

Pure SQL

Three steps for better performing and more scalable database access

The late 1990s and the beginning of the 21st century is the era of in-memory representation of database information. Dozens of toolkits have been made available to provide code-centric abstractions for your relational data. The object-oriented view of data was the highest goal. Databases were nothing more than a cumbersome means for persistence — just dumb data stores. But are these tools — and the associated thought patterns — the best choice for highly scalable and high-performing applications?

Essentially all databases in wide use today have been optimized to work with tabular data and its relations. SQL itself is designed around set-based operations and not for the manipulation of single records. This view is in direct opposition to the object-oriented world, which is based on working with distinct objects or single entities. Even though you can use a diverse set of collections to represent relationships and sets in your object-oriented applications, one question remains: Does it really make sense to keep and modify in-memory copies of your data in high-transaction environments?

Highly Scalable UPDATES and the End of Automated Tools

In fact, it is quite likely that you will reach some limits of any automated tool — DataSets, DataAdapters or custom object/relational map-

ping code — as soon as your application requires highly scalable transactional database access. But don't take my word for it — let me instead work with you on a sample application in a number of iterations.

As the basis for the suggestions in this article, I'll look at a small fragment of an application for maintaining article information, inventory levels and orders. This application has to track an article ID, its name and the available inventory for each article. When an order is placed, your program has to verify the article's inventory level, because your business analysts identified a business rule that states that orders must only be accepted for articles which are in stock. In addition, this available amount has to be decremented if the order is accepted.

Step One: Separate Your Data

In the first iteration, some developers may consider creating a table named "Article" with fields "ArticleID," "Name" and "UnitsInStock." In fact, even sample databases like *Northwind*—which is shipped with every instance of SQL Server—are designed in the same way.

I believe that this is wrong. To create more scalable systems, you should design your database with more than just strict normalization in mind. You should also identify the transaction volume for each piece of data and group them accordingly. In the application above, this means

Ingo Rammer

<http://www.thinktecture.com/staff/ingo>



Ingo Rammer is co-founder of thinktecture, a company providing in-depth consulting, support and training services for software architects and developers.

thinktecture
<http://www.thinktecture.com>

He regularly shares his knowledge at industry conferences and events around the world, including TechEd, WinDev, DevWeek, NDC, VS.NET Connections, and DevDays. Ingo is the award-winning author of the books *Advanced .NET Remoting* and *Advanced .NET Remoting in VB.NET*. He is currently working on two new books.

that you should actually create two tables: "Article," with fields "ArticleID," "Name," and other static information. And a second table "Inventory," with fields "ArticleID" and "UnitsInStock." This allows your application *always* to work with article information without being affected by transaction locks on the "Inventory" table.

After identifying the best table structure, you now have to choose the right database-access strategy for your code. Should you map your data to .NET objects? DataSets? Or just use DataCommands and their Execute* methods? I tend to recommend a very pragmatic solution for most applications: For all pieces of data that are mostly read — and that seldom participate in high-transactional UPDATES — simply choose the *most comfortable* approach. According to your personal preferences and project focus, this could include the use of an object/relational mapper or DataSet.

If you decide to use an object/relational mapper, I'd recommend that you *do not* implement this functionality on your own. There are a number of excellent products available, with costs ranging from *zero* to a few hundred USD or EUR per developer. I know that the temptation is high to build your own O/R mapper, but this is a complex task which does not usually bring you closer to solving the business requirements for your application.

For all kinds of data that are changed with high transaction counts, however, you should consider either manually creating all UPDATE and DELETE statements or using appropriately optimized stored procedures. Please bear in mind that *automatically generated* SQLs or stored procedures usually will not allow you to achieve the highest possible levels of scalability and performance. The reason for this is that the underlying tools have to generate very generic SQL, making application-specific optimizations impossible. After all, only the developer of the application—you—knows about its detailed database access requirements.

A Short Trip to Optimistic Concurrency

One of the core problems of automatically generated database-access methods is that they usually offer a generic, optimistic-concurrency model. The main assumption behind these optimistic

models is that *most of the time* only one user will change a given piece of data. Corrective action will only be taken in the negative case — if a previous change by another user is detected during the update.

This assumption is perfectly valid for most near-static data, such as articles and customers. This optimistic approach, however, will have very negative effects if you are working with data that are changed in high-transaction environments. It can be the cause of excessively long database locks, or even deadlocks.

But what exactly is an UPDATE for optimistic concurrency, and why is it bad for transactions?

When changing data in this way, the first step is to load the current value of a record into some in-memory representation, such as DataSet. After this, your application can modify the data to represent the desired "final" state that should be stored in the database after your transaction. As soon as your application decides to keep the change, it generates an UPDATE statement, in which the WHERE clause contains not only the record's primary key but also a timestamp, or the previous values of the changed fields.

Before showing you some SQL examples, let me say one more thing: I am going to show you literal SQL without the recommended definition of query parameters. I do this solely for illustrative purposes, as doing so makes it easier to talk about these statements. In your actual application, you should *always* use query parameters, stored procedures, or at least check all your input data before passing it to your database server. You need to do this at least to protect yourself against SQL-injection security attacks. The principles I discuss below apply equally to the creation of these prepared queries or stored procedures.

If your *Products* table for article 42 shows a current inventory level of 15 and you would like to store an order for five pieces, SQL similar to the following would be generated: UPDATE Inventory SET UnitsInStock = 10 WHERE ArticleID = 42 and UnitsInStock = 15. The second part of the WHERE clause (*UnitsInStock = 15*) ensures that the UPDATE fails—or returns zero hits—if some other user modifies the amount in the meantime. In this case, your application receives an Exception and has to reload the record, re-apply the

changes and try again with a new UPDATE statement.

This is the first major disadvantage of using automatically generated SQL with data modified in high transaction counts: If a large number of users are running the same application — modifying the same data — these additional re-SELECT/re-UPDATE cycles will happen quite often.

Step Two: Reduce Deadlocks by Defining a Specific UPDATE Strategy!

Let's further suppose that two purchase orders are entered at the same time. The first PO is for two pieces each of articles 42, 43 and 44. The second PO is also for two pieces of each article, but in reverse order: 44, 43, and 42. Both purchase orders are stored at the same time in two transactions.

Processing starts with the first order and the UPDATE of article 42, which is successful. After this — depending on the time delay — the second order will change the inventory level of article 44. Subsequently, article 43 will be changed according to the first PO. As a next step, your application will try to change article 43 again, this time to reflect the changes of the second PO. This UPDATE will not succeed, because the corresponding record is currently locked by the first transaction. Consequently, the processing of the first order will continue with an attempt made to update its last article. This change, however, will also fail, as article 44 is still locked by the second transaction. This is a classic deadlock scenario: The first transaction waits for article 44, the second for article 43. Each record is locked by the other transaction. In most modern databases, this situation is automatically detected; one transaction is designated the deadlock victim and aborted. Your application receives a corresponding *Exception* and has to restart the transaction from the beginning.

The interesting question is whether these deadlocks can be avoided, and if not, whether the likelihood of their occurring can be reduced.

A general way to avoid similar situations is to define and follow an update strategy. In the application above, it would be enough simply to execute the UPDATES sorted by the affected articles' IDs. In that case, you would always UPDATE the articles in the sequence 42, 43, 44 — no matter what sequence users followed when entering the elements of the two purchase orders.

It therefore does not cause a deadlock if the processing of the second PO hits the lock of article 42. The first PO will be completed successfully, because neither article 43 nor article 44 is locked. The lock on article 42 will be removed after completion of the first transaction, allowing the second PO also to be stored successfully without any need for a retry.

This guideline can be slightly generalized to say that resources should always be accessed in the same sequence. This includes not just single records, but also access to database tables in general.

In most cases, you can achieve this kind of *UPDATE strategy* only with an explicit, application-specific database-access code. Automated tools generally will process all rows in the exact sequence in which they have been entered, leading inevitably to a database code that's prone to deadlock.

But still: If you are following an in-memory, optimistic concurrency approach as described above, there is yet another drawback. When the second order is processed, your application has to send additional SELECTs and UPDATEs to compensate for the changed inventory level. The total database interaction for these two purchase orders consists, therefore, of nine SELECTs and nine UPDATEs.

Step Three: Reduce the Number of Database Interactions

The original reason for the first SELECT was the need to read the current inventory level, to verify the stock level (according to the "business rule"), and to decrement the in-memory representation reflecting the new stock level after the transaction. Interestingly enough, this entire operation can be handed over to the database.

If, for example, you had 15 pieces of article 42 in stock and received an order for four pieces, you could (in the worst case) work as follows: You send a SELECT to the database, transform the result into a DataSet using a DataAdapter, change the inventory level in the DataSet to the new value of 11, and finally use a DataAdapter to send a corresponding UPDATE statement to the database. This requires two database round-trips (in the positive case, two more if someone has changed the value in the meantime), and two (or again, four) transformations to/from in-memory XML in the DataSet.

Let's think the unthinkable: How about not using *any* in-memory representation of the inventory level? Instead of using a DataSet, you would hypothetically only call the necessary SQL statement or stored procedure directly via an IDbCommand object. Instead of pushing your data through SELECT->DataAdapter->DataSet->DataAdapter->UPDATE, you would send one simple SQL statement to your database.

Sound too good to be true?

Actually I would suspect that this is the way most people designed their software before Resultsets, Recordsets, DataSets, and similar constructs became available. In fact, you can bring about the same behavior simply by sending this SQL statement to your database: UPDATE Inventory SET UnitsInStock = UnitsInStock - 4 WHERE ArticleID = 42 AND UnitsInStock >= 4. This statement decrements the inventory level (UnitsInStock = UnitsInStock - 4) after verifying the current stock level (UnitsInStock >= 4). It is also highly scalable, as it does not introduce further SELECT/UPDATE cycles, which would prolong the duration of the transaction.

After executing this statement, its return value will contain the number of affected rows. If this number equals 1, you immediately know that the business rule has been satisfied, and that the inventory level has been successfully decremented. You only have to send an additional SELECT statement if you received 0 as the return value, which means that there were not enough units in stock when the original statement was executed. In fact, if you use SQL Server, you could even send a batched statement consisting of the UPDATE and the SELECT to acquire the necessary data in just one round trip. It is important to note that a change in the inventory level initiated by a different user in the meantime will not require additional SELECT/UPDATE statements. Different processing is necessary only if the business rule has not been satisfied.

When processing two orders at the same time — as in the example above — this allows you to reduce the number of database interactions from 18 SQL statements (nine SELECTs plus nine UPDATES) to just six UPDATES.

This leads to shorter transaction times, resulting in the quicker release of database locks. As a consequence, your application will exhibit better performance and scalability. And all this just by performing three easy steps: separating your data according to transaction volume, choos-

ing an UPDATE strategy to eliminate deadlocks, and reducing the number of database round trips with application-specific, optimized SQL.

About Abstractions

When creating software today, we have the advantage of being able to rely on a number of abstractions. All these abstractions hide complexity; without them, software would not be as easily created as it is today. It would simply not be possible or feasible to implement business software based on x86 assembly language.

Every abstraction shields us from the complexity of the underlying environment. What we tend to forget from time to time, however, is that there is usually a number of reasons for the underlying complexity. If you look at databases in particular: as soon as you demand performance and high scalability, you will need to know about SQL optimizations, and indexing and locking strategies. Most of these things are hidden when you use certain abstractions. If you—as an architect—choose the wrong abstraction, a developer might not be able to use these optimizations.

But if this is true, there's one thing which really worries me: If an architect has to select a matching level of abstraction, doesn't this also mean that s/he also has to know about all the arcane details of SQL Server locking? Or, as a client of mine put it after a long discussion of deadlocking and blocking differences for Oracle and SQL Server: "*Do we really have to be database experts?*"

How do you make these decisions in your projects? Who determines which level of abstraction to use? Who has to know about SQL Server details: the architect or the developer? I'm looking forward to your ideas and opinions at ingo.rammer@thinktecture.com.

My company, thinktecture, helps software architects and developers who need to implement .NET and Web services solutions. We provide project-specific training, consulting, coaching, and emergency troubleshooting services to help you create architectures and applications which live up to the demands. If matters are critical, one of us can usually be on site within 48 hours throughout Europe.

About thinktecture



thinktecture is a European company providing in-depth consulting, support and training services for software architects and developers. It has been founded by the world-renowned authors and speakers Christian Weyer and Ingo Rammer. Additional thinktecture associate trainers are Christian Nagel and Ralf Westphal. All of thinktecture's staff are published authors and members of the Microsoft Regional Director program which only consists of 140 members worldwide.

We put quality highly above quantity, and our four technical staff members are known by developers not only for their books, articles, and regular columns about software architecture in multiple languages, but also for their talks, trainings and seminars in Europe, the US, and Africa (Amsterdam, Boston, Brussels, Casablanca, Frankfurt, Istanbul, Los Angeles, Munich, New Orleans, Orlando, Palm Springs, Vienna, Zurich, and many more). Each of thinktecture's experts has close ties to Microsoft product teams in their area of expertise.

Bernhard-Krieg Strasse 4
97845 Neustadt am Main
Germany

Phone: +49 9393 99 31 61
Fax: +49 9393 99 31 62

office@thinktecture.com
http://www.thinktecture.com

Responsible for all aspects of this publication: Ingo Rammer.

Copyright 2004 thinktecture—Ingo Rammer und Christian Weyer GBR. Bernhard-Krieg-Strasse 4. 97845 Neustadt Am Main, Germany. <http://www.thinktecture.com>. All rights reserved. No warranties of any kind. Including, but not limited to: correctness, and applicability, Ingo Rammer also reserves the right to adjust his opinions at any later point in time.

If you would like to receive future issues of the Architecture Briefings, you are invited to subscribe—also free—at

<http://www.thinktecture.com/NL>

Syndication/Translation

Architecture Briefings are released in advance in the following high-quality developer magazines:

Germany



<http://www.dotnet-magazin.de>

Italy



<http://online.infomedia.it/riviste/Vbj/>

We are looking for translations and syndication partners for other languages! Please direct your requests regarding commercial redistribution and translation to office@thinktecture.com