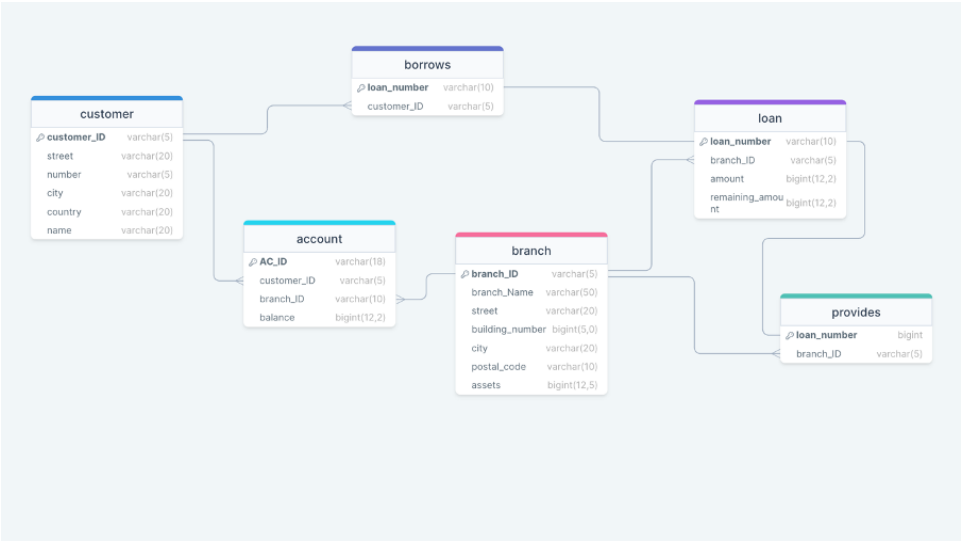
Assignment 1: Data Extraction & Integration

Dimitrios Diamantidis - 7766408
Johan Hensman - 1118994
Nicky Schilperoort - 5755354

Task 1: Database (re)design

a. For all three relations we added attributes that could be useful for data extraction. The attributes “postal code” and “assets” were added to the already existing attributes in the relation “branch”. These attributes give more information about the location and the funds of the branches. To the relation “customer”, the attribute “name” was added. This attribute provides extra information on the identity of the customer. The attribute “remaining amount” was added to the “loan” relation. This attribute provides information on how much money still needs to be paid by the customer.

b. Three main entities were already given: branch, customer and loan. We looked for relations that could be a link between the already existing main entities. We came up with the entities “borrows”, “provides” and “account”. The relation “borrows” connects the entities “loan” and “customer”, which means all loans can be attributed to a certain customer. “provides” connects “branch” and “loan”, so that we are able to see how many loans are provided per branch. “account” connects “branch” and “customer”. We thought it would be useful for customers to be able to have more than one account, for example when someone wants to open an account with their partner.

c. 

d.

Customer and Account (1:N)	Branch and Provides (1:N)
Customer and Borrows (1:N)	Provides and Loan (1:1)
Account and Branch (1:N)	Borrows and Loan (1:1)
Branch and Loan (1:N)	

Cardinalities

One customer can have multiplies accounts and they can also borrow money several times, that’s why the relationships between Customer-Account and Customer-Borrows tables are represented in One to Many. Furthermore, a single branch can have several accounts and loans corresponding to that branch, resulting in One to Many relationships for the tables

Branch-Account and Branch-Loan. Another One to Many relationship exists between Branch and Provides, as several loans can be provided by the same branch. Finally, the relationship between Loan-Provides and Loan-Borrows are One to One, while the Provides and Borrows tables represent the Loan table.

e. In the following table, the primary keys are listed for each table of the database schema.

Table	Primary key
Branch	Branch_ID
Loan	Loan_number
Customer	Customer_ID
Account	AC_ID
Provides	Loan_number
Borrows	Loan_number

The relationships between the tables are listed below with the foreign keys and their associated primary keys. To clarify the relation, the corresponding table to the keys is also listed.

Relation	Foreign key	Primary key
Loan - Branch	Branch_ID (Loan)	Branch_ID (Branch)
Account - Customer	Customer_ID (Account)	Customer_ID (Customer)
Account - Branch	Branch_ID (Account)	Branch_ID (Branch)
Borrows - Loan	Loan_number (Borrows)	Loan_number (Loan)
Borrows - Customer	Customer_ID (Borrows)	Customer_ID (Customer)
Provides - Branch	Branch_ID (Provides)	Branch_ID (Branch)
Provides - Loan	Loan_number (Provides)	Loan_number (Loan)

f. ✓

g. To see if the relation “Account” is in BCNF, the functional dependencies (FD) of the table need to uphold one of the two rules:

1: For every X -> Y in the table, the FD must be trivial

Or

2: For every X -> Y in the table, X is a superkey of R

In the table “Account”, the only non-trivial FDs are: AC_ID -> customer_ID, AC_ID -> branch_ID and AC_ID -> balance. As AC_ID is a superkey of the table, the rules of the BCNF are satisfied, so the relation is in BCNF.

Task 2: Querying the database

a. **Natural language**

“Find information about the branch corresponding to each bank account.”

Relational algebra

$\Pi_{AC\_ID,\ customer\_ID,\ balance,\ branch\_Name,\ street,\ building\_number}(\sigma_{ac.branch\_ID\ =\ br.branch\_ID}(\rho_{ac}(account) \bowtie \rho_{br}(branch)))$

SQL

SELECT *AC_ID*, *customer_ID*, *balance*, *branch_Name*, *street*, *building_number*

FROM *account ac*

LEFT JOIN *branch br* **ON** *ac.branch_ID* = *br.branch_ID*;

b. **Natural language**

"Find the average loan of customers for each branch."

Relational algebra

$\Pi_{loan.branch\_ID,\ branch\_Name,\ avg\_amount}(_{loan.branch\_ID}\gamma_{avg(amount)}(loan)$

$(loan \bowtie_{loan.branch\_ID\ =\ branch.branch\_ID} branch)))$

SQL

SELECT *loan.branch_ID*, *branch_Name*, **AVG** (*amount*)

FROM *loan*

JOIN *branch* **ON** *loan.branch_ID* = *branch.branch_ID*

GROUP BY loan.*branch_ID*;

c. **Natural language**

"Find the customer(s) with the highest balance in their account."

Relational algebra

$\Pi_{customer\_id,\ balance}(\sigma_{balance=\gamma\ max(balance)(account)}(account))$

SQL

SELECT *customer_ID*, *balance*

FROM *account*

WHERE *balance* = (**Select max** (*balance*)

From *account*);

Task 3: Data extraction and entity resolution using Python

a. The file with the code is divided into different chunks. The first chunk imports all the necessary libraries that are needed for this assignment. Then the functions 'create_connection' and 'run_query' are created. The 'create_function' has a database file as input, which is called ASS1.db in our case, and tries to make a connection with the database using the sqlite3 library. If the connection is successful, it returns a connection object, otherwise, it returns None. The 'run_query' takes as input a connection object and a query and returns the result of the query with the use of the class cursor. Now that all necessary functions are created, the actual code for the connection and the queries can be written as can be seen in the next code chunk. It first selects the correct file for the database and tries to make a connection with the create_connection function. If the connection is successful, the queries are given to the function 'run_query' with the object and it outputs the results as expected.
b. The libraries that are needed for this part are already imported as described in 3a. A new function 'convert_db_table_to_DF' is created to convert a table to a dataframe. The function uses the 'run_query' to obtain the table headers and the content separately and combines them in a dataframe using the panda library. Just like in 3a, we establish a connection with the database and use the new function to receive the customer table as a dataframe.
c. Having access to the Customers table from the previous step, we need to compare each record of this table with the others in order to provide those with a similarity greater than 0.7. First, we create a list of each row in the Customer table. After that, we create a double "for" loop to compare all these data with each other, while we also use the "jaccard_similarity" function to compare two rows at a time and find their similarity. Finally, it displays the identifiers only those whose similarity number is above 0.7.