University for the Creative Arts

BERLIN SCHOOL OF BUSINESS & INNOVATION

**Essay / Assignment Title: Database system design for a stock exchange market**

**Programme title: MSc Data analytics**

**Name: Naseef kavate**

**Year: 2023**

# TABLE OF CONTENTS

## Statement of compliance with academic ethics and the avoidance of plagiarism

I honestly declare that this dissertation is entirely my own work and none of its part has been copied from printed or electronic sources, translated from foreign sources and reproduced from essays of other researchers or students. Wherever I have been based on ideas or other people texts I clearly declare it through the good use of references following academic ethics.

(In the case that is proved that part of the essay does not constitute an original work, but a copy of an already published essay or from another source, the student will be expelled permanently from the postgraduate program).

Name and Surname (Capital letters): Naseef kavate

.....

Date: 30/05/2023

# INTRODUCTION

As a platform for traders to engage in the financial markets, stock exchange platforms now serve a vital role in enabling the buying and selling of assets. Behind the scenes, dependable and well-designed database systems play a key role in the smooth operation of a stock exchange. For an efficient working of the stock exchange market the client data, stock data and the transaction done need to be well organized. There comes the role of a database management system, in which it ensures the data integrity, scalability, security and more features. These data need to be analyzed for the smooth functioning of the system, which can be done by running queries (SQL).

This assignment aims to build an efficient database management system using MySQL workbench, in which the tables are created for storing the data and keeping track of the financial data. The fundamentals of database architecture will be covered, along with the effects of the CAP theorem on distributed systems and the attributes of MySQL, a well-known database management system. With this information, we will be better equipped to create scalable and effective databases, weigh the pros and disadvantages of consistency and availability, and use MySQL to create dependable database solutions.

# CHAPTER ONE: AN OVERVIEW OF A STOCK MARKET DATABASE MANAGEMENT SYSTEM

Efficient data storage and retrieval techniques are crucial for designing a robust database system for a stock exchange market. These strategies require large amounts of real-time data to be processed quickly in order to make informed decisions. Therefore, it is essential that the underlying database system can handle such demands without compromising performance or accuracy. One approach to achieving this goal is through the use of distributed databases which store and process data across multiple nodes rather than relying on a single central server. This technique not only improves overall throughput but also provides fault tolerance by ensuring that failure of one node does not lead to complete system failure.

**Designing a Solid Foundation**:

The initial phase of designing a database system involves identifying what all data are required for the exchange platform. In the market clients or customers need to register first, so a table is required to store their data. The list of stock needs to be listed from where the clients can look up the prices and buy or sell the ones they needed. The stock prices should be in such a way that clients are able to track the performance of each stock. The clients are able to buy or sell the stock in the platform designed, so an history of transaction done need to be kept to track the amount by which they have done trading. The price history of each stock needs to be tracked to update the real time price of the stock, whenever the price of a stock changes the current price should be updated in a log table. The quantities of stocks held by each client need to be stored which allows the customers to keep track of the quantities of stock they have and to take decision whether to buy or sell. To monitor the stock which customers would like to monitor can be tracked using a table for that.

**Managing Transactions and Account Balances**:

The system was designed to handle various types of transactions, including buy and sell orders, ensuring accurate execution and updating of account balances. Advanced validation mechanisms were implemented to verify account balances before executing transactions, preventing unauthorized trades due to insufficient funds or insufficient quantity of the stock.

**Describing the entities and attributes:**

After identifying the key entities which is to be included in the database it needs to be well organized with the necessary attributes to obtain a meaningful table. Each entity is identified by a unique key which is called the primary key. The entities and attributes are given below;

1. stocks_list: Each stock listed in the market need to be identified by a unique key 'ID' for each stock. It has other attributes 'symbol' which is unique character string, 'company_name' mentioning the company selling their stocks, 'Current_price' which keep record of the live price of the stock, 'Open_price' giving the opening price of the stock in the market, 'High_price' giving the highest price of that stock in a particular day and 'low_price' giving the lowest price of stock in that day. 'Quantity' giving the number of stocks currently available in the market.
2. Clients: The entity where the customer's data are stored. Each customer is identified by an 'ID'. The other attributes are 'Name' which keeps the names of clients, 'Email' which stores the mail address of the client and 'Account_balance' which stores the balance of the client that can be used for trading.
3. Stock_transaction: The entity where the transaction details of the customers are stored. Each transaction is identified by an 'ID'. It contains the attributes 'order_type' mentioning whether buy or sell of the stock, 'Quantity' which stores the amount of stock traded, 'order_date' storing the current date, 'Amount' storing the amount by which the customer trade.
4. Stock_price_history: Records the old price of each stock. Each stock's price history has a unique 'ID'. It also includes the attribute 'Date' to store the date at which price is changed.
5. Client_stock_portfolio: The table which records the quantity of stocks held by a customer. Each portfolio is identified by an 'ID' which is unique. The attributes are 'quantity' which mentions the quantity of stock held and 'stock_value' which gives the total value of the stock held by a client.
6. Account_balances: The entity which keeps log of the balance amount of each customer who has done trading in the market. The balance amount of a client is identified by a unique key 'ID', the other attributes are 'balance_amount' which records the client's balance after each trade and 'balance_date' column to keep track of the date.

**Identifying the relationships**:

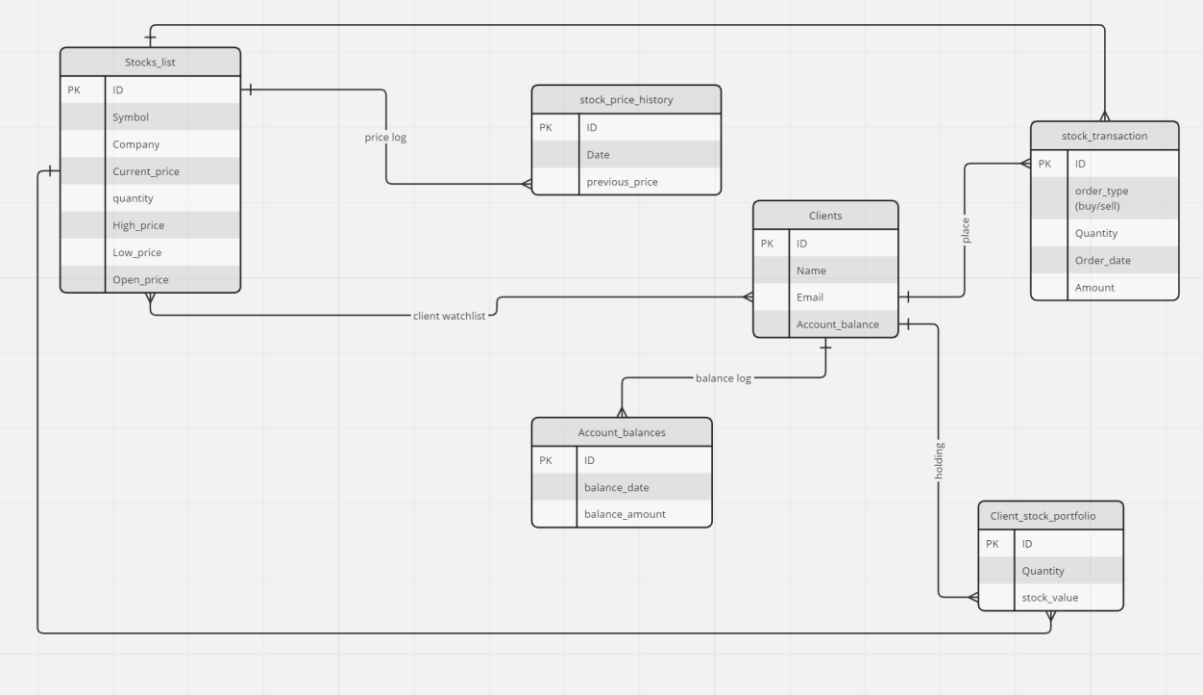
After identifying the entities, the relationships between them need to be established. These relationships include:

- One-to-many relationship between stocks_list and stock_price_history: Each stock can have multiple price history entries, while each price history entry is associated with a single stock. So, an attribute ‘stock_ID’ can be included in stock_price_history table, which is called as the foreign key which act as the link between both tables.
- One-to-many relationship between stocks_list and stock_transaction: Each stock can be associated with multiple transactions, while each transaction is linked to a single stock. So, ‘stock_ID’ is a foreign key in stock_transaction table.
- One-to-many relationship between stocks_list and client_stock_portfolio: Each stock can be in multiple stock portfolio, while each portfolio is linked to a single stock. So, ‘stock_ID’ is a foreign key in client_stock_portfolio table.
- Many-to-many relationship between stocks_list and clients: Each stock can be in multiple watchlist of client’s and each client’s watchlist can contain multiple stocks. Create a separate table containing two foreign keys’ ‘stock_ID’ and ‘watchlist_ID’.
- One-to-many relationship between clients and stock_transaction: Each client can place multiple orders, while each order is associated with a single client. The ‘client_ID’ will be the associated foreign key in stock_transaction table.
- One-to-many relationship between clients and account_balances: Each client can have multiple account balance entries, while each balance entry is associated with a single client. The ‘client_ID’ will be the foreign key in account_balances table.
- One-to-many relationship between Clients and Watchlist: Each client can have multiple entries in their watchlist, while each watchlist entry is linked to a single client. The ‘client_ID’ act as the link and is the foreign key in watchlist table.
- One-to-many relationship between clients and client_stock_portfolio: Each client can have multiple portfolios for each stock, while each portfolio is associated with a single client. The ‘client_ID’ act as the foreign key and can be included in attributes of client_stock_portfolio.

**Drawing the entity relationship diagram:**

A database's relationships between entities are shown visually in entity relationship diagrams, or ER diagrams. It helps to clarify the logical structure of the database and its relationships by showing how entities are linked to one another through associations.

Using the miro online tool, it was able to make an ER diagram as following;




The ‘PK’ here refers to primary key which is unique for each entity. One-to-many relationship between entities is shown by;

Many-to-many relationship is shown by;

# CHAPTER TWO: DESIGNING THE FOUNDATION-CREATING RELATIONAL DBMS TABLES FOR A STOCK EXCHANGE DATABASE

After having the entities and relationships between them, a solid database is to be designed in a relational database management system (RDBMS) for data integrity and security. Here the financial data is to be secured and need to handle large volumes of data. A highly efficient DBMS that can be used here is MySQL which ensures data integrity and consistency. The query language SQL can be used to get valuable insight from database. This system has high reliability and flexibility making it one of the popular DBMS for companies to handle secured data.

**Creating a new schema in MySQL:**

After connecting to a server, as the first step create a new schema in MySQL, where we can insert new tables, columns and mention the relationships among them.

```
1    CREATE SCHEMA `stock_exchange_market` ;
```

After creating a schema, the entities or tables is to be created.

Creating ‘stocks_list’ table;

```
1  CREATE TABLE `stock_exchange_market`.`stocks_list` (
2    `ID` INT NOT NULL AUTO_INCREMENT,
3    `symbol` VARCHAR(45) NULL,
4    `company` VARCHAR(45) NULL,
5    `current_price` INT NULL,
6    `open_price` INT NULL,
7    `high_price` INT NULL,
8    `low_price` INT NULL,
9    `quantity` INT NULL,
10   PRIMARY KEY (`ID`),
11   UNIQUE INDEX `ID_UNIQUE` (`ID` ASC) VISIBLE);
12
```

The column ‘stocks_list.ID’ is the primary key here in the table. ‘INT’ data type is used to store the integer value for that column and ‘VARCHAR(45)’ data type is used to store the variable character length in MySQL. The primary key is set to be not null, auto increment and unique.

Creating ‘clients’ table;

```
CREATE TABLE `stock_exchange_market`.`clients` (
  `ID` INT NOT NULL AUTO_INCREMENT,
  `Name` VARCHAR(45) NULL,
  `Email` VARCHAR(45) NULL,
  `Account_balance` INT NULL,
  PRIMARY KEY (`ID`),
  UNIQUE INDEX `ID_UNIQUE` (`ID` ASC) VISIBLE);

```

The ‘clients.ID’ column is the primary key which is unique for each client.

Creating ‘stock_transaction’ table;

```
CREATE TABLE `stock_exchange_market`.`stock_transaction` (
  `ID` INT NOT NULL AUTO_INCREMENT,
  `order_type` VARCHAR(45) NULL,
  `quantity` INT NULL,
  `order_date` DATE NULL,
  `amount` INT NULL,
  `stock_ID` INT NULL,
  `client_ID` INT NULL,
  PRIMARY KEY (`ID`),
  UNIQUE INDEX `ID_UNIQUE` (`ID` ASC) VISIBLE,
  INDEX `client_ordered_idx` (`client_ID` ASC) VISIBLE,
  INDEX `stock_ordered_idx` (`stock_ID` ASC) VISIBLE,
  CONSTRAINT `client_ordered`
    FOREIGN KEY (`client_ID`)
    REFERENCES `stock_exchange_market`.`clients` (`ID`)
    ON DELETE NO ACTION
    ON UPDATE NO ACTION,
  CONSTRAINT `stock_ordered`
    FOREIGN KEY (`stock_ID`)
    REFERENCES `stock_exchange_market`.`stocks_list` (`ID`)
    ON DELETE NO ACTION
    ON UPDATE NO ACTION);
```

The data type for the column ‘order_date’ is ‘DATE’, which is used to store the date of transaction. ‘stock_ID’ is the foreign key which is referenced to ‘ID’ column in ‘stocks_list’ table and ‘client_ID’ is the foreign key referenced to ‘ID’ column of ‘clients’ table. This is done using the tab ‘foreign key’ in table properties of ‘stock_transaction’.

Creating ‘stock_price_history’ table;

```
CREATE TABLE `stock_exchange_market`.`stock_price_history` (
  `ID` INT NOT NULL AUTO_INCREMENT,
  `Date` DATE NULL,
  `previous_price` INT NULL,
  `stock_ID` INT NULL,
  PRIMARY KEY (`ID`),
  UNIQUE INDEX `ID_UNIQUE` (`ID` ASC) VISIBLE,
  INDEX `stock_log_idx` (`stock_ID` ASC) VISIBLE,
  CONSTRAINT `stock_log`
    FOREIGN KEY (`stock_ID`)
    REFERENCES `stock_exchange_market`.`stocks_list` (`ID`)
    ON DELETE NO ACTION
    ON UPDATE NO ACTION);

```

Here the ‘stock_ID’ column is the foreign key which is referenced to ‘ID’ column of ‘stocks_list’ table.

Creating ‘account_balance’ table;

```
CREATE TABLE `stock_exchange_market`.`account_balances` (
  `ID` INT NOT NULL AUTO_INCREMENT,
  `balance_date` DATE NULL,
  `balance_amount` VARCHAR(45) NULL,
  `client_ID` INT NULL,
  PRIMARY KEY (`ID`),
  UNIQUE INDEX `ID_UNIQUE` (`ID` ASC) VISIBLE,
  INDEX `client_balance_idx` (`client_ID` ASC) VISIBLE,
  CONSTRAINT `client_balance`
    FOREIGN KEY (`client_ID`)
    REFERENCES `stock_exchange_market`.`clients` (`ID`)
    ON DELETE NO ACTION
    ON UPDATE NO ACTION);

```

The ‘client_ID’ is the common link between tables ‘account_balances’ and ‘clients’.

Creating ‘client_stock_portfolio’ table;

```
CREATE TABLE `stock_exchange_market`.`client_stock_portfolio` (
  `ID` INT NOT NULL AUTO_INCREMENT,
  `quantity` INT NULL,
  `stock_value` INT NULL,
  `client_ID` INT NULL,
  `stock_ID` INT NULL,
  PRIMARY KEY (`ID`),
  UNIQUE INDEX `ID_UNIQUE` (`ID` ASC) VISIBLE,
  INDEX `client_portfolio_idx` (`client_ID` ASC) VISIBLE,
  INDEX `stock_portfolio_idx` (`stock_ID` ASC) VISIBLE,
  CONSTRAINT `client_portfolio`
    FOREIGN KEY (`client_ID`)
    REFERENCES `stock_exchange_market`.`clients` (`ID`)
    ON DELETE NO ACTION
    ON UPDATE NO ACTION,
  CONSTRAINT `stock_portfolio`
    FOREIGN KEY (`stock_ID`)
    REFERENCES `stock_exchange_market`.`stocks_list` (`ID`)
    ON DELETE NO ACTION
    ON UPDATE NO ACTION);
```

The foreign keys in this table are ‘stock_ID’ and ‘client_ID’.

Creating ‘stock_watchlist’ table;

```
CREATE TABLE `stock_exchange_market`.`stock_watchlist` (
  `stock_ID` INT NOT NULL,
  `client_ID` INT NOT NULL,
  INDEX `client_watchlisted_idx` (`client_ID` ASC) VISIBLE,
  INDEX `stock_watchlisted_idx` (`stock_ID` ASC) VISIBLE,
  CONSTRAINT `client_watchlisted`
    FOREIGN KEY (`client_ID`)
    REFERENCES `stock_exchange_market`.`clients` (`ID`)
    ON DELETE NO ACTION
    ON UPDATE NO ACTION,
  CONSTRAINT `stock_watchlisted`
    FOREIGN KEY (`stock_ID`)
    REFERENCES `stock_exchange_market`.`stocks_list` (`ID`)
    ON DELETE NO ACTION
    ON UPDATE NO ACTION);

```

This junction table ‘stock_watchlist’ between ‘clients’ and ‘stocks_list’ can be used to store the data of client’s monitored stock. Here there is no primary key, both ‘client_ID’ and ‘stock_ID’ are foreign keys.

**Entering the data in tables;**

As the next step meaningful data are entered to the tables created.

10 client’s data are entered in the ‘clients’ table;

```
INSERT INTO `stock_exchange_market`.`clients` (`ID`, `Name`, `Email`, `Account_balance`) VALUES ('1', 'John', 'john12@gmail.com', '25000');
INSERT INTO `stock_exchange_market`.`clients` (`ID`, `Name`, `Email`, `Account_balance`) VALUES ('2', 'James', 'jamesbg@gmail.com', '22000');
INSERT INTO `stock_exchange_market`.`clients` (`ID`, `Name`, `Email`, `Account_balance`) VALUES ('3', 'Selena', 'sele34@gmail.com', '30000');
INSERT INTO `stock_exchange_market`.`clients` (`ID`, `Name`, `Email`, `Account_balance`) VALUES ('4', 'Sanjana', 'sanj435@gmail.com', '35000');
INSERT INTO `stock_exchange_market`.`clients` (`ID`, `Name`, `Email`, `Account_balance`) VALUES ('5', 'Farzeen', 'farzu23@yahoo.com', '38000');
INSERT INTO `stock_exchange_market`.`clients` (`ID`, `Name`, `Email`, `Account_balance`) VALUES ('6', 'Liya', 'liya218@hotmail.com', '23000');
INSERT INTO `stock_exchange_market`.`clients` (`ID`, `Name`, `Email`, `Account_balance`) VALUES ('7', 'Joe', 'joel435@gmail.com', '28000');
INSERT INTO `stock_exchange_market`.`clients` (`ID`, `Name`, `Email`, `Account_balance`) VALUES ('8', 'Rahees', 'rahees89@gmail.com', '40000');
INSERT INTO `stock_exchange_market`.`clients` (`ID`, `Name`, `Email`, `Account_balance`) VALUES ('9', 'Muhiz', 'muhiz888@gmail.com', '23500');
INSERT INTO `stock_exchange_market`.`clients` (`ID`, `Name`, `Email`, `Account_balance`) VALUES ('10', 'Leo', 'leo98@gmail.com', '32000');
```

The SQL query INSERT INTO is used here in MySQL to insert values in the columns. We get all the client's data from the query;

```
SELECT * FROM stock_exchange_market.clients;
```

| ID | Name | Email | Account_balance |
|---|---|---|---|
| 1 | John | john12@gmail.com | 25000 |
| 2 | James | jamesbg@gmail.com | 22000 |
| 3 | Selena | sele34@gmail.com | 30000 |
| 4 | Sanjana | sanj435@gmail.com | 35000 |
| 5 | Farzeen | farzu23@yahoo.com | 38000 |
| 6 | Liya | liya218@hotmail.com | 23000 |
| 7 | Joe | joel435@gmail.com | 28000 |
| 8 | Rahees | rahees89@gmail.com | 40000 |
| 9 | Muhiz | muhiz888@gmail.com | 23500 |
| 10 | Leo | leo98@gmail.com | 32000 |
| NULL | NULL | NULL | NULL |

Next the stock listed in the market are stored in the 'stocks_list' table, the data are entered.

```
INSERT INTO `stock_exchange_market`.`stocks_list` (`ID`, `symbol`, `company`, `current_price`, `open_price`, `high_price`, `low_price`, `quantity`) VALUES ('1', 'AAPL', 'Apple Inc.', '98', '102', '105', '80', '500');
INSERT INTO `stock_exchange_market`.`stocks_list` (`ID`, `symbol`, `company`, `current_price`, `open_price`, `high_price`, `low_price`, `quantity`) VALUES ('2', 'MSFT', 'Microsoft', '52', '47', '53', '45', '600');
INSERT INTO `stock_exchange_market`.`stocks_list` (`ID`, `symbol`, `company`, `current_price`, `open_price`, `high_price`, `low_price`, `quantity`) VALUES ('3', 'GOOG', 'Alphabet', '102', '95', '105', '90', '550');
INSERT INTO `stock_exchange_market`.`stocks_list` (`ID`, `symbol`, `company`, `current_price`, `open_price`, `high_price`, `low_price`, `quantity`) VALUES ('4', 'ORCL', 'Oracle', '39', '35', '40', '25', '980');
INSERT INTO `stock_exchange_market`.`stocks_list` (`ID`, `symbol`, `company`, `current_price`, `open_price`, `high_price`, `low_price`, `quantity`) VALUES ('5', 'FB', 'Facebook', '59', '65', '59', '50', '1000');
INSERT INTO `stock_exchange_market`.`stocks_list` (`ID`, `symbol`, `company`, `current_price`, `open_price`, `high_price`, `low_price`, `quantity`) VALUES ('6', 'JPM', 'JP Morgan', '86', '90', '91', '80', '650');
INSERT INTO `stock_exchange_market`.`stocks_list` (`ID`, `symbol`, `company`, `current_price`, `open_price`, `high_price`, `low_price`, `quantity`) VALUES ('7', 'INTC', 'Intel', '58', '46', '59', '47', '1100');
INSERT INTO `stock_exchange_market`.`stocks_list` (`ID`, `symbol`, `company`, `current_price`, `open_price`, `high_price`, `low_price`, `quantity`) VALUES ('8', 'MA', 'Mastercard', '88', '102', '103', '85', '700');
INSERT INTO `stock_exchange_market`.`stocks_list` (`ID`, `symbol`, `company`, `current_price`, `open_price`, `high_price`, `low_price`, `quantity`) VALUES ('9', 'CSCO', 'Cisco', '53', '50', '55', '53', '850');
INSERT INTO `stock_exchange_market`.`stocks_list` (`ID`, `symbol`, `company`, `current_price`, `open_price`, `high_price`, `low_price`, `quantity`) VALUES ('10', 'PEP', 'Pepsico', '65', '70', '68', '60', '900');
INSERT INTO `stock_exchange_market`.`stocks_list` (`ID`, `symbol`, `company`, `current_price`, `open_price`, `high_price`, `low_price`, `quantity`) VALUES ('11', 'KO', 'Cocacola', '76', '70', '76', '68', '1100');
INSERT INTO `stock_exchange_market`.`stocks_list` (`ID`, `symbol`, `company`, `current_price`, `open_price`, `high_price`, `low_price`, `quantity`) VALUES ('12', 'CAT', 'Caterpillar', '85', '98', '100', '85', '430');
```

The stock details are obtained in table using SQL as;

```
SELECT * FROM stocks_list;
```

We get the table

| ID | symbol | company | current_price | open_price | high_price | low_price | quantity |
|---|---|---|---|---|---|---|---|
| 1 | AAPL | Apple Inc. | 98 | 102 | 105 | 80 | 500 |
| 2 | MSFT | Microsoft | 52 | 47 | 53 | 45 | 600 |
| 3 | GOOG | Alphabet | 102 | 95 | 105 | 90 | 550 |
| 4 | ORCL | Oracle | 39 | 35 | 40 | 25 | 980 |
| 5 | FB | Facebook | 59 | 65 | 59 | 50 | 1000 |
| 6 | JPM | JP Morgan | 86 | 90 | 91 | 80 | 650 |
| 7 | INTC | Intel | 58 | 46 | 59 | 47 | 1100 |
| 8 | MA | Masterc... | 88 | 102 | 103 | 85 | 700 |
| 9 | CSCO | Cisco | 53 | 50 | 55 | 53 | 850 |
| 10 | PEP | Pepsico | 65 | 70 | 68 | 60 | 900 |
| 11 | KO | Cocacola | 76 | 70 | 76 | 68 | 1100 |
| 12 | CAT | Caterpillar | 85 | 98 | 100 | 85 | 430 |
| NULL | NULL | NULL | NULL | NULL | NULL | NULL | NULL |

**Stored procedure function**;

The data for ‘stock_transaction’ table, ‘client_stock_portfolio’ table, ‘account_balances’ table can be inserted by making a stored procedure function for buying and selling stocks by the client.

The stored procedure for buying stocks by clients is as following;

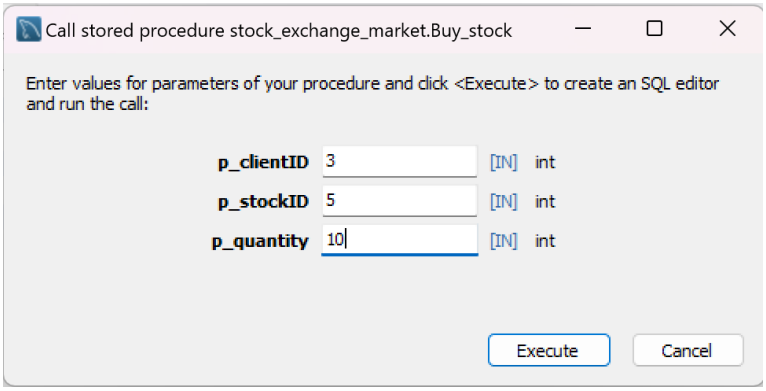
```
CREATE PROCEDURE `Buy_stock` (in p_clientID int, in p_stockID int, in p_quantity int)
BEGIN
    declare v_balance int;
    declare stockprice int;
    declare stockquantity int;
    declare c_qty int;
    select Account_balance into v_balance  from clients where clients.ID = p_clientID;
    select current_price into stockprice from stocks_list  where stocks_list.ID = p_stockID;
    select quantity into stockquantity from stocks_list where stocks_list.ID = p_stockID;
    select quantity into c_qty from client_stock_portfolio where client_stock_portfolio.client_ID = p_clientID and client_stock_portfolio.stock_ID = p_stockID;

    if p_quantity <= stockquantity then
        if v_balance >= (p_quantity * stockprice) then
            update clients set Account_balance = Account_balance - (p_quantity * stockprice) where clients.ID = p_clientID;
            update stocks_list set Quantity = Quantity - p_quantity where stocks_list.ID = p_stockID;
            insert into stock_transaction(order_type, quantity, order_date, amount, stock_ID, client_ID) values('Buy', p_quantity, now(), (p_quantity * stockprice), p_clientID, p_stockID);
            insert into account_balances(balance_date, balance_amount, client_ID) values(now(), (v_balance - (p_quantity * stockprice)), p_clientID);
        if c_qty is null then
            insert into client_stock_portfolio(quantity, stock_value, stock_ID, client_ID) values(p_quantity, (p_quantity * stockprice), p_stockID, p_clientID);
        else
            update client_stock_portfolio set quantity = quantity + p_quantity where client_stock_portfolio.client_ID = p_clientID and client_stock_portfolio.stock_ID = p_stockID;
            update client_stock_portfolio set stock_value = stock_value + (p_quantity * stockprice)
                where client_stock_portfolio.client_ID = p_clientID and client_stock_portfolio.stock_ID = p_stockID;
        end if;
        select 'Success' as result;
    else
        select 'Insufficient balance' as result;
    end if;
else
    select 'Insufficient quantity of stock available' as result;
end if;
END$$
```

In the above procedure function the ‘Buy_stock’ function takes input as client_ID, stock_ID and quantity to be bought. The current_price of stock, client’s account balance, the total quantity of the corresponding stock and the quantity of the corresponding stock in client’s portfolio is taken from the database using ‘SELECT’ queries.

The conditions are checked whether the quantity ordered is less than the available stock quantity and the client's account balance is sufficient to buy the stock. If the conditions are satisfied, the 'account balance' in 'client' table is updated by deducing the amount by which client purchased the stock, which is obtained by multiplying quantity and price of stock. The quantity of stock is updated in 'stocks_list' table by reducing the amount of stock bought. A new row is inserted in the table stock_transaction with the corresponding values, order_type as 'Buy' and create a new row for account_balances table to keep track of the client's balance amount.

If the quantity of a particular stock in the client's stock portfolio is not zero then UPDATE the quantity and stock value of that stock. If the quantity of that stock is zero in portfolio, then INSERT a new row for the stock and the client.

Running the stored procedure function will open a window;




The client with ID = 3 bought 10 stocks of ID = 5.

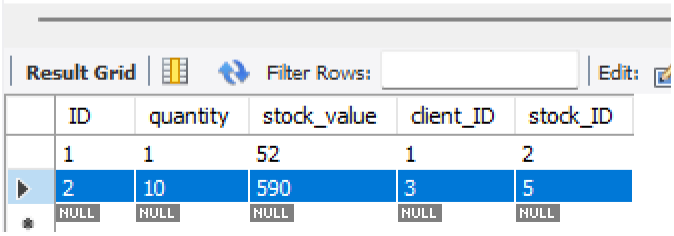
From the above table we have that client with ID = 3 is 'Selena' having an account balance of 30000. The stock with ID = 5 is FB having a current_price of 59.

After executing this procedure, the stock_transaction table is updated;

| ID | order_type | quantity | order_date | amount | stock_ID | client_ID |
|---|---|---|---|---|---|---|
| 1 | Buy | 3 | 2023-05-29 | 156 | 1 | 2 |
| 2 | Buy | 6 | 2023-05-29 | 312 | 1 | 2 |
| 3 | Sell | 5 | 2023-05-29 | 260 | 1 | 2 |
| 4 | Sell | 3 | 2023-05-29 | 156 | 1 | 2 |
| 5 | Buy | 3 | 2023-05-29 | 156 | 1 | 2 |
| 6 | Sell | 3 | 2023-05-29 | 156 | 1 | 2 |
| 7 | Buy | 10 | 2023-05-29 | 590 | 3 | 5 |
| NULL | NULL | NULL | NULL | NULL | NULL | NULL |

A portfolio will be created for client ID = 3 as;

```
1 •  SELECT * FROM client_stock_portfolio;
2
```

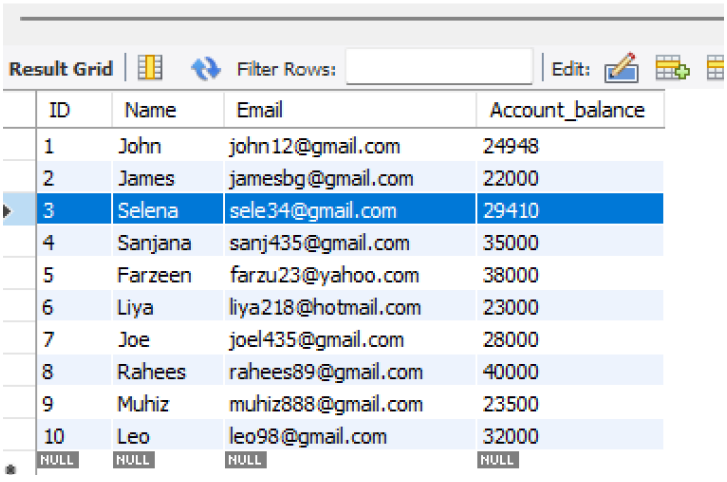


Result Grid | Filter Rows: | Edit:

| ID | quantity | stock_value | client_ID | stock_ID |
|---|---|---|---|---|
| 1 | 1 | 52 | 1 | 2 |
| 2 | 10 | 590 | 3 | 5 |
| NULL | NULL | NULL | NULL | NULL |

The account balance of the client ID = 3 will be updated from 25000 to 29410.

```
1 •  SELECT * FROM stock_exchange_market.clients;
```



Result Grid | Filter Rows: | Edit:

| ID | Name | Email | Account_balance |
|---|---|---|---|
| 1 | John | john12@gmail.com | 24948 |
| 2 | James | jamesbg@gmail.com | 22000 |
| 3 | Selena | sele34@gmail.com | 29410 |
| 4 | Sanjana | sanj435@gmail.com | 35000 |
| 5 | Farzeen | farzu23@yahoo.com | 38000 |
| 6 | Liya | liya218@hotmail.com | 23000 |
| 7 | Joe | joel435@gmail.com | 28000 |
| 8 | Rahees | rahees89@gmail.com | 40000 |
| 9 | Muhiz | muhiz888@gmail.com | 23500 |
| 10 | Leo | leo98@gmail.com | 32000 |
| NULL | NULL | NULL | NULL |

And the quantity of stock listed will be updated for ID = 3 from 1000 to 990.

```
1 •   SELECT * FROM stock_exchange_market.stocks_list;
```

Result Grid | Filter Rows: | Edit: | Export/Import:

| ID | symbol | company | current_price | open_price | high_price | low_price | quantity |
|---|---|---|---|---|---|---|---|
| 1 | AAPL | Apple Inc. | 98 | 102 | 105 | 80 | 500 |
| 2 | MSFT | Microsoft | 52 | 47 | 53 | 45 | 599 |
| 3 | GOOG | Alphabet | 102 | 95 | 105 | 90 | 550 |
| 4 | ORCL | Oracle | 39 | 35 | 40 | 25 | 980 |
| 5 | FB | Facebook | 59 | 65 | 59 | 50 | 990 |
| 6 | JPM | JP Morgan | 86 | 90 | 91 | 80 | 650 |
| 7 | INTC | Intel | 58 | 46 | 59 | 47 | 1100 |
| 8 | MA | Mastercard | 88 | 102 | 103 | 85 | 700 |
| 9 | CSCO | Cisco | 53 | 50 | 55 | 53 | 850 |
| 10 | PEP | Pepsico | 65 | 70 | 68 | 60 | 900 |
| 11 | KO | Cocacola | 76 | 70 | 76 | 68 | 1100 |
| 12 | CAT | Caterpillar | 85 | 98 | 100 | 85 | 430 |
| NULL | NULL | NULL | NULL | NULL | NULL | NULL | NULL |

The stored procedure function for selling a stock is as follows:

```
CREATE PROCEDURE `Sell_stock` (in s_clientID int, in s_stockID int, in s_quantity int )
BEGIN
    declare v_balance int;
    declare stockprice int;
    declare stockquantity int;
    declare c_qty int;

    select Account_balance into v_balance from clients where clients.ID = s_clientID;
    select current_price into stockprice from stocks_list where stocks_list.ID = s_stockID;
    select quantity into stockquantity from stocks_list where stocks_list.ID = s_stockID;
    select quantity into c_qty from client_stock_portfolio where client_stock_portfolio.client_ID = s_clientID and client_stock_portfolio.stock_ID = s_stockID;

    if s_quantity <= c_qty then
        update clients set Account_balance = Account_balance + (s_quantity * stockprice) where clients.ID = s_clientID;
        update stocks_list set Quantity = Quantity + s_quantity where stocks_list.ID = s_stockID;
        insert into stock_transaction(order_type, quantity, order_date, amount, stock_ID, client_ID) values('Sell', s_quantity, now(), (s_quantity * stockprice), s_clientID, s_stockID);
        insert into account_balances(balance_date, balance_amount, client_ID) values(now(), (v_balance + (s_quantity * stockprice)), s_clientID);

        if s_quantity = c_qty then
            delete from client_stock_portfolio where client_stock_portfolio.client_ID = s_clientID and client_stock_portfolio.stock_ID = s_stockID;
        else
        update client_stock_portfolio set quantity = quantity - s_quantity where client_stock_portfolio.client_ID = s_clientID and client_stock_portfolio.stock_ID = s_stockID;
        update client_stock_portfolio set stock_value = stock_value - (s_quantity * stockprice)
            where client_stock_portfolio.client_ID = s_clientID and client_stock_portfolio.stock_ID = s_stockID;
        end if;
        select 'success' as result;
    else
        select 'Insufficient quantity of stock' as result;
    end if;
END$$
```

Here in sell_stock function the condition that needed to be checked is whether the quantity of the stock that needed to be sold by the client is less than or equal to the quantity available in stock portfolio of the client. Correspondingly other tables are updated.

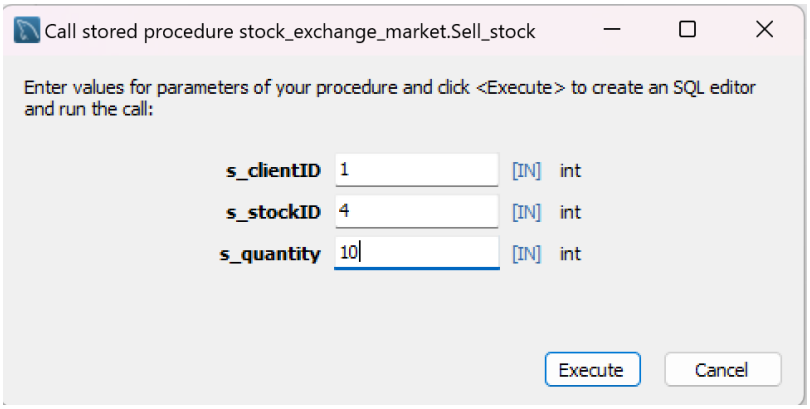
If the quantity of a particular stock sold is equal to the quantity available in client's portfolio, then the row is deleted.

Checking the client_stock_portfolio table;

```
1 •  SELECT * FROM stock_exchange_market.client_stock_portfolio;
```

| ID | quantity | stock_value | client_ID | stock_ID |
|---|---|---|---|---|
| 1 | 1 | 52 | 1 | 2 |
| 2 | 10 | 590 | 3 | 5 |
| 3 | 30 | 3060 | 5 | 3 |
| 4 | 20 | 1960 | 6 | 1 |
| 5 | 23 | 1219 | 2 | 9 |
| 6 | 10 | 390 | 1 | 4 |
| 7 | 10 | 530 | 4 | 9 |
| 8 | 20 | 1040 | 9 | 2 |
| 9 | 28 | 2744 | 2 | 1 |
| NULL | NULL | NULL | NULL | NULL |

Running sell_stock procedure function for client ID = 1, selling 10 stocks with ID = 4.




The portfolio table after selling the stock is,

| ID | quantity | stock_value | client_ID | stock_ID |
|---|---|---|---|---|
| 1 | 1 | 52 | 1 | 2 |
| 2 | 10 | 590 | 3 | 5 |
| 3 | 30 | 3060 | 5 | 3 |
| 4 | 20 | 1960 | 6 | 1 |
| 5 | 23 | 1219 | 2 | 9 |
| 7 | 10 | 530 | 4 | 9 |
| 8 | 20 | 1040 | 9 | 2 |
| 9 | 28 | 2744 | 2 | 1 |
| NULL | NULL | NULL | NULL | NULL |

The row containing all the quantities of the stock is deleted.

The sell row will be added in the stock_transaction table;

| ID | order_type | quantity | order_date | amount | stock_ID | client_ID |
|---|---|---|---|---|---|---|
| 1 | Buy | 3 | 2023-05-29 | 156 | 1 | 2 |
| 2 | Buy | 6 | 2023-05-29 | 312 | 1 | 2 |
| 3 | Sell | 5 | 2023-05-29 | 260 | 1 | 2 |
| 4 | Sell | 3 | 2023-05-29 | 156 | 1 | 2 |
| 5 | Buy | 3 | 2023-05-29 | 156 | 1 | 2 |
| 6 | Sell | 3 | 2023-05-29 | 156 | 1 | 2 |
| 7 | Buy | 10 | 2023-05-29 | 590 | 3 | 5 |
| 8 | Buy | 30 | 2023-05-29 | 3060 | 5 | 3 |
| 9 | Buy | 20 | 2023-05-29 | 1960 | 6 | 1 |
| 10 | Buy | 23 | 2023-05-29 | 1219 | 2 | 9 |
| 11 | Buy | 10 | 2023-05-29 | 390 | 1 | 4 |
| 12 | Buy | 10 | 2023-05-29 | 530 | 4 | 9 |
| 13 | Buy | 20 | 2023-05-29 | 1040 | 9 | 2 |
| 14 | Buy | 28 | 2023-05-29 | 2744 | 2 | 1 |
| 15 | Sell | 10 | 2023-05-29 | 390 | 1 | 4 |

**Trigger function in MySQL**:

In MySQL, a trigger is a database object that is associated with a specific table and automatically executed in response to certain events, such as insertions, updates, or deletions of data in the table.

To store the price history for each stock, create a trigger function BEFORE UPDATE in 'stocks_list' table, from where the current_price of stock is moved to the 'stock_price_history' table whenever the current_price is updated.


The IF statement checks if the current price (new.current_price) is different from the previous price (old.current_price) in the updated row.

If the prices are different, an INSERT statement is executed to insert a new record into the price_history table.

# CHAPTER THREE: GETTING VALUABLE INSIGHTS FROM THE DATA

SQL (structured query language) provides a powerful and flexible way to retrieve valuable insights from data. By writing queries, we can analyze, aggregate, and filter data to gain meaningful information and make informed decisions based on the results obtained.

1. Getting the top 5 total portfolio value and the client having it;

```
select clients.Name, sum(stock_value) as portfolio_value from client_stock_portfolio c
join clients on clients.ID = c.client_ID
group by client_ID order by portfolio_value desc limit 5;
```

The stock_value for each client_ID in client_stock_portfolio gives the corresponding stock's value. Join is used to connect the clients table, to get the name of client. To find the total portfolio value, we need to group the same clients by client_ID and to get the top 5 clients, order the values in descending order limiting 5 values.

| Name | portfolio_value |
|---|---|
| James | 3963 |
| Farzeen | 3450 |
| Joe | 2898 |
| Liya | 1960 |
| John | 1724 |

2. Getting the stock details that's been monitored or watch listed most by the clients;

```
select stock_ID, s.symbol, s.company, count(*)  from stock_watchlist w
join stocks_list s on s.ID = w.stock_ID
group by stock_ID
order by count(*) desc limit 1;
```

The count of the stock that's been in the watchlist of the greatest number of clients is obtained from stock_watchlist table and is joined to stocks_list table with the key stock_ID to get the stock details.

| stock_ID | symbol | company | count(*) |
|---|---|---|---|
| 1 | AAPL | Apple Inc. | 4 |

This query can be used to send notification for the customers when this stock's price changes.

3. Getting the least bought stock by the clients from the market;

```
select s.symbol, s.company, sum(t.quantity) as 'total quantity bought' from stock_transaction t
join stocks_list s on s.ID = t.stock_ID
where t.order_type = 'Buy'
group by stock_ID
order by sum(quantity) limit 1;
```

The quantity column in stock_transaction table gives the quanity of each stock bought or sold. Joining with stocks_list table gives the corresponding stock with minimum total quantity.

| | symbol | company | total quantity bought |
|---|---|---|---|
| ▶ | ORCL | Oracle | 10 |

Only 10 stocks of Oracle are been bought from the market, this can be reported to the company about the performance of stock.

4. Getting the highly priced stock and clients who bought it and how much;

```
select c.Name, t.quantity, s.symbol, s.company, s.current_price as 'price of stock' from stock_transaction t
join clients c on c.ID = t.client_ID
join stocks_list s on s.ID = t.stock_ID
where t.order_type = 'Buy' and s.current_price = (select max(current_price) from stocks_list);
```

The result is;

| | Name | quantity | symbol | company | price of stock |
|---|---|---|---|---|---|
| ▶ | Farzeen | 10 | GOOG | Alphabet | 102 |
| | Muhiz | 18 | GOOG | Alphabet | 102 |

The high-priced stock is ‘GOOG’ and been bought by 2 clients. This report can be given to the company about their stock performing in market.

5. Finding the client details who is actively trading in stock market;

```
select c.Name, c.Email, c.Account_balance, count(*) as 'no. of trading'
from stock_transaction t
join clients c on c.ID = t.client_ID
group by client_ID order by count(*) desc limit 3;
```

Which gives,

| | Name | Email | Account_balance | no. of trading |
|---|---|---|---|---|
| ▶ | James | jamesbg@gmail.com | 18037 | 7 |
| | Muhiz | muhiz888@gmail.com | 21797 | 4 |
| | Sanjana | sanj435@gmail.com | 34470 | 4 |

This gives the top 3 customers who traded most in the market. They are valuable customers of the stock market.

6. Get the client details having high account balance and the stocks in their portfolio;

```
select p.quantity, s.symbol, c.Name as 'client_name' from clients c
join client_stock_portfolio p on p.client_ID = c.ID
join stocks_list s on s.ID = p.stock_ID
where Account_balance = (select max(Account_balance) from clients)
```

This gives the client_name with high account balance and the stocks, quantity held by the client.

| | quantity | symbol | client_name |
|---|---|---|---|
| ▶ | 30 | GOOG | Farzeen |
| | 10 | ORCL | Farzeen |

7. Getting the top 3 stock in the market with high transaction volume;

```
select sum(amount) as 'transaction volume',s.symbol as 'stock_symbol', s.company from stock_transaction t
join stocks_list s on s.ID = t.stock_ID
group by stock_ID order by sum(amount) desc limit 3
```

This gives the result;

| | transaction volume | stock_symbol | company |
|---|---|---|---|
| ▶ | 5900 | MA | Mastercard |
| | 3963 | MSFT | Microsoft |
| | 3648 | AAPL | Apple Inc. |

8. Getting the details of a stock that hasn't been traded in a day of market;

```
select s.ID, s.symbol, s.company from stocks_list s
left join stock_transaction t on s.ID = t.stock_ID
where t.order_date is Null
```

The left join, joins the stocks list table to transaction table, and for the stock that haven't traded the right columns in transaction will be null.

The stocks are;

| | ID | symbol | company |
|---|---|---|---|
| ▶ | 10 | PEP | Pepsico |
| | 11 | KO | Cocacola |
| | 12 | CAT | Caterpillar |

This data can be given to the companies about their stocks.

# CHAPTER FOUR: TRANSACTION ANALYSIS AND CONSISTENCY OF AN EFFICIENT DATABASE SYSTEM

The complex world of stock exchange market database systems requires a delicate balance between the need for efficient transaction processing, strong consistency guarantees, and horizontal scalability.

**CAP Theorem:**

This serves as a pivotal concept that highlights the trade-offs between consistency, availability, and partition tolerance. This provides a framework for analyzing this balance by forcing an understanding of what must be sacrificed when prioritizing one aspect over another. By considering these trade-offs carefully during system design and maintenance efforts in trading environments that demand high reliability with minimal downtime or data loss risks from network partitions or failures. Financial professionals can ensure their databases remain protected while delivering tangible business benefits to clients who rely on consistent real-time information about stock prices at all times regardless of disruptions caused by either technical issues or cyber-attacks aimed at manipulation through false orders placed by rogue traders trying to dupe unsuspecting investors into buying low-quality assets at inflated prices.

**Role played by MySQL:**

Managing a stock exchange market database system can be an intricate task due to the vast number of transactions that occur daily. The need for consistency in these transactions is paramount, as any discrepancies can cause significant financial losses. Moreover, scalability becomes a concern as the volume and complexity of data increases over time. However, with MySQL's robust features and capabilities, these challenges can be addressed effectively. MySQL's ACID compliance ensures transactional consistency by guaranteeing atomicity, consistency, isolation, and durability in every transaction performed on the database. Additionally, MySQL provides built-in support for horizontal scaling through sharding techniques such as partitioning or replication. Therefore, ensuring that it meets industry standards for scalability while maintaining reliability in handling large volumes of transactions all at once. Using MySQL in managing stock exchange market

databases would not only increase efficiency but also boost productivity within this fast-paced environment where time is money.

**Why can't we use NoSQL:**

NoSQL databases are able to scale parallel and lodge flexible data. The reason that it cannot be used here is that NoSQL databases are based on availability and partition tolerance, which means they offer weaker consistency guarantees and limited transactional capabilities when compared to relational databases like MySQL. While NoSQL offers easy and automatic scaling, better performance, and high availability, these benefits can come at a cost when dealing with important financial information where accuracy is supreme.

Moreover, MySQL remains the preferred choice for critical applications because of its strong consistency guarantees along with robust transactional capabilities which ensures that all changes made to the database are logged consistently without any errors or discrepancies. Therefore, it is important that companies consider both the advantages and limitations before deciding which type of database management system they need for their business operations.

# CONCLUDING REMARKS

A stock exchange market database system requires high levels of consistency and scalability to handle large transactional volumes efficiently, this can come at the cost of reduced availability in some cases. As we have seen, MySQL solves these challenges by offering reliable ACID-compliant transactions and scalable architecture with clustering support. Additionally, its intuitive query language makes it easy for developers to work with complex data models.

In this assignment a foundation was created for the database design of a stock exchange market. The data was entered and manipulated using SQL to get the information from the data which can be collectively used for analytics purposes. The consistency and integrity of the database is evaluated using CAP and ACID theorem.

# BIBLIOGRAPHY


- Nazrul, S.S. (2018) *Cap theorem and Distributed Database Management Systems*, *Medium*. Available at: https://towardsdatascience.com/cap-theorem-and-distributed-database-management-systems-5c2be977950e (Accessed: 25 May 2023).
- Shivam, S. (2021) *Significance of acid vs base vs Cap philosophy in data science*, *Medium*. Available at: https://medium.com/analytics-vidhya/significance-of-acid-vs-base-vs-cap-philosophy-in-data-science-2cd1f78200ce (Accessed: 28 May 2023).
- randomwalkerrandomwalker 11911 silver badge33 bronze badges *et al.* (1967) *How to structure A stock market data database, Quantitative Finance Stack Exchange*. Available at: https://quant.stackexchange.com/questions/61699/how-to-structure-a-stock-market-data-database (Accessed: 27 May 2023).