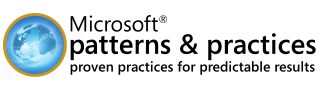
Prescriptive Architecture

**Chapter 14 — Improving SQL Server Performance**

|  |
| --- |
| **Retired Content** |
| This content is outdated and is no longer being maintained. It is provided as a courtesy for individuals who are still using these technologies. This page may contain URLs that were valid when originally published, but now link to sites or pages that no longer exist. |

[](http://www.microsoft.com/practices)

**Improving .NET Application Performance and Scalability**

J.D. Meier, Srinath Vasireddy, Ashish Babbar, Sharon Bjeletich and Alex Mackman  
Microsoft Corporation

May 2004

**Related Links**

[Home Page](http://msdn.microsoft.com/en-us/library/ff649152.aspx) for Improving .NET Application Performance and Scalability

[Chapter 12, Improving ADO.NET Performance](http://msdn.microsoft.com/en-us/library/ff647768.aspx)

[Checklist: SQL Server Performance](http://msdn.microsoft.com/en-us/library/ff647681.aspx)

[Send feedback](mailto:Scale@microsoft.com) to Scale@microsoft.com

[patterns & practices Library](http://www.microsoft.com/practices)

**Summary:** This chapter provides proven development and configuration techniques to help improve the performance and scalability of SQL Server. Topics covered include designing efficient schemas, optimizing queries, tuning indexes, writing efficient stored procedures, analyzing and understanding execution plans, and much more.

**Contents**

[Objectives](http://msdn.microsoft.com/en-us/library/ff647793.aspx#scalenetchapt14 _topic1)  
[Overview](http://msdn.microsoft.com/en-us/library/ff647793.aspx#scalenetchapt14 _topic2)  
[How to Use This Chapter](http://msdn.microsoft.com/en-us/library/ff647793.aspx#scalenetchapt14 _topic3)  
[SQL: Scale Up vs. Scale Out](http://msdn.microsoft.com/en-us/library/ff647793.aspx#scalenetchapt14 _topic4)  
[Performance and Scalability Issues](http://msdn.microsoft.com/en-us/library/ff647793.aspx#scalenetchapt14 _topic5)  
[Schema](http://msdn.microsoft.com/en-us/library/ff647793.aspx#scalenetchapt14 _topic6)  
[Queries](http://msdn.microsoft.com/en-us/library/ff647793.aspx#scalenetchapt14 _topic7)  
[Indexes](http://msdn.microsoft.com/en-us/library/ff647793.aspx#scalenetchapt14 _topic8)  
[Transactions](http://msdn.microsoft.com/en-us/library/ff647793.aspx#scalenetchapt14 _topic9)   
[Stored Procedures](http://msdn.microsoft.com/en-us/library/ff647793.aspx#scalenetchapt14 _topic10)  
[Execution Plans](http://msdn.microsoft.com/en-us/library/ff647793.aspx#scalenetchapt14 _topic11)  
[Execution Plan Recompiles](http://msdn.microsoft.com/en-us/library/ff647793.aspx#scalenetchapt14 _topic12)  
[SQL XML](http://msdn.microsoft.com/en-us/library/ff647793.aspx#scalenetchapt14 _topic13)  
[Tuning](http://msdn.microsoft.com/en-us/library/ff647793.aspx#scalenetchapt14 _topic14)  
[Testing](http://msdn.microsoft.com/en-us/library/ff647793.aspx#scalenetchapt14 _topic15)  
[Monitoring](http://msdn.microsoft.com/en-us/library/ff647793.aspx#scalenetchapt14 _topic16)  
[Deployment Considerations](http://msdn.microsoft.com/en-us/library/ff647793.aspx#scalenetchapt14 _topic17)  
[Summary](http://msdn.microsoft.com/en-us/library/ff647793.aspx#scalenetchapt14 _topic18)  
[Additional Resources](http://msdn.microsoft.com/en-us/library/ff647793.aspx#scalenetchapt14 _topic19)

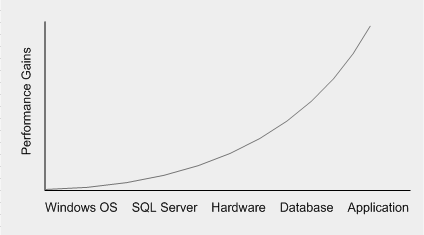
**Objectives**

* Design efficient schemas.
* Optimize queries.
* Fine-tune indexes.
* Perform efficient transactions.
* Build efficient stored procedures.
* Analyze and understand execution plans.
* Identify and eliminate execution plan recompiles.
* Avoid scalability pitfalls when you use SQL XML.
* Tune Microsoft SQL Server.
* Test and monitor your data access performance.
* Consider how deployment decisions impact performance and scalability.

**Overview**

There are many issues that affect SQL Server performance and scalability. This chapter discusses these issues, starting with data design and ending with deployment. The chapter emphasizes the techniques you can use to obtain the biggest gains in performance and scalability. You usually can obtain the biggest gains by creating efficient data access code for the application and by using correct development techniques. You usually do not obtain such big gains in performance and scalability by changing SQL Server configuration settings.

Figure 14.1 shows where your performance design and tuning efforts are best focused.



**Figure 14.1: Focus for performance design and tuning efforts**

The graph is meant to reflect the typical situation and to underscore the point that you obtain the best performance and scalability gains in application development. Indexing is considered part of the application development effort, although it is also part of administration.

**How to Use This Chapter**

Use this chapter to apply proven strategies and best practices for designing and writing high-performance interop code. To get the most out of this chapter, do the following:

* **Jump to topics or read from beginning to end**. The main headings in this chapter help you locate the topics that interest you. Alternatively, you can read the chapter from beginning to end to gain a thorough appreciation of performance and scalability design issues.
* **Measure your application performance**. Read the "ADO.NET / Data Access" and ".NET Framework Technologies" sections of Chapter 15, "[Measuring .NET Application Performance](http://msdn.microsoft.com/en-us/library/ff647791.aspx)" to learn about the key metrics that you can use to measure application performance. It is important for you to measure application performance so that you can accurately identify and resolve performance issues.
* **Test your application performance**. Read Chapter 16, "[Testing .NET Application Performance](http://msdn.microsoft.com/en-us/library/ff647788.aspx)" to learn how to apply performance testing to your application. It is important for you to apply a coherent testing process and to analyze the results.
* **Tune your application performance**. Read the "ADO.NET Tuning" and "SQL Server Tuning" sections of Chapter 17, "[Tuning .NET Application Performance](http://msdn.microsoft.com/en-us/library/ff647813.aspx)" to learn how to resolve performance issues that you identify through the use of tuning metrics.
* **Use the accompanying checklist in the "Checklists" section of this guide**. Use the "[Checklist: SQL Server Performance](http://msdn.microsoft.com/en-us/library/ff647681.aspx)" checklist to quickly view and evaluate the guidelines presented in this chapter.

**SQL: Scale Up vs. Scale Out**

Scaling up refers to moving an application to a larger class of hardware that uses more powerful processors, more memory, and quicker disk drives. Scaling out refers to an implementation of federated servers, where consumer-class computers are added and where data is then partitioned or replicated across them. You can scale out by using functional partitioning. For example, you might scale out by putting your Customer Relationship Management (CRM) functionality on one server and your Enterprise Resource Planning (ERP) functionality on another server. Or, you could scale out by using data partitioning. For example, you might scale out by creating updatable partitioned views across databases.

Do not consider scaling up or scaling out until you are certain that you are getting the best performance that you can through application optimization. Consider the following scenarios when it comes to addressing two common scalability bottlenecks:

* **Processor and memory-related bottlenecks**. Scaling up is usually a good approach if your bottlenecks are processor related or memory related. By upgrading to a faster processor or by adding more processors, you maximize use of your existing hardware resources. You can resolve memory bottlenecks by adding additional memory or by upgrading existing memory. The /3GB switch in the Boot.ini file and Address Windowing Extensions (AWE) also help maximize memory use.

For more information about AWE, search for "AWE SQL Server" (without quotation marks) on the Microsoft support site at [http://support.microsoft.com](http://support.microsoft.com/).

* **Disk I/O**–**related bottlenecks**. Scaling up can also help to resolve disk I/O–related bottlenecks. This form of bottleneck usually occurs in online transaction processing (OLTP) applications where an application performs random disk reads and writes, in contrast to sequential access in online analytical processing (OLAP) applications. For OLTP applications, the I/O load can be spread by adding disk drives. Adding memory also helps reduce I/O load. Because the I/O load is reduced, the size of the SQL Server buffer cache increases. As a result, page faults are reduced.

Consider the following guidelines before you decide to scale up or scale out:

* **Optimize the application before scaling up or scaling out**.
* **Address historical and reporting data**.
* **Scale up for most applications**.
* **Scale out when scaling up does not suffice or is cost-prohibitive**.

**Optimize the Application Before Scaling Up or Scaling Out**

Before you decide to scale up or to scale out, you need to be sure that it is required. Scaling out works best when you plan and design for it in the early stages of your application development life cycle. Changing your application after it is in production so that you can scale up or scale out is expensive. In addition, certain initial design decisions that you make may prevent you from scaling out later.

You can resolve most performance and scalability issues by performing the optimization steps that are outlined in the rest of this chapter. These optimizations help reduce the impact of bottlenecks that are caused by specific design or implementation techniques. These optimizations also help ensure that existing resources are fully utilized. For example, with optimization, you can resolve bottlenecks that are caused by inefficient locking, unprepared SQL statements, poor indexes that lead to increased CPU utilization, and memory or disk I/O utilization.

In practice, you need to simulate your data usage and growth early in the application life cycle by using a realistic workload. Simulating your data usage and growth helps you identify scalability issues sooner rather than later so that you can modify your design and approach to mitigate those issues.

**Address Historical and Reporting Data**

Historical data may become very large over time and may cause long-running queries. Consider partitioning historical data by some range, and implement a way to limit older data. Either move the older data offline, or implement a separate data warehouse that contains the older data.

Reporting needs may also be very resource intensive. You may consider upgrading your database server or scaling out to meet your reporting needs. By implementing a data warehouse or a reporting server, you may be able to provide faster response times and less resource contention. Additionally, a data warehouse or a reporting server is easier to manage than multiple servers in a federated server scheme.

**More Information**

For more information about how to partition historical data, see "Partition Tables Vertically and Horizontally" later in this chapter.

**Scale Up for Most Applications**

If you still have high levels of system resource use after you tune your application and after you address historical and reporting data issues, consider replacing slow hardware components with new, faster components. Or, consider adding more hardware to your existing server.

High levels of system resource use include high CPU utilization, high memory use, and excessive disk I/O. The new components you might add include additional processors or memory. Alternatively, consider replacing your existing server with a new, more powerful server.

Ensure that any configuration changes take full advantage of the new hardware. For example, you may need to use the /3GB switch in the Boot.ini file. This is an easy next step for both migration and maintenance reasons. You should perform tests to help determine the new server capacity that you require.

**More Information**

For more information about testing, see Chapter 16, "[Testing .NET Application Performance](http://msdn.microsoft.com/en-us/library/ff647788.aspx)"

**Scale Out When Scaling Up Does Not Suffice or Is Cost-Prohibitive**

If your application still does not perform well enough, you can consider scaling out or implementing a federated servers option. These approaches usually require certain tables to be horizontally partitioned so that they reside on separate servers. The approaches may also require some replication between servers of the main domain tables that also have to be available on a partition.

Disaster recovery and failover are also more complex for federated servers. You have to determine if the benefit of this added complexity outweighs the cost advantage of being able to use consumer-class computers for federated servers.

**More Information**

For general information about SQL Server scalability, see "SQL Server Scalability FAQ" at <http://www.microsoft.com/sql/prodinfo/previousversions/scalabilityfaq.mspx>.

For more information about federated servers, see "Federated SQL Server 2000 Servers" at <http://msdn.microsoft.com/en-us/library/aa174502(SQL.80).aspx>.

For general information about application scalability, see "[How To: Scale .NET Applications](http://msdn.microsoft.com/en-us/library/ff650667.aspx)" in the "How To" section of this guide.

**Performance and Scalability Issues**

The main issues relating to SQL Server that affect the performance and the scalability of your application are summarized in this section. Later sections in this chapter provide strategies and technical implementation details to help you prevent or resolve each of the following issues:

* **Not knowing the performance and scalability characteristics of your system**. If performance and scalability of a system are important to you, the biggest mistake that you can make is to not know the actual performance and scalability characteristics of important queries, and the effect the different queries have on each other in a multiuser system. You achieve performance and scalability when you limit resource use and handle contention for those resources. Contention is caused by locking and by physical contention. Resource use includes CPU utilization, network I/O, disk I/O, and memory use.
* **Retrieving too much data**. A common mistake is to retrieve more data than you actually require. Retrieving too much data leads to increased network traffic, and increased server and client resource use. This can include both columns and rows.
* **Misuse of transactions**. Long-running transactions, transactions that depend on user input to commit, transactions that never commit because of an error, and non-transactional queries inside transactions cause scalability and performance problems because they lock resources longer than needed.
* **Misuse of indexes**. If you do not create indexes that support the queries that are issued against your server, the performance of your application suffers as a result. However, if you have too many indexes, the insert and update performance of your application suffers. You have to find a balance between the indexing needs of the writes and reads that is based on how your application is used.
* **Mixing OLTP, OLAP, and reporting workloads**. OLTP workloads are characterized by many small transactions, with an expectation of very quick response time from the user. OLAP and reporting workloads are characterized by a few long-running operations that might consume more resources and cause more contention. The long-running operations are caused by locking and by the underlying physical sub-system. You must resolve this conflict to achieve a scalable system.
* **Inefficient schemas**. Adding indexes can help improve performance. However, their impact may be limited if your queries are inefficient because of poor table design that results in too many join operations or in inefficient join operations. Schema design is a key performance factor. It also provides information to the server that may be used to optimize query plans. Schema design is largely a tradeoff between good read performance and good write performance. Normalization helps write performance. Denormalization helps read performance.
* **Using an inefficient disk subsystem**. The physical disk subsystem must provide a database server with sufficient I/O processing power to permit the database server to run without disk queuing or long I/O waits.

**Schema**

Good, efficient schema design is essential for high performance data access. Consider the following guidelines when you design your database schema:

* **Devote the appropriate resources to schema design**.
* **Separate OLAP and OLTP workloads**.
* **Normalize first, denormalize for performance later**.
* **Define all primary keys and foreign key relationships**.
* **Define all unique constraints and check constraints**.
* **Choose the most appropriate data type**.
* **Use indexed views for denormalization**.
* **Partition tables vertically and horizontally**.

**Devote the Appropriate Resources to Schema Design**

Too many organizations design tables at the last minute when the tables are needed for their queries. Take the time and devote the resources that are needed to gather the business requirements, to design the right data model, and to test the data model. Make sure that your design is appropriate for your business and that the design accurately reflects the relationships between all objects. Changing a data model after your system is already in production is expensive, time consuming, and inevitably affects a lot of code.

**Separate OLAP and OLTP Workloads**

OLAP and OLTP workloads on one server have to be designed to not impede each other. OLAP and reporting workloads tend to be characterized by infrequent, long-running queries. Users are rarely waiting impatiently for the queries to complete. OLTP workloads tend to be characterized by lots of small transactions that return something to the user in less than a second. Long-running queries for analysis, reports, or ad-hoc queries may block inserts and other transactions in the OLTP workload until the OLAP query completes.

If you need to support both workloads, consider creating a reporting server that supports the OLAP and reporting workloads. If you perform lots of analysis, consider using SQL Server Analysis Services to perform those functions.

**Normalize First, Denormalize for Performance Later**

You achieve a good, logical database design by applying normalization rules to your design. Normalization provides several benefits such as reducing redundant data. When you reduce redundant data, you can create narrow and compact tables. However, overnormalization of a database schema may affect performance and scalability. Obtaining the right degree of normalization involves tradeoffs. On the one hand, you want a normalized database to limit data duplication and to maintain data integrity. On the other hand, it may be harder to program against fully normalized databases, and performance can suffer.

Addresses are one part of a data model that is typically denormalized. Because many systems store multiple addresses for companies or people over long periods of time, it is relationally correct to have a separate address table and to join to that table to always get the applicable address. However, it is common practice to keep the current address duplicated in the person table or even to keep two addresses because this type of information is fairly static and is accessed often. The performance benefits of avoiding the extra join generally outweigh the consistency problems in this case.

The following denormalization approaches can help:

* Start with a normalized model, and then denormalize if necessary. Do not start with a denormalized model and then normalize it. Typically, each denormalization requires a compensating action to ensure data consistency. The compensating action may affect performance.
* Avoid highly abstracted object models that may be extremely flexible but are complex to understand and result in too many self-joins. For example, many things can be modeled by using an Object table, an Attributes table, and a Relationship table. This object model is very flexible, but the self-joins, the alias joins, and the number of joins become so cumbersome that it is not only difficult to write queries and understand them, but performance and scalability suffer. For an abstract object model, try to find some common object types that can be used as subtypes under the generic Object type, and then try to find the best balance between flexibility and performance.

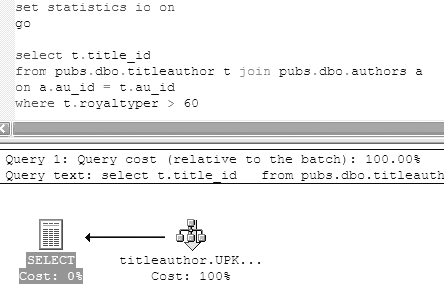
**Define All Primary Keys and Foreign Key Relationships**

Primary keys and foreign key relationships that are correctly defined help ensure that you can write optimal queries. One common result of incorrect relationships is having to add DISTINCT clauses to eliminate redundant data from result sets.

When primary and foreign keys are defined as constraints in the database schema, the server can use that information to create optimal execution plans.

Declarative referential integrity (DRI) performs better than triggers do, and DRI is easier to maintain and troubleshoot than triggers are. DRI is checked by the server before the server performs the actual data modification request. When you use triggers, the data modification requests are inserted in the Inserted and Deleted temporary system tables, and the trigger code is run. Depending on the trigger code, the final modifications are then made or not made,

The sample screen shot in Figure 14.2 shows an execution plan that accesses only one table, although two tables are included in a join in the query. Because there is a declared foreign key relationship between the **authors** table and the **titleauthor** table, and the **au\_id** column in the **titleauthor** table is not allowed to be null, the optimizer knows it does not need to access the **authors** table to resolve the query. The result of the SET STATISTICS IO command also shows that the **authors** table is never accessed.

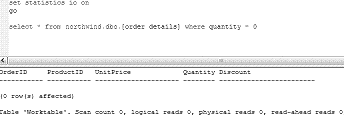


**Figure 14.2: Sample execution plan**

**Define All Unique Constraints and Check Constraints**

Unique constraints and check constraints provide more information for the optimizer to use to create optimal execution plans. A unique constraint gives the optimizer information about the expected results. A check constraint can be used to determine whether a table or index has to be accessed to find a result set.

Figure 14.3 shows a query that references a table that is not scanned at execution time because the optimizer knows from the check constraint that no rows can be returned. To try this example, create a check constraint on the **Quantity** column that allows only values greater than zero. The SET STATISTICS IO command output shows no physical or logical reads and a scan count of zero. The output shows this because the constraint information answered the query.

[](http://msdn.microsoft.com/en-us/library/Ff647793.ch14-check-constraint-example(en-us,PandP.10).gif)

**Figure 14.3: Example of a check constraint that prevents unnecessary reads**

**More Information**

For more information, see MSDN article, "SET STATISTICS IO," at <http://msdn.microsoft.com/en-us/library/aa259191(SQL.80).aspx>.

**Choose the Most Appropriate Data Type**

Choose the most appropriate data type, with the appropriate size and nullability. Consider each of the following when you are choosing a data type:

* Try to choose the smallest data type that works for each column. Also, choose the most appropriate type because both explicit and implicit conversions may be costly in terms of the time that it takes to do the conversion. They also may be costly in terms of the table or index scans that may occur because the optimizer cannot use an index to evaluate the query.
* Try to avoid nullable foreign key columns to limit the amount of outer joins that might need to be written. Outer joins tend to be more expensive to process than inner joins. If there are cases where the foreign key value may not be known, consider adding a row in the other table that would be the unknown case. Some database architects use one row for the unknown case, one row for the case that is not applicable, and one row for the case that is not determined yet. This approach not only allows for inner joins rather than outer joins, but it provides more information about the actual nature of the foreign key value.
* Columns that use the text data type have extra overhead because they are stored separately on text/image pages rather than on data pages. Use the **varchar** type instead of **text** for superior performance for columns that contain less than 8,000 characters.
* The **sql\_variant** data type allows a single column, parameter, or variable to store data values of different data types like **int** and **nchar**. However, each instance of a **sql\_variant** column records the data value and additional metadata. The metadata includes the base data type, maximum size, scale, precision, and collation. While **sql\_variant** provides flexibility, the use of **sql\_variant** affects performance because of the additional data type conversion.
* Unicode data types like **nchar** and **nvarchar** take twice as much storage space compared to ASCII data types like **char** and **varchar**. The speed factors specific to SQL Server are discussed in the article referenced in the following "More Information" section. However, note that strings in the Microsoft .NET Framework and in the Microsoft Windows 2000 kernel are Unicode. If you need or anticipate needing Unicode support, do not hesitate to use them.

**More Information**

For more information, see the "Performance and Storage Space" section of "International Features in Microsoft SQL Server 2000" on MSDN at <http://msdn.microsoft.com/en-us/library/aa902644(SQL.80).aspx>.

**Use Indexed Views for Denormalization**

When you have joins across multiple tables that do not change frequently, such as domain or lookup tables, you can define an indexed view for better performance. An indexed view is a view that is physically stored like a table. The indexed view is updated by SQL Server when any of the tables that the indexed view is based on are updated. This has the added benefit of pulling I/O away from the main tables and indexes.

**Partition Tables Vertically and Horizontally**

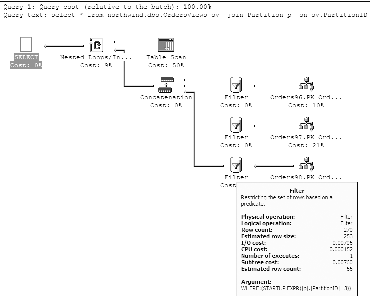
You can use vertical table partitioning to move infrequently used columns into another table. Moving the infrequently used columns makes the main table narrower and allows more rows to fit on a page.

Horizontal table partitioning is a bit more complicated. But when tables that use horizontal table partitioning are designed correctly, you may obtain huge scalability gains. One of the most common scenarios for horizontal table partitioning is to support history or archive databases where partitions can be easily delineated by date. A simple method that you can use to view the data is to use partitioned views in conjunction with check constraints.

Data-dependent routing is even more effective for very large systems. With this approach, you use tables to hold partition information. Access is then routed to the appropriate partition directly so that the overhead of the partitioned view is avoided.

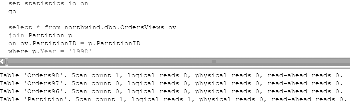
If you use a partitioned view, make sure that the execution plan shows that only the relevant partitions are being accessed. Figure 14.4 shows an execution plan over a partitioned view on three orders tables that have been horizontally partitioned by the **OrderDate** column. There is one table per year for 1996, 1997, and 1998. Each table has a **PartitionID** column that has a check constraint. There is also a partition table that includes a **PartitionID** and the year for that partition. The query then uses the partition table to get the appropriate **PartitionID** for each year and to access only the appropriate partition.

Although the graphical query plan includes both tables in the plan, moving the mouse over the **Tip on the Filter** icon shows that this is a start filter, as seen in the STARTUP clause in the argument of the filter. A start filter is a special type of filter that you want to see in plans that use partitioned views.

[](http://msdn.microsoft.com/en-us/library/Ff647793.ch14-execution-plan-with-filter-details(en-us,PandP.10).gif)

**Figure 14.4: An execution plan that shows the filter details**

Note that the SET STATISITCS IO output shown in Figure 14.5 shows that only the Orders98 table was actually accessed.

[](http://msdn.microsoft.com/en-us/library/Ff647793.ch14-set-statistics-io-output(en-us,PandP.10).gif)

**Figure 14.5: SET STATISTICS IO output**

**More Information**

For more information about the graphical execution plan, see "Graphically Displaying the Execution Plan Using SQL Query Analyzer" on MSDN at <http://msdn.microsoft.com/en-us/library/aa178423(SQL.80).aspx>.

**Queries**

Writing efficient queries in SQL Server is more an exercise in writing elegant relational queries than in knowing specific tricks and syntax tips. Generally, a well-written, relationally correct query written against a well-designed relationally correct database model that uses the correct indexes produces a system that performs fairly well and that is scalable. The following guidelines may help you create efficient queries:

* **Know the performance and scalability characteristics of queries**.
* **Write correctly formed queries**.
* **Return only the rows and columns needed**.
* **Avoid expensive operators such as NOT LIKE**.
* **Avoid explicit or implicit functions in WHERE clauses**.
* **Use locking and isolation level hints to minimize locking**.
* **Use stored procedures or parameterized queries**.
* **Minimize cursor use**.
* **Avoid long actions in triggers**.
* **Use temporary tables and table variables appropriately**.
* **Limit query and index hints use**.
* **Fully qualify database objects**.

**Know the Performance and Scalability Characteristics of Queries**

The best way to achieve performance and scalability is to know the characteristics of your queries. Although it is not realistic to monitor every query, you should measure and understand your most commonly used queries. Do not wait until you have a problem to perform this exercise. Measure the performance of your application throughout the life cycle of your application.

Good performance and scalability also requires the cooperation of both developers and database administrators. The process depends on both query development and index development. These areas of development typically are found in two different job roles. Each organization has to find a process that allows developers and database administrators to cooperate and to exchange information with each other. Some organizations require developers to write appropriate indexes for each query and to submit an execution plan to the database architect. The architect is responsible for evaluating the system as a whole, for removing redundancies, for finding efficiencies of scale, and for acting as the liaison between the developer and the database administrator The database administrator can then get information on what indexes might be needed and how queries might be used. The database administrator can then implement optimal indexes.

In addition, the database administrator should regularly monitor the SQL query that consumes the most resources and submit that information to the architect and developers. This allows the development team to stay ahead of performance issues.

**Write Correctly Formed Queries**

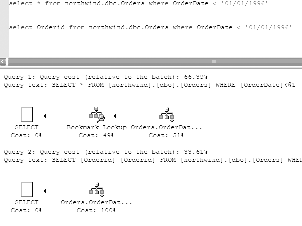
Ensure that your queries are correctly formed. Ensure that your joins are correct, that all parts of the keys are included in the ON clause, and that there is a predicate for all queries. Pay extra attention to ensure that no cross products result from missing ON or WHERE clauses for joined tables. Cross products are also known as Cartesian products.

Do not automatically add a DISTINCT clause to SELECT statements. There is no need to include a DISTINCT clause by default. If you find that you need it because duplicate data is returned, the duplicate data may be the result of an incorrect data model or an incorrect join. For example, a join of a table with a composite primary key against a table with a foreign key that is referencing only part of the primary key results in duplicate values. You should investigate queries that return redundant data for these problems.

**Return Only the Rows and Columns Needed**

One of the most common performance and scalability problems are queries that return too many columns or too many rows. One query in particular that returns too many columns is the often-abused SELECT \* FROM construct. Columns in the SELECT clause are also considered by the optimizer when it identifies indexes for execution plans. Using a SELECT \* query not only returns unnecessary data, but it also can force clustered index scans for the query plan, regardless of the WHERE clause restrictions. This happens because the cost of going back to the clustered index to return the remaining data from the row after using a non-clustered index to limit the result set is actually more resource-intensive than scanning the clustered index.

The query shown in Figure 14.6 shows the difference in query cost for a SELECT \* compared to selecting a column. The first query uses a clustered index scan to resolve the query because it has to retrieve all the data from the clustered index, even though there is an index on the **OrderDate** column. The second query uses the **OrderDate** index to perform an index seek operation. Because the query returns only the **OrderID** column, and because the **OrderID** column is the clustering key, the query is resolved by using only that index. This is much more efficient; the query cost relative to the batch is 33.61 percent rather than 66.39 percent. These numbers may be different on your computers.

[](http://msdn.microsoft.com/en-us/library/Ff647793.ch14-select-comparison(en-us,PandP.10).gif)

**Figure 14.6: Difference in query cost for a SELECT \* query compared to selecting a column**

Often, too many rows are returned because the application design allows a user to select large result sets from search forms. Returning hundreds or even thousands of results to a user stresses the server, the network, and the client. A large amount of data is generally not what the end user requires. Use a design pattern that supports paging, and return only a page or two of the requested data at a time.

Queries that call other queries that return too many columns and rows to the calling query are another often-overlooked consideration. This includes queries that are written as views or table-valued functions or views. Although views are useful for many reasons, they may return more columns than you need, or they may return all the rows in the underlying table to the calling query.

**More Information**

For more information about data paging, see "[How To: Page Records in .NET Applications](http://msdn.microsoft.com/en-us/library/ff650700.aspx)" in the "How To" section of this guide.

**Avoid Expensive Operators Such as NOT LIKE**

Some operators in joins or predicates tend to produce resource-intensive operations. The LIKE operator with a value enclosed in wildcards ("%*a value*%") almost always causes a table scan. This type of table scan is a very expensive operation because of the preceding wildcard. LIKE operators with only the closing wildcard can use an index because the index is part of a B+ tree, and the index is traversed by matching the string value from left to right.

Negative operations, such as <> or NOT LIKE, are also very difficult to resolve efficiently. Try to rewrite them in another way if you can. If you are only checking for existence, use the IF EXISTS or the IF NOT EXISTS construct instead. You can use an index. If you use a scan, you can stop the scan at the first occurrence.

**Avoid Explicit or Implicit Functions in WHERE Clauses**

The optimizer cannot always select an index by using columns in a WHERE clause that are inside functions. Columns in a WHERE clause are seen as an expression rather than a column. Therefore, the columns are not used in the execution plan optimization. A common problem is date functions around **datetime** columns. If you have a **datetime** column in a WHERE clause, and you need to convert it or use a data function, try to push the function to the literal expression.

The following query with a function on the **datetime** column causes a table scan in the NorthWind database, even though there is an index on the **OrderDate** column:

SELECT OrderID FROM NorthWind.dbo.Orders WHERE DATEADD(day, 15,

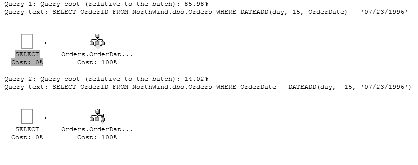
OrderDate) = '07/23/1996'

However, by moving the function to the other side of the WHERE equation, an index can be used on the **datetime** column. This is shown in the following example:

SELECT OrderID FROM NorthWind.dbo.Orders WHERE OrderDate = DATEADD(day,

-15, '07/23/1996')

The graphical execution plan for both of these queries is shown in Figure 14.7, which shows the difference in plans. Note the **Scan** icon for the first query and the **Seek** icon for the second query. Figure 14.7 also shows the comparative difference in query costs between the two queries; the first query has an 85.98 percent cost compared to the 14.02 percent cost for the second query. The costs on your computer may be different.

[](http://msdn.microsoft.com/en-us/library/Ff647793.ch14-query-comparison(en-us,PandP.10).gif)

**Figure 14.7: Query comparison**

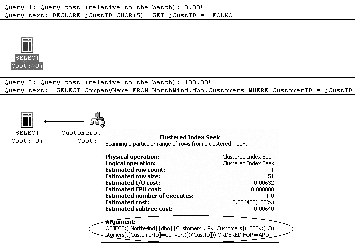
Implicit conversions also cause table and index scans, often because of data type mismatches. Be especially wary of **nvarchar** and **varchar** data type mismatches and **nchar** and **char** data type mismatches that cause an implicit conversion. You can see these in the following execution plan. The following example uses a local variable of type **char** against an **nchar** column in the Customers table. The type mismatch causes an implicit conversion and a scan in this example:

DECLARE @CustID CHAR(5)

SET @CustID = 'FOLKO'

SELECT CompanyName FROM NorthWind.dbo.Customers WHERE CustomerID = @CustID

Figure 14.8 shows the results of the type mismatch.

[](http://msdn.microsoft.com/en-us/library/Ff647793.ch14-type-mismatch(en-us,PandP.10).gif)

**Figure 14.8: Output showing an implicit conversion**

**Use Locking and Isolation Level Hints to Minimize Locking**

Locking has a huge impact on performance and scalability. Locking also affects perceived performance because of the wait for the locked object. All applications experience a certain level of locking. The key is to understand the type of locking that is occurring, the objects that are being locked, and most importantly, the duration of each locking occurrence.

There are three basic types of locks in SQL Server:

* Shared
* Update
* Exclusive

**Note** There are also intent, schema, and bulk update locks, but these locks are less significant and are not addressed in this chapter.

Shared locks are compatible with other shared locks, but they are not compatible with exclusive locks. Update locks are compatible with shared locks, but they are not compatible with exclusive locks or with other update locks. Exclusive locks are not compatible with shared locks, update locks, or other exclusive locks. Different types of locks are held for different amounts of time to obtain the requested isolation level.

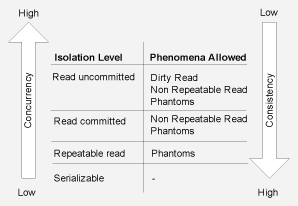
There are four ANSI isolation levels that can be specified for transactions in SQL Server:

* Read uncommitted
* Read committed
* Repeatable read
* Serializable

Each of these isolation levels allow zero or more of the isolation level phenomena to occur:

* **Dirty reads**. Dirty reads are transactions that see the effects of other transactions that were never committed.
* **Nonrepeatable reads**. Nonrepeatable reads are transactions that see only committed data from other transactions. In a nonrepeatable read, data changes when it is referenced multiple times in the transaction.
* **Phantoms**. Phantoms are transactions that see or that do not see rows that are inserted or deleted from another transaction that is not committed yet.

The default isolation level in SQL Server 2000 is read committed. Figure 14.9 shows the phenomena that are allowed at each isolation level.



**Figure 14.9: ANSI isolation levels**

Instead of accepting the default SQL Server isolation level of read committed, you can explicitly select the appropriate isolation level for code. You can do this by using isolation levels or locking hints.

**WITH NOLOCK and WITH READUNCOMMITTED**

If you designed your application to use locking hints, use the WITH (NOLOCK) or WITH (READUNCOMMITTED) table hint in SELECT statements to avoid generating read locks that may not be required. This can provide a significant increase in scalability, especially where SELECT statements are run at a serializable isolation level because the SELECT statement is called within an explicit transaction that starts in a middle-tier object using Microsoft Transaction Server (MTS), COM+, or Enterprise Services. Another approach is to determine if the transaction as a whole can run at a lower isolation level. You can use the SET TRANSACTION ISOLATION LEVEL command to change the isolation level for all transactions in a SQL Server session.

**UPDLOCK**

A common technique for handling deadlocks is to use an UPDLOCK table hint on SELECT statements that are commonly involved in transactions that deadlock. The UPDLOCK issues update locks, and it holds the locks until the end of the transaction. The typical shared lock that is issued by a SELECT statement is only held until the row has been read. By holding the update lock until the end of the transaction, other users can still read the data but cannot acquire a lock that you may need later. This is a common deadlock scenario.

**TABLOCK**

You can use the TABLOCK table hint to improve performance when you use the bulk insert command. When there are large amounts of inserts, requesting a lock on the entire table helps by relieving the lock manager of the overhead of managing dynamic locking. However, requesting a lock on the entire table blocks all other users on the table. It therefore is not something you should do when other users need to use the system.

To use locking and isolation level locking hints effectively, you have to understand locking behavior in SQL Server and the specific needs of your application. You can then select the best mechanism for key queries. In general, the default isolation level and locking of SQL Server is best, but you can increase scalability by using other locking hints when you need to.

**Use Stored Procedures or Parameterized Queries**

Significant work has been done in SQL Server 2000 to optimize dynamic code, especially with the addition of the **sp\_executesql** system stored procedure and the ability to reuse execution plans for parameterized queries. However, stored procedures still generally provide improved performance and scalability.

Consider the following issues when you decide whether to store your SQL commands on the server by using stored procedures or to embed commands in your application by using embedded SQL statements:

* **Logic separation**. When you design your data access strategy, separate business logic from data manipulation logic for performance, maintainability and flexibility benefits. Validate business rules before you send the data to the database to help reduce network trips. Separate your business logic from data manipulation logic to isolate the impact of database changes or business rule changes. Use stored procedures to clarify the separation by moving the data manipulation logic away from the business logic so the two do not become intertwined. Establish standards that identify the proper coding standards to avoid intermingling of logic.
* **Tuning and deployment**. Stored procedure code is stored in the database and allows database administrators to review data access code and to tune both the stored procedures and the database independent of the deployed application. You do not always need to redeploy your application when stored procedures change. Embedded SQL is deployed as part of the application code and requires database administrators to profile the application to identify the SQL that is actually used. This complicates tuning, and you must redeploy the application if any changes are made.
* **Network bandwidth**. Source code for stored procedures is stored on the server, and you only send the name and parameters across the network to the server. However, when you use embedded SQL, the full source of the commands must be transmitted each time the commands are run. By using stored procedures, you can reduce the amount of data sent to the server, particularly when large SQL operations are frequently run.
* **Simplified batching of commands**. Stored procedures offer simplified and more maintainable batching of work.
* **Improved data security and integrity**. Stored procedures are strongly recommended to ensure data security, to promote data integrity, and to support performance and scalability. Administrators can secure the tables against direct access or manipulation. Users and applications are granted access to the stored procedures that enforce data integrity rules. Using embedded SQL typically requires advanced permissions on tables and may allow unauthorized modification of data.
* **SQL injection**. Avoid using dynamically generated SQL with user input. SQL injection may occur when malicious user input is used to perform unauthorized actions such as retrieving too much data or destructively modifying data. Parameterized stored procedures and parameterized SQL statements can both help reduce the likelihood of SQL injection. By using the parameters collections, you force parameters to be treated as literal values rather than executable code. You should also constrain all user input to reduce the likelihood of a SQL injection attack.

**More Information**

For more information about how to prevent SQL injection, see Chapter 14, "Building Secure Data Access," in *Improving Web Application Security: Threats and Countermeasures* at <http://msdn.microsoft.com/en-us/library/aa302430.aspx>.

**Minimize Cursor Use**

Cursors force the database engine to repeatedly fetch rows, negotiate blocking, manage locks, and transmit results. Use forward-only and read-only cursors unless you need to update tables. More locks may be used than are needed, and there is an impact on the tempdb database. The impact varies according to the type of cursor used.

The forward-only, read-only cursor is the fastest and least resource-intensive way to get data from the server. This type of cursor is also known as a firehose cursor or a local fast-forward cursor. If you feel that you really need to use a cursor, learn more about the different types of cursors, their locking, and their impact on the tempdb database.

Often, cursors are used to perform a function row by row. If there is a primary key on a table, you can usually write a WHILE loop to do the same work without incurring the overhead of a cursor. The following example is very simple but demonstrates this approach:

declare @currid int

select @currid = min(OrderID)

from Orders where OrderDate < '7/10/1996'

while @currid is not null

begin

print @currid

select @currid = min(OrderID)

from Orders

where OrderDate < '7/10/1996'

and OrderID > @currid

end

**More Information**

For more information about cursors, see "Transact-SQL Cursors" at <http://msdn.microsoft.com/en-us/library/aa172595(SQL.80).aspx>.

**Avoid Long Actions in Triggers**

Trigger code is often overlooked when developers evaluate systems for performance and scalability problems. Because triggers are always part of the INSERT, UPDATE, or DELETE calling transactions, a long-running action in a trigger can cause locks to be held longer than intended, resulting in blocking of other queries. Keep your trigger code as small and as efficient as possible. If you need to perform a long-running or resource-intensive task, consider using message queuing to accomplish the task asynchronously.

**Use Temporary Tables and Table Variables Appropriately**

If your application frequently creates temporary tables, consider using the **table** variable or a permanent table. You can use the **table** data type to store a row set in memory. **Table** variables are cleaned up automatically at the end of the function, stored procedure, or batch that they are defined in. Many requests to create temporary tables may cause contention in both the tempdb database and in the system tables. Very large temporary tables are also problematic. If you find that you are creating many large temporary tables, you may want to consider a permanent table that can be truncated between uses.

Table variables use the tempdb database in a manner that is similar to how table variables use temporary tables, so avoid large table variables. Also, table variables are not considered by the optimizer when the optimizer generates execution plans and parallel queries. Therefore, table variables may cause decreased performance. Finally, table variables cannot be indexed as flexibly as temporary tables.

You have to test temporary table and table variable usage for performance. Test with many users for scalability to determine the approach that is best for each situation. Also, be aware that there may be concurrency issues when there are many temporary tables and variables that are requesting resources in the tempdb database.

**Limit Query and Index Hints Use**

Although the previous section discusses how to use table hints to limit locking, you should use query and index hints only if necessary. Query hints include the MERGE, HASH, LOOP, and FORCE ORDER hints that direct the optimizer to select a specific join algorithm. Index hints are table hints where a certain index is specified for the optimizer to use. Generally the optimizer chooses the most efficient execution plan. Forcing an execution plan by specifying an index or a join algorithm should be a last resort. Also, remember that SQL Server uses a cost-based optimizer; costs change over time as data changes. Hints may no longer work for a query, and the hint may never be reevaluated.

If you find that the optimizer is not choosing an optimal plan, try breaking your query into smaller pieces. Or, try another approach to the query to obtain a better plan before you decide to use hard-coded query hints.

**Fully Qualify Database Objects**

By fully qualifying all database objects with the owner, you minimize overhead for name resolution, and you avoid potential schema locks and execution plan recompiles. For example, the SELECT \* FROM dbo.Authors statement or the EXEC dbo.CustOrdersHist statement performs better than the SELECT \* FROM Authors or the EXEC CustOrderHist statements. In systems that have many stored procedures, the amount of time that is spent to resolve a non-qualified stored procedure name adds up.

**Indexes**

Indexes are vital to efficient data access. However, there is a cost associated with creating and maintaining an index structure. Having a large number of indexes on a table may result in faster select statements, but slower insert, update, and delete statements. The performance overhead varies by application and database. If you have a large number of indexes on a table, you increase the chance that the optimizer will choose a suboptimal index for a query plan.

Data from columns that comprise a particular index are stored in an index page. Indexes are built on B-tree structures formed of 8-KB index pages. There are *clustered* indexes and *non-clustered* indexes in SQL Server. With non-clustered indexes, the leaf level nodes contain only the index data with a pointer to the associated data page where the remaining data resides. As a result, data access that uses a non-clustered index may cause extra reads of the data from the data page. With clustered indexes, the leaf level nodes of the B-tree contain the actual data rows for the table. There is only one clustered index per table. Remember that the clustering key is used in all non-clustered indexes as the row identifier, so choose them wisely.

Indexing is an art and is dependant on data distribution, cost, usage, and an understanding of how SQL Server uses indexes. It takes time to get it right. Use the following guidelines to help create efficient indexes:

* **Create indexes based on use**.
* **Keep clustered index keys as small as possible**.
* **Consider range data for clustered indexes**.
* **Create an index on all foreign keys**.
* **Create highly selective indexes**.
* **Consider a covering index for often-used, high-impact queries**.
* **Use multiple narrow indexes rather than a few wide indexes**.
* **Create composite indexes with the most restrictive column first**.
* **Consider indexes on columns used in WHERE, ORDER BY, GROUP BY, and DISTINCT clauses**.
* **Remove unused indexes**.
* **Use the Index Tuning Wizard**.

**Create Indexes Based on Use**

Indexes come at a cost that must be balanced between write and read operations. Write operations may be negatively and positively affected by indexes. Read operations are mostly benefited by indexes. You have to understand the way that your system is used to find the optimal indexes. Esoteric discussions about the degree to which insert operation performance is affected by indexes are of limited value if the number of insert operations is small, and your system performs intensive read operations. Spend time evaluating indexes for the most commonly queried tables, the most commonly used queries, and the most problematic queries. Design indexes to support these tables and queries in a systemic manner. As mentioned previously, designing indexes is an art, not a science. It takes knowledge of your system to create effective indexes.

Do not create indexes if a table is rarely queried, or if a table does not ever seem to be used by the optimizer. Avoid indexes on **bit**, **text**, **ntext**, or **image** data types because they are rarely used. Avoid very wide indexes and indexes that are not selective.

**Keep Clustered Index Keys As Small As Possible**

Because non-clustered indexes store clustered index keys as their row locators. The row locators reference the actual data row. Therefore, keep clustered index keys as small as possible.

**Consider Range Data for Clustered Indexes**

If you frequently query the database for ranges of rows by using clauses such as BETWEEN, or operators such as > and <, consider a clustered index on the column specified by the WHERE clause. Generally, clustered indexes are not as effective for primary keys in transaction tables, but they are very effective for primary keys in domain or lookup tables that may never be used other than in joins. In general, every table should have a clustered index unless there is a demonstrated performance reason not to have one.

**Create an Index on All Foreign Keys**

Be sure to create an index on any foreign key. Because foreign keys are used in joins, foreign keys almost always benefit from having an index.

**Create Highly Selective Indexes**

Create indexes that exhibit high selectivity. In other words, create indexes that have many distinct values. For example, an index on a region column may have a small number of distinct values. Therefore, there may not be enough distinct values for the optimizer to use. Another example of an item that may not have enough distinct values is a **bit** column. Since there are only two values, an index cannot be very selective and as a result, the index may not be used.

Use the DBCC SHOW\_STATISTICS command on a table or index to better understand the statistics on columns in an index. In the output of this command, density is used to indicate selectivity. Density is calculated as one divided by the number of distinct values. Therefore, a unique index has a density of *1/number of rows*. For example, a table with 1,000 rows would have a density of 0.001. An index on a **bit** column has a density of 0.5 because you divide one by the only two possible unique values in a **bit** column. The smaller the density number is, the greater the selectivity.

The best numbers to use for density are the **All Density** numbers in the DBCC SHOW\_STATISTICS command output, not the **Density** number in the first result that is produced.

Figure 14.10 shows the DBCC SHOW\_STATISTICS command output for the PK\_Orders index on an orders table. The output shows a very selective density because it uses the primary key.

[Description: Click here for larger image](http://msdn.microsoft.com/en-us/library/Ff647793.ch14-dbcc-show-stats-output(en-us,PandP.10).gif)

**Figure 14.10: DBCC SHOW\_STATISTICS output**

**More Information**

For more information about statistics, see "Statistics Used by the Query Optimizer in SQL Server 2000" at <http://msdn.microsoft.com/en-us/library/aa902688(SQL.80).aspx>.

**Consider a Covering Index for Often-Used, High-Impact Queries**

Queries that are frequently called, problematic queries, or queries that use lots of resources are good candidates for a covering index. A covering index is an index that includes all the columns that are referenced in the WHERE and SELECT clauses. The index "covers" the query, and can completely service the query without going to the base data. This is in effect a materialized view of the query. The covering index performs well because the data is in one place and in the required order. A covering index may improve scalability by removing contention and access from the main table.

**Use Multiple Narrow Indexes Rather than a Few Wide Indexes**

SQL Server can use multiple indexes per table, and it can intersect indexes. As a result, you should use multiple narrow indexes that consist of only one column because narrow indexes tend to provide more options than wide composite indexes.

Also, statistics are only kept for the first column in a composite index. Multiple single column indexes ensure statistics are kept for those columns. Composite indexes are of greatest value as covering indexes. Because the first column is the column with statistics, you typically use composite indexes if that column is also a reference in the WHERE clause.

A side consideration of creating smaller indexes is the use of the CHECKSUM function to create a hash index on a very wide column. This allows you to create smaller indexes. It is a good approach when you need an index on long character columns where you also need to limit your use of space.

**Create Composite Indexes with the Most Restrictive Column First**

When you create a composite index, remember that only the first column stores statistics. Try to make that column the most restrictive column. If the composite index is not selective enough, the optimizer may not use it. Also, a WHERE clause that does not use all the columns included in the composite index may cause the index not to be used. For example, a WHERE clause that skips a column in the middle of the composite index may cause the index not to be used.

**Consider Indexes on Columns Used in WHERE, ORDER BY, GROUP BY, and DISTINCT Clauses**

Consider creating an index on columns that are used in WHERE clauses and in aggregate operations such as GROUP BY, DISTINCT, MAX, MIN, or ORDER BY. These generally benefit from an index, although you need to measure and test to validate that there is a benefit in your scenario.

**Remove Unused Indexes**

Be sure to remove all unused or out-of-date indexes. Unused or out-of-date indexes continue to impact write operations because they need to be maintained even though they are not used. They are still used by the optimizer in execution plan considerations. You can use SQL Profiler to determine the indexes that are used.

**Use the Index Tuning Wizard**

The Index Tuning Wizard (ITW) uses the same information and statistics that the optimizer uses to create an execution plan. You should use this tool to obtain guidance and tips on index options that might otherwise be overlooked. However, it is not the only tool, and system knowledge is still the best way to create efficient indexes. Capture a representative trace by using SQL Profiler as input to the ITW for more system-wide index suggestions.

**Transactions**

Efficient transaction handling significantly enhances scalability. You have to be careful to code transactions correctly. Transactions hold locks on resources that can block other transactions. The following recommendations are some of the more effective things that you can do to create efficient transactions:

* **Avoid long-running transactions**.
* **Avoid transactions that require user input to commit**.
* **Access heavily used data at the end of the transaction**.
* **Try to access resources in the same order**.
* **Use isolation level hints to minimize locking**.
* **Ensure that explicit transactions commit or roll back**.

**Avoid Long-Running Transactions**

Locks are held during transactions, so it is critical to keep transactions as short as possible. Do not forget that you can start transactions from the application layer. A common technique is to do all the needed validation checking before you start the transaction. You still have to check again during the transaction, but you avoid many situations where you start a transaction and then have to roll back the transaction.

**Avoid Transactions that Require User Input to Commit**

Be careful not to start a transaction that requires user input to commit. In this case. the transaction locks are held until the input is received at some indeterminate time in the future.

**Access Heavily Used Data at the End of the Transaction**

Try to put all the read operations in a transaction at the beginning, put the write operations at the end, and put the most contentious resources at the very end. This ensures that the shortest locks are held against the resources that are most often used by others. Creating your transactions in this way helps limit blocking of other transactions.

**Try to Access Resources in the Same Order**

Reduce deadlocks by using resources in the same order. For example, if stored procedures SP1 and SP2 use tables T1 and T2, make sure that both SP1 and SP2 process T1 and T2 in the same order. Otherwise, if SP1 uses T1 and then T2, and SP2 uses T2 and then T1, each stored procedure could be waiting to use a resource that the other stored procedure is already using. The result is a deadlock when this happens.

Avoiding locking conflicts in a multiuser scenario is not easy. Your goal should be to reduce deadlock opportunities and to reduce the length of time locks are held.

**Use Isolation Level Hints to Minimize Locking**

If your business logic allows it, lower the isolation level to one that is less restrictive. A common way to lower isolation levels is to use the WITH NOLOCK hint on SELECT statements in transactions.

**Ensure That Explicit Transactions Commit or Roll Back**

All transactional code should have explicit error handling that either commits or rolls back on an error. This type of error handling allows open transactions that are holding locks to release the locks when the transaction cannot complete. Otherwise, the transaction never would release the lock. You can use the DBCC OPENTRAN command to find transactions that may be open for long periods.

**Stored Procedures**

Stored procedures provide improved performance and scalability to your systems. When you develop stored procedures, keep the following recommendations in mind:

* **Use Set NOCOUNT ON in stored procedures**.
* **Do not use the sp\_ prefix for custom stored procedures**.

**Use Set NOCOUNT ON in Stored Procedures**

Use the SET NOCOUNT ON statement to prevent SQL Server from sending the DONE\_IN\_PROC message for each statement in a stored procedure. For example, if you have eight operations in a stored procedure and you have not used this option eight messages are returned to the caller. Each message contains the number of affected rows for the respective statement.

**Do Not Use the Sp\_ Prefix for Custom Stored Procedures**

SQL Server always looks in the master database for a stored procedure that begins with the **sp\_** prefix. SQL Server then uses any supplied qualifiers such as the database name or owner. Therefore, if you use the **sp\_** prefix for a user-created stored procedure, and you put it in the current database, the master database is still checked first. This occurs even if you qualify the stored procedure with the database name. To avoid this issue, use a custom naming convention, and do not use the **sp\_** prefix.

**Execution Plans**

To improve performance, it is critical to understand and measure the current performance of your T-SQL code. A common mistake is to invest too much effort in writing an elegant piece of code and in worrying about specific coding tricks and tips. Instead, you should remember to look at the execution plan of your queries and understand how your queries are run. The following guidelines outline some of the main ways you can improve the performance and scalability of T-SQL code:

* **Evaluate the query execution plan**.
* **Avoid table and index scans**.
* **Evaluate hash joins**.
* **Evaluate bookmarks**.
* **Evaluate sorts and filters**.
* **Compare actual versus estimated rows and executions**.

**Evaluate the Query Execution Plan**

In SQL Query Analyzer, enable the **Display Execution Plan** option, and run your query against a representative data load to see the plan that is created by the optimizer for the query. Evaluate this plan, and then identify any good indexes that the optimizer could use. Also, identify the part of your query that takes the longest time to run and that might be better optimized. Understanding the actual plan that runs is the first step toward optimizing a query. As with indexing, it takes time and knowledge of your system to be able to identify the best plan.

**Avoid Table and Index Scans**

Table and index scans are expensive operations, and they become more expensive as data grows. Investigate every table or index scan that you see in an execution plan. Can an index be created that would allow a seek operation instead of a table scan? Eliminating unnecessary I/O caused by scans is one of the quickest ways to obtain a substantial improvement in performance.

Not all table or index scans are bad. The optimizer selects a scan for tables that have fewer than a few hundred rows, and a clustered index scan may be the most effective option for some queries. However, you generally should avoid scans.

**Evaluate Hash Joins**

Make sure that you investigate hash joins in a query execution plan. A hash join may be the best option, but frequently, a hash join is selected because there are no indexes that the optimizer can use to perform an efficient nested loop or merge join. In the absence of indexes, a hash join is the best option. However, better indexing may occur from a nested loop or merge join. Note that hash joins are also fairly CPU intensive. If you have high CPU usage, and you do not feel that enough work is being performed against the server to justify this, evaluate the execution plans by using SQL Profiler to find out if you have a lot of hash joins.

Queries that use a parallel execution plan often have to perform hash joins to recombine the finished parallel streams. Hash joins in this scenario are usually optimal and should not be a concern.

**Evaluate Bookmarks**

A bookmark in an execution plan indicates that an index was used to limit the table and that a bookmark was then used to probe the clustered index or the heap table to retrieve more data that is not available in the index. A bookmark is often used in this way to retrieve columns that are in a SELECT clause. This means that at least twice the I/O is necessary to retrieve the results.

A bookmark is not always a problem, but you should find out if adding a covering index might be more effective. A bookmark may not be a problem if the original index was very selective, in which case few bookmark lookups are needed. However, a bookmark to the data from an index that was not very selective would be problematic, especially if the resulting table rows spread across a significant percentage of the pages in the table.

**Evaluate Sorts and Filters**

Sorts and filters are both CPU intensive and memory intensive because the server performs these operations in memory. When there are instances of sorts and filters, find out if you can create an index that would support the sorting or the filtering. Filtering is often the result of an implicit conversion, so investigate the filter to learn if a conversion occurred. Sorts and filters are not always bad, but they are key indicators of potential problems, and you should investigate them further.

**Compare Actual vs. Estimated Rows and Executions**

When you read the output from a SHOWPLAN statement, start from the most-indented row that has the highest incremental change in the **TotalSubtreeCost** column. Carefully evaluate both the index selection and the optimizer's estimate by using the SET STATISTICS PROFILE ON command. This command runs the statement, so only use it on SELECT statements or T-SQL code that does not modify data, or you can preface the command with a BEGIN TRAN /ROLLBACK statement.

As an alternative, use the new profiler **Performance:Showplan Statistics** event in SQL 2000. This event belongs to event class 98. This event reports four columns that show estimated and actual rows and executions. You must select the Binary Data column before the profiler event adds data to the T-SQL or **SP:stmtcompleted** events.

Substantial differences in the estimated row count may indicate the optimizer had out-of-date statistics or skewed statistics. For example, if the estimated row count is 2 rows, and the actual row count is 50,000, the optimizer may have had out-of-date statistics or skewed statistics. Try using the UPDATE STATISTICS WITH FULLSCAN command.

**More Information**

For more information about the statistics that are used by Query Optimizer, see "Statistics Used by the Query Optimizer in Microsoft SQL Server 2000" at <http://msdn.microsoft.com/en-us/library/aa902688(SQL.80).aspx>.

For more information about query recompilation, see "Query Recompilation in SQL Server 2000" at <http://msdn.microsoft.com/en-us/library/aa902682(SQL.80).aspx>.

For more information about troubleshooting slow-running queries, see Knowledge Base article 243589, "HOW TO: Troubleshoot Slow-Running Queries on SQL Server 7.0 or Later," at <http://support.microsoft.com/default.aspx?scid=kb;en-us;243589>.

**Execution Plan Recompiles**

Performance is affected every time a query results in the creation of a new execution plan or when a plan is recompiled. Recompiles are not always a bad thing. The initial plan that was created may not be optimal for other calls or data may have changed. A recompile might be needed to create a better plan. The optimizer generally causes a recompile when it is necessary; however, there are steps that you can take to ensure that recompilation does not occur when it is not needed. The following guidelines help you avoid frequent recompiles:

* **Use stored procedures or parameterized queries**.
* **Use sp\_executesql for dynamic code**.
* **Avoid interleaving DDL and DML in stored procedures, including the tempdb database DDL**.
* **Avoid cursors over temporary tables**.

**Use Stored Procedures or Parameterized Queries**

The server saves execution plans for stored procedures and parameterized queries under most circumstances. This allows them to be reused on later calls.

**Use Sp\_executesql for Dynamic Code**

If you must use dynamic code in your application, try to wrap it in the **sp\_executesql** system stored procedure. This system stored procedure permits you to write parameterized queries in T-SQL and saves the execution plan for the code. If the dynamic code has little chance of being called again, there is no value in saving the execution plan because the execution plan will eventually be removed from the cache when the execution plan expires. Evaluate whether an execution plan should be saved or not. Note that wrapping code in the **sp\_executesql** system stored procedure without using parameters does not provide compile time performance savings.

Dynamic code is often used for query builder applications, it is often resource-intensive, and it is often reused in this scenario. Using the **sp\_executsql** system stored procedure to wrap this code can help improve performance.

**Avoid Interleaving DDL and DML in Stored Procedures, Including the Tempdb database DDL**

Interleaving data definition language (DDL) and data manipulation language (DML) in stored procedures is one of the most common causes of stored procedure recompiles. A common scenario is to create a temporary table, to insert data into that table, to create an index, and then to select data from the table. This sequence of events typically causes a recompile. To avoid recompiles, put all the DDL at the beginning of the stored procedure, and put the DML after the DDL.

The following code shows a stored procedure that creates a table (DDL), inserts data into that table (a DML statement), creates an index (a DDL statement), and then selects data from the table (another DML statement):

CREATE PROCEDURE RecompileExample @employeeID int

AS

SET NOCOUNT ON

CREATE TABLE #EmployeeOrders(OrderID int not null)

INSERT #EmployeeOrders

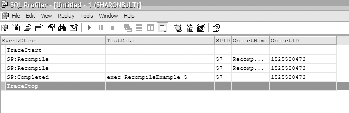
SELECT OrderID from Northwind.dbo.Orders WHERE EmployeeID = @EmployeeID

CREATE CLUSTERED INDEX EC ON #EmployeeOrders(OrderID)

SELECT \* FROM #EmployeeOrders ORDER BY OrderID

GO

By running SQL Profiler and capturing the **SP:Recompile** events, you can see a recompile every time the procedure that interleaves DDL and DML is run. This is shown in Figure 14.11. The recompiles that occur for this simple sample code are not likely to take much time. However, more complex queries may result in significant cost for the recompiles.

[](http://msdn.microsoft.com/en-us/library/Ff647793.ch14-sql-profiler-showing-recompiles(en-us,PandP.10).gif)

**Figure 14.11: SQL Profiler showing recompiles**

The following code puts all the DDL at the beginning so that there is no interleaving of DDL and DML. This means that a recompile is not required.

CREATE PROCEDURE NoRecompileExample @employeeID int

AS

SET NOCOUNT ON

CREATE TABLE #EmployeeOrders (OrderID int not null)

CREATE CLUSTERED INDEX EC ON #EmployeeOrders(OrderID)

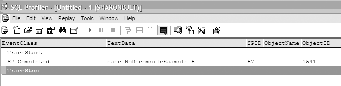
INSERT #EmployeeOrders

SELECT OrderID from Northwind.dbo.Orders WHERE EmployeeID = @EmployeeID

SELECT \* FROM #EmployeeOrders ORDER BY OrderID

GO

The SQL Profiler trace shown in Figure 14.12 for the revised code no longer shows a recompile.

[](http://msdn.microsoft.com/en-us/library/Ff647793.ch14-profiler-ouput-with-no-recompiles(en-us,PandP.10).gif)

**Figure 14.12: Profiler output with no recompiles**

**Avoid Cursors over Temporary Tables**

A cursor that has a DECLARE statement that selects data from a temporary table almost always causes a recompile. As a result, avoid using cursors over temporary tables.

**More Information**

For more information about query recompilation, see "Query Recompilation in SQL Server 2000" on MSDN at <http://msdn.microsoft.com/en-us/library/aa902682(SQL.80).aspx>.

**SQL XML**

SQL Server 2000 added a series of new XML features. While these are popular and flexible, you should be aware of some of the following scalability issues that are involved in using these new features:

* **Avoid OPENXML over large XML documents**.
* **Avoid large numbers of concurrent OPENXML statements over XML documents**.

**Avoid OPENXML over Large XML Documents**

Be aware that there are limitations to the amount of memory that is available to the OPENXML construct over an XML document operation. This operation builds a Document Object Model (DOM) in the SQL buffer space that can be much larger than the original document size. Also, this operation is limited to one eighth of the buffer space, and large XML documents may consume this memory fairly quickly and cause an out-of-memory error on the server. Do not create large systems based on this functionality without conducting significant load testing. You might also want to use the XML bulk load option if possible.

**Avoid Large Numbers of Concurrent OPENXML Statements over XML Documents**

You also have to consider the issue with OPENXML when you use OPENXML to batch inserts. This is a fairly common operation because it is an effective way to issue a group of inserts with one statement. Issuing a group of inserts reduces the overhead of multiple insert statements and multiple round trips. However, be aware that this approach may not be very scalable because of the aforementioned memory limitations.

**More Information**

For more information about OPENXML, see "Using OPENXML" on MSDN at <http://msdn.microsoft.com/en-us/library/ms187897(SQL.90).aspx>.

**Tuning**

To improve performance, it is critical to understand and measure the current performance of code. Note that tuning is an ongoing, iterative process. Because SQL Server uses a cost-based optimizer and because costs may change, the efficiency of particular queries can change over time.

The following guidelines outline some of the main ways to improve the performance and scalability of your T-SQL code:

* **Use SQL Profiler to identify long-running queries**.
* **Take note of small queries called often**.
* **Use sp\_lock and sp\_who2 to evaluate blocking and locking**.
* **Evaluate waittype and waittime in master..sysprocesses**.
* **Use DBCC OPENTRAN to locate long-running transactions**.

**Use SQL Profiler to Identify Long-Running Queries**

Use the SQL Profiler **SQLProfiler** **TSQL\_Duration** template to identify queries with the longest durations. Long-running queries have the greatest potential for locking resources longer, blocking other users, and limiting scalability. They are also the best candidates for optimization. Reviewing long-running queries is a continuous process and requires representative loads to ensure effective tuning.

In some cases, using the SQL Profiler templates is somewhat limiting when you use them to measure the change in performance while you test new indexes or application design changes. SQL Profiler can become a very powerful baseline tool when you save the results of your performance test as a trace file. Trace files use the .trc extension. Beginning with SQL Server 2000, you can use these trace files to write automated reports that quantitatively measure gains in the performance of your application for certain query types that otherwise would not be grouped properly when using the template. There is the clever **fn\_trace\_gettable** trace reporting function. This trace function is shown in the following sample code:

SELECT Count(\*) as CountOfEvents,

AVG(Duration) AS AvgDuration,

SUM(Duration) AS [SumDuration],

SUBSTRING(TextData, 1, 30) AS [Text Data]

FROM ::fn\_trace\_gettable('F:\MyTrace.trc',default)

WHERE EventClass in (10,12) -- BatchCompleted, RPC Completed

GROUP BY SUBSTRING(TextData, 1, 30)

ORDER BY SUM(Duration) DESC

**Take Note of Small Queries Called Often**

Often, small queries that run fairly quickly but that are called often are overlooked. Use SQL Profiler to identify queries that are called often, and then try to optimize them. Optimizing one query that runs hundreds of times may be more effective than optimizing one long-running query that runs only once.

You can modify the **fn\_trace\_gettable** sample code to order by SUM(CPU) so that it becomes an effective tool for identifying small queries or stored procedures that may be called thousands of times in an hour. When you tally their CPU costs, these queries can represent a huge expense to the overall performance of your SQL Server. By correctly designing indexes and avoiding bookmark lookups, you can shorten each call by milliseconds. Over time, this can amount to a big saving.

This kind of reporting is also useful for arranging small queries by reads and writes. Before you upgrade your subsystem, consider using these reporting methods to identify small queries. You can then plan application design changes that may help consolidate these small queries into larger batches.

**Use Sp\_lock and Sp\_who2 to Evaluate Locking and Blocking**

Use the **sp\_ lock** and **sp\_who2** system stored procedures to find out which locks are acquired by your query. You should use the least restrictive locks possible. Investigate queries that result in table locks. These table locks block other users from accessing the entire table and can severely limit scalability.

**Evaluate Waittype and Waittime in master..sysprocesses**

OLTP servers may report degradation of simple insert operations, update operations, and delete operations over time. The increase in average duration might occur due to a **sysprocesses.waittype** value of 0x0081. This is the log writer, and this **waittype** value means there is a delay in execution while your system process ID (SPID) waits on the two-phase commit process to the transaction log. You can measure this delay by capturing the **sysprocesses.waittime** value. This value may indicate that your transaction log is on the same spindle set (volume) as your data. It may also indicate that you do not have an adequate I/O subsystem where the log file exists, or that you have an inappropriately configured IO subsystem where the log file exists.

Your database administrator should also pay close attention to common locked resources. These can indicate a specific problem in a particular table. Specific **Waittype** values can be an early indication that your server is underpowered or overusing the disk, the CPU, or the transaction log. You can find out if your server is underpowered or overusing resources by taking snapshots of the **sysprocesses** and **syslockinfo** tables approximately every five seconds and by measuring how long a SPID waited.

**More Information**

For more information about **sp\_lock** , see "sp\_lock" in the "Transact-SQL Reference" on MSDN at <http://msdn.microsoft.com/en-us/library/aa238824(SQL.80).aspx>.

For more information about how to analyze blocking, see the following Knowledge Base articles:

* 2244453, "INF: Understanding and Resolving SQL Server 7.0 or 2000 Blocking Problems," at <http://support.microsoft.com/default.aspx?scid=kb;EN-US;224453>.
* 324885, "Support WebCast: Microsoft SQL Server: Rapid Blocker Script Analysis," at <http://support.microsoft.com/default.aspx?scid=kb;en-us;324885&Product=sql2k>.

**Use DBCC OPENTRAN to Locate Long-Running Transactions**

Run the DBCC OPENTRAN command to discover transactions that are open for long periods of time. This command has to be run repeatedly over the discovery period because transactions come and go. However, transactions that are reported continuously by this command may be running too long or are not committing at all. You should investigate these transactions.

**Testing**

The performance of a query that is not tuned can vary dramatically depending on the size of the data. A query that takes less than a second in a small database can take minutes in a database that has millions of rows. If your production database is large, populate your test database with an equivalent amount of data during development and testing. This gives you an opportunity to test the performance of your application with realistic data and to find the queries that need to be optimized. Ensure that you check your query execution plans by using tables that contain a realistic amount and distribution of data.

When you populate tables with large amounts of test data, follow these guidelines:

* Ensure that your transaction logs do not fill up. Using a simple loop mechanism can fill your transaction log for every single insert.
* Budget your database growth.
* Use tools to populate data.

The SQL Server resource kit provides valuable tools for generating test data in your database such as Database Hammer and Database Generator. For more information, see Chapter 39, "Tools, Samples, eBooks, and More," on the SQL Server Resource CD. This content is also available online at <http://technet.microsoft.com/en-us/sqlserver/bb972143.aspx>.

* Use existing production data.

If your application is going to be used against an existing database, consider making copies of the production data in your development, testing, and staging environments. If your production database contains sensitive data such as salary information, student grades, or other sensitive data, make sure that you strip it out or randomize it.

* Use common user scenarios with a balance between read and write operations.
* Use testing tools to perform stress and load tests on the system.

**More Information**

For more information about performance testing, see Chapter 16, "[Testing .NET Application Performance](http://msdn.microsoft.com/en-us/library/ff647788.aspx)"

**Monitoring**

You can think of a database system as a living and growing thing that you must continuously monitor and tune. Tuning is not an event; it is an ongoing process. Metrics, counters, and performance should be proactively reviewed on a regular basis. The following guidelines help you to maintain the performance and scalability of your application as your database ages:

* **Keep statistics up to date**.
* **Use SQL Profiler to tune long-running queries**.
* **Use SQL Profiler to monitor table and index scans**.
* **Use Performance Monitor to monitor high resource usage**.
* **Set up an operations and development feedback loop**.

**Keep Statistics Up to Date**

SQL Server uses a cost-based optimizer that is sensitive to statistical information provided on tables and indexes, such as the number of rows in a table and the average key length. Without correct and up-to-date statistical information, SQL Server may end up with a less optimal execution plan for a particular query.

Statistics that are maintained on each table in SQL Server to aid the optimizer in cost-based decision making include the number of rows, the number of pages used by the table, and the number of modifications made to the keys of the table since the last statistics update. In addition to maintaining statistics on indexed columns, it is possible to maintain statistics on columns that are not indexed.

Out-of-date or missing statistics are indicated by warnings when the execution plan of a query is graphically displayed in SQL Query Analyzer. The table name is displayed in red text. Monitor the **Missing Column Statistics** event class by using SQL Profiler so that you know when statistics are missing. To turn on the **Update** statistics option for a database, right-click the database in SQL Server Enterprise Manager, and then click **Properties**. Click the **Option** tab, and then select the **Auto update statistics** check box. In addition, you can run the **sp\_updatestats** system stored procedure from SQL Query Analyzer in the database to update the statistics for that database.

Use the UPDATE STATISTICS command or the **sp\_updatestats** system stored procedure to manually update statistics after large changes in data, or on a daily basis if there is a daily window available.

**More Information**

For more information, see Knowledge Base article 195565, "INF: How SQL Server 7.0 and SQL Server 2000 Autostats Work," at <http://support.microsoft.com/default.aspx?scid=kb;en-us;195565>.

**Use SQL Profiler to Tune Long-Running Queries**

Periodically use the SQL Profiler as described earlier to continuously tune long-running queries. As statistics and usage change, the queries that appear as the longest queries will change.

**Use SQL Profiler to Monitor Table and Index Scans**

Periodically use the SQL Profiler to continuously search for table and index scans. As statistics and usage change, the table and index scans that appear will change.

**Use Performance Monitor to Monitor High Resource Usage**

Periodically use the Performance Monitor to identify areas of high resource usage, and then investigate.

**Set Up an Operations and Development Feedback Loop**

Implement regular communications between production and operations personnel and the development group. Ensure all parties are exchanging information related to performance and scalability or development changes that might affect performance and scalability.

**More Information**

For more information about Performance Monitor or System Monitor and SQL Server, see "Monitoring Server Performance and Activity" at <http://msdn.microsoft.com/en-us/library/aa173860(SQL.80).aspx>.

**Deployment Considerations**

Physical deployment is an important factor for performance and scalability. However, a common mistake in performance and scalability tuning is to initially focus on scaling up or scaling out the hardware. Although a computer with limited resources does hinder performance, you can make the largest gains by minimizing resource usage as described earlier in this chapter. You should only consider adding hardware after you limit CPU use, network I/O, disk I/O, and memory use.

Physical configuration requirements are very specific for different scenarios, so they are not covered in depth in this section. However, the following are some important guidelines you should keep in mind:

* **Use default server configuration settings for most applications**.
* **Locate logs and the tempdb database on separate devices from the data**.
* **Provide separate devices for heavily accessed tables and indexes**.
* **Use the appropriate RAID configuration**.
* **Use multiple disk controllers**.
* **Pre-grow databases and logs to avoid automatic growth and fragmentation performance impact**.
* **Maximize available memory**.
* **Manage index fragmentation**.
* **Keep database administrator tasks in mind**.

**Use Default Server Configuration Settings for Most Applications**

SQL Server uses optimal configuration settings when it is newly installed. Changing the configuration settings may actually decrease performance except in certain high load situations. Thoroughly test any configuration change before making it to ensure that the change really improves performance or scalability. One exception is the memory setting, which is discussed later in this section.

To find out if your server settings comply with common best practices, you can download the Microsoft SQL Server 2000 Best Practices Analyzer tool at [Content link no longer available, original URL:"http://www.microsoft.com/downloads/details.aspx?FamilyId=B352EB1F-D3CA-44EE-893E-9E07339C1F22&displaylang=en"] .

**More Information**

For more information about SQL Server configuration settings, see Knowledge Base article 319942, "HOW TO: Determine Proper SQL Server Configuration Settings," at <http://support.microsoft.com/default.aspx?scid=kb;en-us;319942>.

**Locate Logs and the Tempdb Database on Separate Devices from the Data**

You can improve performance by locating your database logs and the tempdb database on physical disk arrays or devices that are separate from the main data device. Because data modifications are written to the log and to the database, and to the tempdb database if temp tables are used, having three different locations on different disk controllers provides significant benefits.

**Provide Separate Devices for Heavily Accessed Tables and Indexes**

If you have an I/O bottleneck on specific tables or indexes, try putting the tables or indexes in their own file group on a separate physical disk array or device to alleviate the performance bottleneck.

**Use the Appropriate RAID Configuration**

For a database server, you should choose hardware-level RAID rather than software RAID. Software RAID is usually cheaper but uses CPU cycles. If CPU utilization is a bottleneck for you, SQL Server may not perform optimally.

Two core RAID levels are of value for a database server:

* Striping with parity (RAID 5)
* Striped mirror (RAID 0+1)

When you choose a RAID level, you have to consider your cost, performance, and availability requirements. RAID 5 is less expensive than RAID 0+1, and RAID 5 performs better for read operations than write operations. RAID 0+1 is more expensive and performs better for write-intensive operations and for accessing the tempdb database.

**Use Multiple Disk Controllers**

A disk controller has a limit on its throughput. Associating too many disks with a single disk controller can lead to I/O bottlenecks.

**More Information**

For more information about how to determine the number of disks per disk controller, see "Microsoft SQL Server 7.0 Performance Tuning Guide" on MSDN at [http://msdn.microsoft.com/library/en-us/dnsql7/html/msdn\_sql7perftune.asp?frame=true#sql7perftune\_diskperform](http://msdn.microsoft.com/en-us/library/aa226166(SQL.70).aspx).

**Pre-Grow Databases and Logs to Avoid Automatic Growth and Fragmentation Performance Impact**

If you have enabled automatic growth, ensure that you are using the proper automatic growth option. You can grow database size by percent or by fixed size. Avoid frequent changes to the database sizes. If you are importing large amounts of data that tend to be of a fixed size on a weekly basis, grow the database by a fixed size to accommodate the new data.

**Maximize Available Memory**

Increasing memory is one of the best ways to improve SQL Server performance because more data can be cached in memory. Enable Address Windowing Extensions (AWE) for higher RAM utilization by SQL Server. Enable the /3GB switch in the Boot.ini file to allow a process to make use of 3 GB of virtual memory. By default, the system uses 2 GB. The operating system limits memory use by a process to 2 GB.

Use performance counters to decide the amount of memory that you need. Some performance counters that you can use to measure your need for memory are listed below:

* The **SQLServer:Buffer Manager**:**Buffer cache hit ratio** counter indicates that data is retrieved from memory cache. The number should be around 90. A lower value indicates that SQL Server requires more memory.
* The **Memory:Available Bytes** counter shows the amount of RAM that is available. Low memory availability is a problem if the counter shows that 10 megabytes (MB) of memory or less is available.
* The **SQLServer:Buffer Manager**: **Free pages** counter should not have a sustained value of 4 or less for more than two seconds. When there are no free pages in the buffer pool, the memory requirements of your SQL Server may have become so intense that the lazy writer or the check pointing process is unable to keep up. Typical signs of buffer pool pressure are a higher than normal number of lazy writes per second or a higher number of checkpoint pages per second as SQL Server attempts to empty the procedure and the data cache to get enough free memory to service the incoming query plan executions. This is an effective detection mechanism that indicates that your procedure or data cache is starved for memory. Either increase the RAM that is allocated to SQL Server, or locate the large number of hashes or sorts that may be occurring.

The memory configuration option is the one server configuration setting that you should evaluate and possibly change if there are processes running on the server other than SQL Server. If so, change the memory option to **Fixed**, and leave enough memory for the operating system and for the other processes that might be running.

**More Information**

For more information about SQL Server memory requirements, see "Inside SQL Server 2000's Memory Management Facilities" at <http://msdn.microsoft.com/en-us/library/aa175282(SQL.80).aspx>.

Also, see Knowledge Base article 274750, "HOW TO: Configure memory for more than 2 GB in SQL Server," at <http://support.microsoft.com/default.aspx?scid=kb;en-us;274750>.

**Manage Index Fragmentation**

As data is modified in a system, pages can split, and data can become fragmented or physically scattered on the hard disk. Use the DBCC SHOWCONTIG command to see the density and the degree of fragmentation for an index for a table.

There are several ways to resolve index fragmentation.

* Drop and recreate the index.
* Use the DBCC DBREINDEX command.
* Use the DBCC INDEXDEFRAG command.

The first two ways hold locks against the system. Therefore, you should only drop and then recreate an index or use the DBCC DBREINDEX command when there are no users on the system.

You can use DBCC INDEXDEFRAG when your system is online because it does not lock resources.

**More Information**

For more information about the DBCC SHOWCONTIG, DBCC DBREINDEX, and DBCC INDEXDEFRAG commands, see the following "Transact-SQL Reference" topics:

* "DBCC SHOWCONTIG" at <http://msdn.microsoft.com/en-us/library/aa258803(SQL.80).aspx>.
* "DBCC DBREINDEX" at <http://msdn.microsoft.com/en-us/library/aa258828(SQL.80).aspx>.
* "DBCC INDEXDEFRAG" at <http://msdn.microsoft.com/en-us/library/aa258286(SQL.80).aspx>.

**Keep Database Administrator Tasks in Mind**

Do not forget to take database administrator tasks into account when you think about performance. For example, consider the impact that database backups, statistic updates, DBCC checks, and index rebuilds have on your systems. Include these operations in your testing and performance analysis.

**Summary**

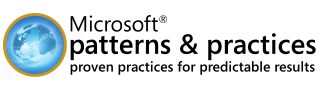
There are many issues that affect SQL Server performance and scalability. This chapter has taken a layered, top-down approach from data design to deployment. This chapter emphasizes the techniques you can use to obtain the biggest gains in performance and scalability.

Remember that you usually can obtain the biggest gains by creating efficient data access code for the application and by using the correct general development techniques. You usually do not obtain such significant gains in performance and scalability by changing SQL Server configuration settings.

**Additional Resources**

For related reading, see the following resources:

* For a printable checklist, see the "[Checklist: SQL Server Performance](http://msdn.microsoft.com/en-us/library/ff647681.aspx)" checklist in the "Checklists" section of this guide.
* Chapter 4, "[Architecture and Design Review of a .NET Application for Performance and Scalability](http://msdn.microsoft.com/en-us/library/ff647792.aspx)"
* Chapter 12, "[Improving ADO.NET Performance](http://msdn.microsoft.com/en-us/library/ff647768.aspx)"
* Chapter 15, "[Measuring .NET Application Performance](http://msdn.microsoft.com/en-us/library/ff647791.aspx)" See the "ADO.NET/Data Access" section.
* Chapter 16, "[Testing .NET Application Performance](http://msdn.microsoft.com/en-us/library/ff647788.aspx)"
* Chapter 17, "[Tuning .NET Application Performance](http://msdn.microsoft.com/en-us/library/ff647813.aspx)" See the "ADO.NET" and "SQL Server" sections.
* "[How To: Optimize SQL Indexes](http://msdn.microsoft.com/en-us/library/ff650692.aspx)" in the "How To" section of this guide.
* "[How To: Optimize SQL Queries](http://msdn.microsoft.com/en-us/library/ff650689.aspx)" in the "How To" section of this guide.
* "[How To: Use SQL Profiler](http://msdn.microsoft.com/en-us/library/ff650699.aspx)" in the "How To" section of this guide.
* *Microsoft SQL Server 2000 Performance Optimization and Tuning Handbook* by Ken England.
* *Handbook of Relational Database Design* by Candace Fleming and Barbara von Halle. Addison-Wesley Publishing Company.
* *Inside Microsoft SQL Server 2000* by Karen Delaney. Microsoft Press®, November 2000.
* "Microsoft SQL Server Performance Tuning and Optimization" at <http://www.sql-server-performance.com/>.

[](http://www.microsoft.com/practices)

|  |
| --- |
| **Retired Content** |
| This content is outdated and is no longer being maintained. It is provided as a courtesy for individuals who are still using these technologies. This page may contain URLs that were valid when originally published, but now link to sites or pages that no longer exist. |

[© Microsoft Corporation. All rights reserved.](http://msdn.microsoft.com/en-us/library/ms369863.aspx)