Stored Procedure

Introduction to Stored Procedure

- A stored procedure in SQL is a group of SQL statements that are stored together in a database. Based on the statements in the procedure and the parameters you pass, it can perform one or multiple DML operations on the database, and return value, if any. Thus, it allows you to pass the same statements multiple times, thereby, enabling reusability.
- When you call a stored procedure for the first time, SQL Server creates an execution plan and stores it in the cache. In the subsequent executions of the stored procedure, SQL Server reuses the plan to execute the stored procedure very fast with reliable performance.

Stored Procedure Contd.

To create a stored procedure you can use the `CREATE PROCEDURE` statement as follows:

```
CREATE PROCEDURE uspProductList
AS
BEGIN
    SELECT
        product_name,
        list_price
    FROM
        production.products
    ORDER BY
        product_name;
END;
```

Stored Procedure Contd.

In this syntax:

The uspProductList is the name of the stored procedure.

The AS keyword separates the heading and the body of the stored procedure.

If the stored procedure has one statement, the BEGIN and END keywords surrounding the statement are optional. However, it is a good practice to include them to make the code clear.

Stored Procedure Parameters

Can add a parameter to the stored procedure to find the products whose list prices are greater than an input price:

```
ALTER PROCEDURE uspFindProducts(@min_list_price AS
DECIMAL)
AS
BEGIN
    SELECT
        product_name,
        list_price
    FROM
        production.products
    WHERE
        list_price >= @min_list_price
    ORDER BY
        list_price;
END;
```

In this example:

- First, we added a parameter named @min_list_price to the uspFindProducts stored procedure. Every parameter must start with the @ sign. The AS DECIMAL keywords specify the data type of the @min_list_price parameter. The parameter must be surrounded by the opening and closing brackets.
- Second, we used @min_list_price parameter in the WHERE clause of the SELECT statement to filter only the products whose list prices are greater than or equal to the @min_list_price

Creating a stored procedure with multiple parameters

- Stored procedures can take one or more parameters. The parameters are separated by commas.
- The following statement modifies the `uspFindProducts` stored procedure by adding one more parameter named `@max_list_price` to it:

```
ALTER PROCEDURE uspFindProducts(
    @min_list_price AS DECIMAL
    ,@max_list_price AS DECIMAL
)
AS
BEGIN
    SELECT product_name, list_price FROM  production.products
    WHERE
        list_price >= @min_list_price AND
        list_price <= @max_list_price
    ORDER BY
        list_price;
END;
```

Example

create a stored procedure to search for products based on a given category and brand.

```
CREATE PROCEDURE
SearchProducts
    @categoryName NVARCHAR(255),
    @brandName NVARCHAR(255)
AS
BEGIN
    SET NOCOUNT ON;
   -- Assuming you have a common ID
for categories and brands in the
products table
```

```
SELECT
p.product_id,p.product_name,p.model_year,
p.list_price,c.category_name,b.brand_name
FROM production.products p
INNER JOIN production.categories c ON
p.category_id = c.category_id
INNER JOIN production.brands b ON
p.brand_id = b.brand_id
    WHERE c.category_name LIKE '%' +
@categoryName + '%' AND b.brand_name
LIKE '%' + @brandName + '%';
END;
```

Variables

What is a variable

A variable is an object that holds a single value of a specific type e.g., integer, date, or varying character string.
We typically use variables in the following cases:

- As a loop counter to count the number of times a loop is performed.
- To hold a value to be tested by a control-of-flow statement such as `WHILE`.
- To store the value returned by a stored procedure or a function

Declaring a variable

To declare a variable, you use the `DECLARE` statement. For example, the following statement declares a variable named `@model_year`:

```
DECLARE @model_year SMALLINT;
```

Example

```
DECLARE @model_year SMALLINT;
SET @model_year = 2018;
SELECT product_name, model_year, list_price  FROM
production.products
WHERE
    model_year = @model_year
ORDER BY
    product_name;
```

Example of Variable

In this above example to get a list of products whose model year is 2018.
First, declare a variable named @product_count with the integer data type
Second, use the SET statement to assign the query's result set to the variable

Working of it

Accumulating values into a variable With Stored Procedure

```
CREATE  PROC uspGetProductList(
    @model_year SMALLINT
) AS
BEGIN
    DECLARE @product_list VARCHAR(MAX);
    SET @product_list = '';
    SELECT @product_list = @product_list + product_name+ CHAR(10)
    FROM  production.products
    WHERE model_year = @model_year
    ORDER BY  product_name;
    PRINT @product_list;
END;
```

Contd.

In this Context Explain it

First, we declared a variable named @product_list with varying character string type and set its value to blank.

Second, we selected the product name list from the products table based on the input @model_year. In the select list, we accumulated the product names to the @product_list variable. Note that the CHAR(10) returns the line feed character.

Third, we used the PRINT statement to print out the product list.

Stored Procedure Output Parameters

Creating output parameters

To create an output parameter for a stored procedure, you use the following syntax:

```
parameter_name data_type OUTPUT
```

Calling stored procedures with output parameters

To call a stored procedure with output parameters, you follow these steps:

- First, declare variables to hold the values returned by the output parameters
- Second, use these variables in the stored procedure call.

Example

Create a Stored Procedure finds products by model year and returns the number of products via the @product_count output parameter:

```
CREATE PROCEDURE uspFindProductByModel (
    @model_year SMALLINT,
    @product_count INT OUTPUT
) AS
BEGIN
    SELECT product_name, list_price FROM production.products
    WHERE
        model_year = @model_year;
    SELECT @product_count = @@ROWCOUNT;
END;
```

Contd.

First, we created an output parameter named @product_count to store the number of products found:

```
@product_count INT OUTPUT
```

Second, after the SELECT statement, we assigned the number of rows returned by the query(@@ROWCOUNT) to the @product_count parameter.

```
SELECT @product_count = @@ROWCOUNT;
```

Once you execute the CREATE PROCEDURE statement above, the uspFindProductByModel stored procedure is compiled and saved in the database catalog.

```
Commands completed successfully.
```

Note that the @@ROWCOUNT is a system variable that returns the number of rows read by the previous statement.

How To Execute It

```
DECLARE @count INT;

EXEC uspFindProductByModel

    @model_year = 2018,

    @product_count = @count OUTPUT;

SELECT @count AS 'Number of products found';
```

Purpose of Stored Procedure

- Reusable: As mentioned, multiple users and applications can easily use and reuse stored procedures by merely calling it.
- Easy to modify: You can quickly change the statements in a stored procedure as and when you want to, with the help of the ALTER TABLE command.
- Security: Stored procedures allow you to enhance the security of an application or a database by restricting the users from direct access to the table.
- Low network traffic: The server only passes the procedure name instead of the whole query, reducing network traffic.
- Increases performance: Upon the first use, a plan for the stored procedure is created and stored in the buffer pool for quick execution for the next time.

Scenario

Problem:-Consider a scenario where you have a sales database with tables for customers, orders, and products. You frequently need to retrieve information about a specific customer's order history, including details of each order and the products purchased.

Contd.

```
SELECT c.customer_id,
c.first_name,
c.last_name,
o.order_id,o.order_date,
p.product_name,oi.quantity,oi.unit_
price FROM customers c
INNER JOIN orders o ON
c.customer_id = o.customer_id
INNER JOIN order_items oi ON
o.order_id = oi.order_id
INNER JOIN products p ON
oi.product_id = p.product_id
WHERE c.customer_id = 123;
```

```
CREATE PROCEDURE GetOrderHistory
    @customerId INT
AS
BEGIN SET NOCOUNT ON;
SELECT o.order_id, o.order_date,
 p.product_name,oi.quantity,oi.unit_price
 FROM sales.orders o
 INNER JOINsales.order_items oi ON
o.order_id = oi.order_id
INNER JOIN production.products p ON
oi.product_id = p.product_id
WHERE o.customer_id = @customerId
ORDER BY o.order_date DESC;
END;
```

ROW_NUMBER() And PARTITION BY

- ROW_NUMBER() is a window function in SQL that assigns a unique sequential integer to each row within a partition of a result set. It is often used for tasks such as ranking, pagination, and filtering based on row numbers.
- The PARTITION BY clause is used in conjunction with window functions like ROW_NUMBER(), RANK(), and DENSE_RANK() to divide the result set into partitions based on one or more columns. The window function is then applied independently within each partition.
- **Syntax**

```
ROW_NUMBER() OVER (PARTITION BY partition_expression, ... ORDER BY sort_expression, ...);
```

- ORDER BY: Specifies the order in which the numbering is assigned within each partition.

Example

Assign a sequential integer to each customer. It resets the number when the city changes.

```
SELECT first_name, last_name, city,
   ROW_NUMBER() OVER (
      PARTITION BY city
      ORDER BY first_name
   ) row_num
FROM
   sales.customers
ORDER BY
   city;
```

Example-2

Display a list of customers by page, where each page has 10 rows.

```
WITH cte_customers AS (
    SELECT
        ROW_NUMBER() OVER(
            ORDER BY first_name, last_name) row_num, customer_id, first_name,
last_name FROM sales.customers)
 SELECT  customer_id, first_name, last_name
FROM
    cte_customers
WHERE
    row_num > 20 AND
    row_num <= 30;
```

Example-3

Create stored procedure with pagination for retrieving a list of products based on the product category and brand.

```
CREATE PROCEDURE GetProductsWithPagination
@CategoryId INT,@BrandId INT,
@PageSize INT,@PageNumber INT
AS
BEGIN
    SET NOCOUNT ON;
    DECLARE @Offset INT;
    SET @Offset = (@PageNumber - 1) * @PageSize;
    SELECT p.product_id, p.product_name AS
product_name,b.brand_name AS
brand_name,c.category_name AS
category_name,p.model_year,p.list_price
    FROM production.products AS p
```

```
INNER JOIN production.brands AS b ON
p.brand_id = b.brand_id
    INNER JOIN production.categories AS c ON
p.category_id = c.category_id
    WHERE
(@CategoryId IS NULL OR p.category_id =
@CategoryId)
AND (@BrandId IS NULL OR p.brand_id =
@BrandId)
ORDER BY p.product_id
    OFFSET @Offset ROWS
    FETCH NEXT @PageSize ROWS ONLY;
END;
```

CASE

- SQL Server CASE expression evaluates a list of conditions and returns one of the multiple specified results. The CASE expression has two formats: simple CASE expression and searched CASE expression. Both of CASE expression formats support an optional ELSE statement.
- Because CASE is an expression, you can use it in any clause that accepts an expression such as SELECT, WHERE, GROUP BY, and HAVING.

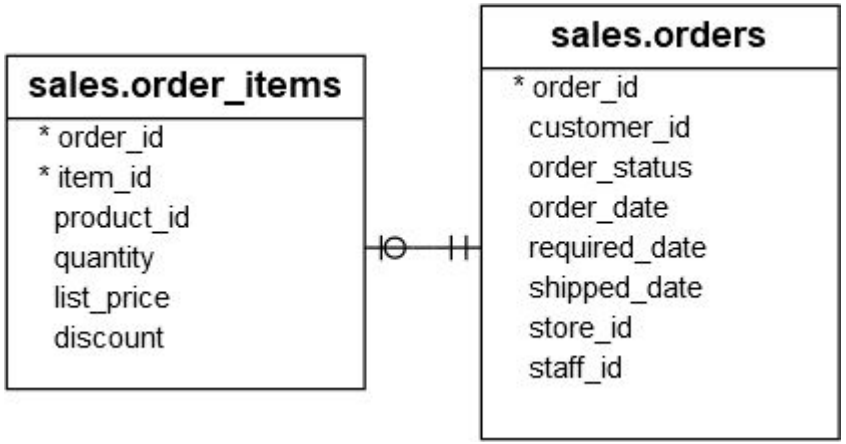
simple CASE expression

- The following shows the syntax of the simple `CASE` expression:

```
CASE input
   WHEN e1 THEN r1
  WHEN e2 THEN r2
  ...
  WHEN en THEN rn
  [ ELSE re ]
END
```

CASE EXAMPLE DESCRIPTION

See the following `sales.orders` and `sales.order_items`



Example

- Search classify sales order by order value

```
SELECT    o.order_id, SUM(quantity * list_price) order_value,
   CASE
       WHEN SUM(quantity * list_price) <= 500  THEN 'Very Low'
       WHEN SUM(quantity * list_price) > 500 AND SUM(quantity * list_price) <= 1000  THEN
'Low'
       WHEN SUM(quantity * list_price) > 1000 AND SUM(quantity * list_price) <= 5000 THEN
'Medium'
       WHEN SUM(quantity * list_price) > 5000 AND SUM(quantity * list_price) <= 10000
THEN 'High'
       WHEN SUM(quantity * list_price) > 10000 THEN 'Very High'
   END order_priority
FROM   sales.orders o
INNER JOIN sales.order_items i ON i.order_id = o.order_id
WHERE YEAR(order_date) = 2018 GROUP BY o.order_id;
```

COALESCE

Coalesce is used to handle the Null values. The null values are replaced with user-defined values during the expression evaluation process. This function evaluates arguments in a particular order from the provided arguments list and always returns the first non-null value.

Properties of the SQL Coalesce function and examples

- The data types of the expressions must be the same
- It can have multiple expressions in it
- Coalesce in SQL is a syntactic shortcut for the Case expression in SQL
- An integer is always evaluated first, and an integer followed by a character expression produces an integer as an output

COALESCE Contd.

- The following illustrates the syntax of the COALESCE expression:

```
COALESCE(e1,[e2,...,en])
```

- In this syntax, e1, e2, … en are scalar expressions that evaluate to scalar values. The COALESCE expression returns the first non-null expression. If all expressions evaluate to NULL, then the COALESCE expression return NULL;
- Because the COALESCE is an expression, you can use it in any clause that accepts an expression such as SELECT, WHERE, GROUP BY, and HAVING.

Example Data

- create a new table named salaries that stores the employee's salaries:

```
CREATE TABLE salaries (staff_id INT PRIMARY KEY,hourly_rate decimal,
 weekly_rate decimal,monthly_rate decimal,CHECK(hourly_rate IS NOT NULL OR
weekly_rate IS NOT NULL OR monthly_rate IS NOT NULL));
```

Each staff can have only one rate either hourly, weekly, or monthly.
Second, insert some rows into the `salaries` table:

```
INSERT INTO salaries(staff_id, hourly_rate, weekly_rate, monthly_rate)VALUES(1,20,
NULL,NULL), (2,30, NULL,NULL), (3,NULL, 1000,NULL),(4,NULL, NULL,6000),(5,NULL,
NULL,6500);
```

Example Contd.

calculate monthly for each staff using the COALESCE expression as shown in the following query:

```
SELECT
    staff_id,
    COALESCE(
        hourly_rate*22*8,
        weekly_rate*4,
        monthly_rate
    ) monthly_salary
FROM
    salaries;
```

DYNAMIC QUERY

- Dynamic Query is a programming technique that allows you to construct SQL statements dynamically at runtime. It allows you to create more general purpose and flexible SQL statement because the full text of the SQL statements may be unknown at compilation. For example, you can use the dynamic SQL to create a stored procedure that queries data against a table whose name is not known until runtime.
- An example of how to create a dynamic query in SQL:
  - Suppose you have a table named "product" with columns Id,Name,brand_id,category_id,mpdel_year,list_price . You want to create a query that retrieves the records based on the Category name input. The user can input any combination of the four columns.
  - To construct a dynamic query for this scenario, you can use the CONCAT and IF functions to dynamically generate the WHERE clause of the SQL query.

DYNAMIC QUERY Contd.

Suppose the user inputs the values for categoryName='Comfort Bicycles' and brandId = 1, and you want to retrieve the records that match these criteria. You can construct the dynamic query as follows:

```
CREATE PROCEDURE GetProductsByCategory @categoryName NVARCHAR(255) = NULL,@id INT = NULL,@name
NVARCHAR(255) = NULL,@brandId INT = NULL
AS BEGIN SET NOCOUNT ON;
    DECLARE @sql NVARCHAR(MAX);
    SET @sql = 'SELECT * FROM product WHERE 1 = 1';
    IF @categoryName IS NOT NULL
        SET @sql = @sql + ' AND category_id IN (SELECT category_id FROM Category WHERE category_name LIKE "%' +
@categoryName + '%")';
    IF @id IS NOT NULL
        SET @sql = @sql + ' AND Id = ' + CAST(@id AS NVARCHAR(10));
    IF @name IS NOT NULL
        SET @sql = @sql + ' AND Name LIKE "%' + @name + '%"';
    IF @brandId IS NOT NULL
        SET @sql = @sql + ' AND brand_id = ' + CAST(@brandId AS NVARCHAR(10));
    EXEC sp_executesql @sql;
END;
```