# USE CASE STUDY REPORT

**Group No.**: Group 13

**Student Names**: Yanming Liu & Mohit Chhajed

## Abstract

People tend to invest because of the low deposit rate in the COVID-19 period. It gave us the insights to create a bank-like investment tool to meet such customer demand. It features customer service, financial service, and supportive product supply. To achieve this, we design the conceptual model via the EER model and UML language. Then, we map the relational model in a normalized way. Next, databases like MySQL, MongoDB, Neo4j are generated with some fake data, and queries with questions like the information of customers, financial expectation of users and annual profits for companies. We also use Python to access the MySQL database and visualize the demo result using matplotlib package and tableau.

## I. Introduction

Coronavirus set off a profound financial slump of dubious span which hampered the US economy, which drove various organizations and families into a monetary fiasco. To help and consider the necessities of general society during these unprecedented times, FED gave an expansive exhibit of activity to restrict the financial harm from the pandemic by loaning trillions to support households, business and medical bills.

This is the place where Commercial banks assume a significant part in the financial system and the economy. As a critical part of the monetary framework, banks allocate funds from savers to borrowers in an efficient manner. They provide specialized financial services, which reduce the cost of obtaining information about both savings and borrowing opportunities. These financial services help to make the general economy more proficient.

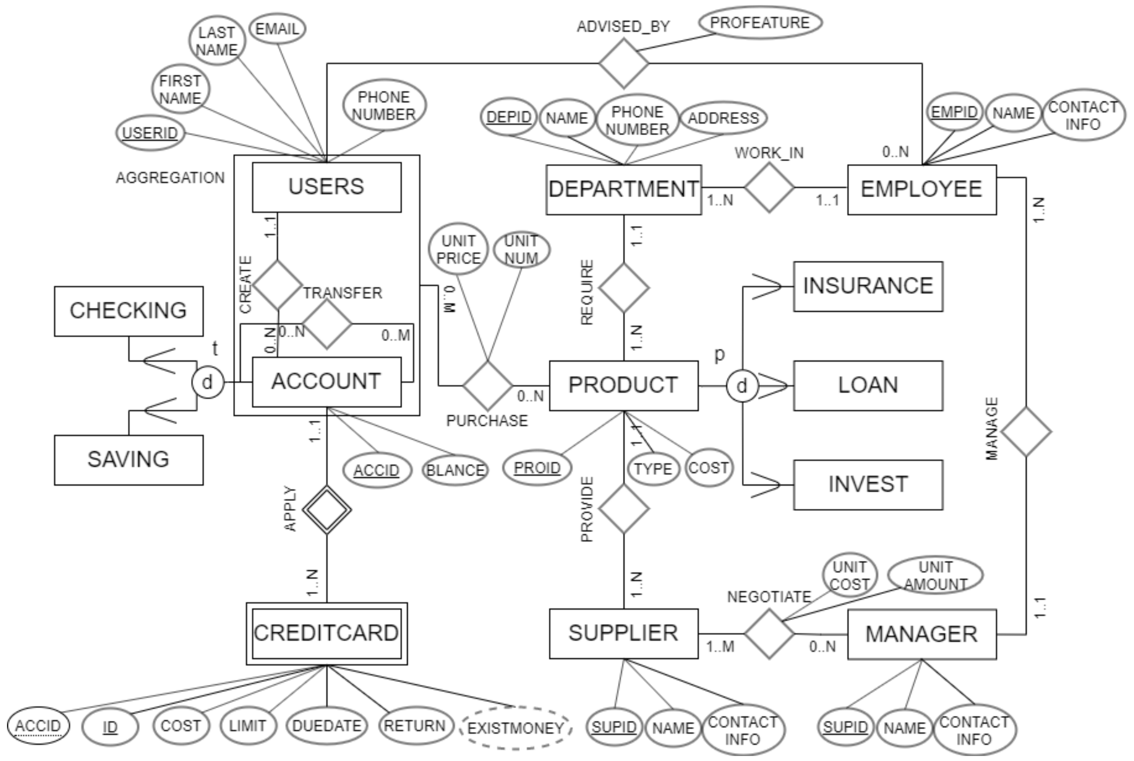
For this we have planned a conceptual model that incorporates many components for the bank customers like fund transfer, applying for loans, buying insurance, cash and checks deposit, cash withdrawal, investing their savings in bond, debenture, stock, mutual funds, and so forth. They can likewise utilize more services, like credit/debit cards and advisory services. Advisory service is a customer-centric channel to ensure customer satisfaction and delightful interaction to provide vital counsel for any above-mentioned service. There are explicit divisions to deal with each assistance and the manager is liable for negotiating the prices with some 3rd party entities giving more ideal arrangements to the clients.

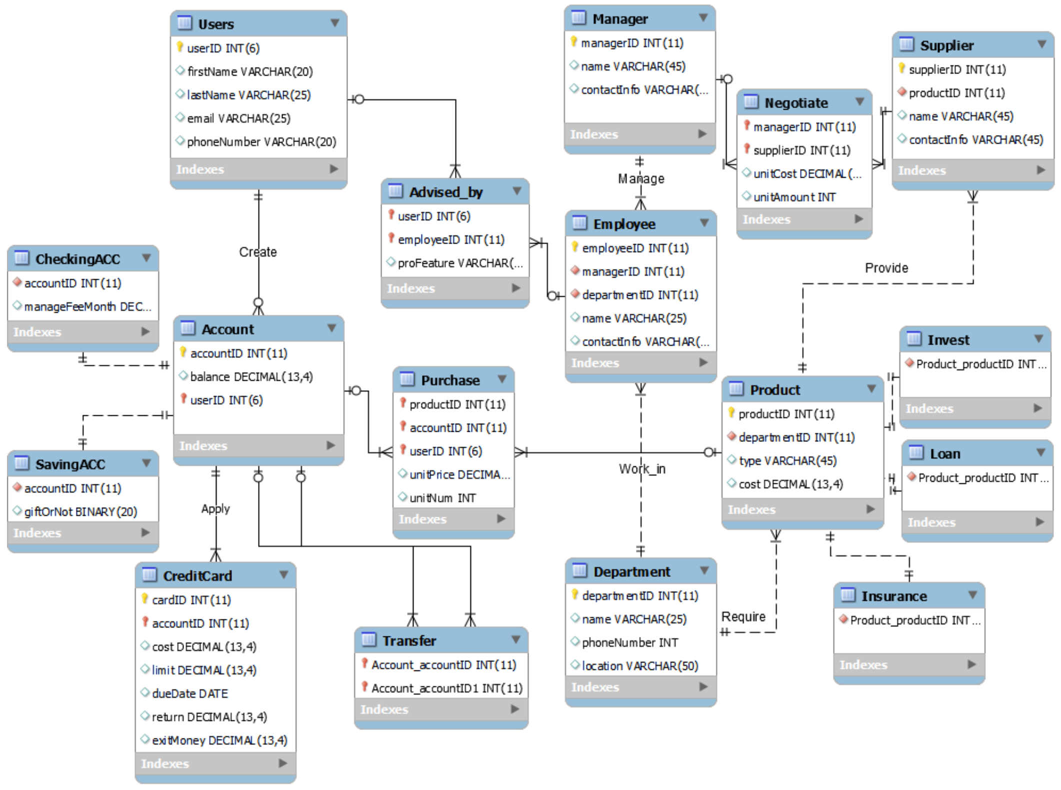
Other requirements…

1) A user can create one to many accounts.
2) An account can transfer money from one account to another.
3) A credit card is a derived entity from the aggregation of user and account. One account can have one to many credit cards. (it cannot have N cards but can have more than one )
4) Exist money is a derived attribute type that gives the current credit limit.
5) A user can purchase various products (loans, stocks, insurance).
6) A user can take advice from multiple employees. Each employee can belong to one or many departments and a manager can manage multiple employees.
7) A supplier can negotiate with zero or many managers to supply the product that the department requires.

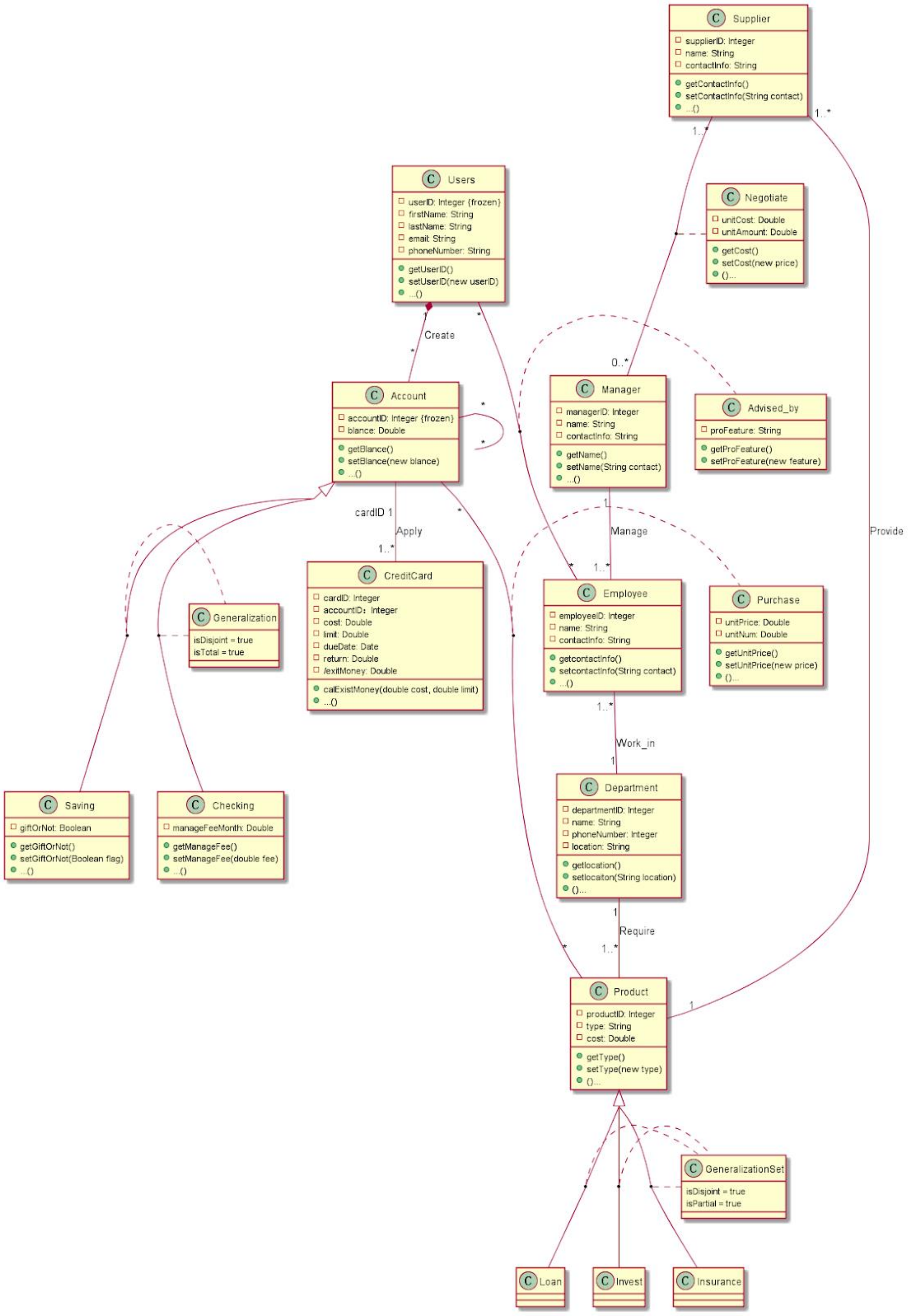
## II. Conceptual Data Modeling

1. EER Diagram | *support by Drawio & MySQL Workbench*





2. UML Diagram | *support by plantUML*



## III. Mapping Conceptual Model to Relational Model

**Primary Key- Underlined** **Foreign Key- *Italicized***

**Users(userID, firstName, lastName, email, phoneNumber)**
**Account(accountID, blance, *a_userID*)**
FOREIGN KEY a_userID refers to userID in Users; NOT NULL
**Aggregation(*userID, accountID* )**
FOREIGN KEY userID refers to userID in Users, accountID refers accountID in Account; NOT NULL
**Transfer(Account_accountID, Account_accountID1)**
FOREIGN KEY *Account_accountID, Account_accountID1* refers to accountID in Account; NOT NULL
**CreditCard(cardID, *c_accountID*, cost, limit, dueDate, return, existMoney)**
FOREIGN KEY *c_accountID* refers to accountID in Account; NOT NULL
**Supplier(supplierID, name, contactInfo, *s_productID*)**
FOREIGN KEY *s_productID* refers to productID in Product; NOT NULL
**Manager(manageID, name, contactInfo)**
**Product(productID, type, cost, *p_departmentID*)**
FOREIGN KEY *p_departmentID* refers to departmentID in Department; NOT NULL
**Employee(employeeID, *e_manageID*, *e_departmentID*, name, contactInfo)**
FOREIGN KEY *manageID* refers to manageID in Manager; NOT NULL
FOREIGN KEY *e_departmentID* refers to departmentID in Department; NOT NULL
**Department(departmentID, name, phoneNumber, location)**
**Advised_by(*ad_userID, ad_employeeID*, proFeature)**
FOREIGN KEY *ad_userID* refers to *userID* in Users; NOT NULL
FOREIGN KEY *ad_employeeID* refers to *employeeID* in Employee; NOT NULL
**Negotiate(*n_manageID, n_supplierID*, unitCost, unitAmount)**
FOREIGN KEY *n_manageID* refers to manageID in Manager; NOT NULL
FOREIGN KEY *n_supplierID* refers to supplierID in Supplier; NOT NULL
**Purchase(*p_accountID, p_productID, p_userID*, unitPrice, unitNum)**
FOREIGN KEY *p_accountID* refers to accountID in Account; NOT NULL
FOREIGN KEY *p_productID* refers to productID in Product; NOT NULL
FOREIGN KEY *p_userID* refers to *userID* in Users; NOT NULL
**Checking(*c_accountID*, manageFeeMonth)**
FOREIGN KEY *c_accountID* refers to accountID in Account; NOT NULL
**Savin(s_accountID, giftOrNot)**
FOREIGN KEY *s_accountID* refers to accountID in Account; NOT NULL
**Loan(*l_productID*)**
FOREIGN KEY *l_productID* refers to productID in Product; NOT NULL
**Invest(*inv_productID*)**
FOREIGN KEY *inv_productID* refers to productID in Product; NOT NULL
**Insurance(*ins_productID*)**
FOREIGN KEY *ins_productID* refers to productID in Product; NOT NULL

The relational model is not perfect, there are some possible losses of semantics when mapping the conceptual model to the one:

- It cannot enforce the minimum cardinality of 1 in the relationship below:
  - The account can have at least one credit card.
  - The manager can have at least one supplier.
  - Each product can be provided by at least one supplier.
  - Each department requires at least one product.
- It cannot enforce the disjoint and total constraints in the specialization of Account and Product.

## IV. Implementation of Relation Model via MySQL and NoSQL

### MySQL:

#### Setup

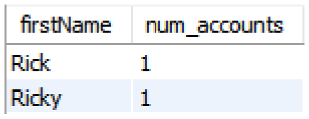
- Draw EER model in MySQL Workbench
- Use Forward Engineer in Database to generate Data Define Language (DDL) statement
- Prepare .csv file for tables under the constraints, like foreign key constraints
- Insert records using Data Import
- Figure out the valuable questions and write queries.

We aim to get the basic information from our customers, like who are they, who are consulting, debt situation, are they rich to get the bonus, and the profit expectation for our companies.

### Query

**# Find the specific number of accounts that users whose first name is like "Ric"**

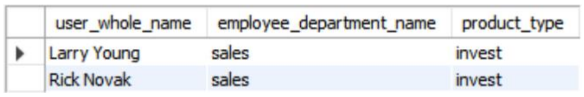
```
use autobankms;
select u.firstName, count(*) as
num_accounts
from users u, account a
where u.userID = a.accountID
and u.firstName like "%Ric%"
group by u.firstName;
```



| firstName | num_accounts |
|---|---|
| Rick | 1 |
| Ricky | 1 |

**# Get the users' full name whose consulting employee belongs to sales department that sold product type is inveset.**

```
select concat(u.firstName, ' ', u.lastName)
as user_whole_name, d.name as
employee_department_name,
p.type as product_type
from users u, advised_by ab, employee e,
department d, product p
where u.userID = ab.userID
and ab.employeeID = e.employeeID
and e.departmentID = d.departmentID
and d.departmentID = p.productID
and d.name = 'sales'
and p.type = "invest"
order by user_whole_name;
```

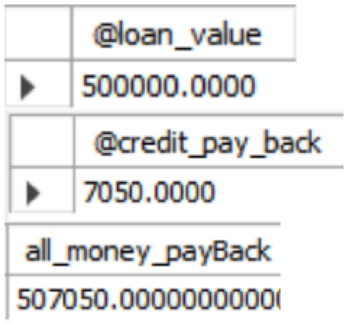


| user_whole_name | employee_department_name | product_type |
|---|---|---|
| Larry Young | sales | invest |
| Rick Novak | sales | invest |

**# Get the money amount should be be paid by users(loan + credit cards)**

```
set @loan_value =
(select (unitNum * unitPrice)
from purchase
where productID in
(select productID from product
where type = 'loan'));

set @credit_pay_back=
(select sum(cost - c.return) from creditcard
c);
select @loan_value;
select @credit_pay_back;
select @loan_value + @credit_pay_back as
all_money_payBack;
```



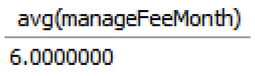
```
@loan_value
500000.0000

@credit_pay_back
7050.0000

all_money_payBack
507050.000000000(
```

**# Get the average manageFeeMonth for the checkingacc**

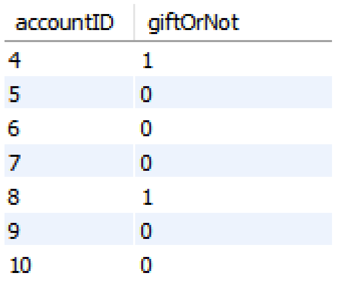
```
select avg(manageFeeMonth) from
checkingacc;
```



```
avg(manageFeeMonth)
6.0000000
```

**# Get all giftOrNot status of savingacc if the accountID do not belongs to checkingacc**

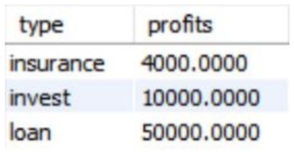
```
create view checkingIDA as (select
accountID, balance from account where
accountID not in (select accountID from
checkingacc));
create view checkingIDV as (select
accountID, balance, if(balance>50000,
"YES", "NO") as giftOrNot from
checkingIDA);
select * from checkingIDV;
insert into savingacc(accountID, giftOrNot)
select accountID, giftOrNot from
checkingIDV;
select * from savingacc;
```



```
accountID   giftOrNot
4           1
5           0
6           0
7           0
8           1
9           0
10          0
```

**# Get the profits that the bank earns for all products.**

```
select pr.type, ((pu.unitPrice - n.unitCost) *
pu.unitNum) as profits
from purchase pu, negotiate n, product pr,
supplier s
where pu.productID = pr.productID
and pr.productID = s.productID
and s.supplierID = n.supplierID
group by pu.productID
```



```
type        profits
insurance   4000.0000
invest      10000.0000
loan        50000.0000
```

**# Get the manager's name and its level.**

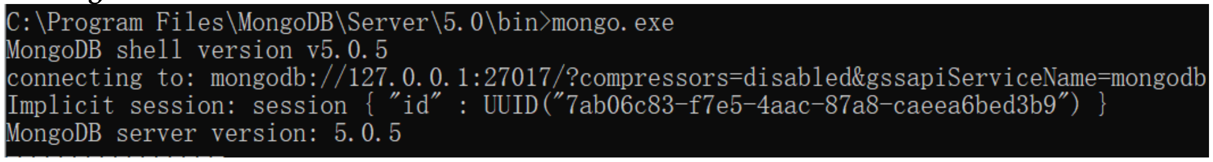
```
with subManager(manageID, name, level) as
(select managerID, name, 1
from manager where reporto = null)
union all
(select m.managerID, m.name, s.level+1
from subManager s, manager m where
s.managerID = m.reporto);

select * from subManger order by level;
```

## MongDB:
## Setup

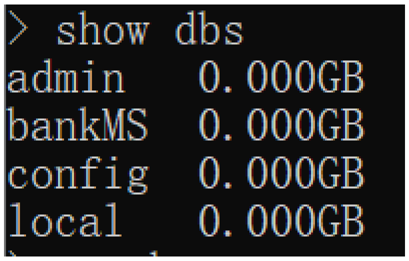
- Prepare .csv file, like creditcard.csv
- Add data in MongoDB client
- Run MongoDB server in terminal
- Open another terminal and write a query

**# Run MongDB shell**



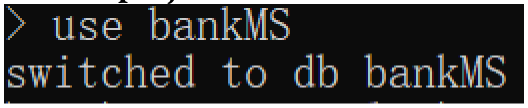
```
C:\Program Files\MongoDB\Server\5.0\bin>mongo.exe
MongoDB shell version v5.0.5
connecting to: mongodb://127.0.0.1:27017/?compressors=disabled&gssapiServiceName=mongodb
Implicit session: session { "id" : UUID("7ab06c83-f7e5-4aac-87a8-caeea6bed3b9") }
MongoDB server version: 5.0.5
```

**# Show all the database**



```
> show dbs
admin   0.000GB
bankMS  0.000GB
config  0.000GB
local   0.000GB
```

**# Select the project database**

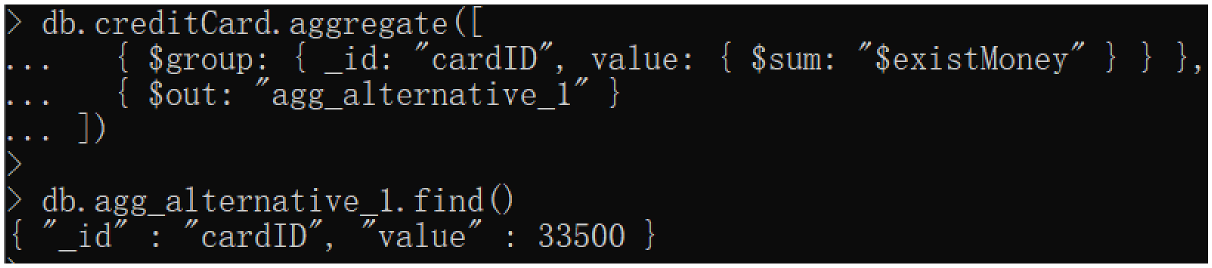


```
> use bankMS
switched to db bankMS
```

Here, we just calculate the sum exist money in creditCard as a demo.

## Query

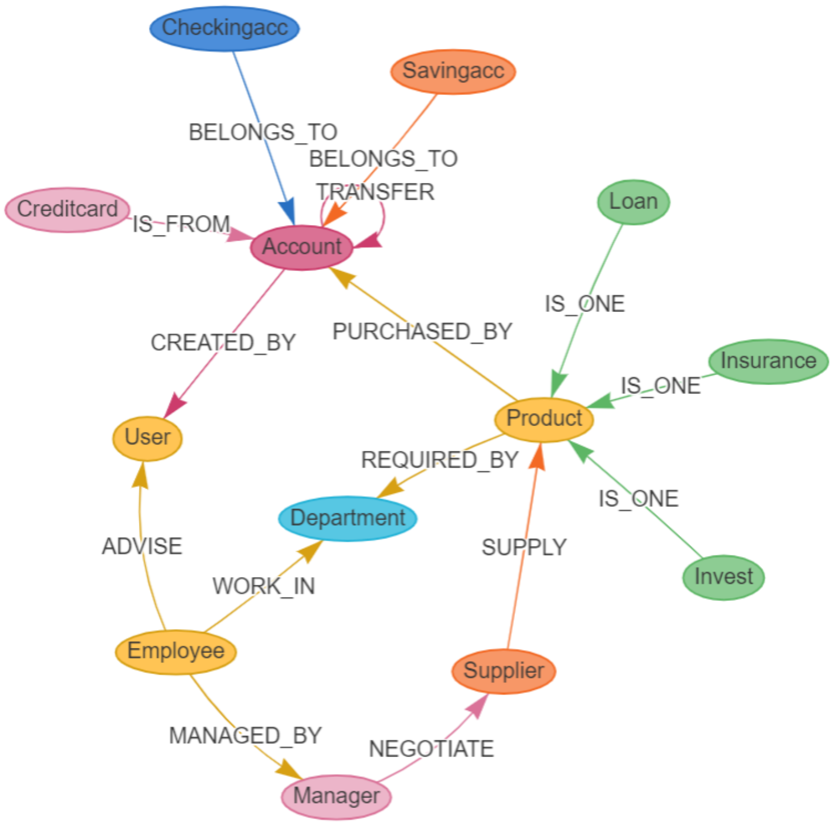
**# Get the sum value of existMoney**



```
> db.creditCard.aggregate([
...     { $group: { _id: "cardID", value: { $sum: "$existMoney" } } },
...     { $out: "agg_alternative_1" }
... ])
>
> db.agg_alternative_1.find()
{ "_id" : "cardID", "value" : 33500 }
```

**NoSQL - NEO4J:**
**Setup:**

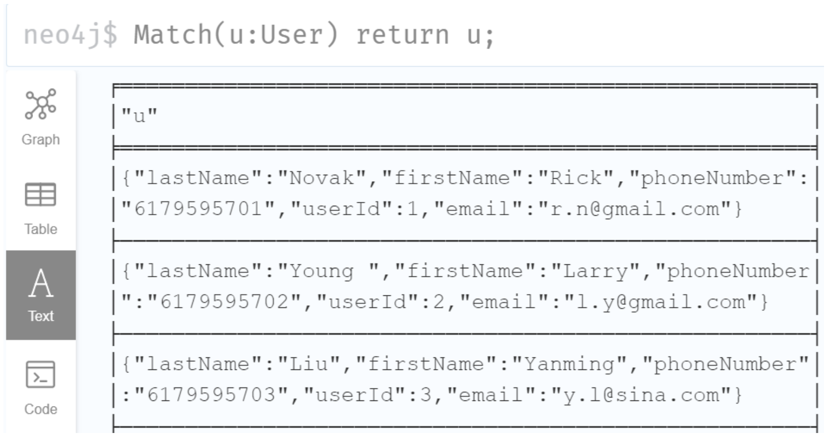
- Use ETL tool to transform MySQL database to Neo4j graph database



- Go to neo4j Browser and run the queries

**Query**

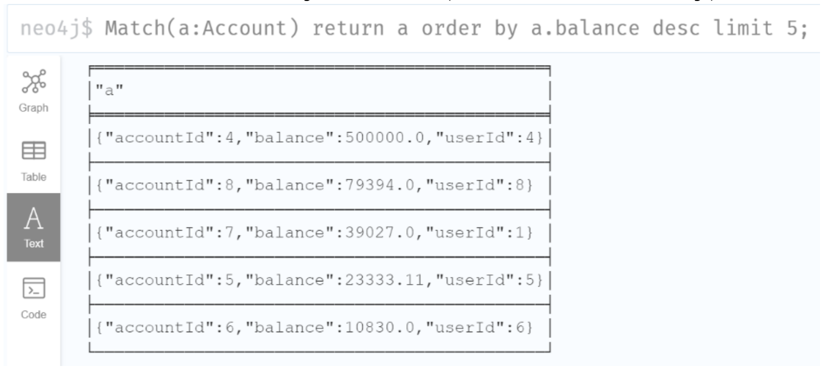
**# Get all the users**



```
neo4j$ Match(u:User) return u;
```

```
╒══════════════════════════════════════════════════════════╕
│"u"                                                       │
╞══════════════════════════════════════════════════════════╡
│{"lastName":"Novak","firstName":"Rick","phoneNumber":│
│"6179595701","userId":1,"email":"r.n@gmail.com"}         │
├──────────────────────────────────────────────────────────┤
│{"lastName":"Young ","firstName":"Larry","phoneNumber│
│":"6179595702","userId":2,"email":"l.y@gmail.com"}       │
├──────────────────────────────────────────────────────────┤
│{"lastName":"Liu","firstName":"Yanming","phoneNumber"│
│:"6179595703","userId":3,"email":"y.l@sina.com"}         │
├──────────────────────────────────────────────────────────┤
```

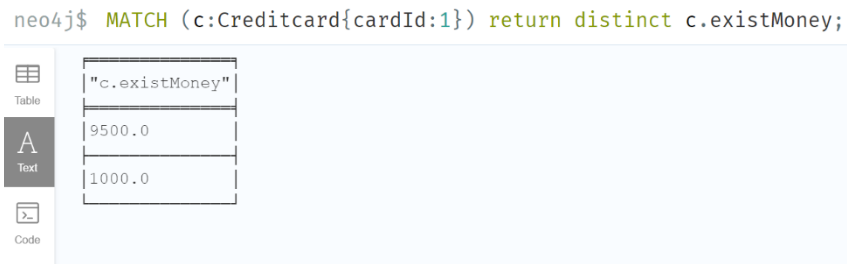
**# The account order by balance (first 5 records only)**



```
neo4j$ Match(a:Account) return a order by a.balance desc limit 5;
```

```
╒══════════════════════════════════════════════╕
│"a"                                           │
╞══════════════════════════════════════════════╡
│{"accountId":4,"balance":500000.0,"userId":4}│
├──────────────────────────────────────────────┤
│{"accountId":8,"balance":79394.0,"userId":8} │
├──────────────────────────────────────────────┤
│{"accountId":7,"balance":39027.0,"userId":1} │
├──────────────────────────────────────────────┤
│{"accountId":5,"balance":23333.11,"userId":5}│
├──────────────────────────────────────────────┤
│{"accountId":6,"balance":10830.0,"userId":6} │
└──────────────────────────────────────────────┘
```

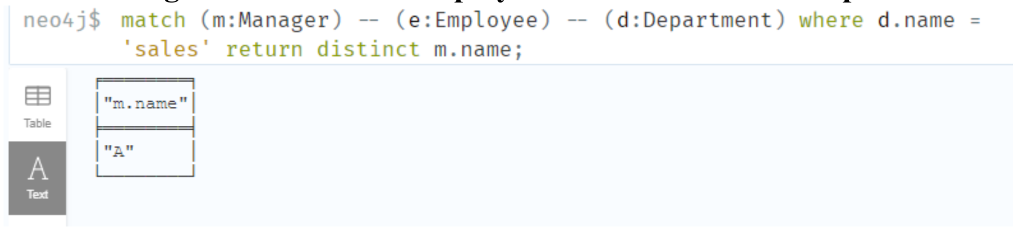
**# Find the distinct existing money from those credit cards with an Id of 1**



```
neo4j$ MATCH (c:Creditcard{cardId:1}) return distinct c.existMoney;
```

```
╒═══════════════╕
│"c.existMoney"│
╞═══════════════╡
│9500.0         │
├───────────────┤
│1000.0         │
└───────────────┘
```

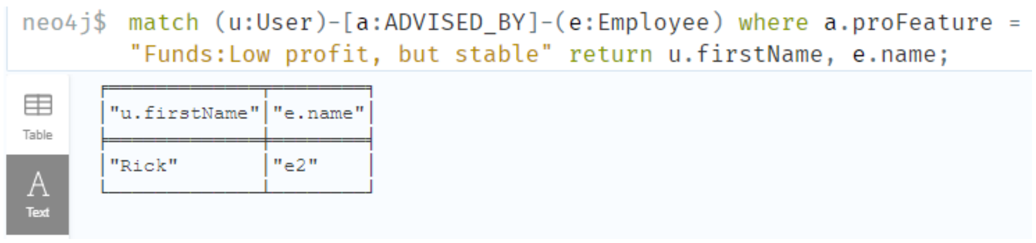
**# The manager's name whose employees works in the sales department.**



```
neo4j$ match (m:Manager) -- (e:Employee) -- (d:Department) where d.name =
       'sales' return distinct m.name;
```

```
╒════════╕
│"m.name"│
╞════════╡
│"A"     │
└────────┘
```

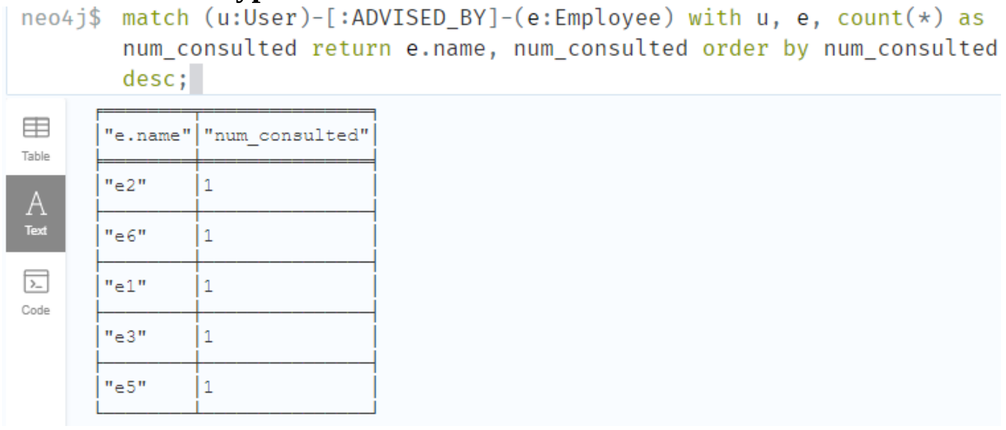
**# Find the user's firstName, employee's name in a consult conversation where the introducing product feature is "Funds: Low profit, but stable"**



```
neo4j$ match (u:User)-[a:ADVISED_BY]-(e:Employee) where a.proFeature =
       "Funds:Low profit, but stable" return u.firstName, e.name;
```

```
╒═════════════╤════════╕
│"u.firstName"│"e.name"│
╞═════════════╪════════╡
│"Rick"       │"e2"    │
└─────────────┴────────┘
```

**# Get a list of employees' names and their consulting times, in descending order. If you like to know which employee is consulted most times, feel free to add parameter limit 1 after the cypher code below.**



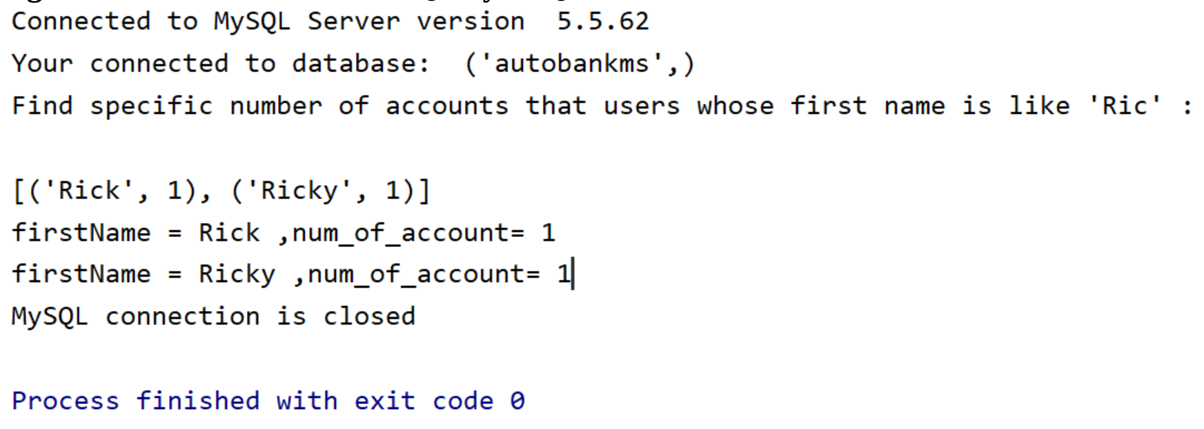
```
neo4j$ match (u:User)-[:ADVISED_BY]-(e:Employee) with u, e, count(*) as
       num_consulted return e.name, num_consulted order by num_consulted
       desc;
```

```
╒════════╤═══════════════╕
│"e.name"│"num_consulted"│
╞════════╪═══════════════╡
│"e2"    │1              │
├────────┼───────────────┤
│"e6"    │1              │
├────────┼───────────────┤
│"e1"    │1              │
├────────┼───────────────┤
│"e3"    │1              │
├────────┼───────────────┤
│"e5"    │1              │
└────────┴───────────────┘
```

## V. Database Access via Python

The database is accessed using Python and visualization of analyzed data is shown below. The connection of MySQL to Python is done using mysql.connector, followed by cursor.excecute to run and fetchall from query, followed by converting the list into a dataframe using pandas library and using matplotlib, tableau to plot the graphs for the analytics.

**Figure 1: Connection & Basic Query & Quit**

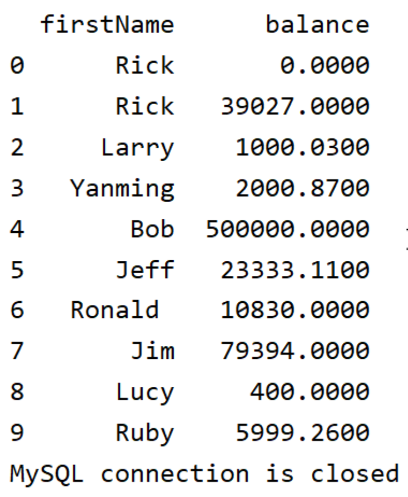


```
Connected to MySQL Server version  5.5.62
Your connected to database:  ('autobankms',)
Find specific number of accounts that users whose first name is like 'Ric' :

[('Rick', 1), ('Ricky', 1)]
firstName = Rick ,num_of_account= 1
firstName = Ricky ,num_of_account= 1
MySQL connection is closed

Process finished with exit code 0
```

**Figure 2: Top 10 User with highest balance in their account**



```
  firstName      balance
0      Rick       0.0000
1      Rick   39027.0000
2     Larry    1000.0300
3   Yanming    2000.8700
4       Bob  500000.0000
5      Jeff   23333.1100
6    Ronald   10830.0000
7       Jim   79394.0000
8      Lucy     400.0000
9      Ruby    5999.2600
MySQL connection is closed
```

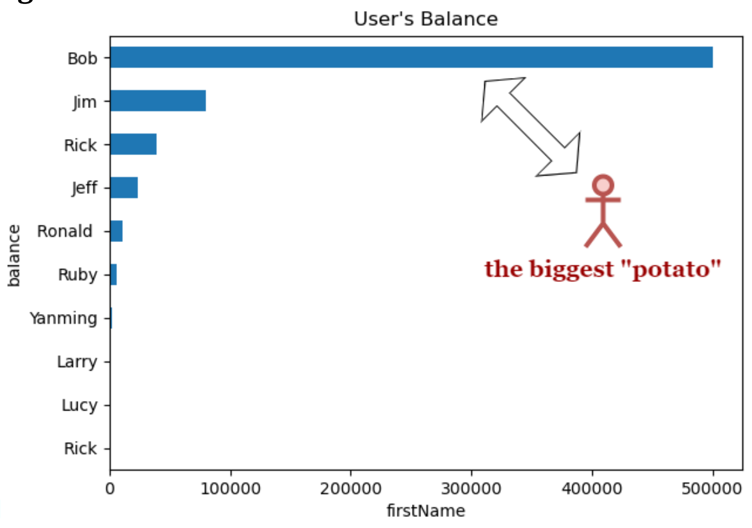
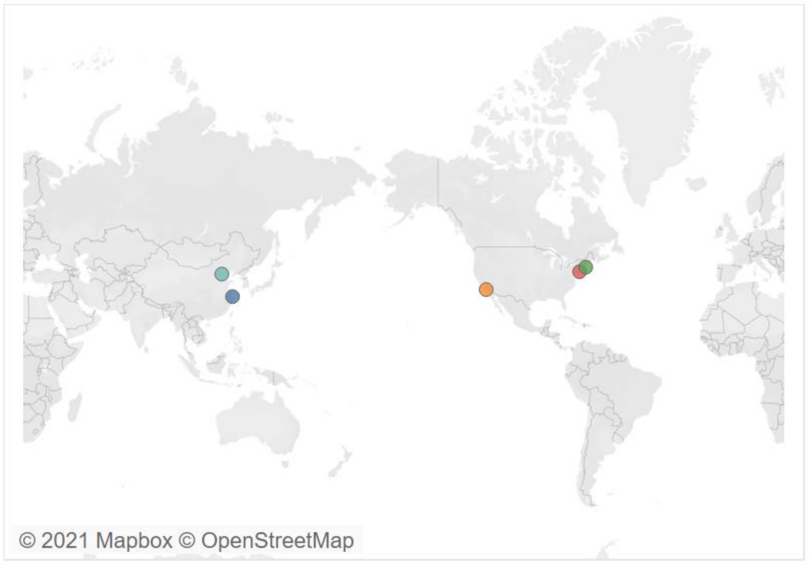


**Figure 3: Where are we?**

Geo-distribution of Dapartments



| Department | |
|---|---|
| 🟦 | after-sales service |
| 🟧 | analysis |
| 🟥 | consult |
| 🟦 | sales |
| 🟩 | tech |

Map based on Longitude (generated) and Latitude (generated). Color shows details about Department. Details are shown for various dimensions.

## Summary & Prospect

In this report, we established an investment database to meet users need, aiming to provide better financial purchase experience, extraordinary consultant service, and efficient product supply chain. We start by designing the conceptual model using EER, UML with specific requirement. Relationship Model is mapping under the normalization rules. Databases are set up with sample data and queries via MySQL, MongoDB, Cypher language. Python get access to the MySQL database, and visualize some insights by matplotlib package and tableau software under the preprocessed data using pandas package.

The trickiest part of the report lies in mapping different concepts of your model into a relational model in a normalized way. These concepts are like weak entity type, derived attribute type, unary relationship type, the cardinality of 1: N, M: N, specialization, aggregation. Also, ensure they are closed to the planned semantics.

Through comparison with MySQL and NoSQL, we conclude that MySQL is better for our start-up investment system, for the small or medium data size of our company at the beginning and its easy, quick, and powerful query capabilities. We need a lot of aggregation calculations for analysis on how to enhance our profits, locate the potential valuable users, and improve the customer services. That is exactly what MySQL works here. In future, we may immigrate our database to a NoSQL database gradually if distribute computing will be required due to the big data of customers. I believe that moment will reach someday.

## Reference

- https://www.wsj.com/articles/us-economy-september-2021-retail-sales-11634246791?mod=djem10point
- https://www.aba.com/about-us/press-room/industry-response-coronavirus
- https://www.federalreserve.gov/newsevents/pressreleases/monetary20200409a.htm
- https://www.economist.com/finance-and-economics/2015/07/02/beautifying-bankruptcy
- https://www.statista.com/statistics/1091911/age-financial-app-users-usa/
- https://fred.stlouisfed.org/series/REVEF52211ALLEST
- https://plantuml.com/
- Map-Reduce Examples — MongoDB Manual
- How-To: Neo4j ETL Tool - Developer Guides