Assignment 1 2025/9/23 9AM

Group Members

- Ilse van Deventer 9996974
- Majdouline Hamdi 9767738
- Zexuan Li 8069182
- Zoé Ricardie 9107096
- Menno Zoetbrood 1084720

Task 1: Database (Re)Design

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

Branch

- `branch_id` : unique id of branches
- `branch_name` : name of branches
- `street` : name of the streets
- `house_number` : number of the houses
- `city` : name of the cities
- `country` : name of the countries
- `postcode` : postcode of the address
- `email` : email address of branches

Customer

- `customer_id` : unique id of customers
- `name` : name of customers
- `gender` : gender of customers
- `street` : name of the streets
- `house_number` : number of houses
- `city` : name of the cities

- `country` : name of the countries
- `phone_number` : unique phone number of each customer
- `email` : unique email of each customer
- `customer_category` : type of customer(regular/premium)
- `date_of_birth` : date of birth of customers
- `create_date` : date of the data of each customer is created

Partners

- `partner_id` : unique id of the partners
- `company_name` : name of the companies
- `country` : name of the countries
- `products_and_services` : products and services from our partners
- `email` : email address of partners
- `industry` : industry type of our partners
- `is_active` : whether our company is still collabrating with

b. Add a minimum of two relations that represent entities. Make sure that there is at least one many-to-many relationship between the entities.

Bill

- `bill_id` : unique id of bills
- `branch_id` : unique id of branches
- `amount` : amount of purchases
- `customer_id` : unique id of customers
- `bill_date` : date of the bill issued
- `due_date` : date of the bill dued
- `status` : status of the bill(draft,issued,paid,overdue,cancelled,refund)
- `payment_methods` : payment_methods(bank_transfer, ideal,apple_pay)

Product

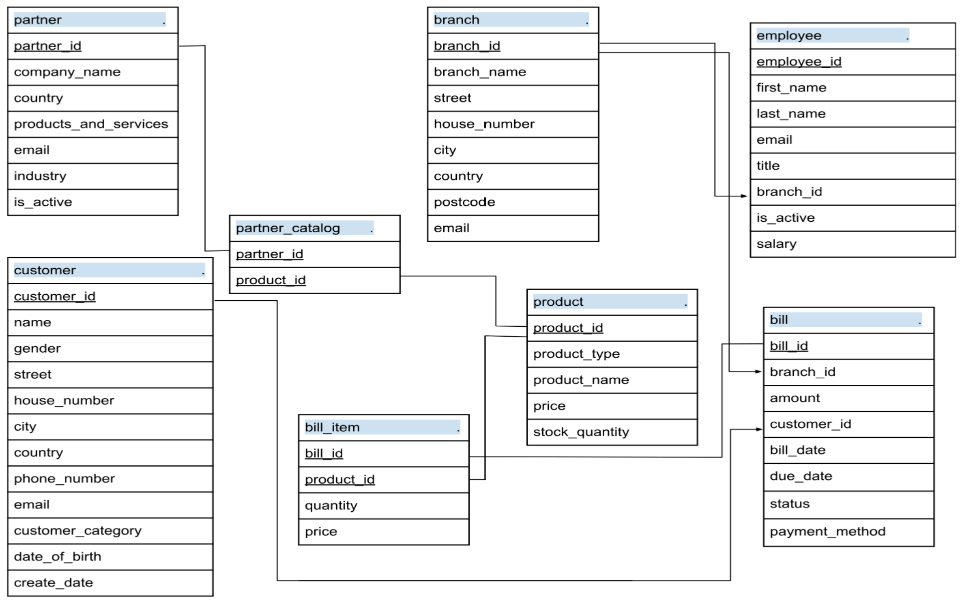
- `product_id` : unique id of products
- `product_type` : type of the products

- `product_name` : name of the products
- `price` : price per product
- `stock_quantity` : number of remain stock

Employee

- `employee_id` : unique id of employees
- `first_name` : first name of each employee
- `last_name` : last name of each employee
- `email` : email of each employee
- `title` : position of each employee
- `branch_id` : unique id of branches
- `is_active` : whether the employee is hired
- `salary` : salary of each employee

c. Draw the schema chart for the database.



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

One-to-Many Relationships

1. **Customer → Bill**: One customer can have multiple bills, each bill belongs to one customer.
   Foreign key(customer_id) in bill referencing to customer_id in customer table.
2. **Branch → Employee**: One branch can have many employees, each employee works in exactly one branch.
   Foreign key(branch_id) in employee referencing to branch_id in branch table.
3. **Branch → Bill**: One branch can issue many bills, each bill belongs to one branch.
   Foreign key(branch_id) in bill referencing to branch_id in branch table.

Many-to-Many Relationships

1. **Bill ↔ Product**: A bill can contain multiple products; a product can appear in multiple bills.
   Join table bill_item with foreign keys bill_id, product_id referencing to bill table and product table
2. **Partner ↔ Product**: A partner can supply multiple products; a product can be supplied by multiple partners.
   Join table partner_catalog with foreign keys partner_id, product_id referencing to partner table and product table

e. Primary Key and Foreign Key Constraints

Bill

- **Primary Key:** `bill_id`
- **Foreign Keys:** `branch_id → branch(branch_id)` , `customer_id → customer(customer_id)`

Customer

- **Primary Key:** `customer_id`

Partner

- **Primary Key:** `partner_id`

Product

- **Primary Key:** `product_id`

Branch

- **Primary Key:** `branch_id`

Employee

- **Primary Key:** `employee_id`
- **Foreign Key:** `branch_id → branch(branch_id)`

Bill_Item

- **Primary Key:** `(bill_id, product_id)`
- **Foreign Keys:** `bill_id → bill(bill_id)` , `product_id → product(product_id)`

Partner_Catalog

- **Primary Key:** `(partner_id, product_id)`
- **Foreign Keys:** `partner_id → partner(partner_id)` , `product_id → product(product_id)`

g. BCNF Analysis

Pick one relation (table) and explain if it is in **BCNF**. (You can add your BCNF justification here.)

Definition of BCNF

A relation `R` is in **Boyce-Codd Normal Form (BCNF)** if, for **all functional dependencies (FDs)** `X → Y`, where `X ⊆ R` and `Y ⊆ R`, **one of the following holds**:

1. The FD is **trivial** (`Y ⊆ X`), or
2. `X` is a **superkey** for `R`.

> A superkey `K` for a relation `R` is a set of attributes such that `K → R`.

Candidate Keys for Customer

The candidate keys for the `Customer` relation are:

- `customer_id` (primary key)
- `phone_number` (unique constraint)
- `email` (unique constraint)

Functional Dependencies

Determinant	Dependent Attributes
`customer_id`	street, name, gender, house_number, city, country, phone_number, email, customer_category, date_of_birth, create_date
`phone_number`	customer_id, street, house_number, city, country, email, customer_category, create_date
`email`	customer_id, street, house_number, city, country, phone_number, customer_category, create_date

BCNF Check

- For each FD, the **determinant** (`customer_id` , `phone_number` , `email`) is a **superkey**.
- Therefore, all FDs satisfy the BCNF condition.

Conclusion: The `Customer` relation is in **BCNF**.

Task 2: Querying the Database

a. Joining More Than Two Tables

Natural Language:

Return the customers' names and genders (together) with the product names they have purchased.

Relational Algebra:

$\pi_{\text{name, gender, product\_name}}\left(((\text{customer} \bowtie \text{bill}) \bowtie \text{bill\_product}) \bowtie \text{product}\right)$

SQL:

```
SELECT c.name AS customer_name,
       c.gender,
       p.product_name
FROM customer c
JOIN bill bl ON c.customer_id = bl.customer_id
JOIN bill_product bp ON bl.bill_id = bp.bill_id
JOIN product p ON bp.product_id = p.product_id;
```

b. Aggregate function:

Natural language:

return the total and average bill amount for each branch.

Relational Algebra:

$$\gamma_{branch\_name;\, \text{SUM}(amount)\,,\, \text{AVG}(amount)}\ (bill\ \bowtie\ branch)$$

SQL:

```
SELECT
     br.branch_name,
     SUM(b.amount) AS total_amount,
     AVG(b.amount) AS avg_amount
FROM bill b
JOIN branch br ON b.branch_id = br.branch_id
GROUP BY br.branch_name;
```

Natural language:

return to me how many employees work at each branch.

Relational Algebra:

$$\gamma_{branch\_name;\ \mathrm{COUNT}(amount)}\ (bill\ \bowtie\ branch)$$

SQL:

```
SELECT br.branch_name,COUNT(e.employee_id) AS total_employees_amoutn
FROM employee e
JOIN branch br ON e.branch_id = br.branch_id
GROUP BY br.branch_name;
```

Natural language:

Return to me the average age of customers grouped by gender.

Relational Algebra:

$$\gamma_{gender,\ AVG(age)}(customer)$$

SQL:

```
SELECT gender, avg(cast((julianday('now') - julianday(date_of_birth)) / 365 AS INT)) AS average_
FROM customer
GROUP BY gender;
```

c. Nested query:

Natural language:

return the customers who have at least one bill with an amount higher than the average of all bills.

Relational Algebra:

$$\delta\left(\pi_{customer\_id,email}\left(\sigma_{amount>(\gamma_{AVG(amount)}(bill))}(customer\ \bowtie\ bill)\right)\right)$$

SQL:

```
SELECT
    DISTINCT c.customer_id,
    c.email
FROM customer c
JOIN bill b ON c.customer_id = b.customer_id
WHERE b.amount > (
    SELECT AVG(amount) FROM bill
);
```

Natural language:

return to me the names of customers who are older than the average age of all customers

Relational Algebra:

$$\pi_{name}\left(\sigma_{age>AVG(age)}(customer)\right)$$

SQL:

```
SELECT name
FROM customer
WHERE (julianday('now') - julianday(date_of_birth)) / 365 > (
    SELECT avg((julianday('now') - julianday(date_of_birth)) / 365)
    FROM customer
);
```

Task 3: Data extraction and entity resolution using Python

Using the database from Task 1, perform the following tasks using Python code:

a. Write a Python code that allows you to connect to the database file, send SQL queries to the database and extract the results.

Before executing any line of code, make sure that Google Colab has access to the “assignment_1.db” file

Overview

1. **Verify Column Data Types**
    - We first examined whether the data type of each column in the SQLite database was correctly interpreted in Python.
    - This was done by querying the `sqlite_master` table, which contains metadata about the database schema.
    - Additionally, the `PRAGMA table_info(table_name)` command was used to examine each table's columns and their data types.
    - This ensures that all column types are accurately transferred before performing further analysis.
2. **Define Key Functions**
   Three functions were defined to facilitate interaction with the SQLite database:
    - `create_connection(db_file)` – establishes a connection to the database.
    - `run_query(conn, query)` – executes a SQL query and returns the results.
    - `convert_db_table_to_DF(conn, table)` – uses the above functions to retrieve a table and convert it into a Pandas DataFrame.

> Note: The variable `conn` is defined in two contexts:
>
> - Globally (cell 2) for general connection and cursor.
> - Locally (inside `create_connection`) for creating a connection object within the function.

Python Code: Create Database Connection

```python
# Load necessary packages
import pandas as pd
import sqlite3
from sqlite3 import Error

# Function to create a connection to the SQLite database
def create_connection(db_file):
    """
    Create a database connection to the SQLite database specified by db_file.

    :param db_file: Path to the database file
    :return: Connection object or None if connection fails
    """
    conn = None
    try:
        conn = sqlite3.connect(db_file)
        print("Connection successful")
    except Error as e:
        print(f"Connection failed: {e}")
    return conn


#Sending the query to the database and receiving the resulting relation

def run_query(conn, query):
    """
    Query all rows in the tasks table
    :param conn: the Connection object
    :return:
    """
    # Create a cursor
    cur = conn.cursor()
    # Send the query to the database
    cur.execute(query)
    # Extract the results of the query
    results = cur.fetchall()
    # Return the results
    return results
```

```
#Creating a dataframe from the resulting relation

def convert_db_table_to_DF(conn, table):
    # get the names of the attributes in the database table
    header_query = "SELECT name FROM pragma_table_info('" + table + "') ORDER BY cid;"
    # print (header_query)
    cols_init = run_query(conn, header_query)
    cols = [cols_init[i][0] for i in range(len(cols_init))]
    # print(cols)
    # get the records of the table
    content_query = "Select * from " + table
    data = run_query(conn, content_query)
    df = pd.DataFrame(data, columns = cols)
    return df
```

b. Read the content of the customer relation (table) into Pandas DataFrame.

Continuing from Part A, we successfully read the `customer` table into a Pandas DataFrame using the previously defined functions. We examined the data types for each column in the `customer` table and found that all columns were of the `object` type. This occurs because Pandas DataFrames do not preserve `varchar` data types, converting all string-type columns to `object` .

However, the `date_of_birth` and `create_date` columns should not be classified as `object` . Therefore, they were converted to the more appropriate `datetime64[ns]` type, ensuring that date-based operations can be performed accurately and efficiently.

Python Code: Read customer Table

```
database = "assignment_1.db"

#create a database connection
conn_trade = create_connection(database)
with conn_trade:
  tab_name = 'customer'
  customer = convert_db_table_to_DF(conn_trade, tab_name)

#display the content of the DataFrame
customer
```

c. Using a similarity function that compares two records (similar to the one in the tutorial), report the customers with similarity > 0.7.

To answer this question, we created a Jaccard similarity function. We chose Jaccard similarity because our comparison is set-based—we are comparing two records (rows) and their values. Additionally, we are examining the overlap of answers between records, making Jaccard the optimal choice.

Python Code: Calculate Similarity

```python
#create a function to compare how similar two sets are
def jaccard_similarity(set1, set2):
    intersection_size = len(set1.intersection(set2))
    union_size = len(set1.union(set2))
    return intersection_size / union_size if union_size != 0 else 0.0

#transform every row into a list of sets (one for each row)
row_sets = [set(customer.iloc[i].to_numpy()) for i in range(len(customer))]

#storing the results in a list
results = []

for i in range(len(row_sets)):
    for j in range(i + 1, len(row_sets)):
        sim = jaccard_similarity(row_sets[i], row_sets[j])
        results.append((i, j, sim))
        if sim > 0.7:
            print(f"Similarity between row {i+1} and row {j+1} = {sim:.3f} which is greater tha

#Row 1 and row 6 have a Jaccard similarity score above 0.7
```