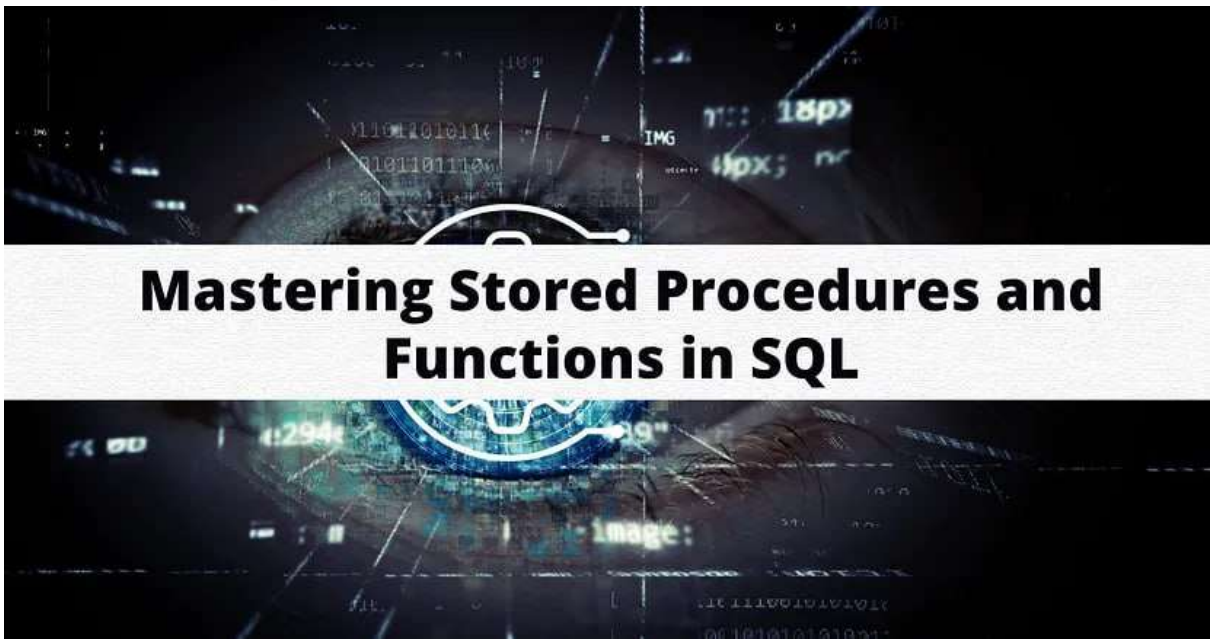
PROCEDURES AND STANDARD FUNCTIONS IN SQL

Ochilboyev Umidjon Ilxom o‘g‘li
Do‘schanov Bekzod Davronbek o‘g‘li
Ismonaliyev Sanjarbek Qanbaraliyevich
Tashkent University of Information Technologies named after Muhammad al-Khwarizmi

Abstract: *A SQL function is a pre-written, reusable code block to perform a specific task. It is created and stored in the database and can be called from an SQL statement or another function. A SQL procedure is a pre-written, reusable code block designed to perform a specific task or set of tasks.*

Keywords: *SQL, Stored Procedures, Function, Optional parameter, Parameterized Queries, SQL Server Management Studio, MySQL Workbench.*

Users can encapsulate and reuse SQL code using the potent programming structures known as stored procedures and functions in SQL. They make it possible for users to define a group of SQL statements as a unique executable unit that can be used repeatedly with various inputs, which makes it a crucial tool for managing intricate databases. Users can write effective, reusable code that can enhance security, ease maintenance, and improve database performance by becoming fluent in SQL’s stored procedures and functions. Database administrators and developers who need to create flexible, efficient SQL code will find this ability to be especially useful. The basic ideas and methods for learning SQL stored procedures and functions are covered in this guide, along with examples of how to use them.



In SQL, procedures and standard functions play crucial roles in database management and manipulation. Here's an overview of each, including their definitions, usage, and examples

Procedures, also known as stored procedures, are a set of SQL statements that can be executed as a single unit. They are stored in the database and can be called by name, allowing for reusable and modular code.

Usage:

- Encapsulate business logic
- Improve performance by reducing network traffic (as multiple statements are executed in a single call)
- Ensure consistent implementation of operations
- Simplify complex operations by breaking them into simpler steps

Creating Stored Procedure

Stored procedures are user-defined routines that execute a set of SQL statements. Creating stored procedures in SQL involves defining a name, parameters (optional), and the SQL code to be executed. Here's the basic syntax for creating a stored procedure in SQL:

```
CREATE PROCEDURE procedure_name
[ @parameter1 datatype [ = default_value ] ]
[ ,@parameter2 datatype [ = default_value ] ]
AS
BEGIN
SQL code to be executed
END
```

Let us split down the various components of this syntax:

The keyword build PROCEDURE is used to build a new stored procedure.

The name of the stored process is procedure_name. @parameter is a parameter that can be given to the stored procedure as an optional parameter. A comma-separated list can be used to specify multiple parameters.

- datatype defines the parameter's data type.
- default_value is the parameter's optional default value.
- AS is the SQL code block's starting term.
- BEGIN and conclusion indicate the beginning and conclusion of the SQL code block.

Simple Select Statement:

Here's an example of a simple stored procedure that executes a select statement:

```
CREATE PROCEDURE spGetCustomers
AS
BEGIN
SELECT * FROM customers
END
```

In this example, we've defined a stored procedure called spGetCustomers that returns all rows from the customers table.

Parameterized Queries:

Stored procedures can also accept input parameters, which can be used to filter or modify the results of a query. Here's an example of a parameterized stored procedure:

```
CREATE PROCEDURE spGetCustomersByCountry
@country nvarchar(50)
AS
BEGIN
SELECT * FROM customers WHERE country = @country
END
```

In this example, we've created a stored procedure called spGetCustomersByCountry that takes a single parameter of type nvarchar named @country.(50). Following that, the stored procedure runs a select statement that filters the customers database based on the value of the @country parameter.

Transactions:

Transactions, which guarantee that a group of SQL statements are executed as a single unit of work, can also be performed using stored procedures. If any part of the transaction fails, the entire transaction is rolled back. Here's an example of a stored procedure that performs a transaction:

```
CREATE PROCEDURE spTransferFunds
@from_account nvarchar(50),
@to_account nvarchar(50),
@amount decimal(18,2)
AS
BEGIN
BEGIN TRANSACTION
UPDATE accounts SET balance = balance - @amount WHERE account_number = @from_account
UPDATE accounts SET balance = balance + @amount WHERE account_number = @to_account
COMMIT TRANSACTION
END
```

In this instance, spTransferFunds is a stored procedure that accepts three arguments: @from_account, @to_account, and @amount. After that, the stored procedure executes two update statements that move the specified amount from the from_account to the to_account and deduct the specified amount from the from_account. The stored procedure then commits the action to the database.

The ability to write SQL stored procedures is essential for database administrators and developers. Complex databases can be handled more easily thanks to stored procedures, which enable the encapsulation and reuse of SQL code. Developers can write effective and reusable SQL code that boosts database performance, makes maintenance easier, and increases security by becoming familiar

with the syntax for creating stored procedures and understanding the different kinds of stored procedures.

Executing Stored Procedures:

Stored procedures are precompiled collections of SQL statements that are run repeatedly and saved in a database. They make database administration more effective and flexible by enabling users to encapsulate and reuse code. The procedures listed below should be followed in order to run a stored procedure from a SQL client like SQL Server Management Studio or MySQL Workbench:

Join the database:

Connecting to the database containing the stored procedure is the first stage. The UI of the SQL client can be used for this.

Where to find the saved procedure.

The stored method should then be found in the database. This can be accomplished by searching the SQL client or perusing the database's schema.

The stored process should be run:

Right-click on the saved procedure to launch it.

Parameter Passing:

Stored procedures can be executed with parameters that allow for flexible and dynamic execution. There are different ways to pass parameters to a stored procedure, including:

Positional Parameters:

Positional parameters are passed to the stored procedure based on their position in the parameter list. For example, if a stored procedure has three parameters, the first parameter will be passed first, followed by the second parameter and then the third parameter.

Named Parameters:

Named parameters are passed to the stored procedure based on their name rather than their position. This allows for more flexible parameter passing and makes the stored procedure easier to read and maintain.

Handling Return Values:

Stored procedures can return values to the calling program or client. There are different ways to handle return values from a stored procedure, including:

Output Parameters:

Output parameters are used to return values from a stored procedure. They are declared in the stored procedure and assigned a value within the stored procedure. The calling program or client can then retrieve the output parameter's value.

Result Sets:

Stored procedures can return result sets that contain one or more rows of data. The calling program or client can then retrieve the result set and process the data.

Return Codes:

Stored procedures can also return a single return code that indicates whether the stored procedure was executed successfully or not. The calling program or client can then use the return code to determine the next course of action.

Executing stored procedures is a powerful feature of SQL that allows users to encapsulate and reuse code. By passing parameters and handling return values, users can make their stored procedures more flexible and dynamic. Understanding how to execute stored procedures and work with their parameters and return values is essential for any database developer or administrator.

Debugging Stored Procedures:

Database developers and admins must debug stored procedures on a regular basis. It is the process of detecting and correcting errors or bugs in SQL code so that it works as expected. This section will go over various debugging techniques for stored procedures, such as using print statements, setting breakpoints, and examining execution plans.

Using Print Statements

Using print statements is one of the easiest and most efficient methods to debug stored procedures. These statements give programmers the ability to print out variable values and other significant data while a stored function is being executed. The source of errors can be swiftly found and fixed by developers by looking at the printed output.

Stored Procedure (SP)	**Function (UDF – User Defined Function)**
SP can return zero , single or multiple values.	Function can return one value which is mandatory.
We can use transaction in SP.	We can't use transaction in UDF.
SP can have input/output parameter.	Only input parameter.
We can called function from SP.	We can't call SP from function.
We can't use SP in SELECT/ WHERE/ HAVING statement.	We can use UDF in SELECT/ WHERE/ HAVING statement.
We can use exception handling using Try-Catch block in SP.	We can't use Try-Catch block in UDF.

Difference between stored procedure and function

Developers only need to add the display command then the variable or value they want to display in order to use print statements. Developers can use the following code, for instance, to display a value that a stored procedure should have returned but did not:

Print @myValue

Setting breakpoints

Setting breakpoints is another efficient technique for troubleshooting stored processes. Breakpoints allow programmers to halt the execution of a stored function at a specific point in order to examine the state of the code. This method is particularly

useful when dealing with complicated stored procedures that have numerous steps and conditions.

Using Functions in Queries:

It is possible for SQL functions to take input parameters, carry out a series of actions, and then return a result. Functions can be used in SQL queries to improve query performance, reduce code duplication, and simplify and streamline complicated processes. The use of user-defined functions in SQL queries, including how to send parameters to them and handle return values, will be covered in this section.

Creating User-Defined Functions:

A user-defined function must first be made before it can be used in a SQL query. The CREATE FUNCTION statement, which defines the function name, input parameters, and return data type, can be used to create a user-defined function. Any legitimate SQL code, such as SELECT, UPDATE, and DELETE commands, may be included in the function body.

Passing Parameters to Functions:

Functions take a number of input parameters — zero or more — that can be used to change how they behave. When calling a function, we can define parameters by enclosing them in parentheses. The amount of parameters and their corresponding data types must line up with the function's definition.

For example, consider the following function definition:

CREATE FUNCTION dbo.GetTotalSales(@StartDate DATE, @EndDate DATE)

RETURNS MONEY AS BEGIN DECLARE @TotalSales MONEY; SELECT @TotalSales = SUM(OrderTotal) FROM Orders WHERE OrderDate BETWEEN @StartDate AND @EndDate; RETURN @TotalSales; END

This function takes two input parameters, @StartDate and @EndDate, which are used to filter the Orders table and calculate the total sales within the specified date range. To call this function and pass the required parameters, we can use the following syntax:

SELECT dbo.GetTotalSales('2022-01-01', '2022-12-31') AS TotalSales;

This will return the total sales for the year 2022.

Handling Return Values:

Any data type, including scalar values like integers, floats, and strings, as well as more sophisticated data types like tables and cursors, can be returned by a function as a singular value. We can either give the return value of a function to a variable or use it directly in an expression to handle it in a SQL query.

For example, consider the following function definition:

CREATE FUNCTION dbo.GetCustomerOrders(@CustomerID INT)

RETURNS TABLEAS RETURN

(SELECT * FROM Orders WHERE CustomerID = @CustomerID);

This function returns a table of all orders placed by a given customer. To use this function in a SQL query, we can call it as follows:

```
DECLARE @Orders TABLE (OrderID INT, OrderDate DATE, OrderTotal MONEY);
INSERT INTO @Orders
SELECT OrderID, OrderDate, OrderTotal
FROM dbo.GetCustomerOrders(123);
SELECT * FROM @Orders;
```

This will insert the results of the dbo.GetCustomerOrders function into a temporary table @Orders, which can then be used in subsequent queries.

Using user-defined functions in SQL queries can greatly simplify and streamline complex operations, allowing for improved query performance and reduced code duplication. By mastering the use of functions, developers can create more efficient and maintainable SQL code.

Conclusion

Finally, mastering SQL stored procedures and functions is a valuable talent for anyone who works with databases. Stored procedures and functions can improve database speed, simplify maintenance, and increase security by encapsulating and reusing SQL code. User-defined functions can also be used in SQL queries to simplify complex operations and minimise code duplication. Understanding how to build, call, and manage stored procedures and functions, as well as using user-defined functions in SQL queries, can result in more efficient and maintainable SQL code. Developers and administrators can use this knowledge to better manage and utilise the power of databases to meet the requirements of their organisations.

REFERENCES:

1. Patrick O'Neil and Elizabeth O'Neil, Database Principles, Programming and Performance, Harcourt Asia Pte. Ltd., First Edition, 2001.

2. Peter Rob and Carlos Coronel, Database Systems Design, Implementation and Management, Thomson Learning-Course Technology, Seventh Edition, 2007.

3. C. J. Date, A. Kannan and S. Swamynathan, An Introduction to Database Systems, Pearson Education, Eighth Edition, 2009.

4. Abraham Silberschatz, Henry F. Korth and S. Sudarshan, Database System Concepts, McGraw-Hill Education (Asia), Fifth Edition, 2006.

5. Atul Kahate, Introduction to Database Management Systems, First Edition, 2006