Introduction:

Team members and their responsibilities:

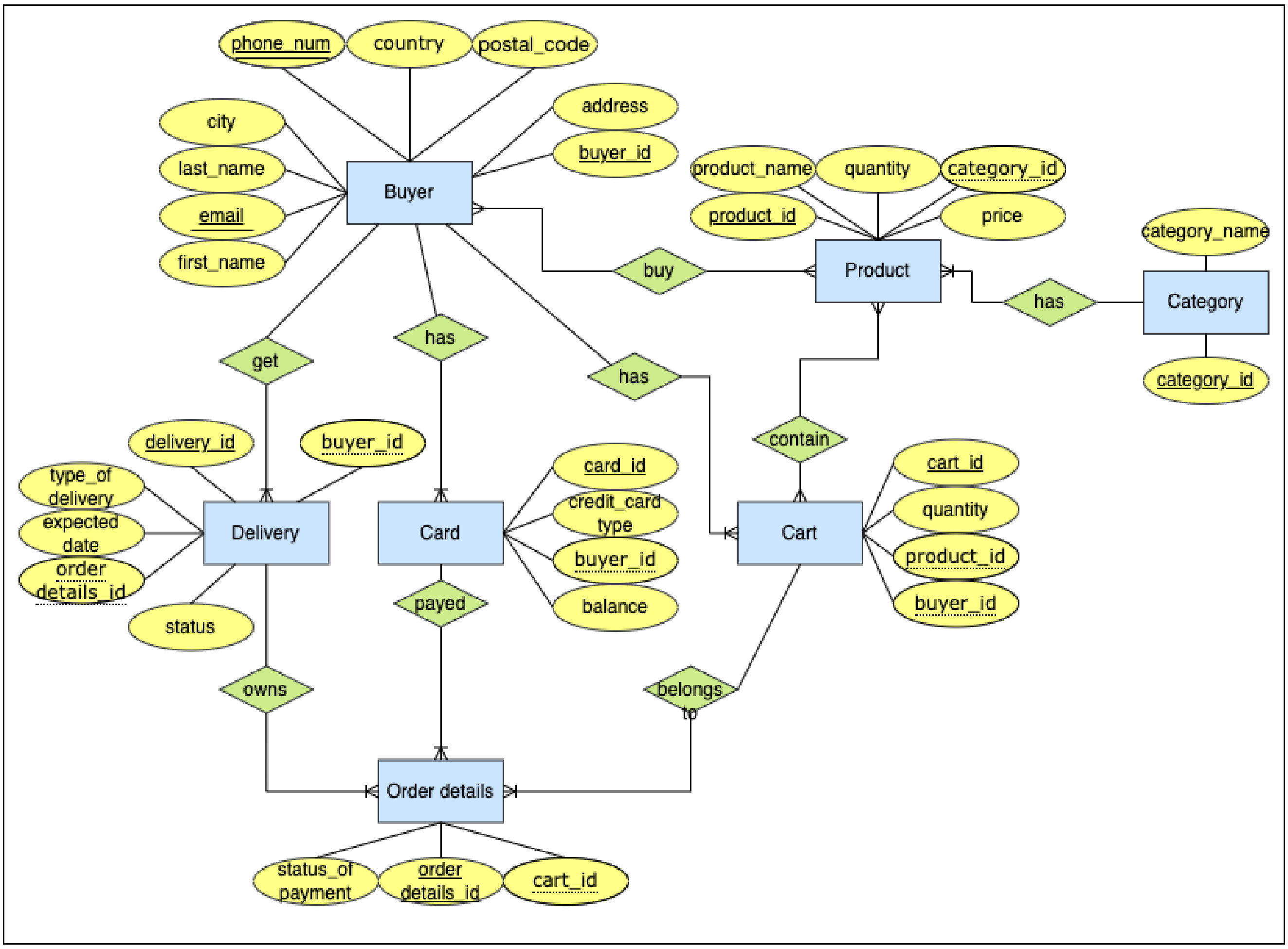
Mukan Zhanbolat 210103219 is responsible function, erd, report
Moldabek Arystan 210103271 is responsible for erd, transaction, trigger.
Bamish Ulykbek 210103222 is responsible for this erd, transaction, procedure
Okapova Azhar 210103270 is responsible for this erd,normalization,procedure
Alimbayeva Dilnaz 210103196 is responsible for this erd,normalization,exception

As a **topic**, we chose an online store of perfumes, cosmetics, self-care products called "*ArystanBeauty's Secret*".

Business process:

The **buyer** visits the website -> sees the **products** by **category** -> adds the products he wants to buy to the **cart** -> familiarizes himself with the **details of the order** -> pays for the order with a **card** -> waits for the **delivery**.
Hence the following entities arise: Buyer, Product, Cart, Order details, Card, Delivery.

ER Diagram:



Functional Dependencies AND normal:
Normalization is the process of removing redundant data.

First Normal Form (1NF) Requirements

The requirement of first normal form (1NF) is very simple and it is that the tables conform to the relational data model and follow certain relational principles.

Thus, for a database to be in 1 normal form, it is necessary that its tables follow the following relational principles:

The table must not contain duplicate rows.
Each table cell stores an atomic value (one non-composite value)
A column stores data of the same type
There are no arrays and lists in any form

Second Normal Form (2NF) Requirements

For a database to be in second normal form (2NF), its tables must meet the following requirements:

The table must be in first normal form
The table must have a key
All non-key columns of the table must depend on the complete key (in case it is a composite one)

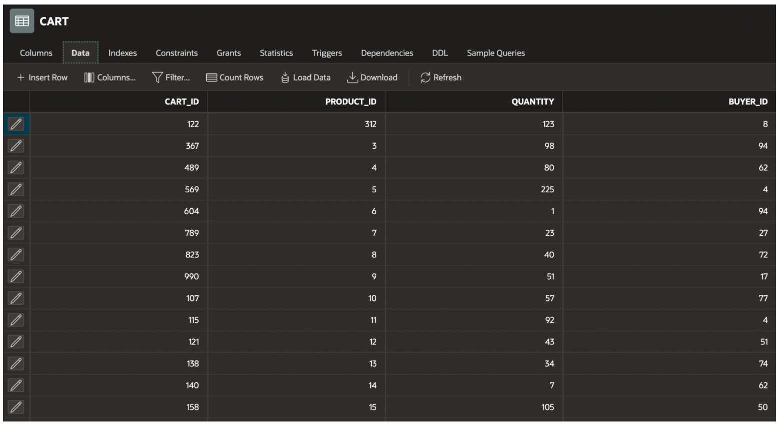
Third Normal Form (3NF) Requirements

The third normal form (3NF) requirement is that tables should not be transitively dependent.

cart(cart_id,quantity,product_id,buyer_id)

```
create table cart (
        cart_id INT,
        product_id INT,
        quantity INT
);
```

cart_id->quantity, product_id
cart_id,product_id->quantity



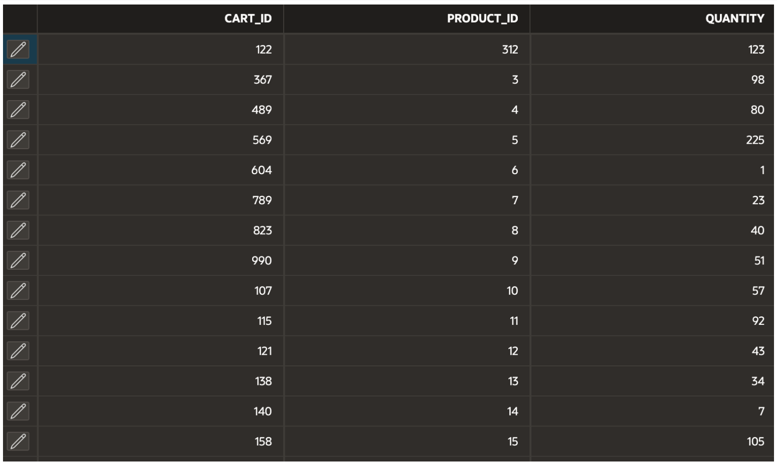
CART_ID	PRODUCT_ID	QUANTITY	BUYER_ID
122	312	123	8
367	3	98	94
489	4	80	62
569	5	225	4
604	6	1	94
789	7	23	27
823	8	40	72
990	9	51	17
107	10	57	77
115	11	92	4
121	12	43	51
138	13	34	74
140	14	7	62
158	15	105	50

The given table is already in 1NF as it has atomic values in each column and a unique primary key cart_id.

Second Normal Form (2NF):

To be in 2NF, the table should have only one non-key attribute that is dependent on the primary key. In our case, we can see that the quantity attribute is dependent on both the cart_id and product_id attributes. Therefore, we need to create a new table cart_items to separate this dependency.

cart_items table: (cart_id, product_id, quantity)



CART_ID	PRODUCT_ID	QUANTITY
122	312	123
367	3	98
489	4	80
569	5	225
604	6	1
789	7	23
823	8	40
990	9	51
107	10	57
115	11	92
121	12	43
138	13	34
140	14	7
158	15	105

The primary key of this table will be a composite key consisting of both cart_id and product_id columns. The quantity attribute will be dependent on this composite key.

Third Normal Form (3NF):

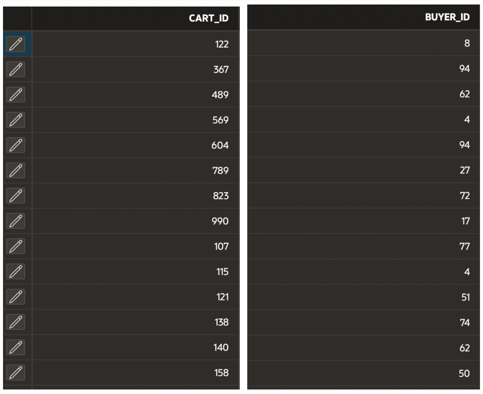
To be in 3NF, the table should not have any transitive dependencies. In our case, we can see that the buyer_id attribute is dependent only on the cart_id attribute, and not on any other attribute in the table. Therefore, we can create a new table carts to separate this dependency.

carts table: (cart_id, buyer_id)

The primary key of this table will be cart_id, and buyer_id will be dependent on this key.

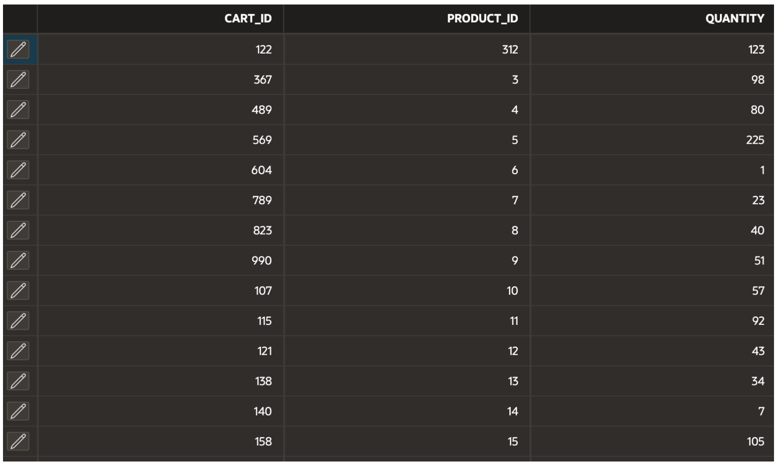
Now, we have three tables: carts, cart_items, and products.

carts_table: (cart_id, buyer_id)



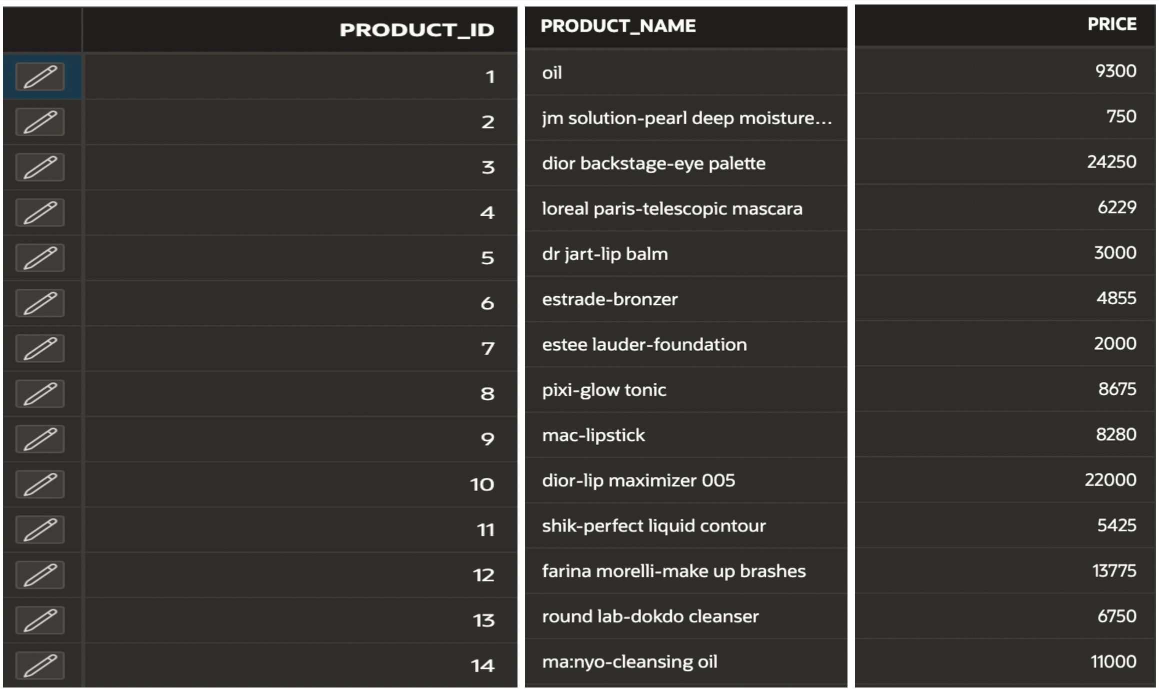
CART_ID	BUYER_ID
122	8
367	94
489	62
569	4
604	94
789	27
823	72
990	17
107	77
115	4
121	51
138	74
140	62
158	50

cart_items table: (cart_id, product_id, quantity)



	CART_ID	PRODUCT_ID	QUANTITY
	122	312	123
	367	3	98
	489	4	80
	569	5	225
	604	6	1
	789	7	23
	823	8	40
	990	9	51
	107	10	57
	115	11	92
	121	12	43
	138	13	34
	140	14	7
	158	15	105

products table: (product_id, product_name, price)

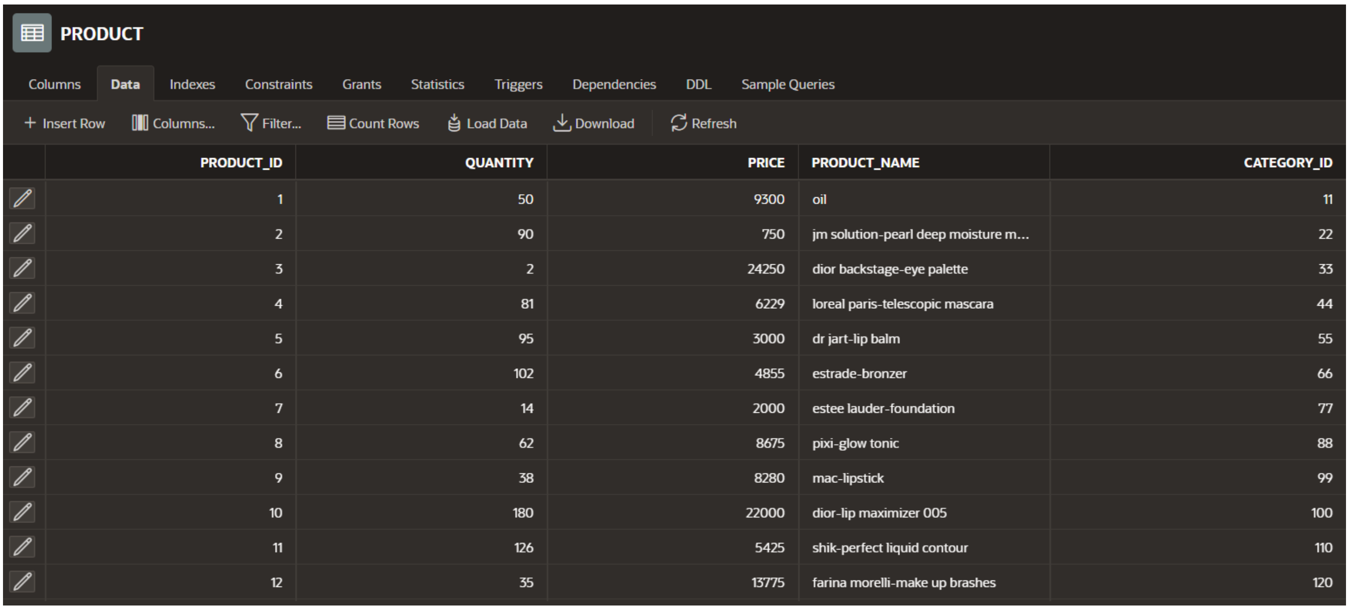


	PRODUCT_ID	PRODUCT_NAME	PRICE
	1	oil	9300
	2	jm solution-pearl deep moisture...	750
	3	dior backstage-eye palette	24250
	4	loreal paris-telescopic mascara	6229
	5	dr jart-lip balm	3000
	6	estrade-bronzer	4855
	7	estee lauder-foundation	2000
	8	pixi-glow tonic	8675
	9	mac-lipstick	8280
	10	dior-lip maximizer 005	22000
	11	shik-perfect liquid contour	5425
	12	farina morelli-make up brashes	13775
	13	round lab-dokdo cleanser	6750
	14	ma:nyo-cleansing oil	11000

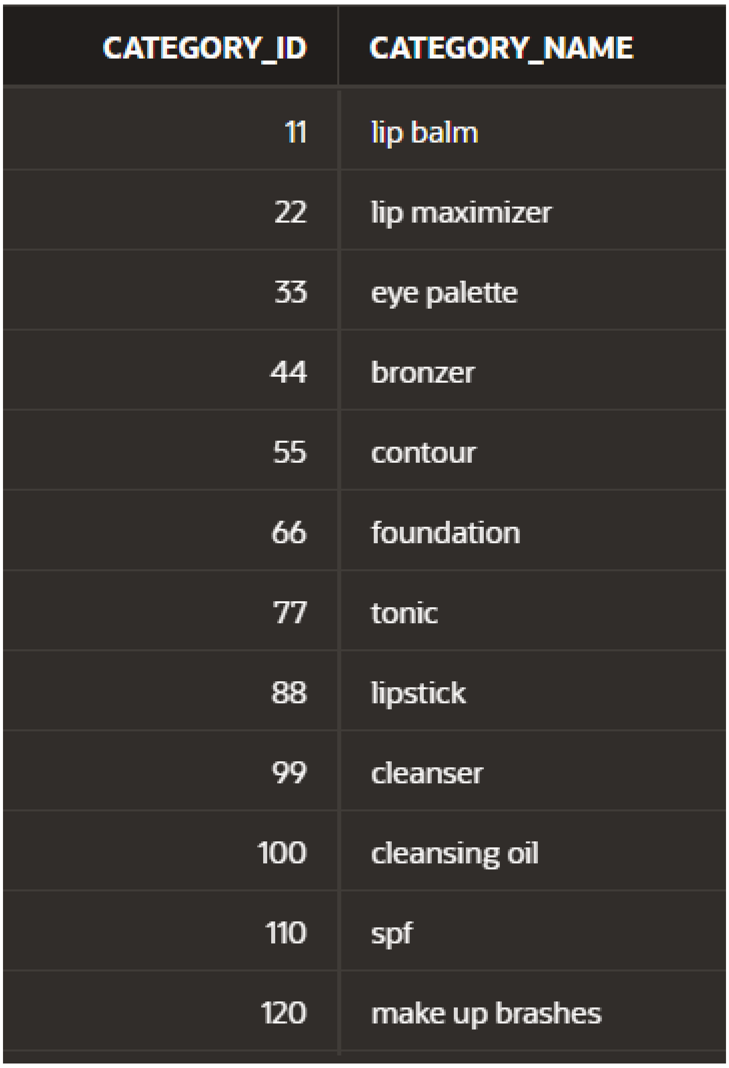
Each table has a unique primary key, and there are no transitive dependencies in any of the tables. Therefore, the tables are now in 3NF.

```
create table product (
        product_id INT,
        cart_id INT,
        quantity INT,
        price INT,
        product_name VARCHAR(50)
    );
```

product(product_id, product_name, quantity, category_id, price)
product_id -> product_name, quantity, category_id, price
category_id -> category_name



PRODUCT_ID	QUANTITY	PRICE	PRODUCT_NAME	CATEGORY_ID
1	50	9300	oil	11
2	90	750	jm solution-pearl deep moisture m...	22
3	2	24250	dior backstage-eye palette	33
4	81	6229	loreal paris-telescopic mascara	44
5	95	3000	dr jart-lip balm	55
6	102	4855	estrade-bronzer	66
7	14	2000	estee lauder-foundation	77
8	62	8675	pixi-glow tonic	88
9	38	8280	mac-lipstick	99
10	180	22000	dior-lip maximizer 005	100
11	126	5425	shik-perfect liquid contour	110
12	35	13775	farina morelli-make up brashes	120



CATEGORY_ID	CATEGORY_NAME
11	lip balm
22	lip maximizer
33	eye palette
44	bronzer
55	contour
66	foundation
77	tonic
88	lipstick
99	cleanser
100	cleansing oil
110	spf
120	make up brashes

this table is in 1NF because all data is represented in atomic values

this table is in 2NF because every non-key attribute in a relation is functionally dependent on the primary key.

The primary key is product_id, and the product_name, quantity, category_id, and price attributes depend on it. This meets his second condition of his 3NF.

There are no transitive dependencies between non-primary attributes as category_name is completely dependent on category_id which is part of the primary key. This satisfies the condition of 3NF.

product category(category_id ,category_name)

```
create table product_category(
category_id INT,
category_name VARCHAR(50)
);
```

category_id->category_name

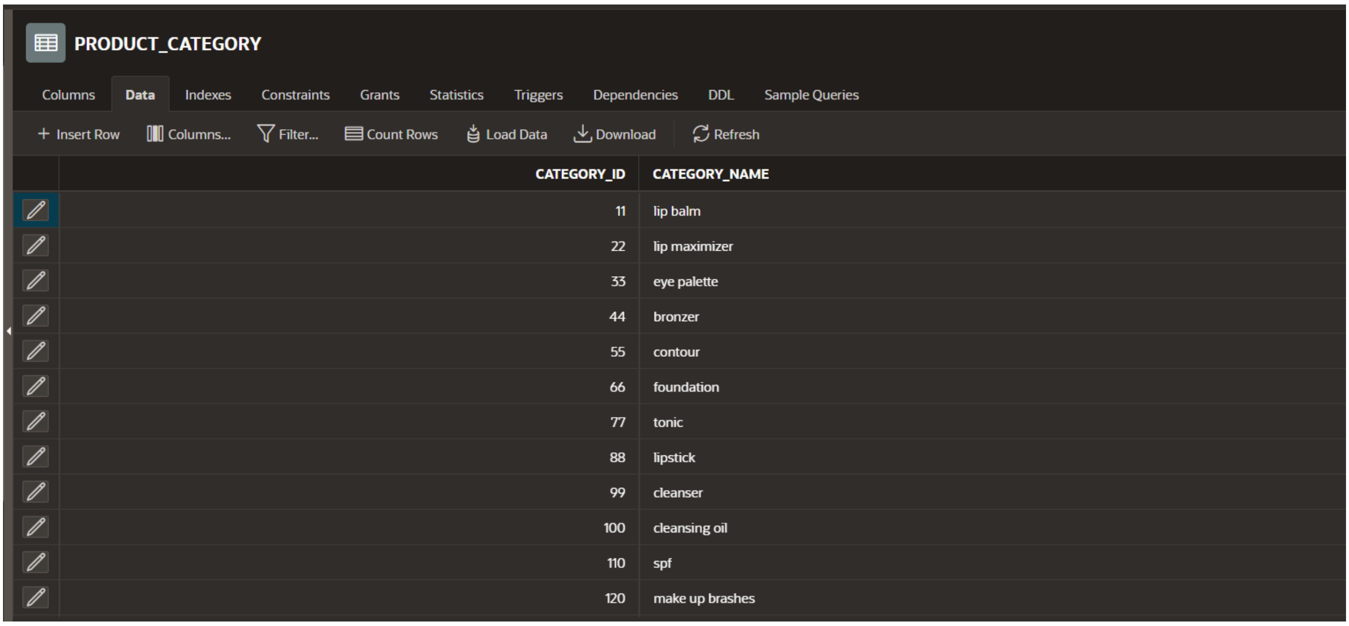
.

this table is in 1NF because all data is represented in atomic values

this table is in 2NF because every non-key attribute in a relation is functionally dependent on the primary key.

product category table simply contains the category_id and category_name columns, it is already in 3NF

category_id is the primary key, and there is just one non-key attribute, category_name, which is completely functionally dependent on the primary key. As a result, the table meets the 3NF requirement.

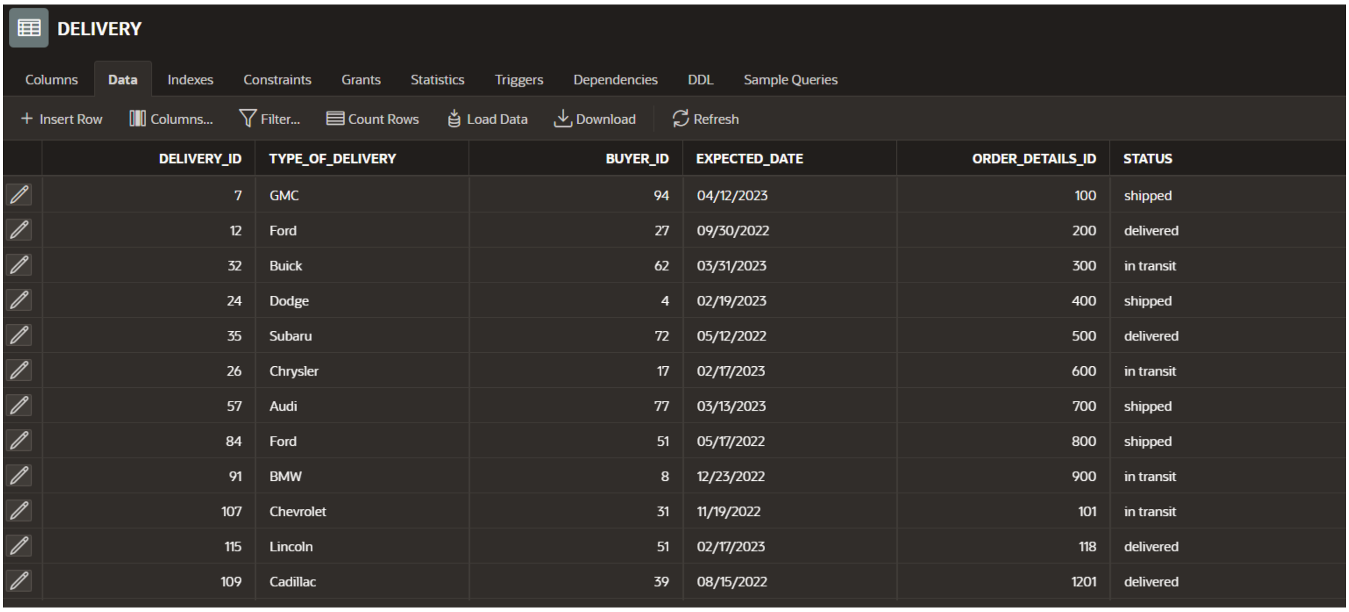


CATEGORY_ID	CATEGORY_NAME
11	lip balm
22	lip maximizer
33	eye palette
44	bronzer
55	contour
66	foundation
77	tonic
88	lipstick
99	cleanser
100	cleansing oil
110	spf
120	make up brashes

delivery(delivery_id,type_of_delivery,expected_date,order_details_id,buyer_id,status)

```
create table delivery (
    delivery_id INT,
    type_of_delivery VARCHAR(50),
    buyer_id INT,
    expected_date DATE,
    order_details_id INT,
    status VARCHAR(50)
);
```

delivery_id -> type_of_delivery, expected_date, order_details_id, buyer_id, status
buyer_id -> order_details_id



DELIVERY_ID	TYPE_OF_DELIVERY	BUYER_ID	EXPECTED_DATE	ORDER_DETAILS_ID	STATUS
7	GMC	94	04/12/2023	100	shipped
12	Ford	27	09/30/2022	200	delivered
32	Buick	62	03/31/2023	300	in transit
24	Dodge	4	02/19/2023	400	shipped
35	Subaru	72	05/12/2022	500	delivered
26	Chrysler	17	02/17/2023	600	in transit
57	Audi	77	03/13/2023	700	shipped
84	Ford	51	05/17/2022	800	shipped
91	BMW	8	12/23/2022	900	in transit
107	Chevrolet	31	11/19/2022	101	in transit
115	Lincoln	51	02/17/2023	118	delivered
109	Cadillac	39	08/15/2022	1201	delivered

this table is in 1NF because all data is represented in atomic values

every non-key attribute in a relation is functionally dependent on the primary key.Therefore, the table is in 2NF.

Delivery_id is the primary key.Because each non-key property is completely dependent on the primary key, there are no partial dependencies.

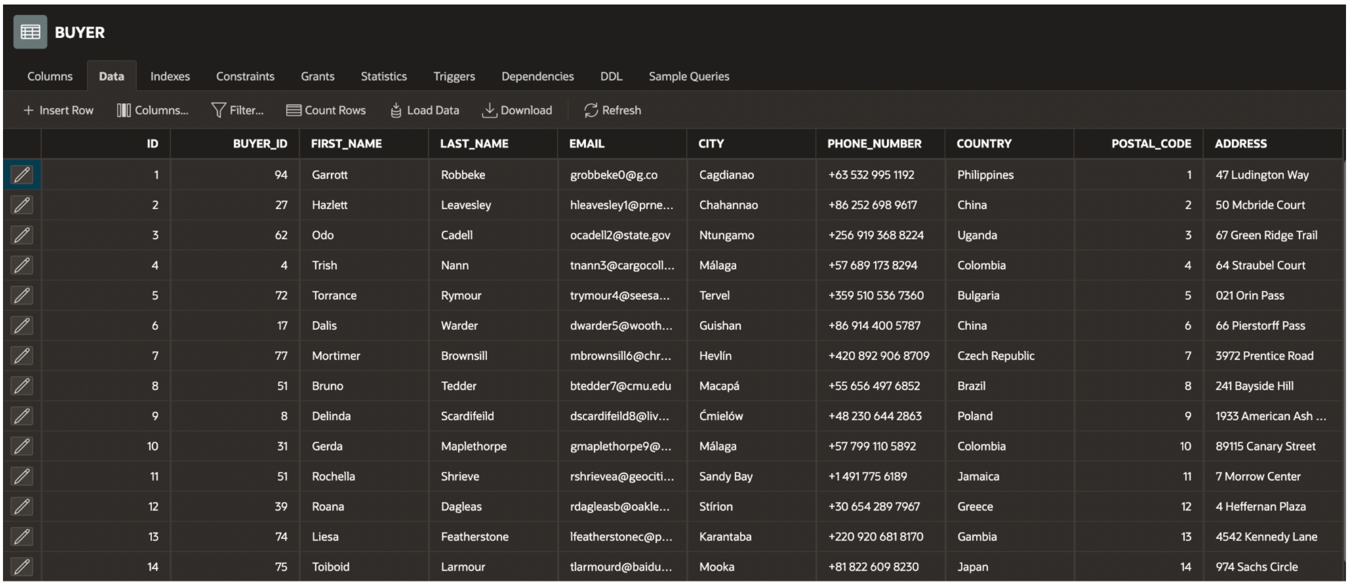
There is no functional relationship between non-key attributes and the primary key. As a result, the relationship is in 2NF and 3NF.

the order_details_id and buyer_id properties are not mutually exclusive, however they are both dependent on the primary key delivery_id. As a result, the relation fulfills the 3NF because there is no transitive functional dependency between order_details_id and buyer_id.

buyer(buyer_id,last_name,first_name,address,tel_no,city,country,email,postal_code)

```
create table buyer (
    buyer_id INT,
    first_name VARCHAR(50),
    last_name VARCHAR(50),
    email VARCHAR(50),
    city VARCHAR(50),
    phone_number VARCHAR(50),
    country VARCHAR(50),
    postal_code INT,
    address VARCHAR(50)
);
```

buyer_id->last_name,first_name,address,tel_no,city,country,email,postal_code
address,city,postal_code->country
buyer_id,address->city,country,postal_code
buyer_id,email->last_name,first_name,address,tel_no

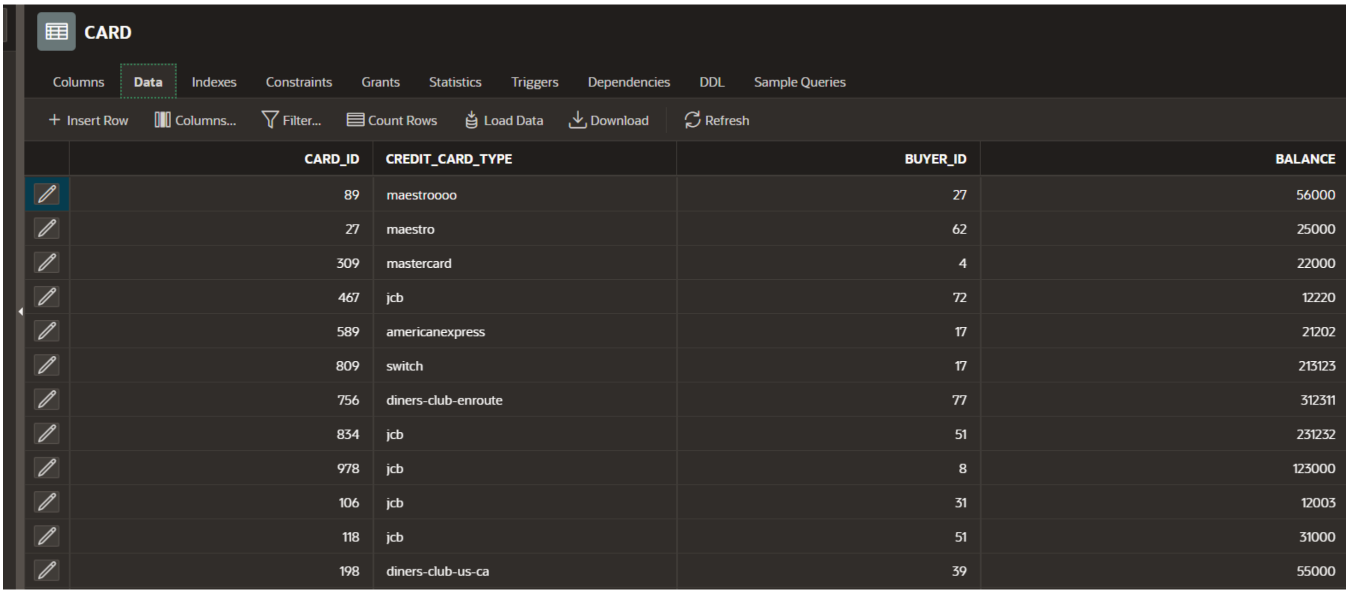


ID	BUYER_ID	FIRST_NAME	LAST_NAME	EMAIL	CITY	PHONE_NUMBER	COUNTRY	POSTAL_CODE	ADDRESS
1	94	Garrott	Robbeke	grobbeke0@g.co	Cagdianao	+63 532 995 1192	Philippines	1	47 Ludington Way
2	27	Hazlett	Leavesley	hleavesley1@prne...	Chahannao	+86 252 698 9617	China	2	50 Mcbride Court
3	62	Odo	Cadell	ocadell2@state.gov	Ntungamo	+256 919 368 8224	Uganda	3	67 Green Ridge Trail
4	4	Trish	Nann	tnann3@cargocoll...	Málaga	+57 689 173 8294	Colombia	4	64 Straubel Court
5	72	Torrance	Rymour	trymour4@seesa...	Tervel	+359 510 536 7360	Bulgaria	5	021 Orin Pass
6	17	Dalis	Warder	dwarder5@wooth...	Guishan	+86 914 400 5787	China	6	66 Pierstorff Pass
7	77	Mortimer	Brownsill	mbrownsill6@chr...	Hevlín	+420 892 906 8709	Czech Republic	7	3972 Prentice Road
8	51	Bruno	Tedder	btedder7@cmu.edu	Macapá	+55 656 497 6852	Brazil	8	241 Bayside Hill
9	8	Delinda	Scardifeild	dscardifeild8@liv...	Ćmielów	+48 230 644 2863	Poland	9	1933 American Ash ...
10	31	Gerda	Maplethorpe	gmaplethorpe9@...	Málaga	+57 799 110 5892	Colombia	10	89115 Canary Street
11	51	Rochella	Shrieve	rshrievea@geociti...	Sandy Bay	+1 491 775 6189	Jamaica	11	7 Morrow Center
12	39	Roana	Dagleas	rdagleasb@oakle...	Stírion	+30 654 289 7967	Greece	12	4 Heffernan Plaza
13	74	Liesa	Featherstone	lfeatherstonec@p...	Karantaba	+220 920 681 8170	Gambia	13	4542 Kennedy Lane
14	75	Toiboid	Larmour	tlarmourd@baidu...	Mooka	+81 822 609 8230	Japan	14	974 Sachs Circle

First, we identify the functional dependencies:
buyer_id -> last_name, first_name, address, tel_no, city, country, email, postal_code
address, city, postal_code -> country
buyer_id, address -> city, country, postal_code
buyer_id, email -> last_name, first_name, address, tel_no
Now, we need to check if it's in 2NF. A relation is in 2NF if it has no partial dependencies, i.e., if all non-key attributes are fully dependent on the primary key.
The primary key for this relation is buyer_id. All other attributes are dependent on buyer_id in some way or another, so the relation is in 2NF.
Next, we need to remove transitive dependencies. A transitive dependency is when a non-key attribute depends on another non-key attribute, which in turn depends on the primary key.
We can see that there is a transitive dependency in the relation:
address, city, postal_code -> country

Here, country is dependent on the combination of address, city, and postal_code. To remove this dependency, we can create a new relation with address, city, postal_code, and country as attributes:
buyer_location(address, city, postal_code, country)
Now, we update the functional dependencies for the remaining attributes in the original relation:
buyer_id -> last_name, first_name, tel_no, email
buyer_id, address -> city, postal_code
Note that we have removed the transitive dependency and created a new relation for it. Now the relation is in 3NF because it meets the conditions of 2NF and has no transitive dependencies.

```
create table card (
 card_id INT,
 credit_card_type VARCHAR(50),
 buyer_id INT,
balance INT
)
```



CARD_ID	CREDIT_CARD_TYPE	BUYER_ID	BALANCE
89	maestroooo	27	56000
27	maestro	62	25000
309	mastercard	4	22000
467	jcb	72	12220
589	americanexpress	17	21202
809	switch	17	213123
756	diners-club-enroute	77	312311
834	jcb	51	231232
978	jcb	8	123000
106	jcb	31	12003
118	jcb	51	31000
198	diners-club-us-ca	39	55000

card(card_id,credit_card_type,buyer_id,balance)
card_id -> credit_card_type, buyer_id,
this table is in 1NF because all data is represented in atomic values

this table is in 2NF because every non-key attribute in a relation is functionally dependent on the primary key.

The primary key is card_id, which uniquely identifies each row in the table. The functional dependency card_id -> credit_card_type, buyer_id, balance states that the values of credit_card_type, buyer_id, and balance are uniquely determined for each given card_id.

Every non-key attribute in 3NF must be directly reliant on the primary key, with no transitive dependencies between non-key attributes. All of the non-key properties (credit_card_type, buyer_id, and balance) in the provided functional dependency are directly reliant on the main key (card_id), hence the table meets the 3NF requirements

```
create table order_detalis (
    status_of_payment VARCHAR(50),
    order_detalis_id INT,
    cart_id INT
);
```

order_details(status_of_payment,order_details_id,cart_id)

order_details_id → status_of_payment,cart_id

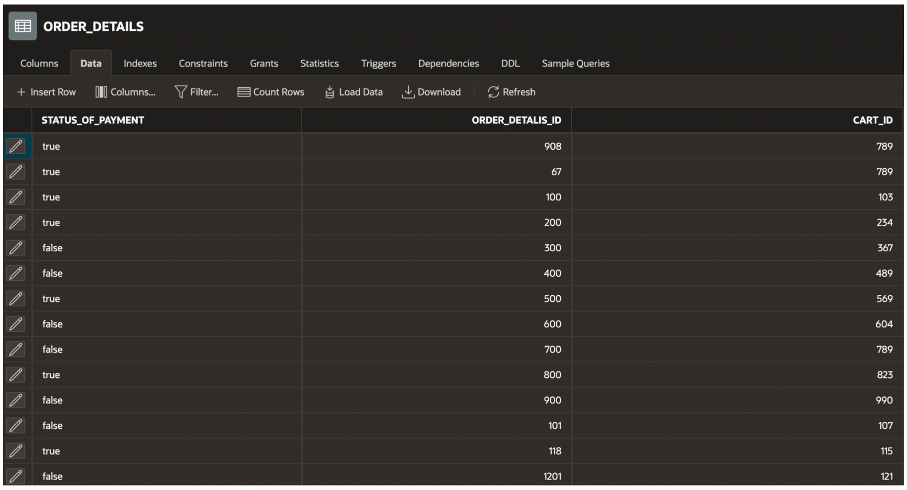
order_details_id,cart_id → status_of_payment

First Normal Form (1NF):

Make sure each attribute is atomic. No attributes with multiple values.

All attributes must depend on the primary key.

Since each attribute has an atomic value and the primary key order_details_id is a single attribute, the specified table already satisfies the first normal form.

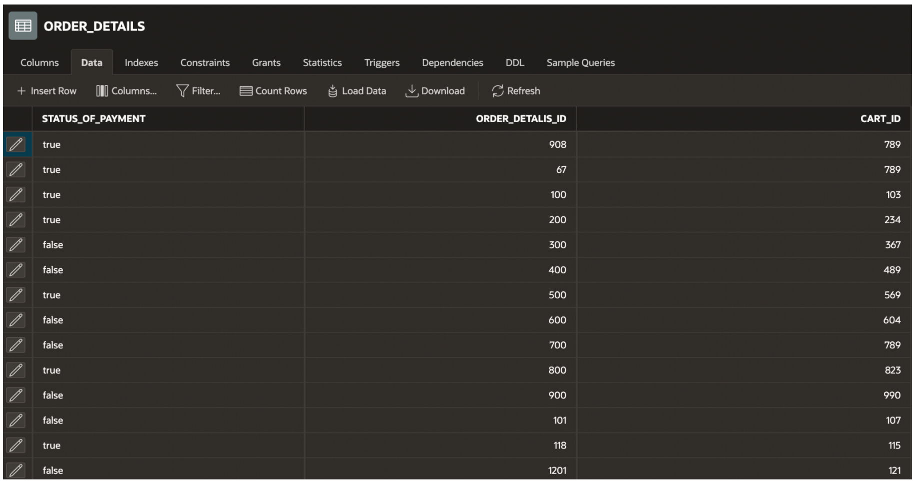


STATUS_OF_PAYMENT	ORDER_DETALIS_ID	CART_ID
true	908	789
true	67	789
true	100	103
true	200	234
false	300	367
false	400	489
true	500	569
false	600	604
false	700	789
true	800	823
false	900	990
false	101	107
true	118	115
false	1201	121

Second Normal Form (2NF):

Satisfies first normal form requirements.

Remove partial dependencies. Attributes that depend only on part of the primary key. In the given table, status_of_payment only depends on the primary key order_details_id. So the table is already 2NF.



STATUS_OF_PAYMENT	ORDER_DETALIS_ID	CART_ID
true	908	789
true	67	789
true	100	103
true	200	234
false	300	367
false	400	489
true	500	569
false	600	604
false	700	789
true	800	823
false	900	990
false	101	107
true	118	115
false	1201	121

Third Normal Form (3NF):

Satisfies the second normal form requirements.

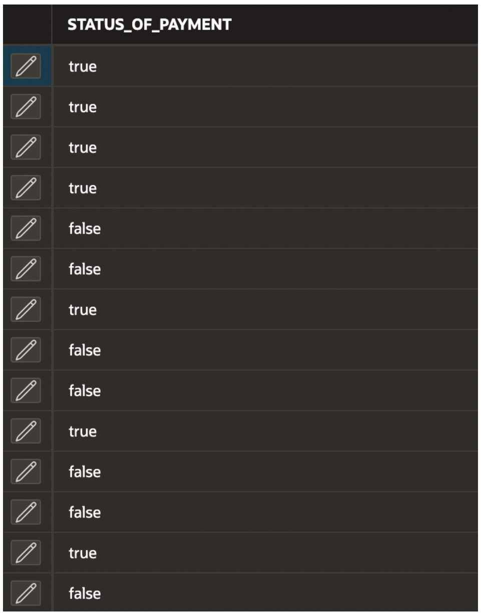
Remove transitive dependencies. Attributes that depend on non-primary key attributes.

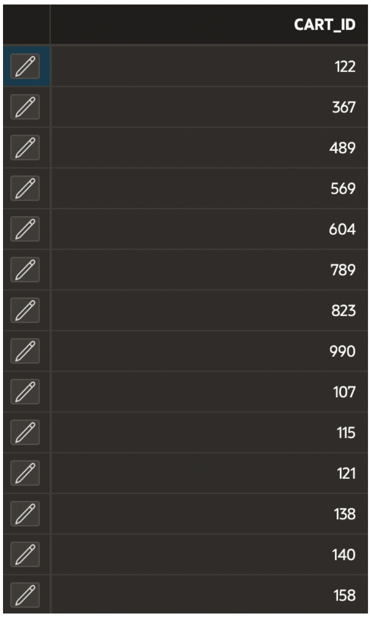
In the given table, we can see that status_of_payment depends on order_details_id which is the primary key and cart_id also depends on his order_details_id. However, status_of_payment does not depend on cart_id. This means that there is a transitive dependency between cart_id and status_of_payment.

To get rid of this transitive dependency, we can create a new table called cart_details with her two attributes: cart_id (primary key) and status_of_payment. The original order_details table has two attributes:

order_details_id and cart_id (both together act as a primary key).

cart_details





STATUS_OF_PAYMENT	CART_ID
true	122
true	367
true	489
true	569
false	604
false	789
true	823
false	990
false	107
true	115
false	121
false	138
true	140
false	158

- Explanation and coding part of each item from “Add the following”:

1) **Procedure which does group by information**

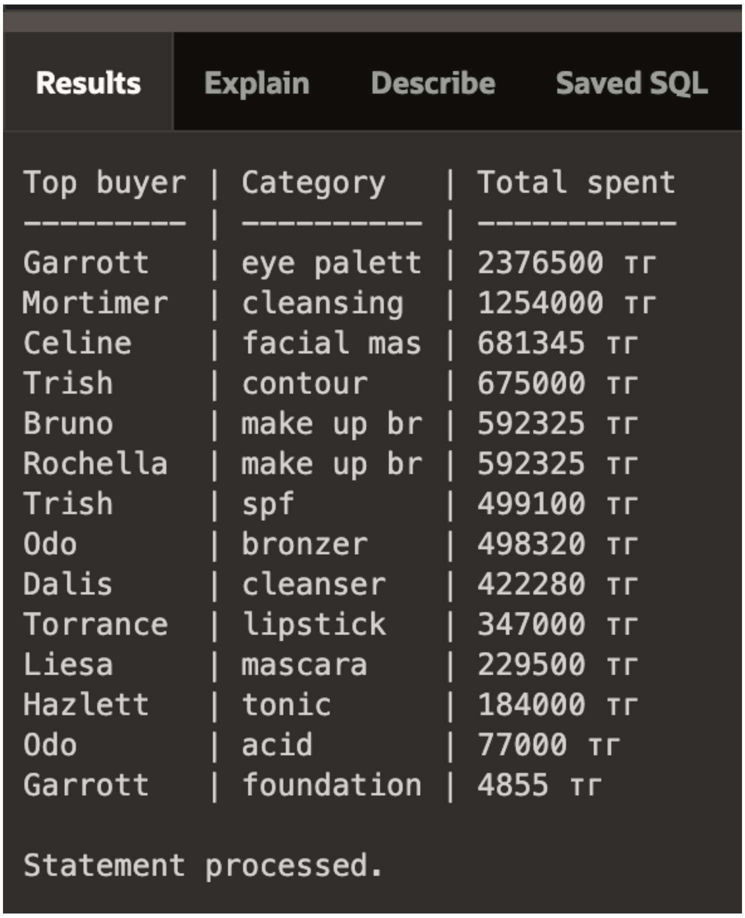
Creating the shows_amount_spent procedure:

```
create or replace PROCEDURE shows_amount_spent
IS
BEGIN
  DBMS_OUTPUT.PUT_LINE('Top buyer | Category    | Total spent ');
  DBMS_OUTPUT.PUT_LINE('--------- | ---------- | ----------- ');
  FOR lo IN(
    SELECT buyer.first_name, product_category.category_name,
     SUM(cart.quantity * product.price) AS total_amount
    FROM buyer, cart, order_details, product, product_category
    WHERE
        buyer.buyer_id = cart.buyer_id AND
        cart.cart_id = order_details.cart_id AND
        cart.product_id = product.product_id AND
        product.category_id = product_category.category_id

    GROUP BY product_category.category_name, buyer.first_name
    ORDER BY total_amount DESC
  )
  LOOP
    DBMS_OUTPUT.PUT_LINE(
        RPAD(lo.first_name, 9) || ' | ' ||
        RPAD(lo.category_name, 10) || ' | ' ||
        RPAD(lo.total_amount || ' тг', 10)
    );
  END LOOP;
END;
```

Calling the procedure:

```
BEGIN
         shows_amount_spent;
END;
```

And we have it coming out:



```
Top buyer | Category   | Total spent
--------- | ---------- | -----------
Garrott   | eye palett | 2376500 тг
Mortimer  | cleansing  | 1254000 тг
Celine    | facial mas | 681345 тг
Trish     | contour    | 675000 тг
Bruno     | make up br | 592325 тг
Rochella  | make up br | 592325 тг
Trish     | spf        | 499100 тг
Odo       | bronzer    | 498320 тг
Dalis     | cleanser   | 422280 тг
Torrance  | lipstick   | 347000 тг
Liesa     | mascara    | 229500 тг
Hazlett   | tonic      | 184000 тг
Odo       | acid       | 77000 тг
Garrott   | foundation | 4855 тг

Statement processed.
```

Explanation of the procedure:

This is a procedure that displays information about top buyers, how much money was spent by buyers on each category of goods.

This procedure uses SQL queries to get the necessary data from multiple database tables. Queries combine data about the buyer, the product, the product category and the quantity of the purchased product from the tables "buyer", "basket", "order_details", "product" and "product_category".

The received data is grouped by the name of the buyer and the name of the product category, and then sorted by the total amount spent by each buyer for each product category.

Then, in a loop, the received information is output using DBMS_OUTPUT.PUT_LINE() procedure. Each line of output data contains the name of the buyer, the name of the product category and the total amount of money spent on products of this category.

2) Function which counts the number of records:

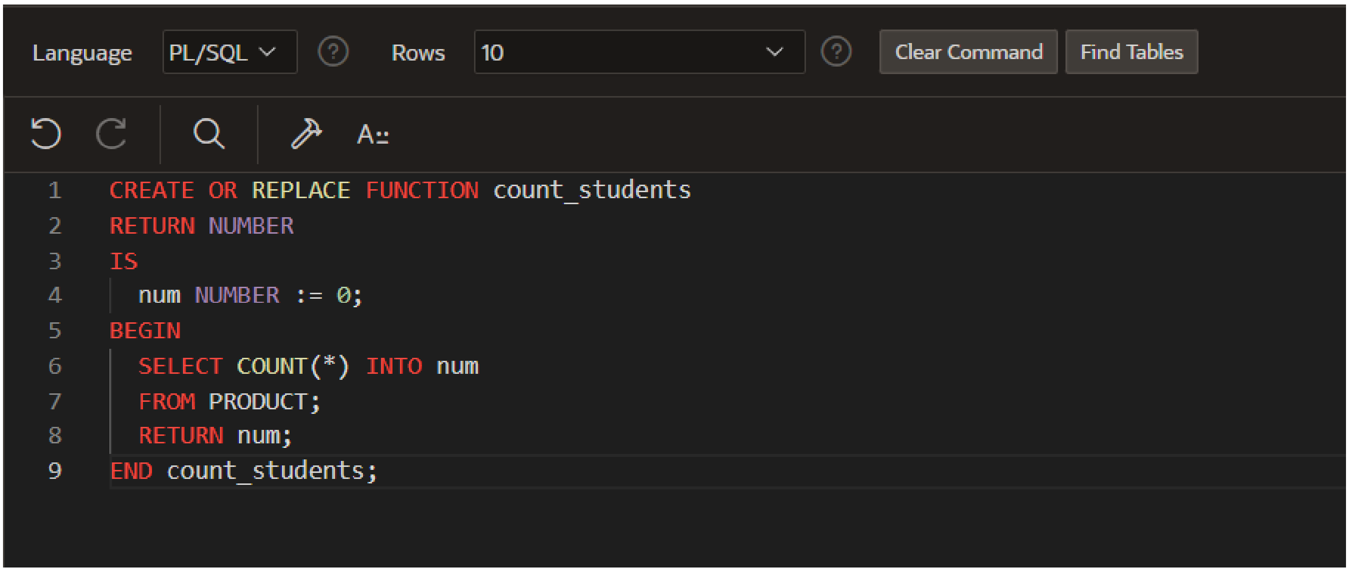
Here is my function which will return a number of records in a table.
Firstly I created a function named *count_students* which will return type NUMBER. Then assigned 0 to a variable *num.* In the BEGIN part selected count of all the records in “table” where instead of table you can write any name of a table and assigned it to the *num* variable. Then returning the value of *num.*

```
CREATE OR REPLACE FUNCTION count_students
RETURN NUMBER
IS
  num NUMBER := 0;
BEGIN
  SELECT COUNT(*) INTO num
  FROM PRODUCT;
  RETURN num;
END count_students;
```

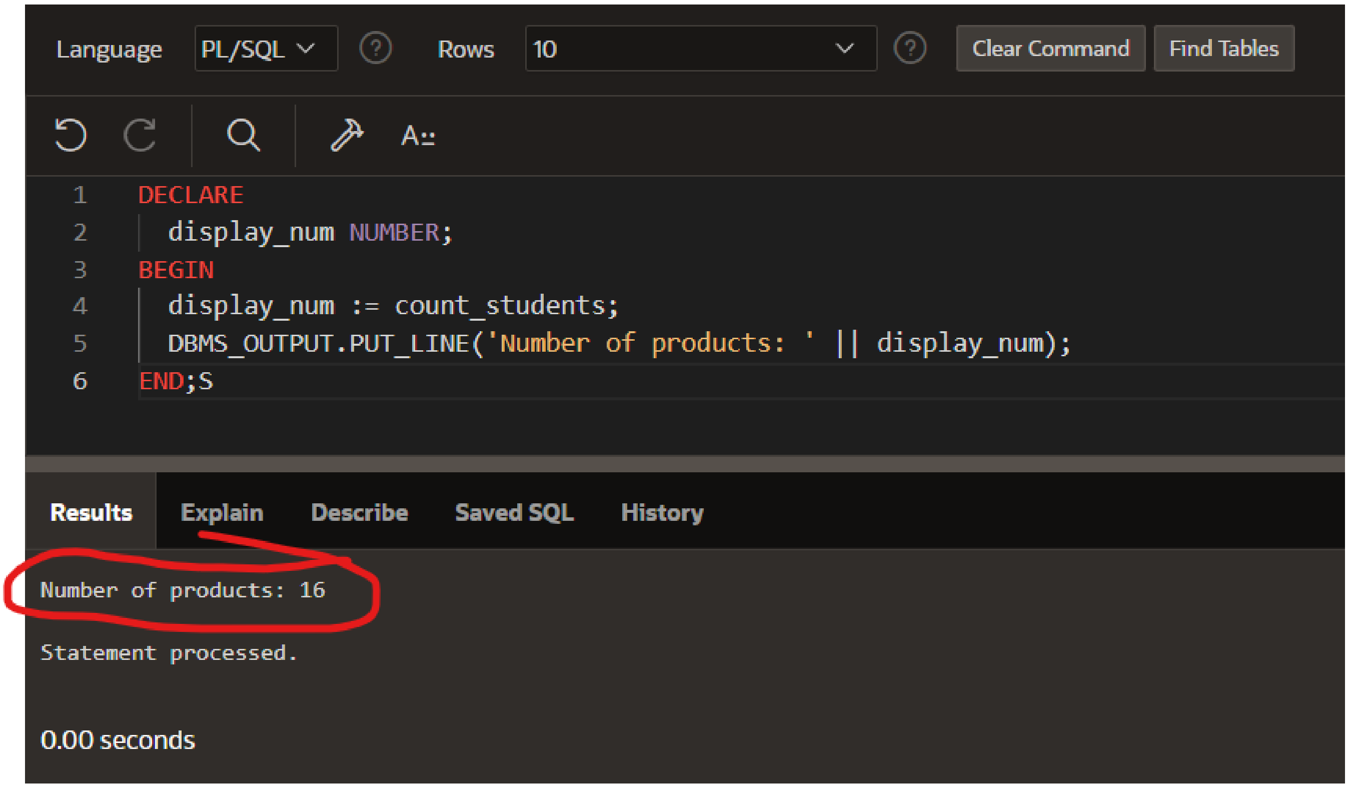
Below is the code to check if the function worked correctly or not, where in the DECLARE part declared NUMBER display_num, then in the BEGIN part assigned the result of our previous function to it and returned its new value to the console.

```
DECLARE
  display_num NUMBER;
BEGIN
  display_num := count_students;
  DBMS_OUTPUT.PUT_LINE('Number of products: ' || display_num);
END;
```

Below are some picture examples of the code.



```
CREATE OR REPLACE FUNCTION count_students
RETURN NUMBER
IS
  num NUMBER := 0;
BEGIN
  SELECT COUNT(*) INTO num
  FROM PRODUCT;
  RETURN num;
END count_students;
```



```
DECLARE
  display_num NUMBER;
BEGIN
  display_num := count_students;
  DBMS_OUTPUT.PUT_LINE('Number of products: ' || display_num);
END;S
```

```
Number of products: 16

Statement processed.

0.00 seconds
```

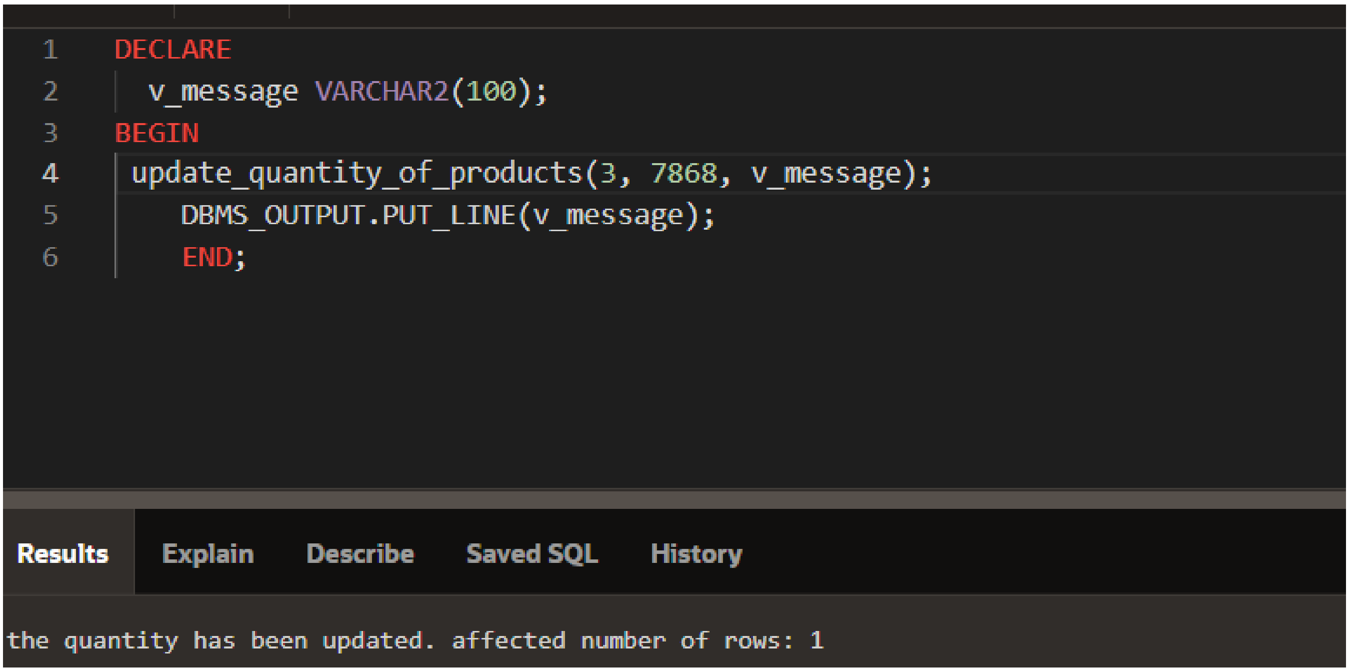
3) Procedure which uses SQL%ROWCOUNT to determine the number of rows affected

In this procedure, we first check if the product with the specified product_id exists in the product table. We do this by executing a SELECT COUNT(*) statement and storing the result in the q_product_count variable. If q_product_count is 0, we set the q_message output parameter to ‘this product does not exist'.

If the product exists, we then check if the q_new_quantity parameter is negative. If it is, we set the q_message output parameter to 'new value cannot be negative.'.

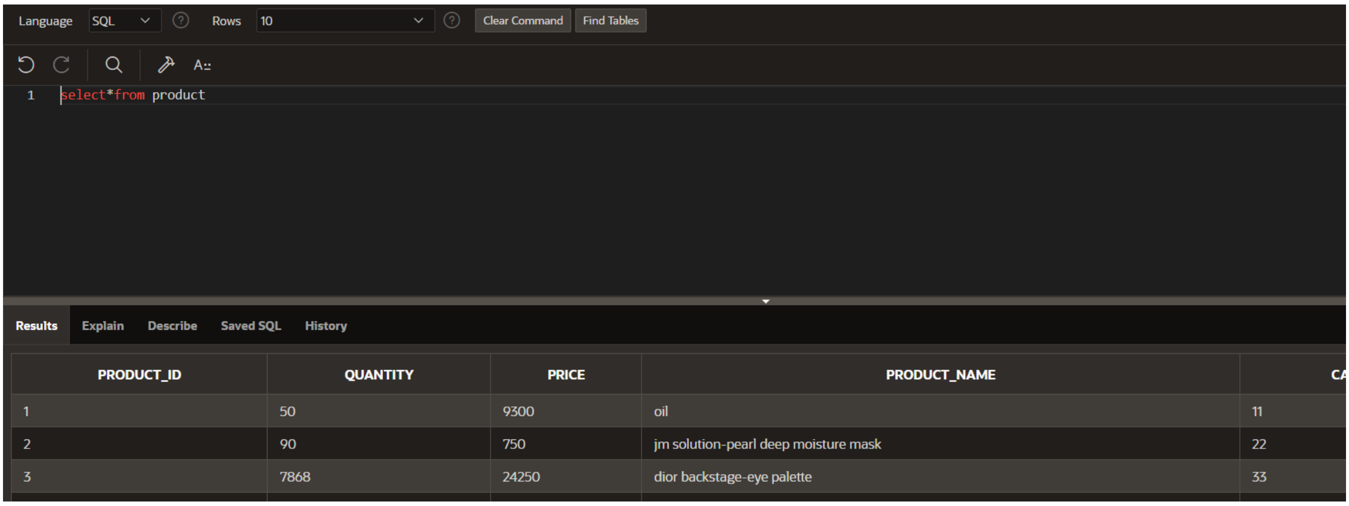
If both checks pass, we execute an UPDATE statement on the product table, setting the stock column to the new value for the specified product. We then set the q_message output parameter to a message that includes the number of rows affected by the UPDATE statement.

```
CREATE OR REPLACE PROCEDURE update_quantity_of_products(
   q_product_id IN NUMBER,
   q_new_quantity IN NUMBER,
   q_message OUT VARCHAR2
) AS
   q_product_count NUMBER;
BEGIN
   SELECT COUNT(*) INTO q_product_count
   FROM product
   WHERE product_id = q_product_id;

   IF q_product_count = 0 THEN
      q_message := 'this product does not exist';
   ELSIF q_new_quantity < 0 THEN
      q_message := 'new value cannot be negative';
   ELSE
      UPDATE product
      SET quantity = q_new_quantity
      WHERE product_id = q_product_id;

      q_message := 'the quantity has been updated. affected number of
    rows: ' || SQL%ROWCOUNT;
   END IF;
END;

procedure call
DECLARE
 v_message VARCHAR2(100);
BEGIN
 update_quantity_of_products(3, 2, v_message);
   DBMS_OUTPUT.PUT_LINE(v_message);
   END;
```



```
DECLARE
  v_message VARCHAR2(100);
BEGIN
 update_quantity_of_products(3, 7868, v_message);
    DBMS_OUTPUT.PUT_LINE(v_message);
    END;
```

```
the quantity has been updated. affected number of rows: 1
```



Language SQL Rows 10

```
select*from product
```

PRODUCT_ID	QUANTITY	PRICE	PRODUCT_NAME	CA
1	50	9300	oil	11
2	90	750	jm solution-pearl deep moisture mask	22
3	7868	24250	dior backstage-eye palette	33

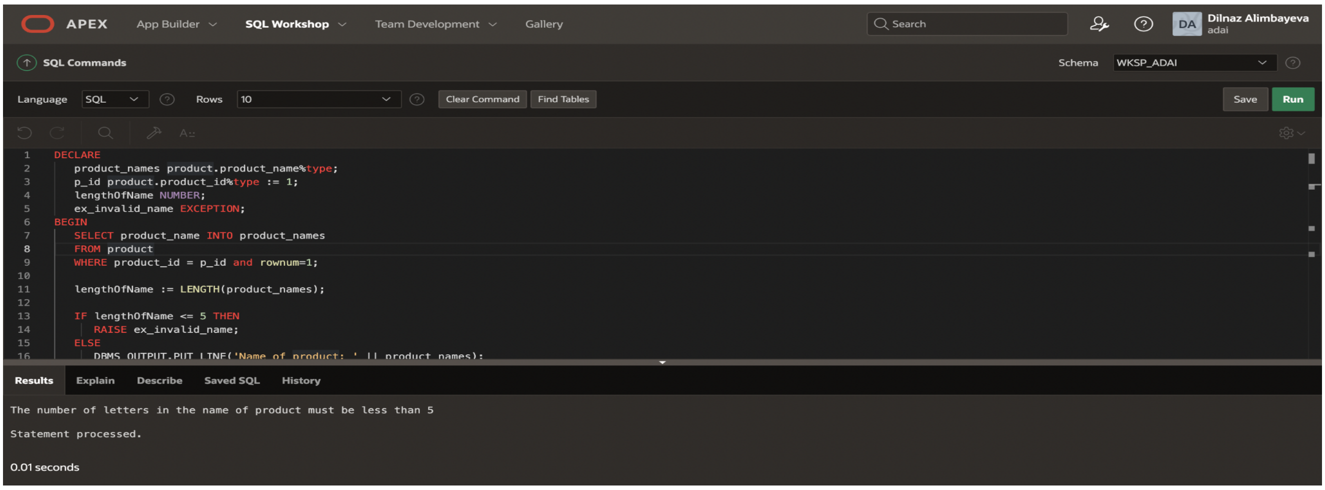
As you see here,we have updated quantity of product which id is equal to 3.

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

Exception

```
DECLARE
  product_names product.product_name%type;
  p_id product.product_id%type := 1;
  lengthOfName NUMBER;
  ex_invalid_name EXCEPTION;
BEGIN
```

```
  SELECT product_name INTO product_names
  FROM product
  WHERE product_id = p_id and rownum=1;
  lengthOfName := LENGTH(product_names);

  IF lengthOfName <= 5 THEN
    RAISE ex_invalid_name;
  ELSE
    DBMS_OUTPUT.PUT_LINE('Name of product: ' || product_names);
  END IF;
EXCEPTION
  WHEN ex_invalid_name THEN
    DBMS_OUTPUT.PUT_LINE('The number of letters in the name of
product must be less than 5');
  WHEN no_data_found THEN
    DBMS_OUTPUT.PUT_LINE('No data found!');
  WHEN others THEN
    DBMS_OUTPUT.PUT_LINE('Error!');
END;
```

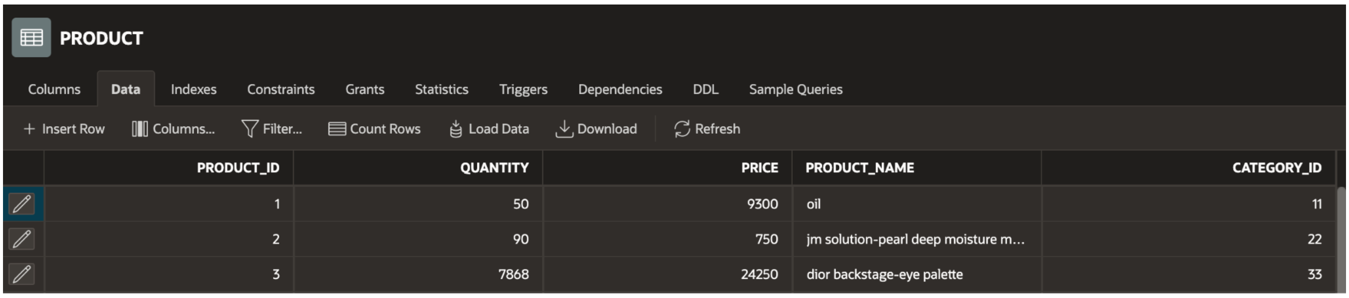


```
DECLARE
   product_names product.product_name%type;
   p_id product.product_id%type := 1;
   lengthOfName NUMBER;
   ex_invalid_name EXCEPTION;
BEGIN
   SELECT product_name INTO product_names
   FROM product
   WHERE product_id = p_id and rownum=1;

   lengthOfName := LENGTH(product_names);

   IF lengthOfName <= 5 THEN
      RAISE ex_invalid_name;
   ELSE
      DBMS_OUTPUT.PUT_LINE('Name of product: ' || product_names);
```

```
The number of letters in the name of product must be less than 5

Statement processed.

0.01 seconds
```

If the length of the product name is less than or equal to 5 characters, the code raises an exception called "ex_invalid_name." Otherwise, it outputs the product name using the DBMS_OUTPUT.PUT_LINE procedure.

The code also includes exception handlers for the "ex_invalid_name" exception, the "no_data_found" exception, and all other exceptions. If the "ex_invalid_name" exception is raised, the code outputs a message indicating that the product name must be less than 5 characters. If the "no_data_found" exception is raised, the code outputs a message indicating that no data was found. For all other exceptions, the code outputs a generic error message.



PRODUCT_ID	QUANTITY	PRICE	PRODUCT_NAME	CATEGORY_ID
1	50	9300	oil	11
2	90	750	jm solution-pearl deep moisture m...	22
3	7868	24250	dior backstage-eye palette	33

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

Code of trigger:

```
create or replace TRIGGER count_rows
BEFORE INSERT ON product
FOR EACH ROW
DECLARE
  count_row NUMBER;
BEGIN
  SELECT COUNT(*) INTO count_row FROM product;
  DBMS_OUTPUT.PUT_LINE('Number of rows before insert in product table: '
     || count_row);
END;
```

Explanation:

Prior to performing an insert operation, the trigger code is intended to retrieve the number of rows currently present in the product table and display it to the console using the DBMS_OUTPUT.PUT_LINE method.

It can be used to keep track of the amount of rows in the table and to monitor changes to the product table. It can be used, for instance, to watch the table's growth rate and predict future resource needs, or to check whether the table has reached a size or limit that can affect performance.

-Transaction:

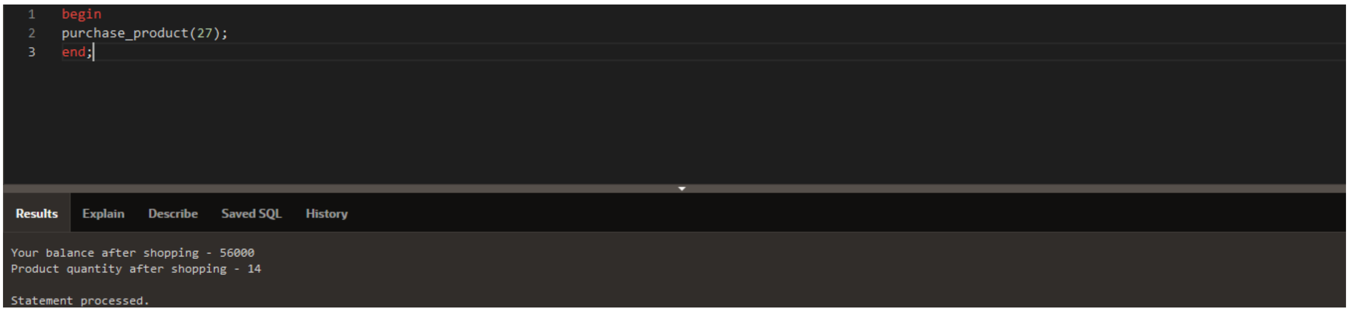
Creating a procedure using a transaction:

```
create or replace PROCEDURE purchase_product (client_id IN NUMBER)
IS
  client_cart_id NUMBER;
  cl_product_id NUMBER;
  cl_quantity NUMBER;
  product_quantity NUMBER;
  cl_price NUMBER;
  client_balance NUMBER;
  cl_total_price NUMBER;
  test_bal number;
  test_qua number;
  ex exception;
  ex_balance exception;
BEGIN

  SELECT product_id INTO cl_product_id FROM cart WHERE buyer_id =
     client_id;

  SELECT balance INTO client_balance FROM card WHERE buyer_id =
     client_id;

  SELECT cart_id INTO client_cart_id FROM cart WHERE buyer_id =
     client_id;

  SELECT quantity INTO cl_quantity FROM cart WHERE product_id =
     cl_product_id;

  SELECT quantity INTO product_quantity FROM product WHERE product_id
     = cl_product_id;

  IF cl_quantity > product_quantity THEN
    RAISE ex;
  END IF;

  SELECT price INTO cl_price FROM product WHERE product_id =
     cl_product_id;
  cl_total_price := cl_quantity * cl_price;

  IF(cl_quantity <= product_quantity and client_balance>= cl_total_price)
      THEN
    UPDATE card SET balance = balance - cl_total_price WHERE buyer_id =
      client_id;
    SELECT balance into test_bal from card where buyer_id = client_id and
      rownum=1;
    dbms_output.put_line('Your balance after shopping - ' || test_bal);
    INSERT INTO order_details (status_of_payment, order_detalis_id, cart_id)
      VALUES ('true', 99999, client_cart_id);
    UPDATE product SET quantity = quantity - cl_quantity WHERE product_id =
      cl_product_id;
    SELECT quantity into test_qua from product where product_id =
      cl_product_id;
    dbms_output.put_line('Product quantity after shopping - ' || test_qua);
  ELSE
    RAISE ex_balance;
  END IF;
  COMMIT;
EXCEPTION
  WHEN ex THEN
    dbms_output.put_line('The product you have selected has run out or you
      have selected more than the quantity of the product');
 WHEN ex_balance THEN
    dbms_output.put_line('Your balance is not enough');
 WHEN OTHERS THEN
    ROLLBACK;
 END;
```

Calling the procedure:

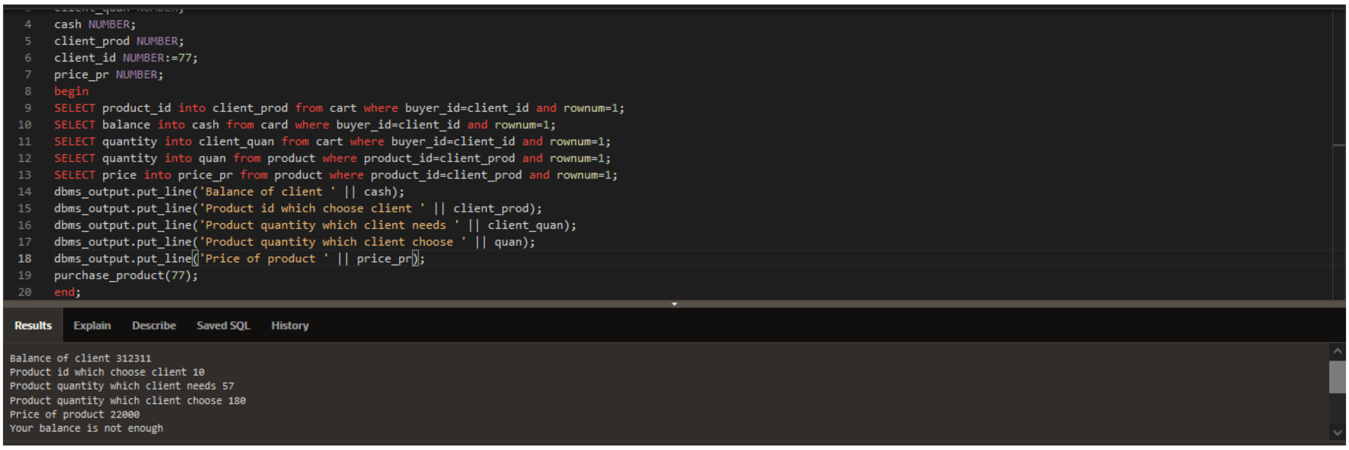
```
begin
        purchase_product(21);
end;
```

Example of procedure and results:



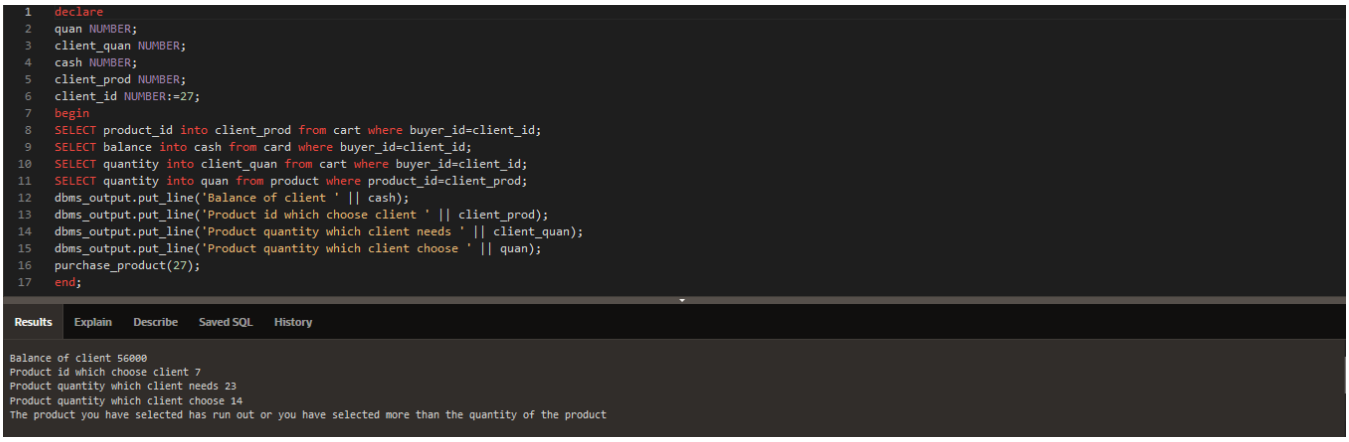
```
begin
purchase_product(27);
end;
```

```
Your balance after shopping - 56000
Product quantity after shopping - 14

Statement processed.
```



```
client_quan NUMBER;
cash NUMBER;
client_prod NUMBER;
client_id NUMBER:=77;
price_pr NUMBER;
begin
SELECT product_id into client_prod from cart where buyer_id=client_id and rownum=1;
SELECT balance into cash from card where buyer_id=client_id and rownum=1;
SELECT quantity into client_quan from cart where buyer_id=client_id and rownum=1;
SELECT quantity into quan from product where product_id=client_prod and rownum=1;
SELECT price into price_pr from product where product_id=client_prod and rownum=1;
dbms_output.put_line('Balance of client ' || cash);
dbms_output.put_line('Product id which choose client ' || client_prod);
dbms_output.put_line('Product quantity which client needs ' || client_quan);
dbms_output.put_line('Product quantity which client choose ' || quan);
dbms_output.put_line('Price of product ' || price_pr);
purchase_product(77);
end;
```

```
Balance of client 312311
Product id which choose client 10
Product quantity which client needs 57
Product quantity which client choose 180
Price of product 22000
Your balance is not enough
```



```
declare
quan NUMBER;
client_quan NUMBER;
cash NUMBER;
client_prod NUMBER;
client_id NUMBER:=27;
begin
SELECT product_id into client_prod from cart where buyer_id=client_id;
SELECT balance into cash from card where buyer_id=client_id;
SELECT quantity into client_quan from cart where buyer_id=client_id;
SELECT quantity into quan from product where product_id=client_prod;
dbms_output.put_line('Balance of client ' || cash);
dbms_output.put_line('Product id which choose client ' || client_prod);
dbms_output.put_line('Product quantity which client needs ' || client_quan);
dbms_output.put_line('Product quantity which client choose ' || quan);
purchase_product(27);
end;
```

```
Balance of client 56000
Product id which choose client 7
Product quantity which client needs 23
Product quantity which client choose 14
The product you have selected has run out or you have selected more than the quantity of the product
```

Explanation of the transaction :

This code is a stored procedure using a transaction to purchase a product. It accepts the buyer's ID as input and uses it to perform several database queries.

First, the procedure selects the product ID from the buyer's basket, and then checks his balance to make sure that he has enough money to make a purchase.

If the customer has enough money, the procedure updates the customer's balance and adds order information to the "order_details" table, and also updates the quantity of goods in the "product" table.

If the customer does not have enough money or the quantity of goods is less than the requested quantity, the procedure generates an exception.

Due to the use of transactions and exceptions, the code also contains commands to roll back the transaction in case of errors.

The procedure outputs information about the customer's balance after purchase and the quantity of goods after purchase to the console using the command "dbms_output.put_line".