SQL Server Development Standards

Ron Johnson

Introduction

Let me begin by stating that I do not suggest that the contents of this paper which I loosely refer to as "*standards*" should be blindly or imperialistically imposed on SQL Server professionals. My intent is to share "*lessons learned*" over the years which have been shown to reduce errors, increase performance, and generally produce better results; I would have called this paper "*best practices*" except that I cannot provide evidence that the contents represent the "best" instead only what has worked in the past. That being said, the contents of this paper describe guidelines for a SQL Server team to adopt, in whole or in part, in order to provide uniform T-SQL and database development practices and avoid common performance pitfalls. The adopted guidelines have the additional benefit of providing new team members an initiation into the team's development methodology.

A development standards document is not simply a naming convention reference and style guide. Development standards provide the uniformity of practice that allows implementation patterns and rules to be more quickly recognized during the code review process. The utilization of uniform practices in development will also ease maintenance efforts for years to come.

The paper is organized by domain with a section for standards related to database, tables and views followed by a section for stored procedure and user-defined function development, and finally a collection of miscellaneous standard practices. This paper does not cover the topic of index creation because there are numerous other detailed sources discussing index development including my own papers.

Database, Tables, and Views

1. If the table contains alternate keys, then use unique constraints on those keys.
2. Use a uniform naming convention for all of the database objects.
3. When designing a database spend the time to consider performance optimization of the tables. It is much more difficult to change the structure of the tables after the database is populated and in use.
4. Normalize the data as necessary for your application. For most designs Third Normal Form is adequate; however, in a high-throughput system de-normalizing tables may be beneficial for performance considerations.
5. A view is a data abstraction mechanism. As such, a view should be used to provide access to specific combinations of table data to the user while protecting the underlying table from direct access. Additionally, a view can simplify queries by defining frequently used complex joins and / or calculations into an abstraction from which the data may be retrieved.

6. To Unicode or not to Unicode. I believe that the use of Unicode is a business decision to be made based on the probability that the business will need to use non-English characters within the database's current lifecycle. A major factor for the business to consider is that Unicode requires twice the storage of non-Unicode datatypes.
7. CHAR versus VARCHAR. In general, if the data is of similar size ***and*** non-nullable then use the smallest CHAR possible for storage. If, on the other hand, the length of the data is indeterminate, then use VARCHAR but define a maximum size otherwise SQL Server will use the default of 30. Again, in general, sorting a VARCHAR is faster than a CHAR.
8. Avoid triggers: enough said.
9. Always define a clustered index unless you have a good, documented reason otherwise.
10. Always define a primary key.
11. Nested views should be avoided for performance reasons.

Stored Procedures and User Defined Functions

1. Document. Thorough documentation should be a part of every developer's mantra. Use a template that includes a comments header such as the templates provided in SSMS. Include copious comments within the body of the code to explain anything that you are programming that would not be absolutely clear to another developer. More is always better.
2. Always filter the return column values. Providing a WHERE clause reduces disk and Network I/O.
3. Always list the columns in an INSERT statement to abstract the structure of the table.
4. Avoid CURSORS. If you feel compelled to use a CURSOR, then talk to a more experienced developer who may be able to show you a more efficient approach.
5. Learn how to use a Numbers / Tally table. In fact, modify the Model database so that every database created on your instance will have a Numbers / Tally table.
6. Limit the use of TempDB.
7. Avoid the NOT operator because it is not SARGable resulting in scans.
8. Avoid using WILDCARDs as the first element in a SARG because it results in scans.
9. SET NOCOUNT ON. From BOL: "eliminates the sending of DONE_IN_PROC messages to the client for each statement in the stored procedure." From this statement the reader can easily deduce the implicit performance improvement.
10. Maintain control of the database by restricting direct access to the data via SQL statements from a client application.
11. Always access tables in the same order within the data access layer to minimize Deadlock potential.
12. I like to *Scaffold* my code with debugging comments. An efficient approach to using scaffolding within the code is to include a @DEBUG parameter to the stored procedure that includes a default value of 0.
13. Use TRY…CATCH blocks for run-time error management.
14. Date values in the form of 2010/10/22 will always be correctly interpreted by SQL Server.

15. When calling a stored procedure use the FQN.

Functions

1. Avoid using user-defined functions (UDF) in WHERE statements. UDFs used in this context generally perform worse than alternative solutions. In most cases, the UDF will process the result set as if it were being processed with a CURSOR.

General Coding

1. IN versus EXISTS**.** The EXISTS clause is usually more efficient. The IN statement requires that the entire subquery dataset be processed. The EXISTS statement requires that the subquery processes until *all applicable values* are processed.
2. UNION versus UNION ALL. The UNION command performs the equivalent of a SELECT DISTINCT on the joined tables' result set which results in slower performance. If there will not be any duplication of rows created by the execution of UNION, then using the UNION ALL statement is a better alternative because the UNION ALL will never look for duplicate rows, hence the query will run much faster than UNION.
3. Use correlated sub-queries before inline queries when developing SQL code. Correlated sub-query contain a reference in the WHERE clause to the outer query which assists in limiting the result set of the sub-query. Inline queries gather a complete result set prior to join and filtering the inline query results. In some cases, inline queries execute once for every row included in a result set and may execute for more than the number of rows returned. This is in effect similar to writing a cursor within query. A correlated sub-query will create a single result set that is joined to all rows.
4. Use SET STATISTICS IO ON in order to review the I/O characteristics of your code.

Summary

As a SQL Server Development Standard this document falls blatantly and purposefully short of the mark. I do not address naming conventions, indentation, or other important stylistic imperatives that improve readability and maintainability. This document's missing elements are those that will allow you the lead DBA / developer to customize a standard for your own use through collaboration with other team members. My hope is that this document provides a starting point, a jump-start to the process of defining a comprehensive and usable SQL Server Development Standard for your team.