MANNING

SAMPLE CHAPTER

SQL SERVER MVP DEEP DIVES

EDITED BY

Paul Nielsen • Kalen Delaney • Greg Low • Adam Machanic • Paul S. Randal • Kimberly L. Tripp

MVP CONTRIBUTORS

John Baird • Bob Beauchemin • Itzik Ben-Gan • Glenn Berry • Aaron Bertrand • Phil Brammer • Robert C. Cain • Michael Coles • John Paul Cook • Hilary Cotter • Louis Davidson • Christopher Fairbairn • Rob Farley • Denis Gobo • Bill Graziano • Dan Guzman • Paul Ibison • Tibor Karaszi • Kathi Kellenberger • Don Kiely • Kevin Kline • Hugo Kornelis • Alex Kuznetsov • Matija Lah • Cristian Lefter • Andy Leonard • Greg Linwood • Bruce Loehle-Conger • Brad McGehee • Paul Nielsen • Pawel Potasinski • Matthew Roche • Dejan Sarka • Edwin Sarmiento • Gail Shaw • Linchi Shea • Richard Siddaway • Jasper Smith • Erland Sommarskog • Scott Stauffer • Tom van Stiphout • Gert-Jan Strik • Ron Talmage • William R. Vaughn • Joe Webb • John Welch • Erin Welker • Allen White

WAR child

Author royalties go to support War Child International

SQL Server MVP Deep Dives

Edited by
Paul Nielsen, Kalen Delaney , Greg Low,
Adam Machanic, Paul S. Randal, Kimberly L. Tripp

Chapter 1

brief contents

1 Louis and Paul's 10 key relational database design ideas

Paul Nielsen and Louis Davidson

Even though the database world is more stable than the app dev world, which seems to have a hot new language every other week (our procs written in SQL Server T-SQL 4.21 still run, for the most part), there are still controversies and key ideas that are worth examining. For the 24 Hours of PASS, we presented a session on Ten Big Ideas in Database Design:

1. Denormalization is for wimps
2. Keys are key
3. Generalize, man!
4. Class <> table
5. Data drives design
6. Sets good, cursors bad
7. Properly type data
8. Extensibility through encapsulation
9. Spaghetti is food, not code
10. `NOLOCK` = no consistency

1. Denormalization is for wimps

A common database development phrase is "Normalize 'till it hurts, then denormalize 'till it works." Poppycock! Although denormalization may have yielded better results for OLTP databases using SQL Server 6.5 and earlier, it is only rarely the case in modern times.

For OLTP or operational databases (not reporting databases or BI) a correctly indexed normalized schema will nearly always perform better than a denormalized

one. The extra work required to double insert, double update, or pivot denormalized data when reading will almost always cost more than the perceived benefit of reading from a single denormalized location. Unless you do performance testing and know your application's profile, it is likely that when you denormalize, you're introducing performance problems that would have been easy to avoid by learning better SQL practices. But the root cause of most less-than-normalized databases isn't an effort to improve performance; it's the incompetence of the person designing the database and the programmers writing the queries.

In specific cases it's good to denormalize even in an OLTP database. For example, within an inventory system, it might make sense to precalculate the inventory transactions and store the quantity on hand for parts per location. Read/write ratios should be carefully considered, and you should never assume that denormalization is required without testing it both ways under load—properly normalized and denormalized. (And consider asking an MVP in the newsgroups if there is a better way to do things—helping people is what the MVP program is all about.)

First normal form is critical: first normal form means not duplicating columns, but it also means one fact per attribute with plain, human-readable data, as well as not having unique rows. Violating first normal form is more common than you might think: smart keys such as product codes that embed data within the product code violate first normal form. If it takes combining multiple attributes to deduce a fact, that violates first normal form. Bitmasked attributes that embed data within a binary column (bit 1 means this, bit 2 means that) is an extreme violation of first normal form because anyone who can figure out binary encoding should know better. Remember, good index utilization is very specifically tied to first normal form. Embedded meaning inside a column (for example, the middle three characters of a five-character string), is not so easily indexed, particularly when it is not the first part of the value.

2. Keys are key

It's downright scary how many production databases have no primary and foreign key constraints, and even more use identity integers or GUIDs as the only uniqueness constraints on their tables. I guess the idea is that the front-end application is ensuring only good data is passed to the database.

I wonder if those developers have ever flown internationally. In 2007, I (Paul) made three trips to Russia (and had a great time). At the Delta ticket check-in counter, they check your passport. When boarding the plane, they check your ticket and your passport. But in Moscow, the passport control officers don't just assume that everything must be okay. They scan your passport, do a quick computer check on your identity, and verify your Russian visa. In the UK, everyone entering is photographed. The U.S. takes fingerprints. These are folks who understand the concept of database enforced keys.

There are two true things about every database I've ever worked with that had no keys. First, the IT manager thought the data was good. Second, the database was full of bad data.

There are some database developers who don't use keys. Don't be that guy.

While we're on the subject of primary keys, composite primary keys lead to composite foreign keys, which eventually lead to joining on nine columns and clustered indexes as wide as the Grand Canyon. Although this is technically not incorrect, it is annoying, it causes poor performance, and it's one of the reasons the concept of surrogate keys was introduced.

Surrogate keys (`INT IDENTITY` columns) are small, fast, and popular, but don't forget to add a unique constraint to the candidate key columns or you'll end up with duplicates and bad data. That's just one more guy you don't want to be.

3. Generalize, man!

Some so-called "normalized" databases are overly complicated, which causes some developers to feel that the more normalized a database is, the more tables it will have. This leads them to say weird things like, "I only normalize to third normal form; otherwise there are too many tables."

A typical cause of an overly complicated database is that the data modeler was overly specific in determining what types of things should be grouped together into an entity. This extreme bloats out the number of tables, makes development both expensive and frustrating, and gives normalization a bad name.

On the other extreme, some database designers merge too many types of things into a single huge gelatinous blob. These database designs appear simple and easy to work with but in fact tend to exhibit integrity issues over time.

The technique that many database developers naturally use but don't name is *generalization*—combining similar types of things to make the databases simpler. A well-generalized database is still perfectly normalized—the difference is that the scope of things in an entity has been explicitly designed. This reduces development time, makes the database more flexible, and increases the life of the database.

The art of database design—moving from the extreme to an elegant design—is all about learning how to generalize entities so you have a database that's both normalized (has integrity) and usable (meets requirements but is not overly complex).

4. Class <> table

There's both a technical and cultural impedance mismatch between the object world and the database world. Relational databases don't readily represent inheritance. The .NET-heads treat the database as nothing more than a data dump, and the database geeks don't care to understand classes, inheritance, and objects. Since most shops have a .NET to database developer or data modeler ratio of about 50 to 1, the database voices are drowned out. In addition, application programming languages are like ear candy to managers just dying to get the job done faster, and we just have T-SQL (good old faithful works-like-a-charm T-SQL, but face it, not the most exciting language). The result is that management is losing any respect it had for the database discipline. The database is under attack and losing the war.

Too often the solution is to use an application layer object-relational mapper, such as NHibernate or the Entity Framework. This provides a poor abstraction for the database. The response is all too often to create a table for every class and a row for every object. This fails to properly represent the classes.

The solution is not something that can be bottled as a magic potion. The object layer needs to be designed for the .NET needs, and the database for the database's needs (that part is easy!).

Part of the solution is to model the object world in the database using supertypes, subtypes, and views that join the tables to present all the objects in a class. But some things are tables in a relational database and multivalued properties in the object. Some classes will expand to many tables. What is required to make this work?

Teamwork. Architects from the relational and object-oriented disciplines need to work together and build an intermediate layer that works. Then we can build tools to make that process easier.

5. Data drives design

According to common wisdom, business logic has no business in the database. If the point is to keep messy hard-coded logic out of T-SQL code, I agree, but business logic is most flexible when it's pushed down from the application front end, down past the data access layer, down deeper than the stored procedure layer, down past the database schema, all the way into the data.

The best place for business logic is in the data. Every variable in the business formula should be stored in the database rather than coded in any programming language. The behavior of the application is then determined not by .NET code, or T-SQL case expressions, but by the data. Joining from the current data set to the business-rules data sets can dynamically plug the correct values into the business formula for the appropriate row. An admin application can modify the business-rule data at any time without code changes. This is the freedom and elegance that a data-driven database adds to the application.

6. Sets good, cursors bad

Perhaps the most egregious error in SQL Server development is the T-SQL cursor. Nearly everyone has heard how evil cursors are. Writing a database cursor is like going to the bank and depositing a million dollars, one dollar at a time, using a million tiny deposits. In the set-based solution, you just hand over a million one dollar bills at once. The same work is done (the money is counted and deposited) but the teller will not have to have a separate conversation with you for each piece of paper. The performance difference is obvious, and you are saving 999,999 deposit slips to boot. Typically a set-based solution will perform a magnitude better than an iterative solution and will scale significantly better.

There are times when writing a cursor is the best technique—when the logic depends on the order of the rows, or when the code is iterating though DDL code

generations, for example. But claiming that "it's really hard to figure out the set-based way to handle this problem" is not a legitimate reason to write a cursor. Unfortunately, cursors are sometimes written because the database design is so horrid that set-based code is nearly impossible.

The key point concerning cursors and database design is that a well-normalized database schema will encourage set-based queries, whereas a denormalized (or never normalized) database schema will discourage set-based code. That's one more reason to normalize that database schema.

7. Properly type data

Don't make every string column `VARCHAR(255)`. Don't make every column that holds money values `MONEY`. You wouldn't serve pigs from a teacup, nor would you serve tea from a pig's trough. Use the right-sized data type; this is the first line of defense in ensuring the integrity of your data. It is amazing how many databases use `VARCHAR(255)` when the front-end application only allows 30 characters and report definitions only allow 20.

TIP Obfuscated database schemas are another real pain, and they serve no purpose. J12K98D is not a reasonable table name.

8. Extensibility through encapsulation

Architects recognize the value of separation of concerns, and encapsulation is the heart of service-oriented architecture. Why isn't encapsulation for the database respected?

Of all the possible SQL Server practices, one of the very worst is application-based ad hoc SQL: any SQL statement that is passed directly from the object layer directly referencing SQL Server tables. Why? Because the tightly-coupled database becomes brittle—a slight change to the database breaks hundreds to thousands of objects throughout the application code, reports, and ETL processes. An abstraction layer lets the database developer modify the database and, so long as the database API isn't changed, the outside world is not affected.

Every individual computing component should be wrapped in a protective layer, encapsulated to hide the complexities of what's inside.

9. Spaghetti is food, not code

Spaghetti coding was very much the norm back in the "good old days" (which weren't all that good, if you ask me). We had BASIC or COBOL, and when you wanted to change the next line that you were executing, you said `GOTO 1000`, which took you to line 1000 of the source code. It was cool at the time, but as programming became more complex, this got out of hand, and people were using `GOTO`s like crazy. The term *spaghetti code* referred to how the control jumped all over the place like a strand of spaghetti.

So you are probably thinking, "We don't even use `GOTO`s in our code." That's true, but what we do is often much worse. Sometimes we write messy code that does

unnecessary dynamic SQL calls that cannot be readily interpreted by the support person, and in other cases we create blocks of code like the following that are difficult to support:

```
IF condition
       QUERY
ELSE IF condition
      QUERY
ELSE
     EXECUTE procedure --which in turn has another IF block like this
END
```

This kind of spaghetti coding pattern is bad enough, but it really gets messy when it is difficult to discern where a problem is coming from because it is happening "somewhere" in the code—not in an easily located call. Consider these examples:

- An `UPDATE` fires a trigger that calls a procedure that inserts data in a table that then fires a trigger that calls a procedure that contains complex business logic.
- A stored procedure uses a temp table created in a different stored procedure, which is then modified in another stored procedure.
- A global temporary table is created in a procedure, and then is used in a different connection.
- A global cursor is opened in a procedure and is used by other procedures that may or may not be called or closed.

One of the reasons things like triggers get a bad name is that there is so much overuse of triggers that fire other triggers and procedures, and programmers don't like unexpected results from the "magic" that triggers can do. And can you blame them? Getting predictable results is very important to the process, and a mess of triggers calling procedures calling triggers with cursors (and so on) makes getting correct results difficult and support nearly impossible.

10. NOLOCK = no consistency

Locks are a very important part of SQL Server to understand. They allow multiple users to use the same resources without stomping on each other, and if you don't understand them, your job of tuning the server is going to be a lot harder.

For many people, the seemingly logical way to get around the problem of locks is to ignore them by using one of these methods:

- Using `SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED` to have all following code neither set nor respect locks
- Using locking hints of `NOLOCK` or `READUNCOMMITTED` on tables in queries

But both of these methods open up more complicated problems for your support status. By ignoring locks, you will get rows that are in the middle of being processed, and they may not be in their final resting place. This means that you may be getting data that was never committed to the database, or data that is in an intermediate state.

What is the solution? Consider how long the locks are held and how much is being locked. The default isolation level for SQL Server, `READ COMMITTED`, will generally only lock one row at a time when it fetches rows. Even inside a transaction, you can get different results for the same query depending on the transaction isolation level. (Only the highest transaction isolation level, serializable, will prevent other users from changing or adding rows that your query has viewed; snapshot isolation level will let your results stay constant by saving changes temporarily for your queries.) However, transaction isolation assumes that the server can discern your actions from the structure of your database. For example, consider the following:

```
UPDATE tableName
  SET  column = 'Test'
  WHERE tableNameKey = 1
```

If I asked you how many rows needed to be locked, you would immediately assume one, right? That depends on the indexing of tableNameKey. If there is no index, an update will scan every row, and the lock taken could lock all readers out of the table. Many locking issues fall into this category, where the query processor has to lock many more rows than would be necessary if it knew that only one row would be affected by this query.

In SQL Server 2005, Microsoft added in something called `SNAPSHOT` isolation level. In `SNAPSHOT` isolation level, when you start a transaction, all the data you read will look exactly as it did when you started the transaction. Currently executing processes don't lock you out; they maintain the previous version for anyone who wants to see it. This is a nifty way to implement a system, but it presents some difficulties for modifying data—you really have to consider how the `SNAPSHOT` isolation level will affect your queries and applications. `SNAPSHOT` transaction isolation uses row versioning to store the before image of the data in tempdb, so fully understanding the impact on your hardware is very important.

All of this leads us to a database setting called `READ_COMMITTED_SNAPSHOT` that will change the way the default `READ COMMITTED` isolation behaves. This setting uses `SNAPSHOT` isolation level at a statement level, so the results you get for a query will only include rows that have been committed; but if someone changes the data after you start getting results, you won't see that. This is generally good enough locking, and it is certainly enough if you are using optimistic locking techniques to protect users from overwriting other users' changes.

One thing that is clear is that concurrency is not an easy problem to solve. In this chapter, we are pointing out that ignoring locks is not the same as tuning your database system because you are sacrificing consistency for speed. If your data really does need to be heavily locked, perhaps it's better to let it be.

Summary

This list of ten key database ideas offers a quick taste of the challenges you'll face when designing and developing database applications. It's not always easy, but unlike

many other computer-related challenges, the theory and practices for designing and implementing a relational database have remained relatively stable for many years. The main reason for this is that when Codd developed the initial relational database system, he took into consideration the kind of stuff we talked about in our eighth point: encapsulation.

Whereas SQL has remained reasonably stable, the engines that run the database servers have become more and more powerful. If you follow the rules, normalize your data schema, and write set-based queries, that SQL statement you write on a small dataset on your own machine will be far easier to optimize when you get to the reality of millions of rows on a SQL Server machine in the cloud, where you don't have access to the hardware, only the SQL engine.

About the authors

Paul Nielsen is a SQL Server data architect who has enjoyed playing with data for three decades. He's the author of the *SQL Server Bible series* (Wiley), is convinced that SSMS is the ultimate UI, and dreams in E/R diagrams and T-SQL. Paul is the founder of NordicDB—a software startup company that offers a specialized SaaS CRM solution for non-profits. Paul lives in Colorado Springs with a wife who is far more beautiful than he deserves and his three children (ages 22, 21, and 5). He speaks at most conferences and offers seminars in database design. His website is http://www.SQLServerBible.com.

Louis Davidson has over 15 years of experience as a corporate database developer and architect. Currently he is the Data Architect for the Christian Broadcasting Network. Nearly all of Louis' professional experience has been with Microsoft SQL Server from the early days to the latest version currently in beta. Louis has been the principal author of four editions of a book on database design, including one for SQL Server 2008. Louis' primary areas of interest are database architecture and coding in T-SQL, and he has experience designing many databases and writing thousands of stored procedures and triggers through the years.

SQL SERVER MVP DEEP DIVES

Editors: Paul Nielsen • Kalen Delaney • Greg Low • Adam Machanic • Paul S. Randal • Kimberly L. Tripp
Technical Editor: Rod Colledge

This is no ordinary SQL Server book. In *SQL Server MVP Deep Dives*, the world's leading experts and practitioners offer a masterful collection of techniques and best practices for SQL Server development and administration. 53 MVPs each pick an area of passionate interest to them and then share their insights and practical know-how with you.

SQL Server MVP Deep Dives is organized into five parts: Design and Architecture, Development, Administration, Performance Tuning and Optimization, and Business Intelligence. In each, you'll find concise, brilliantly clear chapters that take on key topics like mobile data strategies, Dynamic Management Views, or query performance.

What's Inside

- Topics important for SQL Server pros
- Accessible to readers of all levels
- New features of SQL Server 2008

Whether you're just getting started with SQL Server or you're an old master looking for new tricks, this book belongs on your bookshelf.

The authors of this book have generously donated 100% of their royalties to support War Child International.

About War Child International

War Child works in conflict areas around the world, advancing the cause of peace by helping hundreds of thousands of children every year. Visit www.warchild.org for more information.

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

ISBN 13: 978-1-935182-04-7
ISBN 10: 1-935182-04-8
55999
9 781935 182047

MANNING $59.99 / Can $74.99 [INCLUDING eBOOK]