C#
Windows, .NET
Win32, VS
DB, Dev
Posted 9 Nov 2002
145,532 views

SQL Server DO's and DONT's

By Daniel Turini

SQL Server database best practices

Prize winner: November, 2002

159 members have rated this article. Result:
Popularity: 10.2. Rating: **4.64** out of 5.

SQL Server DO's and DON'Ts

So, you are now the leader of a SQL Server based project and this is your first one, perhaps migrating from Access. Or maybe you have performance problems with your SQL Server and don't know what to do next. Or maybe you simply want to know of some design guidelines for solutions using SQL Server and designing Database Access Layers (DAL): this article is for you.

Even if you are not using SQL Server, most of these design guidelines apply to other DBMS, too: Sybase is a very similar environment for the programmer, and Oracle designs may benefit from this too. I won't show here how to use specific T-SQL tricks, nor won't give you miracle solutions for your SQL Server problem. This is by no means a complete, closed issue. What I intend to do is give you some advices for a sound design, with lessons learned through the last years of my life, seeing the same design errors being done again and again.

DO know your tools.

Please, don't underestimate this tip. This is the best of all of those you'll see in this article. You'd be surprised of how many SQL Server programmers don't even know all T-SQL commands and all of those effective tools SQL Server has.

"What? I need to spend a month learning all those SQL commands I'll never use???" you might say. No, you don't need to. But spend a weekend at MSDN and browse through all T-SQL commands: the mission here is to learn a lot of what can and what can't be done. And, in the future, when designing a query, you'll remember "Hey, there's this command that does exactly what I need", and then you'll refer again to MSDN to see its exact syntax.

In this article I'll assume that you already know the T-SQL syntax or can find about it on MSDN.

DON'T use cursors

Let me say it again: DON'T use cursors. They should be your preferred way of killing the performance of an entire system. Most beginners use cursors and don't realize the performance hit they have. They use memory; they lock tables in weird ways, and they are slow. Worst of all, they defeat most of the performance optimization your DBA can do. Did you know that every `FETCH` being executed has about the same performance of executing a `SELECT`? This means that if your cursor has 10,000 records, it will execute about 10,000 `SELECT`s! If you can do this in a couple of `SELECT`, `UPDATE` or `DELETE`, it will be much faster.

Beginner SQL programmers find in cursors a comfortable and familiar way of coding. Well, unfortunately this lead to bad performance. The whole purpose of SQL is specifying **what** you want, not **how** it should be done.

I've once rewritten a cursor-based stored procedure and substituted some code for a pair of traditional SQL queries. The table had only 100,000 records and the stored procedure used to take 40 minutes to process. You should see the face of the poor programmer when the new stored procedure took 10 seconds to run!

Sometimes it's even faster to create a small application that gets all the data, proccess it and update the server. T-SQL was not done with loop performance in mind.

If you are reading this article, I need to mention: there is no good use for cursors; I have never seen cursors being well used, except for DBA work. And good DBAs, most of the time, know what they are doing. But, if you are reading this, you are not a DBA, right?

DO normalize your tables

There are two common excuses for not normalizing databases: performance and pure laziness. You'll pay for the second one sooner or later; and, about performance, don't optimize what's not slow. Often I see programmers de-normalizing databases because "this will be slow". And, more frequent than the inverse, the resulting design is slower. DBMSs were designed to be used with normalized databases, so design with normalization in mind.

DON'T SELECT *

This is hard to get used, I know. And I confess: often I use it; but try to specify only the columns you'll need. This will:

1. Reduce memory consumption and network bandwidth
2. Ease security design
3. Gives the query optimizer a chance to read all the needed columns from the indexes

DO know how your data will be/is being acessed

A robust index design is one of the good things you can do for your database. And doing this is almost an art form. Everytime you add an index to a table, things get faster on `SELECT`, but `INSERT` and `DELETE` will be much slower. There's a lot of work in building and mantaining indexes. If you add several indexes to a table to speed your `SELECT`, you'll soon notice locks being held for a long time while updating indexes. So, the question is: what is being done with this table? Reading or Updating data? This question is tricky, specially with the `DELETE` and `UPDATE`, because they often involve a `SELECT` for the `WHERE` part and after this they update the table.

DON'T create an index on the "Sex" column

This is useless. First, let's understand how indexes speed up table access. You can see indexes as a way of quickly partitioning a table based on a criteria. If you create an index with a column like "Sex", you'll have only two partitions: Male and Female. What optimization will you have on a table with 1,000,000 rows? Remember, mantaining an index is slow. Always design your indexes with the most sparse columns first and the least sparse columns last, e.g, Name + Province + Sex.

DO use transactions

Specially on long-running queries. This will save you when things get wrong. Working with data for some time you'll soon discover some unexpected situation which will make your stored procured crash.

DO beware of deadlocks

Always access your tables on the same order. When working with stored procedures and transactions, you may find this soon. If you lock the table A then table B, always lock them in this very same order in all stored procedures. If you, by accident, lock the table B and then table A in another procedure some day you'll have a deadlock. Deadlocks can be tricky to find if the lock sequence is not carefully designed.

DON'T open large recordsets

A common request on programming forums is: "How can I quickly fill this combo with 100,00 items?". Well, this is an error. You can't and you shouldn't. First, your user will hate browsing through 100,000 records to find the right one. A better UI is needed here, because you should ideally show no more that 100 or 200 records to your users.

DON'T use server side cursors

Unless you know what your are doing. Client side cursors often (not always) put less overhead on the network and on the server, and reduce locking time.

DO use parametrized queries

Sometimes I see in programming forums, questions like: "My queries are failing with some chars, e.g. quotes. How can I avoid it?". And a common answer is: "Replace it by double quotes". Wrong. This is only a workaround and will still fail with other chars, and will introduce serious security bugs. Besides this, it will trash the SQL Server caching system, which will cache several similar queries, instead of caching only one. Learn how to use parameterized queries (in ADO, through the use of the Command Object, or in ADO.NET the SqlCommand) and never have these problems again.

DO always test with large databases

It's a common pattern programmers developing with a small test database, and the end user using large databases. This is an error: disk is cheap, and performance problems will only be noticed when it's too late.

DON'T import bulk data with INSERT

Unless strictly necessary. Use DTS or the BCP utility and you'll have both a flexible and fast solution.

DO beware of timeouts

When querying a database, the default timeout is often low, like 15 seconds or 30 seconds. Remember that report queries may run longer than this, specially when your database grows.

DON'T ignore simultaneous editing

Sometimes two users will edit the same record at the same time. When writing, the last writer wins and some of the updates will be lost. It's easy to detect this situation: create a timestamp column and check it before you write. If possible, merge changes. If there is a conflict, prompt the user for some action.

DON'T do SELECT max(ID) from Master when inserting in a Detail table.

This is another common mistake, and will fail when two users are inserting data at the same time. Use one of `SCOPE_IDENTITY`, `IDENT_CURRENT`, and `@@IDENTITY`. Avoid `@@IDENTITY` if possible because it can introduce some nasty bugs with triggers.

DO Avoid NULLable columns

When possible. They consume an extra byte on each NULLable column in each row and have more overhead associated when querying data. The DAL will be harder to code, too, because everytime you access this column you'll need to check

I'm not saying that NULLs are the evil incarnation, like some people say. I believe they can have good uses and simplify coding when "missing data" is part of your business rules. But sometimes NULLable columns are used in situations like this:

CustomerName1
CustomerAddress1
CustomerEmail1
CustomerName2
CustomerAddress2
CustomerEmail3
CustomerName1
CustomerAddress2
CustomerEmail3

This is horrible. Please, don't do this, normalize your table. It will be more flexible and faster, and will reduce the NULLable columns.

DON'T use the TEXT datatype

Unless you are using it for really large data. The TEXT datatype is not flexible to query, is slow and wastes a lot of space if used incorrectly. Sometimes a VARCHAR will handle your data better.

DON'T use temporary tables

Unless strictly necessary. Often a subquery can substitute a temporary table. They induce overhead and will give you a big headache when programming under COM+ because it uses a database connection pool and temporary tables will last forever. In SQL Server 2000, there are alternatives like the TABLE data type which can provide in-memory solutions for small tables inside stored procedures too.

DO learn how to read a query execution plan

The SQL Server query analyzer is your friend, and you'll learn a lot of how it works and how the query and index design can affect performance through it.

DO use referential integrity

This can be a great time saver. Define all your keys, unique constraints and foreign keys. Every validation you create on the server will save you time in the future.

Conclusion

As I've said before, this is by no means a complete SQL Server performance and best practices guide. This would take a complete book to cover. But I really believe that this is a good start, and if you follow these practices, surely you will have much less trouble in the future.

About Daniel Turini

I develop software since I was 11. In the past 20 years, I developed software and used very different machines and languages, since Z80 based ones (ZX81, MSX) to mainframe computers. I still have passion for ASM, though no use for it anymore.
Professionally, I developed systems for managing very large databases, mainly on Sybase and SQL Server. Most of the solutions I write are for the financial market, focused on credit systems.
To date, I learned about 20 computer languages. As the moment, I'm in love with C# and the .NET framework, although I only can say I'm very proficient at C#, VB.NET(I'm not proud of this), T/SQL, C++ and libraries like ATL and STL.
I hate doing user interfaces, whether Web based or not, and I'm quite good at doing server side work and reusable components.
I'm the technical architect and one of the authors of Crivo, the most successful automated credit and risk assessment system available in Brazil.
You can visit my blog here.

Click here to view Daniel Turini's online profile.

Discussions and Feedback

109 comments have been posted for this article. Visit **http://www.codeproject.com/cs/database/sqldodont.asp** to post and view comments on this article.