DBMS Project

Course Name: Database Management Systems 2 (INF 305)

Project Title: Online Cosmetics Store

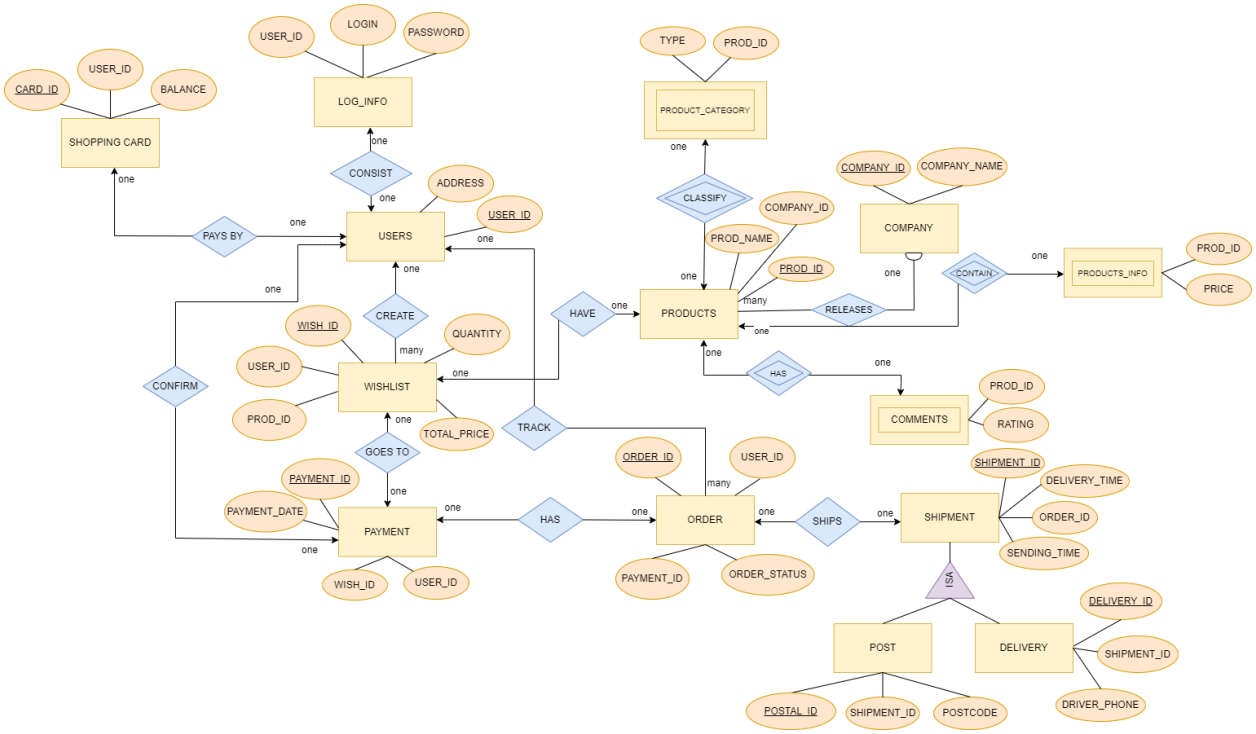
Group Members: Berdikozha Bayan (210103228)
Bagasharova Aidana (210103405)
Azhen Sayat (210107068)
Sultan Zhanylkhan (220103398)
Ayaulym Aiyp (210103102)

I. Introduction.

This project is an user based shopping system for a cosmetic shop. With the rise of the internet and the increasing use of mobile devices, more and more people are turning to online shopping for their beauty needs. In this project, we will be developing an online shopping system for cosmetics, which will utilize a database management system (DBMS) to store and manage the product information and customer data.

The online shopping system for cosmetics will be a web-based application that allows customers to browse and purchase a wide range of cosmetics products from the comfort of their own home. The system will have a user-friendly interface that will enable customers to search for products, view product details, add products to their wishlist and order online. Overall, this E-commerce system will create a bridge between users and the online cosmetics store, making shopping for beauty products a hassle-free and enjoyable experience.

II. Entity Relationship Diagram:



Each company should release at least 1 product. There can not be a product that is not made any company. So, between company and products - referential integrity constraint.

The shopping site has huge number of products. Each product identified by their product_ID and has product_name. Products have comments which consist product_id and rating. Products are classifies by categories.

There are many users who can create wishlist and list their products. After the user makes his choice, he can confirm the payment online. Every user pay with the help of own shopping_card. After payment, order will be accepted. User can track their order by a unique order_ID. Also order have the order status attribute which consist of numbers from

1 to 4 and each number has its own meanings. (1 – payment is done, please wait; 2 – order pending by user; 3 – order is being delivered; 4 - order received by user.)

Order ships by Shipment. However, order status only with numbers 3 and 4 can go to the Shipment. There are 2 categories of Shipment: Post and Delivery. If the User chooses Delivery, each shipment has their own delivery_id and also information about driver (driver_phone). If the user chooses Post, each post shipment identified by postal_id and with the help of postal code find an address of user.

ERD have 3 weak entity sets (product_category, comments and products_info). These weak entity sets cannot be uniquely identified by their own attributes.

III. Explanation of why the structure follows normal forms:

1. USERS table.
   user_id -> address

1NF requires that each column in a table must contain atomic values, meaning that there should be no repeating groups. In the USERS table, each column only contains a single value, which means it satisfies the 1NF requirement. 2NF requires that each non-key column should be functionally dependent on the primary key. This means that any column that is not part of the primary key should depend on the entire primary key, rather than only part of it. In the USERS table, the user_id column is the primary key, and the address column depends on the entire primary key. There are no other non-key columns in the table, so the 2NF requirement is also satisfied. We can see that "address" is directly dependent on "user_id", the primary key, and there are no other non-key columns in the table. This means that there are no transitive dependencies between non-key columns, and the "user_table" is already in 3NF.

2. LOG_INFO table.
   user_id -> login, password

In the "log_info" table, each column contains only one value, so the table is in 1NF. In the "log_info" table, the primary key is user_ID, which uniquely identifies each user. Both the login and password columns are dependent on the user_ID, so the table is in 2NF. There are no transitive dependencies, meaning that no non-key column is dependent on another non-key column. In the "log_info" table, there are no transitive dependencies, so the table is also in 3NF.

3. SHOPPING_CARD table.
   card_id -> user_id, balance

The table is already in 1NF since each column contains atomic values, and there are no repeating groups. The table is also in 2NF because it has only one candidate key (card_id) which is a unique identifier for each row, and all non-key attributes (user_id, balance) are fully dependent on the candidate key. The table is also in 3NF because there are no transitive dependencies between the attributes. In other words, there is no attribute that is dependent on another non-key attribute.

4. WISHLIST table.
   wish_id -> user_id, prod_id, total_price,quantity .

The table is already in 1NF since each column contains atomic (indivisible) values. The table appears to have a composite primary key (wish_id and user_id), which may suggest that there could be partial dependencies. However, assuming that each wish_id is unique to a particular user (a user can have multiple wishlists), then there are no partial dependencies. In this case, all non-key attributes (prod_id, total_price, quantity) depend on the entire primary key. Therefore, the table is in 2NF. The table is also in 3NF . there is no attribute that is dependent on another non-key attribute.

5. PRODUCTS table.
   prod_id -> company_id, product_name

Table in 1NF, each column contains atomic values. The table has a candidate key (prod_id), which is a unique identifier for each row. Both

non-key attributes (company_id and product_name) are fully dependent on the candidate key. Therefore, the table is in 2NF. The table is also in 3NF because there is no attribute that is dependent on another non-key attribute.

6. PRODUCTS_INFO table.
   prod_id -> price

   We created this table in order to avoid transitive dependence in table Products and put the price attribute in a separate table. This table also in 3NF.

7. PRODUCT_CATEGORY table.
   prod_id -> prod_type

   The table is already in 1NF since each column contains atomic (indivisible) values, and there are no repeating groups. There are no partial dependencies. In this case, all non-key attributes (prod_type) depend on the entire primary key. Therefore, the table is in 2NF and also in 3NF.

8. COMPANY table.
   company_id -> company_name

The table is already in 1NF since each column contains atomic (indivisible) values. The table has a candidate key (company_id), which is a unique identifier for each row. The only non-key attribute (company_name) is fully dependent on the candidate key. Therefore, the table is in 2NF. The table is also in 3NF because there are no transitive dependencies between the attributes. In other words, there is no attribute that is dependent on another non-key attribute.

9. COMMENTS table.
   prod_id -> rating

This table also in 1nf, 2nf and 3nf. Non-key attributes (rating) are fully dependent on the entire primary key. Therefore, the table is in 2NF. There are no transitive dependencies between the attributes, also table in 3NF.

10. PAYMENT table.
    payment_id -> wish_id, payment_date, user_id

Each column contains atomic (indivisible) values, and there are no repeating groups, so table in 1NF. Each payment is uniquely identified by its payment_id and is associated with a single wishlist identified by its wish_id, then there are no partial dependencies. All non-key attributes (payment_date, user_id) depend on the entire primary key. Therefore, the table is in 2NF. The table is also in 3NF because there are no transitive dependencies between the attributes.

11. ORDER table.
    order_id -> payment_id, user_id, order_status.

The table is already in 1NF since there are no repeating groups. The table has a candidate key (order_id), which is a unique identifier for each row. All non-key attributes (payment_id, user_id, order_status) are fully dependent on the candidate key. Therefore, the table is in 2NF. The table is also in 3NF because there are no transitive dependencies between the attributes.

12. SHIPMENT table .
    shipment_id -> order_id, delivery_time,sending_time

The table already in 1nf (single values). The table has a candidate key (shipment_id), which is a unique identifier for each row. All non-key attributes (order_id, delivery_time, sending_time) are fully dependent on the candidate key. Therefore, the table is in 2NF.The table is also in 3NF because there is no attribute that is dependent on another non-key attribute.

13. POST table.
    postal_id -> shipment_id, postcode.

There are no repeating groups, table in 1nf. The table has a candidate key (postal_id), which is a unique identifier for each row. All non-key attributes (shipment_id, postcode) are fully dependent on the candidate key. Therefore, the table is in 2NF.The table is also in 3NF because there are no transitive dependencies between the attributes

14. DELIVERY table.
    delivery_id -> user_id, shipment_id, driver_phone

    There are no repeating groups (1NF). The table has a candidate key (delivery_id), which is a unique identifier for each row. All non-key attributes (user_id, shipment_id, driver_phone) are fully dependent on the candidate key. Therefore, the table is in 2NF.The table is also in 3NF because there are no transitive dependencies.

IV. Explanation and coding part of each item.

- **Procedure**

  This stored procedure retrieves the sum of total_price for each user in the wishlist table (with group by).

- Creating procedure:

```
CREATE OR REPLACE PROCEDURE sumprice AS
BEGIN
 FOR prodt IN (
SELECt user_id, sum(total_price) as pr_sum
FROM wishlist
GROUP BY user_id) LOOP
DBMS_OUTPUT.PUT_LINE('Users id: ' || prodt.user_id || ', Total price:
' ||prodt.pr_sum);
```

```
 END LOOP;
EXCEPTION
WHEN OTHERS THEN
DBMS_OUTPUT.PUT_LINE('Unfortunately, this error occurred: ' ||
SQLERRM);
END;
```

The procedure begins with a BEGIN keyword, which indicates the start of the procedure's executable code. Next, a FOR loop is used to iterate over the results of a SELECT statement. The SELECT statement retrieves the user_id and the sum of the total_price for each user from the wishlist table. Within the FOR loop, the DBMS_OUTPUT.PUT_LINE function is used to display a message for each user_id and its corresponding sum of total_price. This message is constructed using string concatenation (||) and the values retrieved from the prodt cursor.

- Calling the procedure:

Procedure works.

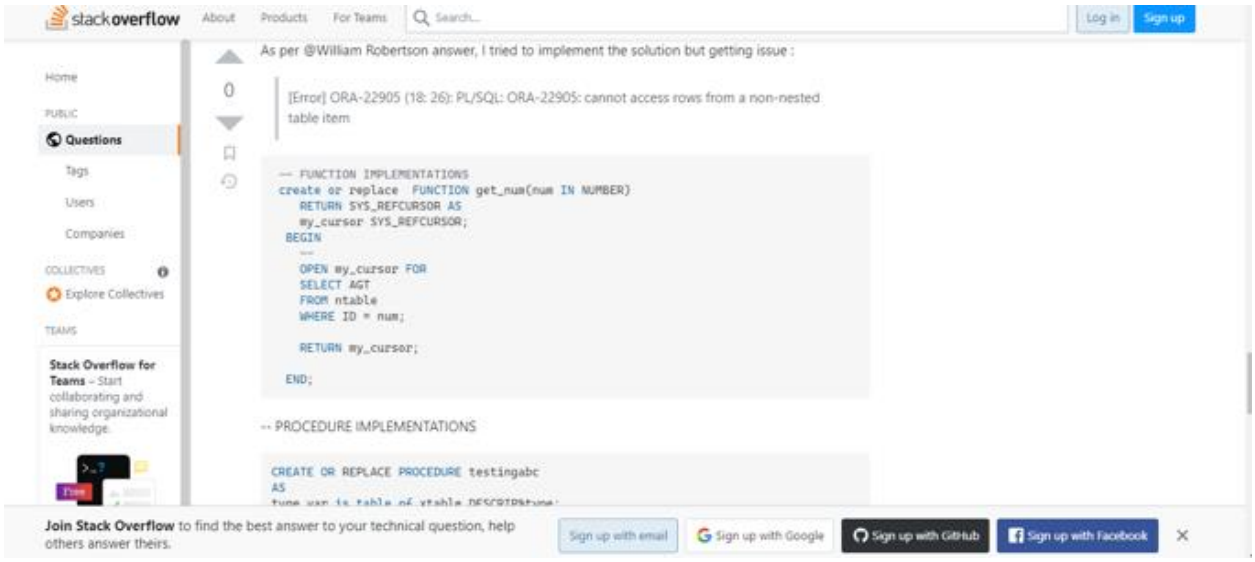
- So, let’s explain the result and take an example user with id 8:

USER_ID	TOTAL_PRICE_8
8	185521

As we can see, total_price is equal to 185521. Procedure works correctly.

- **Function**

  The function num_of_orders_to_shipment counts order_id From Orders Table, which have order_status of 3 or 4 (3 stands for “3 – order is being delivered; 4 - order received by user.”).(Since I needed a Syntax how to add cursor into Function, I searched it from the Internet:)



- *Code:*

```
CREATE OR REPLACE FUNCTION num_of_orders_to_shipment
(c_orders IN
SYS_REFCURSOR)
RETURN NUMBER IS
    c_order_id orders.order_id%TYPE;
    c_order_status orders.order_status%TYPE;
    cnt number:=0;
BEGIN
```

```
    LOOP
    FETCH c_orders INTO c_order_id, c_order_status;
    EXIT WHEN c_orders%NOTFOUND;
    IF c_order_status = 3 or c_order_status = 4 THEN
        cnt:=cnt+1;
    END IF;
    END LOOP;
    CLOSE c_orders;
    RETURN cnt;
END;
/
```

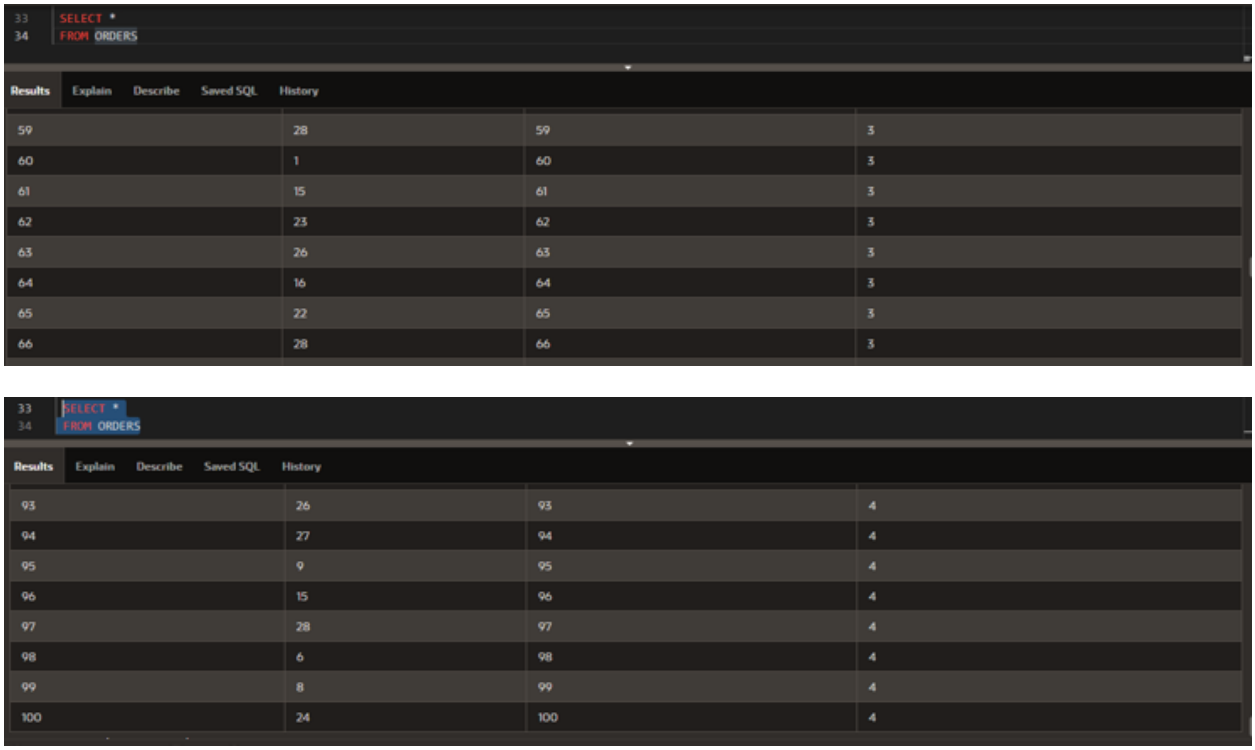
- *Function Call:*

```
DECLARE
    cnt_of_orders_to_shipment number:=0;
    c_orders SYS_REFCURSOR;
BEGIN
    OPEN c_orders FOR
    SELECT order_id, order_status FROM orders;
    cnt_of_orders_to_shipment := num_of_orders_to_shipment(c_orders);
    dbms_output.put_line('Number of orders that can go to shipment is: ' ||
cnt_of_orders_to_shipment);
 END;
 /
```

- *Result:*

- *Explanation of the Result:*

In our “Orders” Table we have orders with status either 3 or 4 starting from order_id 59 to 100. So the total number of them is 42, as function returned.



- **User-Defined Exception**

Requirement: Add user-defined exception which disallows to enter title of item (e.g. book) to be less than 5 characters.

First, we declare variables to assign user input and exception name:

```
DECLARE
  PROD_ID PRODUCTS.PROD_ID%TYPE;
  PROD_NAME PRODUCTS.PROD_NAME%TYPE;
  COMPANY_ID PRODUCTS.COMPANY_ID%TYPE;
  PROD_TYPE PRODUCT_CATEGORY.PROD_TYPE%TYPE;
  invalid_name EXCEPTION;
```

Here we are taking input from user and putting them into our recently declared variables.

Checking if product name has at least 5 characters, if it is false, we raise our recently declared exception:

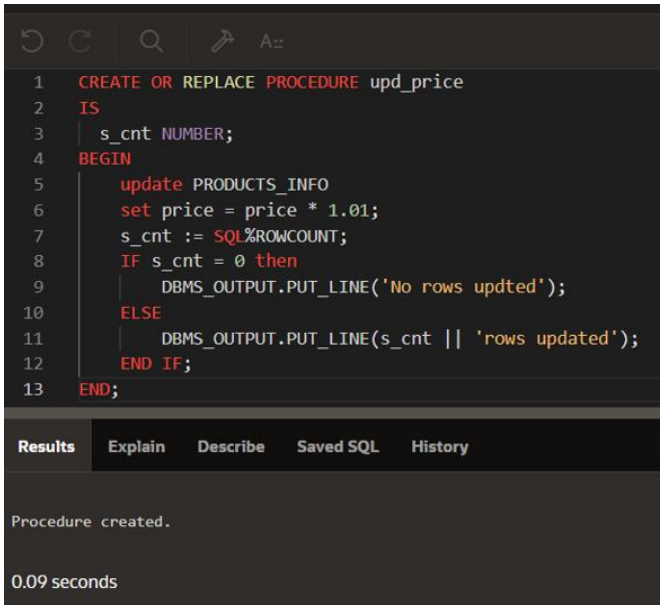
```
BEGIN
  PROD_ID := :PROD_ID;
  PROD_NAME := :PROD_NAME;
  COMPANY_ID := :COMPANY_ID;
  PROD_TYPE := :PROD_TYPE;

 IF LENGTH(PROD_NAME) < 5 THEN
    RAISE invalid_name;
 END IF;
  INSERT INTO PRODUCTS (PROD_ID, PROD_NAME, COMPANY_ID) VALUES (PROD_ID, PROD_NAME, COMPANY_ID);
  INSERT INTO PRODUCT_CATEGORY (PROD_ID, PROD_TYPE) VALUES (PROD_ID, PROD_TYPE);
```

Here we are defining exception, and giving system certain commands, if this kind of exception occurs:

```
EXCEPTION
  WHEN invalid_name THEN
  dbms_output.put_line('Invalid product name');
END;
```

- **SQL%Rowcount**



```
CREATE OR REPLACE PROCEDURE upd_price
IS
  s_cnt NUMBER;
BEGIN
    update PRODUCTS_INFO
    set price = price * 1.01;
    s_cnt := SQL%ROWCOUNT;
    IF s_cnt = 0 then
        DBMS_OUTPUT.PUT_LINE('No rows updted');
    ELSE
        DBMS_OUTPUT.PUT_LINE(s_cnt || 'rows updated');
    END IF;
END;
```

Results | Explain | Describe | Saved SQL | History

Procedure created.

0.09 seconds

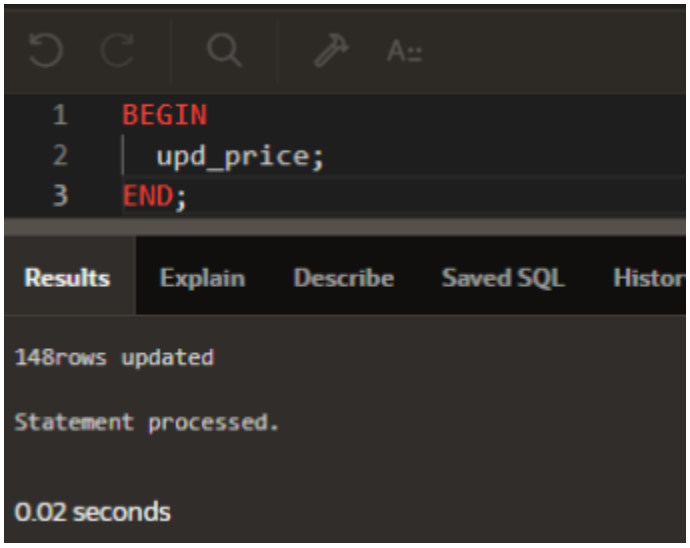
This is a PL/SQL stored procedure named "upd_price".

The procedure updates the "price" column in the "PRODUCTS_INFO" table by multiplying it by 1.01.

The variable "s_cnt" is used to store the number of rows that were updated, which is obtained using the SQL%ROWCOUNT attribute after the update statement.

If no rows were updated, it outputs the message "No rows updated" using the DBMS_OUTPUT.PUT_LINE procedure. Otherwise, it outputs the number of rows updated using the same procedure.

The purpose of this procedure appears to be to increase the price of all products in the "PRODUCTS_INFO" table by 1% (i.e., multiplying the price by 1.01).



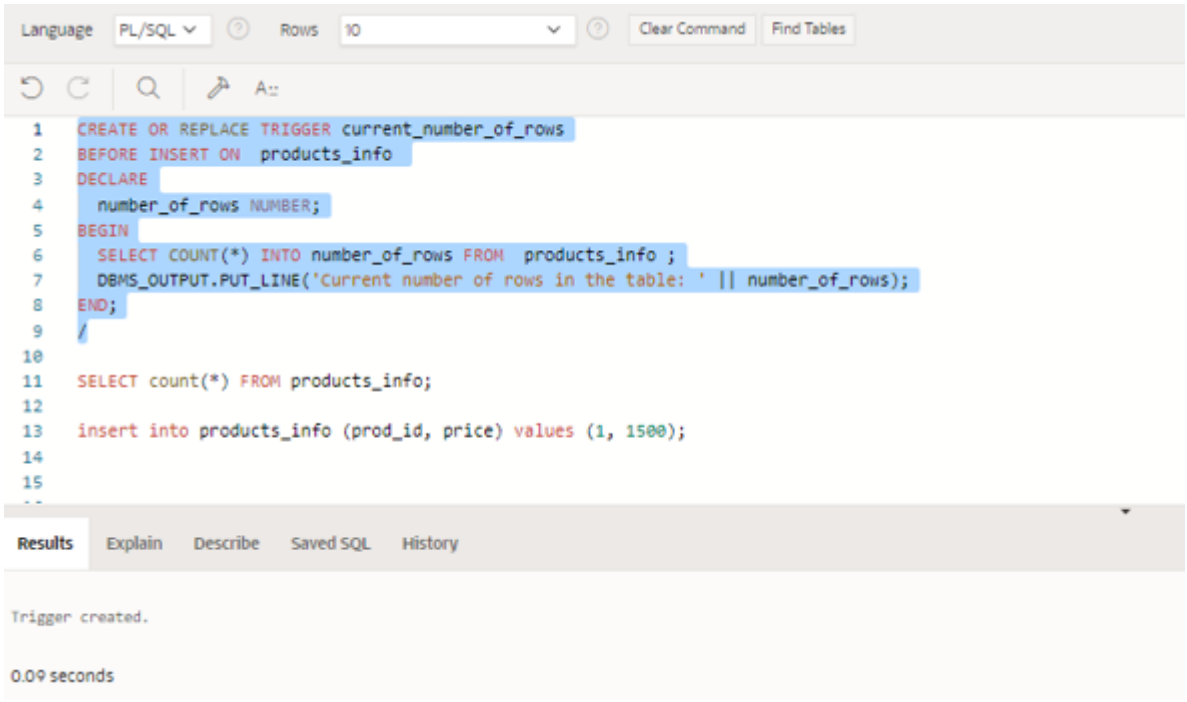
The purpose of this block is to execute the "upd_price" procedure, which updates the price of all products in the "PRODUCTS_INFO" table by 1%.

- **Trigger**

- Create a trigger before insert on any entity which will show the current number of rows in the table

```
CREATE OR REPLACE TRIGGER current_number_of_rows
BEFORE INSERT ON  products_info
DECLARE
  number_of_rows NUMBER;
BEGIN
  SELECT COUNT(*) INTO number_of_rows FROM  products_info ;
  DBMS_OUTPUT.PUT_LINE('Current number of rows in the table: ' ||
number_of_rows);
END;
/
```

- Created a trigger called current_number_of_rows that shows the number of rows in the products_info table



- First, let's check how many rows we have in the table.

Result: 147 rows

- Triggering a Trigger

Let us perform some DML operations on the products_info table. Here is one INSERT statement, which will create a new record in the table :

insert into products_info (prod_id, price) values (1, 1500);

As we can see the trigger works. The result shows the number of rows before adding any data to the table.

After adding data, the number of rows in the table is 148.