CS4221 Tutorial 1: Relational Database Design and PostgreSQL

Yao LU

2024 Semester 2

Objective

By the end of this tutorial, you will:

- Understand principles of relational database design and normalization.
- Design and implement a database schema using PostgreSQL.
- Populate the database with sample data.
- Write SQL queries to retrieve, analyze, and manipulate data.

Prerequisites

Before starting, ensure you have:

- Python (preferred 3.9) installed.
- Jupyter Notebook installed.
- Basic understanding of SQL and Python.

Setting Up PostgreSQL

Step 1: Install PostgreSQL

- Download from `https://www.postgresql.org/download/`.
- Note down username, password, and port.
- Verify PostgreSQL is running.

Step 2: Create a Database

- Follow the guidelines in the Juypter Notebook to creat a table.

Introduction to Normalization

Example Context: Singgah Technologies

Singgah Technologies has been tasked with creating a payment portal database that includes:

- **Customers:** Each customer is identified by a unique Social Security Number (SSN). Additional attributes include first name, last name, and country of residence.
- **Credit Cards:** Each credit card has a unique number, a type (e.g., Visa, MasterCard), and is linked to a customer.
- **Merchants:** Merchants are identified by a unique code and include attributes such as name and country.
- **Transactions:** Each transaction involves a credit card and a merchant. It is recorded with an identifier, transaction date, and amount.

Let's design a schema to efficiently store and retrieve data related to payments and transactions while adhering to relational database principles.

First Normal Form (1NF)

What is 1NF?

- A table is in **First Normal Form (1NF)** if it meets the following criteria:
  - Each column contains atomic (indivisible) values.
  - Each row is unique.
  - There are no repeating groups or arrays within a column.
- The goal of 1NF is to ensure that the data is stored in a clear and organized manner, avoiding redundancy and anomalies.

Non-1NF Table:

Social Security Number	Name	Credit Cards
S123456789	Alice Tan	Visa:1234, Master:5678
S987654321	Bob Lim	Visa:2345

Issues with Non-1NF:

- The `Credit Cards` column contains multiple values, violating the atomicity requirement.
- Querying specific credit card details becomes complex and error-prone.

First Normal Form (1NF)

Fix (1NF):

SSN	Name	Card Type	Card Number
S123456789	Alice Tan	Visa	1234
S123456789	Alice Tan	Master	5678
S987654321	Bob Lim	Visa	2345

Benefits of 1NF:

- Simplifies querying and updating data.
- Eliminates redundancy within a column.
- Forms a foundation for higher normalization forms.

Second Normal Form (2NF)

What is 2NF?

- A table is in **Second Normal Form (2NF)** if it:
  - Is already in 1NF.
  - Has no partial dependency: no non-prime attribute depends on a part of a composite primary key.
- The goal of 2NF is to ensure that all attributes are fully functionally dependent on the whole primary key.

Non-2NF Table:

Card Number	Card Type	Customer Name
1234	Visa	Alice Tan
5678	Master	Alice Tan
2345	Visa	Bob Lim

Issues with Non-2NF:

- Customer Name depends on SSN, not on Card Number.

Second Normal Form (2NF)

Fix (2NF):

- Decompose the table into two smaller tables.

Decomposed Tables:

Credit Cards Table:

Card Number	Card Type
1234	Visa
5678	Master
2345	Visa

Customers Table:

SSN	Customer Name
S123456789	Alice Tan
S987654321	Bob Lim

Third Normal Form (3NF)

What is 3NF?

- A table is in **Third Normal Form (3NF)** if it:
  - Is already in 2NF.
  - Has no transitive dependencies: no non-prime attribute depends on another non-prime attribute.
- The goal of 3NF is to eliminate dependencies between non-key attributes.

Non-3NF Table:

Transaction ID	Card Number	Merchant Name	Merchant Country
1	1234	Store A	Singapore
2	5678	Store B	Malaysia

Issues with Non-3NF:

- Merchant Country depends on Merchant Name, not on Transaction ID.

Third Normal Form (3NF)

Fix (3NF):

- Decompose the table into two smaller tables.

Decomposed Tables:

Transactions Table:

Transaction ID	Card Number	Merchant Name
1	1234	Store A
2	5678	Store B

Merchants Table:

Merchant Name	Merchant Country
Store A	Singapore
Store B	Malaysia

Boyce-Codd Normal Form (BCNF)

What is BCNF?

- A table is in **Boyce-Codd Normal Form (BCNF)** if it:
  - Is already in 3NF.
  - For every functional dependency (X $\rightarrow$ Y), X is a superkey.
- BCNF eliminates anomalies caused by functional dependencies where the determinant is not a superkey.

BCNF in Our Example:

- The 3NF example provided above already satisfies BCNF because every determinant in the functional dependencies is a superkey.
- No further decomposition is required.

ER Diagram

Creating an ER Diagram in Python

- To create and visualize an Entity-Relationship (ER) diagram, we use the `graphviz` Python package.
- This package helps generate diagrams programmatically.

Installation: ‘pip install graphviz’

Code to Draw ER Diagram:

```
# Define entities
er_diagram.node('Customers', 'Customers\nssn
(PK)\nfirst_name\nlast_name \ncountry', shape='box')

# Define relationships
# One customer can have 0 to N credit cards.
er_diagram.edge('Credit_Cards', 'Customers', label='0..N to 1..1
(owner_ssn)')

# TODO: Add your code here
```

Table Creation: PostgreSQL

Customers Table:

```
CREATE TABLE customers (
    ssn CHAR(11) PRIMARY KEY,
    first_name VARCHAR(32),
    last_name VARCHAR(32),
    country VARCHAR(16)
);
```

Credit Cards Table:

```
CREATE TABLE credit_cards (
    number VARCHAR(20) PRIMARY KEY,
    type VARCHAR(32),
    ssn CHAR(11) REFERENCES customers(ssn)
);
```

Table Creation: PostgreSQL - Merchants Table

Merchants Table:

```
CREATE TABLE merchants (
    code CHAR(10) PRIMARY KEY,
    name VARCHAR(64),
    country VARCHAR(16)
);
```

Table Creation: PostgreSQL - Transactions Table

Transactions Table:

```
CREATE TABLE transactions (
    identifier INTEGER PRIMARY KEY,
    number VARCHAR(20) REFERENCES credit_cards(number),
    code CHAR(10) REFERENCES merchants(code),
    datetime TIMESTAMP,
    amount NUMERIC
);
```

Populate Tables

Using Mockaroo for Sample Data:

- Use tools like Mockaroo to generate realistic sample data for:
  - 100 customers.
  - 300 credit cards.
  - 20 merchants.
  - 300 transactions.
- Mockaroo allows you to export data in SQL or CSV format for quick integration.

Populate Tables

How to Generate Tables Manually:

- For tables without foreign keys (`customers`, `merchants`), directly generate from Mockaroo and insert data.
- For tables with foreign keys (`credit_cards`, `transactions`), generate temporary tables and assign keys using SQL.

Example for `credit_cards` Table:

```
INSERT INTO credit_cards
SELECT comb.ssn, comb.number, comb.type
FROM (
    SELECT c.ssn, cc.number, cc.type,
    ROW_NUMBER() OVER(PARTITION BY cc.number) as row
    FROM credit_cards_temp cc, customers c
    WHERE RANDOM() < 0.2
) comb
WHERE comb.row = 1
ORDER BY comb.number;
```

Populate Transactions Table

Example for `transactions` Table:

```
INSERT INTO transactions
SELECT ROW_NUMBER() OVER () as identifier, *
FROM (
    SELECT cc.number, m.code, t.datetime, t.amount
    FROM transactions_temp t, merchants m,
    credit_cards cc
    ORDER BY RANDOM() LIMIT 3000
) temp;
```

Ensure Proper Constraints:

- Maintain data consistency by ensuring foreign key relationships are respected.
- Verify domain constraints for all columns (e.g., date formats, numeric ranges).

Pre-prepared Sample Data

Using Pre-prepared SQL Files:
To simplify the setup, we provide SQL files with pre-generated data:

```python
with open('code/CCMerchants.sql', 'r') as file:
    query_to_insert_merchants = file.read()

with open('code/CCCustomers.sql', 'r') as file:
    query_to_insert_customers = file.read()

with open('code/CCTransactions.sql', 'r') as file:
    query_to_insert_transactions = file.read()

with open('code/CCCreditCards.sql', 'r') as file:
    query_to_insert_card = file.read()
```

Execute these queries to populate tables with sample data.
Example Queries to Test Data:

```sql
SELECT * FROM customers LIMIT 5;
SELECT * FROM credit_cards LIMIT 5;
SELECT * FROM merchants LIMIT 5;
SELECT * FROM transactions LIMIT 5;
```

Task 1: Customers with Both JCB and Visa Credit Cards

Problem Statement:

- Find the first and last names of customers in Singapore who own both JCB and Visa credit cards.
- Ensure that each customer is uniquely identified, even if their names are identical to others in the database.
- Output should not print the same customer more than once.

Correct Query for Task 1

Efficient Solution Using Self-Joins:

```
SELECT c.first_name, c.last_name
FROM customers c, credit_cards cc1, credit_cards cc2
WHERE c.ssn = cc1.ssn
  AND c.ssn = cc2.ssn
  AND cc1.type = 'jcb'
  AND cc2.type = 'visa'
  AND c.country = 'Singapore'
GROUP BY c.ssn, c.first_name, c.last_name;
```

Explanation:

- `cc1` and `cc2`: Two instances of the `credit_cards` table are joined with `customers`.
- Filters ensure only customers in Singapore with both JCB and Visa cards are retrieved.
- `GROUP BY` ensures uniqueness by grouping results by SSN and name.

Alternate Solution with Subqueries

Using Subqueries with IN:

```sql
SELECT c.first_name, c.last_name
FROM customers c
WHERE c.ssn IN (
    SELECT cc1.ssn
    FROM credit_cards cc1
    WHERE cc1.type = 'jcb'
)
AND c.ssn IN (
    SELECT cc2.ssn
    FROM credit_cards cc2
    WHERE cc2.type = 'visa'
)
AND c.country = 'Singapore';
```

Explanation:

- Subqueries find SSNs of customers with JCB and Visa cards separately.
- Main query retrieves customer details for SSNs found in both subqueries.
- Less efficient than self-joins, especially for large datasets.

Common Mistake: Incorrect Query

Example of an Incorrect Query:

```
SELECT DISTINCT c.first_name, c.last_name
FROM customers c, credit_cards cc1, credit_cards cc2
WHERE c.ssn = cc1.ssn
  AND c.ssn = cc2.ssn
  AND cc1.type = 'jcb'
  AND cc2.type = 'visa'
  AND c.country = 'Singapore';
```

Why It's Wrong:

- `DISTINCT` removes duplicate rows but does not ensure unique customers by SSN.
- Customers with the same name but different SSNs might cause ambiguity.

Task 1: Summary and Insights

Key Points:

- Use `GROUP BY` with SSN to uniquely identify customers.
- Avoid over-reliance on `DISTINCT`, as it may not resolve all duplicate issues.
- Self-joins are generally more efficient than subquery-based solutions.

Task 2: Number of Credit Cards per Customer

Problem Statement:

- Find how many credit cards each customer owns.
- Print the customer's Social Security Number (SSN) and the count of credit cards.
- Include customers who own no credit cards and print zero for them.

Correct Query for Task 2

Solution Using `LEFT OUTER JOIN`:

```sql
SELECT c.ssn, COUNT(cc.number) AS card_count
FROM customers c
LEFT OUTER JOIN credit_cards cc
ON c.ssn = cc.ssn
GROUP BY c.ssn;
```

Explanation:

- `LEFT OUTER JOIN` ensures all customers are included, even if they have no credit cards.
- `COUNT(cc.number)` counts the number of credit cards owned by each customer.
- `GROUP BY c.ssn` groups results by the customer's unique SSN.

Incorrect Query for Task 2

Example of a Wrong Query:

```
SELECT cc.ssn, COUNT(*)
FROM credit_cards cc
GROUP BY cc.ssn;
```

Why It's Wrong:

- Customers without credit cards are excluded because it uses `credit_cards` as the base table.
- It doesn't account for customers who own no credit cards.

Key Takeaway: Always use a `LEFT OUTER JOIN` to ensure the inclusion of all customers.

Task 3: Largest Transaction per Card Type

Problem Statement:

- Find the transaction identifier of the transactions with the largest amount for each type of credit card.
- Use aggregate queries to identify the maximum transaction amount per card type.

Correct Query for Task 3

Solution Using Aggregate Query:

```sql
SELECT t1.identifier
FROM transactions t1
JOIN credit_cards cc1 ON t1.number = cc1.number
WHERE (cc1.type, t1.amount) IN (
    SELECT cc2.type, MAX(t2.amount)
    FROM transactions t2
    JOIN credit_cards cc2 ON t2.number = cc2.number
    GROUP BY cc2.type
);
```

Explanation:

- The inner query finds the maximum amount for each credit card type.
- The outer query retrieves the transaction identifiers matching those maximum values.

Slower Query in Task 3

Using Subqueries with `ALL`:

```sql
SELECT t1.identifier
FROM transactions t1
JOIN credit_cards cc1 ON t1.number = cc1.number
WHERE t1.amount = ALL (
    SELECT MAX(t2.amount)
    FROM transactions t2
    JOIN credit_cards cc2 ON t2.number = cc2.number
    WHERE cc1.type = cc2.type
);
```

Performance Considerations:

- This approach is slower because it evaluates the `MAX` function multiple times for each type.

Slower Query in Task 3

Example of a Slow Query:

```sql
SELECT t1.identifier
FROM transactions t1
JOIN credit_cards cc1 ON t1.number = cc1.number
WHERE EXISTS (
    SELECT MAX(t2.amount)
    FROM transactions t2
    JOIN credit_cards cc2 ON t2.number = cc2.number
    WHERE cc2.type = cc1.type
    HAVING t1.amount = MAX(t2.amount)
);
```

Why It's Suboptimal:

- Uses `EXISTS` with `HAVING`, which results in repeated evaluations for each row.
- Significantly slower on large datasets.

Task 4: Largest Transactions Without Aggregates

Problem Statement:

- Print the transaction identifiers of the transactions with the largest amount for each type of credit card.
- Do not use aggregate functions (e.g., `MAX`, `GROUP BY`).

Correct Query for Task 4

Using `ALL` for Comparison:

```sql
SELECT t1.identifier
FROM transactions t1
JOIN credit_cards cc1 ON t1.number = cc1.number
WHERE t1.amount >= ALL (
    SELECT t2.amount
    FROM transactions t2
    JOIN credit_cards cc2 ON t2.number = cc2.number
    WHERE cc2.type = cc1.type
);
```

Explanation:

- `WHERE t1.amount >= ALL (...)` : Ensures `t1.amount` is the largest among all transactions of the same card type.
- The subquery filters transactions by matching credit card types.
- This avoids using aggregate functions like `MAX`.

Key Takeaways and Submission Reminder

Key Takeaways:

- **Normalization:** Ensure data integrity by applying 1NF, 2NF, 3NF, and BCNF to avoid redundancy and anomalies.
- **SQL Queries:** Learn how to write efficient queries using:
  - `JOIN`, `GROUP BY`, and subqueries.
  - Techniques to ensure uniqueness, like `DISTINCT`, `GROUP BY`, and using primary keys.
- **Performance Considerations:** Compare and choose between self-joins, subqueries, and aggregate functions for efficiency.
- **Validation:** Always test queries with edge cases to ensure accuracy.

Submission Reminder:

- Submit a Jupyter Notebook containing:
  - Python code to generate the ER diagram (e.g., using `graphviz`).
  - Outputs to all tasks with SQL queries.

Thank You!