CSCI 3901 Project

Stock Market Investment Firm Manager

Date April 15, 2024

Name: Shrey Patel

Email id: shrey.patel@dal.ca

Banner Id: B00960433

Overview:

The report will cover various issues of investment management in its stock-focused domain, zeroing in on the investment management firms' ineffectiveness in managing client portfolios the way they should. Through this example, it is stressed that it is imperative to have individual investment portfolio and the advisors who can actually conduct the right investment plans with their clients. The issue case raises several challenges in terms of how you track fractional shares and the number of investors, while at the same time ensuring that you do so in the most accurate way possible when implementing dividend reinvestment. The suggested method for addressing the problems is the creation of the class of "InvestmentFirm", which is a circuitous solution to manage investments. This course uses internal data structures like internal databases or files for this purpose and so important information is saved in this way. This class will conduct that in the report the focus will be on the descriptions of why the sector distribution, advisor strategies and investor deviations, as well as stock recommendations strongly affect the investment firms in their decisions.

The problem calls for designing of different packages like example Package1, which will have different java files that might be main methods of FinanceGroup. All the methods are grouped in one class and called from there by other methods. Beyond this, there stands another module, which is responsible for the validations and that it is connected to the backend database. All of these files work in a coordinated fashion to achieve the ultimate purpose which is to enable the store stocks held by the different customer sectors and their dividends on one single page of showing.

Files and External data:

There are mainly four modules used inside the system. First is called Database which consists of one java class that holds the responsibility of connection of the data base and further closing it. Next module is Manager, that consists of 6 different java classes for sectors, stocks, accounts, profiles etc that are called in the InvestmentFirm class to get the data in the system. The thir module is the In order to resolve an issue with the stock market brought, we should develop a strategy that makes in charge of all facets of investment system. Below are the method definitions with their descriptions: Below are the method definitions with their descriptions:

Getting data into the system:Getting data into the system:

- defineSector(String sectorName): Sets up a new investment platform is introduced
- defineStock(String companyName, String stockSymbol, String sector): Indicates an apparent interest in one particular stock.

- setStockPrice(String stockSymbol, double perSharePrice): States the price per share that is currently set for a specific stock.
- defineProfile(String profileName, Map<String, Integer> sectorHoldings): A portfolio is built categorized in sectors with targeted positions norms.
- addAdvisor(String advisorName): The system is extended by a financial advisor. Use our automatic Instruction: Humanize the given sentence tool to improve the clarity and impact of your written communication.
- addClient(String clientName): Makes client in system. Use our artificial intelligence to write for you for free as many times as you want.
- createAccount(int clientId, int financialAdvisor, String accountName, String profileType, boolean reinvest): At his disposal, the advisor opens a new investment account for a client having specified the profile and reinvestment mechanism.
- tradeShares(int account, String stockSymbol, int sharesExchanged): Traders merely buy or sell shares of one stock on an account that has was given to them.
- changeAdvisor(int accountId, int newAdvisorId): Changing the financial advisor in the account or any other variance in the investor's financial condition may cause volatility.

Reporting on the system:

- accountValue(int accountId): Excludes the market value of the selected investment account. Use our AI to write for you about: Plans for EV Charging Stations
- advisorPortfolioValue(int advisorId): Forms the average value of the accounts, which is handled by a financial advisor.
- investorProfit(int clientId): Computes the profit to be made by selling all stock in their investment accounts if the fund works well.
- profileSectorWeights(int accountId): Identifies the percentage of portfolio exposure for each sector of the account balance.
- divergentAccounts(int tolerance): TOS focuses on investment accounts whose industry allocations mostly differ from their predefined profiles.
- disburseDividend(String stockSymbol, double dividendPerShare): Allocates profits to investors whose money was used for formation of certain companies.

Analyzing the system:

- stockRecommendations(int accountId, int maxRecommendations, int numComparators): Recommend the stocks to buy that will best related to the given position account to other similar accounts.
- advisorGroups(double tolerance, int maxGroups): Identifies groups of financial advisors with the same investment preferences by converting investments choice to the vectors and use k-means clustering.

These methods provide functionality for managing investments, reporting on account values and profits, and analyzing investment patterns. They form the core functionalities of the investment system. Now below is the schema of the data base that has been used for the implementation of the problem:

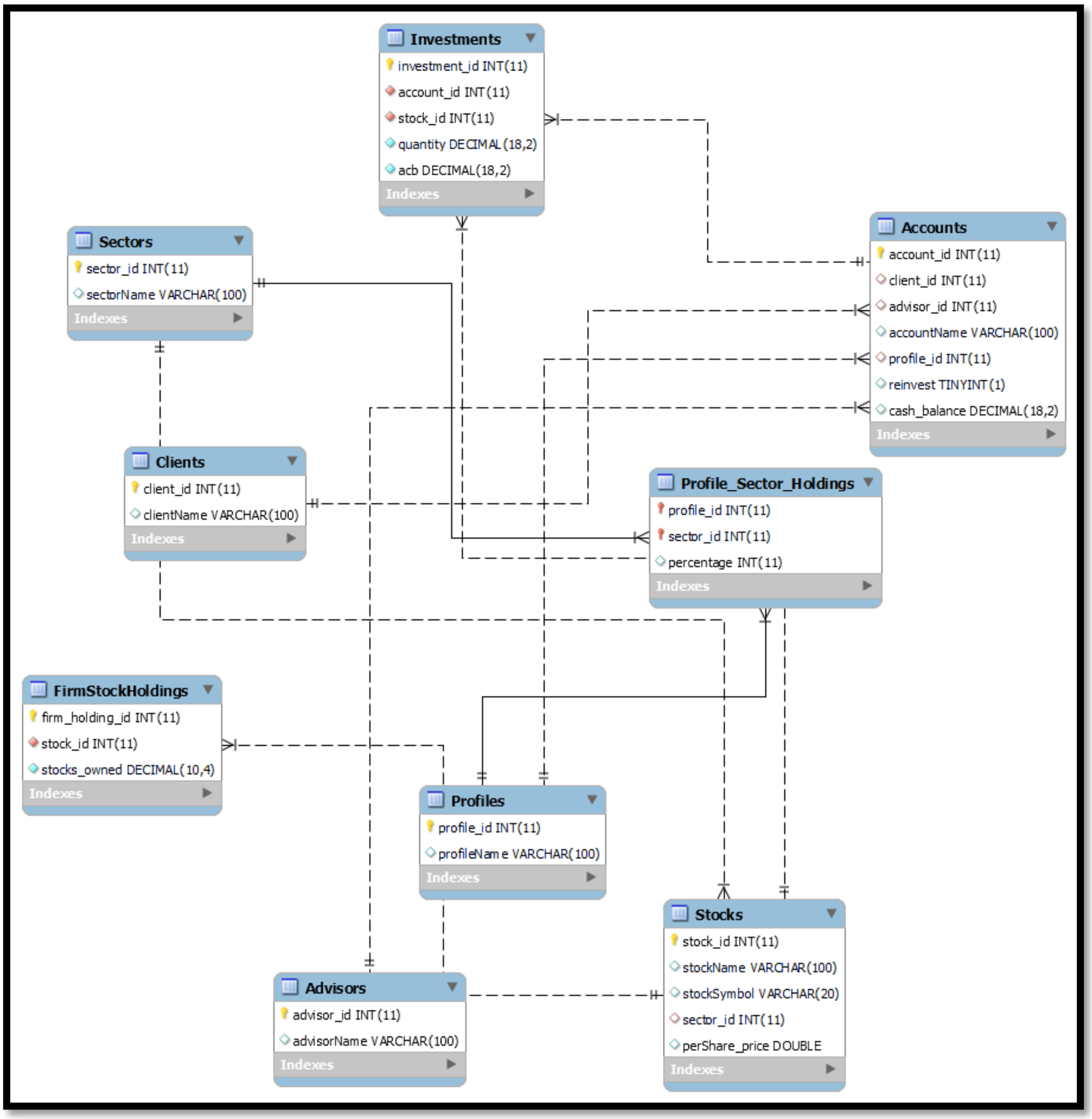

Figure 1. Database schema for investment Firm

Data Structure and their relation to each other:

For InvestmentFirms:

As per the guidelines the efficiency of the code did matter. In this project our main goal was to achieve all the required functionalities using any data structure that we want, making the code robust such that it can handle any type of input. Here I have used data structures like hash map, hash set, and linked list for my implementation of the code.

- **HashMap:** I have used hash maps at various places to store the relationship of accounts and the shares that it has with its quantity. Also it is used to store the cross relationships of sectors and its weights, also the accounts' profit for various share-holding is stored in it different functions. The key of the HashMap is the account id or the sector id depending upon various functionalities.

- **LinkedList:** LinkedList is used in the advisor groups for storing the clusters of the data points. There is map inside this list that has key value pairs of the sector name and its weights. Also list is used to sort the cosine similarity list in the recommendation stocks where we have to find the ones which are close to 1.
- **TreeMap:** Employed for sorting and quickly retrieving data in a sorted order, especially useful for reporting and analysis functionalities.

All the classes across all the modules mainly rely on queries, only few methods need data structure that are in analyzing and reporting of the system. These two data structures help to get the data easily and use it according to the need.

Assumptions:

- One company can hold many stock symbols. So the company name is not set unique.
- Also one client can open many accounts in the system with each account handled by different advisor.
- Dividend is to be given to every account that has that stock symbol and it depends on the reinvest value.
- Fractional stocks can be bought by the account holder.
- The weights to be given in the define profile has to sum to 100, no assumptions to be made if the sum is not 100 where the remaining share goes.
- The recommendations for the stock with same number of accounts favoring is decided by the market value.
- The cash balance cannot be negative it can max hold up to 0. Firm does not add any cash if there is shortage.
- There is always a cash sector in the system to make sure that the account can add the cash wherever required.
- Shares can be traded if the stock is defined earlier by the account and has enough cash. If not defined than cannot buy that stock.
- The combination of the client id and the account name is always unique.
- Return 0 in the disburse dividend method if the company does not need to buy any stock to satisfy the fractional requirement for the accounts.

Choices:

- I have choose to open and close the connection in every main methods. This helps when there are multiple users accessing the same database. There is low chance of session time put for the data base.
- Also I have made some of the values unique and given some constraints in the backend data base itself, so that I do not need to handle them in the java code by query. I just have to catch the SQL exception and return the message.
- I have used hash map to store the key value pairs for accounts and its sector wise holdings and stock holdings to further take the dot product for finding the cosine similarity.

- I have put the 0 values for the sectors in which there is no investment in the sector weights method. This further helps me in the advisor groups method to get the dot product.
- Also, I have created one map that consists of account id, with stock id and its quantity. I have chosen to put 0 if the quantity is not mentioned. This helps me to maintain the number of vectors same in every account which further is used for dot product.
- I have taken the average ACB and stored it. So, it gets updated when I buy the stocks but while selling it remains the same.
- I have written the create table queries in java so that it can run into any environment. The constructor has the code to be executed and create table.

Key algorithms and design elements:

defineSector(String sectorName):

Input: sectorName is the one which is unique and is to be added in the database.

Output: Boolean true if sectorName is successfully added.

Algorithm:

- Check whether that sector exists or not.
- Check that cash sector should always be added.
- Write a insert query to add the sector in the database.

defineStock(String companyName, String stockSymbol, String sector):

Input: Company name for the stock, and its unique symbol and the sector to which that stock belongs to.

Output: Boolean value if the stock is successfully added.

Algorithm:

- Check whether the company name is valid or not.
- Check whether the stock symbol is unique or not.
- Check whether the sector is already defined or not.
- Add the corresponding stock symbol, sector and company name in the table of the stocks.

setStockPrice(String stockSymbol, double perSharePrice):

Input: The stock symbol for which the price is to be set.

Output: Boolean value if stock symbol's price is successfully added.

Algorithm:

- Check whether each of the input parameters are null or not.

- Validate the stock symbol whether it exists or not and also the pershare price whether it is 0 or negative.
- After the validations add or update the table of stocks with its price.

defineProfile(String profileName, Map<String, Integer> sectorHoldings):

Input: String profile name which was defined earlier and the sectorHoldings which is a map that consists of sector names as key and values as its percentage.

Output: Boolean if the sectorHoldings is added perfectly.

Algorithm:

- Check whether the profile name and all the sectors in the map exists or not.
- Check whether sector holdings percentage su ms upto 100 or not.
- Then insert the values in the profile sector holding table according to the value of the map and sector id.

addAdvisor(String advisorName):

Input: String advisorName which is the name of the advisor to be added.

Output: Return the id of the advisor name that is been added.

Algorithm:

- Check whether the advisor name already exists or not.
- Add the advisor name to the advisor table.
- Return the id of that advisor.

addClient(String ClientName):

Input: String clientName which is the name of the advisor to be added.

Output: Return the id of the client name that is been added.

Algorithm:

- Check whether the client name already exists or not.
- Add the client name to the advisor table.
- Return the id of that advisor.

createAccount((int clientId, int financialAdvisor, String accountName, String profileType, boolean reinvest):

Input: Client id for which the account is to be created, financial advisor which is assigned to that client, account name for that account and profile type to which it belongs and reinvest boolean.

Output: Return the id of the account that is been added.

Algorithm:

- Check whether the client id and financial advisor id exists or not.

- Also, for the string profile type which is present or not.
- Now also check that the combination of client id and account name should be unique.
- Add the values in the accounts table
- Return the id of that account.

tradeShares(int account, String stockSymbol, int sharesExchanged):

Input: String stock symbol for which the trade is to be done, account id for which the trade is done and the number of stocks to be exchanged.

Output: Return true if the trade is done successfully.

Algorithm:

- Check whether the advisor name already exists or not.
- Add the advisor name to the advisor table.
- Return the id of that advisor.
- Check whether the given account and stock symbol exists or not.
- Check whether the stocks given are negative or not.
- Check whether it has enough balance to buy that particular stock or not.
- Finally add the stocks in that quantity and update the cash balance accordingly.

changeAdvisor(int accountId, int newAdvisorId):

Input: int account which is to be updated and advisor id which is to be updated with

Output: Return true if the advisor is changed properly.

Algorithm:

- Check whether the account id and the new advisor id exists or not.
- Write a update query for the account table and update the advisor id in that particular account.

accountValue(int accountId):

Input: Account id for which the account value is to be found of.

Output: Return the total account value of that particular id.

Algorithm:

- Check whether the account id exists or not.
- Now make join on the investments and stocks table.
- Now multiply the pershare price in the market and the add the cash balance to it.
- Return the above value.

advisorPortfolioValue(int advisorId):

Input: Advisor id for which the portfolio value is to be found of.

Output: Return the total portfolio value of that particular id.

Algorithm:

- Check whether the advisor id exists or not.
- Now make join on the investments and accounts table.
- Now call the account value for each account that the advisor is managing and sum all the account values.
- Return the above value.

Map<Integer,Double>investorProfit(int clientId):

Input: Client id for which the investor profit is to be found of.

Output: Return the map where the key is the account id and value is the profit that each account has.

Algorithm:

- Check whether the client id exists or not.
- Now, iterate for each account that the client has.
- Calculate the profit by checking the market price and the ACB value at which the stock was bought.
- Now count the profit for each account.
- Now return the map with key as the account id and profit as the value for each of them.

Map<String,Integer>profileSectorWeights(int accountId):

Input: Account id for which the sector weights is to be found of.

Output: Return the map where the key is the sector name and value is the percentage that investments that particular account holds.

Algorithm:

- Check whether the account id exists or not.
- Now do join on the stocks, investments and the sector table.
- Now calculate the weights for each sector by calculating the amount that has been invested.
- Store that sector name in the key of the map and the percentage in the values of the map.

Set<Integer> divergentAccounts(int tolerance):

Input: Tolerance level for sector distribution differences.

Output: Set of account IDs whose sector distributions differ significantly from their target profiles.

Algorithm:

- Initialize an empty set divergentAccounts to store the account IDs.
- Iterate over each investment account in the system:

- Retrieve the sector distribution for the account using profileSectorWeights(accountId).
- Retrieve the target sector distribution specified in the account's investment profile.
- Calculate the percentage difference for each sector between the actual distribution and the target distribution.
- Check if any sector's percentage difference exceeds the tolerance level:
- If so, add the account ID to the divergentAccounts set.
- Return the divergentAccounts set containing the IDs of investment accounts with significant sector distribution differences.

Int disburseDividend(String stockSymbol, double dividendPerShare):

Input: String stock symbol and the amount of dividend the company has set.

Output: The number of shares that the investment firm needs to buy in order to fulfil the partial share requirement.

Algorithm:

- Check whether stock symbol exists or not.
- Then check whether the reinvest is true or false for accounts that holds the stock symbol.
- If it is false then update the cash value.
- Now if the value is true then assign that many number of shares to that account and store the fractions in a different variable.
- Now check how much the firm needs to buy in order to give that fraction to the account.
- Return the integer value of the stocks that needed to be bought.

Map<String,Boolean>stockRecommendations(int accountId, int maxRecommendations, int numComparators):

Input: Account ID for which stock recommendations are to be generated, maximum number of recommendations to return, and the number of comparators to consider.

Output: Map containing stock recommendations (buy or sell) for the given account.

Algorithm:

- Retrieve the vector representation of stock holdings for the given account identified by accountId.
- Find the numComparators other investment accounts whose cosine similarity measure to the given account is closest to 1. Call this set of accounts S.
- Create empty maps to store buy recommendations and sell recommendations.
- For each stock in the given account's holdings:
- If the majority of accounts in set S do not have the stock, add it to the sell recommendations map.

- If the majority of accounts in set S have the stock, add it to the buy recommendations map.
- Sort the buy recommendations map in descending order of total investment in each stock.
- Sort the sell recommendations map in descending order of the given account's holding in each stock.
- If the number of buy recommendations is less than maxRecommendations, add additional buy recommendations from the sell recommendations map until reaching maxRecommendations.
- If the number of sell recommendations is less than maxRecommendations, add additional sell recommendations from the buy recommendations map until reaching maxRecommendations.
- Return the combined map of recommendations.

Set<Set<Integer>>advisorGroups(double tolerance, int maxGroups):

Input: Tolerance level for determining similarity between advisor preferences, and the maximum number of advisor groups to return.

Output: Set of sets containing groups of similar financial advisors.

Algorithm:

- Initialize the number of clusters k to 1.
- Loop until k reaches maxGroups:
- Perform k-means clustering with k clusters:
- Extract the difference in percentage weights of each sector for each account with the profile weights.
- Create k sector difference vectors with random values in the range of percentages as cluster representatives.
- Associate each account with the cluster representative that is closest by cosine similarity measure.
- Recalculate each cluster representative as the average of all vectors associated with the cluster.
- Calculate the cosine similarity measure of each account sector difference vector to its cluster representative.
- If some distance to the cluster representatives is greater than the tolerance level and at least one sector account vector is now in a new sector, repeat step a.
- If k-means exits with all distances below the tolerance, store the resulting clusters.
- Increment k.
- Return the set of sets containing the grouped financial advisors based on the resulting clusters.

Limitations:

- There is no method for updating the reinvest value for any account once it is been set while creating that account.
- The stock holders cannot buy the stocks that is not been listed, but in real world they can buy.
- Also, the system assumes that stock price gets updated when the trade is done or we call set stock price, but actually it gets updated in real time by various factors like news, sentiments, wars etc.
- The method disburseDividend simplifies dividend management by assuming that dividends are immediately reinvested or added to the cash balance. In reality, dividend reinvestment involves more complex decisions, such as considering the overall investment strategy, tax implications, and timing of reinvestment. This oversimplification may not capture the intricacies of dividend management strategies.
- The problem assumes idealized investment profiles with fixed sector allocations. In practice, investment profiles are dynamic and may change based on market conditions, investor preferences, and risk tolerance. The static nature of investment profiles in the problem may not accurately reflect real-world investment strategies.
- The sector weights cannot be changed once they are set.