# Задание БД № 5


# Явор Йорданов Чамов

# №: 21621577

# Курс: 3

# Група: 1 б

# Специалност: СИТ

**Задание ПБД** **5**

Да се проектира и реализира база от данни за **МАГАЗИН**, която да съхранява следната информация:

- *Продукт* – наименование, група, цена
- *Служител* – име, позиция, телефон
- *Клиент* - име, телефон
- *Продажба* – продукт, клиент, служител, дата, цена

Правила:

- Всеки продукт принадлежи на една група,
- Всеки клиент може да закупи много продукти.

Базата от данни трябва да е **НОРМАЛИЗИРАНА** и да позволява:

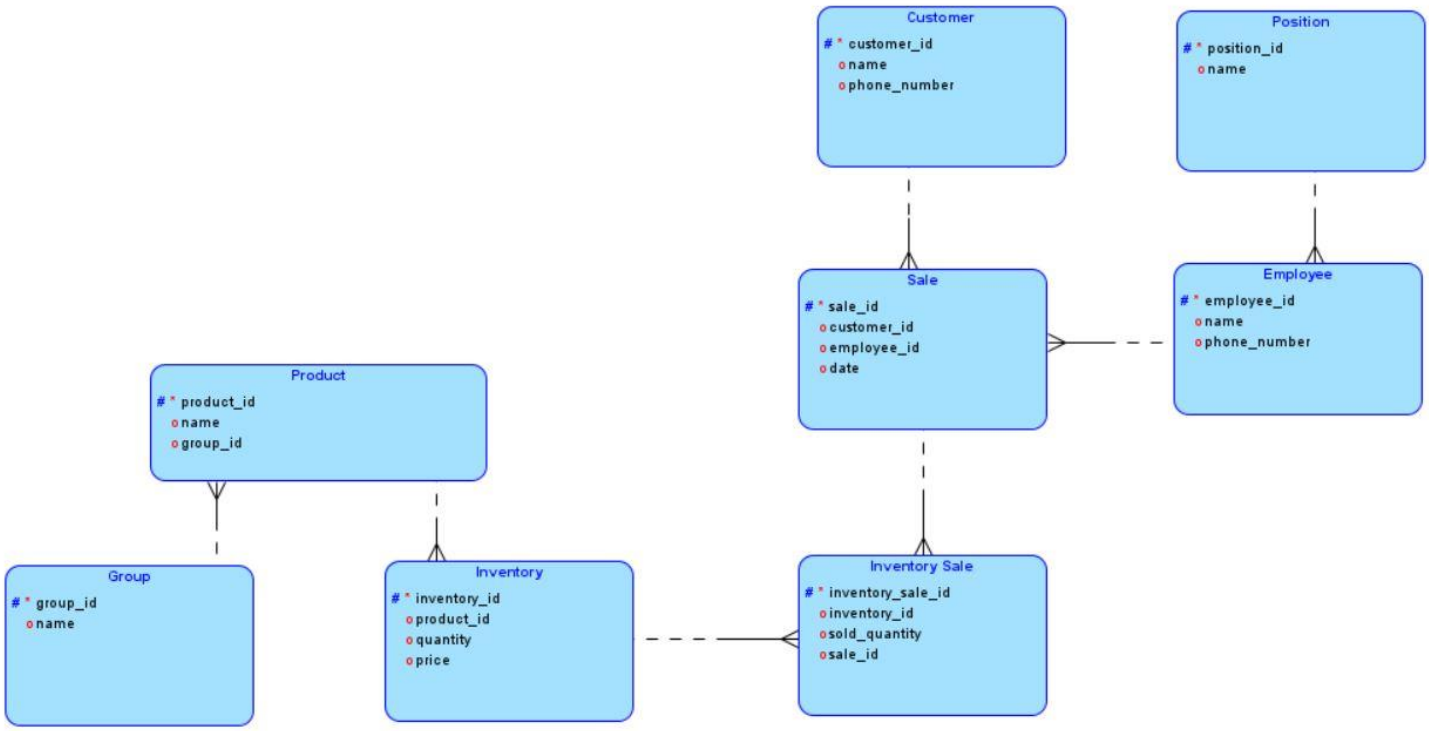
1. **Въвеждане и корекции** на данни.
2. **Търсене/закупуване** на продукти по: цена, наименование, група.
3. **Справки** за:

- продажби за период,
- продажби за служител, подредени по дата,
- продажби за клиент.

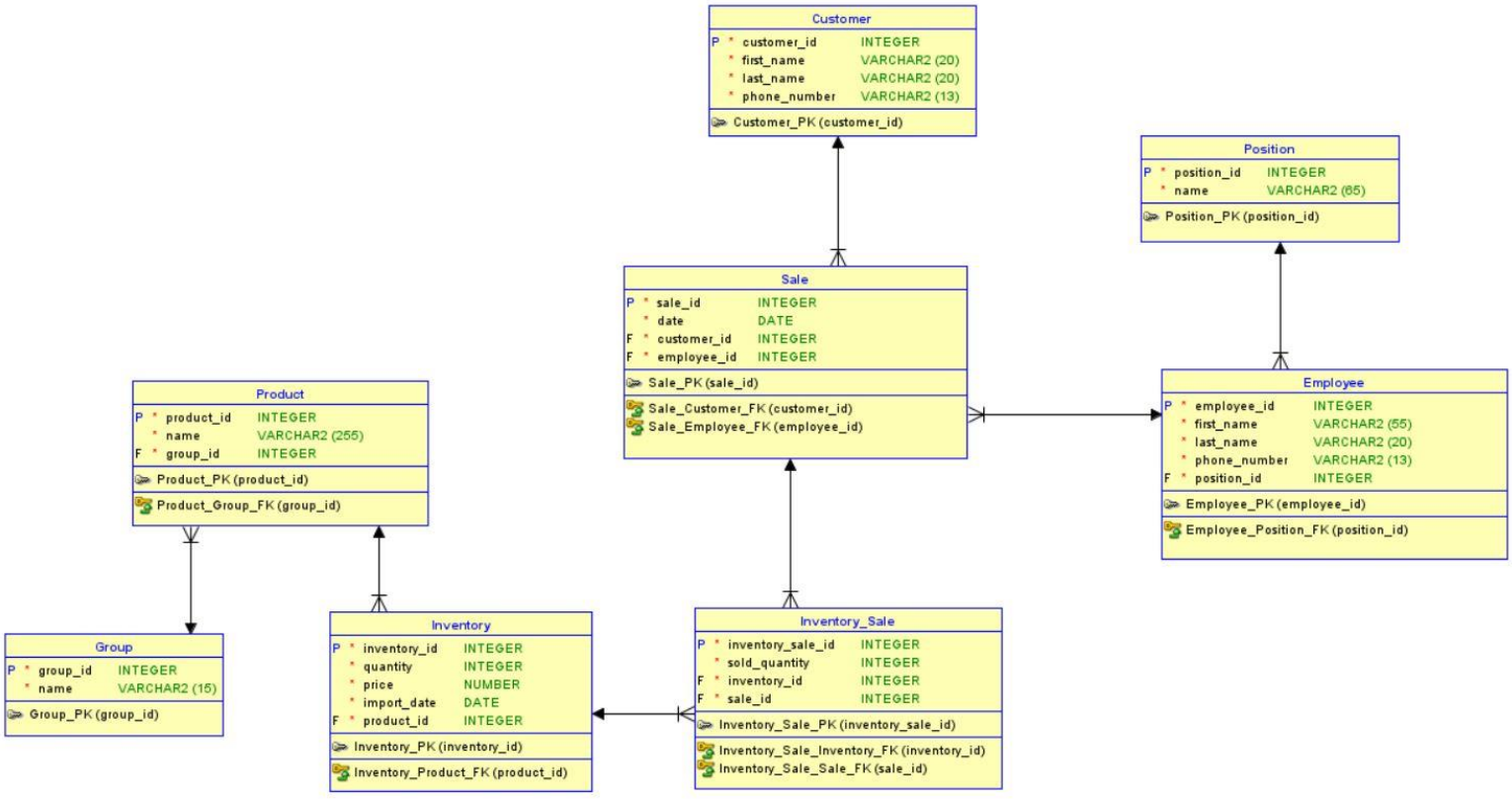
Документацията към реализирания проект трябва да съдържа:

- Задание
- Модели (Oracle Data Modeler) 4 седм./10 т.
- SQL команди – DDL, DML 7 седм./5 т.
- PL/SQL процедури/тригери/курсори 13 седм./25 т.
  докум./10 т.

# Модели (*Oracle Data Modeler*)



# Модели (*Oracle Data Modeler*)



# *SQL* команди – *DDL*

## Таблици и връзки между тях.

```
CREATE TABLE customer (
    customer_id  INTEGER NOT NULL,
    first_name   VARCHAR2(20) NOT NULL,
    last_name    VARCHAR2(20) NOT NULL,
    phone_number VARCHAR2(13) NOT NULL UNIQUE
);

ALTER TABLE customer ADD CONSTRAINT customer_pk PRIMARY KEY ( customer_id );

CREATE TABLE employee (
    employee_id  INTEGER NOT NULL,
    first_name   VARCHAR2(55) NOT NULL,
    last_name    VARCHAR2(20) NOT NULL,
    phone_number VARCHAR2(13) NOT NULL UNIQUE,
    position_id  INTEGER NOT NULL
);

ALTER TABLE employee ADD CONSTRAINT employee_pk PRIMARY KEY ( employee_id );

CREATE TABLE "GROUP" (
    group_id INTEGER NOT NULL,
    name     VARCHAR2(15) NOT NULL UNIQUE
);

ALTER TABLE "GROUP" ADD CONSTRAINT group_pk PRIMARY KEY ( group_id );

CREATE TABLE inventory (
    inventory_id INTEGER NOT NULL,
    quantity     INTEGER NOT NULL,
    price        NUMBER NOT NULL,
    import_date  DATE NOT NULL,
    product_id   INTEGER NOT NULL
);

ALTER TABLE inventory ADD CONSTRAINT inventory_pk PRIMARY KEY ( inventory_id
);

CREATE TABLE inventory_sale (
    inventory_sale_id INTEGER NOT NULL,
    sold_quantity     INTEGER NOT NULL,
    inventory_id      INTEGER NOT NULL,
    sale_id           INTEGER NOT NULL
);
```

```
ALTER TABLE inventory_sale ADD CONSTRAINT inventory_sale_pk PRIMARY KEY (
inventory_sale_id );

CREATE TABLE position (
    position_id INTEGER NOT NULL,
    name        VARCHAR2(65) NOT NULL UNIQUE
);

ALTER TABLE position ADD CONSTRAINT position_pk PRIMARY KEY ( position_id );

CREATE TABLE product (
    product_id INTEGER NOT NULL,
    name       VARCHAR2(255) NOT NULL,
    group_id   INTEGER NOT NULL
);

ALTER TABLE product ADD CONSTRAINT product_pk PRIMARY KEY ( product_id );

CREATE TABLE sale (
    sale_id     INTEGER NOT NULL,
    "date"      DATE NOT NULL,
    customer_id INTEGER NOT NULL,
    employee_id INTEGER NOT NULL
);

ALTER TABLE sale ADD CONSTRAINT sale_pk PRIMARY KEY ( sale_id );

ALTER TABLE employee
    ADD CONSTRAINT employee_position_fk FOREIGN KEY ( position_id )
        REFERENCES position ( position_id );

ALTER TABLE inventory
    ADD CONSTRAINT inventory_product_fk FOREIGN KEY ( product_id )
        REFERENCES product ( product_id );

ALTER TABLE inventory_sale
    ADD CONSTRAINT inventory_sale_inventory_fk FOREIGN KEY ( inventory_id )
        REFERENCES inventory ( inventory_id );

ALTER TABLE inventory_sale
    ADD CONSTRAINT inventory_sale_sale_fk FOREIGN KEY ( sale_id )
        REFERENCES sale ( sale_id );

ALTER TABLE product
    ADD CONSTRAINT product_group_fk FOREIGN KEY ( group_id )
        REFERENCES "GROUP" ( group_id );

ALTER TABLE sale
    ADD CONSTRAINT sale_customer_fk FOREIGN KEY ( customer_id )
```

```
        REFERENCES customer ( customer_id );

ALTER TABLE sale
    ADD CONSTRAINT sale_employee_fk FOREIGN KEY ( employee_id )
        REFERENCES employee ( employee_id );
```

## Последователности (*Sequences*) с цел автоматично изчисляване на първичен ключ.

```
-- The following sequences are used to generate unique identifiers
-- for various tables in the database.
-- Each sequence starts at 100 and increments by 10.
-- These sequences ensure that each record in the associated table receives
-- a distinct and predictable ID value.

-- Sequence for the customer table
CREATE SEQUENCE customer_seq START WITH 100 INCREMENT BY 10;

-- Sequence for the employee table
CREATE SEQUENCE employee_seq START WITH 100 INCREMENT BY 10;

-- Sequence for the group table
CREATE SEQUENCE group_seq START WITH 100 INCREMENT BY 10;

-- Sequence for the inventory table
CREATE SEQUENCE inventory_seq START WITH 100 INCREMENT BY 10;

-- Sequence for the inventory_sale table
CREATE SEQUENCE inventory_sale_seq START WITH 100 INCREMENT BY 10;

-- Sequence for the position table
CREATE SEQUENCE position_seq START WITH 100 INCREMENT BY 10;

-- Sequence for the product table
CREATE SEQUENCE product_seq START WITH 100 INCREMENT BY 10;

-- Sequence for the sale table
CREATE SEQUENCE sale_seq START WITH 100 INCREMENT BY 10;
```

**_Trigger_-и изпълняващи се преди събитие _INSERT_ за всяка една от таблиците.**

```
-- The triggers below are designed to automate the generation of unique
identifiers
-- for records in their respective tables. Before a new record is inserted,
the trigger
-- fetches the next value from the corresponding sequence and assigns it to
the record's primary key.
-- Trigger for the customer table: Assigns a unique customer_id before
insertion
CREATE OR REPLACE TRIGGER customer_bir
BEFORE INSERT ON customer
FOR EACH ROW
BEGIN
  :NEW.customer_id := customer_seq.NEXTVAL;
END;
/

-- Trigger for the employee table: Assigns a unique employee_id before
insertion
CREATE OR REPLACE TRIGGER employee_bir
BEFORE INSERT ON employee
FOR EACH ROW
BEGIN
  :NEW.employee_id := employee_seq.NEXTVAL;
END;
/

-- Trigger for the group table: Assigns a unique group_id before insertion
CREATE OR REPLACE TRIGGER group_bir
BEFORE INSERT ON "GROUP"
FOR EACH ROW
BEGIN
  :NEW.group_id := group_seq.NEXTVAL;
END;
/

-- Trigger for the inventory table: Assigns a unique inventory_id before
insertion
CREATE OR REPLACE TRIGGER inventory_bir
BEFORE INSERT ON inventory
FOR EACH ROW
BEGIN
  :NEW.inventory_id := inventory_seq.NEXTVAL;
END;
/
```

```
-- Trigger for the inventory_sale table: Assigns a unique inventory_sale_id
before insertion
CREATE OR REPLACE TRIGGER inventory_sale_bir
BEFORE INSERT ON inventory_sale
FOR EACH ROW
BEGIN
  :NEW.inventory_sale_id := inventory_sale_seq.NEXTVAL;
END;
/

-- Trigger for the position table: Assigns a unique position_id before
insertion
CREATE OR REPLACE TRIGGER position_bir
BEFORE INSERT ON position
FOR EACH ROW
BEGIN
  :NEW.position_id := position_seq.NEXTVAL;
END;
/

-- Trigger for the product table: Assigns a unique product_id before insertion
CREATE OR REPLACE TRIGGER product_bir
BEFORE INSERT ON product
FOR EACH ROW
BEGIN
  :NEW.product_id := product_seq.NEXTVAL;
END;
/

-- Trigger for the sale table: Assigns a unique sale_id before insertion
CREATE OR REPLACE TRIGGER sale_bir
BEFORE INSERT ON sale
FOR EACH ROW
BEGIN
  :NEW.sale_id := sale_seq.NEXTVAL;
END;
/
```

**Индексация на колони, по които често се извършват заявки с цел филтриране и извличане на данни.**

```
-- This index optimizes searches based on the product name.
CREATE INDEX idx_product_name ON product(name);

-- This index optimizes operations on the GROUP table's name column.
```

```sql
CREATE INDEX idx_group_name ON "GROUP"(name);

-- This index facilitates quicker searches, sorts on the price in the
inventory table.
CREATE INDEX idx_inventory_price ON inventory(price);

-- This index aids in operations that filter, sort sales based on the date.
CREATE INDEX idx_sale_date ON sale("date");
```

# SQL команди – DML

## Вкарване на данни в таблиците.

```
-- Populate the GROUP table.
CREATE OR REPLACE PROCEDURE add_group(p_group_name IN "GROUP".name%TYPE) AS
  v_group_count INTEGER;
BEGIN
  SELECT COUNT(*)
  INTO v_group_count
  FROM "GROUP"
  WHERE UPPER(name) = UPPER(p_group_name);

  IF v_group_count > 0 THEN
    RAISE_APPLICATION_ERROR(-20000, 'A group with the same name already
exists.');
  END IF;

  INSERT INTO "GROUP" (name) VALUES (p_group_name);
EXCEPTION
  WHEN OTHERS THEN
    ROLLBACK;
    RAISE;
END add_group;
/

BEGIN
  add_group('Electronics');
  add_group('Apparel');
  add_group('Furniture');
  add_group('Toys');
  add_group('Grocery');
  add_group('Books');
  add_group('Music');
  add_group('Video Games');
  add_group('Sports');
  add_group('Outdoors');
EXCEPTION
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE('Error occurred: ' || SQLERRM);
END;
/
```

```sql
-- Populate the product table.
CREATE OR REPLACE PROCEDURE add_product(
    p_name     IN product.name%TYPE,
    p_group_id IN product.group_id%TYPE
) AS
  v_product_count NUMBER;
  v_group_count NUMBER;
BEGIN

  SELECT COUNT(*)
  INTO v_product_count
  FROM product
  WHERE UPPER(name) = UPPER(p_name);

  IF v_product_count > 0 THEN
    RAISE_APPLICATION_ERROR(-20001, 'Product name already exists.');
  END IF;

  SELECT COUNT(*)
  INTO v_group_count
  FROM "GROUP"
  WHERE group_id = p_group_id;

  IF v_group_count = 0 THEN
    RAISE_APPLICATION_ERROR(-20002, 'Group ID does not exist.');
  END IF;

  INSERT INTO product (name, group_id) VALUES (p_name, p_group_id);
  DBMS_OUTPUT.PUT_LINE('New product added successfully.');

EXCEPTION
  WHEN OTHERS THEN
    ROLLBACK;
    RAISE;
END add_product;
/


BEGIN
  add_product('Smartphone', 100);
  add_product('T-Shirt', 110);
  add_product('Sofa', 120);
  add_product('Action Figure', 130);
  add_product('Pasta', 140);
  add_product('Novel', 150);
  add_product('Guitar', 160);
  add_product('PS5 Game', 170);
  add_product('Football', 180);
  add_product('Tent', 190);
```

```
  add_product('Laptop', 100);
  add_product('Bluetooth Headphones', 100);
  add_product('Jeans', 110);
  add_product('Sneakers', 110);
  add_product('Coffee Table', 120);
  add_product('Bookshelf', 120);
  add_product('Board Game', 130);
  add_product('Remote Control Car', 130);
  add_product('Organic Milk', 140);
  add_product('Eggs', 140);
EXCEPTION
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE('Error occurred: ' || SQLERRM);
END;
/



-- Populate the position table.
CREATE OR REPLACE PROCEDURE add_position(p_name IN position.name%type) IS
  v_count NUMBER;
BEGIN
  SELECT COUNT(*)
  INTO v_count
  FROM position
  WHERE UPPER(name) = UPPER(p_name);

  IF v_count > 0 THEN
    RAISE_APPLICATION_ERROR(-20001, 'Position name must be unique. The
provided name already exists.');
  END IF;

  INSERT INTO position (name) VALUES (p_name);

  DBMS_OUTPUT.PUT_LINE('New position "' || p_name || '" has been added
successfully.');

EXCEPTION
  WHEN OTHERS THEN
    ROLLBACK;
    RAISE;
END add_position;
/

BEGIN
  add_position('Manager');
  add_position('Salesperson');
  add_position('Cashier');
  add_position('Storekeeper');
```

```
  add_position('Cleaner');
  add_position('Security');
  add_position('Driver');
  add_position('Assistant Manager');
  add_position('Warehouse Worker');
  add_position('Delivery Person');
EXCEPTION
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE('Error occurred: ' || SQLERRM);
END;
/

-- Populate the employee table.
CREATE OR REPLACE PROCEDURE add_employee(
  p_first_name   IN employee.first_name%TYPE,
  p_last_name    IN employee.last_name%TYPE,
  p_phone_number IN employee.phone_number%TYPE,
  p_position_id  IN employee.position_id%TYPE
) AS
  v_phone_count  NUMBER;
  v_pos_count    NUMBER;
BEGIN

  SELECT COUNT(*)
  INTO v_phone_count
  FROM (
    SELECT phone_number FROM employee
    UNION
    SELECT phone_number FROM customer
  )
  WHERE phone_number = p_phone_number;

  IF v_phone_count > 0 THEN
    RAISE_APPLICATION_ERROR(-20000, 'Phone number already exists.');
  END IF;

  SELECT COUNT(*)
  INTO v_pos_count
  FROM position
  WHERE position_id = p_position_id;

  IF v_pos_count = 0 THEN
    RAISE_APPLICATION_ERROR(-20001, 'Position ID does not exist.');
  END IF;

  INSERT INTO employee (first_name, last_name, phone_number, position_id)
  VALUES (p_first_name, p_last_name, p_phone_number, p_position_id);

EXCEPTION
```

```
  WHEN OTHERS THEN
    ROLLBACK;
    RAISE;
END add_employee;
/



BEGIN
  add_employee('Alice', 'Green', '555-556-5551', 100);
  add_employee('Charlie', 'Black', '555-556-5553', 120);
  add_employee('Debbie', 'White', '555-556-5554', 130);
  add_employee('Eve', 'Gray', '555-556-5555', 140);
  add_employee('Grace', 'Teal', '555-556-5557', 160);
  add_employee('Ivy', 'Blue', '555-556-5559', 180);
  add_employee('Jack', 'Red', '555-556-5560', 190);
  add_employee('Liam', 'Stone', '555-556-5561', 170);
  add_employee('Ava', 'Sun', '555-556-5565', 150);
  add_employee('Sophia', 'Rain', '555-556-5566', 140);
  add_employee('Lucas', 'River', '555-556-5567', 130);
  add_employee('Mia', 'Ocean', '555-556-5568', 120);
  add_employee('Ethan', 'Forest', '555-556-5569', 160);
  add_employee('Isabella', 'Mountain', '555-556-5570', 180);
  add_employee('Bob', 'Brown', '555-556-5552', 110);
  add_employee('Frank', 'Purple', '555-556-5556', 110);
  add_employee('Hank', 'Yellow', '555-556-5558', 110);
  add_employee('Olivia', 'Wood', '555-556-5562', 110);
  add_employee('Noah', 'Sand', '555-556-5563', 110);
  add_employee('Emma', 'Grass', '555-556-5564', 110);
  COMMIT;
EXCEPTION
  WHEN OTHERS THEN
    ROLLBACK;
    DBMS_OUTPUT.PUT_LINE('Error occurred: ' || SQLERRM);
END;
/


-- Populate the customer table.
CREATE OR REPLACE PROCEDURE add_customer(
  p_first_name   IN customer.first_name%TYPE,
  p_last_name    IN customer.last_name%TYPE,
  p_phone_number IN customer.phone_number%TYPE
) AS
  v_phone_count NUMBER;
BEGIN
  SELECT COUNT(*)
  INTO v_phone_count
```

```
  FROM (
    SELECT phone_number FROM customer
    UNION
    SELECT phone_number FROM employee
  )
  WHERE phone_number = p_phone_number;

  IF v_phone_count > 0 THEN
    RAISE_APPLICATION_ERROR(-20001, 'The phone number is already in use.');
  END IF;

  INSERT INTO customer (first_name, last_name, phone_number)
  VALUES (p_first_name, p_last_name, p_phone_number);

  DBMS_OUTPUT.PUT_LINE('New customer added successfully.');
EXCEPTION
  WHEN OTHERS THEN
    ROLLBACK;
    RAISE;
END add_customer;
/


BEGIN
  add_customer('John', 'Smith', '555-555-5551');
  add_customer('Jane', 'Doe', '555-555-5552');
  add_customer('Robert', 'Johnson', '555-555-5553');
  add_customer('Mary', 'Davis', '555-555-5554');
  add_customer('David', 'Lee', '555-555-5555');
  add_customer('Peter', 'Parker', '555-555-5556');
  add_customer('Clark', 'Kent', '555-555-5557');
  add_customer('Bruce', 'Wayne', '555-555-5558');
  add_customer('Tony', 'Stark', '555-555-5559');
  add_customer('Steve', 'Rogers', '555-555-5560');
  add_customer('Diana', 'Prince', '555-555-5561');
  add_customer('Barry', 'Allen', '555-555-5562');
  add_customer('Arthur', 'Curry', '555-555-5563');
  add_customer('Hal', 'Jordan', '555-555-5564');
  add_customer('Oliver', 'Queen', '555-555-5565');
  add_customer('Selina', 'Kyle', '555-555-5566');
  add_customer('Natasha', 'Romanoff', '555-555-5567');
  add_customer('Bruce', 'Banner', '555-555-5568');
  add_customer('Wanda', 'Maximoff', '555-555-5569');
  add_customer('Scott', 'Lang', '555-555-5570');
EXCEPTION
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE('Error occurred: ' || SQLERRM);
END;
/
```

```sql
-- Populate the "inventory" table
CREATE OR REPLACE PROCEDURE add_inventory(
    p_quantity    IN inventory.quantity%TYPE,
    p_price       IN inventory.price%TYPE,
    p_import_date IN inventory.import_date%TYPE,
    p_product_id  IN inventory.product_id%TYPE
) AS
  v_product_count NUMBER;
BEGIN

  SELECT COUNT(*)
  INTO v_product_count
  FROM product
  WHERE product_id = p_product_id;

  IF v_product_count = 0 THEN
    RAISE_APPLICATION_ERROR(-20001, 'Product does not exist.');
  END IF;

  IF p_price <= 0 THEN
    RAISE_APPLICATION_ERROR(-20002, 'Price must be a positive number.');
  END IF;

  IF p_quantity < 0 THEN
    RAISE_APPLICATION_ERROR(-20003, 'Quantity cannot be negative.');
  END IF;

  INSERT INTO inventory (quantity, price, import_date, product_id)
  VALUES (p_quantity, p_price, p_import_date, p_product_id);
EXCEPTION
  WHEN OTHERS THEN
    ROLLBACK;
    RAISE;
END add_inventory;
/

BEGIN
  add_inventory(50, 300.0, TO_DATE('10/01/2023', 'MM/DD/YYYY'), 100);
  add_inventory(200, 20.0, TO_DATE('10/02/2023', 'MM/DD/YYYY'), 110);
  add_inventory(10, 500.0, TO_DATE('10/03/2023', 'MM/DD/YYYY'), 120);
  add_inventory(100, 15.0, TO_DATE('10/04/2023', 'MM/DD/YYYY'), 130);
  add_inventory(250, 2.0, TO_DATE('10/05/2023', 'MM/DD/YYYY'), 140);
  add_inventory(60, 10.0, TO_DATE('10/06/2023', 'MM/DD/YYYY'), 150);
  add_inventory(15, 100.0, TO_DATE('10/07/2023', 'MM/DD/YYYY'), 160);
  add_inventory(70, 60.0, TO_DATE('10/08/2023', 'MM/DD/YYYY'), 170);
  add_inventory(40, 25.0, TO_DATE('10/09/2023', 'MM/DD/YYYY'), 180);
  add_inventory(30, 120.0, TO_DATE('10/10/2023', 'MM/DD/YYYY'), 190);
```

```
  add_inventory(50, 300.0, TO_DATE('10/11/2023', 'MM/DD/YYYY'), 100);
  add_inventory(30, 700.0, TO_DATE('10/12/2023', 'MM/DD/YYYY'), 200);
  add_inventory(100, 80.0, TO_DATE('10/13/2023', 'MM/DD/YYYY'), 210);
  add_inventory(150, 40.0, TO_DATE('10/14/2023', 'MM/DD/YYYY'), 220);
  add_inventory(120, 60.0, TO_DATE('10/15/2023', 'MM/DD/YYYY'), 230);
  add_inventory(20, 150.0, TO_DATE('10/16/2023', 'MM/DD/YYYY'), 240);
  add_inventory(25, 80.0, TO_DATE('10/17/2023', 'MM/DD/YYYY'), 250);
  add_inventory(90, 30.0, TO_DATE('10/18/2023', 'MM/DD/YYYY'), 260);
  add_inventory(60, 50.0, TO_DATE('10/19/2023', 'MM/DD/YYYY'), 270);
  add_inventory(200, 3.0, TO_DATE('10/20/2023', 'MM/DD/YYYY'), 280);
  add_inventory(300, 4.0, TO_DATE('10/21/2023', 'MM/DD/YYYY'), 290);
  add_inventory(30, 44.0, TO_DATE('10/15/2023', 'MM/DD/YYYY'), 220);
  add_inventory(70, 50.0, TO_DATE('10/18/2023', 'MM/DD/YYYY'), 220);
EXCEPTION
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE('Error occurred: ' || SQLERRM);
END;
/


-- Populate the sale table.
CREATE OR REPLACE PROCEDURE add_sale(
    p_date         IN sale."date"%TYPE,
    p_customer_id  IN sale.customer_id%TYPE,
    p_employee_id  IN sale.employee_id%TYPE,
    p_sale_id      OUT sale.sale_id%TYPE
) AS
  v_customer_exists BOOLEAN;
  v_employee_exists BOOLEAN;
BEGIN

  v_customer_exists := customer_exists(p_customer_id);

  IF NOT v_customer_exists THEN
    RAISE_APPLICATION_ERROR(-20003, 'Customer ID does not exist.');
  END IF;

  v_employee_exists := has_sales_permission(p_employee_id);

  IF NOT v_employee_exists THEN
    RAISE_APPLICATION_ERROR(-20004, 'Employee ID does not exist or is not
permitted to process sales.');
  END IF;

  INSERT INTO sale ("date", customer_id, employee_id) VALUES (p_date,
p_customer_id, p_employee_id) RETURNING sale_id INTO p_sale_id;

  DBMS_OUTPUT.PUT_LINE('New sale added successfully.');
```

```
EXCEPTION
  WHEN OTHERS THEN
    ROLLBACK;
    RAISE;
END add_sale;
/


BEGIN
  add_sale(TO_DATE('10/1/2023', 'MM/DD/YYYY'), 100, 110);
  add_sale(TO_DATE('10/2/2023', 'MM/DD/YYYY'), 110, 110);
  add_sale(TO_DATE('10/3/2023', 'MM/DD/YYYY'), 120, 170);
  add_sale(TO_DATE('10/4/2023', 'MM/DD/YYYY'), 100, 170);
  add_sale(TO_DATE('10/5/2023', 'MM/DD/YYYY'), 140, 250);
  add_sale(TO_DATE('10/6/2023', 'MM/DD/YYYY'), 150, 250);
  add_sale(TO_DATE('10/7/2023', 'MM/DD/YYYY'), 160, 250);
  add_sale(TO_DATE('10/8/2023', 'MM/DD/YYYY'), 100, 170);
  add_sale(TO_DATE('10/9/2023', 'MM/DD/YYYY'), 180, 110);
  add_sale(TO_DATE('10/23/2023', 'MM/DD/YYYY'), 200, 110);
  add_sale(TO_DATE('10/10/2023', 'MM/DD/YYYY'), 100, 110);
  add_sale(TO_DATE('10/11/2023', 'MM/DD/YYYY'), 210, 110);
  add_sale(TO_DATE('10/12/2023', 'MM/DD/YYYY'), 120, 170);
  add_sale(TO_DATE('10/13/2023', 'MM/DD/YYYY'), 230, 170);
  add_sale(TO_DATE('10/14/2023', 'MM/DD/YYYY'), 100, 250);
  add_sale(TO_DATE('10/15/2023', 'MM/DD/YYYY'), 250, 250);
  add_sale(TO_DATE('10/16/2023', 'MM/DD/YYYY'), 260, 250);
  add_sale(TO_DATE('10/17/2023', 'MM/DD/YYYY'), 270, 170);
  add_sale(TO_DATE('10/18/2023', 'MM/DD/YYYY'), 280, 110);
  add_sale(TO_DATE('10/19/2023', 'MM/DD/YYYY'), 290, 110);
END;
/


-- Populate the "inventory_sale" table
CREATE OR REPLACE PROCEDURE add_inventory_sale(
    p_sold_quantity IN inventory_sale.sold_quantity%TYPE,
    p_inventory_id  IN inventory_sale.inventory_id%TYPE,
    p_sale_id       IN inventory_sale.sale_id%TYPE
) AS
  v_inventory_count NUMBER;
  v_sale_count      NUMBER;
BEGIN

  IF p_sold_quantity <= 0 THEN
    RAISE_APPLICATION_ERROR(-20001, 'Sold quantity must be a positive
number.');
  END IF;

  SELECT COUNT(*)
```

```
  INTO v_inventory_count
  FROM inventory
  WHERE inventory_id = p_inventory_id;

  IF v_inventory_count = 0 THEN
    RAISE_APPLICATION_ERROR(-20002, 'Inventory ID does not exist.');
  END IF;

  SELECT COUNT(*)
  INTO v_sale_count
  FROM sale
  WHERE sale_id = p_sale_id;

  IF v_sale_count = 0 THEN
    RAISE_APPLICATION_ERROR(-20003, 'Sale ID does not exist.');
  END IF;

  INSERT INTO inventory_sale (sold_quantity, inventory_id, sale_id)
  VALUES (p_sold_quantity, p_inventory_id, p_sale_id);

  DBMS_OUTPUT.PUT_LINE('Inventory sale record added successfully.');

EXCEPTION
  WHEN OTHERS THEN
    ROLLBACK;
    RAISE;
END add_inventory_sale;
/


BEGIN
  add_inventory_sale(1, 100, 100);
  add_inventory_sale(2, 110, 110);
  add_inventory_sale(3, 100, 120);
  add_inventory_sale(4, 120, 130);
  add_inventory_sale(5, 140, 140);
  add_inventory_sale(6, 130, 150);
  add_inventory_sale(7, 110, 160);
  add_inventory_sale(8, 150, 170);
  add_inventory_sale(9, 120, 180);
  add_inventory_sale(10, 160, 190);
  add_inventory_sale(11, 200, 130);
  add_inventory_sale(12, 210, 140);
  add_inventory_sale(13, 220, 200);
  add_inventory_sale(14, 230, 210);
  add_inventory_sale(15, 240, 220);
  add_inventory_sale(16, 250, 230);
  add_inventory_sale(17, 260, 240);
  add_inventory_sale(18, 270, 250);
```

```
  add_inventory_sale(19, 280, 260);
  add_inventory_sale(20, 290, 270);
EXCEPTION
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE('Error: ' || SQLERRM);
END;
/
```

**Процедури, функции, курсори за търсене на продукт по име, група, интервална цена (>, <, <>).**
**Процедура и за изчисляване на броят страници след търсене (*pagination*).**

**Процедурата приема като параметри *batch_start* и *batch_end*, тези стойности посочат броят записи, които да бъдат извлечени от базата при всяка заявка. По подразбиране броят им е 10. Направено е с цел прилагане на *pagination* и предотвратяване на случаите за извличане на голямо количество данни от базата наведнъж, което може да доведе до различни проблеми с производителността (напр. *Memory leak*).**

```
-- This PL/SQL block retrieves and displays a list of products from the
-- database based on user-defined criteria such as price intervals,
-- product name patterns, and group name patterns. The results can also
-- be limited by specifying batch start and end values. The procedure joins
-- the product, inventory, and GROUP tables to provide a comprehensive view of
-- each product, including its ID, name, group name, price, available
quantity, and import date.
--
-- Parameters:
--  :price_start_interval:
--    The minimum price in the desired price range.
--    If not specified, there's no minimum boundary.
--  :price_end_interval:
--    The maximum price in the desired price range.
--    If not specified, there's no maximum boundary.
--  :product_name_pattern:
--    A string pattern to filter products by name.
```

```
--    Products with names matching this pattern (case-insensitive)
--    will be retrieved. If unspecified, all product names will be considered.
--  :group_name_pattern:
--    A string pattern to filter products by their associated group name.
--    Products belonging to groups matching this pattern (case-insensitive)
will be retrieved.
--    If not provided, all group names are considered.
--  :batch_start:
--    The starting row number for the batch of records to be retrieved.
--  :batch_end:
--    The ending row number for the batch of records to be retrieved.
--
-- Validations:
--   Both :price_start_interval and :price_end_interval should not be
negative.
--   If either is negative, a message "Price intervals cannot be negative."
will be displayed.
--   :price_start_interval should not exceed :price_end_interval.
--   If this happens, a message "Start interval cannot be greater than end
interval." will be shown.
-- Both :batch_start and :batch_end should not be negative.
--   If either is negative, a message "Batch start and end values should be
positive." will be displayed.
--   :batch_start should not exceed :batch_end.
--   If this happens, a message "Batch start cannot be greater than batch
end." will be shown.
--
-- Output:
--   Products that fit within the specified price range will be displayed,
--   ordered by their product IDs and import dates.
--   If no products are found within the range, the message
--   "No records found for the given price interval." is shown.
--
-- Exceptions:
--   In case of any exceptions, the cursor, if open, will be closed, and the
exception will be logged.
CREATE OR REPLACE PROCEDURE fetch_products(
    p_batch_start IN NUMBER DEFAULT 1,
    p_batch_end IN NUMBER DEFAULT 10,
    p_price_start_interval IN NUMBER DEFAULT NULL,
    p_price_end_interval IN NUMBER DEFAULT NULL,
    p_product_name_pattern IN VARCHAR2 DEFAULT NULL,
    p_group_name_pattern IN VARCHAR2 DEFAULT NULL
) AS
  v_product_id         NUMBER;
  v_product_name       VARCHAR2(255);
  v_group_name         VARCHAR2(255);
  v_price              NUMBER;
  v_available_quantity INTEGER;
```

```
  v_import_date        DATE;
  v_has_records        BOOLEAN := FALSE;
  v_start_row          NUMBER := NVL(p_batch_start, 1);
  v_end_row            NUMBER := NVL(p_batch_end, 10);

  CURSOR product_cursor IS
    SELECT
      subq.product_id,
      subq.name AS product_name,
      subq.group_name,
      subq.price,
      subq.quantity AS available_quantity,
      subq.import_date
    FROM
      (SELECT
        p.product_id,
        p.name,
        g.name AS group_name,
        i.price,
        i.quantity,
        i.import_date,
        ROW_NUMBER() OVER (ORDER BY p.product_id ASC, i.import_date ASC) AS
row_num
      FROM
        product p
      JOIN
        inventory i ON p.product_id = i.product_id
      JOIN
        "GROUP" g ON p.group_id = g.group_id
      WHERE
        filter_price(i.price, p_price_start_interval, p_price_end_interval) =
1
        AND
        filter_product_name(p.name, p_product_name_pattern) = 1
        AND
        filter_group_name(g.name, p_group_name_pattern) = 1
      ) subq
    WHERE
      subq.row_num BETWEEN v_start_row AND v_end_row;
BEGIN
  -- Validations for price intervals
  validate_price_intervals(p_price_start_interval, p_price_end_interval);

  -- Validations for batch numbers
  validate_batch_numbers(v_start_row, v_end_row);

  OPEN product_cursor;

  LOOP
```

```
    FETCH product_cursor INTO v_product_id, v_product_name, v_group_name,
v_price, v_available_quantity, v_import_date;
    EXIT WHEN product_cursor%NOTFOUND;

    v_has_records := TRUE;
    DBMS_OUTPUT.PUT_LINE('Product ID: ' || v_product_id || ', Product Name: '
|| v_product_name || ', Group Name: ' || v_group_name || ', Price: ' ||
v_price || ', Available Quantity: ' || v_available_quantity || ', Import Date:
' || TO_CHAR(v_import_date, 'MM/DD/YYYY'));
  END LOOP;

  CLOSE product_cursor;

  IF NOT v_has_records THEN
    DBMS_OUTPUT.PUT_LINE('No records found for the provided criteria.');
  END IF;
EXCEPTION
  WHEN OTHERS THEN
    IF product_cursor%ISOPEN THEN
      CLOSE product_cursor;
    END IF;
    DBMS_OUTPUT.PUT_LINE('Error: ' || SQLERRM);
END fetch_products;
/



-- This procedure calculates the total number of pages required to display
products based
-- on user-specified filtering criteria and batch size. It takes into account
the price
-- range, product name pattern, group name pattern, and desired batch size to
determine
-- how many pages of products there would be if the products were divided into
batches of the specified size.
--
-- Input Parameters:
--   :price_start_interval:
--     The minimum price in the desired price range.
--     If not specified, there's no minimum boundary.
--   :price_end_interval:
--     The maximum price in the desired price range.
--     If not specified, there's no maximum boundary.
--   :product_name_pattern:
--     A pattern for product names.
--     The procedure will count products whose names match this pattern.
--     If not specified, all product names are included.
--   :group_name_pattern:
--     A pattern for group names.
```

```
--     The procedure will count products that belong to groups whose names
match this pattern.
--     If not specified, all group names are included.
--   :batch_start:
--     The starting number of the desired batch. If not specified, the default
is 1.
--   :batch_end:
--     The ending number of the desired batch. If not specified, the default
is 10.
--
-- Validations:
--   Price intervals should not be negative.
--     If either price_start_interval or price_end_interval is negative,
--     the message "Price intervals cannot be negative." will be displayed.
--   :price_start_interval should not exceed price_end_interval.
--     If this happens, a message "Start interval cannot be greater than end
interval." will be shown.
--   Batch start and end values should be positive.
--     If not, the message "Batch start and end values should be positive."
will be displayed.
--   :batch_start should not exceed :batch_end.
--     If this happens, a message "Batch start cannot be greater than batch
end." will be shown.
--
-- Output:
--   The procedure outputs the total number of pages as a message in
--   the format "Total pages: X", where X is the calculated number of pages.
--
-- Error Handling:
--   In case of exceptions or errors, relevant error messages will be
--   displayed, providing information about the nature of the error.
CREATE OR REPLACE PROCEDURE get_total_pages(
  p_batch_start IN NUMBER DEFAULT 1,
  p_batch_end IN NUMBER DEFAULT 10,
  p_price_start_interval IN NUMBER DEFAULT NULL,
  p_price_end_interval IN NUMBER DEFAULT NULL,
  p_product_name_pattern IN VARCHAR2 DEFAULT NULL,
  p_group_name_pattern IN VARCHAR2 DEFAULT NULL
) AS
  v_batch_size  NUMBER;
  v_total_pages NUMBER;
  v_start_row   NUMBER := NVL(p_batch_start, 1);
  v_end_row     NUMBER := NVL(p_batch_end, 10);
BEGIN

  -- Validations for price intervals
  validate_price_intervals(p_price_start_interval, p_price_end_interval);

  -- Validations for batch numbers
```

```
  validate_batch_numbers(v_start_row, v_end_row);

  -- Calculate batch size
  v_batch_size := v_end_row - v_start_row + 1;

  SELECT CEIL(COUNT(*)/v_batch_size) INTO v_total_pages
  FROM product p
  JOIN inventory i ON p.product_id = i.product_id
  JOIN "GROUP" g ON p.group_id = g.group_id
  WHERE
    filter_price(i.price, p_price_start_interval, p_price_end_interval) = 1
    AND
    filter_product_name(p.name, p_product_name_pattern) = 1
    AND
    filter_group_name(g.name, p_group_name_pattern) = 1;

  DBMS_OUTPUT.PUT_LINE('Total pages: ' || v_total_pages);
EXCEPTION
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE('Error: ' || SQLERRM);
END get_total_pages;
/



-- The procedure serves a dual purpose.
-- Initially, it fetches and displays products based on the
-- given criteria which include
-- filtering by price range, product name, and group name.
-- Subsequently, it calculates and outputs the total number of
-- pages these products span based on provided
-- batch start and end numbers.
CREATE OR REPLACE PROCEDURE display_products_and_pages(
  p_batch_start IN NUMBER DEFAULT 1,
  p_batch_end IN NUMBER DEFAULT 10,
  p_price_start_interval IN NUMBER DEFAULT NULL,
  p_price_end_interval IN NUMBER DEFAULT NULL,
  p_product_name_pattern IN VARCHAR2 DEFAULT NULL,
  p_group_name_pattern IN VARCHAR2 DEFAULT NULL
) AS
BEGIN
  fetch_products(p_batch_start, p_batch_end, p_price_start_interval,
p_price_end_interval, p_product_name_pattern, p_group_name_pattern);
  get_total_pages(p_batch_start, p_batch_end, p_price_start_interval,
p_price_end_interval, p_product_name_pattern, p_group_name_pattern);
EXCEPTION
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE('An error occurred: ' || SQLERRM);
```

```
END display_products_and_pages;
/



-- This procedure fetches and displays products based on provided parameters
and then
-- calculates the total number of pages based on batch start and end numbers.
BEGIN
  display_products_and_pages(:batch_start, :batch_end, :price_start_interval,
:price_end_interval, :product_name_pattern, :group_name_pattern);
END;
/



-- The function aids in identifying products within a specified price range.
-- It takes in a product's price and optional price interval boundaries.
-- The function assesses whether the product's price falls within these
bounds.
-- If no bounds are specified, the product's price defaults to matching
itself.
-- A return value of 1 represents a successful match, whereas 0 denotes no
match.
CREATE OR REPLACE FUNCTION filter_price(i_price NUMBER, p_price_start_interval
NUMBER DEFAULT NULL, p_price_end_interval NUMBER DEFAULT NULL) RETURN NUMBER
IS
BEGIN
    IF (i_price >= NVL(p_price_start_interval, i_price) AND i_price <=
NVL(p_price_end_interval, i_price)) THEN
        RETURN 1;
    ELSE
        RETURN 0;
    END IF;
END filter_price;
/



-- The function provides a mechanism to match product names against a specific
pattern.
-- Given a product name and an optional matching pattern, the function
evaluates
-- if the product name conforms to the pattern. In the absence of a specified
pattern, the product name matches itself.
-- A return value of 1 signifies a match, while 0 indicates no match.
CREATE OR REPLACE FUNCTION filter_product_name(p_name VARCHAR2,
p_product_name_pattern VARCHAR2 DEFAULT NULL) RETURN NUMBER IS
BEGIN
```

```
    IF (UPPER(p_name) LIKE '%' || UPPER(NVL(p_product_name_pattern, p_name))
|| '%') THEN
        RETURN 1;
    ELSE
        RETURN 0;
    END IF;
END filter_product_name;
/
```

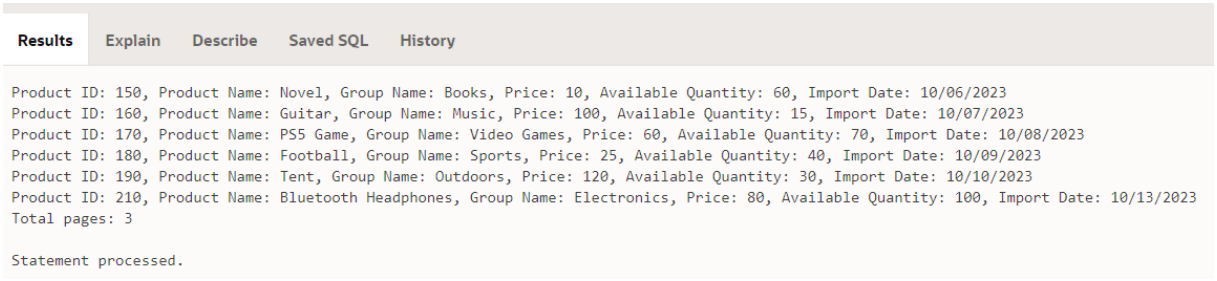
```
-- The function is designed to facilitate group name-based filtering in
product queries.
-- It accepts a group name and an optional pattern as its parameters. If the
-- given group name matches the pattern, or if no pattern is provided, the
function returns a 1, indicating
-- a successful match. Otherwise, it returns a 0.
CREATE OR REPLACE FUNCTION filter_group_name(g_name VARCHAR2,
p_group_name_pattern VARCHAR2 DEFAULT NULL) RETURN NUMBER IS
BEGIN
  IF (UPPER(g_name) LIKE '%' || UPPER(NVL(p_group_name_pattern, g_name)) ||
'%') THEN
    RETURN 1;
  ELSE
    RETURN 0;
  END IF;
END filter_group_name;
/
```

```
-- This procedure validates the given price intervals. It checks for two main
conditions:
-- 1. Ensures that neither the start nor the end price intervals are negative.
-- 2. Ensures that the start price interval is not greater than the end price
interval.
-- If any of these conditions are violated, the procedure raises a custom
application error with a descriptive message.
CREATE OR REPLACE PROCEDURE validate_price_intervals(
  p_price_start_interval IN NUMBER,
  p_price_end_interval IN NUMBER
) AS
BEGIN
  IF (p_price_start_interval IS NOT NULL AND p_price_start_interval < 0) OR
```

```
    (p_price_end_interval IS NOT NULL AND p_price_end_interval < 0) THEN
    RAISE_APPLICATION_ERROR(-20001, 'Price intervals cannot be negative.');
  ELSIF p_price_start_interval > p_price_end_interval THEN
    RAISE_APPLICATION_ERROR(-20002, 'Start interval cannot be greater than end
interval.');
  END IF;
END validate_price_intervals;
/
```

```
-- The procedure is designed to ensure the validity of batch boundaries
-- provided to product-fetching functions and procedures. It checks if the
batch start and
-- end values are positive and confirms that the start value doesn't surpass
the end value.
-- If any of these conditions aren't met, it raises an application-specific
error.
CREATE OR REPLACE PROCEDURE validate_batch_numbers(
  p_batch_start IN NUMBER,
  p_batch_end IN NUMBER
) AS
BEGIN
  IF (p_batch_start IS NOT NULL AND p_batch_start <= 0) OR
    (p_batch_end IS NOT NULL AND p_batch_end <= 0) THEN
    RAISE_APPLICATION_ERROR(-20003, 'Batch start and end values should be
positive.');
  ELSIF p_batch_start > p_batch_end THEN
    RAISE_APPLICATION_ERROR(-20004, 'Batch start cannot be greater than batch
end.');
  END IF;
END validate_batch_numbers;
/
```

Примерен резултат:




## Процедура, колекия, функции, тригери за реализиране на продажда и закупуване на продукти.

```
-- This procedure facilitates the purchase of products. It validates the
-- sales permissions of the involved employee, creates a sales record,
-- and updates the inventory based on the products purchased.
--
-- Parameters:
--   p_product_ids:    An array of product IDs that the customer wants to
purchase.
--   p_requested_qtys: The corresponding array of quantities for each product
ID.
--   p_customer_id:    The ID of the purchasing customer.
--   p_employee_id:    The ID of the employee processing the sale.
--
-- Outputs:
--   Inserts a new record into the 'sale' table.
--   Inserts records into 'inventory_sale' table for each product purchased.
--   Outputs a success message to DBMS_OUTPUT.
--   Can raise application errors under various conditions (e.g., insufficient
stock,
--   invalid employee permissions, etc.)
--
-- Exceptions:
--  Raises an error if a customer does not exist.
```

```
--  Raises an error if an employee doesn't have sales permissions.
--  Raises an error if there's insufficient stock for a product.
--  Raises an error if the product ID or employee ID is not found.
CREATE OR REPLACE PROCEDURE buy_products(
  p_product_ids      IN NUMBER_TABLE_TYPE,
  p_requested_qtys   IN NUMBER_TABLE_TYPE,
  p_customer_id      IN NUMBER,
  p_employee_id      IN NUMBER
) AS
  v_available_qty         NUMBER;
  v_sale_id               NUMBER;
  v_remaining_qty         NUMBER;
  v_current_inventory_qty INTEGER;
  v_current_inventory_id  INTEGER;
  v_employee_position     VARCHAR2(65);
  v_sold_product_output   VARCHAR2(1000);
BEGIN

  IF NOT customer_exists(p_customer_id) THEN
    RAISE_APPLICATION_ERROR(-20013, 'Customer ID not found.');
  END IF;

  IF NOT has_sales_permission(p_employee_id) THEN
    RAISE_APPLICATION_ERROR(-20012, 'This employee is not allowed to process
sales.');
  END IF;

  add_sale(SYSDATE, p_customer_id, p_employee_id, v_sale_id);

  FOR i IN 1..p_product_ids.COUNT LOOP
    v_available_qty := get_total_available_qty(p_product_ids(i));

    IF v_available_qty < p_requested_qtys(i) THEN
      RAISE_APPLICATION_ERROR(-20010, 'Insufficient stock available for
product ' || get_product_name(p_product_ids(i)));
    END IF;

    v_remaining_qty := p_requested_qtys(i);
    FOR rec IN (SELECT inventory_id, quantity FROM inventory WHERE product_id
= p_product_ids(i) ORDER BY import_date ASC) LOOP
      IF v_remaining_qty <= 0 THEN EXIT; END IF;
      v_current_inventory_qty := LEAST(v_remaining_qty, rec.quantity);
      v_current_inventory_id := rec.inventory_id;

      add_inventory_sale(v_current_inventory_qty, v_current_inventory_id,
v_sale_id);

      v_remaining_qty := v_remaining_qty - v_current_inventory_qty;
    END LOOP;
```

```
  END LOOP;

  DBMS_OUTPUT.PUT_LINE('Purchase processed successfully!');

EXCEPTION
  WHEN NO_DATA_FOUND THEN
    RAISE_APPLICATION_ERROR(-20011, 'Product ID or Employee ID not found.');
  WHEN OTHERS THEN
    ROLLBACK;
    RAISE;
END buy_products;
/


CREATE OR REPLACE FUNCTION get_product_name(p_product_id IN
product.product_id%TYPE)
RETURN VARCHAR2 AS
  v_product_name product.name%TYPE;
BEGIN
  SELECT name INTO v_product_name FROM product WHERE product_id =
p_product_id;
  RETURN v_product_name;
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    RAISE_APPLICATION_ERROR(-20001, 'Product with ID ' || p_product_id || '
does not exist.');
  WHEN OTHERS THEN
    RAISE;
END get_product_name;
/




-- This trigger is invoked after a new record
-- is inserted into the inventory_sale table. For every new sale entry, it
automatically
-- updates the quantity in the inventory table, deducting the sold quantity
from the existing inventory.
-- The deduction is based on matching the inventory_id from
-- the newly inserted inventory_sale record to the corresponding inventory_id
in the inventory table.
CREATE OR REPLACE TRIGGER tr_update_inventory_after_sale
AFTER INSERT ON inventory_sale
FOR EACH ROW
BEGIN
  UPDATE inventory
  SET quantity = quantity - :NEW.sold_quantity
```

```
  WHERE inventory_id = :NEW.inventory_id;
END tr_update_inventory_after_sale;
/


-- This trigger is fired after a new sale record is inserted into the "sale"
table.
-- It fetches the related employee and customer information based on the newly
inserted sale record.
-- The trigger then prints out the sale date, employee's name, position,
customer's name, and phone number.
CREATE OR REPLACE TRIGGER tr_after_insert_on_sale
AFTER INSERT ON sale
FOR EACH ROW
DECLARE
  v_employee_first_name VARCHAR2(255);
  v_employee_last_name VARCHAR2(255);
  v_employee_position VARCHAR2(255);
  v_customer_first_name VARCHAR2(255);
  v_customer_last_name VARCHAR2(255);
  v_customer_phone VARCHAR2(255);
BEGIN
  SELECT e.first_name, e.last_name, p.name, c.first_name, c.last_name,
c.phone_number
  INTO v_employee_first_name, v_employee_last_name, v_employee_position,
    v_customer_first_name, v_customer_last_name, v_customer_phone
  FROM employee e
  JOIN position p ON e.position_id = p.position_id
  JOIN customer c ON c.customer_id = :NEW.customer_id
  WHERE e.employee_id = :NEW.employee_id;
  DBMS_OUTPUT.PUT_LINE('Sale Date: ' || TO_CHAR(:NEW."date") ||
                       ', Employee: ' || v_employee_first_name || ' ' ||
v_employee_last_name ||
                       ', Position: ' || v_employee_position ||
                       ', Customer: ' || v_customer_first_name || ' ' ||
v_customer_last_name ||
                       ', Phone: ' || v_customer_phone);
EXCEPTION
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE('Error occurred in trg_after_insert_on_sale: ' ||
SQLERRM);
END;
/


-- This trigger is executed after a new record is inserted into the
"inventory_sale" table.
```

```
-- It fetches the product's name and price from the associated inventory and
then prints out
-- the product's name, the sold quantity, individual price, and the total
price for that sale.
CREATE OR REPLACE TRIGGER tr_after_insert_on_inventory_sale
AFTER INSERT ON inventory_sale
FOR EACH ROW
DECLARE
  v_product_name VARCHAR2(255);
  v_price NUMBER;
BEGIN
  SELECT p.name, i.price
  INTO v_product_name, v_price
  FROM product p
  JOIN inventory i ON i.product_id = p.product_id
  WHERE i.inventory_id = :NEW.inventory_id;

  DBMS_OUTPUT.PUT_LINE('Product: ' || v_product_name ||
                       ', Quantity: ' || TO_CHAR(:NEW.sold_quantity) ||
                       ', Price: ' || TO_CHAR(v_price) ||
                       ', Total: ' || TO_CHAR(:NEW.sold_quantity * v_price));
EXCEPTION
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE('Error occurred in trg_after_insert_on_sale: ' ||
SQLERRM);
    RAISE;
END;
/
```

```
-- This function retrieves the total available quantity of a specific product
from the inventory.
-- It takes a product ID as an argument and sums up all quantities related to
that product ID across
-- all inventory entries. If the product does not exist in the inventory or
has no stock, the function returns 0.
CREATE OR REPLACE FUNCTION get_total_available_qty(p_product_id NUMBER) RETURN
NUMBER IS
    v_total_qty NUMBER;
BEGIN
    SELECT SUM(i.quantity)
    INTO v_total_qty
    FROM product p
    JOIN inventory i ON p.product_id = i.product_id
```

```
    WHERE p.product_id = p_product_id;

    RETURN NVL(v_total_qty, 0);
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        RETURN 0;
    WHEN OTHERS THEN
        RAISE;
END get_total_available_qty;
/

-- The function determines if an employee, identified by their employee_id,
-- has the necessary permissions to process sales. The function retrieves
-- the employee's position from the database and checks if it matches any of
the predefined
-- roles that are allowed to handle sales. If the employee holds any of these
roles,
-- the function returns TRUE; otherwise, it returns FALSE.
-- In the event the provided employee_id does not exist,
-- an error is raised indicating that the employee ID was not found.
CREATE OR REPLACE FUNCTION has_sales_permission(p_employee_id IN NUMBER)
RETURN BOOLEAN AS
  v_employee_position VARCHAR2(65);
BEGIN
  SELECT pos.name
  INTO v_employee_position
  FROM employee emp
  JOIN position pos ON emp.position_id = pos.position_id
  WHERE emp.employee_id = p_employee_id;

  IF v_employee_position IN ('Manager', 'Salesperson', 'Assistant manager',
'Cashier') THEN
    RETURN TRUE;
  ELSE
    RETURN FALSE;
  END IF;

EXCEPTION
  WHEN NO_DATA_FOUND THEN
    RAISE_APPLICATION_ERROR(-20011, 'Employee ID not found.');
  WHEN OTHERS THEN
    RAISE;
END has_sales_permission;
/
```

```
-- The function determines whether a customer, identified by a given customer
ID,
-- exists in the database. It queries the `customer` table and checks for the
presence
-- of the specified customer ID. The function returns `TRUE` if the customer
exists and `FALSE` otherwise.
CREATE OR REPLACE FUNCTION customer_exists(p_customer_id IN NUMBER) RETURN
BOOLEAN AS
  v_customer_count INTEGER;
BEGIN
  SELECT COUNT(*)
  INTO v_customer_count
  FROM customer
  WHERE customer_id = p_customer_id;

  RETURN v_customer_count > 0;

EXCEPTION
  WHEN OTHERS THEN
    RAISE;
END customer_exists;
/

-- The code defines a PL/SQL type named NUMBER_TABLE_TYPE. This type
-- represents a nested table, where each element in the table is of
-- the NUMBER data type.
CREATE OR REPLACE TYPE NUMBER_TABLE_TYPE IS TABLE OF NUMBER;
```

```
-- This PL/SQL anonymous block invokes the `buy_products` procedure.
-- The block initializes arrays with product IDs and requested quantities,
then calls the procedure
-- to simulate a product purchase for a given customer and employee. If any
error occurs during execution,
-- an appropriate error message will be displayed.
DECLARE
  v_product_ids NUMBER_TABLE_TYPE := NUMBER_TABLE_TYPE(220, 110);
  v_requested_qtys NUMBER_TABLE_TYPE := NUMBER_TABLE_TYPE(200, 5);
BEGIN
  buy_products(
    p_product_ids => v_product_ids,
    p_requested_qtys => v_requested_qtys,
```

```
    p_customer_id => 100,
    p_employee_id => 110
  );
EXCEPTION
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE('Error: ' || SQLERRM);
END;
/
```

```
Sale Date: 10/26/2023, Employee: Charlie Black, Position: Cashier, Customer: John Smith, Phone: 555-555-5551
Product: Jeans, Quantity: 150, Price: 40, Total: 6000
Product: Jeans, Quantity: 30, Price: 44, Total: 1320
Product: Jeans, Quantity: 20, Price: 50, Total: 1000
Product: T-Shirt, Quantity: 5, Price: 20, Total: 100
Purchase processed successfully!
```

```
Sale Date: 10/26/2023, Employee: Charlie Black, Position: Cashier, Customer: John Smith, Phone: 555-555-5551
Error: ORA-20010: Insufficient stock available for product ID 220
```

## Процедури, курсор за гененериране на справка за продажби в период.

```
-- Generates a sales report for a specific date range and batch interval.
-- The report displays detailed information about each sale, including
-- the sale ID, sale date, product name, quantity sold, unit price, and total
amount.
--
-- Parameters:
--   p_start_date    : The starting date for the report. Defaults to one month
before the current date.
--   p_end_date      : The ending date for the report. Defaults to the current
date.
--   p_batch_start   : Starting row number for batch processing. Defaults to
1.
--   p_batch_end     : Ending row number for batch processing. Defaults to 10.
--
-- Output:
--   Displays the sales report using DBMS_OUTPUT.
--
-- Exceptions:
--   The procedure uses validate_batch_numbers and validate_dates to ensure
correct
```

```
--   batch numbers and dates. Any validation error or database error will be
displayed.
CREATE OR REPLACE PROCEDURE generate_sale_report(
  p_start_date  IN DATE DEFAULT TRUNC(ADD_MONTHS(SYSDATE, -1)),
  p_end_date    IN DATE DEFAULT TRUNC(SYSDATE),
  p_batch_start IN NUMBER DEFAULT 1,
  p_batch_end   IN NUMBER DEFAULT 10
) AS
  v_start_date    DATE := NVL(p_start_date, TRUNC(ADD_MONTHS(SYSDATE, -1)));
  v_end_date      DATE := NVL(p_end_date, TRUNC(SYSDATE));
  v_start_row     NUMBER := NVL(p_batch_start, 1);
  v_end_row       NUMBER := NVL(p_batch_end, 10);
  v_sale_id       NUMBER;
  v_sale_date     DATE;
  v_products      VARCHAR(255);
  v_sold_quantity NUMBER;
  v_unit_price    NUMBER;
  v_sale_total         NUMBER;

  CURSOR sale_cursor IS
    SELECT
      sale_id,
      sale_date,
      LISTAGG(product_name || ' (Qty: ' || sold_quantity || ', Unit Price: '
|| unit_price || ')', '; ')
        WITHIN GROUP (ORDER BY product_name) AS products,
      SUM(total) AS sale_total
    FROM
      (SELECT
          s.sale_id,
          s."date" AS sale_date,
          p.name AS product_name,
          isale.sold_quantity,
          i.price AS unit_price,
          (i.price * isale.sold_quantity) AS total,
          ROW_NUMBER() OVER (ORDER BY s."date" ASC, s.sale_id ASC) AS row_num
        FROM sale s
        JOIN inventory_sale isale ON s.sale_id = isale.sale_id
        JOIN inventory i ON i.inventory_id = isale.inventory_id
        JOIN product p ON p.product_id = i.product_id
        WHERE s."date" BETWEEN v_start_date AND v_end_date
      )
      WHERE row_num BETWEEN v_start_row AND v_end_row
    GROUP BY sale_id, sale_date
    ORDER BY sale_date, sale_id;

BEGIN
  validate_batch_numbers(v_start_row, v_end_row);
  validate_dates(v_start_date, v_end_date);
```

```
  DBMS_OUTPUT.PUT_LINE('Sale report: ' || TO_CHAR(v_start_date, 'DD-MON-YYYY')
|| ' to ' || TO_CHAR(v_end_date, 'DD-MON-YYYY'));

  OPEN sale_cursor;
  LOOP
    FETCH sale_cursor INTO v_sale_id, v_sale_date, v_products, v_sale_total;
    EXIT WHEN sale_cursor%NOTFOUND;

    DBMS_OUTPUT.PUT_LINE(
      'Sale date: ' || TO_CHAR(v_sale_date, 'DD-MON-YYYY') || ', ' ||
      'Products: ' || v_products || ', ' ||
      'Total Sale Value: ' || TO_CHAR(v_sale_total, '9999.99')
    );
  END LOOP;
  CLOSE sale_cursor;

EXCEPTION
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE('Error generating sale report: ' || SQLERRM);
END generate_sale_report;
/



-- Validates the given date parameters ensuring they are not in the future,
-- and that the start date is not greater than the end date.
CREATE OR REPLACE PROCEDURE validate_dates(
  p_start_date IN DATE,
  p_end_date   IN DATE
) AS
BEGIN
  IF p_start_date > SYSDATE OR p_end_date > SYSDATE THEN
    RAISE_APPLICATION_ERROR(-20001, 'Provided dates must not be in the
future.');
  END IF;

  IF p_start_date > p_end_date THEN
    RAISE_APPLICATION_ERROR(-20002, 'Start date must not be greater than end
date.');
  END IF;

EXCEPTION
  WHEN OTHERS THEN
    RAISE;
END validate_dates;
/
```

```
BEGIN
  generate_sale_report(:start_date, :end_date, :batch_start, :batch_end);
END;
/
```

```
Sale report: 02-OCT-2023 to 02-NOV-2023
Sale date: 02-OCT-2023, Products: T-Shirt (Qty: 2, Unit Price: 20), Total Sale Value:    40.00
Sale date: 03-OCT-2023, Products: Smartphone (Qty: 3, Unit Price: 300), Total Sale Value:   900.00
Sale date: 04-OCT-2023, Products: Smartphone (Qty: 11, Unit Price: 300); Sofa (Qty: 4, Unit Price: 500), Total Sale Value:  5300.00
Sale date: 05-OCT-2023, Products: Laptop (Qty: 12, Unit Price: 700); Pasta (Qty: 5, Unit Price: 2), Total Sale Value:  8410.00
Sale date: 06-OCT-2023, Products: Action Figure (Qty: 6, Unit Price: 15), Total Sale Value:    90.00
Sale date: 07-OCT-2023, Products: T-Shirt (Qty: 7, Unit Price: 20), Total Sale Value:   140.00
Sale date: 08-OCT-2023, Products: Novel (Qty: 8, Unit Price: 10), Total Sale Value:    80.00
Sale date: 09-OCT-2023, Products: Sofa (Qty: 9, Unit Price: 500), Total Sale Value:  4500.00
```

## Процедури, курсор, функция за гененериране на справка за продажби на служител.

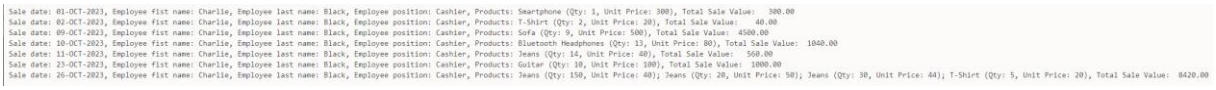
```
-- Generates a sales report for a specified employee, presenting details about
-- sales made, products sold, and the associated quantities and prices.
-- The report will also display the sales date, employee's name, position, and
product information.
-- Results can be limited to a specified range (batch) for pagination or
batching purposes.
-- Raises an error if the employee does not exist or if they do not have the
permission to process sales.
-- If no sales are found for the employee, a corresponding message is
displayed.
--
-- Parameters:
-- p_employee_phone_number: The phone number of the employee for whom the
report is generated.
-- p_batch_start: The starting row number for the batch (default is 1).
-- p_batch_end  : The ending row number for the batch (default is 10).
CREATE OR REPLACE PROCEDURE generate_employee_sale_report(
  p_employee_phone_number IN VARCHAR2,
  p_batch_start IN NUMBER DEFAULT 1,
  p_batch_end   IN NUMBER DEFAULT 10
) AS
```

```
  v_start_row           NUMBER := NVL(p_batch_start, 1);
  v_end_row             NUMBER := NVL(p_batch_end, 10);
  v_sale_id             NUMBER;
  v_sale_date           DATE;
  v_employee_first_name VARCHAR(255);
  v_employee_last_name  VARCHAR(255);
  v_employee_position   VARCHAR(255);
  v_products            VARCHAR(255);
  v_sale_total          NUMBER;
  v_sales_found         BOOLEAN := FALSE;

  CURSOR employee_sale_cursor IS
  SELECT
    subq.sale_id,
    subq.sale_date,
    subq.employee_first_name,
    subq.employee_last_name,
    subq.employee_position,
    LISTAGG(subq.product_name || ' (Qty: ' || subq.sold_quantity || ', Unit
Price: ' || subq.unit_price || ')', '; ')
      WITHIN GROUP (ORDER BY subq.product_name) AS products,
    SUM(subq.total) AS sale_total
  FROM
    (SELECT
        s.sale_id,
        s."date" AS sale_date,
        e.first_name as employee_first_name,
        e.last_name as employee_last_name,
        pos.name as employee_position,
        p.name AS product_name,
        isale.sold_quantity,
        i.price AS unit_price,
        (i.price * isale.sold_quantity) AS total,
        ROW_NUMBER() OVER (ORDER BY s."date" ASC, s.sale_id ASC) AS row_num
      FROM sale s
      JOIN inventory_sale isale ON s.sale_id = isale.sale_id
      JOIN inventory i ON i.inventory_id = isale.inventory_id
      JOIN product p ON p.product_id = i.product_id
      JOIN employee e ON s.employee_id = e.employee_id
      JOIN position pos ON e.position_id = pos.position_id
      WHERE e.phone_number = p_employee_phone_number
    ) subq
  WHERE subq.row_num BETWEEN v_start_row AND v_end_row
  GROUP BY subq.sale_id, subq.sale_date, subq.employee_first_name,
subq.employee_last_name, subq.employee_position;


BEGIN
  validate_batch_numbers(v_start_row, v_end_row);
```

```
  IF NOT
has_sales_permission(get_employee_id_by_phone(p_employee_phone_number)) THEN
    RAISE_APPLICATION_ERROR(-20012, 'This employee is not allowed to process
sales.');
  END IF;

  IF NOT employee_exists(get_employee_id_by_phone(p_employee_phone_number))
THEN
    RAISE_APPLICATION_ERROR(-20013, 'This employee does not exist!');
  END IF;

  OPEN employee_sale_cursor;
  LOOP
    FETCH employee_sale_cursor INTO v_sale_id, v_sale_date,
v_employee_first_name, v_employee_last_name, v_employee_position,
                            v_products, v_sale_total;

    EXIT WHEN employee_sale_cursor%NOTFOUND;

    v_sales_found := TRUE;

    DBMS_OUTPUT.PUT_LINE(
      'Sale date: ' || TO_CHAR(v_sale_date, 'DD-MON-YYYY') || ', ' ||
      'Employee fist name: ' || v_employee_first_name || ', ' ||
      'Employee last name: ' || v_employee_last_name || ', ' ||
      'Employee position: ' || v_employee_position || ', ' ||
      'Products: ' || v_products || ', ' ||
      'Total Sale Value: ' || TO_CHAR(v_sale_total, '9999.99')
    );

  END LOOP;
  CLOSE employee_sale_cursor;

  IF NOT v_sales_found THEN
    DBMS_OUTPUT.PUT_LINE('No sales found for employee with phone number: ' ||
p_employee_phone_number);
  END IF;

EXCEPTION
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE('Error generating sale report: ' || SQLERRM);
END generate_employee_sale_report;
/


CREATE OR REPLACE FUNCTION get_employee_id_by_phone(
  p_phone_number IN employee.phone_number%TYPE
) RETURN employee.employee_id%TYPE AS
```

```
  v_employee_id employee.employee_id%TYPE;
BEGIN
  SELECT employee_id
  INTO v_employee_id
  FROM employee
  WHERE phone_number = p_phone_number;

  RETURN v_employee_id;

EXCEPTION
  WHEN NO_DATA_FOUND THEN
    RAISE_APPLICATION_ERROR(-20001, 'No employee found with the given phone
number.');
  WHEN OTHERS THEN
    RAISE;
END get_employee_id_by_phone;
/



-- This function checks whether an employee with the given ID exists in the
employee table.
-- It returns TRUE if the employee exists, and FALSE otherwise.
CREATE OR REPLACE FUNCTION employee_exists(
  p_employee_id IN NUMBER
) RETURN BOOLEAN AS
  v_count NUMBER(1);
BEGIN
  SELECT COUNT(*)
  INTO v_count
  FROM employee
  WHERE employee_id = p_employee_id;

  IF v_count > 0 THEN
    RETURN TRUE;
  ELSE
    RETURN FALSE;
  END IF;
EXCEPTION
  WHEN OTHERS THEN
    RAISE;
END employee_exists;
/



BEGIN
  generate_employee_sale_report(:employee_phone_number, :batch_start,
:batch_end);
```

```
END;
/
```




## Процедури, курсор, функция за гененериране на справка за продажби на клиент подредени по дата.

```
-- This procedure generates a sale report for a specific customer, detailing
each sale transaction
-- including products, quantities, and total amounts. The report is ordered by
the sale date.
-- Pagination can be applied to the report by providing starting and ending
batch numbers.
--
-- Parameters:
--   p_phone_number: The phone number of the customer for whom the report will
be generated.
--   p_batch_start: The starting batch number for pagination (default is 1).
--   p_batch_end:   The ending batch number for pagination (default is 10).
--
-- Errors:
--   If the provided customer does not exist in the system, an error will be
raised.
--   Any other unexpected errors will be captured and displayed using
DBMS_OUTPUT.
CREATE OR REPLACE PROCEDURE generate_customer_sale_report(
  p_phone_number IN VARCHAR2,
  p_batch_start IN NUMBER DEFAULT 1,
  p_batch_end   IN NUMBER DEFAULT 10
) AS
  v_start_row           NUMBER := NVL(p_batch_start, 1);
  v_end_row             NUMBER := NVL(p_batch_end, 10);
  v_sale_id             NUMBER;
  v_sale_date           DATE;
  v_customer_first_name VARCHAR(255);
  v_customer_last_name  VARCHAR(255);
  v_products            VARCHAR(255);
  v_sale_total          NUMBER;
  v_sales_found         BOOLEAN := FALSE;
```

```
  CURSOR customer_sale_cursor IS
    SELECT
      subq.sale_id,
      subq.sale_date,
      subq.customer_first_name,
      subq.customer_last_name,
      LISTAGG(subq.product_name || ' (Qty: ' || subq.sold_quantity || ', Unit
Price: ' || subq.unit_price || ')', '; ')
        WITHIN GROUP (ORDER BY subq.product_name) AS products,
    SUM(subq.total) AS sale_total
    FROM
      (SELECT
          s.sale_id,
          s."date" AS sale_date,
          c.first_name as customer_first_name,
          c.last_name as customer_last_name,
          p.name AS product_name,
          isale.sold_quantity,
          i.price AS unit_price,
          (i.price * isale.sold_quantity) AS total,
          ROW_NUMBER() OVER (ORDER BY s."date" ASC, s.sale_id ASC) AS row_num
        FROM sale s
        JOIN inventory_sale isale ON s.sale_id = isale.sale_id
        JOIN inventory i ON i.inventory_id = isale.inventory_id
        JOIN product p ON p.product_id = i.product_id
        JOIN customer c ON s.customer_id = c.customer_id
        WHERE c.phone_number = p_phone_number
        ORDER BY s."date"
      ) subq
    WHERE subq.row_num BETWEEN v_start_row AND v_end_row
    GROUP BY subq.sale_id, subq.sale_date, subq.customer_first_name,
subq.customer_last_name;


BEGIN
  validate_batch_numbers(v_start_row, v_end_row);

  IF NOT customer_exists(get_customer_id_by_phone(p_phone_number)) THEN
    RAISE_APPLICATION_ERROR(-20013, 'This customer does not exist!');
  END IF;

  OPEN customer_sale_cursor;
  LOOP
    FETCH customer_sale_cursor INTO v_sale_id, v_sale_date,
v_customer_first_name, v_customer_last_name,
                          v_products, v_sale_total;
    EXIT WHEN customer_sale_cursor%NOTFOUND;

    v_sales_found := TRUE;
```

```
    DBMS_OUTPUT.PUT_LINE(
      'Sale date: ' || TO_CHAR(v_sale_date, 'DD-MON-YYYY') || ', ' ||
      'Customer fist name: ' || v_customer_first_name || ', ' ||
      'Customer last name: ' || v_customer_last_name || ', ' ||
      'Products: ' || v_products || ', ' ||
      'Total Sale Value: ' || TO_CHAR(v_sale_total, '9999.99')
    );

  END LOOP;
  CLOSE customer_sale_cursor;

  IF NOT v_sales_found THEN
    DBMS_OUTPUT.PUT_LINE('No sales found for customer with phone number: ' ||
p_phone_number);
  END IF;

EXCEPTION
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE('Error generating sale report: ' || SQLERRM);
END generate_customer_sale_report;
/


CREATE OR REPLACE FUNCTION get_customer_id_by_phone(
  p_phone_number IN customer.phone_number%TYPE
) RETURN customer.customer_id%TYPE AS
  v_customer_id customer.customer_id%TYPE;
BEGIN
  SELECT customer_id
  INTO v_customer_id
  FROM customer
  WHERE phone_number = p_phone_number;

  RETURN v_customer_id;

EXCEPTION
  WHEN NO_DATA_FOUND THEN
    RAISE_APPLICATION_ERROR(-20001, 'No customer found with the given phone
number.');
  WHEN OTHERS THEN
    RAISE;
END get_customer_id_by_phone;
/
```

```
-- This anonymous PL/SQL block is designed to execute the
generate_customer_sale_report procedure
-- for a specific customer using the provided customer ID and optional batch
start and end values
-- for pagination.
BEGIN
  generate_customer_sale_report(:customer, :batch_start, :batch_end);
END;
/
```