CS 1555/2055 – DATABASE MANAGEMENT SYSTEMS (SPRING 2021)
DEPT. OF COMPUTER SCIENCE, UNIVERSITY OF PITTSBURGH

Project: BetterFuture

Release: April 1, 2021 Due: April 24, 2021 @ 8:00pm

Project Goal

The goal of this project is to design and implement an electronic investing system of 401(k) retirement accounts. The core of such a system is a database system with a set of ACID transactions. Our system, which we call "BetterFuture", allows personal investors to buy shares of mutual funds (of stocks, bonds, or mixed of both), exchange-traded shares, and keep track of their BetterFuture investments.

HW5 can be thought as the first project milestone, which included the descriptions and schemas of the database on which the system is based and you implemented a number of core triggers, functions and stored procedures. In this the project you will develop the "full" electronic investing system using Java, PostgreSQL and JDBC.

Descriptionl

The database for *BetterFuture* contains 8 tables plus an auxiliary table, explained as follows. You are required to define all of the structural and semantic integrity constraints and their modes of evaluation. For both structural and semantic integrity constraints (e.g., UNIQUE, NOT NULL, CHECK), you must state your assumptions as comments in your database creation script.

BetterFuture Tables:

- **Mutual Funds Information**
  - MUTUALFUND (symbol, name, description, category, c_date)
    PK (symbol)
    Datatype
    - symbol: varchar(20)
    - name: varchar(30)
    - description: varchar(100)
    - category: varchar(10)
    - c_date: date

    BetterFuture offers a variety of mutual funds such as money-market, real-estate, short-term-bonds, long-term-bonds, balance-bonds-stocks, social-responsibility-bonds-stocks, general-stocks, aggressive-stocks, and international-markets-stocks. Each mutual fund has a name (e.g., money-market), is identified by its symbol (e.g., SRBS), and belongs to one or more categories: fixed, bonds, stocks and, mixed. For example, money-market and real-estate are fixed, while balance-bonds-stocks, and social-responsibility-bonds-stocks are mixed. The database also maintains a description for each fund and the date when the fund was created. In addition, the database keeps track of the closing price of each mutual fund which is used in the following day for purchases, exchanges-traded shares

and calculation of investments.

- CLOSING_PRICE (symbol, price, p_date)
  PK (symbol, p_date)
  FK (symbol) → MUTUALFUND(symbol)
  Datatype
  - symbol: varchar(20)
  - price: decimal(10, 2)
  - p_date: date

- **Customers Information**
  - CUSTOMER(login, name, email, address, password, balance)
    PK (login)
    Datatype
    - login: varchar(10)
    - name: varchar(20)
    - email: varchar(30)
    - address: varchar(30)
    - password: varchar(10)
    - balance: decimal(10, 2)

    For every customer, we store their name, address, email address, a unique login-name and password. Each customer may own some funds and also has their allocation preferences based on which assets are allocated. Every time a customer deposits an amount to their 401(k) account, this amount is invested in shares based on the allocation preferences expressed as percentages. Customers can change their allocation preferences once a month. A customer can distribute their investment to as many funds as they want. Each fund will have a corresponding percentage and the sum of the percentages should be 100%.
  - ALLOCATION(allocation_no, login, p_date)
    PK (allocation_no)
    FK (login) → CUSTOMER(login)
    Datatype
    - allocation_no: int
    - login: varchar(10)
    - p_date: date

    A customer should create an allocation entry first, and then associate it with a mutual fund as a preference (i.e., each entry in this table corresponds to one or more symbols in the table PREFERS). This table is also used for auditing the customer allocation. The allocation used for the investments is always the most recent one.
  - PREFERS(allocation_no, symbol, percentage)
    PK (allocation_no, symbol)
    FK (allocation_no) → ALLOCATION(allocation_no)
    FK (symbol) → MUTUALFUND(symbol)
    Datatype
    - allocation_no: int

    - symbol: varchar(20)
    - percentage: decimal(3, 2)

  - OWNS(login, symbol, shares)
    PK (login, symbol)
    FK (login) → CUSTOMER(login)
    FK (symbol) → MUTUALFUND(symbol)
    Datatype
    - login: varchar(10)
    - symbol: varchar(20)
    - shares: int

- **Administrators Information**

  - ADMINISTRATOR(login, name, email, address, password)
    PK (login)
    Datatype
    - login: varchar(10)
    - name: varchar(20)
    - email: varchar(30)
    - address: varchar(30)
    - password: varchar(10)

    Due to company rules, administrators **cannot** be customers of BetterFuture. Thus, we keep their information in the table above.

- **Transactions History Information**
  - TRXLOG(trx_id, login, symbol, t_date, action, num_shares, price, amount)
    PK (trx_id)
    FK (login) → CUSTOMER(login)
    FK (symbol) → MUTUALFUND(symbol)
    Datatype
    - trx_id: serial
    - login: varchar(10)
    - symbol: varchar(20)
    - t_date: date
    - action: varchar(10)
    - num_shares: int
    - price: decimal(10, 2)
    - amount: decimal(10, 2)

    The history of trading transactions is maintained in the TRXLOG table. Each transaction records the customer, the mutual fund involved in the transaction, the number of shares, the price of the mutual fund and the total amount. The date when the transaction was processed and the action taken are also stored. The attribute action can be either 'deposit' or 'sell' or 'buy'.
    Note: the ***trx_id*** attribute is of type serial, that is auto incremented, and the documentation can be found at:
    https://www.postgresql.org/docs/13/datatype-numeric.html

Auxiliary Table:

- **Our "pseudo" Date Information**

  – MUTUAL_DATE (p_date)
  PK (p_date)
  Datatype
  * p_date: date

  You must maintain a "pseudo" date (not the real system date) in this (auxiliary) table. The reason for making such date, and not use system one is to make it easy to generate scenarios (time traveling) to debug and test your project. All dates should be based on this "pseudo" date. No attribute in the *BetterFuture* tables should refer to the MUTUAL_DATE table. MUTUAL_DATE has only one tuple, inserted as part of initialization and is updated during time traveling.

Sample data: the provided example tuples above and in the sample-data file are intended for clarification of the description, and are not sufficient for testing the correctness of your solution. You need to add or modify the sample-data accordingly in order to test your solution.

Triggers

In HW5, you developed two triggers: `buy_on_date`, and `buy_on_price`. In this phase you are asked to develop three additional triggers. You should feel free to develop more triggers to make your application more robust and efficient.

- You should create a trigger, called `price_initialization`, that adds a closing price to the corresponding symbol. The closing price should be the lowest closing price of the most recent prices of all the symbols.
- You should create a trigger, called `sell_rebalance`, that adjusts the balance of a customer-when shares are sold.
- You should create a trigger, called `price_jump`, that sells the shares automatically for a customer when the new closing price has a big difference ($10) with the previous most recent closing price.

Bringing it all together

For this project, you need to design two easy-to-use interfaces for two different user groups: *administrators* and *customers*. A simple text-oriented menu with different options is sufficient enough for this project. You may design nice graphic interfaces, but it is not required and it carries *NO bonus points*. A good design may be a two level menu, where the lower level menu provides different options for different users, depending on whether the user is an administrator or customer.

Each menu contains a set of tasks as described below. In implementing these tasks, you can use the sample solution (database schema and sample SQL functions, stored procedures and triggers) of HW5 as a starting point. You may need to modify them in developing a robust JAVA app. You can create temporary tables and (materialized) views if needed, but you should not change the table schemas without permission. You are also expected to follow the best practices of program decomposition, using (stored) functions, (stored) procedures, and views as appropriate.

You are also required to develop and submit a JAVA driver program to demonstrate the correctness of your BetterFuture backend by calling all of the above functions. *(Hint: You may want to develop the driver program as you develop the functions as a way to test them.)*

Administrator Interface

1: Erase the database
2: Add a customer
3: Add new mutual fund
4: Update share quotes for a day
5: Show top-k highest volume categories
6: Rank all the investors
7: Update the current date (i.e., the "pseudo" date)

Note: You should process the INSERT statements of each table as a single transaction. For instance, all INSERT statements for inserting the sample data of the mutual funds information should be processed in a single transaction, while all INSERT statements for customers should occur in a separate transaction from mutual funds information.

Task #1: Erase the database
Ask the user to verify deletion of all the data.
Simply delete all the tuples of all the tables in the database.

Task #2: Add a customer
Ask the user to supply the customer's information (login, name, email, password) and the initial balance. If the balance is not given, the balance should be initialized with **0**. Then insert the information into the appropriate tables.

Task #3: Add new mutual fund
Ask the user to supply the needed information (column_name, symbol, name, description, category, c_date) to create a new mutual fund.

Task #4: Update share quotes for a day
Ask the user to supply the filename where all the prices for all the mutual funds, and then load them into the appropriate table(s).
Here is an example of such an input file:

```
MM,15.00
RE,14.20
STB,11.40
LTB,14.00
BBS,10.50
SRBS,12.00
```

Task #5: Show top-k highest volume categories
Ask the user to supply the k value, and display the corresponding categories. The categories in the result are the top k categories based on the number of shares owned by customers.

Task #6: Rank all the investors
The system should rank the customers based on the total value of shares that they owned according to the most recent closing price.

Task #7: Update the current date (i.e., the "pseudo" date)
Ask the user to supply a date to be set as the current date (p_date) in MUTUAL_DATE table.

Customer Interface

1: Show the customer's balance and total number of shares
2: Show mutual funds sorted by name
3: Show mutual funds sorted by prices on a date
4: Search for a mutual fund
5: Deposit an amount for investment
6: Buy shares
7: Sell shares
8: Show ROI (return of investment)
9: Predict the gain or loss of the customer's transactions
10: Change allocation preference
11: Rank the customer's allocations
12: Show portfolio [proc]

Task #1: Show the customer's balance and total number of shares
Display the name, balance and total number of shares owned by the customer.

Task #2: Show mutual funds sorted by name
Display to the user all the mutual funds and their information sorted alphabetically.

Task #3: Show mutual funds sorted by prices on a date
Ask the user to supply a date, and then display to the user all the mutual funds and their information sorted based on the prices high to low, marking those owned by the customer.

Task #4: Search for a mutual fund
Ask the user to supply up to two keywords, and then return a text with the products that contain ALL these keywords in their mutual fund description in the format: "[symbol_1, symbol_2, ...]". Where symbol_1 and symbol_2 are symbols for the products that contain the search keywords in their description.

Task #5: Deposit an amount for investment
Ask the user to supply an amount for investment (a deposit), then use that to buy shares based on their allocation preference.
Note that a 'deposit' transaction should make a set of 'buy' transactions.
Investing should either result in buying shares of all mutual funds as specified in the allocation or none. Buying mutual funds partially is not an option. Thus, when an amount is deposited, and the new balance in the account is not sufficient to buy all the shares as specified in the allocation, that amount is just deposited in the account. In addition, any remaining amount after an investment becomes the new balance in the account.

Task #6: Buy shares
Ask the user to provide: (i) the symbol and number of shares of the mutual fund they want to buy; or (ii) the symbol and the amount to be used in the trade. In both cases, the required amount should not exceed the balance in the user's account.
(i) When a user buys shares by specifying the number of shares to be bought, if the balance is not sufficient to buy all the shares, no share is bought.

(ii) When a user buys shares by specifying an amount which is equal or less of the balance in their account, the user buys the maximum number of shares based on the specified amount and the remaining amount remains in the balance.
Also, the price of a share on a given day is the closing price of the most recent trading day.
Note that the balance should be updated automatically by the respect trigger.

Task #7: Sell shares
Ask the user to provide the symbol and number of shares of the mutual fund they want to sell. The resulting amount from the sell is added in the customer's balance which can be used for buying shares of another mutual fund in future.
Note that the balance should be updated automatically by the respect trigger.

Task #8: Show ROI (return of investment)
Display the symbol, the mutual fund name and the return of investment (ROI) of the customer.
$ROI = \frac{Current\ Value\ of\ Investment - Cost\ of\ Investment}{Cost\ of\ Investment}$

Task #9: Predict the gain or loss of the customer's transactions
Display the difference for each transaction amount with the "predicted" transaction amount, which is based on the most recent closing price. You should also display next to the difference, the status of how successful was the decision of buying or selling. The statuses are 3: (i)"loss", if the predicted buy transaction has a higher price or the predicted sell transaction has a lower price than the already processed transaction; (ii)"profit", if the predicted buy transaction has a lower price or the predicted sell transaction has a higher price than the already processed transaction; (iii)"hold", if the predicted buy transaction or the predicted sell transaction has the same price with the already processed transaction.

Task #10: Change allocation preference
Ask the user to provide the symbol of mutual fund and the percentage for all the desired funds. The total of all percentages should be 100% to be updated successfully.
Note that the allocation can change only once per day

Task #11: Rank the customer's allocations
Display the rank of the customer's allocations based on the ROI. Considering the ROI calculation, you should use the most recent closing price for all the allocation periods.
Note that the allocation can change only once per day

Task #12: Show portfolio
This task generates a performance report of a user's portfolio based on their owned shares. That means, find out what mutual funds the customer owns, and find out their current_values (the most recent prices of currently owned mutual funds multiplied by the number of owned shares), their cost (the total purchase price, for all currently owned mutual funds), their adjusted_cost (the cost value minus the sum of all the sales values of a given stock), as well as their yield (the current_value minus the adjusted_cost). The report will contain the following: 1) mutual fund symbols, 2) number of shares owned of each mutual fund, 3) current_value of each mutual fund, 4) cost value for each mutual fund, 5) the adjusted_cost for each mutual fund, 6) the yield for each mutual fund, and 7) the total value of the portfolio on the current date.

You must verify the login name and password of all users at the beginning. By default, we assume a pre-existing administrator with a login name "admin" and a password "root". This information can be inserted into your table after you create the database. You can add yourselves as the first customers.

How to proceed

Before implementing the required new/modified SQL queries within your application program, you should test them using pgsql (after you populate the database with test data). Only after you are confident that you have the correct query should you start implementing it in JDBC.

The tasks have been ordered so that you continue building on previous tasks as you move along. Therefore, it is advisable to implement them in order, starting from administrative task #1.

You *should use* views, stored functions and procedures when implementing your tasks. Recall that stored functions/procedures as well as triggers can be used to make your code more efficient besides enforcing integrity constraints. You should feel free to use additional triggers as you feel necessary.

All errors should be captured "gracefully" by your application. That is, for all tasks, you are expected to check for and properly react to any errors reported by PostgreSQL (DBMS), providing appropriate success or failure feedback to the user. Also, your application must check the input data from the user for validity and to avoid any vulnerabilities (e.g., SQL injection). You need to create your own test data starting with the example data above.

You need to design the **SQL transactions** appropriately and when necessary, use the concurrency control mechanisms supported by PostgreSQL (e.g., isolation level, locking modes) to make sure that inconsistent state will not occur. There are two situations that you should handle:

- If the same user logs in and runs functions on two different terminals concurrently, the application semantics should remain still intact. For example, if the customer attempts to buy shares from two different terminals for a sum higher than his/her balance, one of the transactions should fail.
- If the administrator updates the share price of a mutual fund and/or updates today's date while a customer is buying/selling shares of the same mutual fund, the state of the database should stay consistent. For example, the administrator updates the per share price of stock XYZ from $10 to $15 while a customer is trying to buy 1 share for XYZ. If the customer's balance is $10, then the system cannot record the purchase of 1 shares at $15.

Each menu contains a set of tasks. Many of which, should be processed in an "all or nothing" fashion. The tasks are described as following:

Finally, you can develop your project on your local machine, but we will only test your code on class3. So even though your code works locally, you should make sure that it also works on the class3 server. **It is your responsibility to submit code that works**.

What to Submit

The second and final milestone should contain, in addition to the SQL part, the Java code, and the driver.

You are required to turn in three files (use team01 to name your files if you are in group 1):

- team01.tar or team01.zip:
  A packed or compressed file that contains all program source files. You could also hand in team01.java if that's the only source file you have.
- team01.sql:
  A PostgreSQL script file that contains the SQL code (e.g., create the database, define the trigger, stored functions and procedures).
- team01-report.doc:
  A user's manual for the system, the system's limitations (e.g., missing or non-implemented functionality) and the possibilities for improvements.

How to submit your project

- Email your Git commit ID and the full web link to your Git repository to cs1555-staff with all team members CC'd. For this milestone, each group can **only submit once**. In addition to submitting the Git commit ID via email to **cs1555-staff with all team members CC'd**, you must **add the TAs as collaborators** (Github usernames: ralseghayer, nasrinklt, and glu99331) to the Github repository. Make sure the GitHub repository is **private**. In your GitHub repository you are required to have the **three files specified in What to Submit**.
- Upload the three submission files of your project through the Web-based submission interface you have used for previous Assignments.Note: **Only one team member** should submit the files through the Web-base submission interface since this is a group assignment.
- Submit your files by the due date **(8:00pm, Apr. 24, 2021). There is no late submission.**
- **It is your responsibility to make sure the project was properly submitted.**

Group Rules

- It is expected that all members of a group will be involved in all aspects of the project development. Division of labor to data engineering (db component) and software engineering (JAVA component) is not acceptable since each member of the group will be evaluated on both components.
- The work in this assignment is to be done *independently* by each group. Discussions with other groups on the assignment should be limited to understanding the statement of the problem. Cheating in any way, including giving your work to someone else will result in an F for the course and a report to the appropriate University authority.

Grading

The project will be graded on correctness, robustness (error-checking, i.e., it produces user-friendly and meaningful error messages) and readability. You will not be graded on efficient code with respect to speed although bad programming will certainly lead to incorrect programs. Programs that fail to compile or run or connect to the database server earn zero and no partial points.

Enjoy your class project!