INFORMATION SYSTEMS
ASSIGNMENT 1

Submission by:
Ayça Avcı [S4505972]
Deepshi Garg [S4199456]

***Note :** Scripts are provided according to their compatibility with PostgreSQL 13 for task 1 and PostgreSQL 12 for task 2, task 3 and task 4.

TASK 1 :

- **1NF:**
  - Using the below script, we obtained 1NF. With 1NF, in each row and column, there are only atomic values, no repeating groups exist. *customer_id* is the primary key for the purchases table.
  - **purchases table:**

```
CREATE TABLE purchases
(customer_id integer PRIMARY KEY NOT NULL,
 customer_name text,
 customer_address text,
 customer_phone text,
 supplier_id integer,
 supplier_name text,
 product_code integer,
 product_title text,
 purchase_date date,
 sales_price float);
```

- **2NF:**
  - In 2NF, every nonkey attribute needs the full primary key for unique identification. Hence, *customer_details* and *supplier_details* tables are created in addition to the *purchases* table.
  - This allows a customer and a supplier to be a part of the DB, even though a purchase between them may not happen.
  - Both *ids* inside the *customer_details* and *supplier_details* tables are primary keys.
  - *customer_id* and *supplier_id* inside the *purchases* table are foreign keys that link *customer_details* and *supplier_details* tables to *purchases* tables respectively.
  - **customer_details table:**

```
CREATE TABLE customer_details
(id integer PRIMARY KEY NOT NULL,
 name text,
 address text,
 phone text);
```

- **supplier_details table:**

```
CREATE TABLE supplier_details
(id integer PRIMARY KEY NOT NULL,
 name text);
```

- **purchases table:**

```
CREATE TABLE purchases
(customer_id integer REFERENCES customer_details (id),
 supplier_id integer REFERENCES supplier_details (id),
 product_code integer,
 product_title text,
 purchase_date date,
 sales_price float);
```

- **3NF:**
  - In 3NF, relation must not have transitive functional dependencies. To achieve this, a *product_details* table is created.
  - This allows a product to be a part of the database even though it may not be purchased by a customer.
  - In the *product_details* table, *code* is the primary key and *supplier_id* is the foreign key that links *supplier_details* table to the *product_details* table.
  - Different from 2NF, the *purchases* table contains *product_code* which is a foreign key that links *product_details* table to *purchases* table.

  - **customer_details table:**

```
CREATE TABLE customer_details
(id integer PRIMARY KEY NOT NULL,
 name text,
 address text,
 phone text);
```

- **supplier_details table:**

```
CREATE TABLE supplier_details
(id integer PRIMARY KEY NOT NULL,
 name text);
```

- **product_details table:**

```
CREATE TABLE product_details
(supplier_id integer REFERENCES supplier_details (id),
 code integer PRIMARY KEY NOT NULL,
 title text);
```

- **purchases table:**

```
CREATE TABLE purchases
(customer_id integer REFERENCES customer_details (id),
 supplier_id integer REFERENCES supplier_details (id),
 product_code integer REFERENCES product_details (code),
 purchase_date date,
 sales_price float);
```

- **BCNF:**
  - In BCNF, for any dependency A -> B, A should be the super key. To achieve this, *entry_id* is created inside the *product_details* table as a primary key and convert *supplier_id* to a foreign key that links *supplier_details* table to purchases table. Now, to reach supplier information can be reached through the *product_details* table only instead of the *purchases* table.
  - To connect *product_details* table to the *purchases* table, *product_entry_id* is added as a foreign key that links *product_details* table to *purchases* table.
  - The *entry_id* being the primary key in *product_details* table also allows for a single product to be supplied by multiple suppliers.
  - The reason why *sales_price* is not included in a *product_details* table is because we want to apply a discount for every 10th purchase of a customer. This means, we want it to change only for one particular purchase made by a particular customer.
  - **customer_details table:**

```
CREATE TABLE customer_details
```

```
(id integer PRIMARY KEY NOT NULL,
 name text,
 address text,
 phone text);
```

- **supplier_details table:**

```
CREATE TABLE supplier_details
(id integer PRIMARY KEY NOT NULL,
 name text);
```

- **product_details table:**

```
CREATE TABLE product_details
(entry_id SERIAL PRIMARY KEY NOT NULL,
 supplier_id integer REFERENCES supplier_details (id),
 code integer,
 title text);
```

- **purchases table:**

```
CREATE TABLE purchases
(purchase_id SERIAL PRIMARY KEY NOT NULL,
 customer_id integer REFERENCES customer_details (id),
 product_entry_id integer REFERENCES product_details (entry_id),
 purchase_date date,
 sales_price float);
```

TASK 2 :

- Following function is created to convert the value of *name* to uppercase

```
CREATE OR REPLACE FUNCTION uppercase_name() RETURNS trigger AS
$uppercase_name$
BEGIN
    NEW.name = UPPER(NEW.name);
    RETURN NEW;
```

```
END;
$uppercase_name$ LANGUAGE plpgsql;
```

- We now add a trigger over the *customer_details* table and the *supplier_details* table to invoke the above function before every Insert or Update operation

```
CREATE TRIGGER uppercase_customer_name BEFORE INSERT OR UPDATE ON
customer_details
    FOR EACH ROW EXECUTE FUNCTION uppercase_name();
```

```
CREATE TRIGGER uppercase_supplier_name BEFORE INSERT OR UPDATE ON
supplier_details
    FOR EACH ROW EXECUTE FUNCTION uppercase_name();
```

TASK 3 :

- We add the following function to return error if the value of *sales_price* is less than or equal to zero

```
CREATE OR REPLACE FUNCTION validate_selling_price() RETURNS trigger AS
$validate_selling_price$
BEGIN
    IF NEW.sales_price <= 0 THEN
        RAISE EXCEPTION 'selling price should be greater than 0';
    END IF;
    RETURN NEW;
END;
$validate_selling_price$ LANGUAGE plpgsql;
```

- Now, we create the following trigger on *purchases* table to invoke the above function before every insert or update event

```
CREATE TRIGGER validate_selling_price BEFORE INSERT OR UPDATE ON purchases
      FOR EACH ROW EXECUTE FUNCTION validate_selling_price();
```

TASK 4 :

- We first create the following function:
    - It retrieves the total number of purchases made by the current customer in variable *num_purchases.*
    - If adding the current purchase to *num_purchases* makes it a multiple of 10, it means that the current purchase is a "10th" purchase for this customer.
    - Thus, if it is not the first purchase, and a "10th" purchase, the *sales_price* is discounted to to 90% value for this purchase for this customer.

```
CREATE OR REPLACE FUNCTION apply_discount() RETURNS trigger AS
$apply_discount$
      DECLARE
            num_purchases integer := (SELECT count(*) FROM purchases WHERE
customer_id = NEW.customer_id);
            mod_val integer := (SELECT mod(num_purchases+1, 10));
      BEGIN
            IF num_purchases > 0 AND mod_val = 0 THEN
                  NEW.sales_price = 0.9* NEW.sales_price;
            END IF;
            RETURN NEW;
      END;
$apply_discount$ LANGUAGE plpgsql;
```

- Following trigger is added on the *purchases* table to invoke the above function before every insert operation

```
CREATE TRIGGER apply_discount BEFORE INSERT ON purchases
      FOR EACH ROW EXECUTE FUNCTION apply_discount();
```