**Database Retrieval:**

Find the following GitHub repository online:

<https://github.com/jaywhtlw45/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

1. **Insert stocks (query the entire file):**

/sql\_files/insert\_company\_ and\_stocks.sql

1. **Insert daily metrics (query the entire file):**

/sql\_files/insert\_daily\_metrics.sql

1. **Insert the customers and accounts (query the entire file):**

/sql\_files/insert\_customers\_and\_accounts.sql

1. **Insert transactions and holdings (query the entire file):**

/sql\_files/insert\_transactions\_and\_holdings.sql

1. **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. |
| *stock\_id* | Key for table entry. |
| *symbol* | The symbol that is associated with the company name. |
| *company\_name* | The name of the company that the stock represents |
| *stock\_exchange* | 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. |
| *metric\_id* | Key for table entry. |
| *stock\_id* | The stock associated with the metrics. Foreign key to Stock Table. |
| *date* | Date of the metric. Dates for an individual stock are unique. |
| *open\_price* | Price of the stock at market open. |
| *close\_price* | Price of the stock at market close. |
| *high\_price* | Highest price during the trading day. |
| *low\_price* | Lowest price during the trading day. |
| *volume* | 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. |
| *customer\_id* | Key for table entry. |
| *first\_name* | First name of customer. |
| *last\_name* | Last name of customer. |
| *email* | Email of customer. Every email is unique. |
| *date\_of\_birth* | Date of birth of customer. |
| *ssn* | 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. |
| *transaciton\_id* | Key for table entry. |
| *stock\_id* | The stock being purchased or sold. Foreign key to the Stock table. |
| *account\_id* | The account making the transaction. Foreign key to the Account table. |
| *type* | An account can either buy or sell stock. |
| *date* | Date of the transaction |
| *stock\_price* | The price of one stock at the time of purchase. |
| *quantity* | Number of stocks purchased for this transaction. |

* Transactions occur when a Customer buys or sells stock.

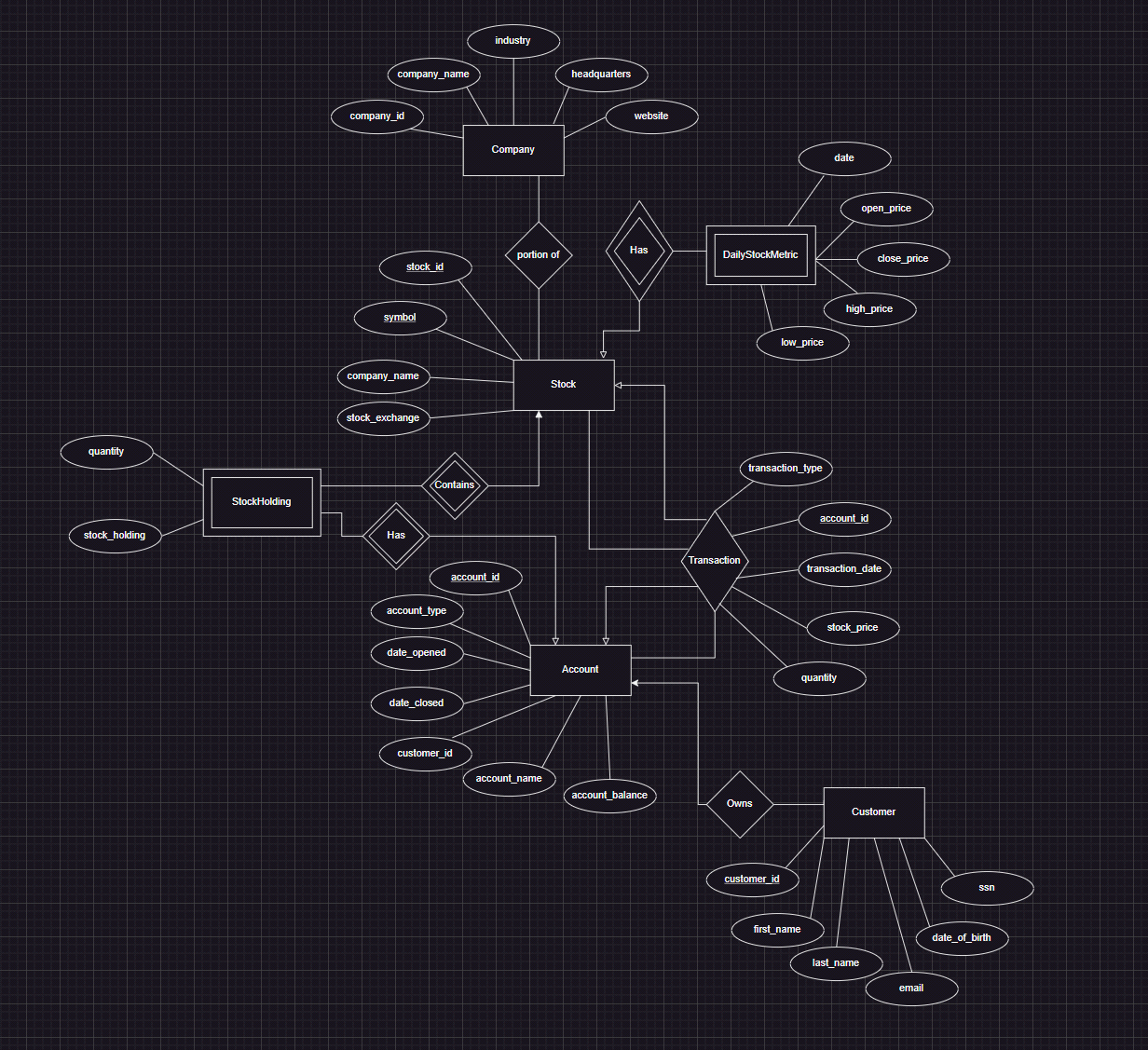
|  |  |
| --- | --- |
| **Account** | Customers hold accounts that can store stock and/or currency. |
| *account\_id* | Key for table entry. |
| *date\_opened* | Date the account was opened. |
| *date\_closed* | Date the account was closed. Set to NULL if the account is open. |
| *customer\_id* | The owner of the account. Foreign key to the Customer table |
| *type* | There are three types of accounts, savings, checkings, and investment. All three accounts can store currency and/or stock. |
| *name* | The name given to the account by the Customer. |
| *balance* | 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. |
| *holding\_id* | Key for table entry. |
| *account\_id* | The account that holds this stock. |
| *stock\_id* | The stock that this account holds. |
| *quantity* | The amount of stock in this stock holding. |
| *total\_investment* | 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** are represented by the double rectangles. Theses 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.

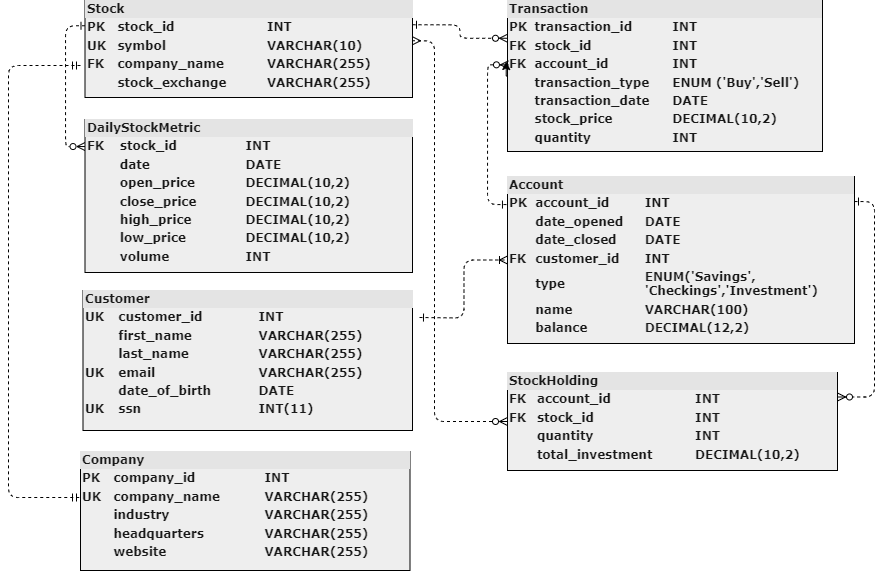
* StockHolding seems to be an intersection entity that ties Account and Stock together.

**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 Shema**



**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;

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

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)***

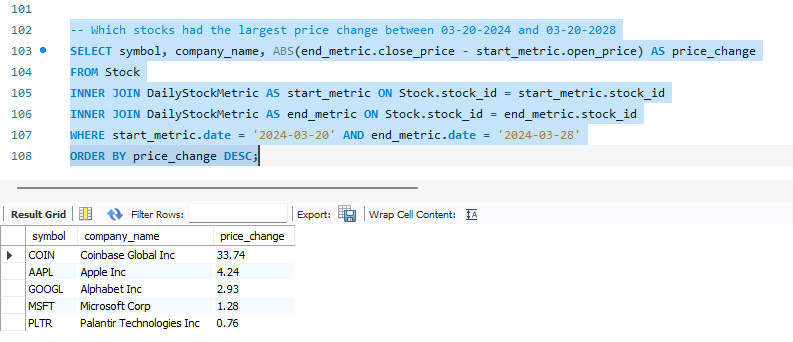
*account\_id, stock\_id -> quantity, total\_investment*

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.

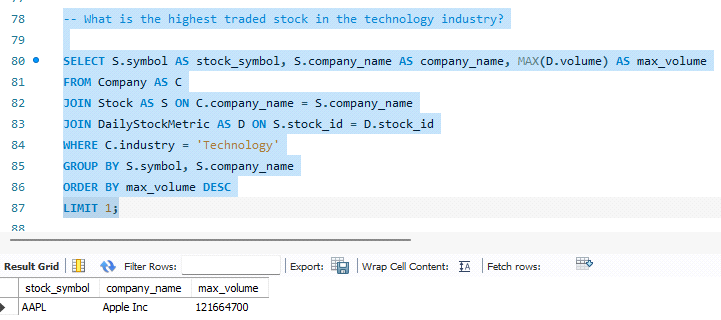
When inspecting each relation in the database there are no multivalued dependencies. Each attribute of each tuple uniquely belongs to that tuple. **The database conforms to 4th normal form.**

**Queries**

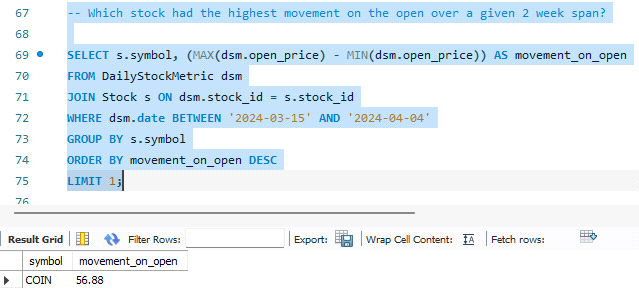
Below is an explanation of 4 example queries. Additional queries are available in the example\_queries.sql file. Throughout the database examples of subqueries, aggregation, insertion, and update queries can be found.



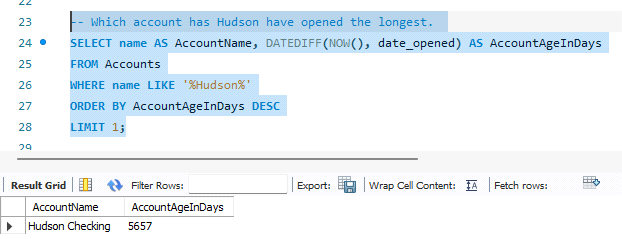
In this query, our goal is to find the stocks largest price change within a specified data. We want our output to be separated by the company symbol and name, so they are in our select. We want the price change and in a positive number so we use ABS when subtracting the close\_price from open\_price, in which we set both to end\_mertric and start\_metric respectively. We want to inner join our DSM table as both start\_metric and end\_metric to our Stock table to associate the stock\_id from both tables. This ensures that it pulls the neccessay data from the same stock. Then we choose our specified dates from the DSM start date and DSM end date, and lastly we order them by the biggest price change to the smallest.



In this query, our goal was to find the highest traded stock in the tech industry. We are selecting from our table Stock, and name it as S, and grab the attributes we want to output, the symbol, the company name, and the volume, which we grab from the table DailyStockMetric we set as D. Since industry is data we need from company, we need to grab it from Company which we set as C, so we join the Stock table with the company table to get the data attribute. We set the company name from table C to be from table S so they both connect to the same company. On the other hand, we join DailyStockMetric table with the stock table, because we also need the stock data volume to get what we want. Then we look in the Company table where it is a 'Technology' industry and group it by the following, symbol, company name, and max volume. We set the limit of the output to only 1 because we wanted to find the highest trade stock, not the highest traded stock from each tech industry.



In this query, our goal was to find the stock with the highest movement on the open over a 2 week time period. We know we will need data from the stock table and DailyMetric table. For our output, we want the symbol of our stock, and the movement on open which we find by subtracting the max open price of the stock from the minimum open price. We join both tables to ensure that we are pulling data from the correct stock, we set dsm's stock id to stocks stock id to ensure this. We set a specified date of a 2 week span and group the outputs by the symbol. Since it is an aggregated query, using the MAX and MIN, we need to use the GROUP BY. We set the the limit to only 1 because we want to see the highest from all stocks.



In this query, we want to see which account Hudson has opened for the longest. The data we will need stems from the Account table. We want the name and number of days in our output, DateDiff subtracts the dates of today's date which is NOW() from the date\_opened which is data from the Accounts table. We set a constraint where the name is like '%Hudson%', because we want to consider the three accounts, 'Hudson checkings', 'Hudson savings', and 'Hudson investment'. This will ensure that it will take the name which has the word "Hudson..... " and not any of the other accounts from the other customers Jaak and Asterios. We then order it by the largest, and limit it to one because we want to see only the longest account.

**Application/Use Cases**

This Financial database has a multitude of 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](https://github.com/jaywhtlw45/Finance-Data-Management) 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/jaywhtlw45/Finance-Data-Management?tab=readme-ov-file#usage>

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