SQL SERVER MVP DEEP DIVES

SAMPLE CHAPTER

Volume 2

EDITED BY

Kalen Delaney • Louis Davidson • Greg Low • Brad McGehee • Paul Nielsen • Paul Randal • Kimberly Tripp

MVP AUTHORS

Johan Åhlén • Gogula Aryalingam • Glenn Berry • Aaron Bertrand • Kevin G. Boles • Robert Cain • Tim Chapman • Denny Cherry • Michael Coles • Rod Colledge John Paul Cook • Louis Davidson • Rob Farley • Grant Fritchey • Darren Gosbell • Sergio Govoni • Allan Hirt • Satya Jayanty • Tibor Karaszi • Jungsun Kim • Tobiasz Koprowski • Hugo Kornelis • Ted Krueger • Matija Lah • Greg Low • Rodney Landrum • Greg Larsen • Peter Larsson • Andy Leonard • Ami Levin • John Magnabosco Jennifer McCown • Brad McGehee • Siddharth Mehta • Ben Miller • Allan Mitchell • Tim Mitchell • Luciano Moreira • Jessica Moss • Shahriar Nikkhah • Paul Nielsen Robert Pearl • Boyan Penev • Pedro Perfeito • Pawel Potasinski • Mladen Prajdić • Abolfazl Radgoudarzi • Denis Reznik • Rafael Salas • Edwin Sarmiento Chris Shaw • Gail Shaw • Linchi Shea • Jason Strate • Paul Turley • William Vaughn • Peter Ward • Joe Webb • John Welch • Allen White • Thiago Zavaschi

The authors of this book support the children of Operation Smile

SQL Server MVP Deep Dives
Volume 2

Edited by Kalen Delaney ▪ Louis Davidson ▪ Greg Low
Brad McGehee ▪ Paul Nielsen ▪ Paul Randal ▪ Kimberly Tripp

Chapter 19

Copyright 2012 Manning Publications

brief contents

PART 1 ARCHITECTURE 1

1 ■ Where are my keys? 3
2 ■ "Yes, we are all individuals"
A look at uniqueness in the world of SQL 16
3 ■ Architectural growth pains 26
4 ■ Characteristics of a great relational database 37
5 ■ Storage design considerations 49
6 ■ Generalization: the key to a well-designed schema 60

PART 2 DATABASE ADMINISTRATION 65

7 ■ Increasing availability through testing 67
8 ■ Page restores 79
9 ■ Capacity planning 87
10 ■ Discovering your servers with PowerShell and SMO 95
11 ■ Will the real Mr. Smith please stand up? 105
12 ■ Build your own SQL Server 2008 performance dashboard 111
13 ■ SQL Server cost recovery 121

14 ■ Best practice compliance with Policy-Based Management 128

15 ■ Using SQL Server Management Studio to the fullest 138

16 ■ Multiserver management and Utility Explorer—best tools for the DBA 146

17 ■ Top 10 SQL Server admin student misconceptions 157

18 ■ High availability of SQL Server in the context of Service Level Agreements 167

PART 3 DATABASE DEVELOPMENT 175

19 ■ T-SQL: bad habits to kick 177

20 ■ Death by UDF 185

21 ■ Using regular expressions in SSMS 195

22 ■ SQL Server Denali: what's coming next in T-SQL 200

23 ■ Creating your own data type 211

24 ■ Extracting data with regular expressions 223

25 ■ Relational division 234

26 ■ SQL FILESTREAM: to BLOB or not to BLOB 245

27 ■ Writing unit tests for Transact-SQL 255

28 ■ Getting asynchronous with Service Broker 267

29 ■ Effective use of HierarchyId 278

30 ■ Let Service Broker help you scale your application 287

PART 4 PERFORMANCE TUNING AND OPTIMIZATION 297

31 ■ Hardware 201: selecting and sizing database server hardware 299

32 ■ Parameter sniffing: your best friend…except when it isn't 309

33 ■ Investigating the plan cache 320

34 ■ What are you waiting for? An introduction to waits and queues 331

35 ■ You see sets, and I see loops 343

36 ■ Performance-tuning the transaction log for OLTP workloads 353
37 ■ Strategies for unraveling tangled code 362
38 ■ Using PAL to analyze SQL Server performance 374
39 ■ Tuning JDBC for SQL Server 384

PART 5 BUSINESS INTELLIGENCE 395

40 ■ Creating a formal Reporting Services report part library 397
41 ■ Improving report layout and visualization 405
42 ■ Developing sharable managed code expressions in SSRS 411
43 ■ Designing reports with custom MDX queries 424
44 ■ Building a scale-out Reporting Services farm 436
45 ■ Creating SSRS reports from SSAS 448
46 ■ Optimizing SSIS for dimensional data loads 457
47 ■ SSIS configurations management 469
48 ■ Exploring different types of enumerators in the SSIS Foreach Loop container 480
49 ■ Late-arriving dimensions in SSIS 494
50 ■ Why automate tasks with SSIS? 503
51 ■ Extending SSIS using the Script component 515
52 ■ ETL design checklist 526
53 ■ Autogenerating SSAS cubes 538
54 ■ Scripting SSAS databases – AMO and PowerShell, Better Together 548
55 ■ Managing context in MDX 557
56 ■ Using time intelligence functions in PowerPivot 569
57 ■ Easy BI with Silverlight PivotViewer 577
58 ■ Excel as a BI frontend tool 585
59 ■ Real-time BI with StreamInsight 597
60 ■ BI solution development design considerations 608

Part 3

Database development

Edited by Paul Nielsen

The simple SELECT statement is still the most complex and powerful single word in computer science. I've been writing database code professionally since the mid-1980s, and I'm still amazed at SELECT's ability to draw in data from numerous sources (and types of sources) and make that data twist and shout, and present it exactly as needed.

The goal of Transact-SQL (T-SQL) is to provide the right question to SQL Server so the query optimizer can generate a query execution plan that's correct and efficient. The code is only one part of the equation. The physical schema plays a huge role in the efficiency of the code and the query execution plan, and indexing is the bridge between the data and the query. Entering the schema + the query + the indexes into the query optimizer yields the query execution plan.

I might argue that the physical schema is the foundation for database performance; nulltheless, I can't disagree that poor T-SQL code has been the source of many a database pain.

This section is where SQL Server MVPs who are passionate about their T-SQL code share their excitement for T-SQL—their passion, like mine, for making data twist and shout:

- Aaron Bertrand opens the section by sharing his ever-popular list of T-SQL best practices in the form of habits worth forgetting.
- Kevin Boles urges us to remember that the point of T-SQL is the query execution plan, not reusable code.

- John Paul Cook reveals the little-known skill of performing regular expression in Management Studio.
- Sergio Govoni explains the new T-SQL features in Denali.
- My friend and former SQL Server Bible tech editor, Hugo Kornelis shows how to create a new data type.
- Matija Lah digs deep in the CLR and comes up with how to use regular expressions in queries and SSIS packages.
- Peter Larsson returns to our Codd roots with an academic but practical chapter on relational division—a complex topic but one that must be mastered by every serious T-SQL developer.
- Ben Miller, our former MVP lead turned MVP, writes about the options for storing blobs and `FILESTREAM`.
- Luciano Moreira discusses a topic that's a passion of mine—running unit tests on your T-SQL code.
- Mladen Prajdić provides an excellent discussion on Service Broker.
- Denis Reznik shows how to effectively use the HierarchyID data type for coding hierarchies.
- And finally, Allen White demonstrates his ninja scalability skills with Service Broker.

Without a doubt, turning a set of SQL Server MVPs loose with the permission to write about whatever gets them excited is bound to turn up some interesting topics, and indeed it did. Happy reading.

About the editor

Paul Nielsen is the founder of Ministry Weaver, a software startup offering an object database for ministries. He focuses on extreme data modeling, database development, and the occasional XMAL and C#, and he was formerly the author of the SQL Server Bible series and the editor of the first volume of *SQL Server MVP Deep Dives*. Paul offers consulting and seminars on data modeling and schema optimization. He lives in Colorado with his family and he's passionate about marriage, adoption, theology, music, writing, and their next family trip to Maui.

19 T-SQL: bad habits to kick

Aaron Bertrand

As a longtime participant in the SQL Server community, I've seen my share of bad habits. I've even formed a few of my own, all of which I'm on the path to correcting. Whether they come from StackOverflow, the MSDN forums, or even from the `#sqlhelp` hash tag on Twitter, one thing is clear: bad habits can spread like poison ivy. The same is definitely true in Transact-SQL (T-SQL), where samples are borrowed, reused, and learned from on a daily basis.

Many people think `SELECT *` is okay because it's easy, or that following coding standards for this one stored procedure is unimportant because it needs to be deployed in a rush, or that they shouldn't have to type that `ORDER BY` statement because the rows "always" come back in the same order. This chapter is meant to take five of these habits and point out (a) why they're problems, and (b) what you can do to avoid them.

Please note that several of the examples in this chapter will use tables and data from the AdventureWorks2008R2 sample database, available from CodePlex at http://sqlserversamples.codeplex.com/.

SELECT *

For as long as there has been T-SQL code, there has been prevalent use of `SELECT *`. Although typically this is the most straightforward way to determine what a table looks like in ad hoc and troubleshooting scenarios, my characterization as a "bad habit" in this case only refers to its use in production code.

In many cases, not all the columns in the table or view are required by the application. Using `SELECT *` in this case can cause wasteful scans or lookups in order to return all of the columns, when the query (and application) may have been satisfied by a covering index at a much lower resource cost. Not to mention there's the additional overhead of sending all the unneeded columns over the network just to have the application ignore them in the first place—or worse, incorporate them into the code, never to be used.

Consider the following two versions of a query against the Sales.SalesOrderDetail table, and imagine that the application only cares about the columns SalesOrderDetailID, ProductID, and SalesOrderID:

```
SELECT *
    FROM [Sales].[SalesOrderDetail]
    WHERE [ProductID] < 20;

SELECT [SalesOrderDetailID],[ProductID],[SalesOrderID]
    FROM [Sales].[SalesOrderDetail]
    WHERE [ProductID] < 20;
```

When looking at the execution plans for these two queries, you can see that the shorthand version is twice as expensive as the version that explicitly names the required columns. In this case the extra cost is caused by a key lookup, as you can see in figure 1. This figure shows SQL Sentry Plan Explorer, which you can download for free at www.sqlsentry.net/plan-explorer/.

Even when all the columns are required, there are other reasons to name your columns instead of using shorthand. One is that schema changes to the underlying

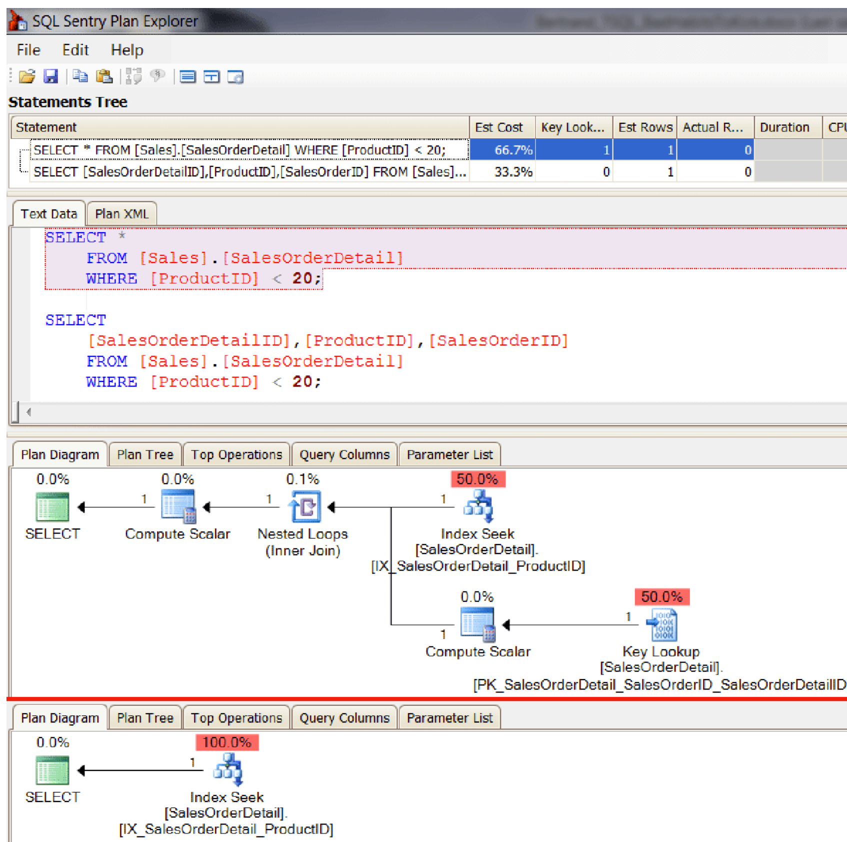

Figure 1 Comparing execution plans: `SELECT *` vs. explicit column list

table(s) can break code relying on ordinal position of columns or performing inserts into other tables without using an explicit column list.

NOTE Please keep in mind that the use of `SELECT *` within an `EXISTS` clause is perfectly fine. The optimizer understands that you won't be returning the columns as data and performs its work without any negative impact on the execution plan.

An example of schema changes causing problems is the case where a view is created using `SELECT *`. If you change the underlying schema, the view can become "confused" because it isn't automatically notified about metadata changes. This means if your view says `SELECT * FROM dbo.foo` and you rename the column x to y, the view will still return a column named x. This issue can get even more convoluted if you then add a different column named x with a different data type.

There's an easy workaround to fix the view in this case: a system stored procedure introduced in SQL Server 2005 named `sys.sp_refreshsqlmodule`. Alternatively, you can prevent `SELECT *` in a view by using `SCHEMABINDING`, which doesn't allow the syntax; this strategy also prevents underlying changes from being made to the table without the user knowingly removing the `SCHEMABINDING` condition from any of the views that reference it.

The excuse I hear the most from people defending the use of `SELECT *` is one borne out of laziness: they complain that they don't want to type out all the column names. Today's tools make this a questionable excuse at best—whether you're using Mladen Prajdic's SSMS Tools Pack, RedGate's SQLPrompt, or the native IntelliSense in SSMS, you'll rarely need to type out the columns manually, thanks to features such as keyword expansion and AutoComplete. And even if you don't use those tools and have disabled IntelliSense, you can drag the entire list of columns by expanding the table in question in Object Explorer, clicking on the Columns node, and dragging it onto the query window, as shown in figure 2.

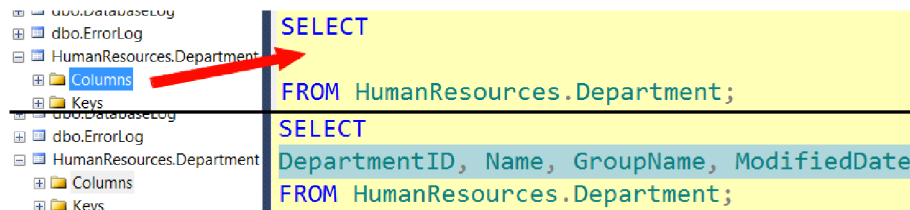

Figure 2 **Dragging the Columns node onto a query window**

Declaring VARCHAR without length

Another habit I see in a lot of code is declaring VARCHAR, NVARCHAR, and other string types without specifying a length. This tendency likely shows up in people with backgrounds in any number of scripting and object-oriented languages, where a string is a string, and it can be as long as you want, as long as it fits in memory. SQL

Server has some inconsistent rules about how long a string can be, depending on how the value is defined. Consider the following examples:

```
DECLARE @x CHAR = 'foo';
SELECT a = @x, b = CAST('foo' AS CHAR), c = CONVERT(CHAR, 'foo');
```

You'd expect in all three cases to see `'foo'` returned, but in fact the first column in the query returns only the letter `'f'`. This is because when a `CHAR`-based variable is declared without a length, the length becomes 1 (and this follows the ANSI standard). On the other hand, when you use `CAST` or `CONVERT` to specify that a string should be a `CHAR`-based type, the length becomes 30.

This behavior can also come into play when you create tables. If you create a table with a column that is `CHAR` based and you don't specify a length, the behavior once again follows the standard, and the column is created with a maximum length of 1:

```
CREATE TABLE dbo.x(y VARCHAR);
GO
```

There are some further oddities in behavior that can make the inconsistent rules tricky to discover; for example, if you try to insert a longer string into the column, you'll get an error message about data truncation:

```
INSERT dbo.x(y) SELECT 'foo';

Msg 8152, Level 16, State 14, Line 2
String or binary data would be truncated.
The statement has been terminated.
```

But if you create a stored procedure that accepts a parameter with the exact same type (`VARCHAR` with no length), there's no error message, and the string is silently truncated and SQL Server quite happily puts the leading character into the column:

```
CREATE PROCEDURE dbo.x_insert
     @y VARCHAR
AS
BEGIN
     SET NOCOUNT ON;
     INSERT dbo.x(y) SELECT @y;
END
GO
EXEC dbo.x_insert @y = 'foo';
SELECT Result = y FROM dbo.x;

Results:
Result
-
f
```

This means that it can take quite a bit of manual effort, or maybe even luck, to discover that the strings you're passing into your stored procedure aren't remaining intact when they get written to the table. Hopefully this kind of thing gets discovered during testing, but having seen this issue make it into production systems, I want to be sure it becomes well known. This problem goes away if you always declare a length for your `CHAR`-based columns.

Not choosing the right data type

There are a lot of cases where you might be tempted to use a specific data type to store certain information, but your first instinct isn't always the best choice. Here are just a few examples I've seen:

- Using `NVARCHAR(MAX)` for zip codes, phone numbers, or URLs
- Using `VARCHAR` for proper names
- Using `DATETIME` when `DATE` or `SMALLDATETIME` will do
- Using `TIME` to store an interval or a duration
- Using `VARCHAR` to store `DATETIME` data

There are many other examples, and I could probably write an entire chapter just on this habit alone, but I want to focus here on the last example—primarily because it's one of the most common that I see out in the wild.

Many people say they want to store `DATETIME` using a `CHAR`-based data type so that they can keep their formatting. They may want to store a date as `"Tuesday, April 5th, 2011"` or as `"04/05/2011"` without having to perform any presentation formatting when they retrieve the data. Another reason I've seen for using `CHAR`-based data types instead of `DATE`/`TIME` types is to store a date without time, or vice versa.

The main problem with doing so is that you lose the ability to validate your data, validation that you get for free when you use a data type such as `DATETIME`, `SMALLDATETIME`, or `DATE`. And in most cases, you also lose the ability to sort the dates and to perform various date-related functions against the data, because SQL Server isn't going to recognize every conceivable format as `DATE`/`TIME` data—especially if the data comes from a form with free text entry, where someone is just as likely to enter "None of your business" as they are to enter a properly formatted and accurate date.

If you're concerned about formatting, please consider that presentation isn't a function of the database. You want to be able to trust your data, and every modern language has easy-to-use methods to present a `DATE`/`TIME` value coming out of SQL Server in whatever predefined or custom format you desire. The next version of SQL Server, codenamed Denali, will have a `FORMAT` function, which will make formatting much easier than currently possible.

If you're using SQL Server 2008 or later, and your need revolves around leaving date or time out of the equation, there's little preventing you from using the new, separate `DATE` and `TIME` types. If you must continue supporting SQL Server 2005 and older versions, you could use a `DATETIME` or `SMALLDATETIME` column with a `CHECK CONSTRAINT` to ensure that the time portion is always midnight, or that the date portion is always some token date (usually January 1, 1900).

If you aren't yet using SQL Server 2008 or later, there's one case that will be hard to support without using this bad habit (and I'll forgive it in this case): you need to store dates that precede 1753, the lower bound on the `DATETIME` data type. Because the SQL Server 2008 `DATE` data type supports dates from 0001-01-01 through 9999-12-31, you should be able to avoid this issue if you target that platform. (And if you're working

on new development that's being targeted at SQL Server 2005, remember that this version is no longer in mainstream support, so there are likely far more concerning limitations coming your way.)

Mishandling date range queries

At my previous job, I reviewed a large number of stored procedures that ran reports against historical data, so I've seen a lot of examples of date range query abuse. The most obvious examples are making a `WHERE` clause "non-SARGable," which basically means that an index on the relevant column can't be used. This is easy to do by applying a function such as `CONVERT()` or `YEAR()` against the relevant column, which ties one of the optimizer's hands behind its back. Examples include stripping the date or time to get all the data for a specific day, or extracting just the year and month to pull the data for a specific month. Here are a couple of examples:

```
SELECT SalesOrderID, ProductID
    FROM Sales.SalesOrderDetail
    WHERE YEAR(ModifiedDate) = 2006;

DECLARE @date SMALLDATETIME = '20060801';

SELECT SalesOrderID, ProductID
    FROM Sales.SalesOrderDetail
    WHERE CONVERT(CHAR(8),ModifiedDate,112) = CONVERT(CHAR(8),@date,112);
```

These methodologies will yield a full scan because SQL Server doesn't know how to translate the filter into a usable seek. These queries can be rewritten as follows, and if there's an index on the ModifiedDate column, it can be used for a much more efficient plan:

```
SELECT SalesOrderID, ProductID
    FROM Sales.SalesOrderDetail
    WHERE ModifiedDate >= '20060101'
    AND ModifiedDate < '20070101';

DECLARE @date SMALLDATETIME = '20060801';

SELECT SalesOrderID, ProductID
    FROM Sales.SalesOrderDetail
    WHERE ModifiedDate >= @date
    AND ModifiedDate < DATEADD(DAY, 1, @date);
```

I also saw a lot of `BETWEEN` queries. The problem with using `BETWEEN` for `DATETIME` or `SMALLDATETIME` data is that the end of the range is quite ambiguous. Imagine you have the following data in a table called dbo.Events:

```
dt
----------------
2011-01-01 14:55
2011-01-01 23:33
2011-01-01 23:59
2011-01-02 00:00
2011-01-02 01:23
```

Now consider a query like this:

```
SELECT COUNT(dt)
    FROM dbo.Events
    WHERE dt BETWEEN '20110101' AND '20110102';
```

If you ask different people, you'll probably get different expectations about whether the count should be 3, 4, or 5. The answer, in this case, will be 4, even though the author of the query probably didn't intend to include any data that occurred on January 2…or perhaps they assumed that all the data was entered without time. So in that case, the query should have included *all* of the data on the second. These cases are much better expressed using the following queries:

```
SELECT COUNT(dt)
    FROM dbo.Events
    WHERE dt >= '20110101'
    AND dt < '20110102';

SELECT COUNT(dt)
    FROM dbo.Events
    WHERE dt >= '20110101'
    AND dt < '20110103';
```

Finally, this can get even more ambiguous if you try to determine the "end" of a day to allow you to continue using `BETWEEN`. This kind of code was commonplace in a former environment:

```
SELECT COUNT(dt)
    FROM dbo.Events
    WHERE dt BETWEEN '20110101' AND '20110101 23:59:59.997';
```

The problems here are numerous. One is that if the data type of the column is `SMALLDATETIME`, the end of the range will round up, and the query will become (without your explicit knowledge):

```
SELECT COUNT(dt)
    FROM dbo.Events
    WHERE dt BETWEEN '20110101' AND '20110102';
```

If the data type of the column is more precise (e.g., `DATETIME2`), then a different problem arises: being 3 milliseconds away from midnight may leave some rows out that should've been counted.

I always use an open-ended date range like those demonstrated in these queries. This way, I know for sure that the optimizer will use an index if it can, and I don't have any doubts about whether I'll include data from the wrong time period.

Making assumptions about ORDER BY

The last bad habit I'm going to talk about here is the reliance on ordering without using an explicit `ORDER BY` clause. You may remember that in SQL Server 2000, you could create a view using `SELECT TOP 100 PERCENT … ORDER BY`, and when you ran a query against the view without an `ORDER BY`, it would return in the expected order.

With optimizer changes in SQL Server 2005, this behavior was no longer observed—and even though the `ORDER BY` was clearly documented as a filter to determine which rows to return, and not necessarily to dictate the order the rows would be returned, many people complained about the behavior change.

The fact in this case is quite simple: if you want a specific order, *always* define an explicit `ORDER BY` on the outer query, even if some inner component of the query or table structure makes you feel like the data should come back in a certain order. If you don't use an `ORDER BY` clause, you're essentially telling SQL Server "Please return this data in whatever order you deem most efficient."

It's a common misconception that a query against a table with a clustered index will always come back in the order of the clustered index. Several of my colleagues have published blog posts proving that this isn't true, so just because you see it in your environment today, don't assume that this is a given. In short, do *not* rely on unguaranteed behavior, even if you "always" observe the same thing. Tomorrow the behavior may be different, because of a statistics change, your plan being expunged from the plan cache, or optimizer behavior changes due to a service pack, hotfix, or trace flag change. These are all things that can happen that you either won't know about or won't immediately correlate with the need to now go and add `ORDER BY` to all of your queries where the ordering behavior has changed.

Does this mean you should add `ORDER BY` to every single query you have? Only if you care about order (though I've found few cases where order truly isn't important, at least when you're returning data to a user or an application). And if you're changing your queries to ensure that your order will be observed tomorrow, make sure to inspect your query plans to determine whether you're performing any unnecessary sorts, because these won't necessarily be associated with the end result. Sort operations are rarely, if ever, going to make your queries run faster.

Summary

Making exceptions to sensible rules, or making assumptions about undocumented behavior, will lead to undesirable results sooner or later. But if you strive to avoid these things, and try to turn bad habits into good habits, you'll have far fewer cases where the exceptions will cause you pain.

About the author

Aaron Bertrand is a senior consultant for SQL Sentry, makers of performance monitoring and event management software for SQL Server, Analysis Services, and Windows. He's been blogging at sqlblog.com since 2006, focusing on manageability, performance, and new features. Aaron has been an MVP since 1997, and he speaks frequently at user group meetings and SQL Saturday events.

SQL SERVER/DATABASE

SQL SERVER MVP DEEP DIVES Volume 2

EDITORS: Kalen Delaney • Louis Davidson • Greg Low • Brad McGehee • Paul Nielsen • Paul Randal • Kimberly Tripp

To become an MVP requires deep knowledge and impressive skill. Together, the 64 MVPs who wrote this book bring about 1,000 years of experience in SQL Server administration, development, training, and design. This incredible book captures their expertise and passion in sixty concise, hand-picked chapters.

SQL Server MVP Deep Dives, Volume 2 picks up where the first volume leaves off, with completely new content on topics ranging from testing and policy management to integration services, reporting, and performance optimization. The chapters fall into five parts: Architecture and Design, Database Administration, Database Development, Performance Tuning and Optimization, and Business Intelligence.

What's Inside

- Discovering servers with PowerShell
- Using regular expressions in SSMS
- Tuning the Transaction Log for OLTP
- Optimizing SSIS for dimensional data
- Real-time BI
- Much more

This unique book is your chance to learn from the best in the business. It offers valuable insights for readers of all levels.

Written by 64 SQL Server MVPs, the chapters were selected and edited by Kalen Delaney and Section Editors Louis Davidson (Architecture and Design), Paul Randal and Kimberly Tripp (Database Administration), Paul Nielsen (Database Development), Brad McGehee (Performance Tuning and Optimization), and Greg Low (Business Intelligence).

Manning Publications and the authors of this book support the children of Operation Smile

For online access to the authors go to manning.com/SQLServerMVPDeepDivesVol2. For a free ebook for owners of this book, see insert.

ISBN 13: 978-1-617290-47-3
ISBN 10: 1-617290-47-5
55999
9 781617 290473

MANNING $59.99 / Can $62.99 [INCLUDING eBOOK]