`

***University of Pittsburgh***

***School of Information Sciences***

***School of Computing and Information & Department of Informatics and Networked Systems***

**INFSCI 2711: Advanced Topics in Database Management**

**Spring 2019**

**FINAL PROJECT REPORT**

**Bank Management System**

Group Members: RMDB: Jing Pang, Haoyang Qian

Neo4j: Ruochi Zhang, Qi Lu

MongoDB: Jie Chen, Rui Liu, Tianqi Xie

1. **Overview**

This project realizes a bank management system which ensures users of different levels have corresponding authority. The detailed description is followed.

1.1 User type:

There are five different types of users. Users of different levels have different function.

There are three types of our returned results. Balance refers to the transferred amount. Sum refers to the total amount of transactions. Count refers to the count of transactions. Here we list the same functions operated by our three database.

1.1.1 Customer:

* + Check all transactions, including balance, sum and count.
  + Check all transactions of a specific card, including balance, sum and count.
  + Check all transactions operated in a specific institute, including balance, sum and count.

1.1.2 Tellers:

* + Check all transactions of a day.

1.1.3 Chief:

Each institute has only one manager.

* + Check all transactions of the corresponding institute, including balance, sum and count.
  + Check all transactions of all specific tellers, including balance, sum and count.
  + Check all transactions of all accounts, including balance, sum and count.

1.1.4 President:

There is only one chief in the system.

* + Check the sum.
  + Check the balance.
  + Check the count.

2.1 Different functions for three databases

2.1.1 Relational Database

* + Data Warehouse

2.1.2 Neo4j

* + PageRank
  + Community Detection Algorithm

2.1.3 MongoDB

* + Geoinformation

**2. Assumptions**

This system is a bank management system. It is designed to realize the function of a bank such as search the information of transactions according to the different authority of users.

* Customers have the capability of checking their own transactions, including balance, sum and count. They can check all the transactions of their accounts, all transactions of a specific card and all transactions of a specific institute.
* Tellers can check all transactions of a day.
* Each bank has a chief and chiefs have the authority of checking all the transactions of banks they are responsible for. Chiefs can check all the transactions of corresponding institute, all transactions of a specific teller and all transactions of a specific account.
* There is only one president in the system. The president can check all the sum, balance and count. Besides, he can do the security detector.
* There are also three different functions for our three database. For the relational database, we focus on the data warehouse. For the neo4j, we apply pagerank and community detection algorithm in our system. For mongodb, we add the geoinfromnation.

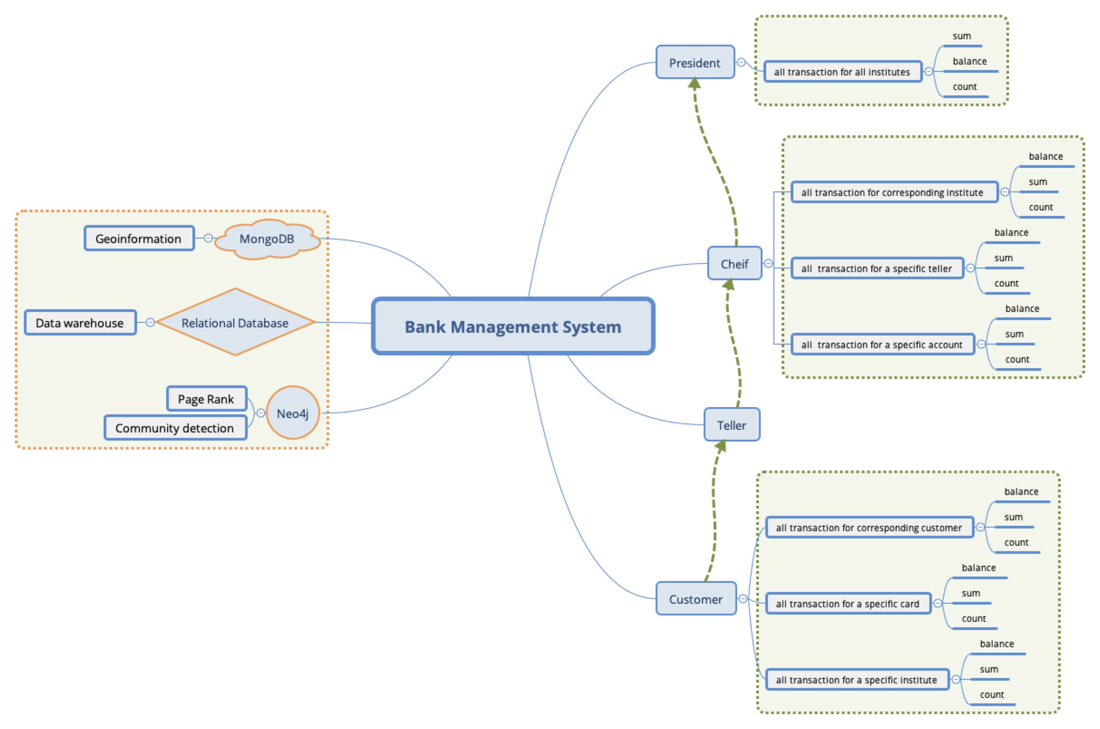


Figure 1: System architecture

**3. Data description**

For this project, we introduced 5 csv files as our data source, and each data source could be connected through specific attribute. The confidential\_info (Sheet 1) file contains bank employees’ information, including the id of bank tellers, bank governor, branch bank presidents and administrator. It also contains customers’ ids and all passwords. We use a specific file to store this information in order to maintain the security of the bank.

Sheet 1. confidential info

|  |  |  |
| --- | --- | --- |
|  | ID | passwd |
| 0 | 49744050 | fnp569891 |
| 1 | 88528785 | loj600073 |
| 2 | 63334672 | sbg923291 |
| 3 | 19815276 | kvf799106 |
| 4 | 44407636 | nlb734642 |
| 5 | 64704182 | ois939101 |
| 6 | 54355280 | ufu317913 |
| 7 | 13017357 | mzr848105 |
| 8 | 34211586 | bzx642303 |
| 9 | 59661975 | uhv675286 |
| ... | ... | ... |

The second file is cust\_acct.csv (Sheet 2), which contains the customers’ ID, and their corresponding account id. In our bank, system, each customer can obtain more than one card and on average, each customer has three cards. Basically, we can figure out the relationship between customers and bank account information that they correlated with.

Sheet 2. cust\_acct

|  |  |  |
| --- | --- | --- |
|  | CUST\_ID | ACCT\_ID |
| 0 | 39697816 | 2265871694 |
| 1 | 39697816 | 4567905843 |
| 2 | 39697816 | 4211888405 |
| 3 | 39697816 | 3241956117 |
| 4 | 39697816 | 5170732060 |
| 5 | 39697816 | 8648357885 |
| 6 | 79657263 | 5319933640 |
| 7 | 79657263 | 3629733703 |
| ... | ... | ... |

The third file is customer\_tb.csv (Sheet 3), which has information of customers personal information, such as gender, age, first date of opening card and the customer id. This file can be connected to the cust\_acct file by customer id.

Sheet 3. customer\_tb

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | C\_GENDER | C\_AGE | C\_FIRST\_WORK\_DATE | CUST\_CODE |
| 0 | 1 | 53 | 1-Aug-86 | 49744050 |
| 1 | 2 | 49 | 1-Dec-90 | 88528785 |
| 2 | 2 | 25 | 23-Feb-16 | 63334672 |
| 3 | 2 | 53 | 1-Dec-81 | 19815276 |
| 4 | 2 | 38 | 1-Aug-03 | 44407636 |
| 5 | 2 | 53 | 1-Nov-84 | 64704182 |
| 6 | 2 | 35 | 1-Jul-05 | 54355280 |
| 7 | 2 | 33 | 10-May-10 | 13017357 |
| ... | ... | ... | ... | ... |

The forth file is the teller.csv (Sheet 4). In this file, we stored some details of bank workers, work id, bank id and worker’s level. The level hierarchy represents different authority of accessing the database. For instance, the low-level bank tellers could only review the transactions that operated by them and cannot check out the account number of that transaction.

Sheet 4. Teller

|  |  |  |  |
| --- | --- | --- | --- |
|  | GUIYDAIH | YNGYJIGO | level |
| 0 | 1041 | 601101 | 1 |
| 1 | 4053 | 601104 | 1 |
| 2 | 7020 | 601129 | 1 |
| 3 | 15005 | 601104 | 0 |
| 4 | 11024 | 601114 | 1 |
| 5 | 1062 | 601138 | 1 |
| 6 | 15025 | 600001 | 1 |
| 7 | 17003 | 601101 | 0 |
| ... | ... | ... | ... |

The last file is also an essential file called transaction.csv (Sheet 5), which contains transaction information such as transaction time, operator id, transaction number, account id and so on. We connect these files by customer ID, transaction ID and employee ID.

Sheet 5. transaction

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | TR\_TM | TR\_NO | OPR\_ID | CUST\_ID | ACCT\_ID | TR\_AM | OPP\_ACCT\_ID |
| 0 | 15-Aug-18 | IN00110117000044 | IN0011 | 49744050 | 9422937241 | 2215 | 4161831387 |
| 1 | 15-Aug-18 | IN00290117000025 | IN0029 | 88528785 | 2553396999 | 169684 | 8852694812 |
| 2 | 15-Aug-18 | IN00110117000010 | IN0011 | 63334672 | 3842340517 | 856189 | 8518307638 |
| 3 | 15-Aug-18 | IN00170117000076 | IN0017 | 19815276 | 5174142869 | 204857 | 7259062305 |
| 4 | 15-Aug-18 | IN00070117000031 | IN0007 | 44407636 | 4809166321 | 548499 | 9041482456 |
| 5 | 15-Aug-18 | IN00020117000024 | IN0002 | 64704182 | 3179422749 | 443569 | 7616159291 |
| 6 | 15-Aug-18 | IN00360117000055 | IN0036 | 54355280 | 1214916287 | 834318 | 5436805195 |
| 7 | 15-Aug-18 | IN00050117000003 | IN0005 | 13017357 | 2964596426 | 462040 | 7644042537 |
| ... | ... | ... | ... | ... | ... | ... | ... |

**4. Data loading process**

In the stage of data preprocessing, we removed some of the fields that were not useful for analysis. Some fields have missing values, and fields with too many missing values are also deleted. If there are only a few fields with missing values, we use the average value or 0 instead. Because we need to make a system that can log in, we randomly generate passwords for all login clients.

MySQL:

Firstly, we create a database in MySQL named “adb”. Secondly, we use DDL language we write for creating 5 original tables to create tables. Then we import data from csv files. After creation of 5 original tables, we create fact tables and dimension tables. Data in fact tables and dimension tables are inserted by SQL statements from 5 original tables. In the process of using the database, we have the attributes we want to add, so we will create new tables and insert data several times to make our tables easier to use.

MongoDB:

First, we create a database named “bankinfo” to store all the data. Then, we use mongoimport to import 5 files as 5 collections which are “confidential” – store all the Id and password, “customer” – store customer basic information like age and gender, “teller” – store teller’s id and institute, “cust\_acct” – store the mapping between customer id and account id, “transaction” – store the transactions , also create a collection “institute\_geo” based on institutes and random generated longitude and latitude, this collection stores all the geospatial information.

Neo4j:

I create Six of entities which are “Account”, “Chief”, “Customer”, “Hall”, “President”, “Teller”. There are too many relationships so I create two indexes for the attribute ID of “Account” and “Teller”. Then I create six relationships between the entities which are “belongTo” between “Account” and “Customer”, “manage” between “Cheif” and “Hall”, “operate” between “Teller” and “Account”, “transaction” between “Account” and “Account”, “rule” between “Cheif” and “Hall”, “workIn” between “Teller” and “Hall”. I use the Python Driver To execute the command looply.

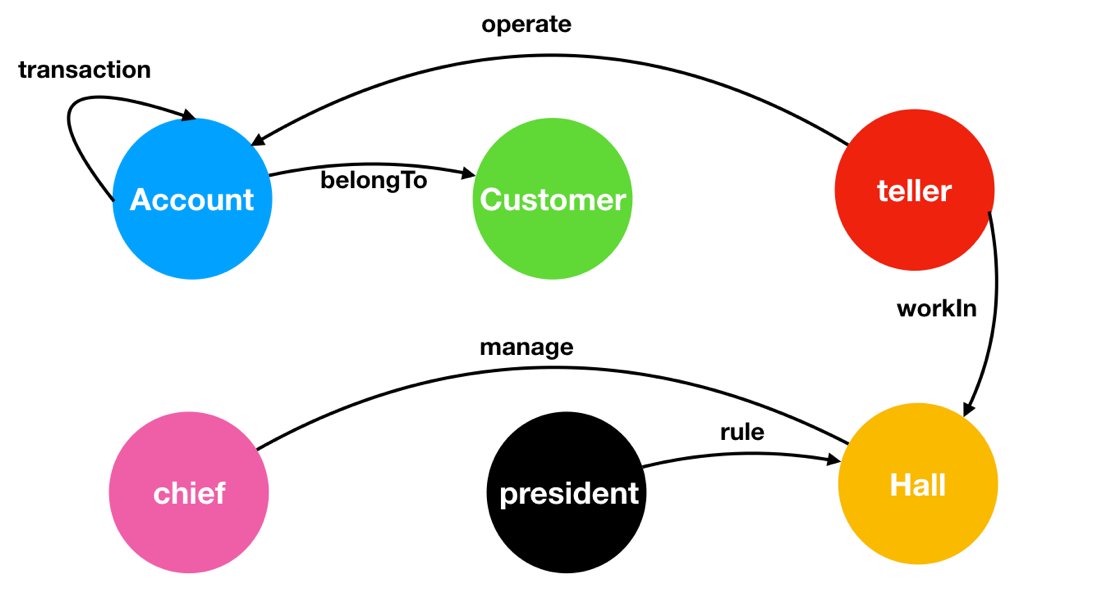


Figure 2: Relationships in neo4j

**5. MySQL**

5.1 Tables

Based on the 5 tables we want to use, we firstly create these tables and import date into these tables. The DDL languages to create these 5 tables are shown as below.

5.2 cust\_acct table

CREATE TABLE IF NOT EXISTS cust\_acct(

tb\_id INT ,

CUST\_ID int,

ACCT\_ID BIGINT,

VALIAD\_PERIOD int,

ACCT\_LEVEL int,

PRIMARY KEY (tb\_id) ) ;

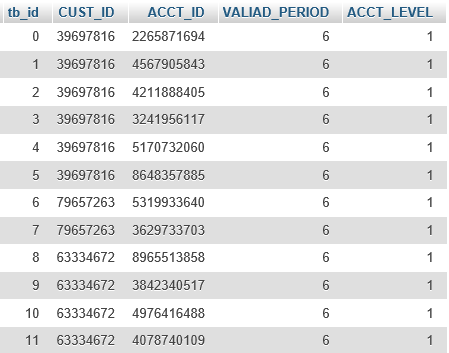


Figure 3: cust\_acct table

CREATE TABLE IF NOT EXISTS teller(

tb\_id INT ,

GUIYDAIH VARCHAR(100),

YNGYJIGO int,

level int,

GENDER VARCHAR(10),

PRIMARY KEY (tb\_id) ) ;

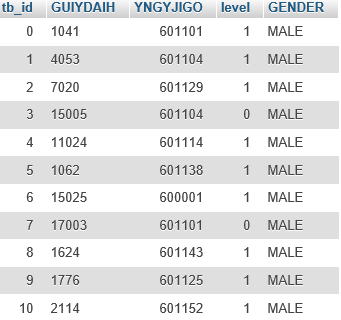


Figure 4: teller table

CREATE TABLE IF NOT EXISTS transaction(

tb\_id INT,

TR\_TM date,

TR\_NO VARCHAR(100),

OPR\_ID VARCHAR(100),

CUST\_ID int,

ACCT\_ID BIGINT,

TR\_AM int,

OPP\_ACCT\_ID BIGINT,

PRIMARY KEY (tb\_id) ) ;



Figure 5: transaction table

create table if not exists customer\_tb(

tb\_id int,

C\_GENDER int,

C\_AGE int,

C\_FIRST\_WORK\_DATE date,

CUST\_CODE int,

primary key (tb\_id)

);

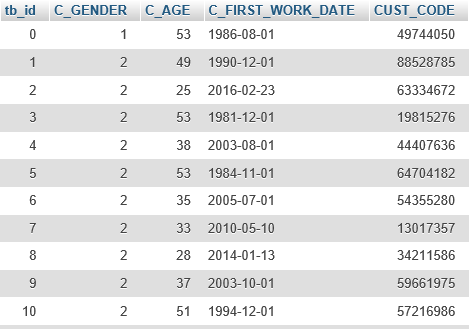


Figure 6: customer\_tb table

create table if not exists confidentical\_info(

tb\_id int,

ID VARCHAR(200),

passwd varchar (100),

primary key (tb\_id)

);

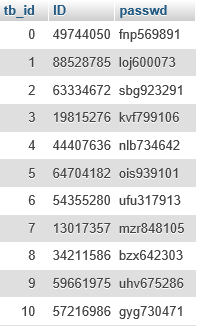


Figure 7: confidentical\_info table

5.3 Star Schema Design

To design star schemas for this project, we need to decide what FACT tables and what dimension tables we choose to use. From previous designed 5 tables, we choose to use CustInFact table, CustOutFact table and OperatorFact Table finally. CustInFact table illustrates account entry of customers. CustOutFact table illustrates account billing of customers. OperatorFact table illustrates information about operators.

We construct three FACT tables from the original dataset. The tables are based on the logistic usage of bank. The FACT table Cust\_In\_Fact construct a table about the information of amount of money transferred into a account in three dimensions. The FACT table Cust\_Out\_Fact built in similar logic about the money transferred out to a account. The last one Operator\_Fact combined a table about numbers of transaction made by the operator.

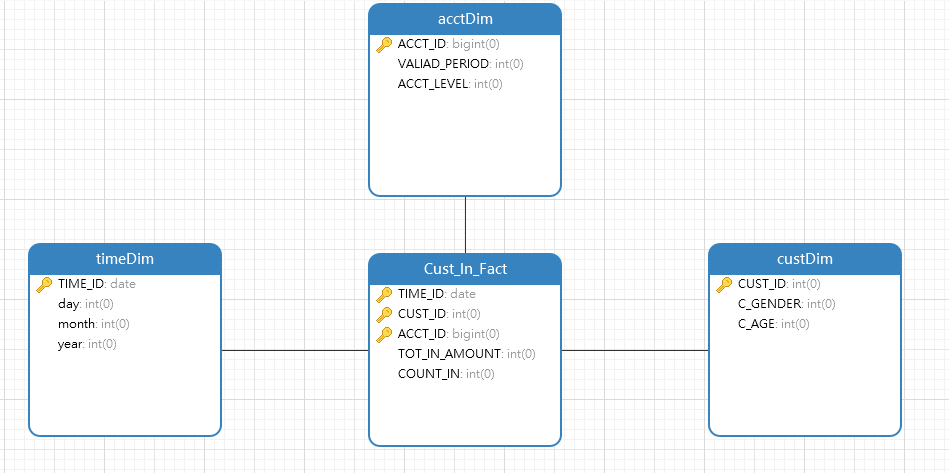


Figure 8: CustIn



Figure 9: CustOut

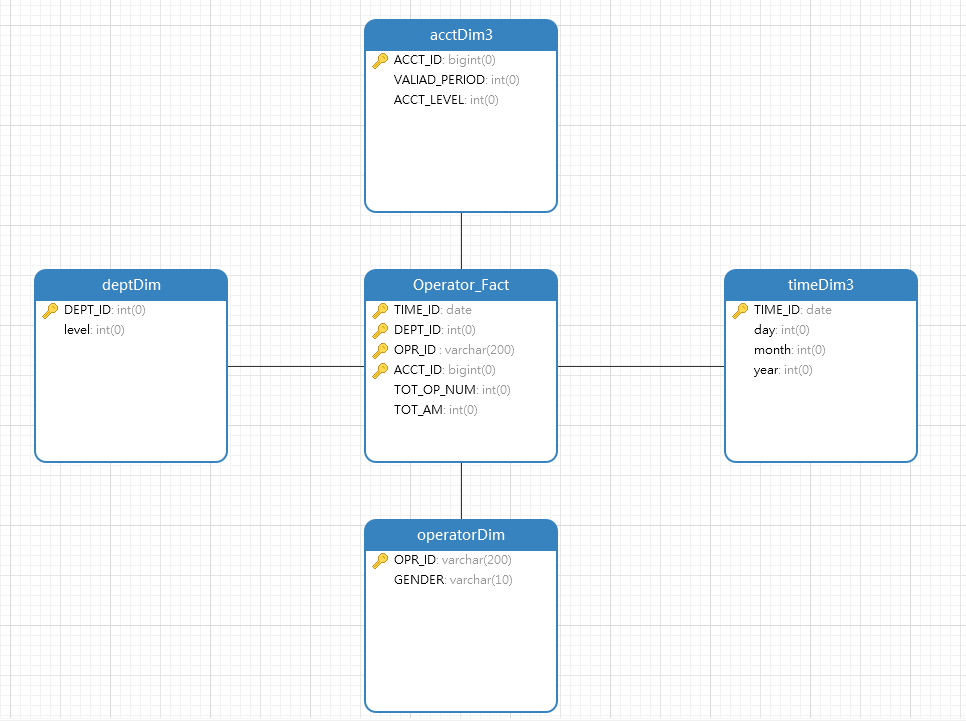


Figure 10: Operator

In order to create these tables, we use CREATE TABLE command to build.

create table if not exists CustOutFact(

TIME\_ID date,

CUST\_ID int,

ACCT\_ID bigint,

TOT\_OUT\_AMOUNT int,

primary key (TIME\_ID, CUST\_ID, ACCT\_ID)

);

create table if not exists CustInFact(

TIME\_ID date,

CUST\_ID int,

ACCT\_ID bigint,

TOT\_IN\_AMOUNT int,

primary key (TIME\_ID, CUST\_ID, ACCT\_ID)

);

create table if not exists timeDim(

TIME\_ID date,

day int,

month int,

year int,

primary key (TIME\_ID)

);

create table if not exists custDim(

CUST\_ID int,

C\_GENDER int,

C\_AGE int,

primary key (CUST\_ID)

) ;

create table if not exists acctDim(

ACCT\_ID bigint,

VALIAD\_PERIOD int,

ACCT\_LEVEL int,

primary key (ACCT\_ID)

);

create table if not exists OperatorFact(

TIME\_ID date,

DEPT\_ID int,

OPR\_ID varchar(200),

TOT\_OP\_NUM int,

primary key (TIME\_ID, DEPT\_ID, OPR\_ID)

);

create table if not exists deptDim(

DEPT\_ID int,

level int,

primary key (DEPT\_ID)

);

create table if not exists operatorDim(

OPR\_ID varchar(200),

GENDER varchar(10),

primary key (OPR\_ID)

);

create table if not exists timeDim(

TIME\_ID date,

day int,

month int,

year int,

primary key (TIME\_ID)

);

To fulfill our desired, we insert data into tables as following logics.

INSERT INTO CustOutFact

SELECT TR\_TM AS TIME\_ID, CUST\_ID, ACCT\_ID, sum(TR\_AM) AS TOT\_OUT\_AMOUNT

FROM transaction

WHERE NOT EXISTS

(

SELECT \*

FROM CustOutFact

WHERE transaction.TR\_TM = CustOutFact.TIME\_ID

and transaction.CUST\_ID = CustOutFact.CUST\_ID

and transaction.ACCT\_ID = CustOutFact.ACCT\_ID

)

GROUP BY TR\_TM, CUST\_ID, ACCT\_ID

;

INSERT INTO CustInFact

SELECT TR\_TM AS TIME\_ID, CUST\_ID , OPP\_ACCT\_ID AS ACCT\_ID, sum(TR\_AM) AS TOT\_IN\_AMOUNT

FROM transaction

WHERE NOT EXISTS

(

SELECT \*

FROM CustInFact

WHERE transaction.TR\_TM = CustInFact.TIME\_ID

and transaction.CUST\_ID = CustInFact.CUST\_ID

and transaction.OPP\_ACCT\_ID = CustInFact.ACCT\_ID

)

GROUP BY TR\_TM, CUST\_ID, OPP\_ACCT\_ID

;

INSERT INTO timeDim

SELECT TR\_TM AS TIME\_ID, DAY(TR\_TM) AS day, MONTH(TR\_TM) AS month, YEAR(TR\_TM) AS YEAR

FROM transaction

WHERE NOT EXISTS

(

SELECT TIME\_ID

FROM timeDim

WHERE transaction.TR\_TM = timeDim.TIME\_ID

)

GROUP BY TR\_TM

;

INSERT INTO custDim

SELECT CUST\_CODE AS CUST\_ID, C\_GENDER, C\_AGE

FROM customer\_tb

WHERE NOT EXISTS

(

SELECT \*

FROM custDim

WHERE customer\_tb.CUST\_CODE = custDim.CUST\_ID

)

;

INSERT INTO acctDim

SELECT ACCT\_ID, VALIAD\_PERIOD, ACCT\_LEVEL

FROM cust\_acct

WHERE NOT EXISTS

(INSERT INTO OperatorFact

SELECT t.TR\_TM AS TIME\_ID, te.YNGYJIGO AS DEPT\_ID, t.OPR\_ID, count(TR\_NO) AS TOT\_OP\_NUM

FROM transaction t, teller te

WHERE t.OPR\_ID = te.GUIYDAIH

AND NOT EXISTS(

SELECT \*

FROM OperatorFact

WHERE t.TR\_TM = OperatorFact.TIME\_ID

and te.YNGYJIGO = OperatorFact.DEPT\_ID

and t.OPR\_ID = OperatorFact.OPR\_ID

)

GROUP BY t.TR\_TM, te.YNGYJIGO, t.OPR\_ID

;

INSERT INTO deptDim

SELECT YNGYJIGO AS DEPT\_ID, level

FROM teller

WHERE NOT EXISTS

(

SELECT

\*

FROM

deptDim

WHERE teller.YNGYJIGO = deptDim.DEPT\_ID)

;

INSERT INTO operatorDim

SELECT GUIYDAIH AS OPR\_ID, GENDER

FROM teller

WHERE NOT EXISTS

(

SELECT

\*

FROM

operatorDim

WHERE teller.GUIYDAIH = operatorDim.OPR\_ID

)

GROUP BY GUIYDAIH, GENDER

;

SELECT \*

FROM acctDim

WHERE cust\_acct.ACCT\_ID = acctDim.ACCT\_ID )

;

We created some pre-aggregated summary tables as shown above. All statements for creation and populating of the summary tables are written. The data in our database is transaction-oriented before the aggregation operation. After pre-aggregation, we organized the data according to certain themes. Each topic corresponds to a macroscopic field of analysis. The data warehouse excludes data that is useless for decision making, providing a concise view of a particular topic.

In nightly scheduled batch job, we want to maintain our data consistency in database. Therefore, we rerun a set of SQL language (INSERT INTO) from previous part. This will make sure that the latest information will update daily.

5.4 Functions and Implementations

5.4.1 For customers

5.4.1.1 All transactions for this customer

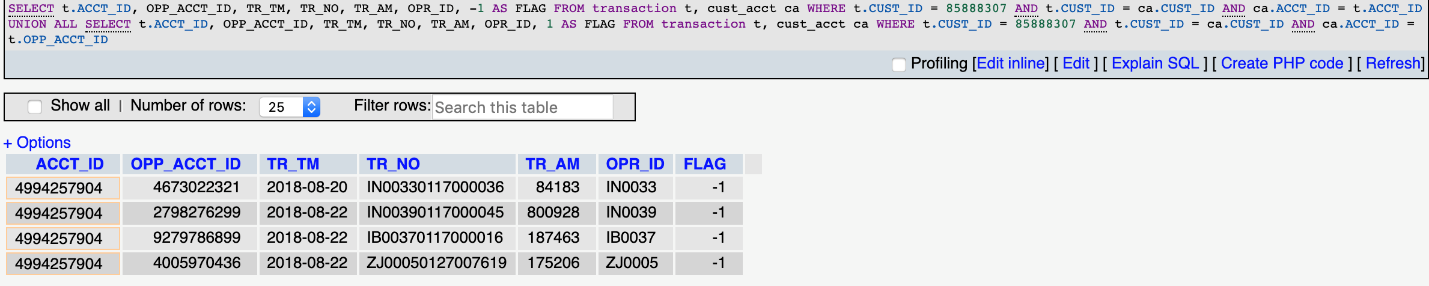


Figure 11: Customer Transaction

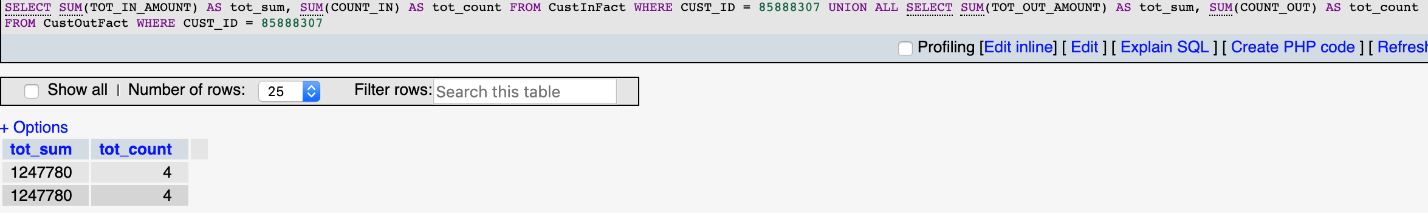


Figure 12: Customer Aggregation

5.4.1.2 All transactions for a specific card

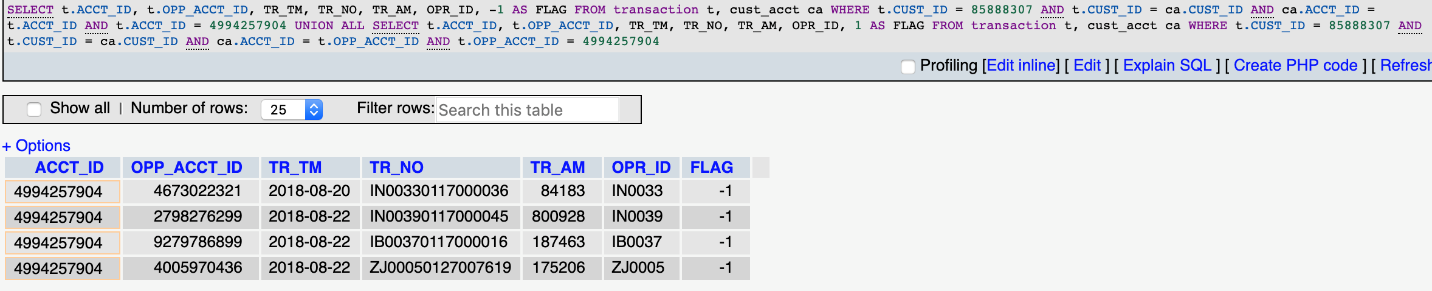


Figure 13: Card Transaction

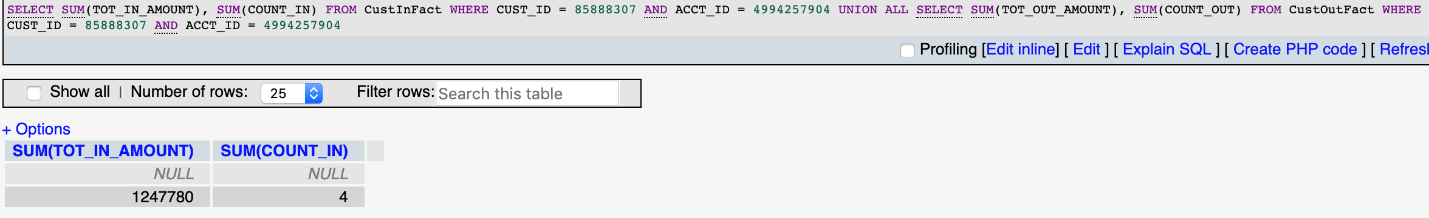


Figure 14: Card Aggregation

5.4.1.3 All transactions for a specific institute

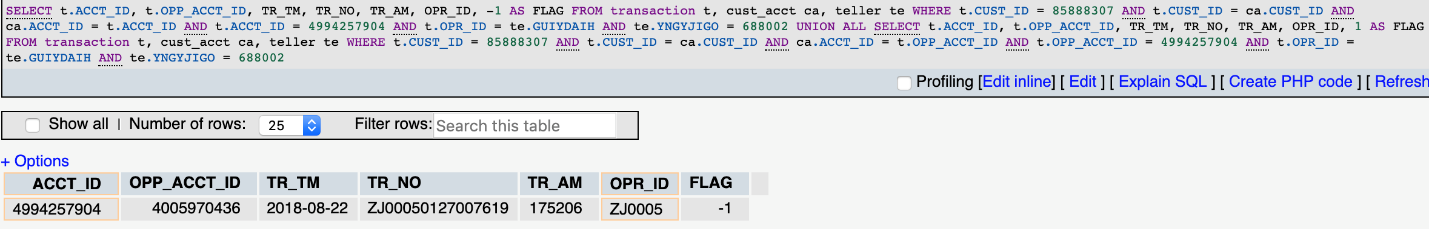


Figure 15: Institute Transaction

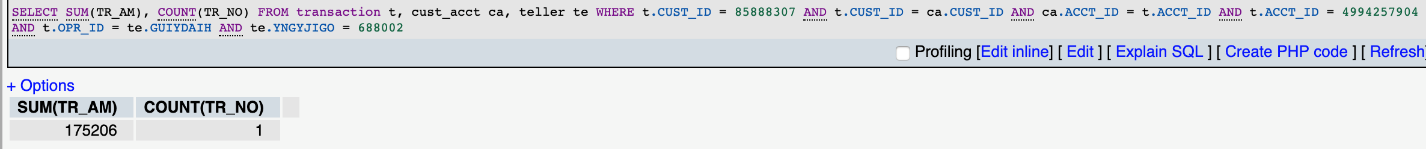


Figure 16: Institute Aggregation

5.4.2 For tellers

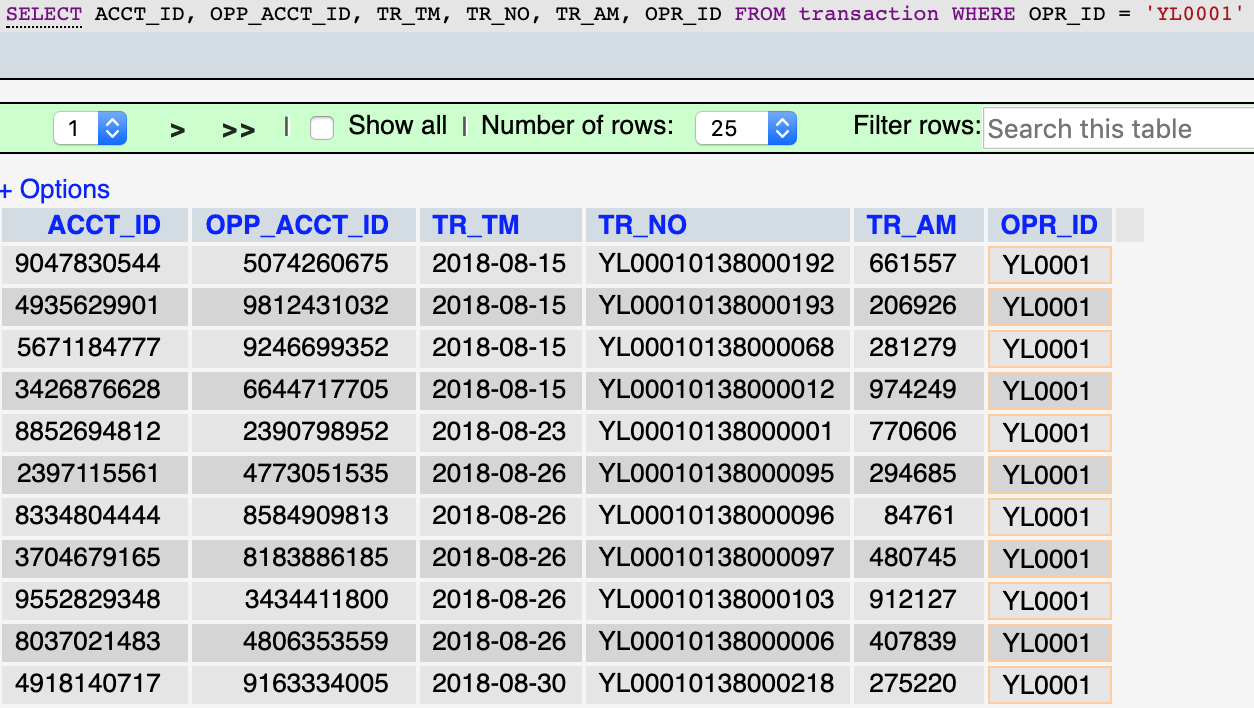


Figure 17: Teller Transaction

5.4.3 For chiefs

5.4.3.1 All transactions for all corresponding institutes

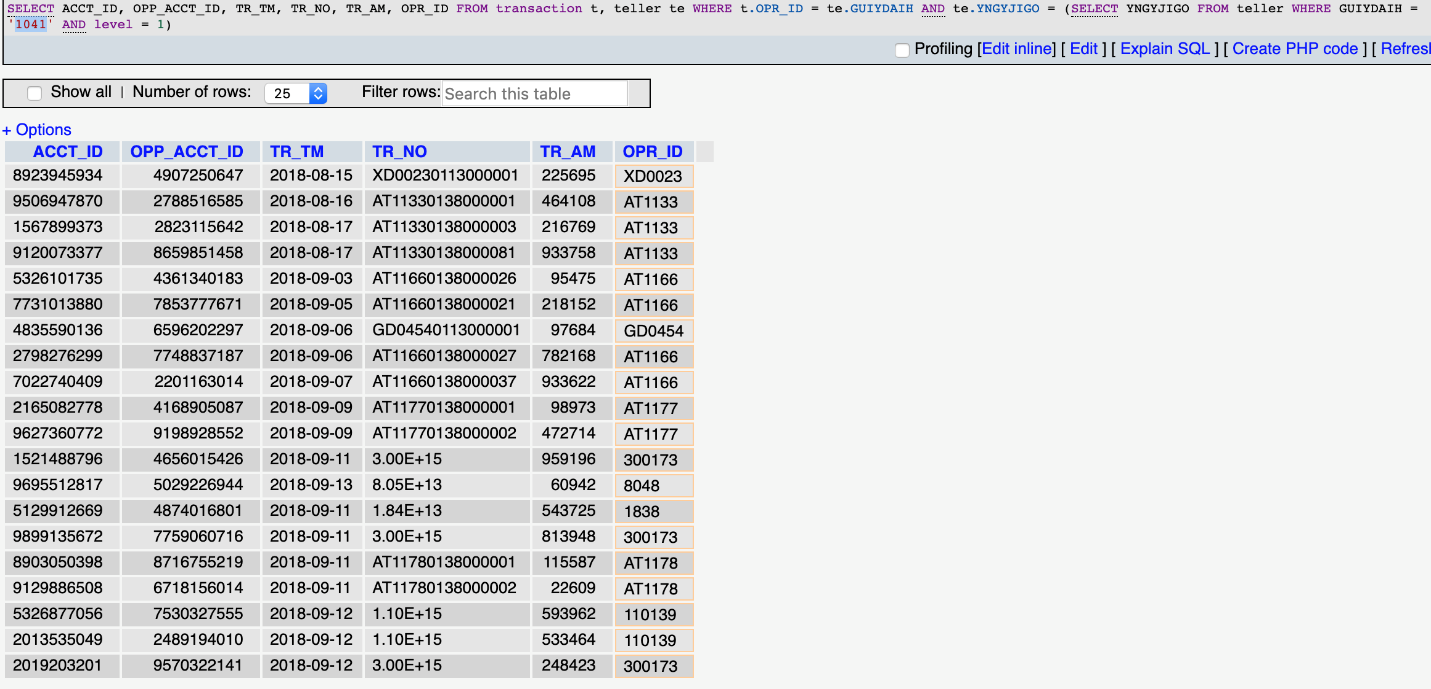


Figure 18: Chief Institute

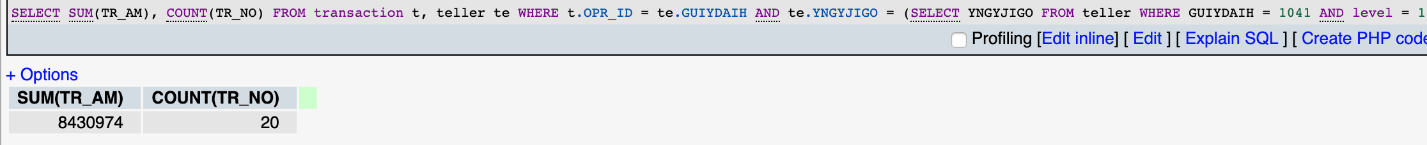


Figure 19: Chief Institute Aggregation

5.4.3.2 All transactions for all specific tellers

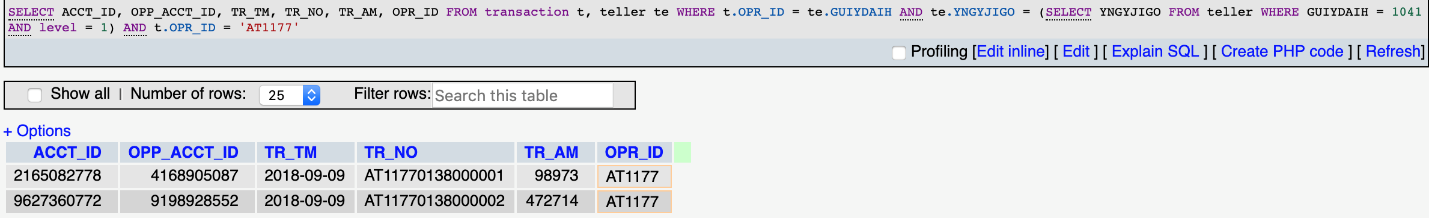


Figure 20: Chief Teller

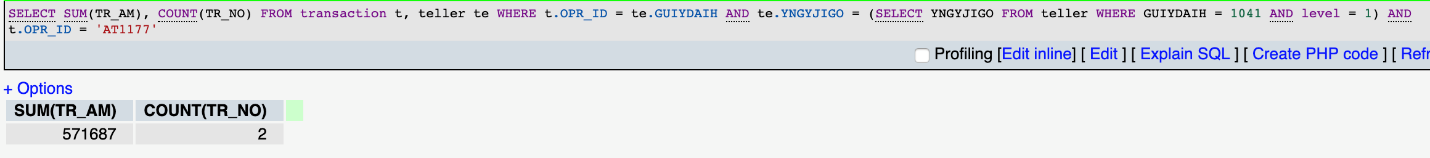


Figure 21: Chief Teller Aggregation

5.4.3.3 All transactions for all accounts

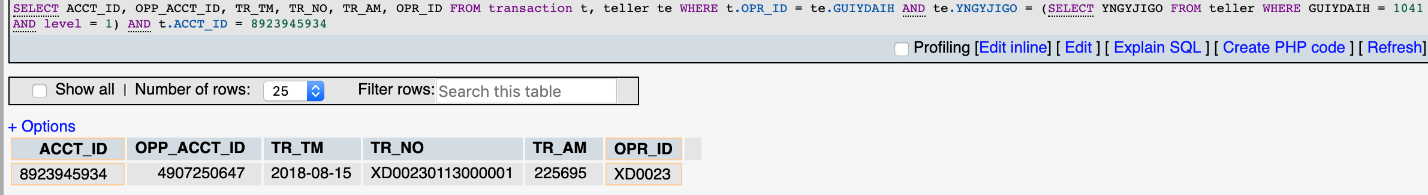


Figure 22: Chief Account

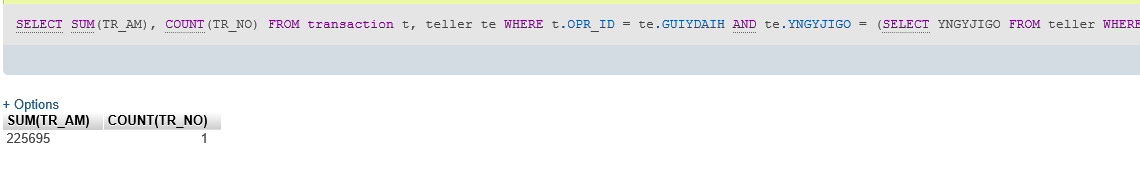


Figure 23: Chief Account Aggregation

5.4.4 For president

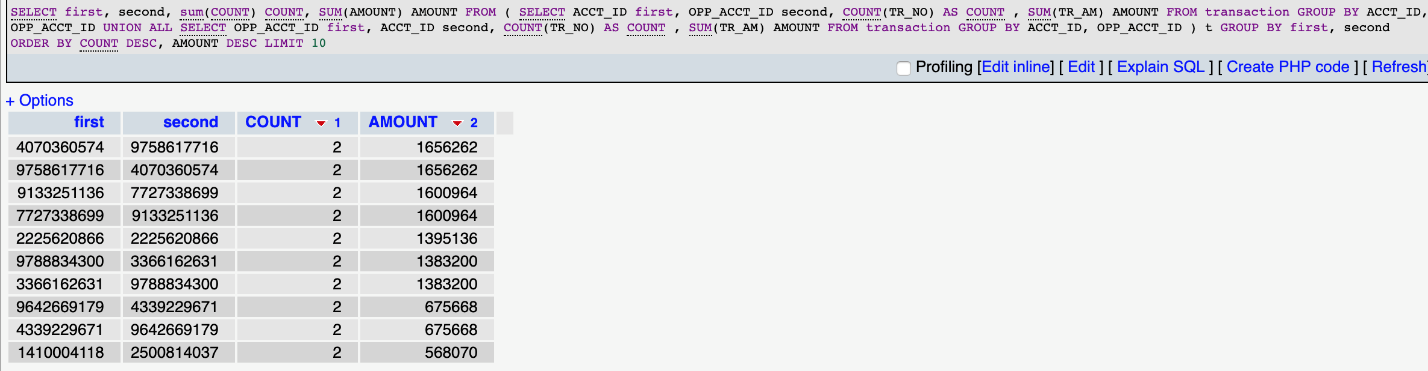


Figure 24: President Detector

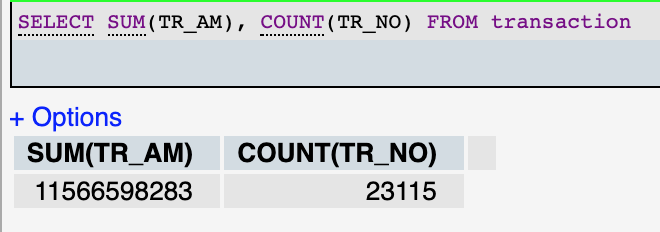


Figure 25: President Aggregation

From the tables above, we used some Data Warehouse queries. In our daily transactions, sometimes we only need to query the transaction table to get the results we want, but sometimes we need to make use of some fact tables to make the query simpler and easier to understand. The data warehouse is integrated. Data in the data warehouse comes from operational data. The required data is extracted from the original data for processing and integration. After unified and integrated, the data warehouse can be accessed. The data warehouse mainly provides data for decision analysis, and the operations involved are mainly data query. Data warehousing is generated in order to further mine data resources and make decisions based on the needs of the database.

The data analyzed by the data warehouse is generally divided into day, week, month, season, year, etc. The front ends must be able to enter the time period for query to meet the requirements of the data warehouse. Daily cycle data requires the highest efficiency. Customers should be able to see data analysis of the past transactions as soon as possible. Due to the large amount of data on the daily basis of the bank, well designed data warehouses are supposed to give data within 24 hours or even 12 hours.

**6. Neo4j**

For Neo4j, we create six entities and six relationships according to our data.

1. Entity

1.1 Customer

Customer entity has four attributes: id, C\_FIRST\_WORK\_DATE, C\_Gender and C\_AGE.

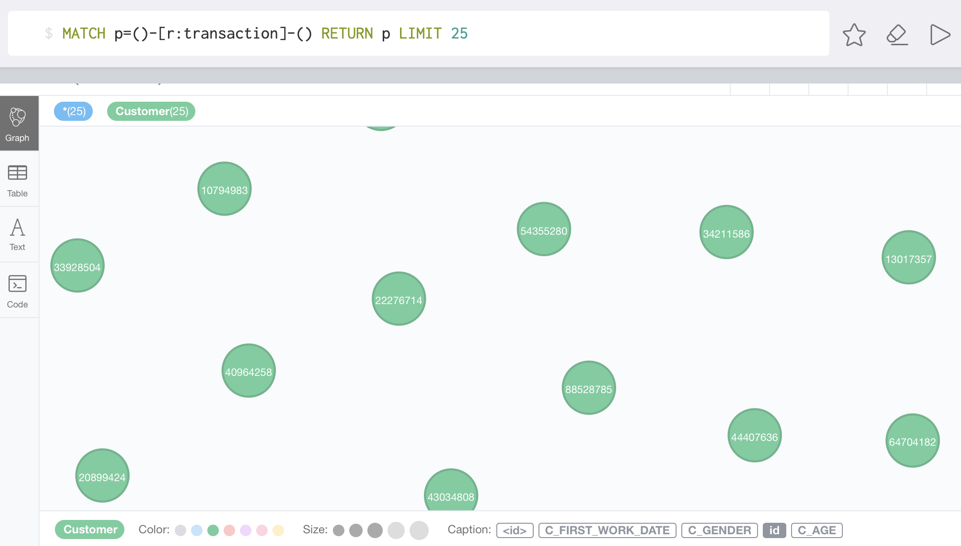


Figure 26: Customer Entity

1.2 Account

Account entity has one attribute: id.

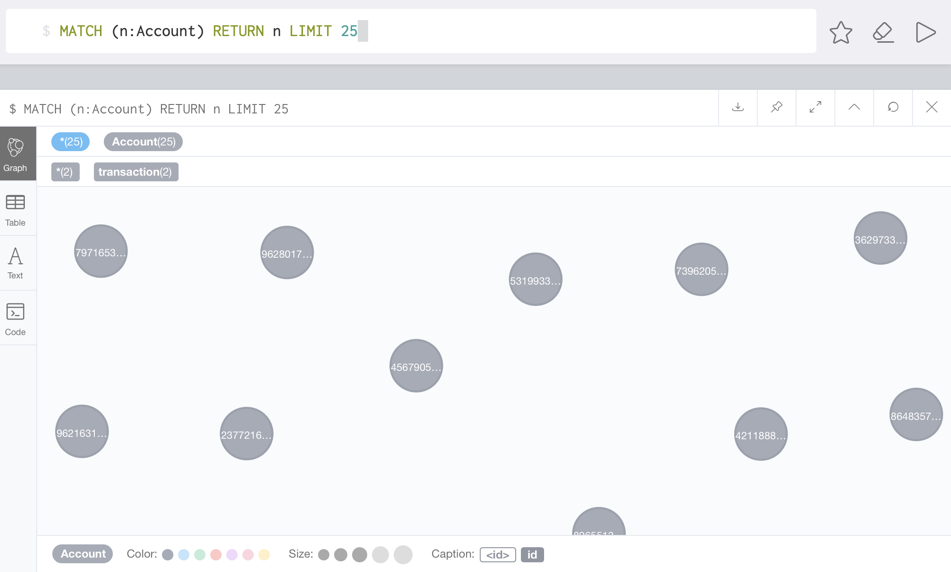


Figure 27：Account Entity

1.3 Hall

Hall entity stands for the bank. It has one attribute: id.

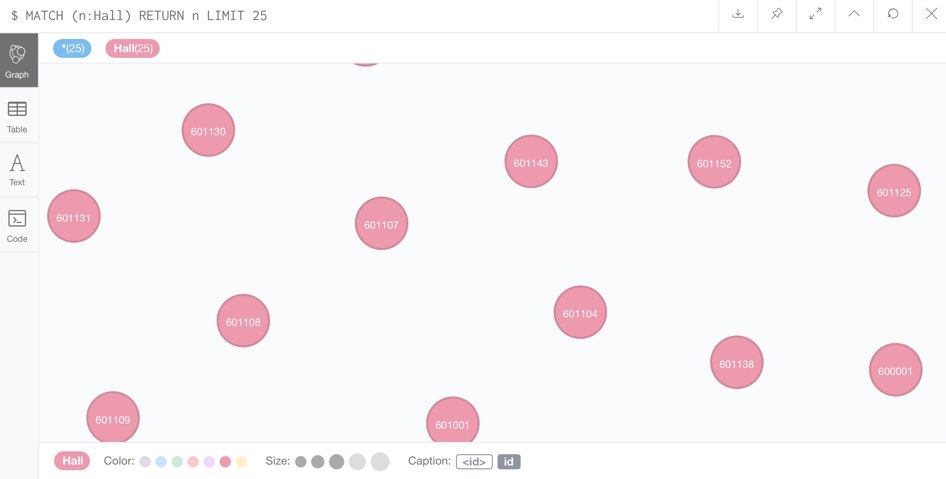


Figure 28: Hall Entity

1.4 Teller

Teller entity has three attributes: id, level and hall.

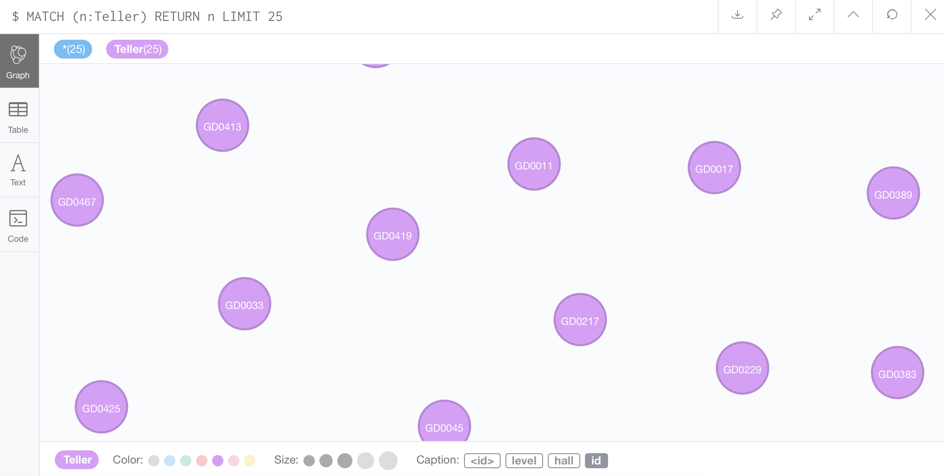


Figure 29: Teller Entity

1.5 Chief

Chief entity has three attributes: id, level and hall.



Figure 30: Chief Entity

1.6 President

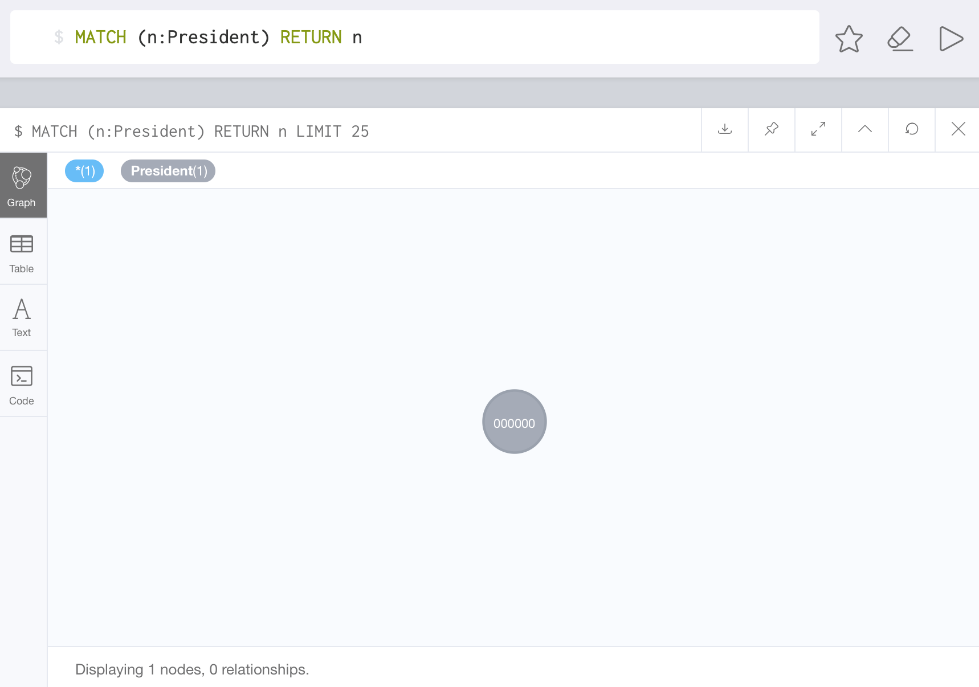


Figure 31: President Entity

1. Relationship

2.1 Belong To

The ‘belong to’ stands for the relationship between customer and the card. Account stands for the card and each account belongs to a customer. One customer can have more than one accounts.

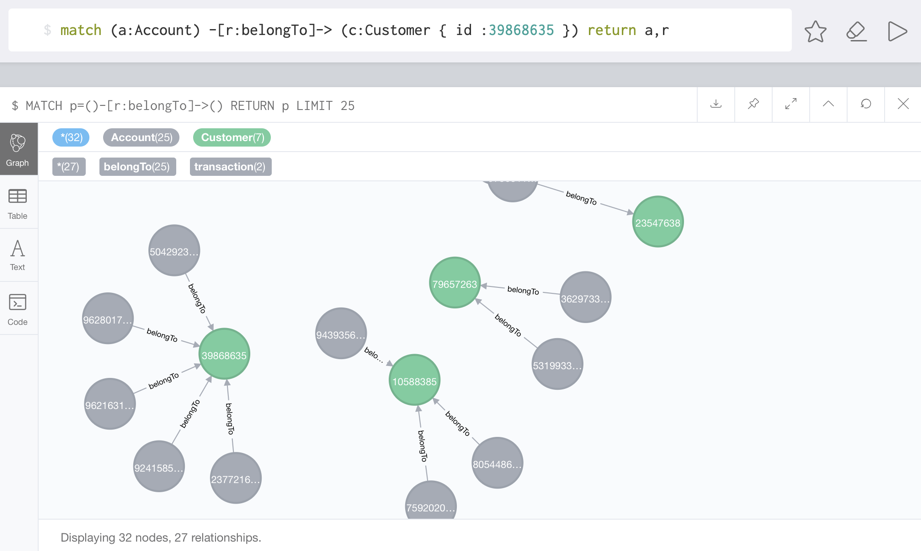


Figure 32: Belong To Relationship

2.2 Transaction

Transaction is the relationship between accounts. This relationship shows the direction of transfer of money between two accounts. One account has more than one transaction records.

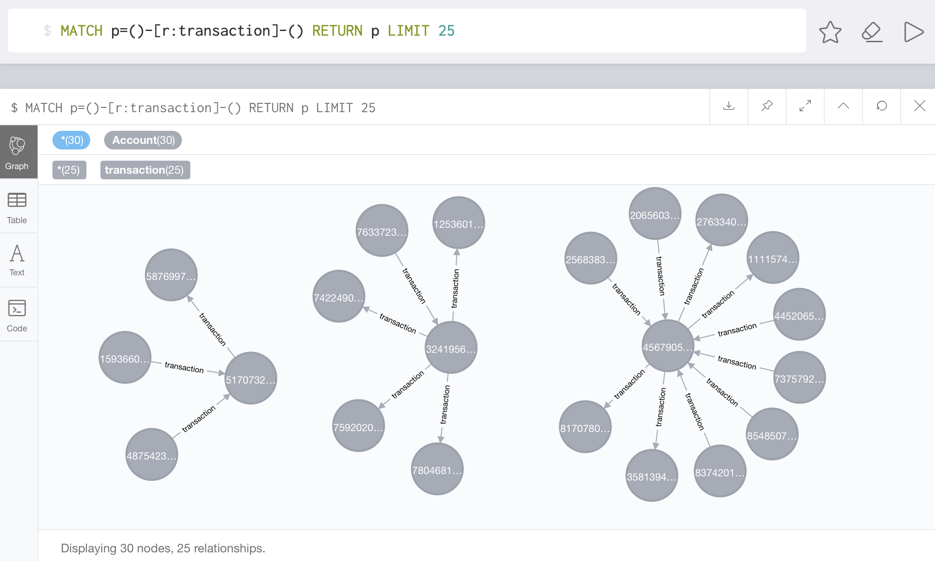


Figure 33: Transaction Relationship

2.3 Serve

This is the relationship between the teller and the account. One account can be operated by different tellers.



Figure 34: Serve Relationship

2.4 Manage Relationship

This is the relationship between chiefs and banks. Each bank has a chief. The chief has the authority to check the transactions of corresponding banks.

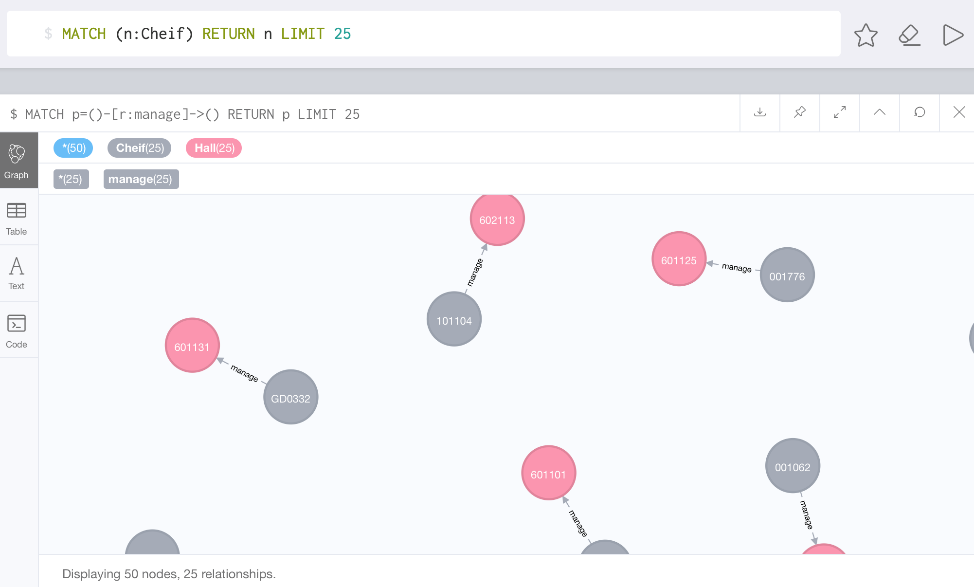


Figure 35: Manage Relationship

2.5 Rule Relationship

This is the relationship between the president and all the banks. The president has the highest authority to rule all the banks.



Figure 36: Rule relationship

2.6 WorkIn Relationship

This is the relationship between tellers and the bank. Each bank has many tellers work in it.



Figure 37: WorkIn Relationship

1. Function

We implement the functions by different type of user.

3.1 Customer

* Check all transactions, including balance, sum and count:

We use the customer with id 39868635 as an example:

*match (a:Account) -[r1:belongTo]-> (c:Customer { id : 39868635 })*

*with r1,a,c*

*match (a:Account {id :* [*a.id*](http://a.id)*}) -[r2:transaction]- (b:Account)*

*return r1,r2,c,a,b*

We get the result as below:

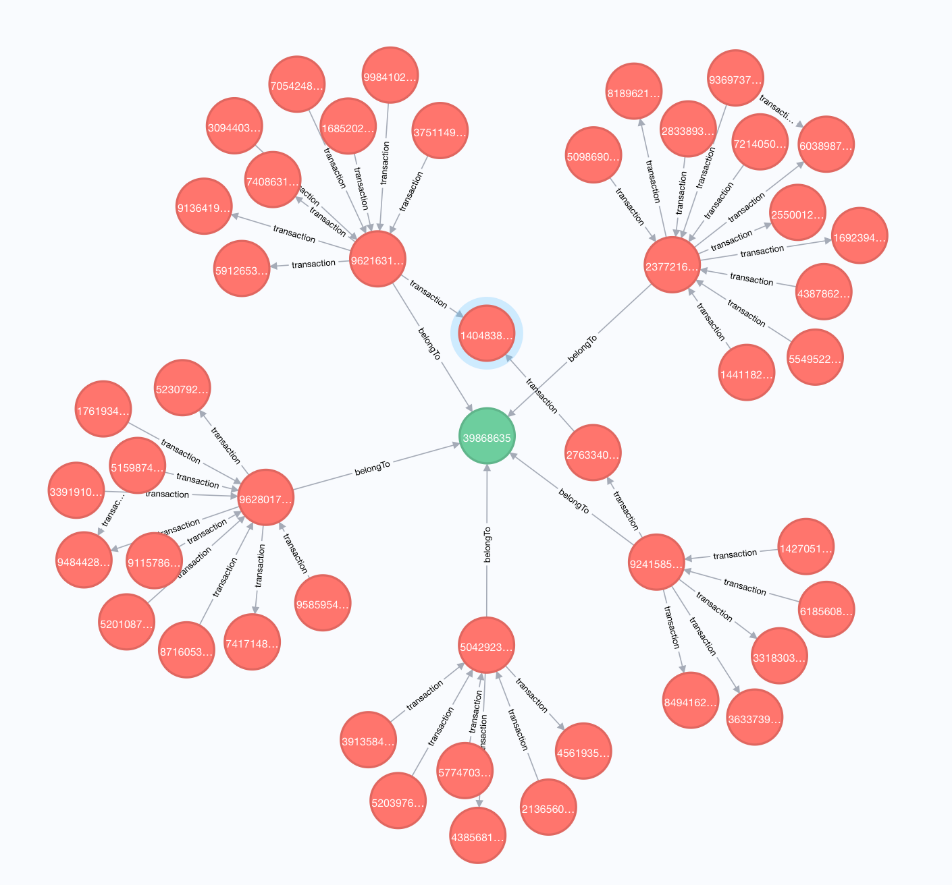


Figure 38: Customer for all customer transactions

For sum:

*match (a:Account) -[:belongTo]-> (c:Customer { id : 39868635 })*

*with a*

*match (a:Account {id :* [*a.id*](http://a.id)*}) -[r:transaction]- (b:Account)*

*return SUM(*[*r.TR\_AM*](r.TR_AM)*)*

For count:

*match (a:Account) -[:belongTo]-> (c:Customer { id : 39868635 })*

*with a*

*match (a:Account {id :* [*a.id*](http://a.id)*}) -[r:transaction]- (b:Account)*

*return COUNT(r)*

For balance:

*match (a:Account) -[:belongTo]-> (c:Customer { id : 39868635 })*

*with a*

*match (a:Account {id :* [*a.id*](http://a.id)*}) -[r1:transaction]-> (b:Account)*

*with a,r1*

*match (a:Account {id :* [*a.id*](http://a.id)*}) <-[r2:transaction]- (c:Account)*

*return (SUM(*[*r2.TR\_AM*](r2.TR_AM)*) - SUM(*[*r1.TR\_AM*](r1.TR_AM)*))*

* Check all transactions of a specific card, including balance, sum and count.

We use the customer with id 39868635 as an example:

*match (a:Account {id : 4567905843}) -[r:transaction]- (b:Account)*

*return r,a,b*

We can get the result as below:

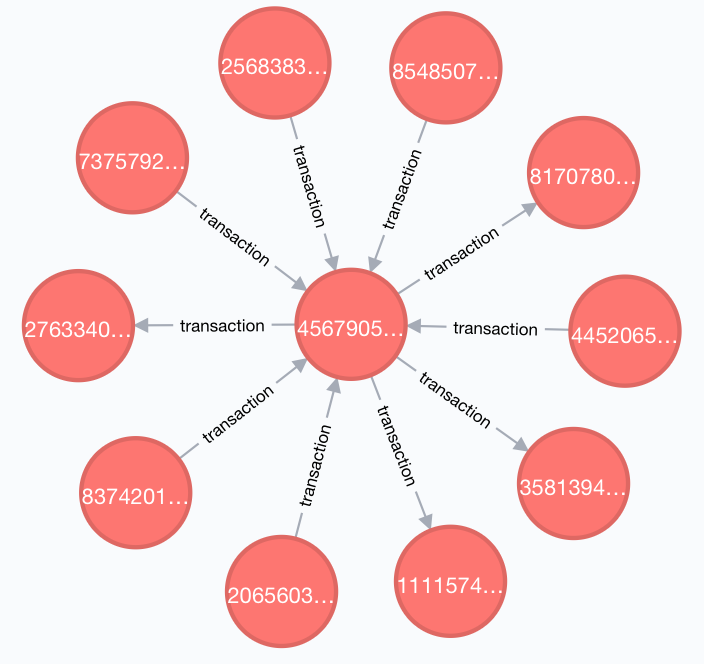


Figure 39: Customer for specific cards transactions

For sum:

*match (a:Account {id : 4567905843}) -[r:transaction]- (b:Account)*

*return SUM(*[*r.TR\_AM*](r.TR_AM)*)*

For count:

*match (a:Account {id : 4567905843}) -[r:transaction]- (b:Account)*

*return COUNT(r)*

For balance:

*match (a:Account {id : 4567905843 }) -[r1:transaction]-> (b:Account)*

*with r1*

*match (a:Account {id : 4567905843 }) <-[r2:transaction]- (c:Account)*

*return (SUM(*[*r2.TR\_AM*](r2.TR_AM)*) - SUM(*[*r1.TR\_AM*](r1.TR_AM)*))*

* Check all transactions operated in a specific institute, including balance, sum and count.

We use the customer with id 39868635 as an example:

*match (a1:Account)-[r2:transaction]-(a:Account)-[r1:belongTo]->(c:Customer { id : 39868635 })*

*with a1,a,c,r1,r2*

*match (h:Hall)<-[r3:workIn]-(t:Teller) where* [*t.id*](http://t.id) *= r2.OPR\_id and* [*h.id*](http://h.id) *= 688005*

*return a1,a,c,t,h,r1,r2,r3*

We can get the result as below:

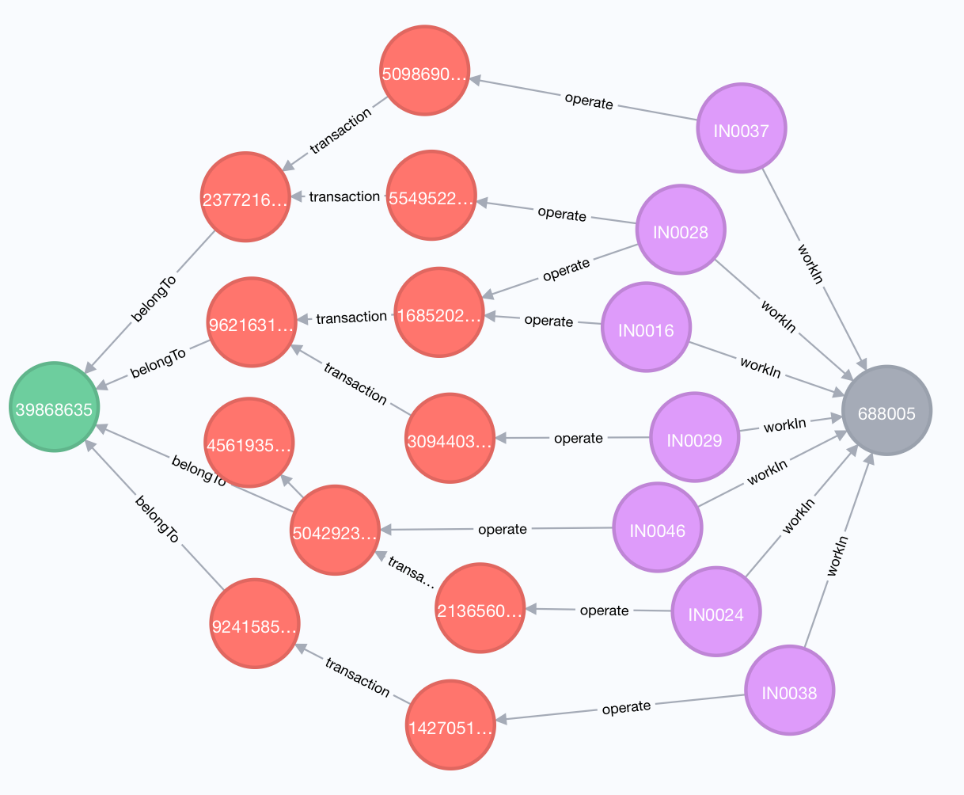


Figure 40: Customer for all specific institute transactions

For sum:

*match (a1:Account)-[r2:transaction]-(a:Account)-[r1:belongTo]->(c:Customer { id : 39868635 })*

*with a1,a,c,r1,r2*

*match (h:Hall)<-[r3:workIn]-(t:Teller) where* [*t.id*](http://t.id) *= r2.OPR\_id and* [*h.id*](http://h.id) *= 688005*

*return SUM(*[*r2.TR\_AM*](r2.TR_AM)*)*

For count:

*match (a1:Account)-[r2:transaction]-(a:Account)-[r1:belongTo]->(c:Customer { id : 39868635 })*

*with a1,a,c,r1,r2*

*match (h:Hall)<-[r3:workIn]-(t:Teller) where* [*t.id*](http://t.id) *= r2.OPR\_id and* [*h.id*](http://h.id) *= 688005*

*return COUNT(r2)*

For balance:

*match (a:Account)-[r:belongTo]->(c:Customer { id : 39868635 })*

*with a,r,c*

*match (a1:Account)-[r1:transaction]->(a:Account)-[r2:transaction]->(a2:Account)*

*with r1,r2,a,r,c*

*match (h:Hall)<-[r3:workIn]-(t:Teller)*

*where (*[*h.id*](http://h.id) *= 688005) and (*[*t.id*](http://t.id) *= r1.OPR\_id) OR (*[*t.id*](http://t.id) *= r2.OPR\_id)*

*return (SUM(*[*r1.TR\_AM*](r1.TR_AM)*) - SUM(*[*r2.TR\_AM*](r2.TR_AM)*))*

3.2 Chief

* Check all transactions of the corresponding institute, including balance, sum and count.

We use chief with id 001041 as the example:

*match (c:Cheif{id: '001041'} )-[m:manage]->(h:Hall)<-[w:workIn]-(t:Teller)-[o:operate] ->(a:Account)*

*with c,m,h,w,t,o,a*

*match (a1:Account)-[r1:transaction]->(a:Account)-[r2:transaction]->(a2:Account)*

*return a1,a2,a,r1,c,m,h,w,t,o Limit 30*

We can get the result as below:

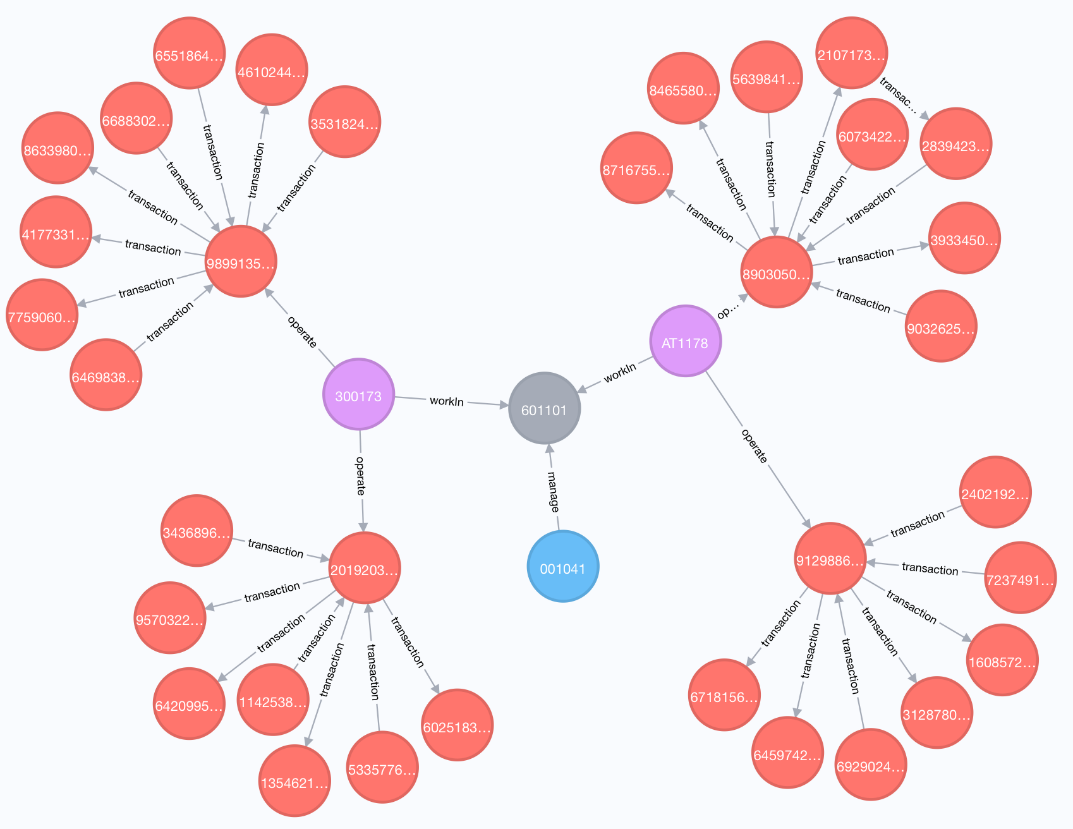


Figure 41: Chief for corresponding institute

For sum:

*match (c:Cheif{id: '001041'} )-[m:manage]->(h:Hall)<-[w:workIn]-(t:Teller)-[o:operate] ->(a:Account)*

*with c,m,h,w,t,o,a*

*match (a1:Account)-[r1:transaction]->(a:Account)-[r2:transaction]->(a2:Account)*

*return SUM(*[*r1.TR\_AM*](r1.TR_AM)*) Limit 30*

For count:

*match (c:Cheif{id: '001041'} )-[m:manage]->(h:Hall)<-[w:workIn]-(t:Teller)-[o:operate] ->(a:Account)*

*with c,m,h,w,t,o,a*

*match (a1:Account)-[r1:transaction]->(a:Account)-[r2:transaction]->(a2:Account)*

*return COUNT(*[*r1.TR\_AM*](r1.TR_AM)*) Limit 30*

For balance

*match (c:Cheif{id: '001041'} )-[m:manage]->(h:Hall)<-*

*[w:workIn]-(t:Teller)-[o:operate] ->(a:Account)*

*with c,m,h,w,t,o,a*

*match (a1:Account)-[r1:transaction]->(a:Account)-[r2:transaction]->(a2:Account)*

*return SUM(r1.transaction) - SUM(r2.transaction)*

* Check all transactions of all specific tellers, including balance, sum and count.

We use chief with id 001041 as the example:

*match (c:Cheif{id: '001041'} )-[m:manage]->(h:Hall)<-[w:workIn]-(t:Teller {id : '300173'})-[o:operate] ->(a:Account)*

*with c,m,h,w,t,o,a*

*match (a1:Account)-[r1:transaction]->(a:Account)-[r2:transaction]->(a2:Account)*

*return a1,a2,a,r1,c,m,h,w,t,o Limit 30*

We get the result as below:

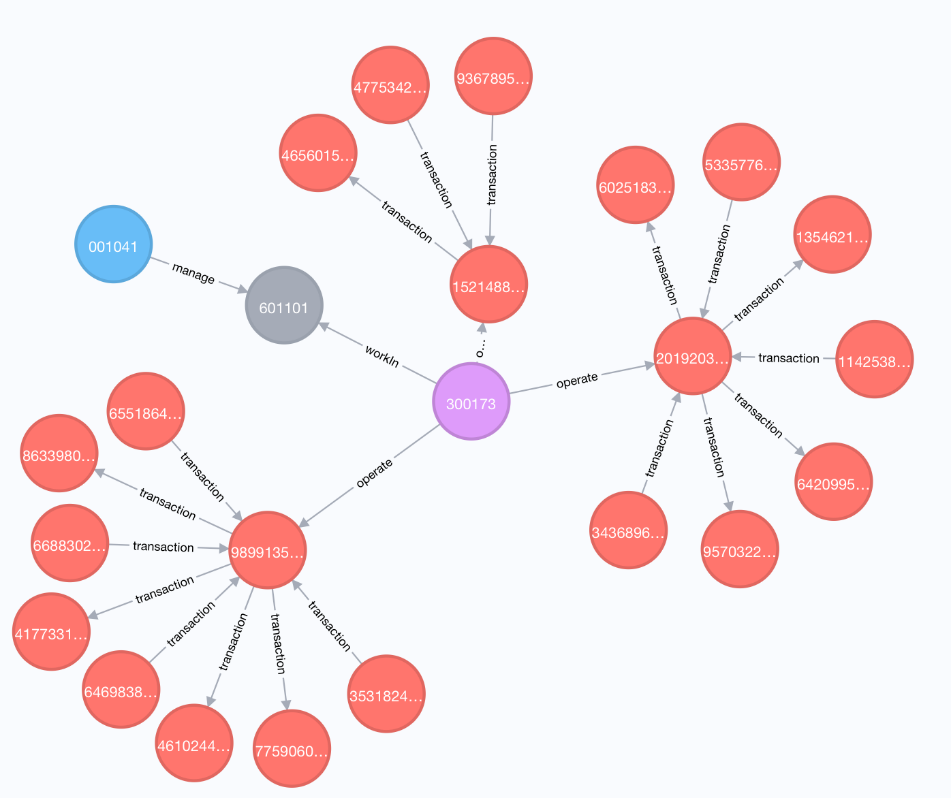


Figure 42: Chief for specific tellers

For sum:

*match (c:Cheif{id: '001041'} )-[m:manage]->(h:Hall)<-[w:workIn]-(t:Teller {id : '300173'})-[o:operate] ->(a:Account)*

*with c,m,h,w,t,o,a*

*match (a1:Account)-[r1:transaction]->(a:Account)-[r2:transaction]->(a2:Account)*

*return SUM(*[*r1.TR\_AM*](r1.TR_AM)*)*

For count:

*match (c:Cheif{id: '001041'} )-[m:manage]->(h:Hall)<-[w:workIn]-(t:Teller {id : '300173'})-[o:operate] ->(a:Account)*

*with c,m,h,w,t,o,a*

*match (a1:Account)-[r1:transaction]->(a:Account)-[r2:transaction]->(a2:Account)*

*return COUNT(r1)*

For balance:

*match (c:Cheif{id: '001041'} )-[m:manage]->(h:Hall)<-[w:workIn]-(t:Teller {id :'300173'})-[o:operate] ->(a:Account)*

*with c,m,h,w,t,o,a*

*match (a1:Account)-[r1:transaction]->(a:Account)-[r2:transaction]->*

*(a2:Account)*

*return SUM(r1.transaction) - SUM(r2.transaction)*

* Check all transactions of all accounts, including balance, sum and count.

We use chief with id 001041 as the example:

*match (c:Cheif{id: '001041'} )-[m:manage]->(h:Hall)<-[w:workIn]-(t:Teller)-[o:operate] ->(a:Account {id : 2019203201})*

*with c,m,h,w,t,o,a*

*match (a1:Account)-[r1:transaction]-(a:Account)*

*return a1,a,r1,c,m,h,w,t,o*

We get the result as below:

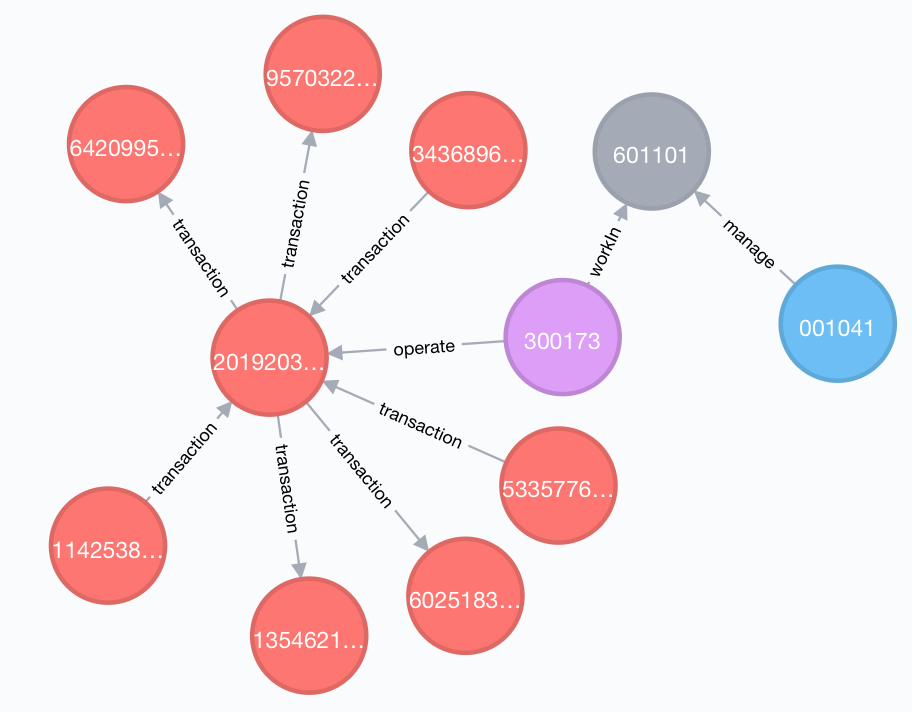


Figure 43: Chief for specific accounts

For sum:

*match (c:Cheif{id: '001041'} )-[m:manage]->(h:Hall)<-[w:workIn]-(t:Teller)-[o:operate] ->(a:Account {id : 2019203201})*

*with c,m,h,w,t,o,a*

*match (a1:Account)-[r1:transaction]-(a:Account)*

*return SUM(*[*r1.TR\_AM*](r1.TR_AM)*)*

For count:

*match (c:Cheif{id: '001041'} )-[m:manage]->(h:Hall)<-[w:workIn]-(t:Teller)-[o:operate] ->(a:Account {id : 2019203201})*

*with c,m,h,w,t,o,a*

*match (a1:Account)-[r1:transaction]-(a:Account)*

*return COUNT(r1)*

For balance:

*match (c:Cheif{id: '001041'} )-[m:manage]->(h:Hall)<-[w:workIn]-*

*(t:Teller)-[o:operate] ->(a:Account {id : 2019203201})*

*with c,m,h,w,t,o,a*

*match (a1:Account)-[r1:transaction]->(a:Account)-[r2:transaction]->*

*(a2:Account)*

*return SUM(r1.transaction) - SUM(r2.transaction)*

3.3 President

There is only one president in the system.

*match (p:President)-[r:rule] ->(h:Hall)<-[w:workIn]-(t:Teller)-[o:operate] ->(a:Account)*

*with p,r,h,w,t,o,a*

*match (a1:Account)-[r1:transaction]->(a:Account)-[r2:transaction]->(a2:Account)*

*return p,r,h,w,t,o,a,a1,r1,r2 Limit 30*

We can get the result as below:

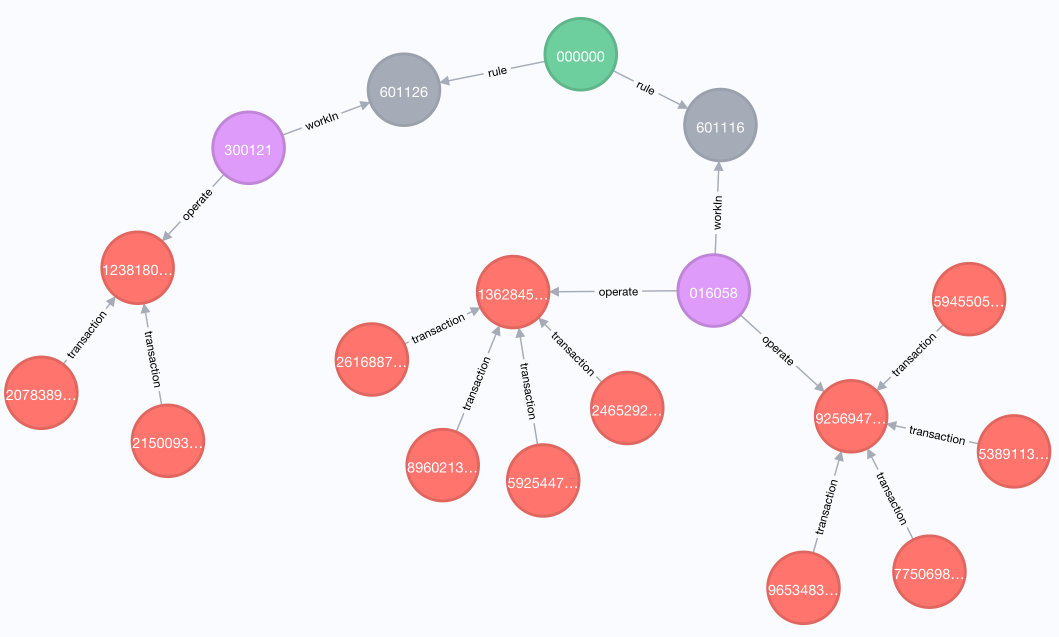


Figure 44: President for all the transactions

* Check the sum.

*match (p:President)-[r:rule] ->(h:Hall)<-[w:workIn]-(t:Teller)-*

*[o:operate] ->(a:Account)*

*with p,r,h,w,t,o,a*

*match (a1:Account)-[r1:transaction]-(a:Account)*

*return SUM(*[*r1.TR\_AM*](r1.TR_AM)*)*

* Check the count.

*match (p:President)-[r:rule] ->(h:Hall)<-[w:workIn]-(t:Teller)-[o:operate] ->(a:Account)*

*with p,r,h,w,t,o,a*

*match (a1:Account)-[r1:transaction]-(a:Account)*

*return COUNT(r1)*

* Check the balance.

*match (p:President)-[r:rule] ->(h:Hall)<-[w:workIn]-(t:Teller)-[o:operate] ->(a:Account)*

*with p,r,h,w,t,o,a*

*match (a1:Account)-[r1:transaction]->(a:Account)-[r2:transaction]->(a2:Account)*

*return SUM(r1.transaction) - SUM(r2.transaction)*

3.4 PageRank (only for Neo4j)

The role of Centrality algorithms in the abnormal teller score is important:

Take the account operated by the teller as an example, the account has different degrees of importance in the network it forms. The importance of the decision can be voted together with other accounts that have transactions with it. An account is voted to be the most unusual account, and it may have many characteristics, such as the starting point or transit point of abnormal funds, such as the initiator of gang fraud. Other accounts that have a relationship with the most anomalous accounts, as well as the most frequently-operated tellers, are the ones we focus on.

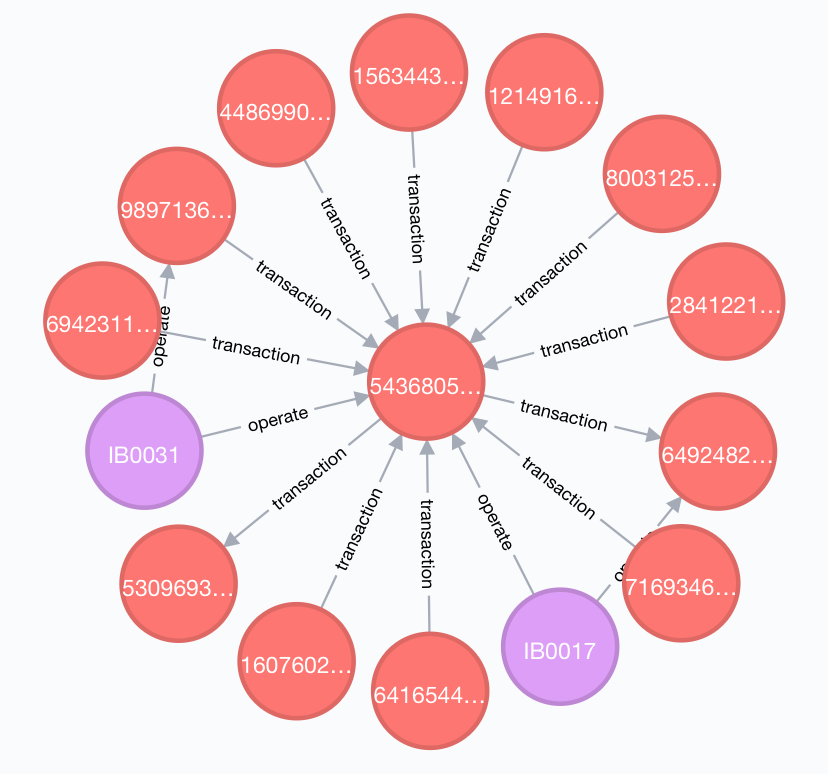


Figure 45: PageRank

*CALL algo.pageRank('Account', 'transaction',*

*{iterations:20, dampingFactor:0.85, write: true,writeProperty:"pagerank"})*

*YIELD nodes, iterations, loadMillis, computeMillis, writeMillis, dampingFactor, write, writeProperty*

*match (a:Account) return a order by a.pagerank DESC limit 1*

*match (t:teller)-[:operate]->(a:Account{id: 5436805195})-[r:transaction]-(b:Account)*

*return t,a,b,r*

3.5 Community detection algorithms(only for Neo4j)

The community detection role in the abnormal teller score:

Take the customer who operates the teller as an example, the community in the network formed by the customer does exist. Customers may only interact frequently with some customers, and different levels of interaction lead to the formation of the community. If the abnormal customer's abnormal behavior is traced back to the source, or the abnormal customer is found to find other abnormal customers, customers belonging to the same community should look for it first. This is the reason that "things are gathered together and people are divided into groups". Calling the community discovery algorithm on a large network is actually based on a certain standard, and on this basis, each community can be further explored. From a computational point of view, community division is equivalent to decomposing tasks, which plays a role in reducing computational complexity. When we found more abnormal customers, the rating of the tellers was more objective.

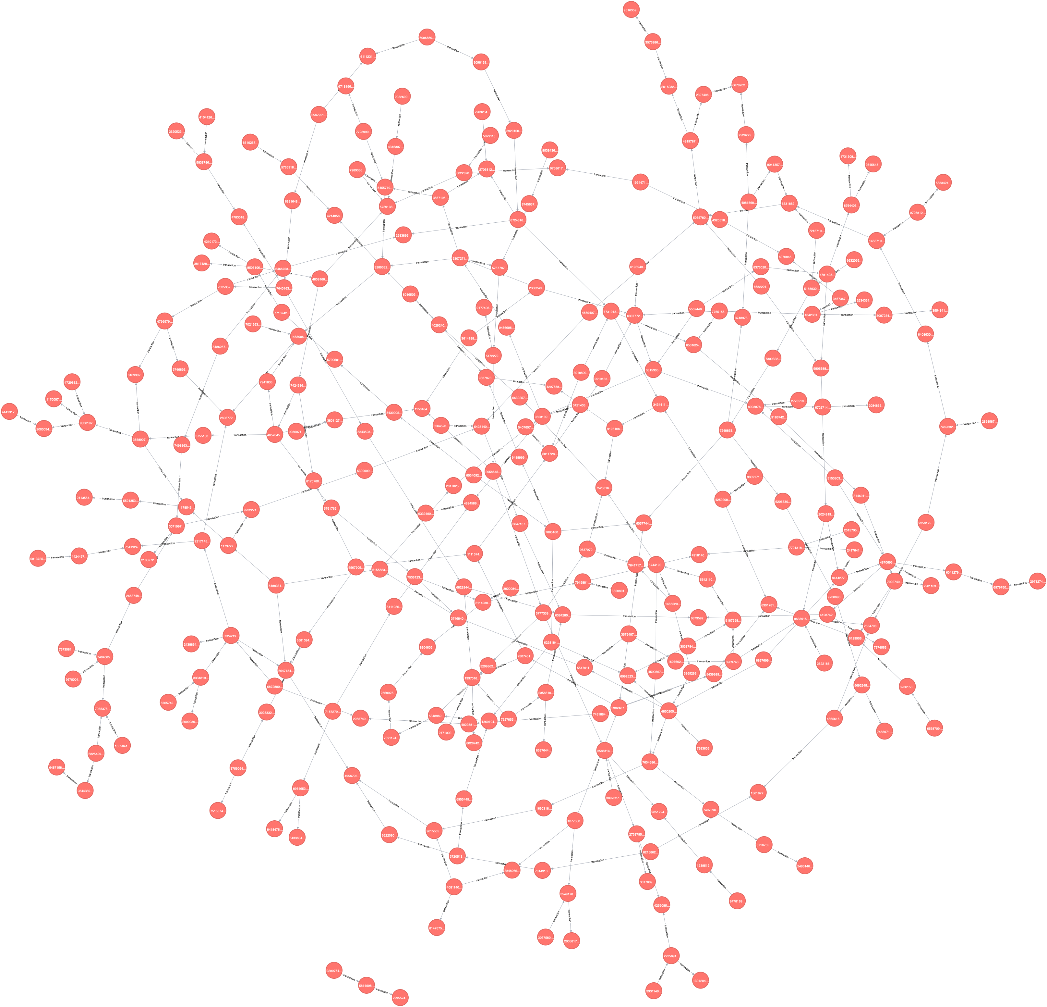


Figure 46: Community detection algorithms

*CALL algo.louvain('Account', 'transaction', {write:true, writeProperty:'community'})*

*YIELD nodes, communityCount, iterations, loadMillis, computeMillis, writeMillis;*

*MATCH (acc1:Account {community: 1})-[r:transaction]-(acc2:Account{community: 1}) return acc1,acc2,r*

**7. MongoDB**

1. Collections

We created 5 collections to contain all data we have.

A screenshot of a cell phone

Description automatically generated

Figure 47: The confidential collection

A screenshot of a cell phone

Description automatically generated

Figure 48: The customer accounts collection

A screenshot of a cell phone

Description automatically generated

Figure 49: The customer collection

A screenshot of a cell phone

Description automatically generated

Figure 50: The teller collection

A screenshot of a cell phone

Description automatically generated

Figure 51: The transaction collection

1. Functions and implementation
   1. For customer
      1. We checked all transactions for corresponding customer and implemented following aggregation functions (we took customer with id 88528785 as an example:
         1. Balance
         2. Sum
         3. Count

Code:

def balance(cards):

transfer\_in = 0

transfer\_out = 0

count = 0

for card in cards:

result\_in = trans\_set.find({"ACCT\_ID": card})

for item in result\_in:

transfer\_in += item['TR\_AM']

count += 1

result\_out = trans\_set.find({"OPP\_ACCT\_ID": card})

for item in result\_out:

transfer\_out += item['TR\_AM']

curr\_balance = transfer\_in - transfer\_out

return curr\_balance, transfer\_in, transfer\_out, count

cust = "88528785"

print('customer id: ', cust)

cards = get\_user\_cards\_mongo(cust)

balance\_curr\_cust, transfer\_in, transfer\_out, count = balance(cards)

print("balance = ", balance\_curr\_cust)

print("transfer in sum = ", transfer\_in)

print('transfer out sum = ', transfer\_out)

print('transfer count = ', count)

result:

customer id: 88528785

balance = -2892292

transfer in sum = 7418236

transfer out sum = 10310528

transfer count = 12

* + 1. We checked all transactions for a specific card, and did aggregation functions based on this card (we still took customer with id 88528785 as an example:
       1. Balance
       2. Sum
       3. Count

Code:

def balance(cards):

all\_tr\_in = {}

all\_tr\_out = {}

all\_balance = {}

all\_count = {}

for card in cards:

transfer\_in = 0

transfer\_out = 0

count = 0

result\_in = trans\_set.find({"ACCT\_ID": card})

for item in result\_in:

transfer\_in += item['TR\_AM']

count += 1

result\_out = trans\_set.find({"OPP\_ACCT\_ID": card})

for item in result\_out:

transfer\_out += item['TR\_AM']

curr\_balance = transfer\_in - transfer\_out

all\_tr\_in[card] = transfer\_in

all\_tr\_out[card] = transfer\_out

all\_balance[card] = curr\_balance

all\_count[card] = count

return all\_tr\_in, all\_tr\_out, all\_balance, all\_count

cust = "88528785"

print('customer id: ', cust)

cards = get\_user\_cards\_mongo(cust)

balance\_curr\_cust, transfer\_in, transfer\_out, count = balance(cards)

for card in cards:

print('curr card = ', card)

print("balance = ", balance\_curr\_cust[card])

print("transfer in sum = ", transfer\_in[card])

print('transfer out sum = ', transfer\_out[card])

print('transfer count = ', count[card])

results:

customer id: 88528785

curr card = 2016232626

balance = 2444621

transfer in sum = 2282611

transfer out sum = 162010

transfer count = 3

curr card = 7787346346

balance = 1621755

transfer in sum = 3048658

transfer out sum = -1426903

transfer count = 3

curr card = 2003727406

balance = 1467140

transfer in sum = 1790697

transfer out sum = -323557

transfer count = 2

curr card = 2553396999

balance = 1884720

transfer in sum = 3188562

transfer out sum = -1303842

transfer count = 4

* + 1. We checked all transactions in one specific bank, and did aggregation functions based on this bank as following ( still take customer with id 88528785 and bank with id 688005) :
       1. Balance
       2. Sum
       3. Count

Code:

cust = "88528785"

ins\_id = '688005'

print('customer id: ', cust)

tellers = teller\_set.find({'YNGYJIGO': ins\_id})

transfer\_sum = 0

transfer\_count = 0

for teller in tellers:

trans = trans\_set.find({'CUST\_ID': cust, 'OPR\_ID': teller['GUIYDAIH']})

for item in trans:

transfer\_sum += item['TR\_AM']

transfer\_count += 1

print('institute id = ',ins\_id)

print('transfer\_sum = ', transfer\_sum)

print('transfer\_count = ', transfer\_count)

result:

customer id: 88528785

institute id = 688005

transfer\_sum = 1755955

transfer\_count = 4

balance = 96313

* 1. For teller
     1. The tellers can check their daily transactions (we took teller id IN0029 as an example)

Code:

def teller\_get\_transaction(user\_id):

result = []

total = 0

items = trans\_set.find({"OPR\_ID": user\_id})

for item in items:

result.append(item)

total += item['TR\_AM']

pipeline0 = [

{"$match": {"OPR\_ID": user\_id}},

{"$group": {"\_id": "$TR\_TM",'total': {'$sum':'$TR\_AM'}}},

{"$sort": {"\_id": -1}}

]

daily\_amount = trans\_set.aggregate(pipeline0)

for item in daily\_amount:

print(item)

result:

{'\_id': '31-AUG-18', 'total': 1642984}

{'\_id': '30-AUG-18', 'total': 1772515}

{'\_id': '29-AUG-18', 'total': 833243}

{'\_id': '28-AUG-18', 'total': 2124588}

{'\_id': '27-AUG-18', 'total': 2622699}

{'\_id': '24-AUG-18', 'total': 2282912}

{'\_id': '23-AUG-18', 'total': 915590}

{'\_id': '22-AUG-18', 'total': 61814}

{'\_id': '20-AUG-18', 'total': 7896669}

{'\_id': '17-AUG-18', 'total': 575148}

{'\_id': '15-AUG-18', 'total': 4926523}

{'\_id': '14-SEP-18', 'total': 1116863}

{'\_id': '13-SEP-18', 'total': 298417}

{'\_id': '12-SEP-18', 'total': 676182}

{'\_id': '11-SEP-18', 'total': 2957015}

{'\_id': '10-SEP-18', 'total': 555327}

{'\_id': '07-SEP-18', 'total': 190059}

{'\_id': '06-SEP-18', 'total': 273138}

{'\_id': '05-SEP-18', 'total': 3213541}

{'\_id': '04-SEP-18', 'total': 1342785}

{'\_id': '03-SEP-18', 'total': 1222024}

­­­­­­­­

* 1. For chief
     1. The chief of each bank can check all transactions for all transactions of corresponding bank (we took bank id 601101 as an example):
        1. Sum
        2. Count

Code:

JIGODAIHAO = '601101'

result = teller\_set.find({'YNGYJIGO': JIGODAIHAO})

tellers = []

teller\_trans = {}

total = 0

count = 0

for pair in result:

teller = pair['GUIYDAIH']

print('teller = ', teller)

tellers.append(teller)

opr\_trans = trans\_set.find({'OPR\_ID': teller})

temp = []

for tran in opr\_trans:

temp.append(tran)

total += tran['TR\_AM']

count += 1

teller\_trans[teller] = temp

print(total)

print(count)

result:

8430974

20

* + 1. The chief of each bank can check all transactions from any specific teller in the same bank (we took teller id 008048 as an example), and implemented following functions:
       1. Sum
       2. Count

Code:

teller = '008048'

t\_trans = teller\_trans[teller]

t\_sum = 0;

t\_count = 0

for item in t\_trans:

t\_sum += item['TR\_AM']

t\_count += 1

print(item)

print(t\_sum)

print(t\_count)

result:

60942

1

* + 1. The chief can check all transactions from a specific account which comes from the same bank (we took account id 2798276299 as an example), we implemented as following:
       1. Sum
       2. Count

Code:

acct\_id = '2798276299'

all\_trans\_curr\_cust = []

cust\_sum = 0;

cust\_count = 0

for teller in tellers:

trans = trans\_set.find({'ACCT\_ID': acct\_id, 'OPR\_ID': teller})

for item in trans:

all\_trans\_curr\_cust.append(item)

cust\_sum += item['TR\_AM']

cust\_count += 1

print(item)

opp\_trans = trans\_set.find({'OPP\_ACCT\_ID': acct\_id, 'OPR\_ID': teller})

for item in opp\_trans:

all\_trans\_curr\_cust.append(item)

cust\_sum -= item['TR\_AM']

cust\_count += 1

print(cust\_sum)

print(cust\_count)

result:

782168

1

* 1. For president
     1. The president can check all transaction amount from all banks

Code:

pipeline = [

{"$group": {"\_id": "$ACCT\_ID",'total': {'$sum':'$TR\_AM'}}},

{"$sort": {"\_id": -1}}

]

result = trans\_set.aggregate(pipeline)

total = 0

for doc in result:

total += doc['total']

print(total)

result:

11566598283

* 1. Exclusive function of MongoDB- geo-information

We can demonstrate all bank’s geo-information by using their latitude and longitude. Provided a coordinate and a radius, we can search all the banks in this range.

Code:

db.institute\_geo.find(

{

location:{

$near:{

$geometry: {

type: "Point",

coordinates: [ 145.234, 73.969 ] },

$minDistance: 0,

$maxDistance: 1000

}

}

}

)

result:

{ "\_id" : ObjectId("5cb6d0054e8f1c6f2a5dcf78"), "ins\_id" : "601152", "location" : { "type" : "Point", "coordinates" : [ 145.23461049583165, 73.96908712824836 ] } }

**8. System Implementation**

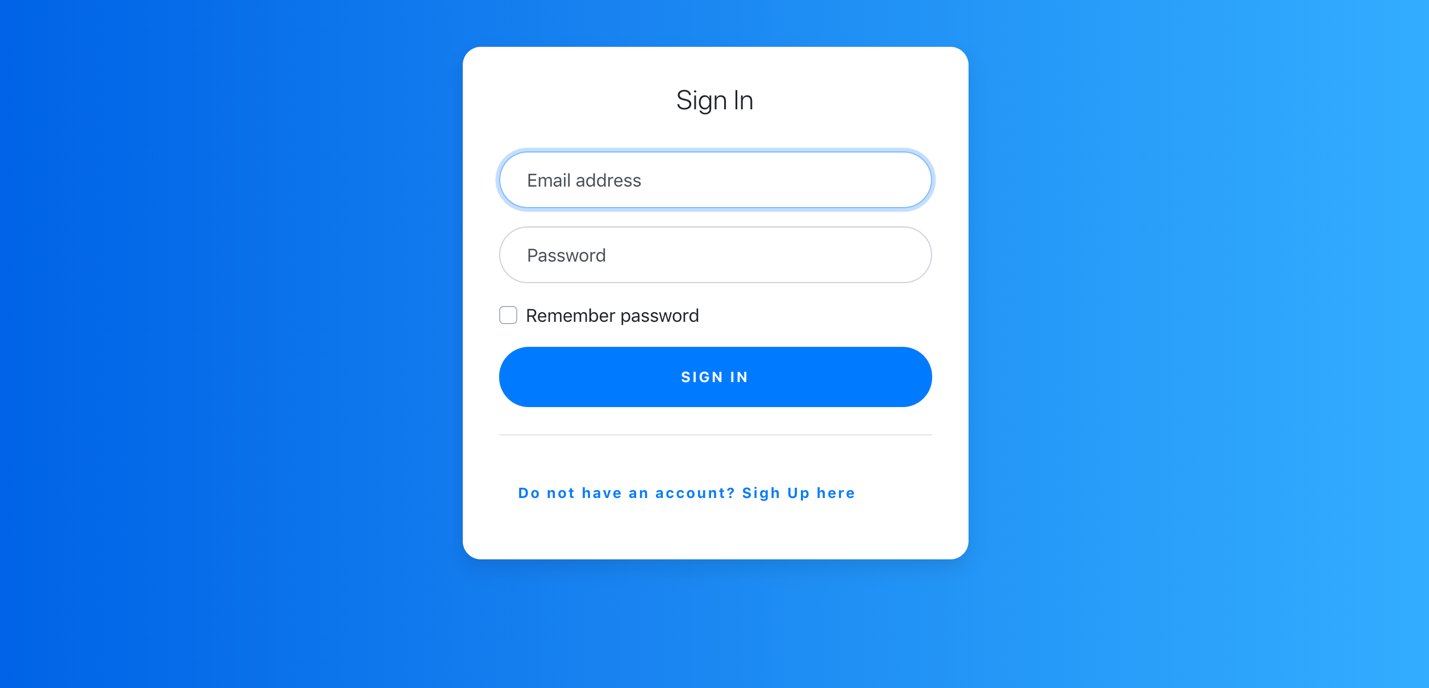


Figure 52: Login Page

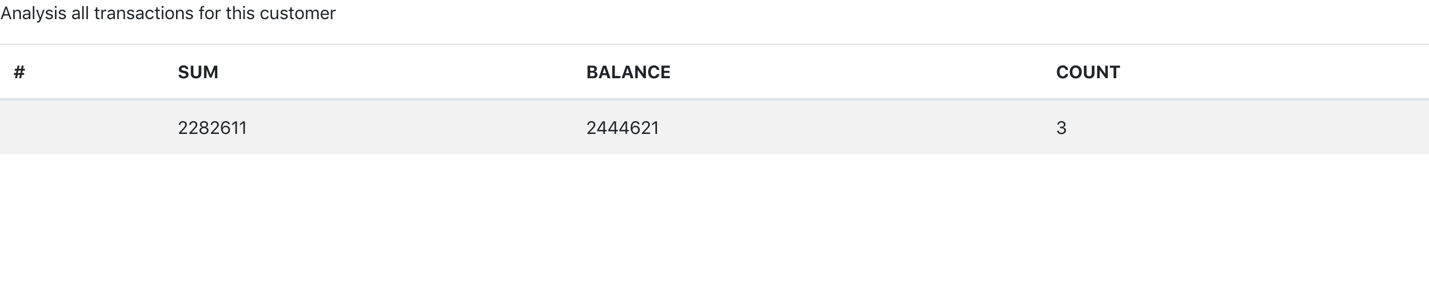


Figure 53: All transactions for the customer

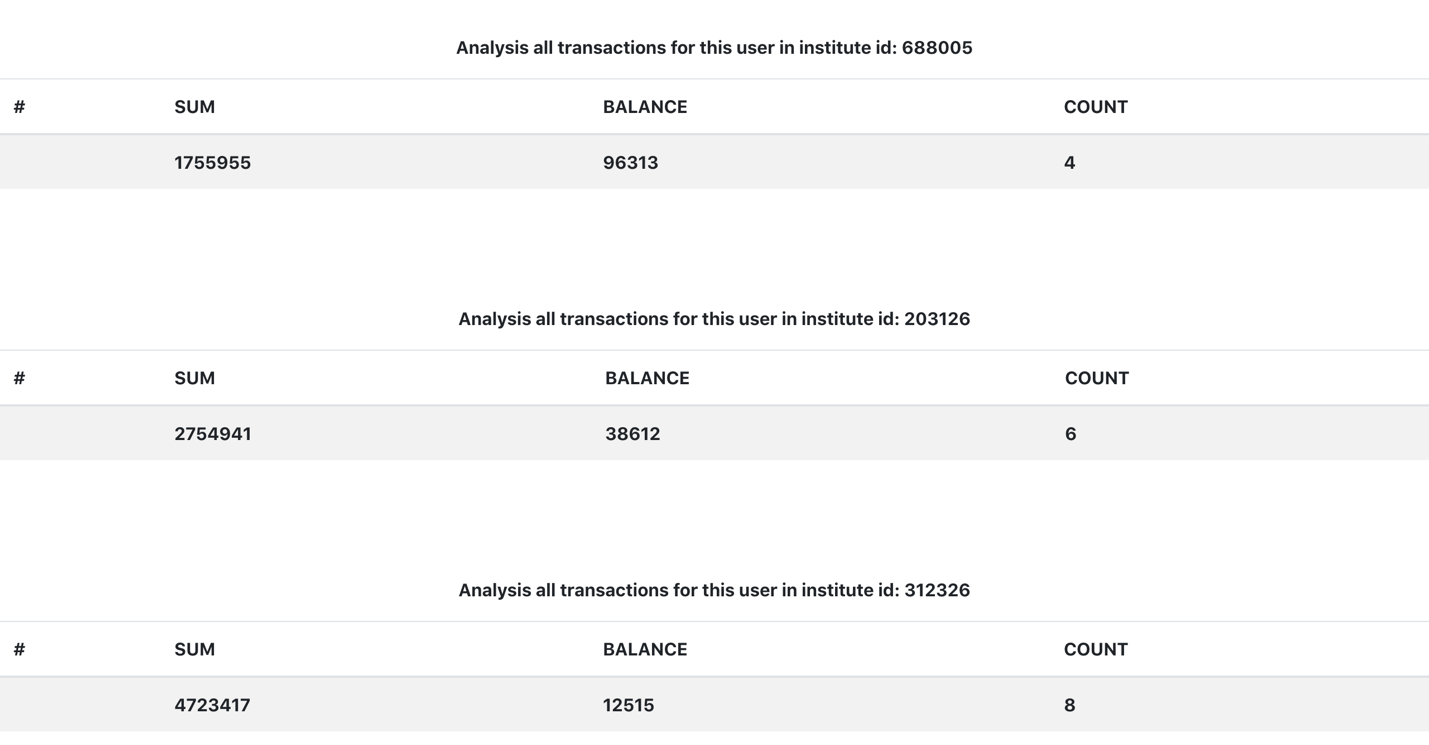


Figure 54: Transactions for the user with specific id

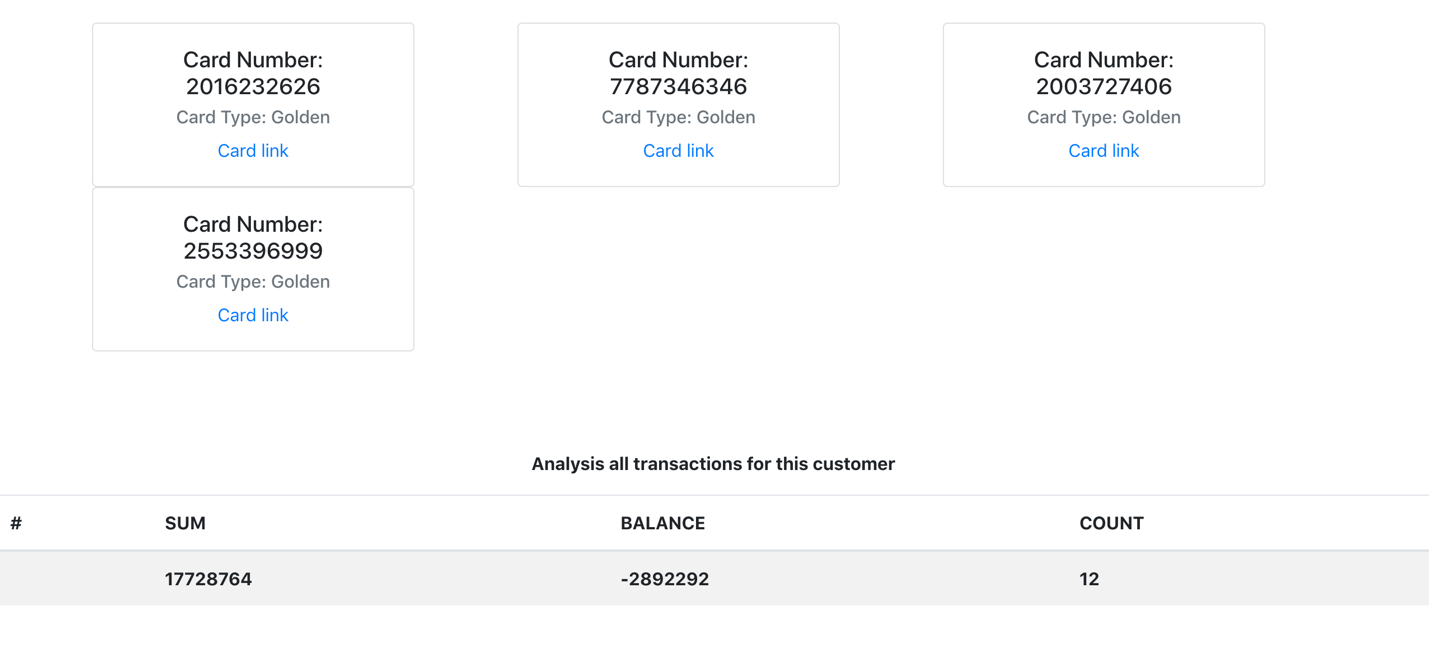


Figure 55: Accounts for the customer

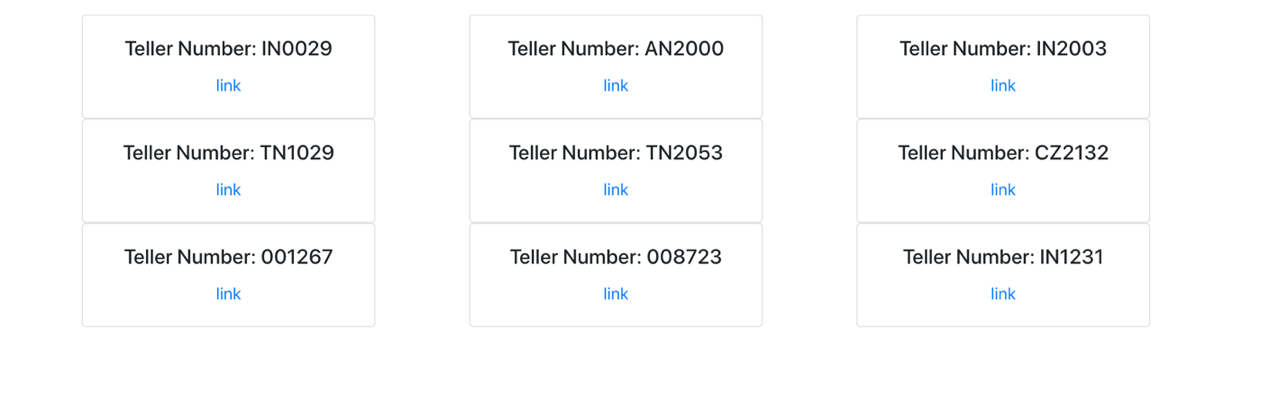


Figure 56: Tellers in one institute

**9. Comparison**

9.1 MySQL

Pros

1. MySQL is an open source database that provides interfaces to support multiple language connection operations.
2. MySQL has small size, fast speed and low total cost of ownership.
3. MySQL has a very flexible and secure privilege and password system. When the client connects to the MySQL server, all password transfers between them are encrypted, and MySQL supports host authentication.
4. MySQL has a very fast and stable thread-based memory allocation system that can be used continuously without worrying about its stability.

Cons

1. MySQL does not fully support unfamiliar keywords.
2. MySQL uses the default IP port, but sometimes these ips will also be broken into by some hackers.
3. For a hierarchical data structure, if you want to use such a flat, tabular structure to store data, it is very difficult to query or acquire data.

9.2 MongoDB

Pros:

1. For specific information which related to geoinformation, MongoDB supports query operations on it. Moreover, the MongoDB provides 2D and 2D sphere index to support the geospatial query.
2. MongoDB supports query operations that perform a text search of string content.
3. MongoDB supports MapReduce to handle large datasets, which can improve the performance of aggregation functions.

Cons:

1. For highly transactional situation, such as our bank management system, which is actually not suitable to store data in MongoDB. Traditional relational databases are still more suitable for applications that require a large number of atomic complex transactions.
2. High memory usage. MongoDB stores key names for each value pairs, and because the database doesn’t have functionally of joins, there is data redundancy. This results in increasing unnecessary usage of memory.
3. Limited data size. MongoDB only allows no more than 16MB document size.

9.3 Neo4j

Pros:

1. Object-Oriented. Explicit semantics for each query user input.
2. Suitable for mining graph relationships.
3. Excellent for real-time big data mining with visualization result.

Cons:

1. Bad support for distribution.
2. Fragment storage is not supported.