DBMS.2

END TERM PROJECT

PROJECT NAME: PET PRODUCTS

PRESENTED TO: AZAMAT SEREK

PRESENTED BY:	BIRZHAN BAIMURZA	210107007
	BEKBOLAT BALGYNBEK	210103205
	NURGABYL ZHANERKE	210107019

PROJECT DESCRIPTION:

- The goal of this project is to develop a database management system (DBMS) that can help in storing and manipulating data in the environment of pet products. The goal is to provide online sales, provide customers with prompt and accurate product information, effectively manage consumer and transaction data, and track inventory, sales transactions, and financial transfers.

- By implementing the proposed system, the pet products will be able to effectively store and manage data, including stock levels, pricing information, and availability. Furthermore, the system will be able to incorporate customer information, such as personal details, purchase history, and credit card information, to facilitate smooth sales transactions. The system will also collect comprehensive data on sales, such as revenue, cost of goods sold, and profits, and generate reports that offer valuable insights into the pet product's performance.

- The system will also have the capabilities to manage online transactions, track the delivery and keep records of the status of payment and delivery. These features ensure that customers promptly receive updated information about their orders and that pet products keeps accurate records of all transactions.

- The intended DBMS is custom-made to cater to the requirements of small and medium-sized businesses, offering them a complete solution for data storage and manipulation, transaction processing, and financial management. Implementing this system would enable pet products to simplify their operations, minimize errors, and improve the customer experience.

PROJECT STRUCTURE:

This database is designed for an online pet store, and it consists of 12 tables that are interrelated through different types of relationships. Here is an overview of the relationships between the tables:

- The Customer table stores information about each customer who makes purchases on the online store. The Customer_ID column is the primary key of this table.

- The User Account table stores information about customer account details, such as username, email, and password. It is related to the Customer table through the Customer_ID foreign key.

- The Addresses table stores information about physical addresses that can be associated with one or more users. The Address_ID column is the primary key of this table.

- The User Addresses table represents a many-to-many relationship between users and addresses. It is related to both the Customer and Addresses tables through foreign keys that together form the composite primary key.

- The Payment Type table stores information about different types of payment methods available to customers. The PaymentType_ID column is the primary key of this table.

- The Payment table stores information about payments made by customers for their orders. It is related to the Customer and Payment Type tables through foreign keys and has a primary key of Payment_ID.

- The Category table stores information about the different categories of products sold on the online store. It has a primary key of Category_ID and is related to itself through the parent Category_ID foreign key.

- The Supplier table stores information about the suppliers of products sold on the online store. It is related to the Addresses table through the AddressID foreign key and has a primary key of Supplier_ID.

- The Product table stores information about the products sold on the online store. It is related to the Category and Supplier tables through foreign keys and has a primary key of ProductID.

- The Order table stores information about the orders placed by customers. It is related to the Customer and Payment tables through foreign keys and has a primary key of OrderID.

- The Order Item table stores information about the individual products that are part of each order. It is related to the Order and Product tables through foreign keys and has a primary key of OrderItemID.

- The Shipment table stores information about the shipments of products to customers. It is related to the Order and Addresses tables through foreign keys and has a primary key of ShipmentID.

Normalization:

The database is designed to satisfy the normal forms of 1NF, 2NF, and 3NF. Each table has a primary key, and each column in the table has a single atomic value, which satisfies the requirements of 1NF. Each non-key column in the table is fully functionally dependent on the primary key, which satisfies the requirements of 2NF. And finally, there is no transitive dependency between non-key columns in the table, which satisfies the requirements of 3NF. The normalization of the database ensures data integrity and efficient data management.

TABLES: USER_ACCOUNT, USER_ADDRESS, ADDRESS, CUSTOMER, PRODUCT, CATEGORY, ORDERING, ORDER_ITEM, SHIPMENT, SUPPLIER, PAYMENT, PAYMENT_TYPE

USER_ACCOUNT:

```
  CREATE TABLE IF NOT EXISTS user_account (
    username VARCHAR(255) PRIMARY KEY,
    customer_id INTEGER NOT NULL,
    email VARCHAR(255) NOT NULL,
    password VARCHAR (32) NOT NULL,
    FOREIGN KEY (customer_id) REFERENCES Customer (id)
);
```

USER_ADDRESS:

```
    CREATE TABLE IF NOT EXISTS user_addresses (
  customer_id INTEGER NOT NULL,
  addrs_id INTEGER NOT NULL,
  PRIMARY KEY (customer_id, addrs_id),
  FOREIGN KEY (customer_id) REFERENCES Customer (id),
  FOREIGN KEY (addrs_id) REFERENCES Address (id)
);
```

ADDRESS:

```
  CREATE TABLE IF NOT EXISTS Address (
  id INTEGER PRIMARY KEY,
  unit_num INTEGER NOT NULL,
  street VARCHAR(255) NOT NULL,
  city VARCHAR(255) NOT NULL,
  region VARCHAR(255) NOT NULL,
  postal_code VARCHAR(255) NOT NULL
);
```

CUSTOMER:

```
   CREATE TABLE IF NOT EXISTS Customer (
    id INTEGER PRIMARY KEY,
    Fname VARCHAR(255) NOT NULL,
    Lname VARCHAR(255) NOT NULL,
    tel_num VARCHAR(20) NOT NULL
 );
```

PRODUCT:

```
  CREATE TABLE IF NOT EXISTS Product (
    id INTEGER PRIMARY KEY,
    category_id INTEGER NOT NULL,
    supplier_id INTEGER NOT NULL,
    name VARCHAR(255) NOT NULL,
    description TEXT NOT NULL,
    price_per_one INTEGER NOT NULL,
    FOREIGN KEY (category_id) REFERENCES Category (id),
    FOREIGN KEY (supplier_id) REFERENCES Supplier (id)
);
```

CATEGORY:

```
  CREATE TABLE IF NOT EXISTS Category (
    id INTEGER PRIMARY KEY,
    subcategory_id INTEGER,
    name VARCHAR(100) NOT NULL,
    FOREIGN KEY (subcategory_id) REFERENCES Category (id)
);
```

ORDERING:

```
  CREATE TABLE IF NOT EXISTS Ordering (
    id INTEGER PRIMARY KEY,
    customer_id INTEGER NOT NULL,
    payment_id INTEGER NOT NULL,
    order_date TIMESTAMP NOT NULL,
    total_price INTEGER NOT NULL,
    status VARCHAR(50) NOT NULL,
    FOREIGN KEY (customer_id) REFERENCES Customer (id),
    FOREIGN KEY (payment_id) REFERENCES Payment (id)
);
```

ORDER_ITEM:

```
  CREATE TABLE IF NOT EXISTS order_item (
    id INTEGER PRIMARY KEY,
    order_id INTEGER NOT NULL,
    product_id INTEGER NOT NULL,
    quantity INTEGER NOT NULL,
    price INTEGER NOT NULL,
    FOREIGN KEY (order_id) REFERENCES Ordering (id),
    FOREIGN KEY (product_id) REFERENCES Product (id)
);
```

SHIPMENT:

```
  CREATE TABLE IF NOT EXISTS Shipment (
    id INTEGER PRIMARY KEY,
    addrs_id INTEGER NOT NULL,
    order_id INTEGER NOT NULL,
    shipment_date TIMESTAMP NOT NULL,
    tracking_num INTEGER NOT NULL,
    FOREIGN KEY (addrs_id) REFERENCES Address (id),
    FOREIGN KEY (order_id) REFERENCES Ordering (id)
);
```

SUPPLIER:

```
  CREATE TABLE IF NOT EXISTS Supplier (
  id INTEGER PRIMARY KEY,
  addrs_id INTEGER NOT NULL,
  name VARCHAR(255) NOT NULL,
  email VARCHAR(255) NOT NULL,
  FOREIGN KEY (addrs_id) REFERENCES Address (id)
);
```

PAYMENT:

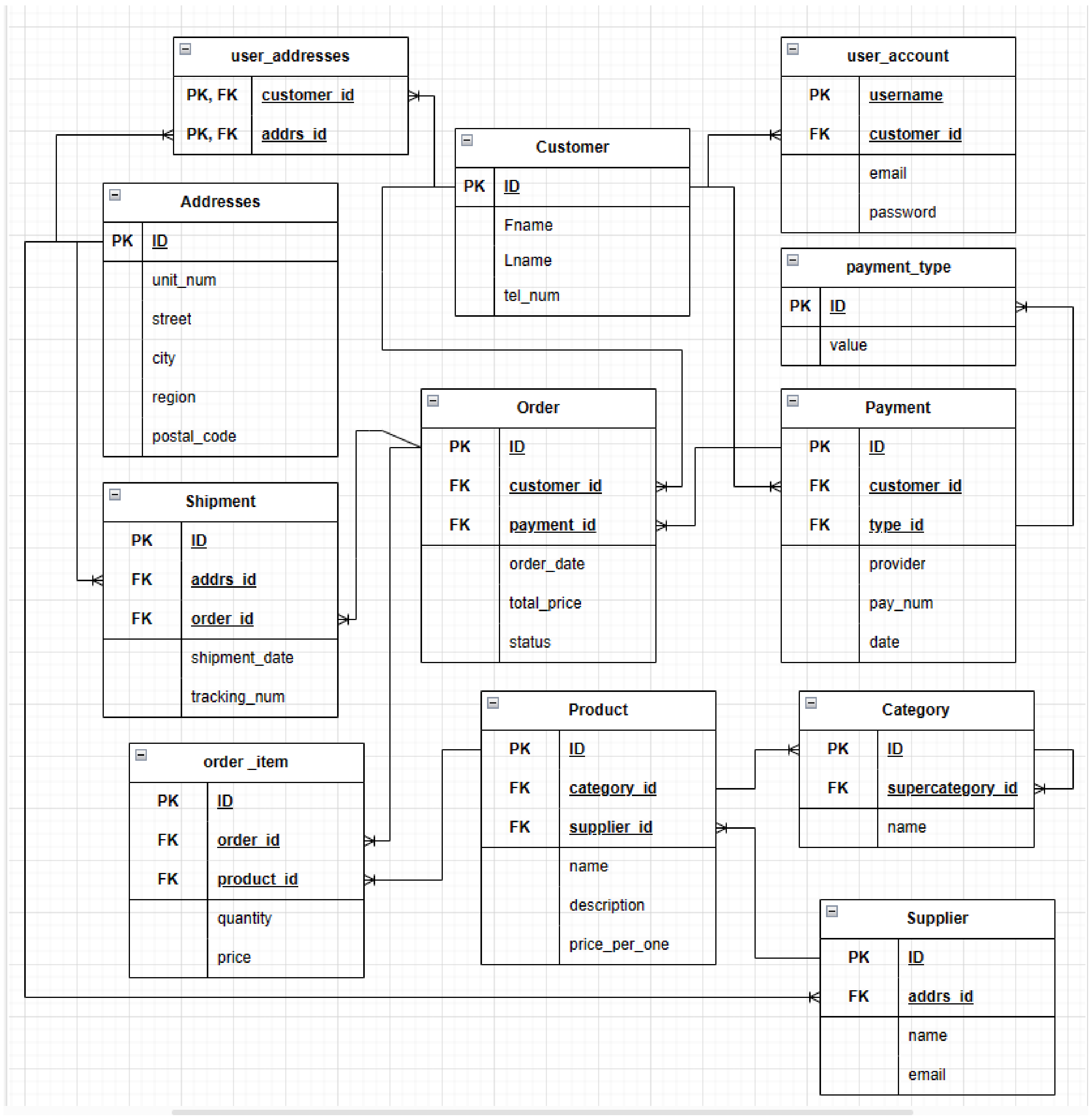
```
  CREATE TABLE IF NOT EXISTS Payment (
    id INTEGER PRIMARY KEY,
    customer_id INTEGER NOT NULL,
    type_id INTEGER NOT NULL,
    provider VARCHAR(50) NOT NULL,
    pay_num INTEGER NOT NULL,
    date DATE NOT NULL,
    FOREIGN KEY (customer_id) REFERENCES Customer (id),
    FOREIGN KEY (type_id) REFERENCES payment_type (id)
);
```

PAYMENT_TYPE:

```
  CREATE TABLE IF NOT EXISTS payment_type (
    id INTEGER PRIMARY KEY,
    value VARCHAR(255) NOT NULL
);
```

ENTITY RELATIONSHIP DIAGRAM (ERD):

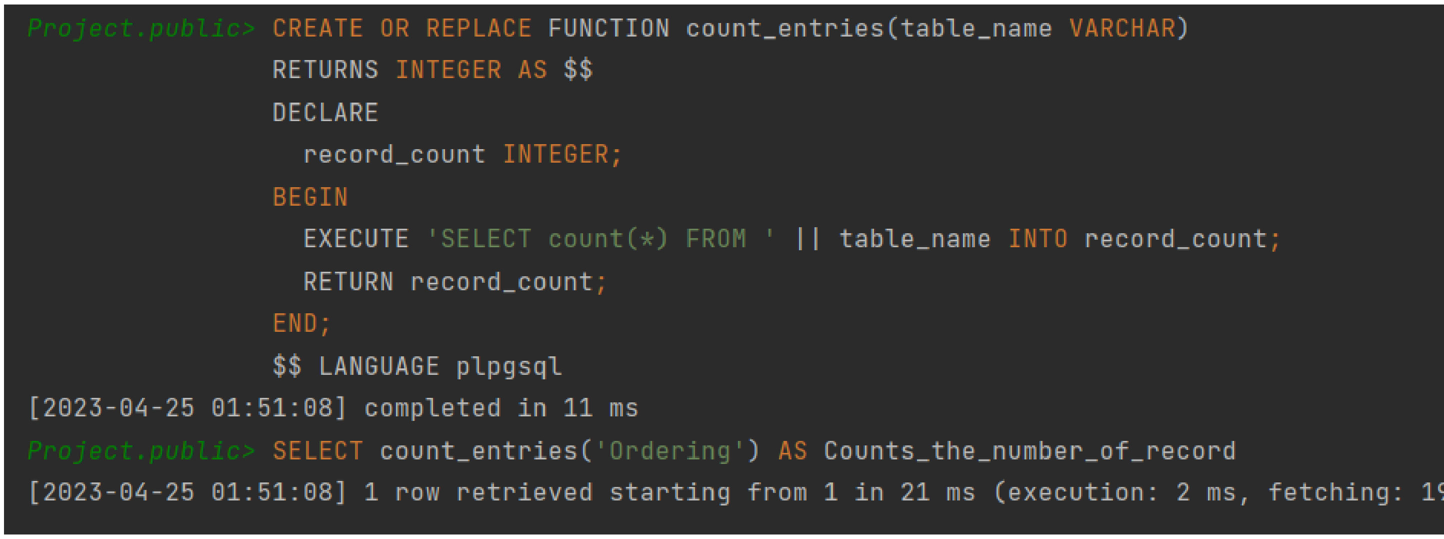
HERE IS THE LINK TO (ERD)



PROCEDURES:

1) PROCEDURE

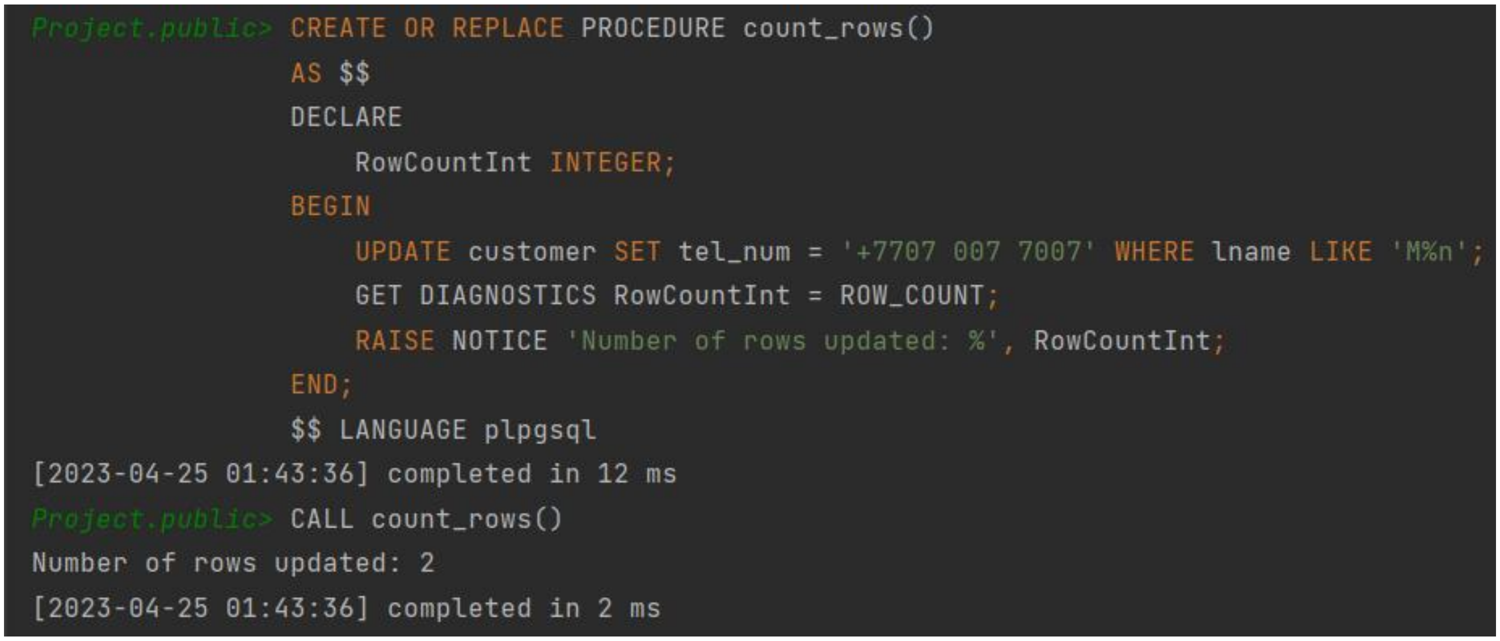
```
CREATE OR REPLACE PROCEDURE get_category_sales()
AS $$
DECLARE
    category_name varchar(255);
    total_sales int;
    total_revenue float;
BEGIN
    FOR category_name, total_sales, total_revenue IN
        SELECT c.name, SUM(oi.quantity) as total_sales, SUM(oi.quantity *
p.price_per_one) as total_revenue
        FROM category c
        JOIN product p ON c.id = p.category_id
        JOIN order_item oi ON p.id = oi.product_id
        GROUP BY c.id
    LOOP
        RAISE NOTICE 'Category Name: %, Total Sales: %, Total Revenue: %',
category_name, total_sales, total_revenue;
    END LOOP;
END;
$$ LANGUAGE plpgsql;
CALL get_category_sales();
```



```
Project.public> CREATE OR REPLACE FUNCTION count_entries(table_name VARCHAR)
                RETURNS INTEGER AS $$
                DECLARE
                  record_count INTEGER;
                BEGIN
                  EXECUTE 'SELECT count(*) FROM ' || table_name INTO record_count;
                  RETURN record_count;
                END;
                $$ LANGUAGE plpgsql
[2023-04-25 01:51:08] completed in 11 ms
Project.public> SELECT count_entries('Ordering') AS Counts_the_number_of_record
[2023-04-25 01:51:08] 1 row retrieved starting from 1 in 21 ms (execution: 2 ms, fetching: 19
```

2) **PROCEDURE**

```
CREATE OR REPLACE PROCEDURE count_rows()
AS $$
DECLARE
    RowCountInt INTEGER;
BEGIN
    UPDATE customer SET tel_num = '+7707 007 7007' WHERE lname LIKE 'M%n';
    GET DIAGNOSTICS RowCountInt = ROW_COUNT;
    RAISE NOTICE 'Number of rows updated: %', RowCountInt;
END;
$$ LANGUAGE plpgsql;
CALL count_rows();
```



```
Project.public> CREATE OR REPLACE PROCEDURE count_rows()
                AS $$
                DECLARE
                    RowCountInt INTEGER;
                BEGIN
                    UPDATE customer SET tel_num = '+7707 007 7007' WHERE lname LIKE 'M%n';
                    GET DIAGNOSTICS RowCountInt = ROW_COUNT;
                    RAISE NOTICE 'Number of rows updated: %', RowCountInt;
                END;
                $$ LANGUAGE plpgsql
[2023-04-25 01:43:36] completed in 12 ms
Project.public> CALL count_rows()
Number of rows updated: 2
[2023-04-25 01:43:36] completed in 2 ms
```

This code creates a procedure called "count_rows". When executed, it will update the phone number of all customers with a last name that starts with "M" and ends with "n" to "+7707 007 7007".

The procedure uses the UPDATE statement with the WHERE clause that filters out the customers with last names that match the specified pattern. It then uses the GET

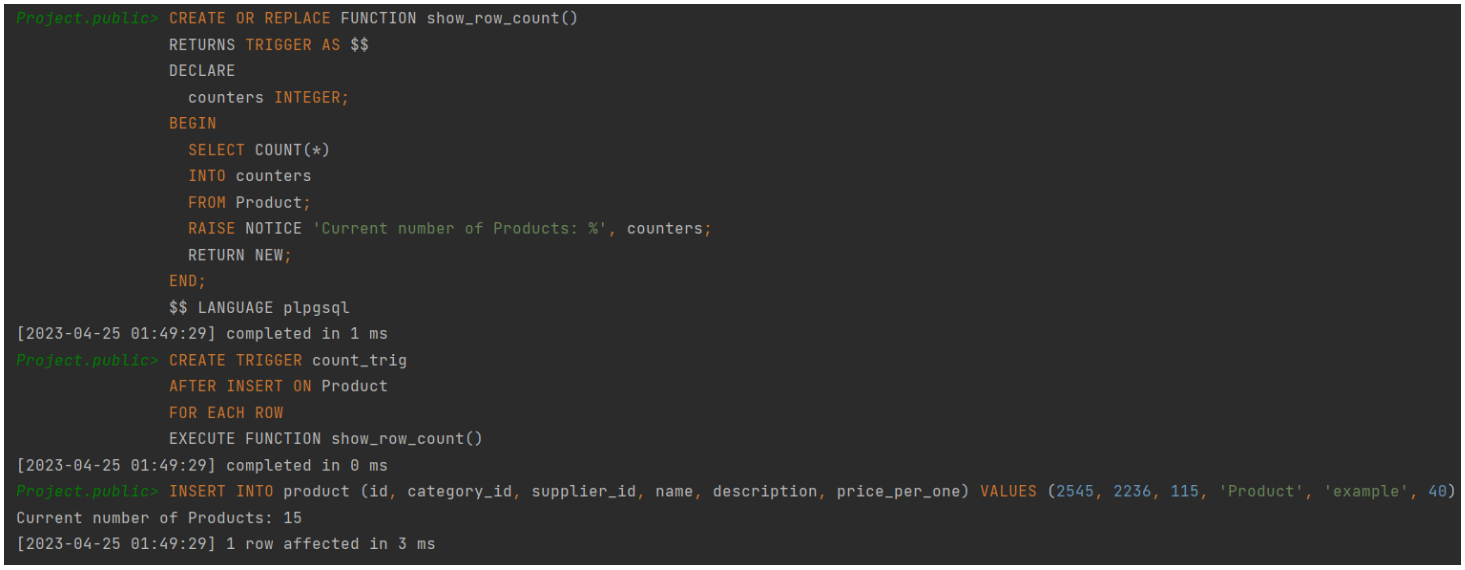
DIAGNOSTICS statement to retrieve the number of rows affected by the UPDATE statement. It finally uses the RAISE NOTICE statement to print out the number of rows that were updated.

To execute the procedure, it can be called using the "CALL" statement followed by the name of the procedure ("count_rows()").

TRIGGER:

```
CREATE OR REPLACE FUNCTION show_row_count()
RETURNS TRIGGER AS $$
DECLARE
  counters INTEGER;
BEGIN
  SELECT COUNT(*)
  INTO counters
  FROM Product;
  RAISE NOTICE 'Current number of Products: %', counters;
  RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER count_trig
AFTER INSERT ON Product
FOR EACH ROW
EXECUTE FUNCTION show_row_count();
INSERT INTO product (id, category_id, supplier_id, name, description, price_per_one)
VALUES (2545, 2236, 115, 'Product', 'example', 40);
```



```
Project.public> CREATE OR REPLACE FUNCTION show_row_count()
                RETURNS TRIGGER AS $$
                DECLARE
                  counters INTEGER;
                BEGIN
                  SELECT COUNT(*)
                  INTO counters
                  FROM Product;
                  RAISE NOTICE 'Current number of Products: %', counters;
                  RETURN NEW;
                END;
                $$ LANGUAGE plpgsql
[2023-04-25 01:49:29] completed in 1 ms
Project.public> CREATE TRIGGER count_trig
                AFTER INSERT ON Product
                FOR EACH ROW
                EXECUTE FUNCTION show_row_count()
[2023-04-25 01:49:29] completed in 0 ms
Project.public> INSERT INTO product (id, category_id, supplier_id, name, description, price_per_one) VALUES (2545, 2236, 115, 'Product', 'example', 40)
Current number of Products: 15
[2023-04-25 01:49:29] 1 row affected in 3 ms
```

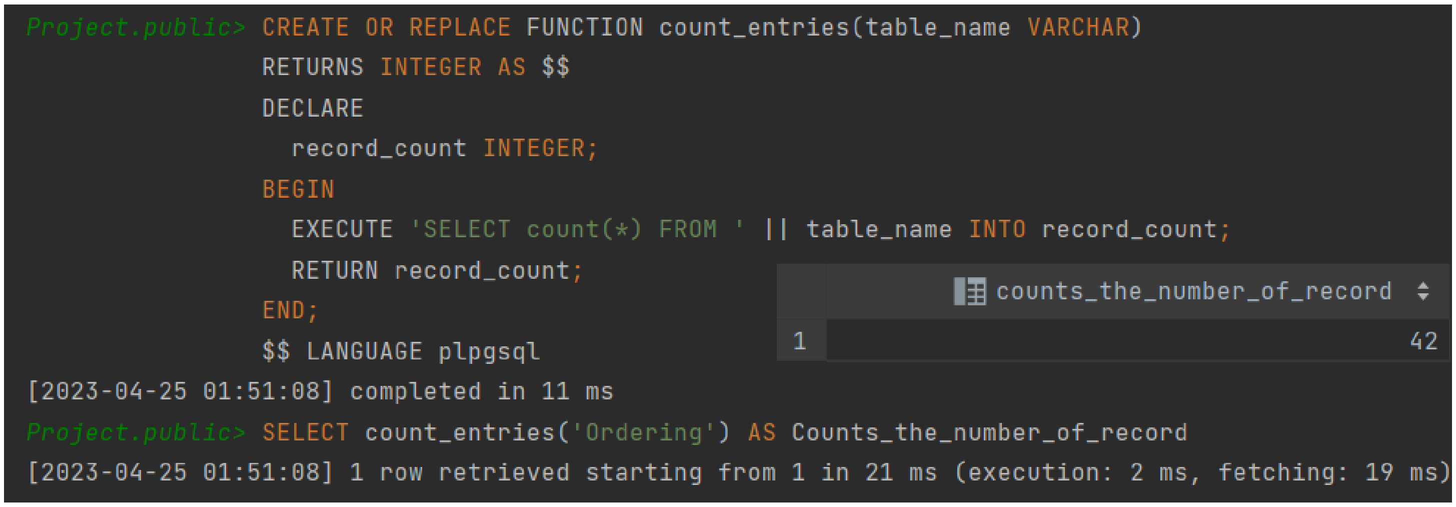
This is a SQL code that creates a function and a trigger for a database table called "Product".

The trigger is activated every time a new row is inserted into the "Product" table. It executes the "show_row_count()" function, which counts the number of rows in the "Product" table and stores it in the "counters" variable. Then, it prints out the current number of rows as a notice message.

FUNCTION:

```
CREATE OR REPLACE FUNCTION count_entries(table_name VARCHAR)
RETURNS INTEGER AS $$
DECLARE
  record_count INTEGER;
BEGIN
  EXECUTE 'SELECT count(*) FROM ' || table_name INTO record_count;
  RETURN record_count;
END;
$$ LANGUAGE plpgsql;

SELECT count_entries('Ordering') AS Counts_the_number_of_record;
```



```
Project.public> CREATE OR REPLACE FUNCTION count_entries(table_name VARCHAR)
                RETURNS INTEGER AS $$
                DECLARE
                  record_count INTEGER;
                BEGIN
                  EXECUTE 'SELECT count(*) FROM ' || table_name INTO record_count;
                  RETURN record_count;
                END;
                $$ LANGUAGE plpgsql
[2023-04-25 01:51:08] completed in 11 ms
Project.public> SELECT count_entries('Ordering') AS Counts_the_number_of_record
[2023-04-25 01:51:08] 1 row retrieved starting from 1 in 21 ms (execution: 2 ms, fetching: 19 ms)
```

	counts_the_number_of_record
1	42

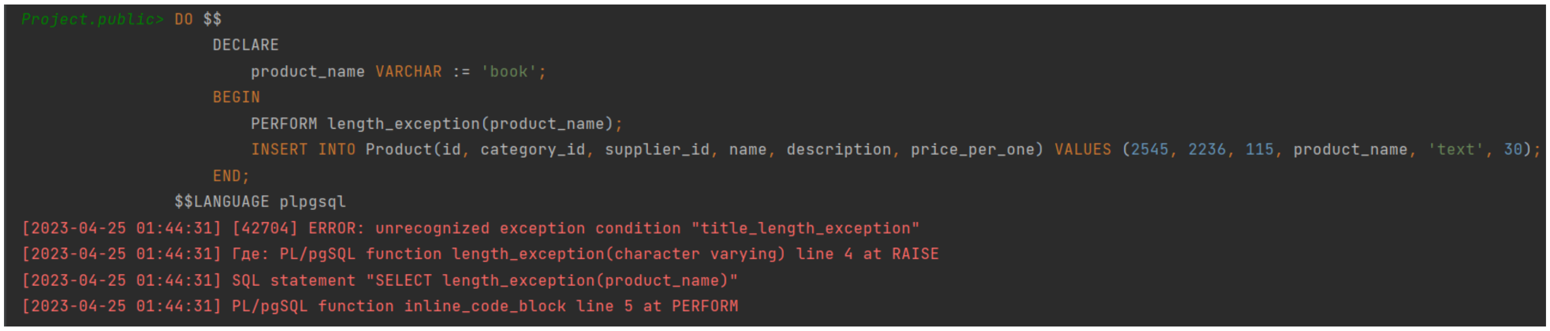
This code defines a function called "count_entries" that takes a table name as input and returns the number of records in that table. The function uses dynamic SQL to execute a SELECT statement with the table name as a parameter. The record count is then stored in the "record_count" variable. Finally, the function returns the value of "record_count".

To use the function, the code calls it with the table name 'Ordering' as the input parameter and then calls the result "Counts_the_number_of_record". This will return a count of the number of records in the 'Ordering' table.

EXCEPTION:

```
CREATE OR REPLACE FUNCTION length_exception(title VARCHAR)
RETURNS VOID AS $$
BEGIN
    IF LENGTH(title) < 5 THEN
        RAISE EXCEPTION 'Title must be at least 5 characters long.'
        USING errcode = 'title_length_exception';
    END IF;
END;
$$ LANGUAGE plpgsql;

DO $$
    DECLARE
        product_name VARCHAR := 'book';
    BEGIN
        PERFORM length_exception(product_name);
        INSERT INTO Product(id, category_id, supplier_id, name, description,
price_per_one) VALUES (2545, 2236, 115, product_name, 'text', 30);
    END;
$$LANGUAGE plpgsql;
```



```
Project.public> DO $$
                    DECLARE
                        product_name VARCHAR := 'book';
                    BEGIN
                        PERFORM length_exception(product_name);
                        INSERT INTO Product(id, category_id, supplier_id, name, description, price_per_one) VALUES (2545, 2236, 115, product_name, 'text', 30);
                    END;
                $$LANGUAGE plpgsql
[2023-04-25 01:44:31] [42704] ERROR: unrecognized exception condition "title_length_exception"
[2023-04-25 01:44:31] Где: PL/pgSQL function length_exception(character varying) line 4 at RAISE
[2023-04-25 01:44:31] SQL statement "SELECT length_exception(product_name)"
[2023-04-25 01:44:31] PL/pgSQL function inline_code_block line 5 at PERFORM
```

This code creates a function called "length_exception" that takes a "title" argument and checks if the length of the title is less than 5 characters. If the length is less than 5, it raises an exception with a custom error message and error code.

The code then executes a "DO" block that declares a variable called "product_name" with a value of 'book'. It then performs the "length_exception" function on "product_name". If the length of "product_name" is less than 5 characters, the function raises an exception.