International Journal of Science and Research (IJSR)
ISSN: 2319-7064
ResearchGate Impact Factor (2018): 0.28 | SJIF (2018): 7.426

Optimizing Stored Procedures: Advanced Techniques for Improved SQL Performance

Vishnupriya S Devarajulu

Email: *vishnupriyasupriya[at]gmail.com*

Abstract: ***For a good Database performance, Optimizing the Stored Procedures plays a crucial role. This article provides key solutions for common issues encountered in Stored Procedure Optimization and covers the best practices such as indexing, efficient joins, minimizing cursor usage, using schema names with object names, using try-catch for error handling etc. Code samples are provided to each solution, making it easier to understand and implement these techniques.***

Keywords: Database Optimization, Stored Procedures, SQL Performance, Indexing, Joins, TRY-CATCH, SET NOCOUNT ON, Cursors, Schema Names, SQL Best Practices

1. Introduction

Stored procedures are the most fundamental and integral component of Database management systems, allowing efficient execution of complex queries and business logic directly within the Database. However, poorly optimized stored procedures can lead to major performance issues, such as slow query execution, heavy resource consumption, and other bottlenecks. This article provides practical solutions and best practices to optimize stored procedures. By following these techniques, Database admins and developers can improve the performance of their stored procedures, for a more efficient and responsive Database system.

2. Key Solutions and Best Practices

Include SET NOCOUNT ON Statement: With every SELECT and DML statement, the SQL server returns a message that indicates the number of affected rows by that statement. This information is mostly helpful in debugging the code, but it is useless after that.

Solution: By setting SET NOCOUNT ON, we can disable the feature of returning this extra information. For stored procedures that contain several statements or contain Transact-SQL loops, setting SET NOCOUNT to ON can provide a significant performance boost.

Code Sample:

```
CREATE PROCEDURE GetCustomerOrders
    @customerId INT
AS
BEGIN
    SET NOCOUNT ON;
    SELECT order_id, order_date, total_amount
    FROM order WHERE customer_id = @customerId;
END;
```

Use Schema Name with Object Name: Not specifying the schema name can lead to performance issues due to additional time required for name resolution and potential ambiguity.

Solution: Always use the schema name when referring to database objects.

Code Sample:

```
-- Without schema name
SELECT order_id, order_date FROM orders;

-- With schema name
SELECT order_id, order_date FROM dbo.orders;
```

Do Not Use the Prefix "sp_" in the Stored Procedure Name: The prefix "sp_" is used for system stored procedures in SQL Server and can lead to name resolution delays.

Solution: Avoid using the "sp_" prefix for user-defined stored procedures.

Code Sample:

```
-- Avoid using sp_ prefix
CREATE PROCEDURE sp_GetCustomerOrders

-- Use a different prefix or naming convention
CREATE PROCEDURE usp_GetCustomerOrders
```

Use IF EXISTS (SELECT 1) Instead of (SELECT): Using SELECT * can be less efficient as it retrieves all columns, even when checking for the existence of records.

Solution: Use IF EXISTS (SELECT 1) to check for the existence of rows more efficiently.

Code Sample:

```
-- Less efficient
IF EXISTS (SELECT * FROM dbo.orders WHERE order_id = 1234)

-- More efficient
IF EXISTS (SELECT 1 FROM dbo.orders WHERE order_id = 1234)
```

Use Proper Indexing: Without proper indexing, the database engine must perform a full table scan to retrieve the required data, which is time-consuming for large datasets.

Solution: Ensure that indexes are created on columns frequently used in WHERE, JOIN, and ORDER BY clauses.

Code Sample:
```
CREATE INDEX idx_customer_id ON orders (customer_id);
```

Avoid Using Cursors: Cursors can be slow because they process each row individually, leading to high resource consumption for large result sets.

Solution: Replace cursors with set-based operations whenever possible.
```
-- Efficient set-based operation
UPDATE customers
SET status = 'active'
WHERE last_order_date > '2019-01-01';
```

Use Join Techniques: Unnecessary joins such as cross joins or missing join conditions, can lead to extremely large result sets and slow query performance.

Solution: Use inner, left, right joins and ensure proper join conditions are specified.

Code Sample:
```
-- Inefficient join
SELECT * FROM orders, customers
WHERE orders.customer_id = customers.customer_id;

-- Efficient join
SELECT * FROM orders
INNER JOIN customers ON orders.customer_id = customers.customer_id;
```

Optimize the Use of Temporary Tables: Overusing or improperly indexing temporary tables can lead to performance issue's due to excessive I/O operations.

Solution: Limit the use of temporary tables and make sure they are indexed appropriately

Code Sample:
```
-- Inefficient temporary table usage
CREATE TEMPORARY TABLE temp_orders AS
SELECT * FROM orders WHERE order_date > '2019-01-01';

-- Efficient temporary table usage with indexing
CREATE TEMPORARY TABLE temp_orders (INDEX idx_order_date (order_date))
AS SELECT * FROM orders WHERE order_date > '2019-01-01';
```

Minimize the Use of Scalar Functions in SELECT Clauses: Scalar functions in SELECT clauses can lead to row-by-row processing, reducing query performance.

Solution: Avoid scalar functions in SELECT clauses and use inline calculations or joins instead.

Code Sample:
```
-- Inefficient use of scalar function
SELECT order_id, dbo.CalculateOrderTotal(order_id) FROM orders;

-- Efficient use with inline calculation
SELECT order_id, SUM(quantity * price) AS total
FROM order_items GROUP BY order_id;
```

Use Query Execution Plans: Not analyzing query execution plans can lead to potential performance issues in query execution.

Solution: Use execution plans to identify and address slow-performing parts of your queries.

Code Sample:
```
-- Get the execution plan for a query
EXPLAIN SELECT * FROM orders WHERE customer_id = 1234;
```

Avoid Select *: When used SELECT * it retrieves all columns from a table, including unnecessary ones, and reduces performance.

Solution: Select only the necessary columns.

Code Sample:
```
-- Inefficient query
SELECT * FROM orders WHERE order_id = 1234;

-- Efficient query
SELECT order_id, customer_id, order_date
FROM orders WHERE order_id = 1234;
```

3. Conclusion

Optimizing stored procedure is essential for maintaining a high-performing and efficient database system. By implementing best practices and techniques discussed in this article we can significantly improve query execution times, resource utilization, and overall database efficiency, leading to a more responsive and reliable system.

References

[1] Microsoft SQL Server Documentation: docs.microsoft.com/sql
[2] Oracle Database Performance Tuning Guide: oracle.com/database
[3] SQLServerCarpenter: https://sqlservercarpenter.com/2018/07/17/best-practices-for-stored-procedures-in-sql-server/
[4] Blog SQL Authority: https://blog.sqlauthority.com/2010/02/16/sql-server-stored-procedure-optimization-tips-best-practices/