**Report**

Introduction to the system

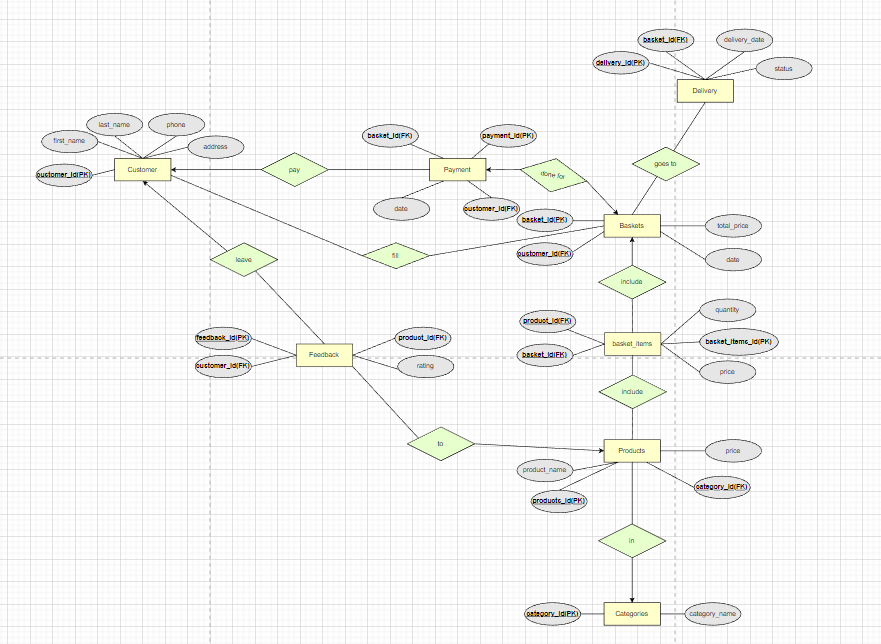
Our online store database is designed to facilitate efficient and secure online ordering of products for customers of all ages. Whether you are looking to purchase electronics, our database provides a user-friendly platform for you to browse products, add them to your basket, and securely process payment.

We Inspired by real-life stores like Sulpak, Technodom, Kaspi.

Our difference from another project, in our project, customers can leave rating the product without buying, but they can still rate if they bought, and edit their feedback

Whether you are a busy professional, a student, or a retiree, our online store database is designed to meet your needs. Shop from the comfort of your own home, and enjoy the convenience of having your purchases delivered right to your doorstep.

ER diagram



**link for erd**: [https://drive.google.com/file/d/1iOxZOQm88byWSkWwK\_Dj9nwf8Su4pCZD/view?usp=sharing](https://app.diagrams.net#G1iOxZOQm88byWSkWwK_Dj9nwf8Su4pCZD)

**ERD explanation**

The ER-diagram consists of 8 entities. The first entity is the Customers table, which stores information about customers such as their unique customer\_id, first name, last name, phone number, and address. This table is linked to the Baskets table through a one-to-many relationship. This means that a single customer can have multiple baskets, but each basket belongs to only one customer. The Baskets table has a foreign key "customer\_id" column that references the primary key "customer\_id" column in the Customers table, establishing the relationship between the two tables.

The Baskets table contains information about each basket, including its unique basket\_id, customer\_id, and date. The table has a one-to-many relationship with the Basket\_items table, which records information about the items included in each basket. Each basket item has a unique basket\_items\_id, quantity, price, and foreign keys "basket\_id" and "product\_id," which reference the "basket\_id" and "product\_id" columns in the Baskets and Products tables, respectively.

The Products table contains information about each product, including its unique product\_id, name, price, and foreign key "category\_id," which references the primary key "category\_id" column in the Categories table. The Products table has a many-to-many relationship with the Basket\_items table, allowing each product to be associated with multiple basket items, and each basket item to contain multiple products.

The Categories table categorizes the products in the Products table and has a one-to-many relationship with the Products table. This means that each category can contain multiple products, but each product belongs to only one category.

The Payments table stores information about each payment made by a customer, including its unique payment\_id, payment\_date, and foreign keys "basket\_id" and "customer\_id," which reference the "basket\_id" and "customer\_id" columns in the Baskets and Customers tables, respectively. The relationship between the Payments table and the Customers table is many-to-one, meaning that a single customer can make multiple payments, but each payment is made by only one customer. The relationship between the Payments table and the Baskets table is "done for," indicating that each payment is made for a specific basket, and each basket can have at most one payment against it.

The Delivery table contains information about each delivery, including its unique delivery\_id, delivery\_date, and status. The Delivery table has a one-to-one relationship with the Baskets table, meaning that each delivery is associated with only one specific basket, and each basket can have at most one delivery. To define a customer in the Delivery table, we can use the customer\_id foreign key column, which references the customer\_id primary key column in the Customers table. When a customer places an order, a new record is added to the Baskets table with the customer\_id value set to the ID of the corresponding customer.

Feedback table: This table contains information about the feedback given by customers for each product. It has a feedback ID, a foreign key to link the feedback to a specific customer, a foreign key to link the feedback to a specific product, and the rating given by the customer.

One-to-many relationship between Feedback and Products: One product can receive feedback from many customers, but each feedback is linked to only one product.

One-to-many relationship between Feedback and Customer: One customer can give feedback on many products, but each feedback is linked to only one customer.

**Normal Form**

To determine if a database schema is in 3rd Normal Form (3NF), we need to ensure that it meets the following conditions:

1.Each table should have a primary key that uniquely identifies each record in the table.

2. All non-key attributes should depend only on the primary key.

3. There should be no transitive dependencies.

Now let's analyze each table in this ER diagram to determine if it is in 3NF:

Customer table: This table has a primary key (customer\_id) that uniquely identifies each record in the table. All non-key attributes (first\_name, last\_name, phone, address) depend only on the primary key. Therefore, this table is in 3NF.

Payment table: This table has a composite primary key (basket\_id, payment\_id) that uniquely identifies each record in the table. All non-key attributes (payment\_date, customer\_id) depend only on the primary key. Therefore, this table is in 3NF.

Baskets table: This table has a primary key (basket\_id) that uniquely identifies each record in the table. All non-key attributes (customer\_id, date) depend only on the primary key. Therefore, this table is in 3NF.

Delivery table: This table has a primary key (delivery\_id) that uniquely identifies each record in the table. All non-key attributes (basket\_id, delivery\_date, status) depend only on the primary key. Therefore, this table is in 3NF.

Basket\_items table: This table has a primary key (basket\_items\_id) that uniquely identifies each record in the table. All non-key attributes (product\_id, basket\_id, quantity, price) depend only on the primary key. Therefore, this table is in 3NF.

Products table: This table has a primary key (product\_id) that uniquely identifies each record in the table. All non-key attributes (product\_name, price, category\_id) depend only on the primary key. Therefore, this table is in 3NF.

Categories table: This table has a primary key (category\_id) that uniquely identifies each record in the table. All non-key attributes (category\_name) depend only on the primary key. Therefore, this table is in 3NF.

Feedback table: This table has a primary key (feedback\_id) that uniquely identifies each record in the table. All non-key attributes (customer\_id, product\_id, rating) depend only on the primary key. Therefore, this table is in 3NF.

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

create or replace TRIGGER trigger\_show\_current\_num\_of\_rows

BEFORE INSERT ON basket\_items

FOR EACH ROW

DECLARE

cnt NUMBER;

BEGIN

SELECT COUNT(\*) INTO cnt FROM basket\_items;

DBMS\_OUTPUT.PUT\_LINE('Current number of rows: ' || cnt);

END;

This is a SQL trigger that is defined using the CREATE OR REPLACE TRIGGER statement. The trigger is named "trigger\_show\_current\_num\_of\_rows" and is defined to execute BEFORE INSERT on the "basket\_items" table.

The FOR EACH ROW clause indicates that the trigger will execute once for every row that is inserted into the "basket\_items" table.

The DECLARE section is used to define a local variable named "cnt" of type NUMBER, which will be used to store the current row count of the "basket\_items" table.

The BEGIN section contains the actual code that will be executed when the trigger is fired. The code first executes a SELECT COUNT(\*) statement to get the current row count of the "basket\_items" table and stores the result in the "cnt" variable.

The DBMS\_OUTPUT.PUT\_LINE function is then called to print a message to the console, indicating the current row count of the table.

**TRIGGER\_BASKET\_FOR\_NEW\_CUSTOMER**

create or replace TRIGGER trigger\_basket\_for\_new\_customer

AFTER INSERT ON Customers

FOR EACH ROW

DECLARE

cnt INTEGER;

BEGIN

SELECT COUNT(\*) + 1 INTO cnt FROM Baskets;

INSERT INTO Baskets(basket\_id, customer\_id, total\_price)

VALUES (cnt, :NEW.customer\_id, 0);

END;

This is a SQL trigger that is designed to create a new basket record for a new customer every time a new row is inserted into the "Customers" table.

The trigger is defined using the CREATE OR REPLACE TRIGGER statement and named "trigger\_basket\_for\_new\_customer". The AFTER INSERT ON clause specifies that the trigger will be executed after each new row is inserted into the "Customers" table.

The DECLARE section defines a local variable named "cnt" of type INTEGER, which will be used to store the new basket ID.

The BEGIN section contains the code that will be executed when the trigger is fired. The code first executes a SELECT COUNT(\*) + 1 statement to get the current number of baskets in the "Baskets" table, and adds 1 to it to obtain the new basket ID. The result is stored in the "cnt" variable.

The code then executes an INSERT INTO statement to insert a new record into the "Baskets" table, with the basket ID set to the value stored in the "cnt" variable, the customer ID set to the new customer's ID (which is available through the ":NEW" keyword), and the total price set to 0.

**TRIGGER\_INSERT\_TOTAL\_PRICE\_BASKET\_ITEM**

create or replace TRIGGER trigger\_insert\_total\_price\_basket\_item

BEFORE INSERT or UPDATE ON basket\_items

FOR EACH ROW

DECLARE

price DECIMAL(10, 2);

BEGIN

SELECT price INTO price FROM Products WHERE product\_id = :NEW.product\_id;

:NEW.total\_price := price \* :NEW.quantity;

END;

this trigger calculates and sets the total price for each new or updated row in the "basket\_items" table, based on the product price and quantity. It does this by retrieving the product price from the "Products" table using the product ID stored in the ":NEW" keyword, and then calculating the total price and setting the value of the "total\_price" column in the ":NEW" keyword to the calculated value.

**TRIGGER\_UPDATE\_BASKETS\_TOTAL\_PRICE**

create or replace TRIGGER trigger\_update\_baskets\_total\_price

FOR INSERT OR UPDATE OR DELETE ON basket\_items

COMPOUND TRIGGER

TYPE coll IS TABLE OF Baskets.basket\_id%TYPE;

basket\_ids coll := coll();

BEFORE STATEMENT IS BEGIN

basket\_ids.DELETE;

END BEFORE STATEMENT;

AFTER EACH ROW IS BEGIN

IF :NEW.basket\_id NOT MEMBER OF basket\_ids THEN

basket\_ids.EXTEND;

basket\_ids(basket\_ids.LAST) := :NEW.basket\_id;

END IF;

END AFTER EACH ROW;

AFTER STATEMENT IS BEGIN

FOR i IN 1..basket\_ids.COUNT LOOP

UPDATE Baskets SET total\_price = (

SELECT SUM(total\_price) FROM basket\_items WHERE basket\_id = basket\_ids(i)

) WHERE basket\_id = basket\_ids(i);

END LOOP;

END AFTER STATEMENT;

END;

This is a SQL compound trigger that is designed to update the total price of each basket in the "Baskets" table whenever a new row is inserted into, updated in, or deleted from the "basket\_items" table.

A compound trigger is used when you need to define multiple trigger timing points (BEFORE, AFTER) and have multiple trigger events (INSERT, UPDATE, DELETE) on a single table.

The trigger is defined using the CREATE OR REPLACE TRIGGER statement and named "trigger\_update\_baskets\_total\_price". The FOR INSERT OR UPDATE OR DELETE ON clause specifies that the trigger will be executed for each insert, update, or delete operation on the "basket\_items" table.

The TYPE declaration defines a collection of the "basket\_id" column data type in the "Baskets" table, which will be used to store the basket IDs affected by the trigger.

The BEGIN section defines the compound trigger, which contains multiple timing points.

The BEFORE STATEMENT timing point is used to clear the "basket\_ids" collection before each trigger execution.

The AFTER EACH ROW timing point is used to add the basket ID of each affected row to the "basket\_ids" collection, if it is not already present.

The AFTER STATEMENT timing point is used to update the total price of each basket in the "Baskets" table, based on the total price of its corresponding rows in the "basket\_items" table. It loops through each unique basket ID in the "basket\_ids" collection, and performs an UPDATE statement on the "Baskets" table to set its total price to the sum of the total prices of its corresponding rows in the "basket\_items" table.

**A check**

create or replace TRIGGER trigger\_show\_total\_price\_payment

after INSERT ON payment

FOR EACH ROW

DECLARE

total\_p NUMBER;

cust\_id number;

BEGIN

SELECT total\_price INTO total\_p FROM baskets where :new.basket\_id = basket\_id;

SELECT customer\_id INTO cust\_id from customers where :new.customer\_id = customer\_id;

DBMS\_OUTPUT.PUT\_LINE(cust\_id || ' total price: ' || total\_p);

END;

This trigger works like check, named trigger\_show\_total\_price\_payment, which is fired automatically by the database system after an INSERT operation is performed on the payment table. The trigger is defined to execute FOR EACH ROW inserted into the table.

The purpose of this trigger is to calculate and display the total price of a basket for a given customer when a new payment is made. The trigger first declares two local variables total\_p and cust\_id of type NUMBER. The variable total\_p will be used to store the total price of the basket, while the variable cust\_id will hold the customer ID of the customer making the payment.

The trigger then uses two SELECT statements to retrieve the values of total\_price and customer\_id from the baskets and customers tables, respectively, based on the basket\_id and customer\_id values in the newly inserted row :new. The values retrieved by these statements are stored in the total\_p and cust\_id variables.

Finally, the trigger uses the DBMS\_OUTPUT.PUT\_LINE procedure to display the total price of the basket for the given customer ID. The output message will be written to the output buffer and can be viewed in the console output of the tool which is executing the SQL code.

**Function which counts the number of records**

create or replace FUNCTION count\_records (

table\_name IN VARCHAR2

) RETURN NUMBER

IS

record\_count NUMBER;

BEGIN

EXECUTE IMMEDIATE 'SELECT COUNT(\*) FROM ' || table\_name INTO record\_count;

RETURN record\_count;

END;

This is a PL/SQL function named "count\_records" that takes a single input parameter "table\_name" of type VARCHAR2 and returns a number. The function is defined to be created or replaced (if it already exists) using the "CREATE OR REPLACE FUNCTION" statement.

The function uses dynamic SQL to execute a SELECT statement that counts the number of records in the table specified by the input parameter "table\_name". The "EXECUTE IMMEDIATE" statement is used to execute the SELECT statement as a string, and the resulting count value is stored in the "record\_count" variable using the "INTO" clause.

The function returns the "record\_count" value, which represents the number of records in the specified table.

This function is useful when the name of the table whose record count is needed is not known at compile time, but rather at run time. The function allows you to pass the table name as a parameter to the function, and the function dynamically generates the SQL query to count the records in the table.

**User-defined exception which disallows to enter title of item to be less than 5 characters.**

create or replace TRIGGER trigger\_exception\_title

BEFORE INSERT ON products

FOR EACH ROW

DECLARE

name\_length EXCEPTION;

BEGIN

IF LENGTH(:new.product\_name) < 5 THEN

RAISE name\_length;

ELSE

DBMS\_OUTPUT.PUT\_LINE('Item added successfully.');

END IF;

EXCEPTION

WHEN name\_length THEN

RAISE\_APPLICATION\_ERROR(-20001, 'Error: Product name must be at least 5 characters');

WHEN OTHERS THEN

DBMS\_OUTPUT.PUT\_LINE('Error: ' || SQLERRM);

END;

This is a PL/SQL trigger named "trigger\_exception\_title" that is defined to be fired before inserting a new record into the "products" table. The trigger is created or replaced (if it already exists) using the "CREATE OR REPLACE TRIGGER" statement.

The trigger is defined to be executed for each row that is being inserted into the "products" table. It declares an exception "name\_length" that is raised when the length of the "product\_name" column of the new row being inserted is less than 5 characters.

Inside the trigger body, an "IF-ELSE" statement checks the length of the "product\_name" column of the new row. If the length is less than 5 characters, then it raises the "name\_length" exception. Otherwise, it prints a message to the console indicating that the item has been added successfully.

In the exception block, the "WHEN" clause is used to catch the "name\_length" exception and raise an application error with the message "Error: Product name must be at least 5 characters". If any other exceptions occur, the "WHEN OTHERS" clause is used to catch them and print their error message to the console using the "SQLERRM" function.

**Procedure which does group by information**

create or replace PROCEDURE group\_basket\_items\_info IS

CURSOR basket\_items\_cur IS

SELECT basket\_id, SUM(quantity) AS total\_quantity, SUM(total\_price) AS total\_price

FROM basket\_items

GROUP BY basket\_id;

BEGIN

FOR basket\_item\_rec IN basket\_items\_cur LOOP

DBMS\_OUTPUT.PUT\_LINE('Basket ID: ' || basket\_item\_rec.basket\_id ||

' | Total Quantity: ' || basket\_item\_rec.total\_quantity ||

' | Total Price: ' || basket\_item\_rec.total\_price);

END LOOP;

END;

The procedure uses a cursor to select data from the "basket\_items" table, sums up the quantity and total price columns for each basket, and then loops through the results and displays the basket ID, total quantity, and total price for each basket item using the DBMS\_OUTPUT.PUT\_LINE function. The final block of code calls the procedure to execute it and display the results.

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

create or replace PROCEDURE update\_customers

(

p\_id IN NUMBER,

p\_value IN VARCHAR2

)

IS

row\_count INTEGER;

BEGIN

UPDATE customers SET first\_name = p\_value WHERE customer\_id = p\_id;

row\_count := SQL%ROWCOUNT;

IF row\_count = 0 THEN

dbms\_output.put\_line('No rows updated.');

ELSIF row\_count = 1 THEN

dbms\_output.put\_line('1 row updated.');

ELSE

dbms\_output.put\_line(row\_count || ' rows updated.');

END IF;

END;

That PL/SQL procedure named "update\_customers" takes two input parameters, an ID number and a string value. It updates the "first\_name" column of the "customers" table with the provided string value for the customer with the specified ID number. It then retrieves the number of rows affected by the update and uses an IF-ELSE statement to print out a message indicating the number of rows updated.

The final block of code calls the procedure with an example ID number and string value to update the corresponding customer's first name.