Functions & Stored procedures

Functions

1. **Function to Calculate Inventory Value:**

```
CREATE OR REPLACE FUNCTION myshop.calculate_inventory_value(id BIGINT,
unit_price NUMERIC)
RETURNS NUMERIC AS $$
DECLARE
    product_quantity INT;
BEGIN
    SELECT quantity INTO product_quantity FROM myshop.products WHERE
product_id = id;
    RETURN product_quantity * unit_price;
END;
$$ LANGUAGE plpgsql;
```

   - `CREATE OR REPLACE FUNCTION`: Defines a new function or replaces an existing one.
   - `RETURNS`: Specifies the return type of the function.
   - `DECLARE`: Declares variables to be used in the function.
   - `BEGIN ... END`: Defines the body of the function.
   - `plpgsql`: Specifies the procedural language for PostgreSQL.
   - Calculates the total value of inventory for a product by multiplying its quantity by a given unit price.

2. **Function to Check Product Availability:**

```
CREATE OR REPLACE FUNCTION myshop.is_product_available(id BIGINT,
threshold INT)
RETURNS BOOLEAN AS $$
DECLARE
    product_quantity INT;
BEGIN
```

```
    SELECT quantity INTO product_quantity FROM myshop.products WHERE
product_id = id;
    IF product_quantity > threshold THEN
        RETURN TRUE;
    ELSE
        RETURN FALSE;
    END IF;
END;
$$ LANGUAGE plpgsql;
```

- Checks if a product's quantity is above a specified threshold and returns a boolean value.

3. **Function to Get Client Name:**

```
CREATE OR REPLACE FUNCTION myshop.get_client_name(_client_id BIGINT)
RETURNS VARCHAR AS $$
BEGIN
    RETURN (SELECT name FROM myshop.clients WHERE client_id = _client_id);
END;
$$ LANGUAGE plpgsql;
```

- Fetches the name of a client based on their `client_id`.

4. **Function to Count Client Orders:**

```
CREATE OR REPLACE FUNCTION myshop.count_client_orders(_client_id BIGINT)
RETURNS INT AS $$
DECLARE
    orders_count INT;
BEGIN
    SELECT COUNT(*) INTO orders_count FROM myshop.orders WHERE client_id =
_client_id;
    RETURN orders_count;
END;
$$ LANGUAGE plpgsql;
```

- Calculates the total number of orders a client has placed.

5. **Usage Examples:**

```
-- Calculate the inventory value of a product with product_id = 1 and a
unit price of $50
```

```
SELECT myshop.calculate_inventory_value(1, 50.0) AS inventory_value;

-- Retrieve the name of the client with client_id = 2
SELECT myshop.get_client_name(2) AS client_name;
```

6. **Generate a Report Listing Product Availability and Total Orders:**

```
SELECT
    p.product_id,
    p.name AS product_name,
    p.quantity AS current_stock,
    is_product_available(p.product_id, 10) AS is_available,  -- Checks if
stock is above the threshold of 10 units
    COALESCE(op.total_orders, 0) AS total_orders
FROM
    myshop.products AS p
LEFT JOIN
    (SELECT product_id, COUNT(order_id) AS total_orders
     FROM myshop.ordered_products
     GROUP BY product_id) AS op ON p.product_id = op.product_id;
```

- `LEFT JOIN`: Combines rows from two tables, including all rows from the left table.
- `COALESCE`: Returns the first non-null value.
- Generates a report that lists product availability and total orders for each product.

7. **Function to List Products Along with Their Categories:**

```
CREATE OR REPLACE FUNCTION myshop.get_product_details()
RETURNS TABLE(product_id BIGINT, product_name VARCHAR, category_name
VARCHAR)
AS $$
BEGIN
    RETURN QUERY
    SELECT p.product_id, p.name, c.name
    FROM products p
    JOIN product_category pc ON p.product_id = pc.product_id
    JOIN categories c ON pc.category_id = c.category_id;
END;
$$ LANGUAGE plpgsql;
```

- Returns a table with product IDs, product names, and category names by joining products and categories.

8. **Define a New Type for Client Order Info:**

```
CREATE TYPE myshop.client_order_info AS (
    client_name VARCHAR,
    order_count INT
);
```

   - `CREATE TYPE`: Defines a new composite type.
   - Defines a type `client_order_info` with `client_name` and `order_count`.

9. **Function to Get Client Orders Using the New Type:**

```
CREATE OR REPLACE FUNCTION myshop.get_client_orders()
RETURNS SETOF myshop.client_order_info AS $$
BEGIN
    RETURN QUERY
    SELECT c.name, COUNT(o.order_id)
    FROM myshop.clients AS c
    JOIN myshop.orders AS o ON c.client_id = o.client_id
    GROUP BY c.name;
END;
$$ LANGUAGE plpgsql;
```

   - `SETOF`: Specifies that the function returns a set of rows.
   - Returns a set of client names and their order counts using the `client_order_info` type.

10. **Function to List All Clients:**

```
CREATE OR REPLACE FUNCTION myshop.list_all_clients()
RETURNS TABLE(_client_id BIGINT, client_name VARCHAR) AS $$
BEGIN
    RETURN QUERY
    SELECT client_id, name FROM myshop.clients;
END;
$$ LANGUAGE plpgsql;
```

   - Returns a table with all client IDs and names.

11. **Function to Get Product Info:**

```
CREATE OR REPLACE FUNCTION myshop.get_product_info(_product_id BIGINT)
RETURNS TABLE(id BIGINT, product_name VARCHAR, product_quantity INT) AS $$
DECLARE
    product_record RECORD;
BEGIN
    -- Fetch the product details into the record variable
    SELECT product_id, name, quantity INTO product_record
    FROM myshop.products
    WHERE product_id = _product_id;

    id := product_record.product_id;
    product_name := product_record.name;
    product_quantity := product_record.quantity;

    RETURN NEXT;
END;
$$ LANGUAGE plpgsql;
```

- `RECORD`: A variable that can hold a row of a query result.
- Returns the details of a product given its ID.

12. **Usage Examples for Functions:**

```
-- Get product details
SELECT * FROM myshop.get_product_details();

-- Get client orders
SELECT * FROM myshop.get_client_orders();

-- List all clients
SELECT * FROM myshop.list_all_clients();

-- Get product info for product_id = 1
SELECT * FROM myshop.get_product_info(1);
```

Stored procedures

1. **Procedure to Process an Order:**

```
CREATE OR REPLACE PROCEDURE myshop.process_order(_order_id BIGINT)
LANGUAGE plpgsql
```

```
AS $$
BEGIN
    -- Attempt to update product quantities based on the order
    UPDATE myshop.products
    SET quantity = myshop.products.quantity - op.quantity
    FROM myshop.ordered_products AS op
    WHERE op.product_id = myshop.products.product_id AND op.order_id =
_order_id;

    -- Commit the transaction
    COMMIT;
EXCEPTION
    WHEN OTHERS THEN
        -- Rollback the transaction in case of any exception
        ROLLBACK;
        -- Log the error for debugging purposes
        RAISE NOTICE 'Failed to process order %: %', order_id, SQLERRM;
END;
$$;
```

- `CREATE OR REPLACE PROCEDURE`: Defines a new procedure or replaces an existing one.
- `LANGUAGE plpgsql`: Specifies the procedural language for PostgreSQL.
- `BEGIN ... END`: Defines the body of the procedure.
- `EXCEPTION`: Handles exceptions that occur during the procedure execution.
- `WHEN OTHERS THEN`: Catches all exceptions.
- `RAISE NOTICE`: Logs a notice message.
- Updates product quantities based on the order and commits the transaction. If an error occurs, rolls back the transaction and logs the error.

2. **Procedure to Calculate Total Order Value:**

```
CREATE OR REPLACE PROCEDURE myshop.total_order_value(_order_id BIGINT, OUT
total_value NUMERIC)
LANGUAGE plpgsql
AS $$
BEGIN
    SELECT SUM(p.price * op.quantity) INTO total_value
    FROM myshop.ordered_products AS op
    JOIN products p ON op.product_id = p.product_id
    WHERE op.order_id = _order_id;
END;
$$;
```

- `OUT`: Specifies an output parameter that returns a value from the procedure.
- Calculates the total value of products ordered and returns this value via the `total_value` output parameter.

3. **Procedure to Adjust Product Quantity:**

```
CREATE OR REPLACE PROCEDURE myshop.adjust_product_quantity(INOUT
product_quantity INT, _product_id BIGINT, adjustment INT)
LANGUAGE plpgsql
AS $$
BEGIN
    -- Adjust the product quantity
    UPDATE myshop.products
    SET quantity = quantity + adjustment
    WHERE product_id = _product_id;

    -- Return the new product quantity
    SELECT quantity INTO product_quantity FROM myshop.products WHERE
product_id = _product_id;
END;
$$;
```

- `INOUT`: Specifies an input/output parameter that can be used both to pass a value to the procedure and to return a value from the procedure.
- Adjusts the quantity of a product and returns the new quantity using the `product_quantity` input/output parameter.

4. **Usage Examples:**

Call the Procedure to Process an Order:

```
CALL myshop.process_order(1);  -- Assuming '1' is a valid order ID
```

- Calls the `process_order` procedure to update inventory for order ID 1.

Call the Procedure to Calculate Total Order Value:

```
CALL myshop.total_order_value(1, total);  -- 'total' will hold the total
value of the order after the call
```

- Calls the `total_order_value` procedure to calculate the total value of order ID 1 and store it in the `total` variable.

Adjust Product Quantity and Retrieve New Quantity:

```
DO $$
DECLARE
  qty INT := 50;
BEGIN
  CALL myshop.adjust_product_quantity(qty, 1, 10);
  RAISE NOTICE 'New Quantity: %', qty;  -- Outputs the new quantity
END;
$$;
```

- `DO`: Executes an anonymous code block.
- Adjusts the quantity of product ID 1 by adding 10 to the initial quantity of 50, then outputs the new quantity.