Assignment 1

Nisse Hermsen, Isabelle de Wolf, Sara Sakhi,
Celis Tittse, Agnieszka Kubica

23 September 2024

Introduction

This report presents the redesign and querying of a banking database. The assignment was completed as a collaborative effort by Nisse Hermsen, Isabelle de Wolf, Sara Sakhi, Celis Tittse, and Agnieszka Kubica. The tasks required adding attributes to existing relations, introducing new entities, and designing the database schema. Additionally, we implemented SQL queries to extract meaningful insights from the database, applied relational algebra, and used Python to perform data extraction and entity resolution. This document outlines the key design decisions, the SQL queries written, and the Python code used to analyse the data.

Task 1: Database redesign

a. Add more attributes to the given relations (branch, customer, loan).

We added the attributes first_name and surname to the customer, phone_number to the branch, and customer_id and currency to the loan. Adding first_name and surname allows for better identification and personalised communication with customers, as it separates the customer's name into meaningful components. The phone_number attribute in the branch table would make it easier to contact specific branches directly, and we thought it would be interesting to insert an optional attribute (we considered a scenario in which not all branches have their phone numbers).

Including customer_id in the loan, the table establishes a foreign key and a direct link between a loan and its associated customer. Adding currency ensures that loans can be tracked in different currencies, facilitating multi-currency transactions and reporting. These additions enhance data accuracy, relational integrity, and user convenience in banking operations.

b. Add a minimum of two relations that represent entities.

Two relations were added to the database, representing the account and employee entities.

The account entity represents the bank accounts customers can keep and branches can maintain. The attributes account_id, and balance were added, as well as customer_id, and branch_id as foreign keys.

The employee entity represents an employee working at a branch. Relevant attributes such as employee_id, first_name, surname, salary, and role were added, as well as branch_id, which serves as a foreign key referring to the branch entity.

c. Draw the schema chart for the database.

The database schema was first drawn on a whiteboard on campus to allow for group work. This drawing was digitised, and the schema chart is visible in figure 1.

Significantly, branch_name was changed to branch_id in the loan relation. This choice was made because branch_id is the primary key of the branch relation and an ID variable. The change facilitates consistency in the database design, as all foreign key variables are ID variables and the primary keys of other ties.

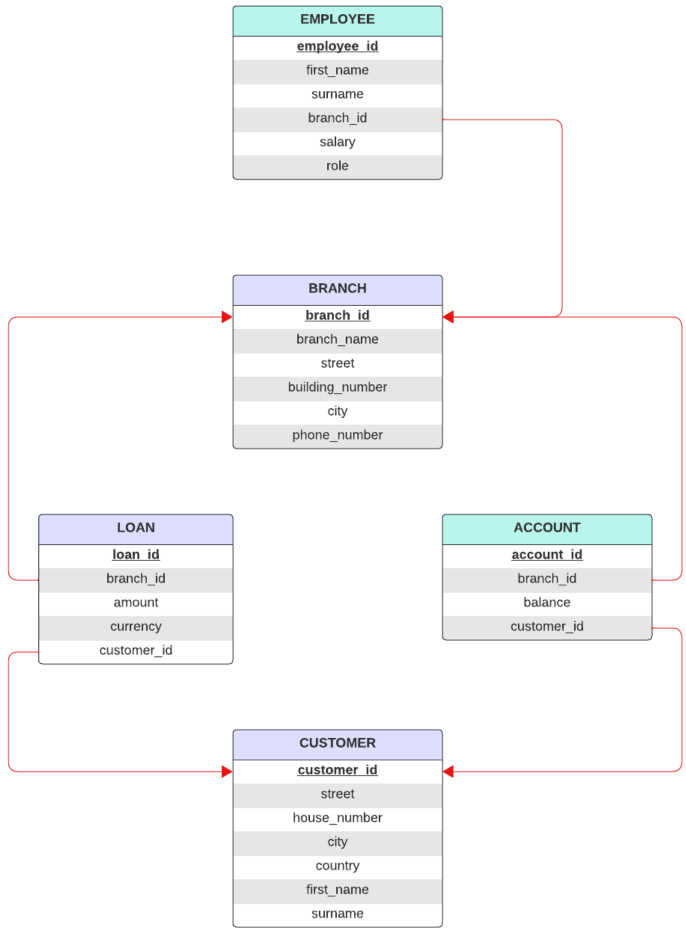

Figure 1: The schema chart of the database.

d. Identify the cardinalities of the relationships between the entities and explain how you can represent the relationships in your design.

The cardinalities of the relationships between the entities in the design are one-to-many. Figure 1 shows this through the red lines connecting the entities.

The relationship between employee and branch is one-to-many. An employee can only be employed at one branch, and a branch can have many employees.

The relationship between loans and branches is also one-to-many. One branch can only maintain a loan, but a branch can maintain many loans. The same idea applies to the account and branch relationship. An account can only be maintained by one branch, but a branch can maintain many accounts; therefore, it is a one-to-many relationship.

A customer can have multiple bank accounts and loans, but a loan and bank account can only be related to one customer. Thus, other customers cannot share a bank account or loan. The team made this conscious choice for this assignment.

e. List all the primary-key-foreign-key constraints in the database.

The corresponding ID was chosen as the primary key for each relation in the database schema. This entails that loan_id is the primary key for the loan relation, employee_id is the primary key for the employee relation, branch_id is the primary key for the branch relation, account_id is the primary key for the account relation, and customer_id is the primary key for the customer relation.

Three relations contain foreign keys that refer to different relations. The employee relation has branch_id as an attribute, a foreign key to the branch relation. The loan relation contains two attributes that are also foreign keys. The branch_id attribute is a foreign key referring to the branch relation, while the customer_id attribute refers to the customer relation. The same principle applies to account relations; They contain two foreign keys. The branch_id attribute is a foreign key for the branch relation, while the customer_id refers to the customer relation.

In figure 1, each entity's primary keys are bold and underlined, while the red arrowheads point to the foreign keys.

f. Write the SQL code that creates the whole database in SQLite and save the file as .sql

The tables were created using SQL in the create.sql file. We used the "autoincrement" statement for the primary keys to ensure the IDs were incremented with every new record. A suiting type was chosen for all attributes. Most attributes are also restricted from containing null values. This is because these fields are considered mandatory in our design. The only field we felt was not compulsory and thus can be null is the phone_number attribute in the branch relation.

Two minor things to note are that the branch_name has to be a unique value. This is to ensure that no two distinct branches carry the same name. The employee relation also contains a salary check. It was decided that every employee should have a salary, so the salary attribute cannot be null. We did, however, include a check to ensure that the salary can never be a negative value.

The create.sql file is included in the assignment submission.

g. Pick one of the relations (tables) and explain if it is in the BCNF or not and why.

For this assignment, we examine whether customer relations are in BCNF. First, all functional dependencies were considered. The only functional dependency in this relation is as follows:

$$customer\_id \rightarrow street, house\_number, city, country, first\_name, surname$$

customer_id is superkey for customer relations. Thus, no functional dependencies that are not trivial or based on the superkey. Therefore, the relation is in BCNF.

Task 2: Querying the database

In this task, we used the database created in Task 1 to write queries demonstrating different SQL features. Specifically, we implemented a query that combines a left outer join, an aggregate function, and a nested query. We started by writing the query in SQL, as can be seen in the SQL file or below:

```
select
    c.customer_id,
    sum(a.balance)
from (
    select *
    from customer
    where country = "Vatican City"
) as c
left join account as a on c.customer_id = a.customer_id
group by c.customer_id
```

Natural Language Explanation:

The task requires a query that finds the total balance in the accounts of customers who live in Vatican City. We must ensure all customers are considered, even those without an account (hence, the left outer join). Additionally, we group the results by customer, summing up their account balances (this represents the aggregate function). Finally, a nested query will filter only those customers from Vatican City.

Query Explanation:

- We are retrieving all customers who live in Vatican City.
- We perform a left outer join between the customer and account tables to include all customers.
- We are summing the account balances for each customer.

Relational Algebra:

$$_{c.customer\_id}\gamma_{sum(balance)}(\rho_c(\sigma_{country=Vatican}(customer)) ⟕ \sigma_a(account))$$

Step 1: We first select the customers from the Customer table who live in Vatican City:

Step 2: Next, we perform a left outer join between this filtered customer table (`C`) and the `Account` table. The join links customers and their associated accounts.

Step 3: Finally, we use the aggregation operation to sum the `balance` for each customer, grouped by their `customer_id`:

Task 3: Data extraction and entity resolution using Python

The entirety of the Task 3 code is called and executed using the `main` function below, which acts as a wrapper function. Sections 3a, 3 b, and 3c explain each sub-function being called.

```
def main():
    # 3.a Establish database connection.
    connection = create_connection("data/assignment_1.sqlite")
    if the connection is None:
        return

    # 3.b Retrieve the customer data.
    df = read_customer(connection)
    print(df)

    # 3.c Calculate pairwise similarity score between customer records.
    sim_df = pairwise_similarity(jaccard_similarity, df)
    print(sim_df)

    print("\nSimilarity scores > 0.7")
    print(sim_df[sim_df["similarity_score"] >= 0.7])
```

a. Write Python code to connect, send, and read from a database.

We opted not to use the cursor object since the panda's library offers a neat built-in function called read_sql that reads the data and casts it to a dataframe. Hence, only a database connection is required, which is created using the following function:

```
def create_connection(db_file: str) -> sqlite3.Connection | None:
"""
```

```python
Create a database connection to the SQLite database specified
by the db_file.

:param db_file: Database file
:return: Connection object or None
"""
try:
    return sqlite3.connect(db_file)
except Exception as e:
    print(e)
    return None
```

If an exception is raised during the creation of the connection, `None` is returned. This is, in turn, recognised by the `main` function, which ceases its execution, preventing further errors during runtime.

b. Read 'customer' data into a Dataframe object.

As mentioned, rather than manually fetching all records from a `cursor` object and casting them to a Dataframe, we use the read_sql function provided by pandas. We first create a string representing our SQL query—which, in our case, selects all data from the `customer` table—and passes this to the function, together with the earlier created connection object.

It should also be noted that the third parameter, index_col, allows us to use the `customer_id` column as an index in the `Dataframe`, as these have similar purposes. This prevented us from accidentally including the `customer_id` attribute in the similarity algorithm (3.c), thus avoiding messy column selections.

```python
def read_customer(connection: sqlite3.Connection) -> pd.DataFrame:
    """
    Use the built-in `read_sql` function to read data from
    the `customer` table and cast the object to a dataframe.
    The `customer_id` attribute is set as the Dataframe index.

    :param connection: Database connection
    :return: DataFrame containing the customer data
    """
    query = """
    select
        *
    from customer
    """

    df = pd.read_sql(query, con=connection, index_col='customer_id')
    return df
```

Running the code above on the database results in the following `Dataframe`:

```
                            street house_number      city        country  \
customer_id
1             Via delle Fondameta            1   Vatican   Vatican City
2            Jeungsan-ro, Mapo-gu           87     Seoul    South-Korea
3                        Broadway          219  New York  United States
4                        Broadway          219  New York  United States

              first_name     surname
customer_id
1            Innocensius          II
2                    Kim  Impossible
3                Orpheus          II
4               Eurydice          II
```

c. Using a pair-wise similarity function, report customers with *similarity* > 0.7.

We chose to apply the Jaccard similarity algorithm to our `customer` data since each attribute (e.g., first_name, street) of the relation can be seen as a single entry in a set, and consequently, an entire record is an instance of a set. Thus, comparing two records naturally means comparing two sets with one another.

Realising that the implementation of this algorithm was two-part, we implemented two functions: Firstly, the `jaccard_similarity` calculates the Jaccard similarity between two records (records referring to two `pandas.Series` objects), and secondly, `pairwise_similarity` applies this calculation to each pairwise combination within a table.

The code below compares two records using the Jaccard similarity algorithm. The function accepts two `pandas.Series` object (or rows) and converts them to a set before using these sets in the equation. Due to casting a `Series` to a `set`, no additional data manipulation overhead was required. Lastly, dividing by zero results in a similarity score of 0.

```
def jaccard_similarity(r1: pd.Series, r2: pd.Series) -> float:
    """
    Calculate the Jaccard similarity score over a pair of Dataframe records.

    :param r1: first record in pair
    :param r2: second record in pair
    :return: Similarity score
    """
    s1 = set(r1)
    s2 = set(r2)

    try:
```

```python
        return len(s1.intersection(s2)) / len(s1.union(s2))
    except ZeroDivisionError:
        return 0
```

When implementing pairwise comparisons between the records of a relation, we considered the code's performance, seeing that the algorithm's complexity would be $O(n^2)$.

We considered using vectorised operations to facilitate this pairwise comparison—using a cross-join and then applying (`Dataframe.apply()`) a custom method upon the data to obtain the results. The cross-join could be realised either using `SQL` when querying the database, or using the `pd.merge()` method after data acquisition. The performance cost would be much lower, so this assignment's implementation would be out-of-scope. Instead, we opted for the more traditional approach of using a double for loop over the records. Whilst not as optimal as vectorised operations, this code aligns more with the material covered.

The function below thus iterates over the `Dataframe` twice and compares each row with one another, excluding comparisons that have already been made and comparing a row to itself. The exclusion criteria are captured in the `if j <= i:  continue` guard. All entries are saved into a nested list, after which the function returns a `Dataframe` object containing said data. The returned data contains each possible pair of records and their similarity score. With the return type being a `Dataframe`, future filtering on the data is intuitive and quick to implement (e.g. when wanting to retrieve all similarity scores > 0.7).

```python
def pairwise_similarity(sim: Callable[[pd.Series, pd.Series], float],
df: pd.DataFrame) -> pd.DataFrame:
    """
    Using the given similarity function, retrieve the pair-wise similarity
    score between records. This implementation can use any similarity function
    and is data-agnostic. The algorithm also avoids duplicate and identity
    comparisons.

    :param sim: Similarity function (e.g. Jaccard similarity)
    :param df: Dataframe containing the records
    :return: Dataframe with pairwise record similarities
    """

    data = []
    for i, r1 in df.iterrows():
        for j, r2 in df.iterrows():
            if j <= i:
                continue

            similarity_score = sim(r1, r2)
            data.append([i, j, r1["first_name"], r2["first_name"], similarity_score])
```

```
    return pd.DataFrame(
        data,
        columns=["customer_id_x", "customer_id_y", "first_name_x", "first_name_y",
        "similarity_score"]
    )
```

It should be noted that the code above is data-agnostic—the indexing and iterating of the data are not specific to the customer Dataframe and can be used for any relation. Additionally, any similarity score algorithm can be used within this function since the argument sim is abstract, allowing any logic to be accepted if the type signature is correct.

Calling the `pairwise_similarity` function on the `customer` data gives the following result:

```
[4 rows x 6 columns]
   customer_id_x  customer_id_y first_name_x first_name_y  similarity_score
0              1              2  Innocensius          Kim          0.000000
1              1              3  Innocensius      Orpheus          0.090909
2              1              4  Innocensius     Eurydice          0.090909
3              2              3          Kim      Orpheus          0.000000
4              2              4          Kim     Eurydice          0.000000
5              3              4      Orpheus     Eurydice          0.714286
```

Applying a filter that only accepts similarity scores > 0.7 gives:

```
   customer_id_x  customer_id_y first_name_x first_name_y  similarity_score
5              3              4      Orpheus     Eurydice          0.714286
```

Conclusion

In conclusion, this assignment focused on redesigning a banking database by adding new attributes and relations. The SQL query implementation demonstrated the use of joins, nested queries, and aggregate functions for efficient data retrieval. A Python-based solution was also developed for database interaction, data extraction, and entity resolution using a pairwise similarity algorithm.