Blog

Listed by date

March, 2007
February, 2007
January, 2007
December, 2006
November, 2006
October, 2006
September, 2006
August, 2006
July, 2006
June, 2006
May, 2006
April, 2006
March, 2006
February, 2006
January, 2006
December, 2005
November, 2005
October, 2005
September, 2005
August, 2005
July, 2005
June, 2005
May, 2005
April, 2005
March, 2005
February, 2005
January, 2005
December, 2004

◂ Previous entry: TECH: Collec... Next entry: TECH: How Co... ▸

March 19

TECH: T-SQL Standard
TECH: T-SQL Standard

From: http://msdn.microsoft.com/sql/default.aspx?pull=/library/en-us/dnsqlpro04/html/sp04l9.asp

T-SQL Coding Standards

Brian Walker

Surprising as it may be, there don't seem to be any "official" T-SQL coding standards. Back in late 1999, I was pleased to discover John Hindmarsh's proposed standards for SQL Server 7.0 and summarized some of his recommendations in a February 2000 editorial. (John's original standards were included in the February 2000 Downloads and are also included in this month's.) More recently, Ron Talmage has written a series of columns with his recommendations for a variety of "best practices," and, of course, the SQL Server team has officially shipped a Best Practices Analyzer for SQL Server (SQLBPA). Now, Brian Walker, a DBA and application developer with more than 25 years of experience, weighs in with advice and tips.

Coding standards are often overlooked when it comes to T-SQL programming, but they're a crucial component of a smoothly operating development team. The coding standards I espouse here are ones that I've developed over the years. No, they're not universally accepted, and, admittedly, some of them are subjective. My message is really more about raising awareness than promoting myself as an arbiter of T-SQL style: The most important thing is to establish some reasonable coding standards and adhere to them. What you'll find in this article is a series of miscellaneous coding standards, hints, and tips for T-SQL programming. They're not arranged in any particular order of priority or importance.

Let's start with formatting. On the surface, the formatting of T-SQL code may not seem that important, but consistent formatting makes it easier for colleagues—be they fellow team members or members of the larger worldwide T-SQL fraternity—to scan and understand your code. T-SQL statements have a structure, and having that structure be visually evident makes it much easier to locate and verify various parts of the statements. Uniform formatting also makes it much easier to add sections to and remove them from complex T-SQL statements for debugging purposes. Here's a formatting example for a SELECT statement:

```
    SELECT C.Name
         , E.NameLast
         , E.NameFirst
         , E.Number
         , ISNULL(I.Description,'NA') AS Description
      FROM tblCompany AS C
      JOIN tblEmployee AS E
        ON C.CompanyID = E.CompanyID
LEFT JOIN tblCoverage AS V
        ON E.EmployeeID = V.EmployeeID
LEFT JOIN tblInsurance AS I
        ON V.InsuranceID = I.InsuranceID
     WHERE C.Name LIKE @Name
       AND V.CreateDate > CONVERT(smalldatetime,
       '01/01/2000')
 ORDER BY C.Name
         , E.NameLast
         , E.NameFirst
         , E.Number
         , ISNULL(I.Description,'NA')

SELECT @Retain = @@ERROR, @Rows = @@ROWCOUNT

IF @Status = 0 SET @Status = @Retain
```

▶ Use four-space indentation for statements within a nested block of code. (The multi-line SELECT statement in the preceding code is a single SQL statement.) Right-align SQL keywords when starting a new line within the same statement. Configure the code editor to use spaces instead of tabs. This makes the formatting consistent regardless of what program is used to view the code.

▶ Capitalize all T-SQL keywords, including T-SQL functions. Use mixed case for variable names and cursor names. Use lowercase for data types.

▶ Keep table name aliases short, but as meaningful as possible. In general, use the capital letters of a table name as an alias and use the AS keyword to specify aliases for tables or fields.

▶ Always qualify field names with table name aliases when there are multiple tables involved in a T-SQL statement. This adds clarity for others and avoids ambiguous references.

▶ Align related numbers into columns when they appear in contiguous lines of code, such as with a series of SUBSTRING function calls. This makes the list of numbers easier to scan.

▶ Use single (not double) blank lines to separate logical pieces of T-SQL code, and do so liberally.
▶ Use appropriate data type declarations and consistent capitalization when declaring T-SQL local variables (such as @lngTableID).

▶ *Always* specify the length of a character data type, and make sure to allow for the maximum number of characters users are likely to need, since characters beyond the maximum

length are simply lost.

► *Always* specify the precision and scale of the decimal data type, which otherwise defaults to an unspecified precision and integer scale.

► Employ error handling, but bear in mind that the error-checking examples in BOL don't work as advertised. The T-SQL statement (IF) used to check the @@ERROR system function actually clears the @@ERROR value in the process and will never capture a value other than zero. (Even if the examples did work, they'd only capture the last error that occurred rather than what is likely to be of more interest—the first error.) The error code must be captured immediately using SET or SELECT as in the preceding example. It should then be transferred to a status variable if the status variable is still zero.

► Avoid using "undocumented" features such as undocumented columns in system tables, undocumented functionality in T-SQL statements, or undocumented system or extended stored procedures.

► Do *not* rely upon any implicit data type conversions. For example, don't assign a character value to a numeric variable assuming that T-SQL will do the necessary conversion. Instead, use the appropriate CONVERT function to make the data types match before doing a variable assignment or a value comparison. Another example: Although T-SQL does an implicit and automatic RTRIM on character expressions before doing a comparison, don't depend on that behavior because compatibility level settings and use of nchar complicate things.

► Do *not* directly compare a variable value to NULL with comparison operators (symbols). If the variable could be null, use IS NULL or IS NOT NULL to do a comparison, or use the ISNULL function.

► Don't use the STR function to perform any rounding; use it with integers only. Use the CONVERT function (going to a different scale) or the ROUND function before converting to a string if the string form of a decimal value is needed. CEILING and FLOOR are other options.

► Be careful with mathematical formulas, since T-SQL may force an expression into an unintended data type. Add point zero (.0) to integer constants if a decimal result is desired.

► Never depend on a SELECT statement returning rows in any particular order unless the order is specified in an ORDER BY clause.

► In general, use an ORDER BY clause with any SELECT statement. A predictable order, even if it's not the most convenient one, is better than an unpredictable order, especially during development or debugging. (You may want to remove the ORDER BY clause before deployment to production.) In those situations where the order of the resulting rows really doesn't matter, don't bother with the overhead of an ORDER BY.

► Don't ever use double quotes in your T-SQL code. Use single quotes for string constants. If it's necessary to qualify an object name, use (non-ANSI SQL standard) brackets around the name.

► As much as possible, use table variables in place of temporary tables with SQL Server 2000. If the table variable will contain a large set of data, be aware that indexing is very limited (primary key index only).

► Create temporary tables early in the routine, and explicitly drop them at the end. Interspersed DDL and DML statements can contribute to excessive recompile activity.

► Recognize that temporary tables are *not* inherently evil, and considered use of them can make some routines much more efficient—for example, when a certain set of data from a large or heavily used table will be referenced repeatedly. However, for a one-off, a derived table may be a better choice.

► Be cautious with table-valued UDFs, because passing in a parameter with a variable, rather than a constant, can result in table scans if the parameter is used in a WHERE clause. And avoid using the same table-valued UDF more than once in a single query. Table-valued UDFs do, however, have some dynamic compilation features that can be handy. *[Related: See Tom Moreau's use of a UDF to populate a table variable in his November 2003 column on generating sequence numbers.—Ed.]*

► Almost every stored procedure should have SET NOCOUNT ON near the beginning, and it's good form to have SET NOCOUNT OFF near the end. *[SET NOCOUNT ON eliminates SQL Server's sending DONE_IN_PROC messages to the client for each statement in a stored procedure.—Ed.]* This standard also applies to triggers.

► You should consider declaring an explicit transaction any time more than one database modification statement is used in a routine. This includes a single statement executed multiple times in a loop.

► Always look for a set-based solution to a problem before implementing a cursor-based approach or a temporary table approach. A set-based approach is usually more efficient.

► Cursors, like temporary tables, aren't inherently evil. A FAST_FORWARD cursor over a small set of data will often outperform other methods of handling row-by-row processing. This is especially true if several tables must be referenced to get the necessary data. A routine that includes "running totals" in the result set often executes fastest when implemented using a cursor. If development time allows, try both a cursor-based approach and a set-based approach to see which one performs best.

► A table of sequence numbers, 1 through N, can be very handy.
► Understand how a CROSS JOIN works, and take advantage of it. For example, you can use a CROSS JOIN effectively between a table of working data and a table of sequence numbers; the result set contains a record for every combination of working data and sequence number.

► I'll wrap things up with this observation: T-SQL code tends to be concise, so if a block of code looks unwieldy or repetitive, there's probably a simpler and better method.

Conclusion

Let me know what you think of my recommendations, and feel free to e-mail me to discuss any of them—or to make suggestions for others. I invite you to think of this as an opening statement in a dialogue.

Sidebar: From Karen's February 2000 Editorial

On the standards front, there's a new, independent effort, spearheaded by SQL Server DBA John Hindmarsh, MCT, MCSE, MCDBA, which is most definitely worth your time. What John has done is write a detailed white paper outlining his recommendations for all sorts of standards associated with SQL Server. The only other similar effort I'm familiar with is Andrew Zanevsky's chapter "Format and Style" in *Transact-SQL Programming* (ISBN 1-56592-401-0). Andrew, as well as *SQL Server Professional* contributors Tom Moreau and Paul Munkenbeck, contributed to John's white paper, as did John's friend and colleague Stephen James. Here's an example of John's recommendations for writing stored procedures:

- Use SQL-92 standard join syntax.
- Use joins in preference to sub or nested queries for improved performance.
- Ensure variables and parameters match table data columns in type and size.
- Ensure all variables and parameters are used or else deleted.
- Keep temporary objects local in scope wherever possible.
- Limit use of temp tables to those created in stored procs.

- Check input parameters for validity.
- Use INSERT…SELECT in preference to SELECT…INTO, to avoid extensive locks.
- Maintain the logical unit of work requirement; don't create extensive or long-running processes where these can be shortened.
- Don't use SELECT * in any code.
- Lay out the procedure with indents, blocks, tabs, and white space–refer to the sample scripts.
- Use uppercase for T-SQL statements.
- Comment procedures extensively to ensure the processes are identified. Use line comments where these help clarify a processing step.
- Include transaction management unless the procedure is to be called from MTS processes. (Write separate procedures for MTS processes.)
- Monitor @@TRANCOUNT to determine transaction responsibility level.
- Avoid GOTO–except for error handling.
- Avoid nested procedures.
- Avoid implicit resolution of object names–make sure all objects are owned by dbo.

12:27 PM | Add a comment | Trackbacks (0) | Blog it