**What is a Stored Procedure ?**

A **Stored Procedure** is like a pre-packaged, reusable script in that you can run whenever you need it. Think of it as a **recipe** for your queries. Instead of writing the same code again and again, you can create a stored procedure that encapsulates your logic, and simply **call** it whenever you need to use that same functionality.

**Why Use Stored Procedures?**

1. **Efficiency**: You can avoid writing repetitive queries in your applications. A stored procedure can be executed multiple times with different parameters, saving you time and effort.
2. **Performance**: Server pre-compiles stored procedures, so they can run faster than executing the same statement multiple times from an application. Pre-compilation means that the execution plan is already optimized and stored.
3. **Security**: By using stored procedures, you can control the access to sensitive data. Users don’t need direct access to the tables; they can only execute the stored procedure, which restricts the direct exposure of your data.
4. **Maintainability**: All the logic is contained within the stored procedure. If you need to change it, you only need to change the stored procedure instead of updating every instance of the code in the application.
5. **Modularity**: Stored procedures allow you to break down complex logic into smaller, more manageable pieces, making your database code easier to manage and update.

**Key Features of a Stored Procedure:**

* **Reusability**: Once created, you can run a stored procedure multiple times with different input parameters.
* **Input Parameters**: Stored procedures can accept **parameters** to make them more dynamic. These can be used to filter data or change the logic of the procedure.
* **Output**: A stored procedure can also **return** values, making it useful for calculations, summaries, or even fetching results.
* **Error Handling**: You can include robust error-handling logic to ensure smooth execution even if something goes wrong.

**Breaking it Down with an Example**

Let’s create an example to understand how a stored procedure works.

**Scenario: We want to calculate the total salary of all employees in a given department. Instead of writing the same code each time, we’ll encapsulate it in a stored procedure.**

**Step 1: Creating the Stored Procedure**

We’ll create a procedure called **GetTotalSalaryByDepartment**. This procedure will accept one input parameter: @DepartmentName. Based on this parameter, the procedure will calculate the sum of salaries for all employees in that department.

CREATE PROCEDURE GetTotalSalaryByDepartment

@DepartmentName VARCHAR (100) -- Input parameter for the department name

AS

BEGIN

SELECT SUM(Salary) AS TotalSalary

FROM Employees

WHERE Department = @DepartmentName;

END;

* **Input**: @DepartmentName – The name of the department for which we want to calculate the total salary.
* **Logic**: The query sums up the salaries (SUM(Salary)) from the Employees table where the department matches the provided @DepartmentName.
* **Output**: The total salary of employees in the specified department.

**Step 2: Executing the Stored Procedure**

Once the procedure is created, you can **call** or **execute** it like this:

EXEC GetTotalSalaryByDepartment 'IT';

This will execute the procedure, passing 'IT' as the input parameter. The output will be the total salary of all employees in the **IT** department.

**Expected Output:**

Let’s assume the Employees table has the following data:

|  |  |  |  |
| --- | --- | --- | --- |
| **EmployeeID** | **Name** | **Department** | **Salary** |
| 1 | Alice | IT | 60000 |
| 2 | Bob | IT | 70000 |
| 3 | Charlie | HR | 50000 |

When you run the stored procedure with 'IT' as input, the output will be:

|  |
| --- |
| **TotalSalary** |
| 130000 |

This shows the sum of the salaries for all employees in the IT department.

No, you do **not** require an additional SELECT statement after using the EXEC command. The stored procedure itself handles the SELECT operation internally. When you execute the stored procedure, it directly returns the result of the SELECT statement defined within it.

**Step 3: Modifying the Stored Procedure (Add Output Parameter)**

Let’s extend this procedure to return not only the **total salary** but also the **average salary** of employees in a department. We’ll use an **output parameter** for the average salary.

CREATE PROCEDURE GetSalaryInfoByDepartment

@DepartmentName VARCHAR(100), -- Input parameter for department name

@TotalSalary DECIMAL(10, 2) OUTPUT, -- Output parameter for total salary

@AverageSalary DECIMAL(10, 2) OUTPUT -- Output parameter for average salary

AS

BEGIN

SELECT @TotalSalary = SUM(Salary), -- Calculate total salary

@AverageSalary = AVG(Salary) -- Calculate average salary

FROM Employees

WHERE Department = @DepartmentName;

END;

Now, when you call this procedure, it will return both the total and the average salary for a given department.

**Step 4: Executing the Procedure with Output Parameters**

To execute the procedure and capture the output values:

DECLARE @Total DECIMAL(10, 2), @Average DECIMAL(10, 2);

EXEC GetSalaryInfoByDepartment 'IT', @Total OUTPUT, @Average OUTPUT;

-- Display the results

SELECT @Total AS TotalSalary, @Average AS AverageSalary;

Note: The reason for using SELECT after executing a stored procedure like this:is due to the use of **output parameters** in the stored procedure. Unlike a SELECT statement inside a stored procedure that directly returns a result set to the client, **output parameters** require the caller to explicitly retrieve and display their values

**Expected Output:**

If you have the same data as before, the output will be:

|  |  |
| --- | --- |
| **TotalSalary** | **AverageSalary** |
| 130000 | 65000.00 |

This shows the total and average salaries for the IT department.

**Stored Procedure with Control Flow (IF/ELSE, Loops)**

Stored procedures can also use control flow logic (like IF, WHILE, etc.).

**Example:**

CREATE PROCEDURE UpdateEmployeeSalary

@EmployeeID INT,

@NewSalary DECIMAL(10, 2)

AS

BEGIN

IF @NewSalary > 100000

BEGIN

PRINT 'Salary exceeds the maximum limit.';

RETURN;

END

UPDATE Employees

SET Salary = @NewSalary

WHERE EmployeeID = @EmployeeID;

PRINT 'Salary updated successfully.';

END;

This procedure:

* Checks if the new salary exceeds a limit.
* If it does, the procedure prints a message and exits.
* If the salary is within the limit, it updates the employee's salary.

**Error Handling in Stored Procedures (TRY...CATCH)**

SQL Server supports error handling using TRY...CATCH blocks.

**Example:**

CREATE PROCEDURE SafeUpdateSalary

@EmployeeID INT,

@NewSalary DECIMAL(10, 2)

AS

BEGIN

BEGIN TRY

UPDATE Employees

SET Salary = @NewSalary

WHERE EmployeeID = @EmployeeID;

PRINT 'Salary updated successfully.';

END TRY

BEGIN CATCH

PRINT 'An error occurred.';

-- You can also raise the error or log it

END CATCH;

END;

In this example, if any error occurs during the salary update, the procedure will print a message and handle the error without breaking the whole process.

**Dropping a Stored Procedure**

If you no longer need a stored procedure, you can drop it:

DROP PROCEDURE procedure\_name;

**Advanced Features of Stored Procedures**

* **Error Handling**: You can handle errors gracefully using TRY...CATCH blocks in to ensure smooth execution, even in the case of unexpected issues.
* **Transaction Control**: You can use **transactions** within stored procedures to ensure that a series of operations (e.g., updates, inserts) are executed atomically. If something goes wrong, you can **rollback** the transaction to maintain data integrity.
* **Control Flow**: Stored procedures allow you to use control flow statements like IF...ELSE, WHILE, and loops, giving you flexibility in how the logic is executed.

Stored procedures are widely used in real-world applications like Amazon or Facebook to handle complex business logic, optimize performance, and improve the maintainability of database systems. While the specific details of how these companies use stored procedures are proprietary, we can provide some **practical scenarios** where stored procedures might be used in such large-scale environments.

**1. E-commerce Applications (Amazon)**

In an e-commerce platform like **Amazon**, stored procedures are likely used for several critical operations, such as managing inventory, processing orders, calculating discounts, and ensuring data consistency.

**Example Use Cases in Amazon:**

**Order Processing**:  
When an order is placed on Amazon, there are multiple steps involved:

Checking stock availability.

Calculating the total cost, including taxes and shipping.

Updating inventory.

Notifying the shipping department.

A stored procedure can be used to handle this process in one atomic transaction to ensure consistency. For instance, the procedure might:

* + Verify product availability.
  + Deduct the stock.
  + Calculate the final price based on the customer’s location.
  + Update the Orders and Inventory tables.
  + Commit or rollback based on success or failure.

CREATE PROCEDURE ProcessOrder

@CustomerID INT,

@OrderID INT,

@TotalCost DECIMAL(10,2) OUTPUT

AS

BEGIN

BEGIN TRANSACTION;

-- Check product availability

IF NOT EXISTS (SELECT 1 FROM Inventory WHERE ProductID = @ProductID AND Quantity > 0)

BEGIN

ROLLBACK TRANSACTION;

RETURN 'Product Not Available';

END

-- Update inventory

UPDATE Inventory

SET Quantity = Quantity - 1

WHERE ProductID = @ProductID;

-- Insert into Orders table

INSERT INTO Orders (OrderID, CustomerID, ProductID, OrderDate, TotalCost)

VALUES (@OrderID, @CustomerID, @ProductID, GETDATE(), @TotalCost);

COMMIT TRANSACTION;

END;

* **Calculating Discounts**:  
  Amazon often offers dynamic discounts based on promotional campaigns or customer loyalty. A stored procedure can automate this process by accepting parameters like CustomerID, TotalPurchaseAmount, and CouponCode, and then calculating and applying the appropriate discount.

CREATE PROCEDURE ApplyDiscount

@CustomerID INT,

@TotalAmount DECIMAL(10, 2),

@CouponCode VARCHAR(50),

@Discount DECIMAL(10, 2) OUTPUT

AS

BEGIN

-- Example logic to calculate discount

IF EXISTS (SELECT 1 FROM Coupons WHERE CouponCode = @CouponCode AND ExpiryDate > GETDATE())

BEGIN

SET @Discount = @TotalAmount \* 0.10; -- 10% discount

END

ELSE

BEGIN

SET @Discount = 0;

END

END;

This stored procedure can be invoked when processing checkout, making sure that the appropriate discounts are applied and updated in the database.

**2. Social Media Applications (Facebook)**

For a large-scale social media platform like **Facebook**, stored procedures are crucial in handling user interactions, posts, notifications, privacy settings, and data aggregation.

**Example Use Cases in Facebook:**

* **User Registration and Profile Creation**: When a new user registers, several steps need to occur simultaneously:

Inserting user information.

Setting up default privacy settings.

Creating an initial profile page.

Sending a welcome email.

A stored procedure can encapsulate these tasks in a single, efficient operation.

CREATE PROCEDURE RegisterNewUser

@UserName VARCHAR(100),

@Email VARCHAR(100),

@Password VARCHAR(100),

@ProfilePic VARCHAR(200),

@UserID INT OUTPUT

AS

BEGIN

BEGIN TRANSACTION;

-- Insert into Users table

INSERT INTO Users (UserName, Email, Password)

VALUES (@UserName, @Email, @Password);

-- Get the newly created UserID

SET @UserID = SCOPE\_IDENTITY();

-- Set default privacy settings

INSERT INTO PrivacySettings (UserID, DefaultSetting)

VALUES (@UserID, 'Public');

-- Create initial profile page

INSERT INTO Profiles (UserID, ProfilePic)

VALUES (@UserID, @ProfilePic);

COMMIT TRANSACTION;

END;

This procedure ensures that user registration and setup are done in a transaction, so everything is handled in one go, and no partial data is stored.

**Example Scenario**

**Tables:**

Assume the following tables exist in the database:

1. **Users Table:**

|  |  |  |  |
| --- | --- | --- | --- |
| **UserID (PK)** | **UserName** | **Email** | **Password** |

1. **PrivacySettings Table:**

|  |  |  |
| --- | --- | --- |
| **SettingID (PK)** | **UserID (FK)** | **DefaultSetting** |

1. **Profiles Table:**

|  |  |  |
| --- | --- | --- |
| **ProfileID (PK)** | **UserID (FK)** | **ProfilePic** |

**Executing the Procedure:**

To register a new user, the procedure can be executed as follows:

DECLARE @NewUserID INT;

EXEC RegisterNewUser

@UserName = 'JohnDoe',

@Email = 'johndoe@example.com',

@Password = 'hashedpassword123', -- In production, store hashed passwords

@ProfilePic = '/images/johndoe.jpg',

@UserID = @NewUserID OUTPUT;

SELECT @NewUserID AS NewUserID;

-

**What Happens:**

**Insert into Users:**

INSERT INTO Users (UserName, Email, Password)

VALUES ('JohnDoe', 'johndoe@example.com', 'hashedpassword123');

A new record is added to the Users table.

Assume the generated UserID is 101.

**Retrieve UserID:**

SET @UserID = SCOPE\_IDENTITY();

Retrieves the UserID (101) of the newly created record.

**Insert into PrivacySettings:**

INSERT INTO PrivacySettings (UserID, DefaultSetting)

VALUES (101, 'Public');

Adds a default "Public" privacy setting for UserID 1

**Insert into Profiles:**

INSERT INTO Profiles (UserID, ProfilePic)

VALUES (101, '/images/johndoe.jpg');

Creates a profile for UserID 101 with the specified profile picture.

**Commit Transaction:**

All operations succeed, and changes are saved to the database.

**Final Database State:**

1. **Users Table:**

|  |  |  |  |
| --- | --- | --- | --- |
| **UserID** | **UserName** | **Email** | **Password** |
| 101 | JohnDoe | johndoe@example.com | hashedpassword123 |

1. **PrivacySettings Table:**

|  |  |  |
| --- | --- | --- |
| **SettingID** | **UserID** | **DefaultSetting** |
| 1 | 101 | Public |

1. **Profiles Table:**

|  |  |  |
| --- | --- | --- |
| **ProfileID** | **UserID** | **ProfilePic** |
| 1 | 101 | /images/johndoe.jpg |

* **News Feed Generation**: Facebook’s news feed is one of the most complex and dynamic features, involving pulling content based on user interests, preferences, friends’ activities, etc. A stored procedure could be used to fetch relevant posts, apply filtering based on privacy settings, and return a curated list for display.

CREATE PROCEDURE GetUserNewsFeed

@UserID INT,

@PageNumber INT,

@PostsPerPage INT

AS

BEGIN

-- Fetch posts by friends

SELECT p.PostID, p.Content, p.PostDate

FROM Posts p

JOIN Friends f ON p.UserID = f.FriendID

WHERE f.UserID = @UserID

ORDER BY p.PostDate DESC

OFFSET (@PageNumber - 1) \* @PostsPerPage ROWS

FETCH NEXT @PostsPerPage ROWS ONLY;

END;

**Example Tables**

1. **Users Table:**

|  |  |
| --- | --- |
| **UserID** | **UserName** |
| 1 | Alice |
| 2 | Bob |
| 3 | Charlie |

1. **Friends Table:**

|  |  |
| --- | --- |
| **UserID** | **FriendID** |
| 1 | 2 |
| 1 | 3 |

1. *(Alice (UserID 1) is friends with Bob (FriendID 2) and Charlie (FriendID 3).)*
2. **Posts Table:**

|  |  |  |  |
| --- | --- | --- | --- |
| **PostID** | **UserID** | **Content** | **PostDate** |
| 101 | 2 | "Bob's first post" | 2024-12-01 10:00:00 |
| 102 | 3 | "Charlie's photo" | 2024-12-01 09:00:00 |
| 103 | 2 | "Bob's second post" | 2024-12-01 08:00:00 |

**Calling the Procedure**

**Example Call:**

To fetch Alice's news feed for **Page 1**, with **2 posts per page**:

EXEC GetUserNewsFeed

@UserID = 1,

@PageNumber = 1,

@PostsPerPage = 2;

**Execution Steps:**

1. **Join Posts and Friends:**

SELECT p.PostID, p.Content, p.PostDate

FROM Posts p

JOIN Friends f ON p.UserID = f.FriendID

WHERE f.UserID = 1;

Result:

|  |  |  |
| --- | --- | --- |
| **PostID** | **Content** | **PostDate** |
| 101 | "Bob's first post" | 2024-12-01 10:00:00 |
| 102 | "Charlie's photo" | 2024-12-01 09:00:00 |
| 103 | "Bob's second post" | 2024-12-01 08:00:00 |

1. **Order by PostDate DESC:**

ORDER BY p.PostDate DESC;

Result:

|  |  |  |
| --- | --- | --- |
| **PostID** | **Content** | **PostDate** |
| 101 | "Bob's first post" | 2024-12-01 10:00:00 |
| 102 | "Charlie's photo" | 2024-12-01 09:00:00 |
| 103 | "Bob's second post" | 2024-12-01 08:00:00 |

1. **Apply Pagination:**

OFFSET (1 - 1) \* 2 ROWS -- Skip 0 rows for Page 1

FETCH NEXT 2 ROWS ONLY; -- Fetch 2 rows

Final Result:

|  |  |  |
| --- | --- | --- |
| **PostID** | **Content** | **PostDate** |
| 101 | "Bob's first post" | 2024-12-01 10:00:00 |
| 102 | "Charlie's photo" | 2024-12-01 09:00:00 |

**For Page 2:**

To fetch the second page:

EXEC GetUserNewsFeed

@UserID = 1,

@PageNumber = 2,

@PostsPerPage = 2;

Pagination Calculation:

OFFSET (2 - 1) \* 2 ROWS -- Skip 2 rows for Page 2

FETCH NEXT 2 ROWS ONLY; -- Fetch 2 rows

Result:

|  |  |  |
| --- | --- | --- |
| **PostID** | **Content** | **PostDate** |
| 103 | "Bob's second post" | 2024-12-01 08:00:00 |

This procedure retrieves posts from a user’s friends (or other filters like groups), returning only the posts for the current page view.

**3. Real-Time Data Processing**

Large-scale platforms like Amazon and Facebook also have real-time data processing needs—such as tracking user actions, processing payments, or generating real-time reports—where stored procedures come in handy to keep performance high and ensure data consistency.

**Example:**

In **Facebook**, if a user likes a post, a stored procedure could handle multiple actions in one step:

* Increase the like count.
* Update the user’s activity log.
* Notify the post owner about the new like.
* Update the news feed for other users who follow or interact with the post.

CREATE PROCEDURE LikePost

@UserID INT,

@PostID INT

AS

BEGIN

BEGIN TRANSACTION;

-- Increment like count on the post

UPDATE Posts

SET LikeCount = LikeCount + 1

WHERE PostID = @PostID;

-- Add entry to ActivityLog

INSERT INTO ActivityLog (UserID, PostID, Action)

VALUES (@UserID, @PostID, 'Liked');

-- Send notification to post owner

INSERT INTO Notifications (UserID, Message)

VALUES ((SELECT UserID FROM Posts WHERE PostID = @PostID), 'Your post was liked.');

COMMIT TRANSACTION;

END;

This stored procedure allows Facebook to handle multiple operations efficiently in response to a single action (liking a post), ensuring the system works seamlessly without slowing down.

**Conclusion**

In **Amazon** and **Facebook**, stored procedures are used to:

* **Automate repetitive tasks** like order processing, inventory management, and profile creation.
* **Encapsulate complex business logic** into manageable units, improving maintainability and reducing redundancy.
* **Ensure data integrity** by handling transactions and error management.
* **Optimize performance** by pre-compiling and storing SQL logic in the database.

These companies likely use stored procedures for any process that requires consistent, repeatable actions, especially when dealing with large-scale databases and real-time operations.

Here are some multiple-choice questions (MCQs) on **Stored Procedures** in SQL:

**1. What is a stored procedure in SQL?**

a) A set of SQL statements that can be executed as a single unit.

b) A query to retrieve data from the database.

c) A table that stores SQL commands.

d) A command used to update a database record.

**Answer:** a) A set of SQL statements that can be executed as a single unit.

**2. Which of the following is TRUE about stored procedures?**

a) They cannot accept parameters.

b) They can only be executed in the context of a transaction.

c) They allow reuse of SQL code by storing the code in the database.

d) They can only be used in SELECT statements.

**Answer:** c) They allow reuse of SQL code by storing the code in the database.

**3. Which keyword is used to execute a stored procedure in SQL?**

a) EXECUTE

b) RUN

c) EXEC

d) CALL

**Answer:** c) EXEC

**4. Which of the following can be used in a stored procedure?**

a) SELECT only

b) SELECT, INSERT, UPDATE, DELETE

c) Only DML commands

d) Only the SELECT command with parameters

**Answer:** b) SELECT, INSERT, UPDATE, DELETE

**5. What is the purpose of the OUTPUT parameter in a stored procedure?**

a) It is used to return values from the procedure to the calling program.

b) It is used to receive values as input into the procedure.

c) It is used to log errors during procedure execution.

d) It is used to update values in the database.

**Answer:** a) It is used to return values from the procedure to the calling program.

**6. Which of the following statements is true regarding stored procedures?**

a) Stored procedures are only used to query data.

b) Stored procedures allow for parameterized queries and business logic implementation.

c) Stored procedures are faster than views.

d) Stored procedures cannot handle errors.

**Answer:** b) Stored procedures allow for parameterized queries and business logic implementation.

**7. Which of the following is the correct syntax for creating a stored procedure?**

a) CREATE PROCEDURE ProcedureName AS BEGIN SQL STATEMENTS END;

b) CREATE FUNCTION ProcedureName BEGIN SQL STATEMENTS END;

c) CREATE PROCEDURE ProcedureName BEGIN SQL STATEMENTS END;

d) CREATE PROCEDURE ProcedureName PARAMETERS (Param1, Param2) BEGIN SQL STATEMENTS END;

**Answer:** a) CREATE PROCEDURE ProcedureName AS BEGIN SQL STATEMENTS END;

**8. Can a stored procedure in SQL return a result set?**

a) Yes, it can return multiple result sets.

b) No, stored procedures cannot return any result set.

c) Yes, but it can only return one result set.

d) Only functions can return result sets.

**Answer:** a) Yes, it can return multiple result sets.

**9. Which of the following is NOT a valid use case for a stored procedure?**

a) Performing complex calculations on data.

b) Returning data to a client application.

c) Managing security and permissions in a database.

d) Creating database objects like tables and views.

**Answer:** c) Managing security and permissions in a database.

**10. What happens if a stored procedure is executed without proper error handling and an error occurs?**

a) The stored procedure will skip the problematic code and continue executing.

b) The procedure will stop executing, and no changes will be made to the database.

c) An error message is automatically returned to the user.

d) The procedure will automatically log the error into a log table.

**Answer:** b) The procedure will stop executing, and no changes will be made to the database.