StrategicBanking

Harbir Aujla

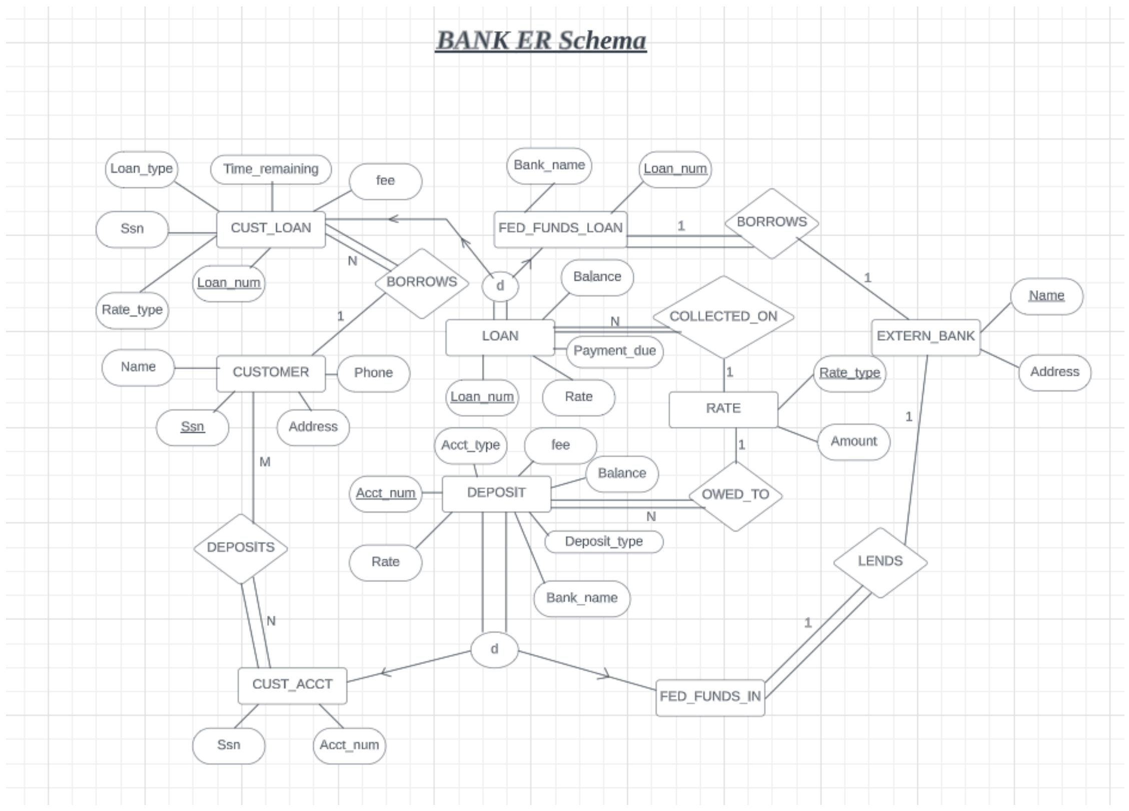
aujla@pnw.edu

Isaiah Rosinski

irosinsk@pnw.edu

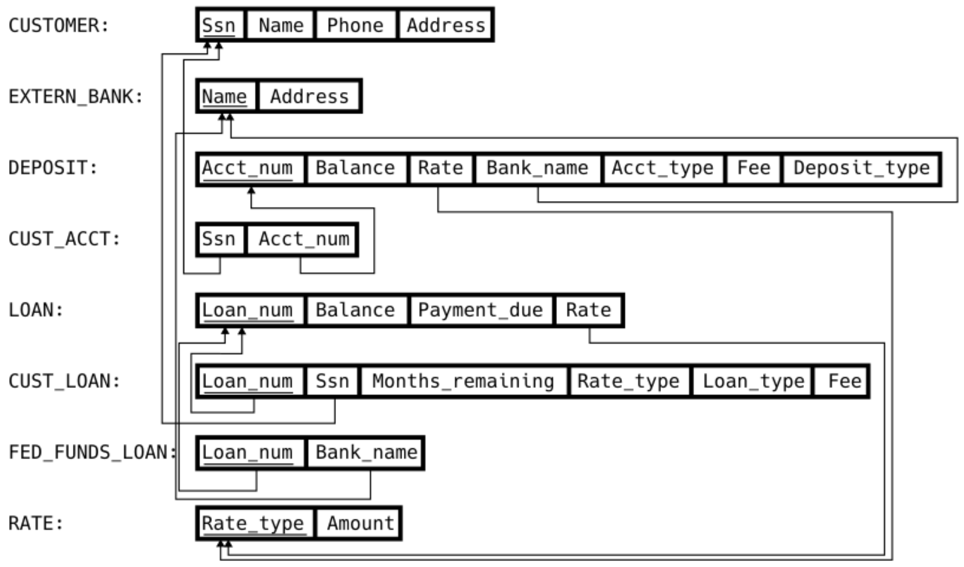
We designed a bank management application, called StrategicBanking. Our database holds data related to the operations of a bank, both customer accounts and internal systems. It is able to determine the bank's overall health, and can adjust the parameters of accounts and loans as needed. We were inspired by the bank management simulator at probanker.com, an academic tool that demonstrates the criterions and constraints required to keep a bank functioning in a healthy, profitable state. This simulator allows the user to bring a bank to profitability by manipulating various decision sets, including quantities of loans issued and deposits accepted, amounts of assets from the bank's portfolio to buy and sell, and interest rates on the various accounts.

Users' requirements are broken down into two groups: customer users and management users. The customer user has access to its account information, balances, and any other information regarding the customer's securities associated with the bank. The management user has the access to look at the bank's securities (its loans and deposits) and the ability to change its interest rate requirements on the securities offered by the bank.



Our E/R diagram can be seen above. We include five primary types of entities: customers, external banks, interest rates, deposits, and loans. Deposits and loans are each broken into two disjoint groups: a deposit is either a customer account or a Federal funds deposit, and a loan is either a customer loan or a Federal funds loan.

All of our constraints follow logically from our users' needs. Every customer loan must be borrowed by one customer, and every customer account must be deposited by one customer, but one customer may have zero or more loans and accounts. Every Federal funds deposit or loan must be associated with an external bank, and each bank may only deposit or loan one set of Federal funds. Every deposit and loan must have an interest rate. For uniqueness constraints, each customer's SSN, each account or loan's ID number, each rate's type, and each bank's name must be unique.


CUSTOMER: Ssn | Name | Phone | Address
EXTERN_BANK: Name | Address
DEPOSIT: Acct_num | Balance | Rate | Bank_name | Acct_type | Fee | Deposit_type
CUST_ACCT: Ssn | Acct_num
LOAN: Loan_num | Balance | Payment_due | Rate
CUST_LOAN: Loan_num | Ssn | Months_remaining | Rate_type | Loan_type | Fee
FED_FUNDS_LOAN: Loan_num | Bank_name
RATE: Rate_type | Amount

We used MariaDB as our database management system. This DBMS is an open-source system designed to be fully backwards compatible with MySQL. Our relational schema table can be seen above. In order to implement the disjoint subclasses of Deposit, we merged the two subclasses into one table and added the Deposit_type field, to indicate whether each deposit is a customer or Federal funds deposit. To implement the disjoint subclasses of Loan, we split Loan into three tables: one for the common attributes, one for customer loans, and one for Federal funds loans. Each relation in the table is in Boyce-Codd normal form: every non-key attribute in each relation is only dependent on that relation's primary key. Note that the table CUST_ACCT has a composite primary key: the combination of Ssn and Acct_num. (As there are no other entities in that table, it is trivially in BCNF.) At initialization, our database contains 12 customers, 11 banks, 15 interest rates, 14 deposit accounts, and 13 loan accounts; our initial data is too lengthy to include in this report, but can be found in our Github repository.

The most frequent queries that our database will handle are customers querying to retrieve and transact on their accounts and loans, and managers querying to retrieve and create new accounts and loans. Our database can also handle more complicated queries; a sampling of these are shown below.

This query will return the total net worth (deposits minus loans) of a given customer SSN, here 999887777.

```
SELECT SUM(DEPOSIT.Balance)-SUM(LOAN.Balance) as Net_worth
FROM DEPOSIT,LOAN
WHERE DEPOSIT.Balance IN (
       SELECT Balance FROM (DEPOSIT NATURAL JOIN CUST_ACCT) WHERE Ssn = "999887777")
AND LOAN.Balance IN (
       SELECT Balance FROM (LOAN NATURAL JOIN CUST_LOAN) WHERE Ssn = "999887777");
+------------+
| Net_worth  |
+------------+
| -362844.87 |
+------------+
```

This query will return the total equity of the bank, its assets (loans, fees) minus liabilities (deposits). It's a simple query, but it's very important to the bank.

```
SELECT SUM(LOAN.Balance) + SUM(CUST_LOAN.Fee) + SUM(DEPOSIT.Fee) - SUM(DEPOSIT.Balance) AS Equity
FROM LOAN, CUST_LOAN, DEPOSIT;
+--------------+
| Equity       |
+--------------+
| 190384589.68 |
+--------------+
```

This query will determine how much more money will be gained or lost on loans by changing a given interest rate. Here, we increase the 30YearVariable rate by .002 (that is, 0.2%).

```
WITH TEMP(Balance, Percent) AS (
       SELECT Balance, Amount AS Percent
       FROM (LOAN JOIN RATE ON Rate = Rate_type)
       WHERE Rate_type = "30YearVariable")
SELECT SUM(Balance * (Percent+.002)) - SUM(Balance * Percent) AS Profit
FROM TEMP;
+--------------+
| Profit       |
+--------------+
| 1369.7920600 |
+--------------+
```

To allow customers and bank managers to interact with the database, we built a web-based frontend. Customers can log into the frontend with their Social Security number, and from there may view their active bank accounts and loans, and may make deposits, withdrawals, or loan payments. Bank managers can enter the frontend and view all customer data, and can create new accounts or loans for their customers, borrow and loan Federal funds, and view and modify interest rates. Managers can also see the results of changing an interest rate, and view some aggregate statistics about their bank. The frontend was built in JavaScript, using NodeJS. The web server was built with ExpressJS, using the Pug templating engine. To generate the live tables, which change based on the current database state and the currently logged in user, the Express server calls a JavaScript function that queries the MariaDB database for the needed information, and returns an HTML-formatted table. This table is passed into the Pug template, which is then rendered onto the user's browser.

Our team worked together on most of the project: we collaborated to design the database's architecture, the E/R diagram and relational schema, and the sample data and queries. Later on, we divided the work more: Isaiah worked on the frontend and UI development for our database design; Harbir assembled the presentation and wrote most of the final report.

Our project has taught us a lot about what is needed to simulate a real-world business like a bank, and how many interlocking parts something like a bank has. We have had to adapt to the various ways that a certain set of data can be changed, whether by a customer or a manager. We have several goals and improvements we look forward to, both in our internal database and on the frontend. In our database, we would like to track loan payments and interest accrued over time, rather than in a static field. We would also like to implement more of a bank's internal functions, like issuing bonds and managing customer investments. In our frontend interface, we would like to add more constraints to account types—for example, preventing withdrawal from a CD or assessing late payment fees on loans. We would also like to add more security and input safety, as well as adding more of the expected functions and a general polish to the website's appearance.

Source code and project documentation: https://github.com/irosinsk-pnw/StrategicBanking