**Use Indexes Wisely:**

Properly indexed tables can significantly improve query performance. Analyze your query execution plans to identify where indexes can be beneficial and create them accordingly. However, avoid over-indexing as it can impact insert, update, and delete performance.

**Write Clear and Readable Code:**

Use indentation, line breaks, and comments to make your SQL code easy to read and understand. This not only helps others who might need to review or modify your code but also aids in debugging and troubleshooting.

**Avoid Using SELECT \*\*\*:**

Instead of selecting all columns using SELECT \*, explicitly specify the columns you need. This reduces unnecessary data transfer and can improve query performance, especially when dealing with large tables.

**Avoid Cursors Whenever Possible:**

Cursors can be resource-intensive and may lead to poor performance, especially for large datasets. Whenever possible, use set-based operations instead of cursors.

**Parameterize Queries:**

Use parameterized queries to prevent SQL injection attacks and improve query plan caching and reuse. Parameterization also helps improve performance by reducing the need for SQL Server to compile similar queries repeatedly.

**Avoid Nested Queries**:

While nested queries can be convenient, they can also be less efficient than equivalent joins. Whenever possible, use JOIN clauses to retrieve data from multiple tables.

**Optimize Joins:**

Ensure that join conditions are properly indexed and that you're using the appropriate join type (e.g., INNER JOIN, LEFT JOIN, etc.) based on your data requirements.

**Minimize the Use of DISTINCT:**

While sometimes necessary, using DISTINCT can impact query performance. Consider whether it's truly needed or if there are alternative approaches to achieve the same result.

**Limit the Use of Functions in WHERE Clauses:**

Functions in WHERE clauses can prevent the optimizer from using indexes efficiently. Try to avoid using functions on columns in the WHERE clause, especially when those columns are indexed.

**Regularly Analyze and Optimize Queries**:

Use SQL Server's query execution plans, Pro-filer, and Database Engine Tuning Advisor to analyze and optimize your queries regularly. Monitor query performance and address any bottlenecks as they arise.

**Use Transactions Appropriately:**

When performing multiple related operations, wrap them in a transaction to ensure data consistency and integrity. However, keep transactions as short as possible to minimize blocking and improve concurrency.

**Regular Maintenance:**

Regularly perform database maintenance tasks such as updating statistics, rebuilding indexes, and shrinking database files to ensure optimal performance.

**Use Meaningful Names:**

Give your stored procedures descriptive names that reflect their purpose and functionality. This makes it easier for developers to understand their role in the database.

**Parameterize Inputs**:

Just like with queries, parameterize inputs in stored procedures to prevent SQL injection attacks and improve performance through query plan caching and reuse.

**Avoid Dynamic SQL:**

Whenever possible, avoid using dynamic SQL in stored procedures. Dynamic SQL can be less efficient and more prone to SQL injection. If dynamic SQL is necessary, use parameterized queries within the dynamic SQL string.

**Error Handling:**

Implement robust error handling within your stored procedures to gracefully handle exceptions and provide meaningful error messages. Utilize TRY...CATCH blocks to catch errors and handle them appropriately.

**Transaction Management:**

Use transactions within stored procedures to ensure data consistency and integrity, especially when performing multiple related operations. Begin and commit/rollback transactions as needed to maintain ACID properties.

**Avoid Heavy Processing:**

While it's possible to perform complex calculations or heavy processing within stored procedures, consider whether this logic can be better handled by the application layer or through other means. Keep stored procedures focused on database operations.

**Modularize**:

Break down complex procedures into smaller, modular components to improve readability, maintainability, and reusability. Consider using user-defined functions or nested stored procedures to encapsulate common logic.

**Testing and Validation:**

Thoroughly test your stored procedures to ensure they function as expected under various scenarios. Validate input parameters and test edge cases to uncover any potential issues before deploying to production.

**Permissions:**

Ensure that appropriate permissions are set for executing stored procedures. Grant EXECUTE permission only to users or roles that need to execute the procedures.

**Documentation:**

Document your stored procedures thoroughly, including their purpose, input parameters, output format, and any side effects. This helps other developers understand how to use the procedures and troubleshoot issues.

**Optimization:**

Regularly review and optimize your stored procedures for performance. Use SQL Server's query execution plans and performance monitoring tools to identify bottlenecks and optimize accordingly.

**Version Control:**

Store your stored procedures (along with other database objects) in version control to track changes, facilitate collaboration, and ensure a reliable backup of your database schema.