

realtimepublishers.comtm

The Definitive Guidetm To

Scaling Out SQL Server 2005

Don Jones

Chapter 2: Scaling Out vs. Better Efficiency	26
Addressing Database Design Issues.....	26
Logically Partitioning Databases	31
Addressing Bottlenecks Through Application Design	34
Minimize Data Transfer	34
Avoid Triggers and Use Stored Procedures.....	35
Use Multiple Application Tiers	36
Use Microsoft Message Queuing Services and Service Broker	39
Plan for Data Archival	40
Fine-Tuning SQL	43
Tuning Indexes.....	43
Using an Appropriate Fillfactor	43
Smart Indexing.....	43
Always Have a Clustered Index.....	44
Using Composite Indexes	44
Improving T-SQL and Queries	45
Always Use a WHERE Clause	46
Avoid Cursors	47
Miscellaneous T-SQL Tips	48
Summary	48

Copyright Statement

© 2005 Realtimedpublishers.com, Inc. All rights reserved. This site contains materials that have been created, developed, or commissioned by, and published with the permission of, Realtimedpublishers.com, Inc. (the "Materials") and this site and any such Materials are protected by international copyright and trademark laws.

THE MATERIALS ARE PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT. The Materials are subject to change without notice and do not represent a commitment on the part of Realtimedpublishers.com, Inc or its web site sponsors. In no event shall Realtimedpublishers.com, Inc. or its web site sponsors be held liable for technical or editorial errors or omissions contained in the Materials, including without limitation, for any direct, indirect, incidental, special, exemplary or consequential damages whatsoever resulting from the use of any information contained in the Materials.

The Materials (including but not limited to the text, images, audio, and/or video) may not be copied, reproduced, republished, uploaded, posted, transmitted, or distributed in any way, in whole or in part, except that one copy may be downloaded for your personal, non-commercial use on a single computer. In connection with such use, you may not modify or obscure any copyright or other proprietary notice.

The Materials may contain trademarks, services marks and logos that are the property of third parties. You are not permitted to use these trademarks, services marks or logos without prior written consent of such third parties.

Realtimedpublishers.com and the Realtimedpublishers logo are registered in the US Patent & Trademark Office. All other product or service names are the property of their respective owners.

If you have any questions about these terms, or if you would like information about licensing materials from Realtimedpublishers.com, please contact us via e-mail at info@realtimedpublishers.com.


[**Editor's Note:** This eBook was downloaded from Content Central. To download other eBooks on this topic, please visit <http://www.realtimepublishers.com/contentcentral/>.]


Chapter 2: Scaling Out vs. Better Efficiency

Scaling out a database is a common technique for addressing performance issues. However, in many cases, performance can be markedly improved through better efficiency on the existing server. In addition, improving single-server efficiency will lay the path to a more successful scale-out project.

Database designs and applications that are flawed or are simply not well-tuned can have a major negative impact on a single-server application; however, they will lead to considerably worse performance in a scaled-out distributed database. A design or application problem that causes issues on a single server will most likely continue to do so on multiple servers. Thus, a well-tuned database (and application) provides a much better distributed database.

This chapter will introduce you to common database design considerations as well as best practices for application design, index usage, and Transact SQL (T-SQL) fine tuning. These tips will enable you to optimize performance for single-server databases, paving the way to a better distributed database in your scale-out project.

 In addition to the methods for improving the structure of the current database discussed in this chapter, there are low-cost ways to scale out the database hardware that incorporate dual-core processors as well as the benefits of the x64. These methods are explained in the Appendix: 64-Bit and High Performance Computing.

 SQL Server 2005 includes many performance enhancements that earlier versions don't offer. However, most of these improvements are integrated throughout the SQL Server engine, making it difficult to point to any one particular T-SQL language element or database programming technique and say that it will give you a performance boost. Many existing applications may run faster simply by upgrading to SQL Server 2005. However, that doesn't mean there is no room for improvement, especially in efficiency. In fact, the tips in this chapter apply equally to SQL Server 2005 and SQL Server 2000 (except where I've specifically noted otherwise by calling out a specific version of the product).

Addressing Database Design Issues

To begin analyzing the efficiency of your database designs, examine databases for major design issues, particularly over-normalization. Many database administrators, designers, data modelers, and developers make an effort to achieve a high degree of normalization—often normalizing to fifth form, if possible. Over-normalization can negatively impact database performance in single-server applications and in a distributed database. Even normalizing to the third form—which is what most designers aim for—can sometimes result in an over-normalized database that isn't as efficient as it could be.

For example, suppose you have two tables in your database and each relies on a lookup table. The two main tables, Table 1 and Table 2, each have a foreign key constraint on one column that forms a relationship with a table named Lookup Table. As the following tables show (the examples provide only the relevant columns), Table 1 contains the addresses for properties that are for sale, and Table 2 contains the addresses for real estate transactions that are in progress. To ensure that addresses are accurate and consistent, Lookup Table provides pieces of the address including street types (for example, Road, Street, Avenue, Drive). Including this information in a lookup table prevents users from inconsistently entering data such as Rd for Road and Dr for Drive.

Table 1

AdrID	StName	StTyp
1	Main	2
2	Elm	2
3	Forest	3

Lookup Table

TypID	StTypName
1	Road
2	Street
3	Avenue

Table 2

AdrID	StName	StTyp
8	Sahara	1
9	Major	1
10	Pierce	3

The problem with this design is that Table 1 and Table 2 do not contain the street type name; instead, they contain the primary key value pointing to the appropriate row in Lookup Table. Thus, querying a complete address from the database requires a multi-table join, which is inherently less efficient than simply querying a single table. This common database design, which is employed by many applications, is inefficient. If this design were used in a real estate application, for example, address-based queries would be very common, and having to join three tables to make this very common query wouldn't be the most efficient way to build the database.

When you distribute this database across servers in a scale-out project, you will run into additional problems. For example, suppose you put Table 1 and Table 2 on different servers; you must then decide on which server to place Lookup Table (see Figure 2.1).

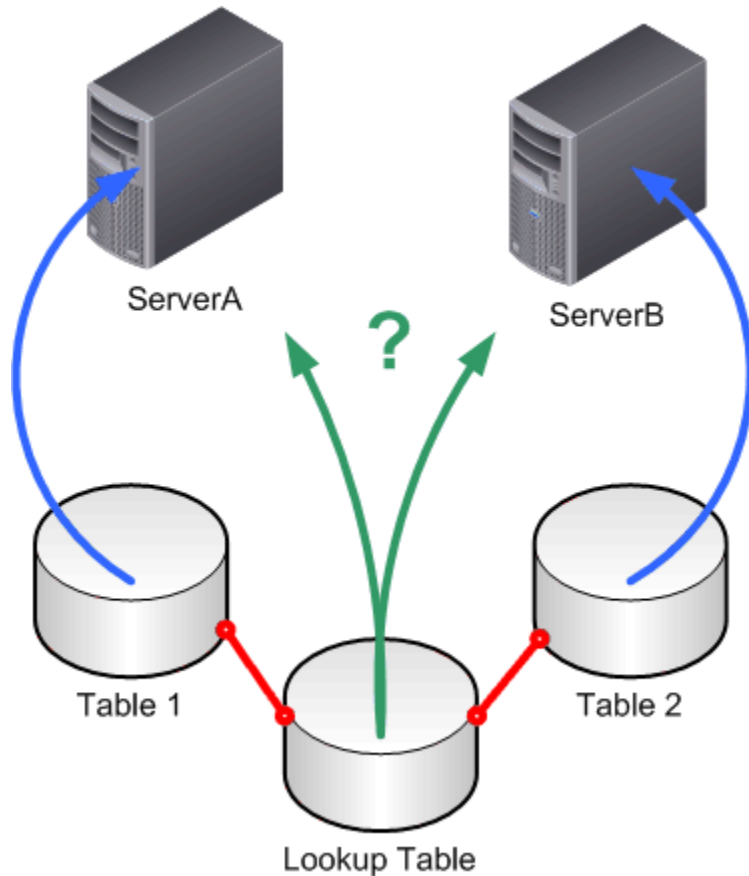


Figure 2.1: Using lookup tables can restrict your scale-out options.

Ideally, as both servers rely on Lookup Table, the table should be on the same physical server as the tables that use it, which is impossible in this case. To work around this problem as well as improve performance, simply *denormalize* the tables, as the following examples illustrate.

Table 1

AdrID	StName	StTyp
1	Main	Street
2	Elm	Street
3	Forest	Avenue

Lookup Table

TypID	StTypName
1	Road
2	Street
3	Avenue

Table 2

AdrID	StName	StTyp
8	Sahara	Road
9	Major	Road
10	Pierce	Avenue

Lookup Table is still used to create user input choices. Perhaps the address-entry UI populates a drop-down list box based on the rows in Lookup Table. However, when rows are saved, the actual *data*, rather than the primary key, from Lookup Table is saved. There is no SQL Server constraint on the StTyp column in Table 1 or Table 2; the UI (and perhaps middle-tier business rules) ensures that only data from Lookup Table makes it into these columns. The result is that the link between Lookup Table and Table 1 and Table 2 is now broken, making it easy to distribute Table 1 and Table 2 across two servers, as Figure 2.2 illustrates.

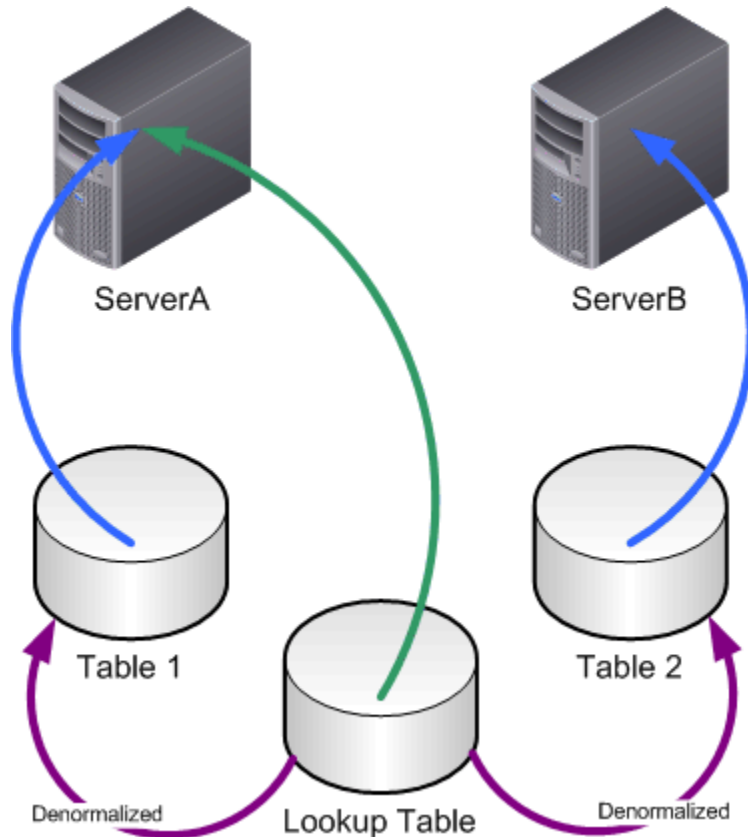


Figure 2.2: Distributing a database is easier without foreign key constraints.

When a user needs to enter a new address into Table 2 (or edit an existing one), the user’s client application will query Table 2 from ServerB and Lookup Table from ServerA. There is no need for SQL Server to maintain cross-server foreign key relationships which, while possible, can create performance obstacles. This example illustrates denormalization: Some of the data (“Street,” “Road,” and so forth) is duplicated rather than being maintained in a separate table linked through a foreign key relationship. Although this configuration breaks the rules of data normalization, it provides for much better performance.



You must always keep in mind why the normalization rules exist. The overall goal is to reduce data redundancy—primarily to avoid multiple-row updates whenever possible. In this real estate example, it’s unlikely that the word “Road” is going to universally change to something else, so it’s more permissible—especially given the performance gains—to redundantly store that value as a part of the address entity rather than making it its own entity.

Denormalization can be a powerful tool—particularly when you’re planning to distribute a database across multiple servers. Maintain foreign key relationships only when any of the following is true:

- The lookup table’s contents change on a regular basis
- The lookup table contains more than 50 or so rows of data
- The lookup table contains more than one data column (in addition to the primary key column)

Unless one of these factors is true, consider denormalizing the data and using the lookup table to populate a UI rather than enforcing a foreign key constraint (in other words, using a client- or middle-tier to enforce the business rules rather than a data-tier constraint). Denormalization will make distributing the data easier.



Normalization is a useful technique; however, this method is generally used to reduce data redundancy and improve data integrity at the cost of performance.

If you have many detached lookup tables after denormalizing, you can create a dedicated SQL Server system to host only those tables, further distributing the data workload of your application. Figure 2.3 shows an example of a scaled-out database with a dedicated “lookup table server.”

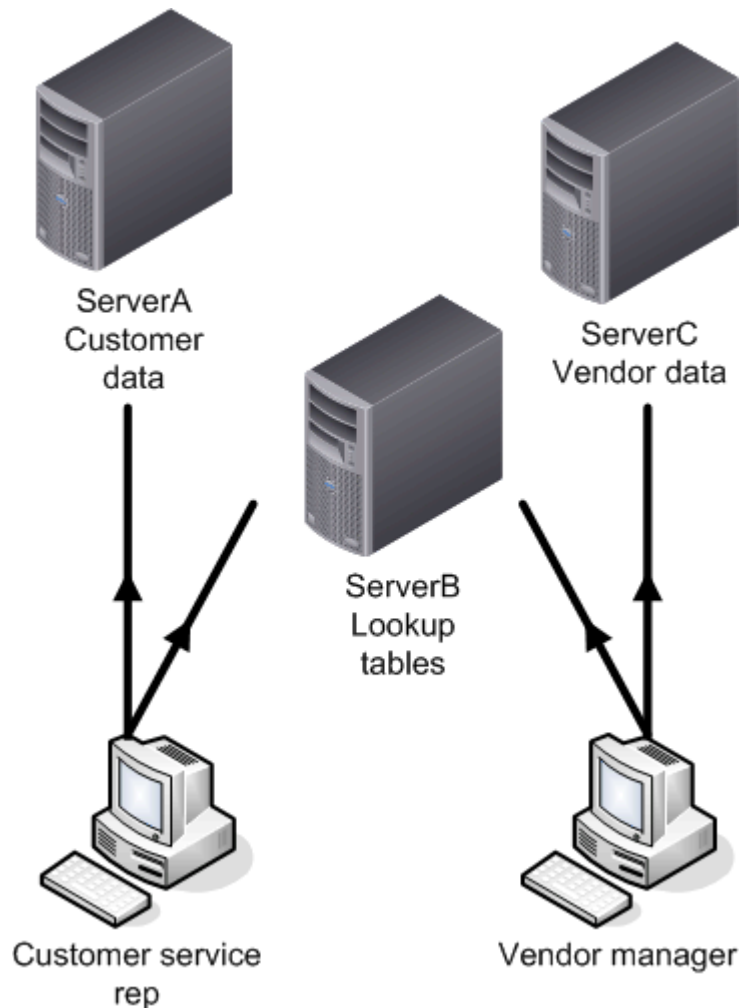


Figure 2.3: Creating a dedicated lookup table can allow you to scale out further.

Client- or middle-tier applications would query ServerB for acceptable values for various input fields; ServerA and ServerC store the actual business data, including the values looked up from ServerB. Because querying lookup tables in order to populate UI elements (such as drop-down list boxes) is a common task, moving this data to a dedicated server helps to spread the application's workload out across more servers.

Logically Partitioning Databases

In addition to denormalizing, determine whether your database design lends itself to *partitioning*. Chapter 1 described two basic ways to distribute a database. The first approach distributes an entire database across multiple servers, making that data available in multiple locations and spreading the workload of the database application. Each copy of the database uses an identical schema and simply contains different rows than the other copies. The other approach distributes the database across multiple servers, placing a portion of the database on each server so that each server has different tables.

For the first approach, you need to have some means of horizontally partitioning your tables. All tables using a unique primary key—such as an IDENTITY column—must have a unique range of values assigned to each server. These unique values permit each server to create new records with the assurance that the primary key values won't conflict when the new rows are distributed to the other servers. For software development purposes, another helpful practice is to have a dedicated column that simply indicates which server owns each row. Figure 2.4 shows a sample table that includes a dedicated column for this purpose.

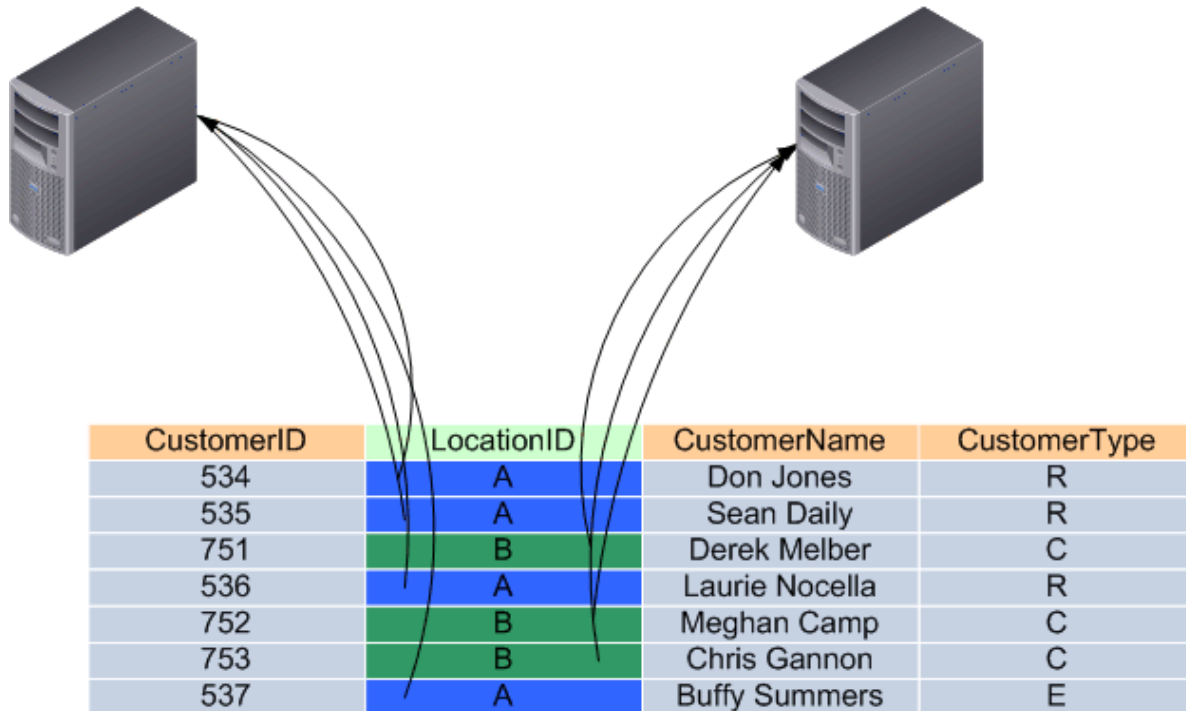


Figure 2.4: Using a column to indicate which server owns each row.

Note that the CustomerID column—the primary key for this table—has different identity ranges assigned to each server as well.

In the other approach, you need to identify logical divisions in your database tables. Ideally, find tables that are closely related to one another by job task or functional use. In other words, group tables that are accessed by the same users who are performing a given job task. Consider the database illustrated in Figure 2.5.

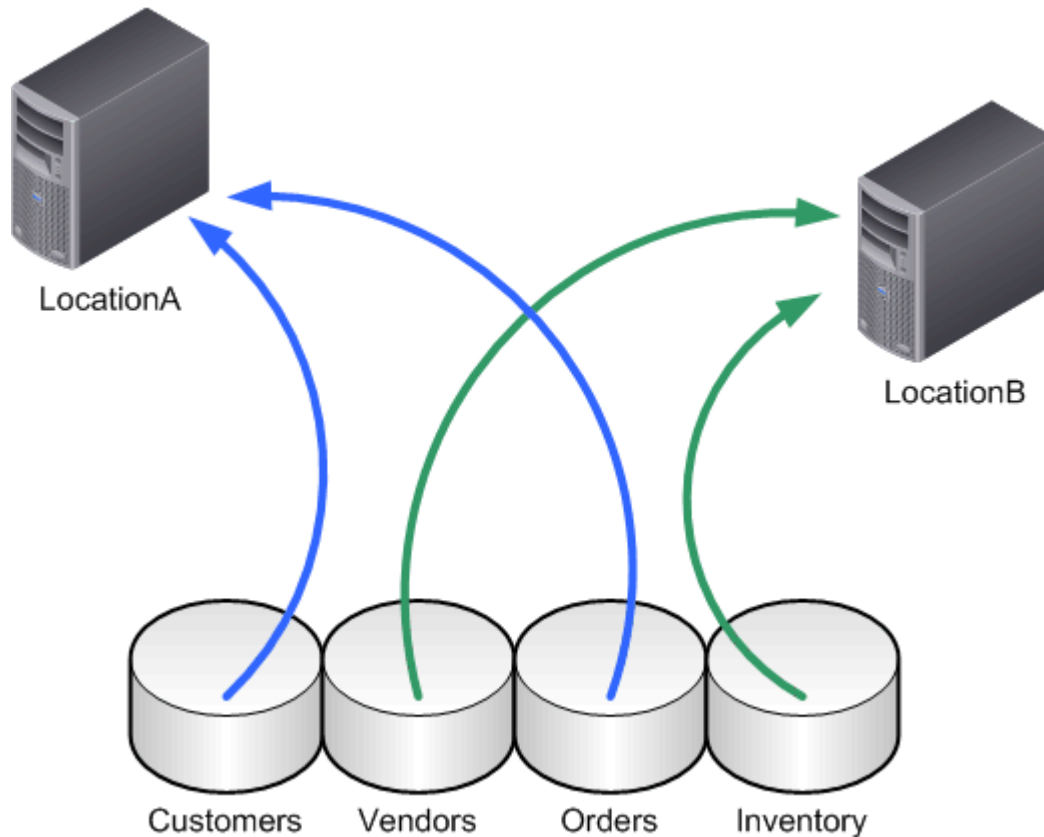



Figure 2.5: Sample database distribution.

It's likely that customer service representatives will access both customer and order data, so it makes sense to distribute those tables to one server. Vendors and inventory data are primarily accessed at the same time, so this data can exist on a separate server. Customer service representatives might still need access to inventory information, and techniques such as views are an easy way to provide them with this information.

If your database doesn't already provide logical division in its tables, work to create one. Carefully examine the tasks performed by your users for possible areas of division for tables.

 Breaking a database into neat parts is rarely as straightforward as simply picking tables. Sometimes you must make completely arbitrary decisions to put a particular table on a particular server. The goal is to logically group the tables so that tables often accessed together are on the same physical server. Doing so will improve performance; SQL Server offers many tools to enable a distributed database even when there aren't clear divisions in your tables.


You can also combine these two scale-out techniques. For example, you might have one server with vendor and inventory information, and four servers with customer and order information. The customer and order tables are horizontally partitioned, allowing, for example, different call centers in different countries to maintain customer records while having access to records from across the entire company.

 I'll describe some of these scale-out decisions in more detail in the next chapter.

Addressing Bottlenecks Through Application Design


Application design—how your applications use SQL Server—can have an incredible impact on performance. Some application design problems can be mitigated through especially clever use of indexes and other SQL Server performance techniques. Other application design problems will continue to be a problem until you redesign the application.

Although application design can cause serious performance problems (or benefits, in the case of a good design) in a single-server database, it can have a much more marked impact in a multi-server, distributed database. By following the rules of good database application design, you can not only fine-tune performance in single-server databases but also help eliminate any performance problems for a distributed database spread across multiple servers.

 Chapter 10 will focus on application design issues in much greater detail, with more specific focus on SQL Server 2005-compatible techniques for improving performance. The tips provided are useful general practices that apply to any database application.

Minimize Data Transfer

Client applications should be designed to query only the data they need from SQL Server. However, client applications that query one row at a time result in unnecessary SQL Server connections and queries that can impede performance. Ideally, client applications query an entire page of data, or a set of pages, at once. For example, if users frequently need to look at product information, and each product represents one screen of data in the client application, design the application to query a dozen products at once. As users move beyond the queried set of products, the application can query a few more products. This technique balances between querying too much data at once and not querying enough.

 The number of rows you query depends entirely on the type of application you're writing. For example, if users frequently query a single product and then spend several minutes updating its information, then simply querying one product at a time from SQL Server is probably the right choice. If users are examining a list of a dozen products, querying the entire list makes sense. If users are paging through a list of a thousand products, examining perhaps 20 at a time, then querying each page of 20 products probably strikes the right balance. The point is to query just what the user will need right at that time or in the next couple of moments.

On the same note, make sure that client applications are designed to minimize database locks and to keep database transactions as short as possible to help minimize locks. Applications should be designed to be able to handle a failed query, deadlock, or other situation gracefully.

These guidelines are *especially* true in a database that will be scaled out across multiple servers. For example, if your scale-out project involves distributed views, locking a row in a view can lock rows in tables across multiple servers. This situation requires all the usual workload of maintaining row locks plus the additional workload required to maintain those locks on multiple servers; an excellent example of how a poor practice in a single-server environment can become a nightmare in a distributed database.

SQL Server cannot determine whether a query is poorly written or will cause a major server problem. SQL Server will dutifully accept and execute every query, so the application's designer must make sure that those queries will run properly and as quickly as possible.

☞ Avoid giving users ad-hoc query capabilities. These queries will often be the worst-running ones on your system because they are not optimized, are not backed by specific indexes, and are not running from a stored procedure. Instead, provide your users with some means of requesting new reports or queries, and allow trained developers and database administrators (DBAs) to work together to implement those queries in the most efficient way possible.

Avoid Triggers and Use Stored Procedures

Triggers are database objects that can serve a useful purpose—they can be used to intercept data and ensure that it is clean, cascade referential integrity changes throughout a hierarchy of table relationships, and so forth. However, SQL Server doesn't optimize triggers efficiently. For example, if you write a trigger that runs whenever a row is deleted, and the trigger then deletes a thousand additional rows, the trigger requires a long time to execute the deletions.

Triggers represent a way to centralize business logic in the data tier, but aren't necessarily the best way to do so. The theory is that triggers are in place to automatically react to any database change, even ones made by ad-hoc queries. If you eliminate inefficient ad-hoc queries from your environment, you don't need triggers.

A better practice is to make database changes by way of stored procedures, which are more efficient and centralize critical operations into the application's data tier. In addition, SQL Server retains the execution plan of stored procedures for future use. If you use stored procedures as your "single point of entry" into SQL Server, triggers are not necessary—you can put the appropriate code into the original stored procedure.

SQL Server makes it quite easy to set up stored procedures as a single point of entry. Simply make sure that the same database user (typically, *dbo*, the built-in *database owner* user that is present in all SQL Server databases) owns the stored procedure objects and all objects (such as tables) the stored procedures reference. Then you can grant users permission to execute the stored procedures and *remove* permission to actually work with the underlying tables and other objects. Any user attempting to submit an ad-hoc query will be denied, forcing the user to employ the stored procedures and enforcing any business logic you've programmed into those stored procedures.

✎ In SQL Server 7.0 and SQL Server 2000, stored procedures could be written only in SQL Server's native T-SQL language. SQL Server 2005, however, embeds the .NET Framework's Common Language Runtime (CLR) within the SQL Server engine. The practical upshot of this inclusion is that you can write stored procedures in any .NET language, such as VB.NET or C#. Visual Studio 2005 provides an integrated development environment that makes creating and debugging this .NET-based stored procedure easier.

The additional power and flexibility offered by .NET provide even more reason to use stored procedures for absolutely every query that SQL Server executes. Stored procedures will become an even better single point of entry into SQL Server, eliminating the need for triggers.

Use Multiple Application Tiers

Large applications need more than just a client application that executes stored procedures. That sort of two-tier design isn't very scalable because it places a lot of workload on the least-scalable component of the application—SQL Server itself. Using a middle tier in your application design allows you to take some of the load off of SQL Server. Clients no longer connect to SQL Server itself; they request data through middle-tier components, and those components access SQL Server directly. The components should, ideally, run on a separate tier of servers. In effect, the middle-tier components become clients of SQL Server, and regular clients become clients of the middle tier.

The middle tier is often referred to as the *business tier* because it is where business logic is implemented. Rather than writing stored procedures to validate data or maintain referential integrity, you write that code into the components running on the business tier. Your business logic can thus be fairly centralized but won't be a burden to SQL Server. Figure 2.6 shows a sample multi-tier application.

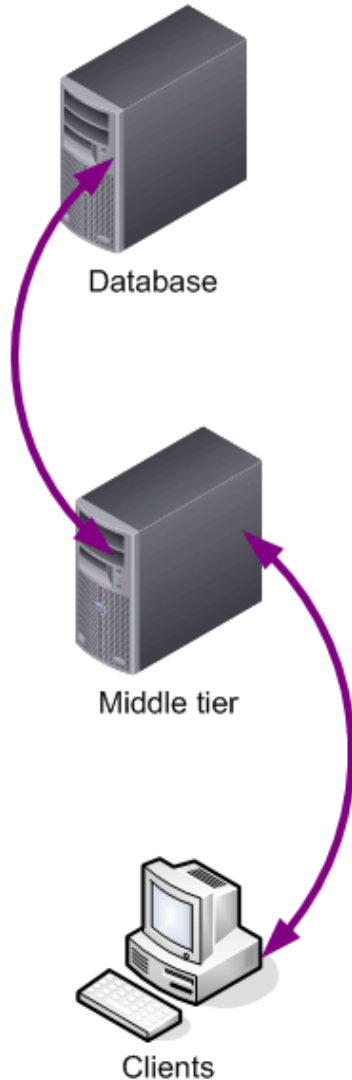


Figure 2.6: A sample multi-tier application design.

The middle tier should still access SQL Server exclusively through stored procedures, but these stored procedures can now be simplified because they don't need to incorporate as much business logic. Thus, they will execute more quickly, allowing SQL Server to support more clients. As your middle-tier servers become overloaded, you simply add more, meeting the client demand. A typical way in which middle-tier servers help offload work from SQL Server is by performing data validation. That way, all queries executed on SQL Server contain valid, acceptable data, and SQL Server simply had to put it in the right place (rather than having stored procedures or other objects validate the data against your business rules). SQL Server is no longer in the business of analyzing data to determine whether it's acceptable. Of course, client applications can perform data validation as well, but middle-tier servers help to better centralize business logic to make changes in that logic easier to implement in the future.

☞ Working with XML? Consider a business tier. SQL Server 2000 and its various Web-based feature releases, as well as SQL Server 2005, support several native XML features that can be very useful. For example, if you're receiving XML-formatted data from a business partner, you can have SQL Server translate, or shred, the XML into a relational data format and use it to perform table updates.

Unfortunately, doing so can be very inefficient (particularly on SQL Server 2000). If you will be working with a great deal of XML-based data, build a middle tier to do so. The middle tier can perform the hard work of shredding XML data and can submit normal T-SQL queries to SQL Server. Execution will be more efficient and you'll have a more scalable middle tier to handle any growth in data traffic.

If you don't *need* to parse (or "shred") the XML, SQL Server 2005 provides a new native XML data type, making it easier to store XML directly within the database. This stored XML data can then be queried directly (using the XQuery syntax, for example), allowing you to work with an XML column as if it were a sort of complex, hierarchical sub-table.

The overall goal is to try to identify intensive processes—such as data validation or XML parsing—and move those to dedicated servers within the middle tier of your overall application. After all, the less work SQL Server *has* to do, the more work it *can* do; because SQL Server is ultimately the only tier capable of querying or updating the database. Moving work away from SQL Server will help maximize its ability to do database work efficiently.

Using multiple tiers can be especially effective in distributed databases. The client application is the most widely distributed piece of software in the application, so the most efficient practice is to avoid making changes to the client application that will require the distribution of an update. By forcing the client to talk only to a middle tier, you can make many changes to the database tier—such as distributing the database across multiple servers or redistributing the database to fine-tune performance—without changing the client application. Figure 2.7 illustrates this flexibility.

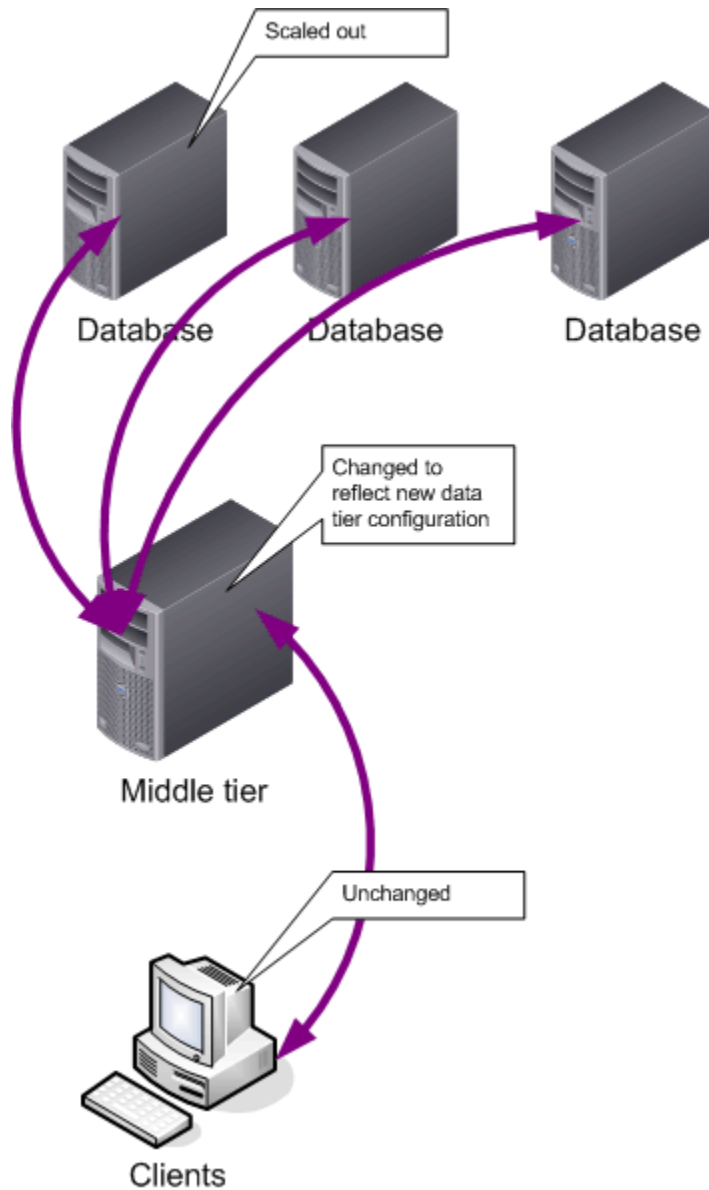


Figure 2.7: Multi-tier applications allow for data tier redesigns without client application changes.

In this configuration, changes to the data tier—such as adding more servers—don't affect the client tier. Instead, the smaller middle tier is updated to understand the new back-end structure. This technique makes it easier to change the data tier to meet changing business demands, without a time-consuming deployment of a new client application.

Use Microsoft Message Queuing Services and Service Broker

Microsoft Message Queuing (MSMQ) services can be a great way to handle long-running queries. Rather than allowing clients (or even middle-tier components) to submit long-running queries directly to SQL Server, clients submit their queries to a queue. A component running on SQL Server pulls these requests one at a time and executes them. When a query finishes, SQL Server places the results on a queue for the requestor. The requestor can check back periodically (or receive notification from MSMQ) to retrieve the results. Figure 2.8 illustrates the process, which is often referred to as *asynchronous* data access.

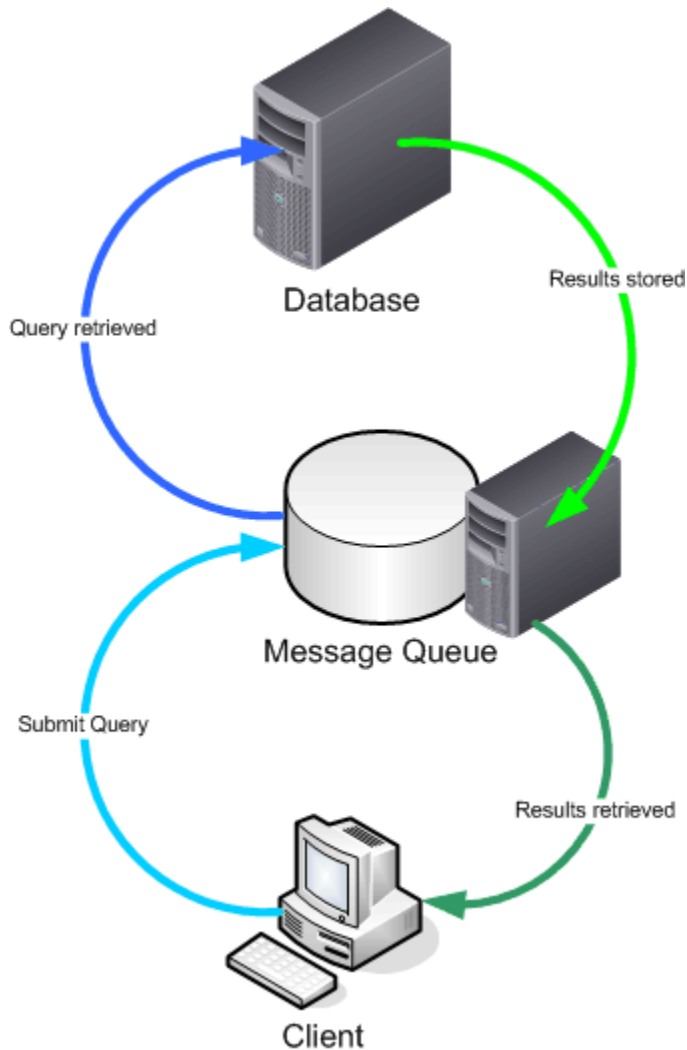



Figure 2.8: MSMQ allows long-running queries (and their results) to be queued.

This technique allows for strict control over long-running queries, and enables requestors to continue working on other projects while they wait for their query results to become available. It isn't necessary for the requestor to be available when the query completes; MSMQ will store the results until the requestor is ready to retrieve them.

SQL Server's Data Transformation Services (DTS) include an MSMQ task that allows DTS to place query results and other information onto an MSMQ message queue. MSMQ is also accessible from COM-based languages, such as Visual Basic, and from .NET Framework applications.

Using MSMQ is an especially effective scale-out technique. For example, if your company routinely prepares large reports based on your database, you might create a standalone server that has a read-only copy of the data. Report queries can be submitted to that server via MSMQ, and the results later retrieved by clients. Use DTS to regularly update the reporting server's copy of the database, or use replication techniques such as snapshots, log shipping, mirroring, and so forth. By offloading the reporting workload to a completely different server, you can retain additional capacity on your online transaction processing (OLTP) servers, improving productivity.


Although SQL Server 2005 can still utilize MSMQ, it also provides a more advanced set of services called Service Broker. Service Broker is a built-in set of functionality that provides message queuing within SQL Server 2005. Service Broker uses XML formatting for messages, and makes it especially straightforward to pass messages between SQL Server 2005 computers. Service Broker can be particularly helpful in scale-out scenarios because it helps to facilitate complex communications between multiple SQL Server 2005 computers.

 Service Broker—specifically, its ability to support scale-out scenarios—will be explored in more detail throughout this book.

Plan for Data Archival

When you initially design your application, plan for a way to move old data out of the application. Typically, databases contain date-sensitive data. So your database might need to contain 3 years' worth of data, after which you can move the oldest year into an archive. Your users might still have occasion to use that archived data, but not on a daily basis.

An effective strategy is to horizontally partition your tables. Doing so provides a second copy of your database (perhaps on a different server) that contains archived data. Client applications will need to be written to query against this old data, if necessary, and you'll need to write stored procedures or DTS packages to move data into the archive every year.

 You can create views that combine the current and archived databases into a single set of virtual tables. Client applications can be written to query the current tables most of the time and to query the views when users need to access archived data. It's the easiest way to implement a distributed-archive architecture.

Why bother with archiving? You archive primarily for performance reasons. The larger your databases, the larger your indexes, and the less efficient your queries will run. If you can minimize database size while meeting your business needs, SQL Server will have a better chance at maintaining a high level of performance over the long haul.

Make sure that your users are aware of the consequences of querying archived data. For example, your client application might present a message that warns *Querying archived data will result in a longer-running query. Your results might take several minutes to retrieve. Are you sure you want to continue?* If you're running queries synchronously, make sure that you give users a way to cancel their queries if the queries require more time than users anticipated. For queries that will take several minutes to complete, consider running the queries asynchronously, perhaps using the MSMQ method described earlier.

☞ Don't forget to update statistics! After you've made a major change, such as removing archived data, be sure to update the statistics on your tables (or ensure that the automatic update database option is enabled) so that SQL Server's query optimizer is aware that your database has shrunk.

Archiving is another great way to scale out a SQL Server application. Some applications spend a considerable portion of their workload querying old data rather than processing new transactions. By keeping an archive database on one or more separate servers, you can break off that workload and preserve transaction-processing capacity on your primary servers. If your application design includes a middle tier, only the middle tier needs to be aware of the distribution of the current and archived data; it can contact the appropriate servers on behalf of clients as required. Figure 2.9 illustrates how an independent archival server can be a part of your scale-out design.

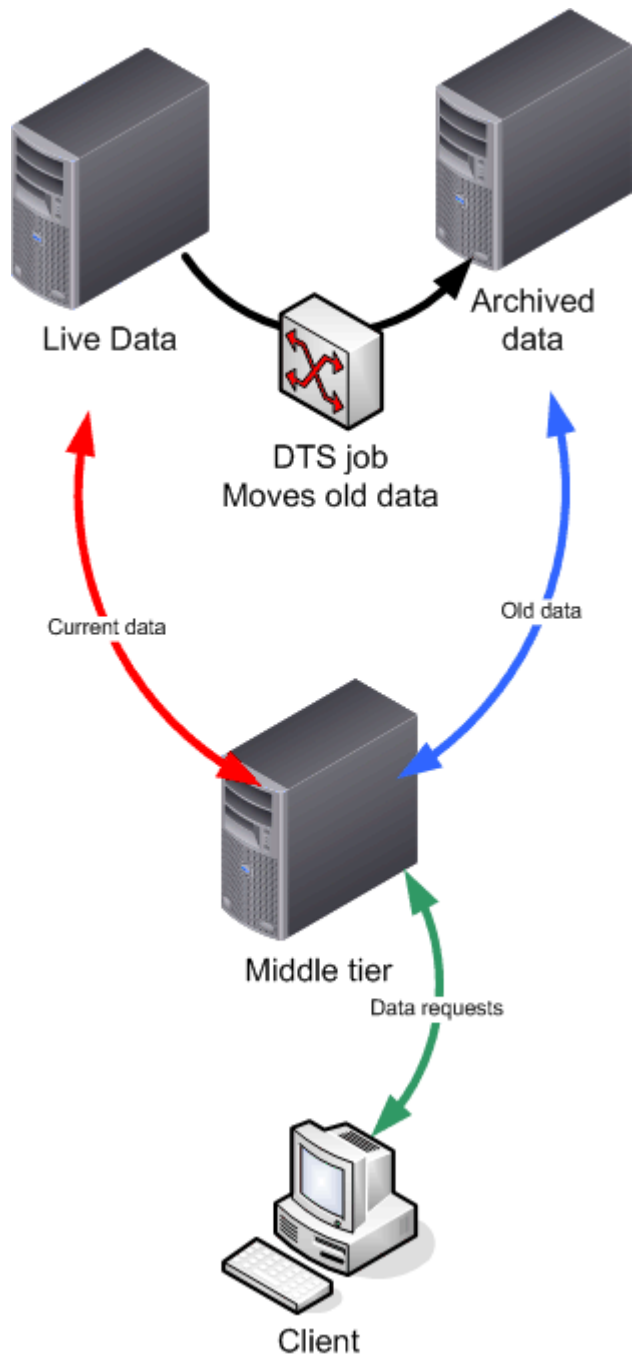


Figure 2.9: Archived data can remain accessible in a dedicated SQL Server computer.

Fine-Tuning SQL

Tuning indexes and improving the efficiency of T-SQL queries will help improve SQL Server's performance. These improvements will help a single server support more work, and they are crucial to scaling out across multiple database servers. Inefficient indexes and inefficient queries can have a serious negative impact on SQL Server's distributed database capabilities, so fine-tuning a database on a single server will help improve performance when the database is spread across multiple servers.

Tuning Indexes

More often than not, indexes are the key to database performance. Thus, you should expect to spend a lot of time fussing with indexes to get them just right. Also, learn to keep track of which indexes you have in each table. You will need to ensure that the index files are on the same physical server as their tables when you begin to divide your database across multiple servers.

Using an Appropriate Fillfactor

When you create a new index, you specify a fillfactor. You do the same when you rebuild an index. SQL Server stores indexes in 8KB pages; the fillfactor specifies how full each 8KB page is when the index is created or rebuilt. (There is a difference between clustered and non-clustered indexes—rebuilding a clustered index rearranges all the data, not just the index.)

A low fillfactor leaves plenty of room for growth but means that SQL Server has to read more pages in order to access the index. A high fillfactor allows SQL Server to read the index as quickly as possible, but as rows are added, the index will be more likely to encounter split pages as the old pages fill up. Split pages are inefficient for SQL Server to read and will require the rebuilding of indexes more frequently.

To determine the optimal fillfactor, understand how your data will grow. Specify the highest appropriate fillfactor that allows enough room for the database to grow between index rebuilds.

Smart Indexing

Many new DBAs throw indexes on every column in a table, hoping that one or two will be useful. Although indexes can make querying a database faster, they slow changes to the database. The more write-heavy a table is, the more careful you need to be when you add your indexes.

Use SQL Server's Index Tuning Wizard (in SQL Server 2000; in SQL Server 2005 it's part of the new Database Engine Tuning Advisor) to get the right indexes on your tables to handle your workload. Used in conjunction with SQL Profiler and a representative query workload, the Index Tuning Wizard is your best first weapon in the battle to properly index your tables.

Indexing isn't a one time event, though. As your database grows, you'll need to reevaluate your indexing strategy. Indexes will need to be periodically rebuilt to ensure best performance. Changes to client applications, database design, or even your server's hardware will change your indexing strategy.

Learn to practice what I call *smart indexing*. Constantly review your indexes for appropriateness. Experiment, when possible, with different index configurations. One way to safely experiment with indexes is to create a testing server that is as close as possible in its configuration to your production server. Use SQL Profiler to capture a day's worth of traffic from your production server, then replay that traffic against your test server. You can change index configurations and replay the day's workload as often as necessary, monitoring performance all the while. When you find the index configuration that works best, you can implement it on your production server and check its performance for improvements.

Always Have a Clustered Index

Remember that a clustered index controls the physical order of the rows in a table, meaning you can have only one clustered index. There is rarely a reason not to have a clustered index. In fact, if you create a nonclustered index and don't already have a clustered index, SQL Server creates a "phantom" clustered index because nonclustered indexes always point to clustered index keys. You might as well create your own clustered index, ensuring that it will be of some use to the queries that you run against the table. If nothing else, create a clustered index on the table's identity column or some other unique column.

The best column for a clustered index is a column with unique values that is used in a number of different queries. For most queries, especially those that return multiple rows, a clustered index is faster than a nonclustered index. Because you only get one clustered index, try to use it where it will have the best impact on performance.

Don't forget that SQL Server 2000 offers a great GUI, in the form of Enterprise Manager (and SQL Server 2005 offers the improved SQL Server Management Studio), for managing indexes. You don't need to deal with complex stored procedures and T-SQL commands to get the right combination of indexes built on your tables. Simply open a table in design mode, right-click the workspace, and select Indexes from the pop-up menu. You'll see the dialog box that lets you manage your indexes, including creating a clustered index.

Using Composite Indexes

SQL Server indexes work best when they have a high degree of uniqueness. You can, of course, create indexes that group several columns together. For example, neither a first name nor last name column will usually be very unique in a customer table. However, the combination of first name and last name will be much more unique. These indexes are referred to as *composite* indexes.

To create composite indexes, you can use Enterprise Manager: Simply use the drop-down list boxes to specify the columns that will belong to the index. Always specify the most unique column first, when possible, because doing so will help SQL Server locate matching rows more quickly during a query.

There are some unique best practices for composite indexes that you should keep in mind (in addition, you should be aware of a SQL Server composite index bug; see the sidebar “The Composite Index Bug” for more information):

- Keep indexes as narrow as possible. In other words, use the absolute minimum number of columns necessary to get the effect you want. The larger the composite index, the harder SQL Server will work to keep it updated and to use it in queries.
- The first column you specify should be as unique as possible, and ideally should be the one used by most queries’ WHERE clauses.
- Composite indexes that are also covering indexes are always useful. These indexes are built from more than one column, and all the columns necessary to satisfy a query are included in the index, which is why the index is said to *cover* the query.
- Avoid using composite indexes as a table’s clustered index. Clustered indexes don’t do as well when they’re based on multiple columns. Clustered indexes physically order the table’s data rows and work best when they’re based on a single column. If you don’t have a single useful column, consider creating an identity column and using that as the basis for the clustered index.

The Composite Index Bug

A fairly well-known SQL Server bug relates to how the query optimizer uses composite indexes in large queries. The bug exists in SQL Server 7.0 and SQL Server 2000; SQL Server 2005’s query optimizer corrects the bug.

When you issue a query that includes a WHERE clause with multiple OR operators, and some of the WHERE clauses rely on a composite index, the query optimizer might do a table scan instead of using the index. The bug occurs only when the query is executed from an ODBC application or from a stored procedure.

Microsoft has documented the bug and provides suggested workarounds in the article “BUG: Optimizer Uses Scan with Multiple OR Clauses on Composite Index” at <http://support.microsoft.com/default.aspx?scid=KB;en-us;q223423>. Workarounds include using index hints to take the choice away from the optimizer and force it to use the index.

How can you tell if this bug is affecting you? Pay close attention to your production query execution plans, which you can view in SQL Profiler. You might also try running an affected query on both SQL Server 2000 and SQL Server 2005 to see how each handles the query.

Improving T-SQL and Queries

What hidden secrets lie within the T-SQL language that can help you improve performance? Quite a few, actually. As you’re writing queries, keep these tips in mind for better performance. As with the other tips in this chapter, improved performance that will be a benefit on a single-server database will carry over to a distributed database that involves multiple servers.

Always Use a WHERE Clause

The advice to always use a WHERE clause might seem basic, but many queries are written to return all rows; the output is then filtered using a client-side trick—all of which is an inefficient process, because it makes SQL Server return data that it doesn't need to return. Thus, always be as specific in your WHERE clauses as possible so that SQL Server is returning as little data as possible. If you're not using a WHERE clause, SQL Server will simply perform a table scan. Thus, indexes are most useful when you've got a WHERE clause to limit rows.

Also, avoid WHERE clauses that aren't *sargeable*. Sargeable is DBA slang for queries that contain a constant value. A WHERE clause such as `WHERE CustomerID < 10` is sargeable because it contains a constant value to compare with the output. The query optimizer can take that WHERE clause and apply an index to make the query perform as quickly as possible. Non-constants in a WHERE clause include

- The `<>`, `!<`, `!>`, `!<` operators
- The IS NULL comparison
- The NOT, NOT EXISTS, NOT IN, NOT LIKE, and LIKE operators. Actually, the LIKE operator is sargeable when you're not using a wildcard as the first character (for example, LIKE 'A%' is sargeable, but LIKE '%A' isn't).
- Any function that includes a column, such as `SUM(OrderTotal)`
- Expressions with the same column on both sides of the operator, such as `CustomerTotal = CustomerTotal + 5`

If you've been careful to specify the columns you want in your SELECT statement, and those columns exist in a covering index, the optimizer will use the covering index rather than performing a slower table scan. Remember, a covering index is one that contains all the columns specified in the SELECT statement.

Consider rewriting queries that aren't sargeable into ones that are. For example:

```
WHERE SUBSTRING(CustomerName,1,2) = 'do'
```

will return all customers whose names begin with "do." The query will require a table scan because the functions aren't sargeable. You could rewrite this query as

```
WHERE CustomerName LIKE 'do%'
```

which is a sargeable query, allowing the query optimizer to use an index on CustomerName (if one exists) to instantly narrow the query to everything that starts with "do."

Avoid Cursors

Cursors are detrimental from a performance perspective. Consider the code sample that Listing 2.1 shows, which is adapted from a sample on <http://www.sql-server-performance.com>.

```

DECLARE @LineTotal money
DECLARE @InvoiceTotal money
SET @LineTotal = 0
SET @InvoiceTotal = 0

DECLARE Line_Item_Cursor CURSOR FOR
SELECT UnitPrice*Quantity
FROM [order details]
WHERE orderid = 10248

OPEN Line_Item_Cursor
FETCH NEXT FROM Line_Item_Cursor INTO @LineTotal
WHILE @@FETCH_STATUS = 0

BEGIN
SET @InvoiceTotal = @InvoiceTotal + @LineTotal
FETCH NEXT FROM Line_Item_Cursor INTO @LineTotal
END

CLOSE Line_Item_Cursor
DEALLOCATE Line_Item_Cursor
SELECT @InvoiceTotal InvoiceTotal

```

Listing 2.1: Sample code that uses a cursor.

This code locates an invoice (10248), adds up all the items on that invoice, and presents a total for the invoice. The cursor is used to step through each line item on the invoice and add its price into the @LineTotal variable. Listing 2.2 shows an easier way that doesn't involve a cursor.

```

DECLARE @InvoiceTotal money
SELECT @InvoiceTotal = sum(UnitPrice*Quantity)
FROM [order details]
WHERE orderid = 10248
SELECT @InvoiceTotal InvoiceTotal

```

Listing 2.2: The sample code modified so that it doesn't involve a cursor.

The new code uses SQL Server's aggregate functions to sum up the same information in fewer lines of code and without using a slower-performing cursor. These aggregate functions can be a big timesaver and return the same results faster than complex cursor operations.

Miscellaneous T-SQL Tips

There are a few miscellaneous tips that don't fit under one category heading. The following tips are additional items to do and to avoid when using T-SQL:

- The **DISTINCT** clause is used to return a rowset that contains no duplicates. Before using the clause, however, ask yourself whether duplicates would be such a bad thing. The **DISTINCT** clause takes a considerable amount of extra time to run, in SQL Server terms, and if you can live without it, you'll increase performance.
- Limit the columns you query to just the ones you actually need. Never type **SELECT ***, even if you want all columns returned. Instead, specifically list each column that you want, and only specify the columns you really need. You'll reduce the amount of data that SQL Server must transmit. Additionally, you'll give SQL Server a better chance to use covering indexes if you limit the information you're querying.
- An effective way to limit the amount of rows a query returns is to use the **TOP** keyword along with **SELECT**. **TOP** allows you to specify a maximum number of rows. SQL Server still executes any **WHERE** clause you specify but will stop processing once it has found the specified number of matching rows, saving processing time. You can **SELECT TOP 100** to return just a hundred rows or **SELECT TOP 10 PERCENT** to select a percentage of the actual result set. Alternatively, you can use the **SET ROWCOUNT** command to limit the number of rows returned for all queries executed by the current connection.

Paging is a function that's often intensive, and prior to SQL Server 2005 there weren't many ways to make it more efficient. SQL Server 2005 provides great new functionality for data paging (when you need to, for example, retrieve 20 rows of data at a time: Rows 1-20, then rows 21-40, and so forth). This new functionality hinges on the new **ROW_NUMBER()** function, which exposes row numbers for each row in a table and allows you to query based on those row numbers, as demonstrated in this sample query:

```
SELECT x.Column FROM (SELECT TOP 30010 A.Column, ROW_NUMBER()
OVER(ORDER BY A.Column) AS TheCount FROM #BigTable A ORDER BY
A.Column) x WHERE x.TheCount BETWEEN 30000 AND 30010
```

This query returns rows 30,000 through 30,010 of the table with a very low query cost, something that was impossible prior to SQL Server 2005.

Summary

These best practices and tips combined with unlimited hours of fine-tuning and tweaking still won't create a single server that has as much raw power as multiple servers. Thus, the reason to consider scaling out—single-server efficiency only gets you so far. However, performance benefits will result in a scale-out environment from fine-tuning the design and performance of databases in a single-server environment. In other words, maximize your efficiency on a single server and you'll reap performance benefits in a scale-out scenario. Databases need to be efficiently designed, queries fine-tuned, indexes put in order, and application architecture cleaned up before beginning the scale-out process. Otherwise, inefficiencies that exist on one server will be multiplied when the database is distributed—effectively sabotaging a scale-out project.

Content Central

[Content Central](#) is your complete source for IT learning. Whether you need the most current information for managing your Windows enterprise, implementing security measures on your network, learning about new development tools for Windows and Linux, or deploying new enterprise software solutions, [Content Central](#) offers the latest instruction on the topics that are most important to the IT professional. Browse our extensive collection of eBooks and video guides and start building your own personal IT library today!

Download Additional eBooks!

If you found this eBook to be informative, then please visit Content Central and download other eBooks on this topic. If you are not already a registered user of Content Central, please take a moment to register in order to gain free access to other great IT eBooks and video guides. Please visit: <http://www.realtimepublishers.com/contentcentral/>.