REPORT

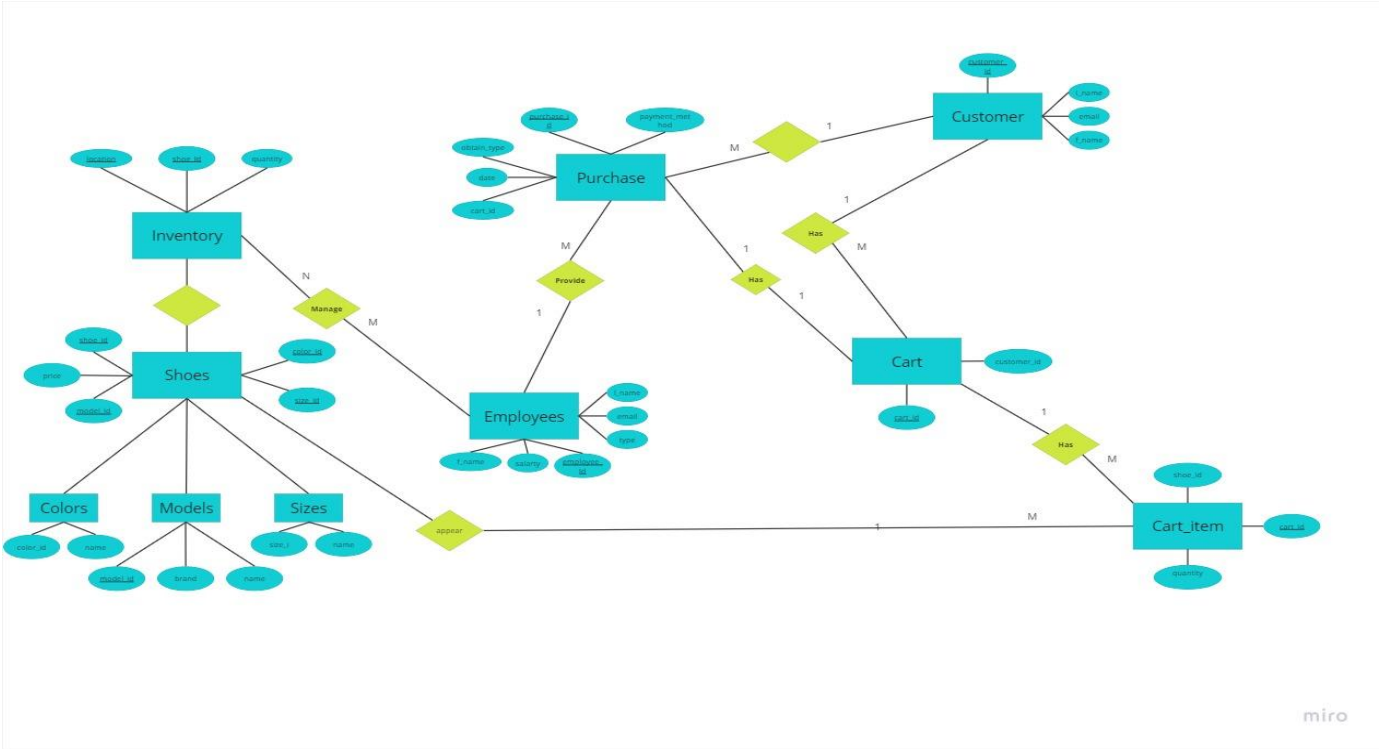
DBMS-2 END TERM PROJECT
ONLINE SHOE SHOP

TEAM MEMBERS:

210103314 | Daniil Lyapin
210103129 | Aldiyar Zhaksylyk
210103318 | Kabylbekov Arsen

Introduction to the system

ER diagram:



Entities: Customer, Purchase, Employees, Inventory, Shoes, Cart, Cart_Item, Colors, Models, Sizes.

This is the ER diagram of our project. The project is an online shoe store. On the diagram, you can see all entities and their attributes.The concept of the database is simple: the customer goes to the page of the online shop, selects shoes from the Shoes table and adds it to the cart. Each new cart item (shoes) is saved separately in Cart_item for each cart. After that, the customer enters data about himself and begins ordering, this is stored in the Customer and Transaction tables. The Employees table stores data about employees. Information about shoes is stored in different tables, such as: Storage, Shoes, Model, Sizes and Colors. This is due to the normalization of tables.

Tables

Customer(customer_ID(pk), first_name, last_name, email)

CUSTOMER_ID	FIRST_NAME	LAST_NAME	EMAIL
1	Gervonta	Davies	gervonta@gmail.com
2	Breaking	Bad	walterwhite@gmail.com
3	Yaslan	Ruchanov	yaslan@gmail.com
4	Daniil	Lyapin	vasya@gmail.com
5	Sultan	Damir	damir@gmail.com
6	Narken	Miras	miras@gmail.com

The given customer information table appears to be in the Third Normal Form (3NF), which means that it satisfies the rules of 1NF and 2NF and also satisfies the following additional rule:

- Every non-key attribute (column) in the table is dependent only on the primary key column(s) and not on any other non-key column.

In this table, we can see that the primary key is the `CUSTOMER_ID` column, and each row is uniquely identified by this column.

The remaining columns (`FIRST_NAME`, `LAST_NAME`, and `EMAIL`) are non-key attributes, and each of them is dependent only on the `CUSTOMER_ID` column and not on any other non-key column.

This table is already in 3NF because all columns are atomic and there are no repeating groups. The `FIRST_NAME`, `LAST_NAME`, and `EMAIL` columns are not repeating groups because they are each dependent on the `CUSTOMER_ID` primary key column. Therefore, this table is properly normalized in 3NF.

Cart(cart_id(pk), customer_id(fk))

CART_ID	CUSTOMER_ID
1	1
2	2
3	3
4	4
5	5
6	6

The provided `Cart` table has two columns: `cart_id` and `customer_id`, where `cart_id` is the primary key and `customer_id` is a foreign key referencing the `CUSTOMER_ID` primary key in the `Customer` table.

This table is already in the first normal form (1NF) because it has a primary key column (`cart_id`) that uniquely identifies each row.

This table is also in the second normal form (2NF) because it meets the requirement that every non-key attribute (column) in the table is fully dependent on the primary key. In this case, there is only one non-key attribute, `customer_id`, and it is dependent on the primary key column `cart_id`.

Regarding the Third Normal Form (3NF), there is no transitive dependency in this table. Therefore, it is already in the Third Normal Form.

In summary, the `Cart` table is properly normalized in 1NF, 2NF, and 3NF.

Employees(employee_id(pk), first_name, email, type, salary)

EMPLOYEE_ID	FIRST_NAME	EMAIL	TYPEL	SALARY
1	Aldiyar	aldiyar@gmail.com	admin	4000
2	Daniil	danya@gmail.com	admin	4000
3	Ilyar	ilyar@gmail.com	courier	500
4	Talantbek	talantbek@gmail.com	courier	500
5	Aidar	aidar@gmail.com	courier	500
6	Diyas	diyas@gmail.com	driver	800
7	Akzhan	akzhan@gmail.com	waitress	100
8	Madi	madi@gmail.com	security	800
9	Miras	yahagi@gmail.com	SMM	2000
10	Aman	aman@gmail.com	security	800

The provided `Employees` table has five columns: `employee_id`, `first_name`, `email`, `type`, and `salary`, where `employee_id` is the primary key column.

This table is already in the first normal form (1NF) because it has a primary key column (`employee_id`) that uniquely identifies each row.

`The primary key column is `employee_id`, which uniquely identifies each row in the table. The remaining columns (`first_name`, `email`, `type`, and `salary`) are non-key attributes, and each of them is dependent only on the primary key column `employee_id`.

This table satisfies all the rules of the Third Normal Form (3NF) because every non-key attribute is dependent only on the primary key column and there is no transitive dependency between the non-key attributes.

Therefore, the given `Employees` table, as it is in the Third Normal Form, is properly normalized.

Cart_Item(cart_id(pk, fk), shoe_id(pk, fk), quantity)

CART_ID	SHOE_ID	QUANTITY
1	4	1
1	3	1
2	6	2
3	1	1
4	8	1
4	6	1
5	2	1
6	7	1
6	4	1
6	3	1

The `Cart_Item` table has a primary key consisting of two columns: `cart_id` and `shoe_id`. Additionally, the table has a third column, `quantity`, which is dependent on both primary key columns.

The table is already in the first normal form (1NF) because it has a primary key that uniquely identifies each row.

The table is also in the second normal form (2NF) because every non-key attribute (`quantity`) is fully dependent on both primary key columns (`cart_id` and `shoe_id`).

Furthermore, the table is in the Third Normal Form (3NF) because there is no transitive dependency in this table. This means that all non-key attributes (`quantity`) are directly dependent on the primary key columns (`cart_id` and `shoe_id`) and not on any other non-key attributes.

In conclusion, the `Cart_Item` table is properly normalized in 1NF, 2NF, and 3NF, making it an effective and efficient way to represent items added to a cart in a relational database.

Purchase(purchase_id(pk), cart_id(fk), payment_method, date, obtain_type)

PURCHASE_ID	CART_ID	PAYMENT_METHOD	PURCHASE_DATE	OBTAIN_TYPE
1	1	kaspi	02/20/2023	delievery
2	2	halyk	02/25/2023	pickup
3	3	kaspi	03/12/2023	pickup
4	4	cash	03/24/2023	delievery
5	5	cash	03/26/2023	pickup
6	6	halyk	03/26/2023	delievery

The provided `Purchase` table has five columns: `purchase_id`, `cart_id`, `payment_method`, `date`, and `obtain_type`.

The table is already in the first normal form (1NF) because it has a primary key (`purchase_id`) that uniquely identifies each row.

The `Purchase` table is also in the second normal form (2NF) because every non-key attribute (`payment_method`, `date`, and `obtain_type`) is fully dependent on the primary key column `purchase_id`.

The `Purchase` table would have the columns `purchase_id`, `cart_id` (as a foreign key to `Cart`), `payment_method`, and `date`. This would remove the potential transitive dependency in the `Purchase` table, as `obtain_type` is now dependent only on `cart_id` and is stored in the `Cart` table.

Therefore, in this case, the `Purchase` table can be normalized in 3NF, as it has a primary key (`purchase_id`), every non-key attribute is fully dependent on the primary key, and there are no transitive dependencies.

Inventory(shoe_id(pk, fk), quantity)

SHOE_ID	QUANTITY
1	7
2	10
3	5
4	4
5	7
6	16
7	12
8	15
9	8
10	9

The table is already in the first normal form (1NF) because it has a primary key (`shoe_id`) that uniquely identifies each row.

The `Inventory` table is also in the second normal form (2NF) because there is only one non-key attribute (`quantity`), which is fully dependent on the primary key column `shoe_id`.

Regarding the Third Normal Form (3NF), the `Inventory` table is already in 3NF. There are no other non-key attributes or dependencies that could cause a transitive dependency or other normalization issues.

Therefore, the `Inventory` table is already normalized in Third Normal Form (3NF).

Shoes(shoe_id(pk), model_id(fk), color_id(fk), size_id(fk), price)

SHOE_ID	MODEL_ID	COLOR_ID	SIZE_ID	PRICE
1	1	2	5	100
2	1	2	6	100
3	2	6	3	240
4	2	7	4	240
5	3	3	5	170
6	3	2	9	180
7	4	9	2	120
8	5	1	2	130
9	6	4	5	140
10	6	2	6	140
11	7	5	8	200
12	7	5	9	200
13	8	8	4	300
14	8	7	5	300
15	9	3	7	60
16	10	7	6	160

The provided `Shoes` table is already in 3NF. Here is why:

1. First Normal Form (1NF): The table has a primary key, `shoe_id`, which uniquely identifies each row in the table.

2. Second Normal Form (2NF): All non-key attributes are fully dependent on the primary key. There are no partial dependencies, i.e., each non-key attribute is dependent on the whole primary key. The non-key attributes (`model_id`, `color_id`, `size_id`, `price`) are all simple attributes and not composite, so there are no combinations of non-key attributes that could form a partial dependency.

3. Third Normal Form (3NF): There are no transitive dependencies between the non-key attributes. Each non-key attribute is dependent only on the primary key, and there are no dependencies between the non-key attributes themselves.

Colors(color_id(pk), name)

COLOR_ID	NAME
1	White
2	Black
3	Red
4	Yellow
5	Purple
6	Green
7	Blue
8	Orange
9	Mixed

Sizes(size_id(pk), name)

SIZE_ID	NAME
1	37
2	38
3	39
4	39,5
5	40
6	40,5
7	41
8	41,5
9	42

Models(model_id(pk), brand, name)

MODEL_ID	BRAND	NAME
1	Nike	Air Force 1
2	Nike	Huarache
3	Nike	Air Max 95
4	Nike	Air Max 97
5	Nike	Air Jordan 1
6	Adidas	Ozweego
7	Nike	Air Monarch IV
8	Barhatniye	Tyagi
9	Adibas	Makasin
10	Nike	Air Skepta

‘Models`, `Colors`, and `Sizes` tables are already in 3NF. Here is why:

1. First Normal Form (1NF): Each table has a primary key (`model_id` for `Models`, `color_id` for `Colors`, and `size_id` for `Sizes`) that uniquely identifies each row in the table.

2. Second Normal Form (2NF): All non-key attributes are fully dependent on the primary key. There are no partial dependencies, i.e., each non-key attribute is dependent on the whole primary key. For each table, there is only one non-key attribute (`name`), and it is a simple attribute and not composite, so there are no combinations of non-key attributes that could form a partial dependency.

3. Third Normal Form (3NF): There are no transitive dependencies between the non-key attributes. Each non-key attribute is dependent only on the primary key, and there are no dependencies between the non-key attributes themselves.

Coding part

Function which counts the number of records:

```
CREATE OR REPLACE FUNCTION count_records(table_name IN VARCHAR2)
    RETURN NUMBER
    IS
    total_count NUMBER;
BEGIN
  EXECUTE IMMEDIATE 'SELECT COUNT(*) FROM ' || table_name INTO total_count;
  RETURN total_count;
END;

Select count_records('shoes') FROM DUAL
```

The function starts by declaring a local variable `total_count` of type `NUMBER`, which will hold the result of the query. It then executes a dynamic SQL statement using the `EXECUTE IMMEDIATE` statement, which concatenates the `table_name` parameter with a SELECT COUNT(*) statement to generate a query to count the number of records in the specified table.

The result of the query is stored in the `total_count` variable using the `INTO` clause, which assigns the result of the query to the specified variable.

Finally, the function returns the value of `total_count`.

To use the function, you can call it in a SQL query like this: `SELECT count_records('shoes') FROM DUAL`. This will return the number of records in the `shoes` table.

Overall, this function allows you to easily count the number of records in any table by passing its name as a parameter to the function.

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

```
CREATE OR REPLACE TRIGGER email_check
BEFORE INSERT ON customer
FOR EACH ROW
DECLARE
  email_error EXCEPTION;
BEGIN
  IF LENGTH(:NEW.email) < 5 OR INSTR(:NEW.email, '@') = 0 THEN
    RAISE email_error;
  END IF;
EXCEPTION
  WHEN email_error THEN
    RAISE_APPLICATION_ERROR(-20001, 'Email must be at least 5 characters and contain
"@" in it.');
END;
```

This is a trigger named "email_check" that is created for the "customer" table. It is executed before each insert operation on the "customer" table for each row being inserted.

The trigger checks the length and format of the email address provided for the new row being inserted. If the email address is less than 5 characters or does not contain the "@" symbol, the trigger raises an exception named "email_error".

In the exception block, the trigger raises an application error with the message "Email must be at least 5 characters and contain "@" in it." with a code of -20001.

Trigger is used to enforce a business rule for the email address of the customers in the "customer" table, ensuring that the email address is at least 5 characters long and contains the "@" symbol. If these conditions are not met, the trigger will prevent the insertion of the row and raise an error.

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

```
CREATE OR REPLACE PROCEDURE count_monthly_purchases AS
  rows_inserted NUMBER;
BEGIN

  FOR m IN 1..12 LOOP

    DECLARE
      start_date DATE := TO_DATE('01-' || m || '-' || EXTRACT(YEAR FROM SYSDATE),
'DD-MM-YYYY');
      end_date DATE := LAST_DAY(start_date);
    BEGIN

      DECLARE
        cur SYS_REFCURSOR;
        count_val NUMBER;
        price_val NUMBER;
      BEGIN
        OPEN cur FOR
          'SELECT COUNT(*), SUM(s.price) FROM Purchase p, Cart_Item ci, Shoes s ' ||
          'WHERE p.cart_id = ci.cart_id AND ci.shoe_id = s.shoe_id AND p.purchase_date >=
:start_date AND p.purchase_date <= :end_date'
          USING start_date, end_date;


        FETCH cur INTO count_val, price_val;
        CLOSE cur;


        EXECUTE IMMEDIATE 'INSERT INTO Monthly_Purchases (Mesyac, Purchase_Count,
Total_Price) ' ||
                          'VALUES (:mesyac, :count_val, :price_val)' USING TO_CHAR(start_date,
'Month YYYY'), count_val, price_val;
        rows_inserted := SQL%ROWCOUNT;
        DBMS_OUTPUT.PUT_LINE(rows_inserted || ' row(s) inserted into Monthly_Purchases
table for mesyac ' || TO_CHAR(start_date, 'Month YYYY') || '.');
      END;
    END;
  END LOOP;

  DBMS_OUTPUT.PUT_LINE('Monthly purchase counts and totals have been calculated and
stored in the Monthly_Purchases table.');
END;
```

This is a procedure named "count_monthly_purchases". The procedure calculates the number of purchases and total price of shoes bought each month and stores the results in a table called "Monthly_Purchases".

The procedure uses a loop to iterate over the 12 months of a year. For each month, it calculates the start and end date, and then uses a cursor to retrieve the count of purchases and total price of shoes bought within that month. It then uses dynamic SQL to insert the results into the "Monthly_Purchases" table.

The procedure also uses the DBMS_OUTPUT.PUT_LINE statement to print the number of rows inserted for each month and a message to indicate when the procedure has completed running.

Overall, this procedure is used to automate the process of counting monthly purchases and storing the results in a separate table for analysis or reporting purposes.

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

```
CREATE OR REPLACE TRIGGER purchase_count_trigger
BEFORE INSERT ON Purchase
FOR EACH ROW
DECLARE
  cnt NUMBER;
BEGIN
  SELECT COUNT(*) INTO cnt
    FROM Purchase
    WHERE TRUNC(purchase_date) = TRUNC(SYSDATE);

    IF cnt >= 2 AND :NEW.obtain_type = 'delievery' THEN
        DBMS_OUTPUT.PUT_LINE('We cannot deliver today, today we already have ' || cnt || '
orders to deliver.');
    END IF;
END;
```

This is a trigger that is executed before an insert operation on the Purchase table. It checks the count of purchases made on the current day and if there are already 2 or more purchases made for the current day and the new purchase is for delivery, it raises a message indicating that delivery is not possible for the day.

The trigger works as follows:
- The trigger is executed for each row being inserted into the Purchase table.
- It retrieves the count of purchases made on the current day using the TRUNC function to get only the date part of the purchase_date column and compares it with the current date obtained from the SYSDATE function.
- If the count of purchases made on the current day is 2 or more and the obtain_type of the new purchase is 'delivery', it raises a message using the DBMS_OUTPUT.PUT_LINE function indicating that delivery is not possible for the day.

Procedure which does group by information

```
CREATE OR REPLACE PROCEDURE group_by_info
IS
BEGIN
  FOR rec IN (
    SELECT type, COUNT(*) AS num_employees
    FROM employees
    GROUP BY type
  )
  LOOP
    DBMS_OUTPUT.PUT_LINE('Type: ' || rec.type || ', Number of Employees: ' ||
rec.num_employees);
  END LOOP;
END;
```

This is a stored procedure named `group_by_info` that uses a cursor to iterate over a result set of an SQL query. The query selects the `type` column and the count of employees grouped by `type` from a table called `employees`.

The stored procedure then loops through each record in the result set using the `FOR` loop, and outputs the values of the `type` and `num_employees` fields for each record using the `DBMS_OUTPUT.PUT_LINE` procedure.

Overall, this stored procedure can be used to get a summary of the number of employees in each `type` category in the `employees` table.