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: Abijith Mullancherry Asokan

Year: 2023

CONTENTS

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): Abijith Mullancherry Asokan

Date: 30/05/2023

INTRODUCTION

Stocks are traded between a variety of parties, including investors, traders, and institutional customers, on the stock exchange market, which is a dynamic and fast-paced environment. A reliable and scalable database system must be in place if a stock exchange market is to be managed effectively. This database system should be able to manage a sizable number of stocks, track their historical prices, and make it easy for clients to purchase and sell securities while maintaining the accuracy of their account balances.

We need to create a thorough database system for a stock exchange market as part of this assignment. The system will be made up of a number of related tables that represent various entities and their connections. Stocks, clients, transactions, and price history are some of the key components. The objective is to develop a trustworthy and effective database schema that enables the archiving and retrieval of stock-related data, keeps track of stock prices over time, and allows users to buy and sell stocks while keeping an eye on their account balances.

We want to lay a strong basis for managing the stock exchange market's operations well by creating this database system. This technology will make it possible to track stock prices in real-time, keep an exhaustive record of price changes, and let users purchase and sell securities. It will also make sure that account balances are appropriately maintained for clients, stopping transactions from happening if clients do not have enough money.

CHAPTER ONE: Creation of a Database Management System

Schema in DBMS:

In our database, we will first create a schema, which is the conceptual layout of the database, named Stock_Exchange in MySQL Workbench. The following code is used to create the same.

```
1      CREATE SCHEMA `Stock_Exchange` ;
2
```

Entity Relation Diagram:

Entities in a database relate to the tables. Relationships between entities are shown visually in an entity-relationship diagram (ERD). It is a modeling method used to create and convey a database system's structure. In an ERD, entities are shown as rectangles, relationships between them as lines The elements of an ERD other than entities are as follows:

Relationships: Relationships describe connections between different kinds of things. They illustrate the connections between different things

Attributes: The qualities or traits of an entity are its attributes. They provide us more details about an object

Primary Key: A primary key is an identifier that is specific to each instance of an object. It is used to identify records in a table in an exclusive manner.

Foreign Key: A foreign key is a reference to another entity's primary key. It creates a connection between two things.

Defining the entities required for the database system:

Entities are items or things that are represented and maintained in a database in a database management system (DBMS). In a relational database, entities are often represented as tables, where each row in the table represents an instance of the entity and each column represents one of its attributes or properties. Relationships between entities are documented in the database using foreign keys and join operations.

The entities designed for the DBMS system for Stock Portfolio are as follows:

- Stock:
    - ID: Primary Key
    - Symbol: Distinct arrangement of character set specific to different stock

- Company: Organization name of the stock
- Current Price: Price of the stock for buying
- Quantity Available: Amount of quantity available to buy
- High_Price: High price in a month
- Low_Price: Low price in a month
- Open_Price: Price at which the stock was available at the start of the day.

- Price History:
  - ID: Primary Key
  - Date: Date of the price
  - Price: Price of the stock at the corresponding time
  - Stock_FK (Foreign Key): Stock for which the price history is related to
- Clients:
  - ID: Primary Key
  - Name: Name of the Client
  - Email
  - Account Balance: The balance available in the clients account.
- Balance History:
  - ID: Primary Key
  - Change Date: Date of change in balance
  - Balance Amount: The balance after the transaction
  - Client ID: Client whose balance has been affected
- Transactions:
  - ID: Primary Key
  - Order Type: Whether the order is a buy or sell.
  - Quantity: Number of stocks involved in the transaction.

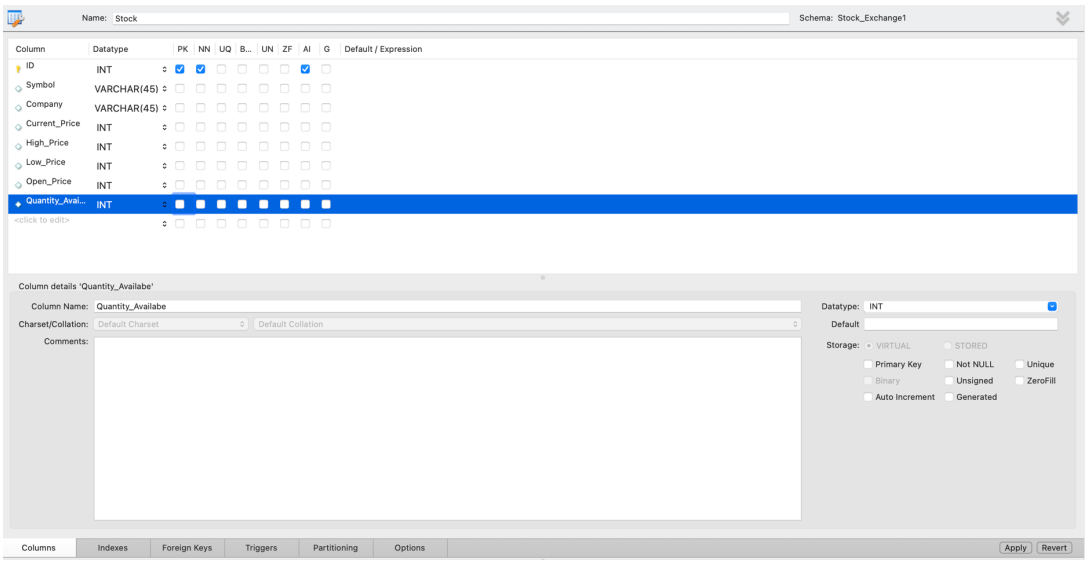
- Order date: Date of the order
- Price: Price of the whole transaction
- Client_FK (Foreign Key): Client who performed the transaction
- Stock_FK (Foreign Key): Stock involved in the transaction

- Client Stock Portfolio:
  - ID: Primary Key
  - Quantity: Amount of stock with the client
  - Stock Value: Total value of the stock with the client.
  - Client_FK(Foreign Key): Client who has the particular stock
  - Stock_FK(Foreign Key): Stock data in the hand of the client
- Watchlist:
  - Client_FK (Foreign Key): Client who has added the stock to watchlist
  - Stock_FK (Foreign Key): Stock data that has been added to the watchlist

With the knowledge of the entities in our hand, we can move forward to the creation of these entities inside our database schema in MySQL as different tables

Creation of tables in MySQL Workbench:

In MySQL Workbench we can create tables easily and assign the properties to each column using simple checkboxes as this is a GUI application. While creating the table, we should take care in specifying whether the columns are primary key, foreign key and also about the character type in the column field to be entered.



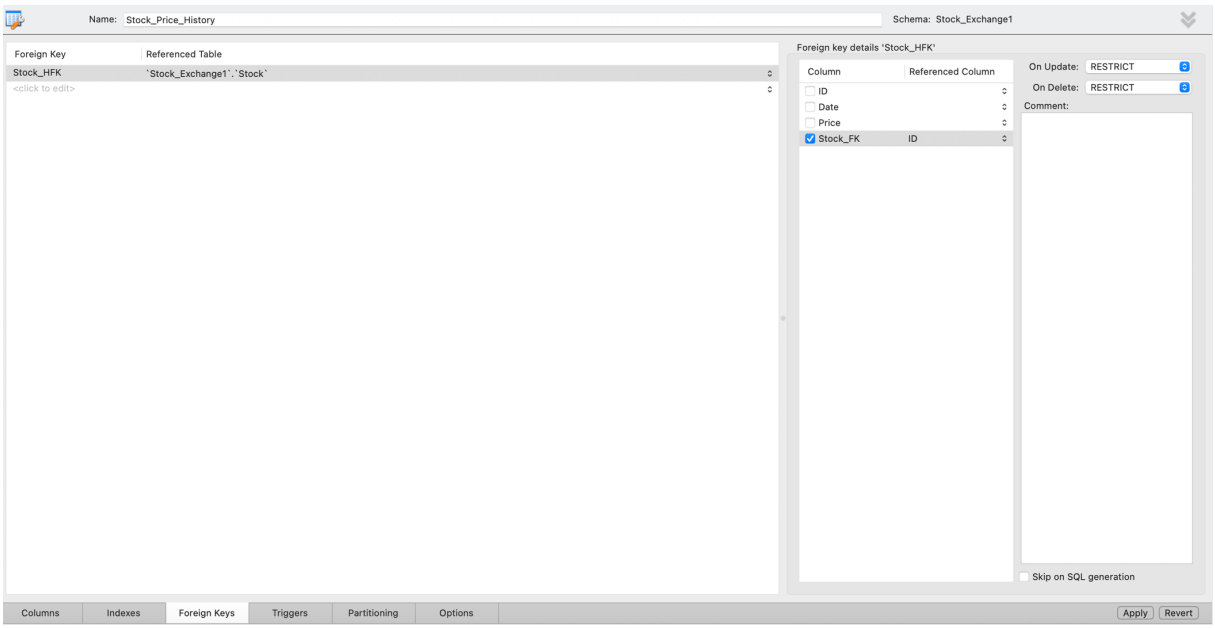
When we click on apply the respective code will be shown before creating the table.

```
CREATE TABLE `Stock_Exchange1`.`Stock` (
  `ID` INT NOT NULL AUTO_INCREMENT,
  `Symbol` VARCHAR(45) NULL,
  `Company` VARCHAR(45) NULL,
  `Current_Price` INT NULL,
  `High_Price` INT NULL,
  `Low_Price` INT NULL,
  `Open_Price` INT NULL,
  `Quantity_Availabe` INT NULL,
  PRIMARY KEY (`ID`));

```

Similarly, we need to create all the tables and assign the foreign keys for all the tables respectively as mentioned above so that our database system will have all the required relations.

The referenced table should be created before declaring foreign key. The declaration of foreign keys plays a crucial role in the proper functioning of the database system.




```sql
CREATE TABLE `Stock_Exchange1`.`Stock_Price_History` (
  `ID` INT NOT NULL AUTO_INCREMENT,
  `Date` DATE NULL,
  `Price` INT NULL,
  `Stock_FK` INT NOT NULL,
  PRIMARY KEY (`ID`),
  INDEX `Stock_HFK_idx` (`Stock_FK` ASC) VISIBLE,
  CONSTRAINT `Stock_HFK`
    FOREIGN KEY (`Stock_FK`)
    REFERENCES `Stock_Exchange1`.`Stock` (`ID`)
    ON DELETE NO ACTION
    ON UPDATE NO ACTION);

```

There are mainly four different types of relations in a DBMS.

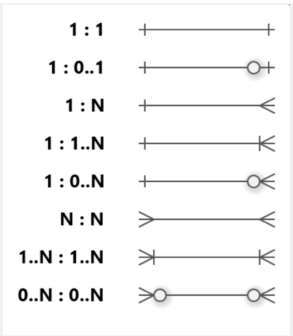
- One instance of one entity is linked to just one instance of another entity, and vice versa, in a one-to-one connection (1:1).
- A one-to-many connection (1:N) occurs when one instance of one entity is linked to numerous instances of another entity, but each instance of the second object is only linked to one instance of the first entity.
- A many-to-one (N:1) link connects several instances of one entity to a single instance of another entity.

- Numerous instances of one entity may be connected to numerous instances of another entity in a many-to-many (N:N) connection. In case of a many to many relation, we need to create an extra table with both foreign keys.

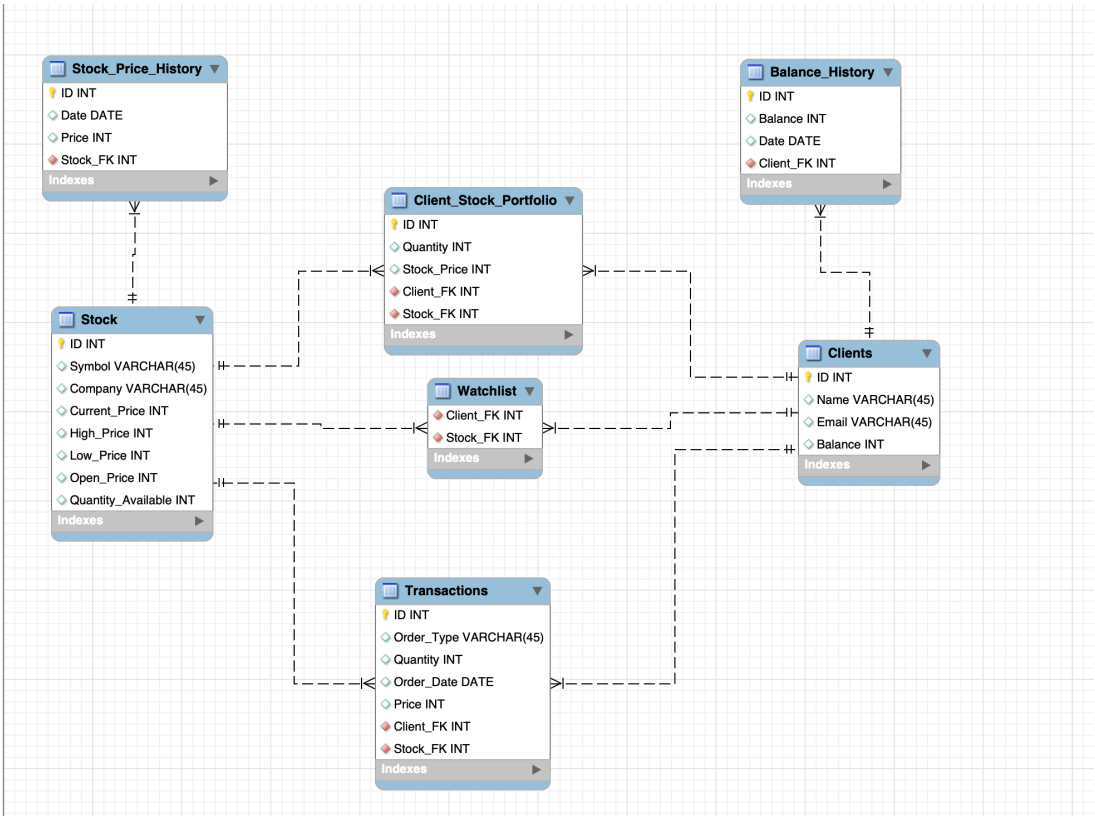
Relations in our Database Architecture:

There are six relations in our database system.

- One to many relation between Stock and Stock_Price_History since one stock can have multiple entries in the history column.
- One to many relation between Stock and Transactions as one stock can be found under different transactions.
- Similarly, in Transactions there is another many to one relation to Clients as one client can make more than one transaction.
- There is a one-to-many relation between Clients and Balance history which tracks the change in balance of the client.
- There are two many to relation from Client_Stock_Portfolio, one to Clients table and Stock table as one client can have multiple stocks and Same stock can be with Multiple Clients.
- There is a many to many relation between Clients and Stocks to track the watchlist of each client. A new table named Watchlist has been created to execute the same.



Once all the tables and foreign keys have been declared, we can then generate the ER Diagram for our system using reverse engineer method in MySQL Workbench.



Our database has been created.

CHAPTER TWO: Data Entry in DBMS

In MySQL Workbench, we can enter values into the table as columns in an Excel and applying them which will execute the same as writing queries to implement the same. Below are the snippets generated while implementing the same.

ID	Name	Email	Balance
1	Abijith	abijit…	50000
2	Hashir	hash…	65000
3	Arav…	aravi…	45000
4	Sarath	sarat…	48000
5	Aksh…	aksh…	40000
6	Haafiz	haafi…	55000
7	Danish	dani…	47888
8	Devan	deva…	38676
9	Dilka…	Dilut…	54786
10	Fenil	mon…	46778
NULL	NULL	NULL	NULL

```
INSERT INTO `Stock_Exchange1`.`Clients` (`ID`, `Name`, `Email`, `Balance`) VALUES ('1', 'Abijith', 'abijithasokan@gmail.com', '50000');
INSERT INTO `Stock_Exchange1`.`Clients` (`ID`, `Name`, `Email`, `Balance`) VALUES ('2', 'Hashir', 'hashir.on.run@gmail.com', '65000');
INSERT INTO `Stock_Exchange1`.`Clients` (`ID`, `Name`, `Email`, `Balance`) VALUES ('3', 'Aravind', 'aravind_rk@gmail.com', '45000');
INSERT INTO `Stock_Exchange1`.`Clients` (`ID`, `Name`, `Email`, `Balance`) VALUES ('4', 'Sarath', 'sarathsgc@gmail.com', '48000');
INSERT INTO `Stock_Exchange1`.`Clients` (`ID`, `Name`, `Email`, `Balance`) VALUES ('5', 'Akshay', 'akshay.asuran@gmail.com', '40000');
INSERT INTO `Stock_Exchange1`.`Clients` (`ID`, `Name`, `Email`, `Balance`) VALUES ('6', 'Haafiz', 'haafiz.serikkal@gmail.com', '55000');
INSERT INTO `Stock_Exchange1`.`Clients` (`ID`, `Name`, `Email`, `Balance`) VALUES ('7', 'Danish', 'danish_chem_nit@gmail.com', '47888');
INSERT INTO `Stock_Exchange1`.`Clients` (`ID`, `Name`, `Email`, `Balance`) VALUES ('8', 'Devan', 'devan_chaathan@gmail.com', '38676');
INSERT INTO `Stock_Exchange1`.`Clients` (`ID`, `Name`, `Email`, `Balance`) VALUES ('9', 'Dilkash', 'Dilutan.Tirur@gmail.com', '54786');
INSERT INTO `Stock_Exchange1`.`Clients` (`ID`, `Name`, `Email`, `Balance`) VALUES ('10', 'Fenil', 'monu040@gmail.com', '46778');

```

The above code will input all the data entered inside the table name Clients. Similarly, we need to enter data into the table Stocks to start using our database for real time usage.

Once we have added data into both Stock and Client table, we need to create procedures for buying and selling stocks. We need to make sure that the procedure will update the Transactions table, Client_Portfolio table as well update the Balance table along with updating stock quantity, balance in Stock and Client table.

```
CREATE DEFINER=`root`@`localhost` PROCEDURE `BuyStock`(c_id int, s_id int, qnt int)
BEGIN
    declare t_bal int;
    declare s_price int;
    declare q_avail int;
    declare bal_aft int;
    declare stk int;
    select Balance into t_bal from Clients where Clients.ID=c_id;
    select Current_Price into s_price from Stock where Stock.ID=s_id;
    select Quantity_Available into q_avail from Stock where Stock.ID=s_id;
    select Quantity into stk from Client_Stock_Portfolio where Client_Stock_Portfolio.Client_Fk = c_id and Client_Stock_Portfolio.Stock_FK = s_id;


    if q_avail > qnt then
        if t_bal >= (s_price * qnt) then
            set bal_aft = t_bal - (s_price * qnt);
            update Clients set Balance= Balance - (s_price * qnt) where Clients.ID=c_id;
            update Stock set Quantity_Available = Quantity_Available - qnt where Stock.ID=s_id;
            insert into Transactions(Order_Type, Quantity, Order_Date, Price, Client_FK, Stock_FK) values('Buy', qnt, now(), s_price * qnt, c_id, s_id);
            insert into Balance_History(Balance, Date, Client_FK) values(bal_aft, now(), c_id);
            if stk IS NULL then
                insert into Client_Stock_Portfolio(Quantity, Stock_Price, Stock_Fk, Client_FK) values(qnt, s_price * qnt , s_id, c_id);
            else
                update Client_Stock_Portfolio set Quantity = Quantity + qnt
                    where Client_Stock_Portfolio.Client_FK = c_id  and Client_Stock_Portfolio.Stock_FK = s_id;
                update Client_Stock_Portfolio set Stock_Price = Stock_Price + (qnt * s_price)
                    where Client_Stock_Portfolio.Client_FK = c_id  and Client_Stock_Portfolio.Stock_FK = s_id;
            end if;
            select 'success' as result;
        else
            select 'Insufficient balance' as result;
        end if;
    else
        select 'Insufficient quantity available' as result;
    end if;
    END
```

- It retrieves the account balance of the client (c_id), the current price of the stock (s_id), the quantity the client is intending to buy(qnt).
- It checks if the available quantity of the stock is greater than the desired quantity (qnt).
- If the available quantity is sufficient, it proceeds to check if the client has enough balance to make the purchase.
- If the client has enough balance, it calculates the new balance after deducting the purchase amount and updates the account balance.
- It updates the available quantity of the stock by subtracting the purchased quantity.
- It inserts a new record into the Stock_Transaction table to track the purchase.
- It inserts a new record into the Balance_History table to track the change in account balance.
- If the stock is already present in the client's stock portfolio (stk is not NULL), it updates the quantity and stock value in the portfolio. Otherwise, it inserts a new record into the Client_Stock_Portfolio table.
- Finally, it returns a result indicating the success or failure of the purchase.

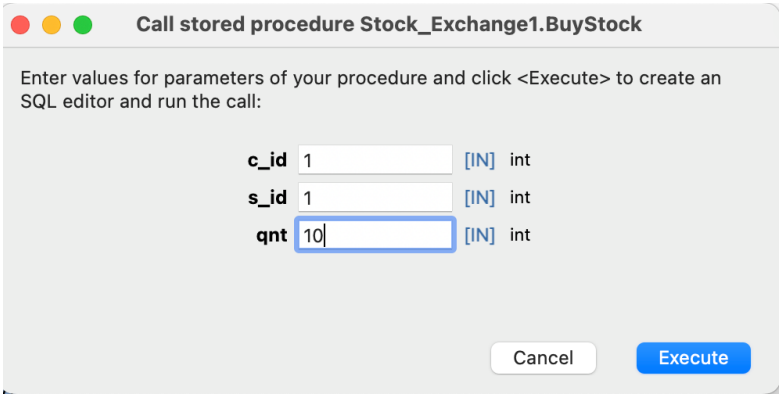
Similarly, a function has been created for selling the stock which does the opposite of the buy function by making the respective changes in other tables and the main tables also.

```
CREATE DEFINER=`root`@`localhost` PROCEDURE `SellStock`(c_id int, s_id int, qnt int)
BEGIN
    declare t_bal int;
    declare s_price int;
    declare q_avail int;
    declare bal_aft int;
    select Balance into t_bal from Clients where Clients.ID=c_id;
    select Current_Price into s_price from Stock where Stock.ID=s_id;
    select Quantity into q_avail from Client_Stock_Portfolio where Client_Stock_Portfolio.Stock_FK=s_id and Client_Stock_Portfolio.Client_FK=c_id;

    if q_avail >= qnt then
            set bal_aft = t_bal + (s_price * qnt);
            update Clients set Balance= Balance + (s_price * qnt) where Clients.ID=c_id;
            update Stock set Quantity_Available = Quantity_Available + qnt where Stock.ID=s_id;
            insert into Transactions(Order_Type, Quantity, Order_Date, Price, Client_FK, Stock_FK) values( 'Sell', qnt, now(), s_price * qnt, c_id, s_id);
            insert into Balance_History(Balance, Date, Client_FK) values(bal_aft, now(), c_id);
            if q_avail = qnt then
                delete from Client_Stock_Portfolio where Client_Stock_Portfolio.Stock_FK=s_id and Client_Stock_Portfolio.Client_FK=c_id;
            else
                update Client_Stock_Portfolio set Quantity= Quantity - qnt
                    where Client_Stock_Portfolio.Stock_FK=s_id and Client_Stock_Portfolio.Client_FK=c_id;
                update Client_Stock_Portfolio set Stock_Value = Stock_Value - (s_price * qnt)
                    where Client_Stock_Portfolio.Stock_FK=s_id and Client_Stock_Portfolio.Client_FK=c_id;
            end if;
                select 'success' as result;
    else
        select 'Insufficient quantity available to sell' as result;
    end if;
    END
```

Clients can buy and sell stock using these two functions. After repetitive running of the two functions. We will have the tables Client_Stock_Portfolio, Balance_History and Transactions with meaningful data inserted automatically. These tables are filled up with data as follows.

Meaningful values have been also added to tables watchlist and Stock_Price_History.



```
1 • SELECT * FROM Stock_Exchange1.Client_Stock_Portfolio;
```

100% 1:1

Result Grid Filter Rows: Search Edit: Expor

ID	Quantity	Stock_Price	Client_FK	Stock_FK
1	14	2304	1	1
2	24	4536	2	3
3	27	2619	3	5
4	15	3735	7	8
5	16	2304	5	1
6	22	4334	8	10
7	11	1463	10	4
8	17	2669	4	2
NULL	NULL	NULL	NULL	NULL

```
1 • SELECT * FROM Stock_Exchange1.Balance_History;
```

100% 1:1

Result Grid Filter Rows: Search Edit:

ID	Balance	Date	Client_FK
1	48560	2023-05-29	1
2	48560	2023-05-29	1
3	60464	2023-05-29	2
4	42381	2023-05-29	3
5	44153	2023-05-29	7
6	37696	2023-05-29	5
7	34342	2023-05-29	8
8	45315	2023-05-29	10
9	48848	2023-05-29	1
10	49136	2023-05-29	1
11	45331	2023-05-29	4
12	47984	2023-05-29	1
NULL	NULL	NULL	NULL

```
1 • SELECT * FROM Stock_Exchange1.Transactions;
```

100% 1:1

Result Grid Filter Rows: Search Edit:

ID	Order_Type	Quantity	Order_Date	Price	Client_FK	Stock_FK
1	Buy	10	2023-05-29	1440	1	1
2	Buy	10	2023-05-29	1440	1	1
3	Buy	24	2023-05-29	4536	2	3
4	Buy	27	2023-05-29	2619	3	5
5	Buy	15	2023-05-29	3735	7	8
6	Buy	16	2023-05-29	2304	5	1
7	Buy	22	2023-05-29	4334	8	10
8	Buy	11	2023-05-29	1463	10	4
9	Sell	2	2023-05-29	288	1	1
10	Sell	2	2023-05-29	288	1	1
11	Buy	17	2023-05-29	2669	4	2
12	Buy	8	2023-05-29	1152	1	1
NULL	NULL	NULL	NULL	NULL	NULL	NULL

For the Price_History table, we need to create a trigger which will store the previous value into the History table with the date value and Stock ID associated with it.

```
CREATE DEFINER = CURRENT_USER TRIGGER `Stock_Exchange1`.`Stock_BEFORE_UPDATE` BEFORE UPDATE ON `Stock` FOR EACH ROW
BEGIN
    INSERT INTO Stock_Price_History(Date, Price, Stock_FK) values(now(), OLD.Current_Price, OLD.ID);
END
```

The above code will insert the old price value into Stock_History_Table once the price is updated. This makes sure that we can keep track of the price over a longer period of time.

This implementation can help in visualizing the variation of price as a graph over large period of time.

```
SELECT * FROM Stock_Exchange1.Stock_Price_History;
```

100% 1:1

Result Grid Filter Rows: Search Edit: Expo

ID	Date	Price	Stock_FK
1	2023-04-27	147	1
2	2023-05-14	128	6
3	2023-05-15	178	10
4	2023-05-29	189	3
5	2023-05-30	144	1
6	2023-05-30	157	2
7	2023-05-30	188	3
NULL	NULL	NULL	NULL

When the current price has been updated in the Stock table, the trigger comes into role and the following data is automatically entered into the Stock_Price_History table as a result which implies the correct working of trigger.

CHAPTER Three: Data Manipulation in DBMS

Running Queries to visualize data upon the requested criteria:

The statements or commands that are used to communicate with the database to obtain, alter, or manipulate data are referred to in MySQL as queries. SQL is a widely used language for handling relational databases, and it is how MySQL queries are constructed.

We can perform different queries to manipulate the data as well as to analyze data once we have enough data to do so.

Query 1: To retrieve the name of Clients who has made transactions

```
Select Balance_History.Balance, Clients.Name from Balance_History
left join Clients on Balance_History.Client_Fk = Clients.ID;
```

We use the LEFT JOIN keyword to join the Balance_History table with the Clients table. The ON keyword specifies the join condition, which is the equality between the Client_Fk column in the Balance_History table and the ID column in the Clients table.

Result:

Balance	Name
48560	Abijith
48560	Abijith
48848	Abijith
49136	Abijith
47984	Abijith
60464	Hashir
42381	Aravind
45331	Sarath
37696	Akshay
NULL	Haafiz
44153	Danish
34342	Devan
NULL	Dilkash
45315	Fenil

Query 2: To retrieve the name of the company with the total number of stock quantity sold

```
1   select T1.quantity_bought, Stock.Company from
2       (select max(Quantity) as quantity_bought, Stock_FK from Client_Stock_Portfolio group by Stock_FK) as T1
3   join Stock on T1.Stock_FK=Stock.ID;
4
```

We first create a subquery by selecting the maximum quantity (MAX(Quantity)) for each Stock_FK from the Client_Stock_Portfolio table and grouping the results by Stock_FK. We give this subquery an alias T1.

Then, in the outer select statement, we join T1 with the Stock table using the common column Stock_FK and retrieve the quantity_bought from T1 and the corresponding Company from the Stock table.

Result:

	quantity_boug...	Company
	16	Exxon Mobile
	17	Dell Inc
	24	Sony Corporation
	11	Fedex Corporation
	27	Wipro Limited
	15	Coca-Cola Enterprises
	22	Microsoft Corporation

Query 3: To retrieve the name of the Client who has the stock with the highest variation

```
select T1.Company, T3.Name
    from (select ID, High_Price-Low_Price as Max_Variation,Company
            from Stock order by Max_Variation DESC Limit 1) as T1
join Client_Stock_Portfolio as T2 ON T1.ID=T2.Stock_FK
join Clients as T3 ON T2.Client_Fk=T3.ID;
```

We first create a subquery by selecting the ID, High_Price - Low_Price as Max_Variation, and Company from the Stock table. We then order the result set in descending order based on Max_Variation and limit the result to only one row using LIMIT 1. We give this subquery an alias T1.

Then, in the outer select statement, we join T1 with the Client_Stock_Portfolio table using the common column ID from T1 and Stock_FK from T2. We then join the result with the Clients table using the common column Client_Fk from T2 and ID from T3. Finally, we retrieve the Company from T1 and Name from T3.

	Company	Name
	Coca-Cola Enterprises	Danish

Query 4: To find the client who has made the most transactions

```
select T2.Name from
    (select Client_FK,count(Client_FK) as num_of_transactions from
            Transactions group by Client_FK order by num_of_transactions DESC limit 1) as T1
join Clients as T2 on T1.Client_FK=T2.ID;
```

We first create a subquery by selecting the Client_FK and counting the number of transactions (COUNT(Client_FK)) for each client from the Transactions table. We then group the results by Client_FK, order them in descending order based on num_of_transactions, and limit the result to only one row using LIMIT 1. We give this subquery an alias T1.

Then, in the outer select statement, we join T1 with the Clients table using the common column Client_FK from T1 and ID from T2. Finally, we retrieve the Name from T2.

Query 5: To know the name of Client and Company name associated with the data in Watchlist

```
select T2.Name, T3.Company from (SELECT * FROM Stock_Exchange1.Watchlist) as T1
join Clients as T2 ON T1.Client_FK = T2.ID
join Stock as T3 ON T1.Stock_FK = T3.ID;
```

We first create a subquery by selecting all columns from the Watchlist table. The subquery is given an alias T1.

Then, in the outer select statement, we join T1 with the Clients table using the common column Client_FK from T1 and ID from T2. We then join the result with the Stock table using the common column Stock_FK from T1 and ID from T3. Finally, we retrieve the Name from T2 and Company from T3.

Query 6: To find the total stock value sold by each stock on which transactions has been made.

```
select T2.Company, (T2.Current_Price*T1.qnt_total) as Total_value_sold from
    (SELECT Stock_FK, sum(quantity) as qnt_total FROM Stock_Exchange1.Transactions group by Stock_FK) as T1
join Stock as T2 on T1.Stock_Fk = T2.ID order by Total_value_sold DESC;
```

We first create a subquery by selecting the Stock_FK column and calculating the sum of the quantity column as qnt_total from the Transactions table. The subquery is given an alias T1.

Then, in the outer select statement, we join T1 with the Stock table using the common column Stock_FK from T1 and ID from T2. We also calculate the total value sold by multiplying the Current_Price from T2 with qnt_total from T1.

Result:

Company	Total_value_s...
Exxon Mobile	7536
Microsoft Corporation	4334
Sony Corporation	4224
Coca-Cola Enterprises	3735
Dell Inc	2805
Wipro Limited	2619
Fedex Corporation	1463

Query 7: To find the stock in hand of the person with lowest account balance

```
select T1.Name, T1.Balance, T3.Company, T2.Quantity from
    (select ID, Balance, Name from Clients order by Balance limit 1) as T1
join Client_Stock_Portfolio as T2 on T1.ID = T2.Client_FK
join Stock as T3 on T2.Stock_FK = T3.ID;
```

We first create a subquery by selecting the ID, Balance, and Name columns from the Clients table. We order the result by the Balance column in ascending order and limit the result to the first row using LIMIT 1. The subquery is given an alias T1.

Then, in the outer select statement, we join T1 with the Client_Stock_Portfolio table using the common column ID from T1 and Client_FK from T2. We also join T2 with the Stock table using the common column Stock_FK from T2 and ID from T3.

Result:

Name	Balance	Company	Quantity
Devan	34342	Microsoft Corporation	22

Query 8: To find the Stock in hand of the person watching the stock which has been sold most

```
select T1.count as Most_bought,T6.Company, T3.Name as Watching_Person, T5.Symbol as Stock_in_hand, T5.Company from
(select sum(Quantity) as count, Stock_FK from Client_Stock_Portfolio group by Stock_FK order by count DESC limit 1) as T1
join watchlist as T2 on T2.Stock_FK = T1.Stock_FK
join Clients as T3 on T3.ID = T2.Client_FK
join Client_Stock_Portfolio as T4 on T4.Client_Fk=T3.ID
join Stock as T5 on T4.Stock_FK = T5.ID
join Stock as T6 on T1.Stock_FK = T6.ID;
```

We first create a subquery by selecting the sum of Quantity as count and Stock_FK from the Client_Stock_Portfolio table. We group the results by Stock_FK, order them by the count in

descending order, and limit the result to the first row using LIMIT 1. The subquery is given an alias T1.

Then, in the outer select statement, we join T1 with the Watchlist table using the common column Stock_FK from both tables. We also join T2 with the Clients table using the common column Client_FK from T2 and ID from T3.

Next, we join T3 with T4 on ID from T3 and Client_FK from T4. We further join T4 with T5 on Stock_FK from T4 and ID from T5. Lastly, we join T1 with T6 on Stock_FK from T1 and ID from T6.

Result:

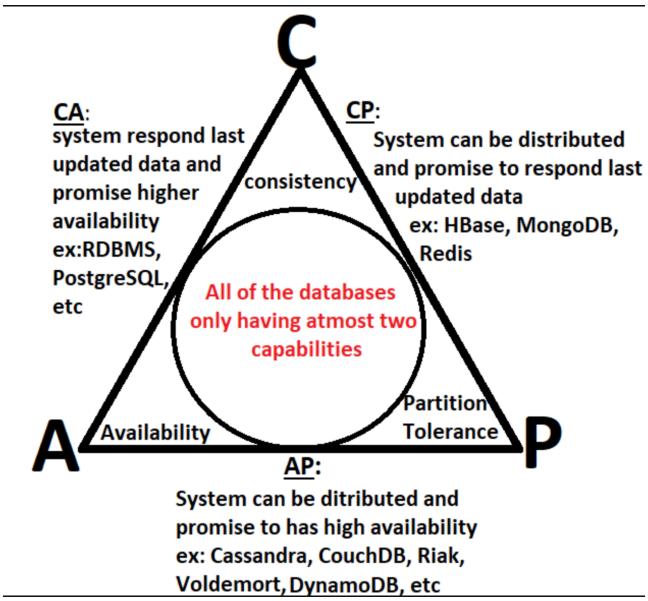
Most_bought	Company	Watching_Person	Stock_in_ha...	Company
30	Exxon Mobile	Sarath	DELL	Dell Inc

Chapter Four: CAP Theorem in the context of stock exchange market

CAP Theorem:

According to the CAP theorem, a distributed system cannot deliver or guarantee more than two of the following three attributes at once:

- Consistency: The same data is displayed simultaneously to all clients.
- Accessibility: Because the system constantly replies, all clients are able to read and publish data at any time.
- Partition tolerance: The system can keep working even if certain nodes stop working or are unable to interact with one another.



Why use Relational DBMS ?

We need a database which can relate different tables in our system to each other for efficient performance. A non-relational database is where there is no tabular rows or columns present. Also, relational database ensures that the data is always consistent and available at all times. According to CAP theorem, in case of a network failure, any one of the factors cannot be held up. For our database, we can tolerate partition tolerance which ensures that the other two factors are present at all times, which is essential for smooth working of the stock exchange market. Relational database ensures that we can achieve the following:

- Our data needs to be structured and organized which can be implemented easily through the use of a relational database system.

- Data integrity, which is an integral part of stock exchange market can be achieved through the use of key constraints such as foreign key, primary key which is part of relational database system.
- Relational database also compliances with ACID properties, which are as follows:
    - Atomicity, which ensures that the transactions are done as a single unit of work.
    - Consistency: It ensures that the data remains valid throughout transactions.
    - Isolation: Helps in performing different transactions at the same time without interference.
    - Durability: Which helps in making sure the data is stable and not lost in case of system failure.
- Relational database also provides us with the tools to query and join between tables which helps in getting meaningful insights of the data for future purposes.
- More than that, relational databases can support large amounts of data and high volume of transactions.
- It also provides built in security features which helps in data integrity and security.
- A solid foundation can be made for data analyzing using relational database system.
- The use of other tools provided by Relational DBMS such as triggers and stored procedures helps us in making the database much more consistent.

CONCLUSION

We have successfully designed and implemented a database in MySQL Workbench while taking several variables into consideration related to stock exchange market, including maintaining stocks, price history, client accounts, and transactions. Usage of MySQL has also helped us in establishing a relational database which maintains data integrity, query flexibility, transaction support, and data consistency.

Understanding the trade-offs involved in database design is largely dependent on the CAP theorem, which asserts that a distributed system cannot simultaneously ensure consistency, availability, and partition tolerance. While consistency and availability are the primary concerns of a relational database like MySQL, partition tolerance may be handled by technologies like replication and clustering.

In MySQL, triggers offer a mechanism to automate responses to certain events or data changes, and they may be used to uphold data integrity, enforce business rules, and promote consistency in the database.

Overall, MySQL provides a solid basis for creating a dependable and effective stock exchange database system with its ACID transactions, data constraints, referential integrity support, replication, and high availability options. To guarantee the best speed, scalability, and security, it is crucial to thoroughly evaluate the unique needs of the stock exchange market and make suitable design decisions. For the database system to remain reliable over time, regular monitoring, performance adjustment, and adherence to best practices are essential.

BIBLIOGRAPHY

- 9, D. and Johnson, J. (2020) *CAP theorem for databases: Consistency, Availability & Partition Tolerance*, *BMC Blogs*. Available at: https://www.bmc.com/blogs/cap-theorem/ (Accessed: 30 May 2023).
- Pattinson, T. (2022) *Relational vs non-relational databases*, *Relational vs Non-Relational Databases*. Available at: https://www.pluralsight.com/blog/software-development/relational-vs-non-relational-databases (Accessed: 30 May 2023).
- Katwal, B. (2020) *MongoDB vs Cassandra vs RDBMS, where do they stand in the cap theorem?*, *Medium*. Available at: https://bikas-katwal.medium.com/mongodb-vs-cassandra-vs-rdbms-where-do-they-stand-in-the-cap-theorem-1bae779a7a15#:~:text=In%20summary%2C%20a%20relational%20database,the%20leader%20or%20master%20node. (Accessed: 30 May 2023).
- *The Cap Theorem in DBMS* (2023) *GeeksforGeeks*. Available at: https://www.geeksforgeeks.org/the-cap-theorem-in-dbms/ (Accessed: 30 May 2023).
- Martinekuan (no date) *Non-relational data and NoSQL - Azure Architecture Center*, *Azure Architecture Center | Microsoft Learn*. Available at: https://learn.microsoft.com/en-us/azure/architecture/data-guide/big-data/non-relational-data (Accessed: 30 May 2023).