyStockMetric AS D ON S.stock_id = D.stock_id
84 WHERE C.industry = 'Technology'
85 GROUP BY S.symbol, S.company_name
86 ORDER BY max_volume DESC
87 LIMIT 1;
88

```

Result Grid	Filter Rows:	Export:	Wrap Cell Content:	Fetch rows:
stock_symbol	company_name	max_volume		
AAPL	Apple Inc	121664700		

In this query, our goal was to find the highest traded stock in the tech industry. We are selecting from our Stock table and aliasing it as S, retrieving the attributes we want to output: the symbol, the company name, and the volume, which we obtain from the DailyStockMetric table, aliased as D. Since the industry is data we need from the company, we grab it from the Company table, aliased as C. Therefore, we join the Stock table with the Company table to retrieve the necessary data attribute. We set the company name from table C to correspond with table S so they both connect to the same company.

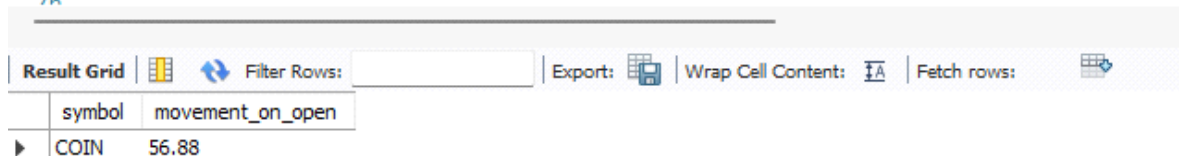
On the other hand, we join the DailyStockMetric table with the Stock table because we also need the stock's volume data to achieve our objective. Then, we filter the Company table to only include those in the 'Technology' industry and group the results by the following: symbol, company name, and maximum volume. We set the limit of the output to only 1 because we want to find the highest traded stock, not the highest traded stock from each tech industry.

Query: Which stock had the highest movement on the open over a 2 week period?

```

67  -- Which stock had the highest movement on the open over a given 2 week span?
68
69  • SELECT s.symbol, (MAX(dsm.open_price) - MIN(dsm.open_price)) AS movement_on_open
70  FROM DailyStockMetric dsm
71  JOIN Stock s ON dsm.stock_id = s.stock_id
72  WHERE dsm.date BETWEEN '2024-03-15' AND '2024-04-04'
73  GROUP BY s.symbol
74  ORDER BY movement_on_open DESC
75  LIMIT 1;
76

```



symbol	movement_on_open
COIN	56.88

In this query, our objective was to find the stock with the highest movement in the open price over a 2-week time period. We require data from both the Stock and DailyMetric tables. For our output, we want to display the symbol of the stock and the movement in the open price, calculated by subtracting the minimum open price from the maximum open price.

To ensure that we retrieve data from the correct stock, we perform an inner join between the two tables, linking the stock's stock_id with the dsm's stock_id. We specify a date range of a 2-week span and group the outputs by the symbol. Since this is an aggregated query involving MAX and MIN functions, we need to use the GROUP BY clause.

Finally, we set the limit to only 1 because we are interested in identifying the stock with the highest movement across all stocks.

Query: Which account has Hudson had open the longest?

```

22
23 -- Which account has Hudson have opened the longest.
24 • SELECT name AS AccountName, DATEDIFF(NOW(), date_opened) AS AccountAgeInDays
25 FROM Accounts
26 WHERE name LIKE '%Hudson%'
27 ORDER BY AccountAgeInDays DESC
28 LIMIT 1;
29

```

Result Grid | Filter Rows: | Export: | Wrap Cell Content: | Fetch rows:

	AccountName	AccountAgeInDays
▶	Hudson Checking	5657

In this query, we aim to identify the account that Hudson has held for the longest period. The required data originates from the Account table. Our desired output includes the account name and the number of days it has been open. To calculate the number of days, we use the DATEDIFF function, subtracting today's date (NOW()) from the date_opened, which is obtained from the Accounts table.

We set a constraint where the name is like '%Hudson%', as we want to consider the three accounts associated with Hudson: 'Hudson checkings', 'Hudson savings', and 'Hudson investment'. This ensures that only the accounts with "Hudson" in their name are considered, excluding those belonging to other customers like Jaak and Asterios.

Next, we order the results by the largest number of days and limit the output to one, as we are interested in seeing only the longest-held account.

Note: This query assumes that Hudson's name is included in all of his accounts. A more robust approach would involve joining the Accounts and Customer tables and then selecting the desired data.

Application/Use Cases

This Financial database has many uses as can be seen by the example queries. There are 2 main functions that the application provides.

- First is the ability to store and answer questions about stock data through use of the Daily Metric Table. Additional tables could be added to store quarterly and yearly financial metrics.
- Second, the database stores customer, account, and transaction information. This means that the database can serve as a backend to a brokerage firm. The firm can answer questions about its customer base and their specific needs based on the data.

As a final use case, the [GitHub repository](#) contains scripts that inject data from Yahoo Finance into the database. Using this method, a firm could store historical financial data for future analysis.

Conclusion

Working with a database with real data is a lot different than learning about databases in class using theory or abstractions. This relation also includes many more tables and attributes than the ones that we have seen in this class before. Building a database that is functional and can be used to gather useful data is a challenging yet satisfying process. Some challenges we had building queries were conflicts in joining tables. Sometimes queries would not work as intended, so we had to rebuild and rerun/check our tables to make sure the queries did what we intended them to do.

Just from the existing attributes, we can run queries that allow us to gather information about the types of accounts that are created, the types of customers that create those accounts, the types of stocks they hold, how stock prices change depending on the industry, which companies have had the greatest successes and failures on the stock market, and so on.

There are already many queries that can be ran using the existing database, but there is always an opportunity to expand. The more thorough the database, the more information can be determined.

Reference

<https://github.com/jaywhitlw45/Finance-Data-Management?tab=readme-ov-file#usage>

<https://finance.yahoo.com/?guccounter=1>