# **T-SQL Coding Standards**

## **Purpose**

Database coding standards are crucial for several reasons:

1. **Consistency:**
   * Consistent coding practices make our data environment easier to maintain and understand. It reduces the learning curve for new team members and allows them to contribute more quickly.
2. **Readability:**
   * Clear and consistent code is easier to read and understand. This is particularly important given the number of people and teams who access the different UMA datastores.
3. **Efficiency:**
   * Well-structured and optimized code can significantly improve the performance of database operations.
4. **Reliability:**
   * Following best practices helps to avoid common errors and bugs, increasing the reliability of data operations.
5. **Security:**
   * Adhering to coding standards can help to prevent security vulnerabilities, such as SQL injection attacks.
6. **Collaboration:**
   * When everyone follows the same standards, it’s easier to collaborate and review each other’s work.
7. **Scalability:**
   * Good coding practices can make it easier to scale the data store as the size of the data and the number of users increase.
8. **Maintainability:**
   * Consistent and well-documented code is easier to maintain and troubleshoot. This helps in long-term maintenance and reduces technical debt.
9. **Compliance:**
   * Following industry standards and best practices can ensure compliance with regulatory requirements and industry standards.
10. **Documentation:**
    * Standardized documentation practices ensure that the purpose and function of code are well understood by current and future team members.
11. **Testing:**
    * Adhering to coding standards makes it easier to write and execute automated tests, ensuring that the code works as intended and reducing the risk of introducing bugs.

## **Names**

Tables, views, columns, and other database objects should have clear and descriptive names. Names should accurately reflect the purpose and content of the object. For more detailed guidelines, refer to the documentation created by Jeff Harris

Do not use spaces in object names. Use underscores (\_) or camelCase (start with a lowercase letter and capitalize the first letter of each subsequent word without spaces or underscores) to separate words..

Always capitalize SQL reserved words (e.g., SELECT, FROM, JOIN, WHERE). This enhances readability and distinguishes SQL keywords from object names Avoid using reserved words as object names

# Always explicitly list column names in SELECT statements and avoid using SELECT \* to retrieve all columns from a table. This practice enhances query performance, reduces network overhead, and ensures clarity in data retrieval by explicitly stating the columns needed

# Always use two-part names when referencing database objects, such as schema.table or table.column. This practice helps to specify the exact location of the object within the database schema, ensuring clarity and avoiding potential naming conflicts

# **Indentation and Formatting**

Always start each line of code with consistent indentation (e.g., four spaces or a tab), to make it easier to read and understand

Use proper spacing between keywords, operators, and names to enhance clarity

Break long SQL statements into multiple lines for readability.

Use C style comments /\* \*/

Include comments to explain complex queries, business logic, or any unusual code.

Comments should be concise, meaningful, and written in clear language.

Update comments when modifying the code to keep them relevant.

End each statement with a semicolon

Do not comment out code as version control

Use the RedGate formatting functionality

## 

## **Data Types**

Use appropriate data types for columns based on the nature of the data. Use the smallest viable datatype

USE datetime2 rather than datetime, unless you need that level of precision (you probably don't)

Avoid using generic data types like VARCHAR(MAX) when a specific length is known.

Be mindful of data type compatibility when joining tables or performing operations.

## **Consistency in Syntax**

Choose a consistent syntax style (e.g., ANSI SQL or vendor-specific) and stick to it.

Avoid mixing syntax styles within the same codebase.

## 

## **Error Handling**

Implement error handling mechanisms such as TRY...CATCH blocks to handle exceptions gracefully.

Provide meaningful error messages to aid troubleshooting.

If a status or error email is to be sent, it should be addressed to a Security Group, rather than an individual employee.

## **Performance Considerations**

Every table gets a Primary Key/ Clustered Index. No exceptions.

Optimize SQL queries for performance by avoiding unnecessary joins, subqueries, and functions.

Use indexes appropriately to speed up data retrieval operations.

Avoid using SELECT

Test the performance impact of queries before deploying them in production.

## **Remote Queries**

Retrieve only the necessary data to minimize network overhead and improve query performance.

Ensure that columns involved in JOIN operations or WHERE clause predicates are indexed on both local and remote servers to optimize query performance

Retrieve the smallest amount of data from the database:

* Select only the most necessary columns.
* Apply filters to fetch only the specific rows you nee

Prefer using OPENQUERY() for executing queries against Linked Servers when feasible, as it can provide better query optimizatio

**Specific Standards for Views**

1. **Structure**:
   * Define columns explicitly.
   * Avoid complex logic in views; keep them simple.
   * Use aliases for columns to ensure consistency.
2. **Example Template**:

CREATE VIEW v\_ViewName

AS

SELECT

Column1 AS Alias1,

Column2 AS Alias2,

...

FROM

TableName

WHERE

Condition

When creating views, ensure to select only the necessary data to minimize retrieval. Each level of your query should strictly select what's needed.

Avoid using columnstore indexes unless you can justify their necessity based on specific performance requirements

Do not use Functions in Predicates, particularly when working with large tables

eg. WHERE CAST(tbl1.ModifiedDateTime AS Date) = tbl2.CreatedDateTime

**Specific Standards for Functions**

1. **Structure**:
   * Ensure functions are deterministic where possible.
   * Document the purpose and usage of the function.
   * Handle NULL values appropriately.
2. **Example Template**:

sql

Copy code

CREATE FUNCTION fn\_FunctionName

(@Param1 INT)

RETURNS INT

AS

BEGIN

DECLARE @Result INT;

-- Function logic here

SET @Result = @Param1 + 1;

RETURN @Result;

END;

**Specific Standards for Indexes**

1. **Naming Conventions**:
   * Use a consistent naming convention for indexes (e.g., IX\_TableName\_ColumnName).
2. **Example Template**:

sql

Copy code

CREATE INDEX IX\_TableName\_ColumnName

ON TableName (ColumnName);

Avoid using IN. Try EXISTS instead.

Do not use GOTO

Avoid using wild card characters at the beginning of a value when using the LIKE keyword.

Don’t use NOLOCK. It doesn’t do what you think it does. If dirty reads are acceptable in your results, use SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED.

## **Stored Procedures**

1. **Structure**:
   * Define input and output parameters clearly.
   * Use SET NOCOUNT ON; at the beginning to prevent extra result sets from interfering with SELECT statements.
   * Include input validation at the start of the procedure.
2. **Example Template**:

sql

CREATE PROCEDURE usp\_ProcedureName

@Param1 INT,

@Param2 NVARCHAR(50)

AS

BEGIN

SET NOCOUNT ON;

-- Validate input parameters

IF @Param1 IS NULL OR @Param2 IS NULL

BEGIN

RAISERROR('Invalid input parameters', 16, 1);

RETURN;

END

BEGIN TRY

-- Main logic here

END TRY

BEGIN CATCH

-- Error handling here

DECLARE @ErrorMessage NVARCHAR(4000);

SET @ErrorMessage = ERROR\_MESSAGE();

RAISERROR(@ErrorMessage, 16, 1);

END CATCH

END;

1. Parameters and variables should match the datatype of the columns being queried.
2. Avoid unnecessary function calls (eg. UPPER()).

## **Security**

Implement security best practices such as parameterized queries to prevent SQL injection attacks.

Limit access to sensitive data by granting appropriate permissions to database users.

## **Version Control**

Store SQL scripts in version control systems like Git to track changes and facilitate collaboration.

Use descriptive commit messages when modifying SQL scripts.

Commenting out code to “preserve” it is not version control.

## **JOINs**

Prefer explicit JOIN syntax (e.g., INNER JOIN, LEFT JOIN) over implicit joins.

Use table aliases to improve readability, especially when joining multiple tables.

Document the join conditions, especially in complex queries, to aid understanding.

Avoid Cartesian joins (cross joins) unless explicitly needed, and ensure proper filtering criteria are applied.

Consider performance implications when choosing between different join types, especially in large datasets.

## **MERGE**

For working with large datasets, investigate the performance differences between MERGE and separate UPDATE/INSERT statements that utilize EXISTS/NOT EXISTS. The differences can sometimes be significant.

When using MERGE, clearly define the target table and the source data, ensuring data types and column mappings align correctly.Include error handling mechanisms to handle potential conflicts or errors during the merge operation.

Document the logic and conditions used in the merge statement for clarity and future maintenance.

## **Cursors and Loops**

Minimize the use of cursors and WHILE loops whenever possible, as they are generally slower compared to set-based operations.

Consider alternatives such as set-based operations (e.g., using UPDATE or INSERT...SELECT) or temporary tables for cursor replacements.

When cursors are necessary, use FORWARD\_ONLY and STATIC cursor types whenever possible to optimize performance.

Close and deallocate cursors explicitly to release resources when they are no longer needed.

Document the cursor logic and purpose clearly to aid understanding and maintenance.

## **Ad-hoc Queries**

Limit the use of ad-hoc queries in production environments due to potential security risks and performance implications.

Encourage parameterized queries to prevent SQL injection attacks and improve query plan caching.

Consider creating stored procedures or views for frequently used ad-hoc queries to promote code reuse and centralize query logic.

Document the purpose, usage, and any security considerations for ad-hoc queries.

Perform thorough testing, including edge cases, when using ad-hoc queries in critical systems to ensure correctness and performance.

Adhering to these SQL code standards when using joins, mergers, cursors, and ad-hoc queries can help maintain consistency, readability, and performance across database codebases while minimizing potential risks and issues.

## **STORED PROCEDURE TEMPLATE**

SET ANSI\_NULLS ON

GO

SET QUOTED\_IDENTIFIER ON

GO

/\*

-- =============================================

-- Author: <Author,,Name>

-- Create date: <Create Date,,>

-- Description: <Description,,>

-- =============================================

--Change History

--Date Changed by Description

\*/

CREATE PROCEDURE <Procedure\_Name, sysname, ProcedureName>

-- Add the parameters for the stored procedure here

<@Param1, sysname, @p1> <Datatype\_For\_Param1, , int> = <Default\_Value\_For\_Param1, , 0>,

<@Param2, sysname, @p2> <Datatype\_For\_Param2, , int> = <Default\_Value\_For\_Param2, , 0>

AS

BEGIN

SET NOCOUNT ON;

SET ARITHABORT ON;

BEGIN TRY

BEGIN TRAN

-- Insert statements for procedure here

COMMIT TRAN

END TRY

BEGIN CATCH

IF @@TRANCOUNT > 0

ROLLBACK TRAN

DECLARE @ErrorMessage NVARCHAR(4000);

DECLARE @ErrorSeverity INT;

DECLARE @ErrorState INT;

SELECT @ErrorMessage = ERROR\_MESSAGE(),

@ErrorSeverity = ERROR\_SEVERITY(),

@ErrorState = ERROR\_STATE();

RAISERROR(@ErrorMessage, @ErrorSeverity, @ErrorState );

END CATCH

END

GO

**Temporary Tables**

**Naming Conventions**

* Prefix temporary tables with # for local temporary tables and ## for global temporary tables.
* Use meaningful and descriptive names, even for temporary tables.

**Usage Guidelines**

1. **When to Use Temporary Tables**:
   * Use temporary tables for storing intermediate results in complex queries.
   * When you need to index the intermediate results.
   * When dealing with a large dataset that needs to be processed multiple times within a stored procedure.
2. **Creating Temporary Tables**:
   * Define columns explicitly.
   * Consider indexing temporary tables if they are large and will be queried multiple times.
3. **Example**:

sql

CREATE TABLE #TempTable (

Column1 INT,

Column2 NVARCHAR(50),

Column3 DATETIME

);

INSERT INTO #TempTable (Column1, Column2, Column3)

SELECT Column1, Column2, Column3

FROM SourceTable

WHERE Condition;

-- Create indexes if needed

CREATE INDEX IX\_TempTable\_Column1 ON #TempTable(Column1);

-- Use the temporary table

SELECT \* FROM #TempTable WHERE Column1 = 1;

-- Drop the temporary table at the end of the procedure

DROP TABLE #TempTable;

**Best Practices**

* **Scope and Lifetime**: Remember that local temporary tables (#TempTable) are scoped to the session and batch. They are automatically dropped when the session ends.
* **Performance**: Index temporary tables if they will be used in join operations or if they contain a large number of rows.
* **Concurrency**: Avoid global temporary tables (##TempTable) unless necessary due to potential conflicts with other sessions.

**Table Variables**

**Naming Conventions**

* Prefix table variables with @ and use camelCase or PascalCase for the name.
* Use meaningful and descriptive names.

**Usage Guidelines**

1. **When to Use Table Variables**:
   * Use table variables for storing a small number of rows.
   * When the table will not require indexing beyond primary keys.
   * For simple, lightweight temporary storage, especially in functions.
2. **Creating Table Variables**:
   * Define columns explicitly.
   * Table variables can have primary keys, unique constraints, and default values but no non-clustered indexes.
3. **Example**:

sql

DECLARE @TempTableVariable TABLE (

Column1 INT PRIMARY KEY,

Column2 NVARCHAR(50),

Column3 DATETIME

);

INSERT INTO @TempTableVariable (Column1, Column2, Column3)

SELECT Column1, Column2, Column3

FROM SourceTable

WHERE Condition;

-- Use the table variable

SELECT \* FROM @TempTableVariable WHERE Column1 = 1;

**Best Practices**

* **Scope and Lifetime**: Table variables are scoped to the batch, stored procedure, or function in which they are declared.
* **Performance**: Table variables are generally stored in memory but may spill to disk if they become too large. They lack statistics, so they may lead to suboptimal query plans for complex queries.
* **Concurrency**: Table variables are isolated to the session and do not conflict with other sessions.

**Choosing Between Temporary Tables and Table Variables**

1. **Size of Data**:
   * Use table variables for small datasets.
   * Use temporary tables for larger datasets or when you need to index the data for performance.
2. **Complexity and Operations**:
   * Use table variables for simple operations and when the dataset is small.
   * Use temporary tables for complex queries, multiple joins, and large datasets.
3. **Performance Considerations**:
   * Temporary tables support indexing and statistics, which can improve performance for complex operations.
   * Table variables have less overhead for small datasets but may lead to inefficient query plans for larger or more complex datasets.

By following these guidelines and best practices, you can effectively use temporary tables and table variables to optimize the performance and maintainability of your SQL Server code