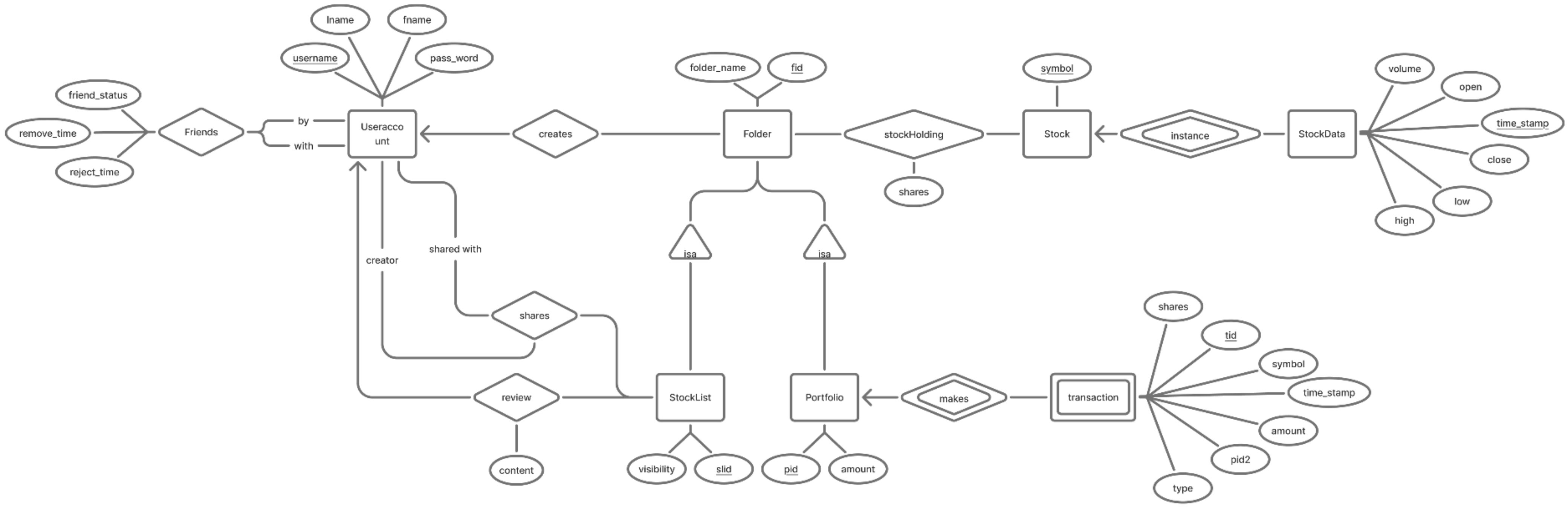
CSCC43 Final Project Report

Authors: Alvin Cao & Yong Le He

E/R Diagram



Relational Schema

- Useraccount(username, fname, name, pass_word)
- Folder(fid, folder_name)
- Portfolio(pid, amount)
- Stocklist(slid, visibility)
- Stock(symbol)
- Stockdata(symbol, timestamp, open, high, low, close, volume)
- Creates(username, fid)
- Stockholding(fid, symbol, shares)
- Friends(uid1, uid2, friend_status, reject_time, remove_time)
- Shares(creator, shared_with, slid)
- Reviews(reviewer, slid, content)
- Transaction(tid, pid, time_stamp, transaction_type, amount, other_pid, symbol, shares)

Normalization

For each of our schemas, we have the following non-trivial FD's

- Useraccount: username, fname -> name, pass_word
- Folder: fid -> folder_name
- Portfolio: pid -> amount
- Stocklist: slid -> visibility
- Stock: None, only one attribute
- Stockdata: symbol, timestamp -> open, high, low, close, volume
- Creates: fid -> username
- Stockholding: fid, symbol -> shares
- Friends: uid1, uid2 -> friend_status, reject_time, remove_time
- Shares: None
- Reviews: reviewer, slid -> content
- Transaction: tid -> pid, time_stamp, transaction_type, amount, other_pid, symbol, shares

All our FDs really just come from the primary key, hence, there aren't any other FDs. This schema is in BCNF because all the right-hand sides of the relations are keys. Since BCNF implies 3NF, it is also in 3NF.

Application Implementation with SQL

In the following section, we chose to report and talk about the more complicated SQL queries, but we have also put the rest of the SQL queries in the appendix for reference.

Portfolio and Stocklist Queries

View all the **stocklists** that is available for a user

A user can view any stock list that belongs to themselves, stock lists that are public and stock lists that have been shared with them. **Portfolios** have a similar query, just without the `visibility` checks.

```
SELECT DISTINCT s.slid, f.folder_name, s.visibility, c.username
FROM Stocklist s
JOIN Folder f ON s.slid = f.fid
JOIN Creates c ON f.fid = c.fid
LEFT JOIN Shares sh ON s.slid = sh.slid
WHERE c.username = :username OR s.visibility = 'public' OR sh.shared_with = :username
```

Viewing one **stock list**

For a user to view a stock list, we first check that the user is able to view it, with similar visibility checks as above. We then join the stock list with `Shareholding` to get all the shares of stocks it contains for displaying. It’s a left join, because we can have stock lists that have no stocks. For **Portfolios**, its the same logic, but without all those visibility checks, just check if the creator is `username`

```
SELECT *
FROM ((SELECT f.*, s.*, c.username
       FROM ((Folder f
              JOIN Stocklist s ON f.fid = s.slid) JOIN Creates c ON f.fid = c.fid)
              WHERE f.fid = :slid AND (EXISTS
              (SELECT 1 FROM Creates c WHERE c.username = :username AND c.fid = :slid)
              OR EXISTS (SELECT 1 FROM Shares sh
                         WHERE sh.shared_with = :username AND sh.slid = :slid)
              OR s.visibility = 'public'))
      NATURAL LEFT JOIN
      (SELECT fid, symbol, share, share * close AS value
        FROM Stockholding NATURAL JOIN StockPrices))
```

<u>Market value of a **stock list**</u>

The market value of a **stocklist** is calculated by adding the market value of all stockholdings (their last closing price). `StockPrices` is a materialized view of latest stock prices (see appendix). Our query joins `Stocklist` with `Stockholding` on a left join, because it is possible that a portfolio has no stockholdings, and we would not want to lose that data. It then calculates the value of each stock holding by multiplying `share` with the market value of the stock, and sums it in the end. **Portfolios** have the same query, except it joins from `Portfolio`, and the `amount` of cash in each portfolio is also added to the market value.

```
SELECT pid, COALESCE(SUM(value),0) AS value
FROM (SELECT pid, amount, close * share as value
       FROM (((SELECT * FROM Stocklist WHERE pid = :pid)
                LEFT JOIN Stockholding ON fid = pid)
               NATURAL LEFT JOIN StockPrices))
GROUP BY pid, amount
```

<u>Buying stocks for a **portfolio**</u>

Buying stocks is split into 4 separate queries. Inserting into `StockHolding` with the number of shares. Finding the price from the `StockPrices` view. Updating portfolio cash amount, and adding a transaction entry to stocks. **Stock lists** have a much simpler logic, and only need to complete step 1.
Selling stocks is basically the same, but reversed. For transferring money between portfolios, we simply update the two portfolio’s cash accounts. If depositing or transfering, its the same as transfering, but we only do it for one portfolio.

```
INSERT INTO Stockholding
      VALUES (:pid, :symbol, :shares)
      ON CONFLICT (fid, symbol)
      DO UPDATE SET share = Stockholding.share + EXCLUDED.share
      RETURNING *
SELECT close
      FROM StockPrices
      WHERE symbol = :symbol
UPDATE Portfolio
      SET amount = amount - :symbol
      WHERE pid = :pid
INSERT INTO Transaction (pid, amount, transaction_type, stock_symbol, stock_shares)
      VALUES (:pid, :price, 'stock', :symbol, :shares)
      RETURNING *
```

Sharing a stock list

```
SELECT * FROM friends
      WHERE (uid1 = :user AND uid2 = :friend) OR (uid1 = :friend AND uid2 = :user) AND
friend_status = 'accepted'
```

The user could only share a stocklist to their friend, the above query checks that they are friends to share stock list. Next, the visibility of the stock list would also have to change once shared unless it is already public. If the `visibility` is `private` then update its visibility to `shared`, this is done in the below query:

```
SELECT visibility FROM Stocklist WHERE slid = :slid
UPDATE stocklist
        SET visibility = 'shared'
        WHERE slid = :slid  AND visibility != 'public'
```

And to keep track of who shared what stock list to another, a tuple is inserted to the 'Shares' relation

```
INSERT INTO Shares
      VALUES (:user, :friend, :slid)
      ON CONFLICT (creator, shared_with, slid) DO NOTHING
      RETURNING *
```

Reviewing a stock list

For a user to review a stocklist, they must have access to it; either they are the creator of that stocklist, it is a public stock list, or that stocklist was shared with the user from another user. The below query checks to see if one of those conditions is met; else user cannot create a review on the stocklist.

```
SELECT 1
      FROM Stocklist s
      LEFT JOIN Shares sh ON s.slid = sh.slid
      WHERE s.slid = :slid
      AND (s.visibility = 'public' OR sh.shared_with = :friend OR s.visibility = 'private')
LIMIT 1
```

Once those conditions are met, then allow the user to create a new review or update an existing review from that same user.

```
INSERT INTO reviews
      VALUES (:user, :friend, :content)
      ON CONFLICT (reviewer, slid)
      DO UPDATE SET content = EXCLUDED.content
      RETURNING *
```

Friends Queries

A user sends a friend request to another user

`uid1` and `uid2` as inputs, where `uid1` is the user that is sending the request and `uid2` is the one receiving it. The query below gets the tuple in the relation 'Friends' that contains both the `:user` and `:friend`:

```
SELECT *, EXTRACT(EPOCH FROM (NOW() - GREATEST(reject_time, remove_time))) AS time
FROM friends
WHERE (uid1 = :user AND uid2 = :friend) OR (uid1 = :friend  AND uid2 = :user)
```

Next, there are validation steps involved to see if the above query returns a tuple, case 1. They are already friends if the `friend_status = accepted` in this case, do nothing

Case 2. where a friend request has been sent from `:friend` and has a `pending` friend_status, in this case we update the status to accepted

```
UPDATE friends
SET friend_status = 'accepted'
WHERE (uid1 = :user AND uid2 = :friend) OR (uid1 = :friend AND uid2 = :user)
RETURNING *
```

Case 3. There is an additional requirement that a user cannot resend the request in less than 5 minutes since rejection or delete. That is where the attributes, `reject_time` and `remove_time` play a role, which is used to check if it is okay to send a friend request, that is the query below:

```
UPDATE friends
SET friend_status = 'pending'
WHERE (uid1 = :user AND uid2 = :friend) OR (uid1 = :friend AND uid2 = :user)
RETURNING *
```

Case 4. There is no existing relation between `:user` and `:friend`; the below query creates that relation

```
INSERT INTO friends
VALUES (:user, :friend, 'pending')
RETURNING *
```

Statistics

Stock Statistics (Beta and COV)

We calculate both the Beta coefficient and COV of a stock in the same query, `Single` contains data about the close prices of a single stock, while `Market` is the close prices of the entire market, which is the sum of all close prices everyday. Beta is calculated by joining `Single` and `Market` on the same day, and taking the correlation of their close values. COV is calculated only from the `Single` table, as it is just the standard deviation of the close prices divided by the average.

Portfolio and Stocklist statistics (correlation matrix)

```
WITH Single AS (SELECT symbol, time_stamp, close FROM stockdata WHERE symbol = :symbol),
Market AS (SELECT time_stamp, SUM(close) AS close
        FROM stockdata
        WHERE symbol != :symbol
        GROUP BY time_stamp),
LastDate AS (SELECT time_stamp as last_date
        FROM StockPrices
        WHERE symbol = :symbol)
SELECT s.symbol, corr(m.close, s.close) AS beta, stddev(s.close) / avg(s.close) AS cov
FROM Market m
JOIN Single s ON m.time_stamp = s.time_stamp
WHERE s.time_stamp >= (SELECT last_date - (:interval || ' days')::INTERVAL
                              FROM LastDate)
GROUP BY s.symbol;
```

Portfolio Statistics (Correlation Matrix)

An explanation of why we chose to do this query in this way is in the query optimization section of the appendix. The first query finds all the stock symbols (`FolderStock`) belonging in the folder (parent of portfolio & stock lists), and joins with itself to find all pairs (`StockPairs`). We add a condition `f1.symbol < f2.symbol` so we only need to process half of the rows, since correlation is associative. We then check for any missing pairs, that is pairs of symbols that are not in `StockCorrelation` (see appendix for details on this table). For all the pairs not in the table, we insert them into the table and calculate their correlation. Then, the second query simply searches the `StockCorrelation` table for all the needed stock pairs in the folder. Since `StockCorrelation` only contains pairs where `f1.symbol < f2.symbol`, we union with the cases where `f1.symbol = f2.symbol` and `f1.symbol > f2.symbol` to construct the whole matrix.

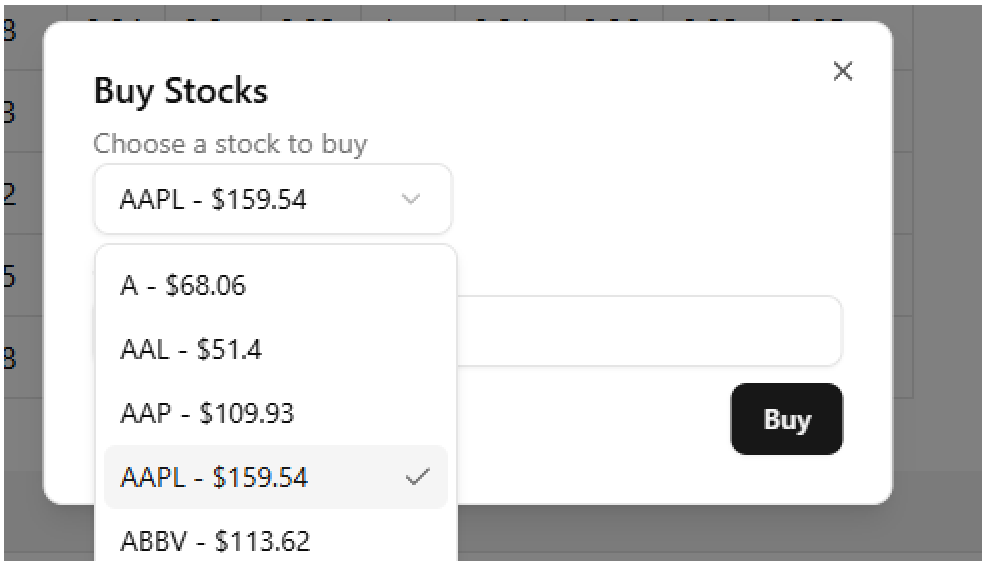
```sql
WITH FolderSymbols AS (
    SELECT symbol
    FROM Stockholding WHERE fid = :fid
),
FolderStock AS (
       SELECT symbol, time_stamp, close
       FROM stockdata
       WHERE symbol IN (SELECT symbol FROM Stockholding WHERE fid = :fid)
),
StockPairs AS (
       SELECT f1.symbol AS s1, f2.symbol AS s2
       FROM FolderSymbols f1
       JOIN FolderSymbols f2
       ON f1.symbol < f2.symbol
),
MissingPairs AS (
       SELECT sp.s1, sp.s2 FROM StockPairs sp
       WHERE NOT EXISTS (
               SELECT 1 FROM StockCorrelation sc
               WHERE sc.s1 = sp.s1 AND sc.s2 = sp.s2
               AND sc.interval_value = :interval || ' days')::INTERVAL)),
LastDate AS (
       SELECT MAX(time_stamp) as last_date FROM FolderStock
)
INSERT INTO StockCorrelation
SELECT mp.s1, mp.s2, (:interval || ' days')::INTERVAL, corr(p1.close, p2.close)
FROM MissingPairs mp
JOIN FolderStock p1 ON p1.symbol = mp.s1
JOIN FolderStock p2 ON p2.symbol = mp.s2 AND p1.time_stamp = p2.time_stamp
WHERE p1.time_stamp >= (
SELECT last_date - (:interval || ' days')::INTERVAL FROM LastDate
)
GROUP BY mp.s1, mp.s2;

WITH FolderSymbols AS (
       SELECT symbol FROM Stockholding WHERE fid = :fid
)
SELECT * FROM StockCorrelation
WHERE interval_value = (:interval || ' days')::INTERVAL
AND s1 IN (SELECT symbol FROM FolderSymbols)
AND s2 IN (SELECT symbol FROM FolderSymbols) AND s1 != s2
UNION ALL
SELECT symbol AS s1, symbol AS s2, (:interval || ' days')::INTERVAL AS interval_value, 1 AS corr
FROM FolderSymbols
UNION ALL
SELECT s2 AS s1, s1 AS s2, interval_value, corr FROM StockCorrelation
WHERE interval_value = (:interval || ' days')::INTERVAL
AND s1 IN (SELECT symbol FROM FolderSymbols)
AND s2 IN (SELECT symbol FROM FolderSymbols) AND s1 != s2
ORDER BY s1,s2;
```

Appendix

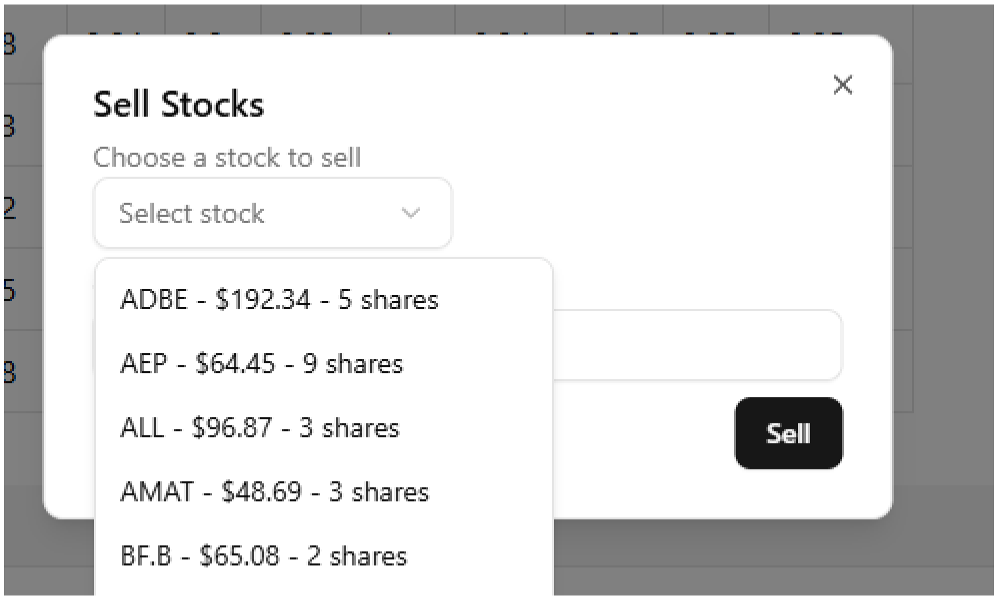
Optimized Interface

A list of our efforts to make the user interface more intuitive to use:

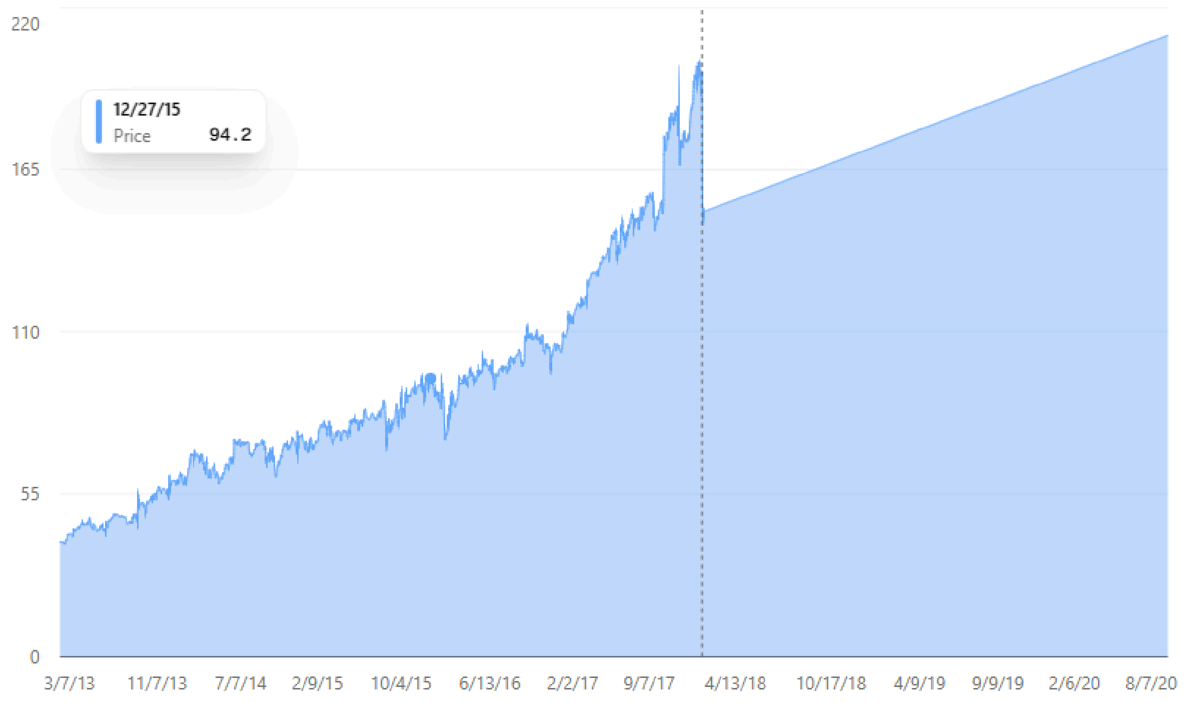
- When a user wants to buy stocks, we give them a list of available stocks to choose from (rather than them having to enter the stock symbol) and each stock's current market value



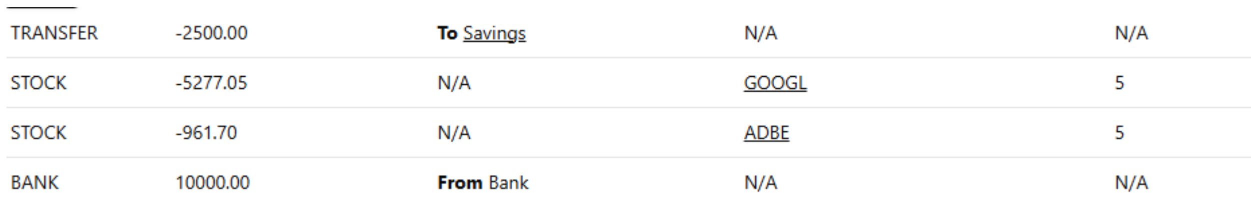
- When selling stocks, we also allow them to choose from the list of stocks they currently own, and we show them how many shares they own and the current market value



- We combined the historical performance and future performance of a stock into one visual graph, with an indicator on the latest price available

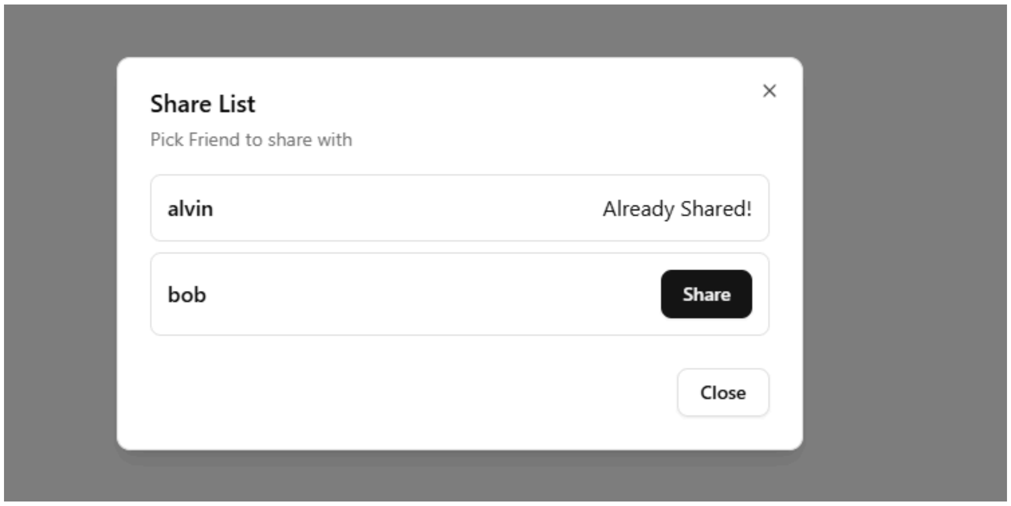


- When viewing transactions for a portfolio, we added links to the stocks or other portfolios mentioned in the transaction



TRANSFER	-2500.00	**To** Savings	N/A	N/A
STOCK	-5277.05	N/A	GOOGL	5
STOCK	-961.70	N/A	ADBE	5
BANK	10000.00	**From** Bank	N/A	N/A

- When sharing stock lists to friends, we make it apparent which friends the stock list was already shared to



Share List

Pick Friend to share with

alvin Already Shared!

bob

Optimized Query Performance

To optimize query performance, we implemented a few techniques

The first technique was by caching the current stock prices into a materialized view. Since current stock prices are used in many queries (buying/selling stocks, market value analysis), we created the following materialized view to cache that data:

```
CREATE MATERIALIZED VIEW StockPrices AS
(
      SELECT symbol, close, time_stamp FROM (
        Stockdata s1 NATURAL JOIN (
          SELECT s2.symbol, MAX(s2.time_stamp) as time_stamp
          FROM Stockdata s2
          GROUP BY symbol
        )
      )
);
```

We analyzed the performance of this by using `psql`, turning on `\timing`, running the raw query and selecting from the materialized view each ten times.

Times in MS	**1.00**	**2.00**	**3.00**	**4.00**	**5.00**	**6.00**	**7.00**	**8.00**	**9.00**	**10.00**	**Avg**
Raw	48.80	30.66	29.27	29.27	30.12	30.10	29.57	29.04	36.61	29.61	32.31
Materialized	0.90	0.55	0.64	0.58	0.59	0.69	0.61	0.55	0.80	0.63	0.65

We saw a 98% query time decrease with the historical stock prices. The decrease will only be more significant as more stock data is entered by the user. The materialized view is refreshed whenever new stock data is added, so that does increase the query time when adding new stock data, but because the other endpoints will be called much more frequently, this tradeoff is acceptable.

Another technique we implemented was to cache the correlation between 2 different stocks. This caching was to help with calculating the correlation matrix of each portfolio. If the portfolio holds many stocks, then the computation needed for calculating the matrix grows exponentially. Therefore, we created a table

```
CREATE TABLE StockCorrelation (
  s1 TEXT NOT NULL REFERENCES Stock(symbol),
  s2 TEXT NOT NULL REFERENCES Stock(symbol),
  interval_value INTERVAL NOT NULL,
  corr  DOUBLE PRECISION,

  PRIMARY KEY (s1, s2, interval_value)
);
```

Which stores the correlation of two stocks in a given time interval. Whenever a user first visits their portfolio page and views the correlation matrix, we calculate the correlation of those stocks on the fly, then store them into the cache table. The next time they, or any other user visits their own portfolio, we only have to calculate the correlation of the stocks that aren't already in cache (detailed query on page 4). We tested the performance on a portfolio that had 20 different stock holdings, and got the following results:

Raw Query: 162.45ms
Cached Query:

- Nothing in cache: 170.82 + 1.28 = 172.1ms
- Everything in cache: 58.90 + 1.44 = 61.34ms
- Half in cache: 113.94 + 1.51 =115.45ms

Therefore, other than the first case where we have nothing in cache, we will also see an improvement in the query times. Similar to the `StockPrices` view above, we remove the pairs from cache if the stock has been updated by the user.

Database Schema

```
CREATE TABLE Useraccount (
    username varchar(25) PRIMARY KEY,
    fname varchar(25),
    lname varchar(25),
    pass_word varchar(50)
);
CREATE TABLE Folder (
    fid SERIAL PRIMARY KEY,
    folder_name varchar(25)
);
CREATE TABLE Portfolio (
    pid SERIAL PRIMARY KEY,
    amount float,
    CHECK (amount >= 0),
    FOREIGN KEY (pid) REFERENCES Folder(fid)
        ON DELETE CASCADE
);
CREATE TYPE vis_enum AS ENUM ('private', 'shared', 'public');
CREATE TABLE Stocklist (
    slid SERIAL PRIMARY KEY,
    visibility vis_enum NOT NULL DEFAULT 'private',
    FOREIGN KEY (slid) REFERENCES Folder(fid)
        ON DELETE CASCADE
);
CREATE TABLE Stock (
    symbol varchar(5) PRIMARY KEY
);
CREATE TABLE Stockdata (
    symbol varchar(5) REFERENCES Stock(symbol),
    time_stamp date,
    open real,
    high real,
    low real,
    close real,
    volume int,
    PRIMARY KEY(symbol, time_stamp)
);
CREATE TABLE Creates (
    username varchar(25)  REFERENCES Useraccount(username) NOT NULL,
    fid int PRIMARY KEY,
    FOREIGN KEY (fid) REFERENCES Folder(fid)
        ON DELETE CASCADE
);
CREATE TABLE Stockholding (
    fid int NOT NULL REFERENCES Folder(fid)
        ON DELETE CASCADE,
    symbol varchar(5)  REFERENCES Stock(symbol) NOT NULL,
    share int NOT NULL CHECK (share > 0),
```

```sql
    PRIMARY KEY (fid, symbol)
);
-- Trigger to delete the stockholding if a user sells all shares
CREATE OR REPLACE FUNCTION CheckStockHoldingSharesFunction()
RETURNS TRIGGER AS $$
BEGIN
    IF NEW.share = 0 THEN
        DELETE FROM Stockholding sh
        WHERE sh.fid = NEW.fid AND sh.symbol = NEW.symbol;
        RETURN NULL;
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;
CREATE OR REPLACE TRIGGER CheckStockHoldingSharesGreaterZero
BEFORE UPDATE OF share ON Stockholding
FOR EACH ROW
EXECUTE FUNCTION CheckStockHoldingSharesFunction();
CREATE TYPE friend_enum AS ENUM ('pending', 'accepted', 'rejected', 'removed');
CREATE Table Friends (
    uid1 VARCHAR(25) REFERENCES Useraccount(username),
    uid2 VARCHAR(25) REFERENCES Useraccount(username),
    friend_status friend_enum DEFAULT 'pending',
    reject_time TIMESTAMP NULL,
    remove_time TIMESTAMP NULL,
    PRIMARY Key(uid1, uid2),
    CHECK (uid1 <> uid2)
);
CREATE TABLE Shares(
    creator VARCHAR(25) REFERENCES Useraccount(username),
    shared_with VARCHAR(25) REFERENCES Useraccount(username),
    slid int REFERENCES Stocklist(slid)
        ON DELETE CASCADE,
    PRIMARY KEY (creator, shared_with, slid)
);
CREATE TABLE Reviews(
    reviewer VARCHAR(25) REFERENCES Useraccount(username),
    slid int REFERENCES Stocklist(slid)
        ON DELETE CASCADE,
    content VARCHAR(4000) DEFAULT '',
    PRIMARY KEY (reviewer, slid)
);
CREATE TYPE transaction_enum AS ENUM ('bank', 'transfer', 'stock');
CREATE Table Transaction (
    tid SERIAL PRIMARY KEY,
    time_stamp TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    pid SERIAL NOT NULL REFERENCES Portfolio (pid)
        ON DELETE CASCADE,
    transaction_type transaction_enum NOT NULL DEFAULT 'bank',
    amount float NOT NULL,
    other_pid INT DEFAULT NULL REFERENCES Portfolio(pid)
        ON DELETE CASCADE,
    stock_symbol VARCHAR(5) REFERENCES Stock(symbol)
```

```
        ON DELETE CASCADE,
    stock_shares int,
    CHECK ((NOT transaction_type = 'bank') OR (other_pid IS NULL AND stock_symbol IS NULL AND
stock_shares IS NULL)),
    CHECK ((NOT transaction_type = 'transfer') OR (other_pid IS NOT NULL AND stock_symbol IS NULL
AND stock_shares IS NULL)),
    CHECK ((NOT transaction_type = 'stock') OR (other_pid IS NULL AND stock_symbol IS NOT NULL
AND stock_shares IS NOT NULL))
);
```

Cache Schema

```
CREATE MATERIALIZED VIEW StockPrices AS
(
      SELECT symbol, close, time_stamp FROM (
        Stockdata s1 NATURAL JOIN (
          SELECT s2.symbol, MAX(s2.time_stamp) as time_stamp
          FROM Stockdata s2
          GROUP BY symbol
        )
      )
);
CREATE TABLE StockCorrelation (
  s1 TEXT NOT NULL REFERENCES Stock(symbol),
  s2 TEXT NOT NULL REFERENCES Stock(symbol),
  interval_value INTERVAL NOT NULL,
  corr  DOUBLE PRECISION,

  PRIMARY KEY (s1, s2, interval_value)
);
```

User SQL queries

Finding user with the password for login

```
SELECT * FROM Useraccount
WHERE username = :username AND pass_word = :password
```

Creating a new user

```
INSERT INTO Useraccount (username, pass_word, fname, lname)
VALUES (:username,:password,:firstname,:lastname)
```

Portfolio queries and Stocklist Queries

Creating a new portfolio:

```
INSERT INTO folder (folder_name)
VALUES (:portfolioname)
RETURNING *;

INSERT INTO creates
VALUES (:username, :fid);

INSERT INTO portfolio
VALUES (:fid, 0);
```

Deleting a portfolio/stocklist

```
DELETE FROM folder
WHERE fid=:fid AND EXISTS (SELECT fid from Creates WHERE fid = :fid AND username = :username)
```

Withdrawing cash

```
UPDATE portfolio
SET amount = amount - :amount
WHERE pid = :pid AND EXISTS (SELECT fid from Creates WHERE fid = :pid AND username = :username)
INSERT INTO Transaction (pid, amount)
VALUES(:pid, -:amount)
RETURNING *
```

Depositing money

```
SELECT * FROM Creates
WHERE (fid=:pid1 OR fid=:pid2) AND username=:username

UPDATE Portfolio
SET amount = amount + :amount
WHERE pid = :pid1

UPDATE Portfolio
SET amount = amount - :amount
WHERE pid = :pid2

INSERT INTO Transaction (pid, transaction_type, amount, other_pid)
VALUES (:pid1,'transfer', :amoung, :pid2), (:pid2,'transfer', -:amount, :pid2)
RETURNING *
```

Depositing from external bank

```
UPDATE portfolio
SET amount = amount + :amount
WHERE pid = :pid AND EXISTS (SELECT fid from Creates WHERE fid = :pid AND username = :username)

INSERT INTO Transaction (pid, amount)
VALUES (:pid, :amount)
RETURNING *
```

Viewing all transactions for a portfolio:

```
SELECT transaction_type, amount, other_pid, folder_name AS other_portfolio, stock_symbol,
stock_shares, time_stamp FROM (
      (Transaction t
       JOIN Creates c ON t.pid = c.fid AND c.username = :username)
       LEFT JOIN Folder f ON other_pid = f.fid)
WHERE pid = :pid
```

Creating a new Stock list:

```
INSERT INTO folder (folder_name)
VALUES (:stocklistname)
RETURNING *;

INSERT INTO creates
VALUES (:username, :slid);

INSERT INTO stocklist
VALUES (:slid, 'private')
```

Updating the visibility
```
UPDATE stocklist
SET visibility = :visibility
WHERE slid = :slid
RETURNING *
DELETE FROM shares
WHERE slid = :slid AND creator = :username
```
Updating review content
```
UPDATE reviews
SET content = :content
WHERE slid = :slid
RETURNING *
```
Query to find stocklist that user shared
```
SELECT shared_with
FROM shares
WHERE slid = :slid and creator = :username
```
View all reviews for a stocklist
```
SELECT r.*, c.username AS owner
FROM ((reviews r JOIN Creates c ON r.slid = c.fid)
        JOIN Stocklist s ON r.slid = s.slid)
WHERE r.slid = :slid AND (r.reviewer = :username OR c.username = :username OR s.visibility =
'public')
```
Adding stocks to stocklist
```
SELECT * FROM Creates
WHERE username=:username AND fid=:slid

SELECT * from Stockholding
WHERE fid = :slid AND symbol = :symbol

INSERT INTO Stockholding
VALUES (:slid, :symbol, :shares)
ON CONFLICT (fid, symbol)
DO UPDATE SET share = Stockholding.share + EXCLUDED.share
RETURNING *
```
Removing a review from a stocklist
```
SELECT * FROM Creates
WHERE username=:username AND fid=:slid

DELETE FROM reviews
WHERE slid=:slid AND reviewer=:username
```
Friend Queries

Accepting a friend request
```
UPDATE friends
SET friend_status = 'accepted'
WHERE (uid2 = :username AND uid1 = :friend) AND friend_status = 'pending'
RETURNING *
```

Rejecting a friend request

```
UPDATE friends
SET friend_status = 'rejected', reject_time = NOW()
WHERE (uid2 = :username AND uid1 = :friend) AND friend_status = 'pending'
RETURNING *
```

View all friend requests from a user

```
SELECT uid1 as requester
FROM friends
WHERE uid2 = :username AND friend_status = 'pending'
```

View all friend requests sent from user

```
SELECT uid2 as requester
FROM friends
WHERE uid1 = :username AND friend_status = 'pending'
```

View all friends

```
SELECT uid1, uid2
FROM friends
WHERE friend_status = 'accepted' AND (uid1 = :username OR uid2 = :username)
```

Removing a friend request

```
UPDATE friends
SET friend_status = 'removed', remove_time = NOW()
WHERE (uid2 = :username AND uid1 = :firend) AND friend_status = 'accepted'
RETURNING *
```

Canceling a friend request

```
DELETE FROM friends
WHERE (uid1 = :username AND uid2 = :friend) AND friend_status = 'pending'
RETURNING *
```