Optimizing Database Performance

Optimizing Database Performance Overview

The goal of performance tuning is to minimize the response time for each query and to maximize the throughput of the entire database server by reducing network traffic, disk I/O, and CPU time. This goal is achieved through understanding application requirements, the logical and physical structure of the data, and tradeoffs between conflicting uses of the database, such as online transaction processing (OLTP) versus decision support.

Performance issues should be considered throughout the development cycle, not at the end when the system is implemented. Many performance issues that result in significant improvements are achieved by careful design from the outset. To most effectively optimize the performance of Microsoft® SQL Server™ 2000, you must identify the areas that will yield the largest performance increases over the widest variety of situations and focus analysis on those areas.

Although other system-level performance issues, such as memory, hardware, and so on, are certainly candidates for study, experience shows that the performance gain from these areas is often incremental. Generally, SQL Server automatically manages available hardware resources, reducing the need (and thus, the benefit) for extensive system-level manual tuning.

Topic	Description
Designing Federated Database Servers	Describes how to achieve high levels of performance, such as those required by large Web sites, by balancing the processing load across multiple servers.
Database Design	Describes how database design is the most effective way to improve overall performance. Database design includes the logical database schema (such as tables and constraints) and the physical attributes such as disk systems, object placement, and indexes.
Query_Tuning	Describes how the correct design of the queries used by an application can significantly improve performance.
Application Design	Describes how the correct design of the user application can significantly improve performance. Application design includes transaction boundaries,

	locking, and the use of batches.
Optimizing Utility and Tool Performance	Describes how some of the options available with the utilities and tools supplied with Microsoft SQL Server 2000 can highlight ways in which the performance of these tools can be improved, as well as the effect of running these tools and your application at the same time.
Optimizing Server Performance	Describes how settings in the operating system (Microsoft Windows NT®, Microsoft Windows® 95, Microsoft Windows 98 or Microsoft Windows 2000) and SQL Server can be changed to improve overall performance.

Optimizing Database Performance

Designing Federated Database Servers

To achieve the high levels of performance required by the largest Web sites, a multitier system typically balances the processing load for each tier across multiple servers. Microsoft® SQL Server™ 2000 shares the database processing load across a group of servers by horizontally partitioning the SQL Server data. These servers are managed independently, but cooperate to process the database requests from the applications; such a cooperative group of servers is called a federation.

A federated database tier can achieve extremely high levels of performance only if the application sends each SQL statement to the member server that has most of the data required by the statement. This is called collocating the SQL statement with the data required by the statement. Collocating SQL statements with the required data is not a requirement unique to federated servers.I It is also required in clustered systems.

Although a federation of servers presents the same image to the applications as a single database server, there are internal differences in how the database services tier is implemented.

Single server tier	Federated server tier
There is one instance of SQL Server on the production server.	There is one instance of SQL Server on each member server.
The production data is stored in one database.	Each member server has a member database. The data is spread through the member databases.
Each table is typically a single entity.	The tables from the original database are horizontally partitioned into member tables. There is one member table per member database, and distributed partitioned views are used to make it appear as if there was a full copy of the original table on each member server.
All connections are made to the	The application layer must be able to

single server, and all SQL statements are processed by the same instance of SQL Server.	collocate SQL statements on the member server containing most of the data referenced by the statement.

While the goal is to design a federation of database servers to handle a complete workload, you do this by designing a set of distributed partitioned views that spread the data across the different servers.

See Also

Federated SQL Server 2000 Servers

Creating a Partitioned View

Optimizing Database Performance

Designing Partitions

Partitioning works well if the tables in the database are naturally divisible into similar partitions where most of the rows accessed by any SQL statement can be placed on the same member server. Tables are clustered in related units. For example, suppose the entry of an order references the **Orders**, **Customers**, and **Parts** tables, along with all tables that record the relationships between customers, orders, and parts. Partitions work best if all the rows in a logical cluster can be placed on the same member server.

Symmetric Partitions

Partitioning is most effective if the tables in a database can be partitioned symmetrically:

- Related data is placed on the same member server, so that most SQL statements routed to the correct member server will have minimal, if any, requirements for data on other member servers. A distributed partitioned view design goal can be stated as an 80/20 rule: Design partitions so that most SQL statements can be routed to a member server, where at least 80 percent of the data is on that server, and distributed queries are needed for 20 percent or less of the data. A good test of whether this can be achieved is to see whether the partition allows all rows to be placed on the same member server as all of their referencing foreign key rows. Database designs that support this goal are good candidates for partitioning.

- The data is partitioned uniformly across the member servers.

  For example, suppose a company has divided North America into regions. Each employee works in one region, and customers make most of their purchases in the state or province where they live. The region and employee tables are partitioned along regions. Customers are partitioned between regions by their state or province. While some queries require data from multiple regions, the data needed for most queries is on the server for one region. Applications route SQL

statements to the member server containing the region inferred from the context of the user input.

Asymmetric Partitions

Although symmetric partitions are the ideal goal, most applications have complex data access patterns that prevent symmetrical partitioning. Asymmetric partitions result in some member servers assuming larger roles than others. For example, only some of the tables in a database may be partitioned, with the tables that have not been partitioned remaining on the original server. Asymmetric partitions can provide much of the performance of a symmetric partition, with these important benefits:

- Dramatically improving the performance of a database that cannot be symmetrically partitioned by asymmetrically partitioning some of its tables.

- Successfully partitioning a large existing system by making a series of iterative, asymmetric improvements. The tables chosen for partitioning in each step are usually those that will give the highest performance gain at that time.

In an asymmetric approach, the original server usually retains some tables that did not fit the partitioning scheme. The performance of these remaining tables is usually faster than in the original system because the member tables move to member servers, reducing the load on the original server.

Many databases can be partitioned in more than one way. The specific partitions chosen for implementation must be those that best meet the requirements of the typical range of SQL statements executed by the business services tier.

Distributed Partitioned Views

You should also design the partitioning in a way that produces routing rules that applications can use to determine which member server can most effectively process each SQL statement. The business services tier must be able to match a piece of user data against the routing rules to find which member server processes the SQL statement.

There are four areas to consider when designing a set of distributed partitioned views to implement a federation of database servers:

- Determine the pattern of SQL statements executed by the application.

  Develop a list of the SQL statements executed by the application during typical processing periods. Divide the list into SELECT, UPDATE, INSERT, and DELETE categories, and order the list in each category by frequency of execution. If the SQL statements reference stored procedures, use the base SELECT, INSERT, UPDATE, and DELETE statements from the stored procedure. If you are partitioning an existing Microsoft® SQL Server™ 2000 database, you can use SQL Profiler to get such a list.

  The recommendation for using the frequency of SQL statements is a reasonable approximation in the typical online transaction processing (OLTP) or Web site database in which distributed partitioned views work best. These systems are characterized by having individual SQL statements that retrieve relatively small amounts of data when compared to the types of queries in a decision support, or OLAP, system. When each SQL statement references a small amount of data, simply studying the frequency of each statement yields a reasonable approximation of the data traffic in the system. Many systems, however, have some group of SQL statements that reference large amounts of data. You may want to take the extra step of weighting these queries to reflect their larger data requirements.

- Determine how the tables are related to each other.

  The intent is to find clusters of tables that can be partitioned along the same dimension (for example, part number or department number) so that all the rows related to individual occurrences of that dimension will end up on the same member server. For example, you may determine that one way to partition your database is by region. To support this, even tables that do not have a region number in their key must be capable of being partitioned in some manner related to a region. In such a database, even when the **Customer** table does not have a region number column, if regions are defined as collections of whole states or provinces, then the **Customer.StateProvince** column can be used to

partition the customers in a manner related to region.

Because they define the relationships between tables, explicit and implicit foreign keys are the prime elements to review in looking for ways to partition data. Study the explicit foreign key definitions to determine how queries would usually use rows in one table to find rows in another table. Also study implicit foreign keys, or ways that SQL statements use values in the rows of one table to reference rows from another table in join operations, even when there is no specific foreign key definition. Because implicit foreign keys are not explicitly defined as part of the database schema, you must review the SQL statements generated by the application to understand whether there are statements that join tables using nonkey columns. These implicit foreign keys are typically indexed to improve join performance, so you should also review the indexes defined in the database.

- Match the frequency of SQL statements against the partitions defined from analyzing the foreign keys.

  Select the partitioning that will best support the mix of SQL statements in your application. If some sets of tables can be partitioned in more than one way, use the frequency of SQL statements to determine which of the partitions satisfies the largest number of SQL statements. The tables most frequently referenced by SQL statements are the ones you want to partition first. Prioritize the sequence in which you partition the tables based on the frequency in which the tables are referenced.

  The pattern of SQL statements also influences the decision on whether a table should be partitioned:

  - Partition a table if more than 5 percent of the statements referencing a table are INSERT, UDATE, or DELETE statements, and the table can be partitioned along the dimension you have chosen.

  - Maintain complete copies of tables on each member server if less than 5 percent of the statements referencing the table are INSERT, UPDATE, or DELETE statements. You will also need to define how updates will be made so that all the copies of the

table are updated. If high transactional integrity is required, you can code triggers that perform distributed updates of all the copies within the context of a distributed transaction. If you do not need high transactional integrity, you can use one of the SQL Server replication mechanisms to propagate updates from one copy of the table to all other copies.

    - Do not partition or copy a table if more than 5 percent of the statements referencing a table are INSERT, UDATE, or DELETE statements, and the table cannot be partitioned along the dimension you have chosen.

- Define the SQL statement routing rules. The routing rules must be able to define which member server can most effectively process each SQL statement. They must establish a relationship between the context of the input of the user and the member server that contains the bulk of the data required to complete the statement. The applications must be able to take a piece of data entered by the user, and match it against the routing rules to determine which member server should process the SQL statement.

See Also

Federated SQL Server 2000 Servers

Creating a Partitioned View

Designing Applications to Use Federated Database Servers

Optimizing Database Performance

Designing Federated Database Servers for High Availability

The data for a large Web site or internal online transaction processing (OLTP) system must be highly reliable. The data must be available 24 hours a day, 7 days a week, 52 weeks a year. In a clustered application tier, the loss of one server may degrade system performance, but it will not stop the entire system. The remaining servers in the cluster rebalance the load until a replacement server can be plugged into the cluster.

Although Microsoft® SQL Server™ 2000 does not support this type of load-balanced clustering, it does support Microsoft Cluster Services failover clustering. Failover clustering supports one to four servers per cluster depending on the operating system. The cluster appears to applications as a single virtual server. If the primary server node fails, another node detects the loss of the primary and automatically starts servicing all requests sent to the virtual server. The cluster remains running under the alternative node until the primary server is repaired or replaced. Failover clustering helps provide high availability, but it does not perform any load balancing.

See Also

Failover Clustering

Optimizing Database Performance

Backing Up and Restoring Federated Database Servers

In a federated-database-server tier, built using distributed partitioned views, the member servers form one logical unit; and you must coordinate the recovery of the member databases to ensure that they remain synchronized properly.

Microsoft® SQL Server™ 2000 does not require that you coordinate backups across member servers. Backups can be taken from each database independently, without regard for the state of the other member databases. Because the backups do not have to be synchronized, there is no processing overhead for synchronization and no blockage of running tasks.

The most important aspect of recovering a set of member databases is the same as recovering any other database: plan and test the recovery procedures before putting the databases into production. You must set up processes to restore all the databases to the same logical point in time. SQL Server includes features to support the recovery of all member databases to the same point in time.

See Also

Backing Up and Restoring Databases

Optimizing Database Performance

Database Design

There are two components to designing a database: logical and physical. Logical database design involves modeling your business requirements and data using database components, such as tables and constraints, without regard for how or where the data will be physically stored. Physical database design involves mapping the logical design onto physical media, taking advantage of the hardware and software features available, which allows the data to be physically accessed and maintained as quickly as possible, and indexing.

It is important to correctly design the database to model your business requirements, and to take advantage of hardware and software features early in the development cycle of a database application, because it is difficult to make changes to these components later.

Optimizing Database Performance

Logical Database Design

Using Microsoft® SQL Server™ 2000 effectively begins with normalized database design. Normalization is the process of removing redundancies from the data. For example, when you convert from an indexed sequence access method (ISAM) style application, normalization often involves breaking data in a single file into two or more logical tables in a relational database. Transact-SQL queries then recombine the table data by using relational join operations. By avoiding the need to update the same data in multiple places, normalization improves the efficiency of an application and reduces the opportunities for introducing errors due to inconsistent data.

However, there are tradeoffs to normalization. A database that is used primarily for decision support (as opposed to update-intensive transaction processing) may not have redundant updates and may be more understandable and efficient for queries if the design is not fully normalized. Nevertheless, data that is not normalized is a more common design problem in database applications than over-normalized data. Starting with a normalized design, and then selectively denormalizing tables for specific reasons, is a good strategy.

For more information, see Normalization.

Whatever the database design, you should take advantage of these features in SQL Server to automatically maintain the integrity of your data:

- CHECK constraints ensure that column values are valid.
- DEFAULT and NOT NULL constraints avoid the complexities (and opportunities for hidden application bugs) caused by missing column values.
- PRIMARY KEY and UNIQUE constraints enforce the uniqueness of rows (and implicitly create an index to do so).
- FOREIGN KEY constraints ensure that rows in dependent tables always have a matching master record.

- IDENTITY columns efficiently generate unique row identifiers.
- **timestamp** columns ensure efficient concurrency checking between multiple-user updates.
- User-defined data types ensure consistency of column definitions across the database.

By taking advantage of these features, you can make the data rules visible to all users of the database, rather than hiding them in application logic. These server-enforced rules help avoid errors in the data that can arise from incomplete enforcement of integrity rules by the application itself. Using these facilities also ensures that data integrity is enforced as efficiently as possible.

See Also

Data Integrity

Optimizing Database Performance

Database Design Considerations: Data Types

SQL Server 2000 treats any fixed-length column that allows null values as fixed-length. Therefore, a **char** column that allows null values is treated as a fixed-length **char** column. As a result, the same data takes more disk space to store and can require more I/O and other processing operations in SQL Server 2000 compared to earlier versions of SQL Server. To resolve this issue , use variable-length columns rather than fixed-length columns. For example, use a **varchar** data type instead of a **char** data type. However, if all the values in a column are the same length or the lengths of the values do not vary by much, it is more efficient to use a fixed-length column.

Text Data Types

Character strings up to 8,000 bytes in length can be stored in columns defined with the **char** and **varchar** data types. Using a **char** or **varchar** data type allows the system-defined character string functions, such as SUBSTRING, to be used on character strings up to 8,000 bytes in length.

See Also

Specifying a Column Data Type

Optimizing Database Performance

Physical Database Design

The I/O subsystem (storage engine) is a key component of any relational database. A successful database implementation usually requires careful planning at the early stages of your project. The storage engine of a relational database requires much of this planning, which includes determining:

- What type of disk hardware to use, such as RAID (redundant array of independent disks) devices. For more information, see RAID.

- How to place your data onto the disks. For more information, see Data Placement Using Filegroups.

- Which index design to use to improve query performance in accessing data. For more information, see Index Tuning Recommendations.

- How to set all configuration parameters appropriately for the database to perform well. For more information, see Optimizing Server Performance.

Optimizing Database Performance

RAID

RAID (redundant array of independent disks) is a disk system that comprises multiple disk drives (an array) to provide higher performance, reliability, storage capacity, and lower cost. Fault-tolerant arrays are categorized in six RAID levels, 0 through 5. Each level uses a different algorithm to implement fault tolerance.

Although RAID is not a part of Microsoft® SQL Server™ 2000, its implementation can directly affect the way SQL Server performs. RAID levels 0, 1, and 5 are typically used with SQL Server.

Note RAID is available only on Microsoft Windows NT 4.0 and Microsoft Windows 2000.

A hardware disk array improves I/O performance because I/O functions, such as striping and mirroring, are handled efficiently in firmware. Conversely, an operating system–based RAID offers lower cost but consumes processor cycles. When cost is a consideration and redundancy and high performance are required, Microsoft Windows® NT® stripe sets with parity or Windows 2000 RAID-5 volumes are a good solution.

Data striping (RAID 0) is the RAID configuration with the highest performance, but if one disk fails, all the data on the stripe set becomes inaccessible. A common installation technique for relational database management systems is to configure the database on a RAID 0 drive and then place the transaction log on a mirrored drive (RAID 1). You can get the best disk I/O performance for the database and maintain data recoverability (assuming you perform regular database backups) through a mirrored transaction log.

If data must be quickly recoverable, consider mirroring the transaction log and placing the database on a RAID 5 disk. RAID 5 provides redundancy of all data on the array, allowing a single disk to fail and be replaced in most cases without system downtime. RAID 5 offers lower performance than RAID 0 or RAID 1 but higher reliability and faster recovery.

Developing a Drive Performance Strategy

By managing the placement of data on drives, you can both improve performance and implement fault tolerance. In the context of managing drive storage for a Microsoft® SQL Server™ 2000 installation, performance refers in part to the speed of read and write operations, and fault tolerance refers to the ability of the system to continue functioning without data loss when part of the system fails.

You can use the following methods to manage the placement of data on disk drives:

- Hardware-based RAID (redundant array of independent disks) above level 0 can protect against data loss in the event of media failure, and can improve performance. For more information, see the documentation provided by the vendor.

- Both Microsoft Windows NT® and Microsoft Windows® 2000-based disk striping, and striping with parity, can improve performance. Disk striping with parity also protects against data loss in the event of media failure.

- Windows NT and Windows 2000-based disk mirroring and duplexing are both fault-tolerance mechanisms that protect against data loss in the event of media failure. They can also improve read performance.

IMPORTANT These fault-tolerance methods do not replace proper backup strategies. You must perform periodic backups to protect your databases and data against catastrophic loss.

For more information about Windows NT and Windows 2000 disk striping, mirroring, and duplexing, see the Windows NT or Windows 2000 documentation.

See Also

Backing Up and Restoring Databases

RAID Levels and SQL Server

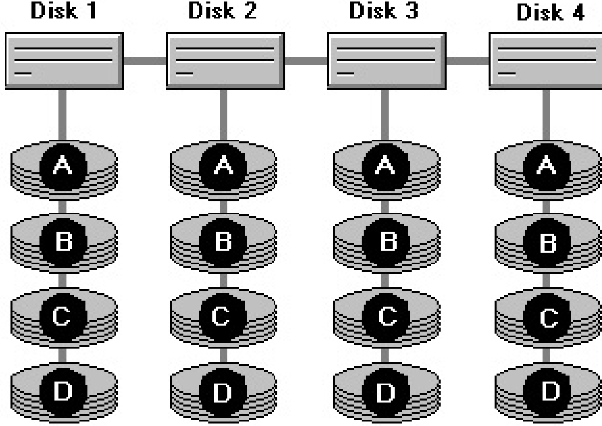
RAID (redundant array of independent disks) levels 0, 1, and 5 are typically implemented with Microsoft® SQL Server™ 2000.

Note RAID levels greater than 10 (1 + 0) offer additional fault tolerance or performance enhancements. These tend to be proprietary systems. For more information about these types of RAID systems, contact the hardware vendor.

Level 0

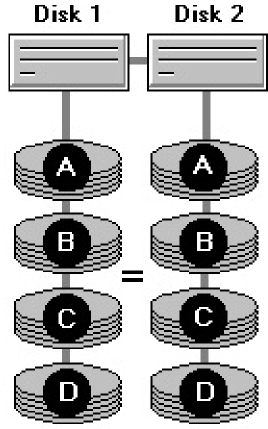
This level is also known as disk striping because of its use of a disk file system called a stripe set. Data is divided into blocks and spread in a fixed order among all disks in an array. RAID 0 improves read/write performance by spreading operations across multiple disks, so that operations can be performed independently and simultaneously.

RAID 0 is similar to RAID 5, but RAID 5 also provides fault tolerance.



Level 1

This level is also known as disk mirroring because of its use of a disk file system called a mirror set. Disk mirroring provides a redundant, identical copy of a selected disk. All data written to the primary disk is written to the mirror disk. RAID 1 provides fault tolerance and generally improves read performance (but may degrade write performance).



Level 2

This level adds redundancy by using an error correction method that spreads parity across all disks. It also employs a disk-striping strategy that breaks a file into bytes and spreads it across multiple disks. This strategy offers only a marginal improvement in disk utilization and read/write performance over mirroring (RAID 1). RAID 2 is not as efficient as other RAID levels and is not generally used.

Level 3

This level uses the same striping method as RAID 2, but the error correction method requires only one disk for parity data. Use of disk space varies with the number of data disks. RAID 3 provides some read/write performance improvement.

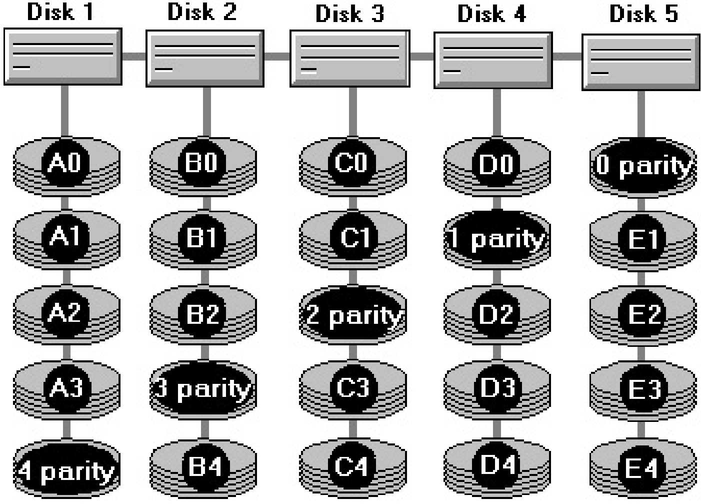
Level 4

This level employs striped data in much larger blocks or segments than RAID 2 or RAID 3. Like RAID 3, the error correction method requires only one disk for parity data. It keeps user data separate from error-correction data. RAID 4 is not as efficient as other RAID levels and is not generally used.

Level 5

Also known as striping with parity, this level is the most popular strategy for new designs. It is similar to RAID 4 in that it stripes the data in large blocks across the disks in an array. It differs in that it writes the parity across all the disks. Data redundancy is provided by the parity information. The data and

parity information are arranged on the disk array so that the two are always on different disks. Striping with parity offers better performance than disk mirroring (RAID 1). However, when a stripe member is missing, read performance degrades (for example, when a disk fails).



Level 10 (1+0)

This level is also known as mirroring with striping. This level uses a striped array of disks, which are then mirrored to another identical set of striped disks. For example, a striped array can be created using five disks. The striped array of disks is then mirrored using another set of five striped disks. RAID 10 provides the performance benefits of disk striping with the disk redundancy of mirroring. RAID 10 provides the highest read/write performance of any of the RAID levels at the expense of using twice as many disks.

Comparing Different Implementations of RAID Levels

There are advantages and disadvantages to using the various implementations of RAID (redundant array of independent disks).

RAID implementation	Advantage	Disadvantage
Microsoft® WindowsNT®-based striping or Windows 2000 RAID-5 volumes	No added hardware cost.	Uses system processing resources.
Hardware-based striping	Does not compete for processor cycles. Best performance of all RAID implementations.	Additional cost of specialized hardware.

RAID solutions typically used with Microsoft SQL Server™ 2000 provide varying levels of redundancy and fault tolerance.

RAID implementation	Advantage	Disadvantage
Hardware-based RAID 3, 5, or 10	Excellent performance. Does not compete for processor cycles.	Cost.
Hardware-based RAID 1	Excellent redundancy. Does not compete for processor cycles.	Additional cost due to more hardware.
Hardware-based RAID 10	Excellent performance. Excellent redundancy.	Additional cost due to more hardware.
Windows NT–based RAID 1 or Windows 2000 mirrored volumes	Good redundancy. Low cost.	Uses system processing resources.
Windows NT– or Windows 2000-based RAID 5	Excellent read performance. Low cost.	Uses system processing resources.

About Hardware-Based Solutions

RAID (redundant array of independent disks) levels 0, 1, 3, and 5 are the levels typically implemented in hardware-based solutions.

Hardware-based RAID uses an intelligent drive controller and a redundant array of disk drives to protect against data loss in the event of media failure and to improve the performance of read/write operations. A disk array is an effective disk-storage solution for computers running Microsoft® SQL Server™2000.

Hardware-based RAID levels 1 through 5 automate redundancy and fault tolerance at the hardware level. All levels (0 through 5) incur no overhead on the system processor. Individual data files are typically spread across more than one disk. It is possible to implement a hardware-based RAID solution that provides your system with seamless, nonstop recovery from media failure.

In general, hardware-based RAID offers performance advantages over Microsoft Windows NT® or Windows® 2000 software-based RAID. For example, you can improve data throughput significantly by implementing RAID 5 through hardware that does not use system software resources. This is accomplished by using more disks at a given capacity than in conventional storage solution. Read/write performance and total storage size can be further improved by using multiple controllers.

Depending on the configuration, hardware-based RAID generally provides good performance. It also makes it much easier to manage multiple disks, allowing you to treat an array of disks as one disk. You may even be able to replace a failed drive without shutting down the system. The disadvantages of a hardware-based solution are cost, and it may lock you into a single vendor.

For more information about implementing hardware-based RAID, contact the hardware vendor.

About Windows NT– and Windows 2000-Based Disk Striping and Striping with Parity

Microsoft® Windows NT®–based disk striping and striping with parity and Windows® 2000 RAID-5 volumes implement RAID features in software, using any hardware compatible with the operating system. Because these are software-based solutions provided with the operating system, they offer a cost advantage.

- Disk striping writes data in stripes across a volume (created from areas of free space). For more information about volumes, see the Windows NT or Windows 2000 documentation.

  These areas are all the same size and are spread over an array of disks (up to 32 disks). Striping writes files across all disks, so data is added to all partitions in the set at the same rate.

  Windows NT-based disk striping and Windows 2000 volume sets implement RAID 0. Disk striping provides the best performance of all Windows NT Server disk-management strategies, but does not provide any fault-tolerance protection.

- Disk striping with parity is similar to disk striping. Disk striping with parity adds a parity-information stripe to each disk partition in the volume. This provides fault-tolerance protection equivalent to that of disk mirroring, but requires much less space for the redundant data. Windows NT-based disk striping with parity and Windows 2000 RAID-5 volumes implement RAID 5.

When a member of a stripe set with parity or RAID-5 volume fails in a severe manner (for example, from a loss of power or a complete head crash), you can regenerate the data for that member of the stripe set from the remaining members.

Stripe sets with parity and RAID-5 volumes are a good solution for data redundancy in a computing environment in which most activity consists of reading data. Disk stripe sets with parity and RAID-5 volumes also improve write performance, but not as much as striping alone. Creating a disk stripe set

with parity or a RAID-5 volume requires at least three physical disks on the server.

Disk striping is available on Windows NT Server, Windows NT Workstation and Windows 2000. However, disk striping with parity is supported only for Windows NT Server and Windows 2000. On a dual-boot computer, stripe sets, including those with parity, are not accessible when running the Microsoft MS-DOS® operating system.

Disk striping with parity or Windows 2000 RAID-5 volumes are recommended over mirroring for applications that require redundancy and are read-oriented, although disk striping with parity and RAID-5 volumes require more system memory.

Disk striping and disk striping with parity are set up and managed using the Windows NT Disk Administrator application, which can be started from the **Administrative Tools** program group. RAID-5 volumes are set up and managed using the Windows 2000 Disk Management application, which can be started from the Computer Management program.

For more information about setting up disk striping or disk striping with parity, see the Windows NT Server or Windows 2000 documentation.

About Windows NT- and Windows 2000-Based Disk Mirroring and Duplexing

Microsoft® Windows NT® and Windows® 2000-based disk mirroring and duplexing implement RAID (redundant array of independent disks) features in software using any hardware compatible with the operating system. Because these are software-based solutions provided with the operating system, they offer a cost advantage.

- Disk mirroring protects against media failure by maintaining a fully redundant copy of a partition on another disk. This provides protection from the downtime and expense involved in recovering lost data and restoring data from a backup storage facility. In a sense, mirroring is continual backup. Mirroring also provides some performance benefits when reading data from disks under heavy I/O loads. Windows NT–based disk mirroring and Windows 2000 mirrored volume implement RAID 1.

- Disk duplexing is a form of mirroring that provides protection against controller failures (in addition to protecting against media failures) by using a different disk controller on the mirror disk.

Disk mirroring and duplexing are features of Windows NT Server. They are not supported for Windows NT Workstation. Mirrored volume is a feature of Windows 2000. On a dual-boot computer, they are not accessible when running the Microsoft MS-DOS® operating system.

Windows NT–based disk mirroring, or duplexing, and Windows 2000 mirrored volumes offer better write performance than Windows NT–based disk striping with parity and Windows 2000 RAID-5 volumes. They also require less system memory and do not show performance degradation during a failure.

The entry cost of Windows NT and Windows 2000-based disk mirroring or duplexing is lower because they require only two or more disks (compared to disk striping with parity and RAID-5 volume, which require three or more

disks). However, mirroring provides less usable disk space (compared to disk striping with parity or RAID-5 volume), so the cost per megabyte is higher.

Disk mirroring and duplexing are implemented by using the Windows NT Disk Administrator application, which can be started from the **Administrative Tools** program group. Mirrored volumes are set up and managed using the Windows 2000 Disk Management application, which can be started from the Computer Management program.

For more information about setting up disk mirroring or duplexing, see the Windows NT Server documentation.

Note The term mirroring is frequently used in Windows NT Server documentation to describe both disk mirroring and duplexing.

Optimizing Database Performance

Partitioning

Partitioning a database improves performance and simplifies maintenance. By splitting a large table into smaller, individual tables, queries accessing only a fraction of the data can run faster because there is less data to scan. Maintenance tasks, such as rebuilding indexes or backing up a table, can execute more quickly.

Partitioning can be achieved without splitting tables by physically placing them on individual disk drives. Placing a table on one physical drive and related tables on a separate drive, for example, can improve query performance because when queries involving joins between the tables are executed, multiple disk heads read data at the same time. Microsoft® SQL Server™ 2000 filegroups can be used to specify on which disks to place the tables.

Hardware Partitioning

Hardware partitioning designs the database to take advantage of the available hardware architecture. Examples of hardware partitioning include:

- Multiprocessors that allow multiple threads of execution, permitting many queries to execute at the same time. Alternatively, a single query may be able to run faster on multiple processors by allowing components of the query to be executed simultaneously. For example, each table referenced in the query can be scanned at the same time by a different thread.

- RAID (redundant array of independent disks) devices that allow data to be striped across multiple disk drives, permitting faster access to the data because more read/write heads read data at the same time. A table striped across multiple drives can typically be scanned faster than the same table stored on one drive. Alternatively, storing tables on separate drives from related tables can significantly improve the performance of queries joining those tables.

Horizontal Partitioning

Horizontal partitioning segments a table into multiple tables, each containing the same number of columns but fewer rows. For example, a table containing 1 billion rows could be partitioned horizontally into 12 tables, with each smaller table representing one month of data for a given year. Any queries requiring a specific month's data reference the appropriate table only.

Determining how to partition the tables horizontally depends on how data is analyzed. Partition the tables so that queries reference as few tables as possible. Otherwise, excessive UNION queries, used to merge the tables logically at query time, can impair performance. For more information about querying horizontally partitioned tables, see Scenarios for Using Views.

Partitioning data horizontally based on age/use is common. For example, a table may contain data for the last five years, but only data from the current year is regularly accessed. In this case, you may consider partitioning the data into five tables, with each table containing data from only one year.

Vertical Partitioning

Vertical partitioning segments a table into multiple tables containing fewer columns. The two types of vertical partitioning are normalization and row splitting.

Normalization is the standard database process of removing redundant columns from a table and placing them in secondary tables linked to the primary table by primary key and foreign key relationships.

Row splitting divides the original table vertically into tables with fewer columns. Each logical row in a split table matches the same logical row in the others. For example, joining the tenth row from each split table re-creates the original row.

Like horizontal partitioning, vertical partitioning allows queries to scan less data, hence increasing query performance. For example, a table containing seven columns, of which only the first four are commonly referenced, may benefit from splitting the last three columns into a separate table.

Vertical partitioning should be considered carefully because analyzing data from multiple partitions requires queries joining the tables, possibly affecting performance if partitions are very large.

See Also

Using Views with Partitioned Data

Optimizing Database Performance

Data Placement Using Filegroups

Microsoft® SQL Server™ 2000 allows you to create tables or indexes on a specific filegroup within your database, rather than across all filegroups in a database. By creating a filegroup on a specific disk or RAID (redundant array of independent disks) device, you can control where tables and indexes in your database are physically located. Reasons for placing tables and indexes on specific disks include:

- Improved query performance.

- Parallel queries.

See Also

Files and Filegroups

Placing Tables on Filegroups

A table can be created on a specific filegroup rather than the default filegroup. If the filegroup comprises multiple files spread across various physical disks, each with its own disk controller, then queries for data from the table will be spread across the disks, thereby improving performance. The same effect can be accomplished by creating a single file on a RAID (redundant array of independent disks) level 0, 1, or 5 device.

If the computer has multiple processors, Microsoft® SQL Server™ 2000 can perform parallel scans of the data. Multiple parallel scans can be executed for a single table if the filegroup of the table contains multiple files. Whenever a table is accessed sequentially, a separate thread is created to read each file in parallel. For example, a full scan of a table created on a filegroup comprising of four files will use four separate threads to read the data in parallel. Therefore, creating more files per filegroup can help increase performance because a separate thread is used to scan each file in parallel. Similarly, when a query joins tables on different filegroups, each table can be read in parallel, thereby improving query performance.

Additionally, any **text**, **ntext**, or **image** columns within a table can be created on a filegroup other than the one that contains the base table.

Eventually, there is a saturation point when there are too many files and therefore too many parallel threads causing bottlenecks in the disk I/O subsystem. These bottlenecks can be identified by using Windows NT® Performance Monitor to monitor the **PhysicalDisk** object and **Disk Queue Length** counter. If the **Disk Queue Length** counter is greater than three, consider reducing the number of files. For more information, see Monitoring Disk Activity.

It is advantageous to get as much data spread across as many physical drives as possible in order to improve throughput through parallel data access using multiple files. To spread data evenly across all disks, first set up hardware-based disk striping, and then use filegroups to spread data across multiple hardware stripe sets if needed.

To create a new table on a specific filegroup

⊞ Transact-SQL

⊞ Enterprise Manager

⊞ SQL-DMO

To place an existing table on a different filegroup

Placing Indexes on Filegroups

By default, indexes are created on the same filegroup as the base table on which the index is created. However, it is possible to create nonclustered indexes on a filegroup other than the filegroup of the base table. By creating the index on a different filegroup, you can realize performance gains if the filegroups make use of different physical drives with their own controllers. Data and index information can then be read in parallel by multiple disk heads. For example if **Table_A** on filegroup **f1** and **Index_A** on filegroup **f2** are both being used by the same query, performance gains can be achieved because both filegroups are being fully used with no contention. However, if **Table_A** is scanned by the query but **Index_A** is not referenced, only filegroup **f1** is used, resulting in no performance gain.

Because you cannot predict what type of access will take place and when it will take place, it could be a safer decision to spread your tables and indexes across all filegroups. This would guarantee that all disks are being accessed since all data and indexes are spread evenly across all disks, no matter which way the data is accessed. This is also a simpler approach for system administrators.

If there is a clustered index on a table, the data and the clustered index always reside in the same filegroup. Therefore, you can move a table from one filegroup to another by creating a clustered index on the base table that specifies a different filegroup on which to create the index (the index can then be dropped, leaving the base table in the new filegroup).

If the indexes of a table span multiple filegroups, all filegroups containing the table and its indexes must be backed up together, after which a transaction log backup must be created. Otherwise, only some of the indexes may be backed up, preventing the index from being recovered if the backup is restored later. For more information, see Using File Backups.

Note An individual table or index can belong to only one filegroup; it cannot span filegroups.

To create a new index on a specific filegroup

⊞ Transact-SQL

⊞Enterprise Manager

⊞SQL-DMO

To place an existing index on a different filegroup

Optimizing Database Performance

Index Tuning Recommendations

Indexes can be dropped, added, and changed without affecting the database schema or application design. Efficient index design is paramount to achieving good performance. For these reasons, you should not hesitate to experiment with different indexes. The Index Tuning Wizard can be used to analyze your queries and suggest the indexes that should be created. For more information, see Index Tuning Wizard.

The query optimizer in Microsoft® SQL Server™ 2000 reliably chooses the most effective index in the majority of cases. The overall index design strategy should provide a good selection of indexes to the query optimizer and trust it to make the right decision. This reduces analysis time and results in good performance over a wide variety of situations.

Do not always equate index usage with good performance, and vice-versa. If using an index always produced the best performance, the job of the query optimizer would be simple. In reality, incorrect choice of indexed retrieval can result in less than optimal performance. Therefore, the task of the query optimizer is to select indexed retrieval only when it will improve performance and to avoid indexed retrieval when it will affect performance.

Recommendations for creating indexes include:

- Write queries that update as many rows as possible in a single statement, rather than using multiple queries to update the same rows. By using only one statement, optimized index maintenance can be exploited.

- Use the Index Tuning Wizard to analyze your queries and make index recommendations. For more information, see Index Tuning Wizard.

- Use integer keys for clustered indexes. Additionally, clustered indexes benefit from being created on unique, nonnull, or IDENTITY columns. For more information, see Using Clustered Indexes.

- Create nonclustered indexes on all columns frequently used in queries. This can maximize the use of covered queries. For more information, see Using Nonclustered Indexes.

- The time taken to physically create an index is largely dependent on the disk subsystem. Important factors to consider are:
    - RAID (redundant array of independent disks) level used to store the database and transaction log files.

    - Number of disks in the disk array (if RAID was used).

    - Size of each data row and the number of rows per page. This determines the number of data pages that must be read from disk to create the index.

    - The columns in the index and the data types used. This determines the number of index pages that have to be written to disk.
- Examine column uniqueness. For more information, see Using Unique Indexes.

- Examine data distribution in indexed columns. Often, a long-running query is caused by indexing a column with few unique values, or by performing a join on such a column. This is a fundamental problem with the data and query, and usually cannot be resolved without identifying this situation. For example, a physical telephone directory sorted alphabetically on last name will not expedite locating a person if all people in the city are named Smith or Jones.

See Also

Statistical Information

Optimizing Database Performance

Optimizing Transaction Log Performance

General recommendations for creating transaction log files include:

- Create the transaction log on a physically separate disk or RAID (redundant array of independent disks) device. The transaction log file is written serially; therefore, using a separate, dedicated disk allows the disk heads to stay in place for the next write operation.

- Set the original size of the transaction log file to a reasonable size to prevent the file from automatically expanding as more transaction log space is needed. As the transaction log expands, a new virtual log file is created, and write operations to the transaction log wait while the transaction log is expanded. If the transaction log expands too frequently, performance can be affected.

- Set the file growth increment percentage to a reasonable size to prevent the file from growing by too small a value. If the file growth is too small compared to the number of log records being written to the transaction log, then the transaction log may need to expand constantly, affecting performance.

- Manually shrink the transaction log files rather than allowing Microsoft® SQL Server™ 2000 to shrink the files automatically. Shrinking the transaction log can affect performance on a busy system due to the movement and locking of data pages.

See Also

Transaction Logs

Virtual Log Files

Optimizing Database Performance

Optimizing tempdb Performance

General recommendations for the physical placement and database options set for the **tempdb** database include:

- Allow the **tempdb** database to automatically expand as needed. This ensures that queries that generate larger than expected intermediate result sets stored in the **tempdb** database are not terminated before execution is complete.

- Set the original size of the **tempdb** database files to a reasonable size to avoid the files from automatically expanding as more space is needed. If the **tempdb** database expands too frequently, performance can be affected.

- Set the file growth increment percentage to a reasonable size to avoid the **tempdb** database files from growing by too small a value. If the file growth is too small compared to the amount of data being written to the **tempdb** database, then **tempdb** may need to constantly expand, thereby affecting performance.

- Place the **tempdb** database on a fast I/O subsystem to ensure good performance. Stripe the **tempdb** database across multiple disks for better performance. Use filegroups to place the **tempdb** database on disks different from those used by user databases.

See Also

Expanding a Database

Optimizing Database Performance

File Systems

Server performance is not affected by the file system used (FAT or NTFS). Your choice of file system should be determined by factors other than performance.

- The File Allocation Table (FAT) file system allows dual booting with computers running Microsoft® MS-DOS®, Microsoft Windows® 95, or Microsoft Windows 98.

- The Microsoft Windows NT® file system (NTFS) has security and recovery advantages.

  If you do not need to dual-boot Windows NT or Windows 2000 with MS-DOS, Windows 95, or Windows 98, NTFS is recommended.

WARNING Microsoft SQL Server™ 2000 data and transaction log files must not be placed on compressed file systems.

For more information about choosing the appropriate file system, see the operating system documentation.

Note When running on Windows NT, SQL Server performance can be improved further if the databases are created on disks formatted using NTFS and, specifically, 64-KB extent sizes. For more information about formatting an NTFS disk, see the Windows NT documentation.

Optimizing Database Performance

Query Tuning

It may be tempting to address a performance problem solely by system-level server performance tuning; for example, memory size, type of file system, number and type of processors, and so forth. Experience has shown that most performance problems cannot be resolved this way. They must be addressed by analyzing the application, queries, and updates that the application is submitting to the database, and how these queries and updates interact with the database schema.

Unexpected long-lasting queries and updates can be caused by:

- Slow network communication.
- Inadequate memory in the server computer, or not enough memory available for Microsoft® SQL Server™ 2000.
- Lack of useful statistics.
- Out-of-date statistics.
- Lack of useful indexes.
- Lack of useful data striping.

When a query or update takes longer than expected, use the following checklist to improve performance.

Note It is recommended that this checklist be consulted prior to contacting your technical support provider.

1. Is the performance problem related to a component other than queries? For example, is the problem slow network performance? Are there any other components that might be causing or contributing to performance degradation? Windows NT Performance Monitor can be

used to monitor the performance of SQL Server and non-SQL Server related components. For more information, see Monitoring with System Monitor.

2. If the performance issue is related to queries, which query or set of queries is involved? Use SQL Profiler to help identify the slow query or queries. For more information, see Monitoring with SQL Profiler.

   The performance of a database query can be determined by using the SET statement to enable the SHOWPLAN, STATISTICS IO, STATISTICS TIME, and STATISTICS PROFILE options.

   - SHOWPLAN describes the method chosen by the SQL Server query optimizer to retrieve data. For more information, see SET SHOWPLAN_ALL.
   - STATISTICS IO reports information about the number of scans, logical reads (pages accessed in cache), and physical reads (number of times the disk was accessed) for each table referenced in the statement. For more information, see SET STATISTICS IO.
   - STATISTICS TIME displays the amount of time (in milliseconds) required to parse, compile, and execute a query. For more information, see SET STATISTICS TIME.
   - STATISTICS PROFILE displays a result set after each executed query representing a profile of the execution of the query. For more information, see SET STATISTICS PROFILE.

   In SQL Query Analyzer, you can also turn on the **graphical execution plan** option to view a graphical representation of how SQL Server retrieves data.

   The information gathered by these tools allows you to determine how a query is executed by the SQL Server query optimizer and which

indexes are being used. Using this information, you can determine if performance improvements can be made by rewriting the query, changing the indexes on the tables, or perhaps modifying the database design. For more information, see Analyzing a Query.

3. Was the query optimized with useful statistics?

   Statistics on the distribution of values in a column are automatically created on indexed columns by SQL Server. They can also be created on nonindexed columns either manually, using SQL Query Analyzer or the CREATE STATISTICS statement, or automatically, if the **auto create statistics** database option is set to **true**. These statistics can be used by the query processor to determine the optimal strategy for evaluating a query. Maintaining additional statistics on nonindexed columns involved in join operations can improve query performance. For more information, see Statistical Information.

   Monitor the query using SQL Profiler or the graphical execution plan in SQL Query Analyzer to determine if the query has enough statistics. For more information, see Error and Warning Event Category.

4. Are the query statistics up-to-date? Are the statistics automatically updated?

   SQL Server automatically creates and updates query statistics on indexed columns (as long as automatic query statistic updating is not disabled). Additionally, statistics can be updated on nonindexed columns either manually, using SQL Query Analyzer or the UPDATE STATISTICS statement, or automatically, if the **auto update statistics** database option is set to **true**. Up-to-date statistics are not dependent upon date or time data. If no UPDATE operations have taken place, then the query statistics are still up-to-date.

   If statistics are not set to update automatically, then set them to do so. For more information, see Statistical Information.

5. Are suitable indexes available? Would adding one or more indexes improve query performance? For more information, see Index Tuning Recommendations.

6. Are there any data or index hot spots? Consider using disk striping. For more information, see Data Placement Using Filegroups and RAID.

7. Is the query optimizer provided with the best opportunity to optimize a complex query? For more information, see Query Tuning Recommendations.

See Also

Advanced Query Concepts

Query Processor Architecture

Monitoring with SQL Server Enterprise Manager

SET

Parallel Query Processing

Optimizing Database Performance

Analyzing a Query

Microsoft® SQL Server™ 2000 offers these ways to present information on how it navigates tables and uses indexes to access the data for a query:

- Graphically display the execution plan using SQL Query Analyzer

  In SQL Query Analyzer, click **Query** and select **Display Execution Plan**. After executing a query, you can select the **Execution Plan** tab to see a graphical representation of execution plan output. For more information, see Graphically Displaying the Execution Plan Using SQL Query Analyzer.

- SET SHOWPLAN_TEXT ON

  After this statement is executed, SQL Server returns the execution plan information for each query. For more information, see SET SHOWPLAN_TEXT.

- SET SHOWPLAN_ALL ON

  This statement is similar to SET SHOWPLAN_TEXT, except that the output is in a concise format. For more information, see SET SHOWPLAN_ALL.

When you display the execution plan, the statements you submit to the server are not executed; instead, SQL Server analyzes the query and displays how the statements would have been executed as a series of operators.

Note Because statements are not executed when the execution plan is displayed, Transact-SQL operations such as creating a table do not cause the table to be created. Therefore, subsequent operations involving the table return errors because the table does not exist.

The best execution plan used by the query engine for individual data manipulation language (DML) and Transact-SQL statements is displayed, and reveals compile-time information about stored procedures, triggers invoked by a batch, and called stored procedures and triggers invoked to an arbitrary number of calling levels. For example, executing a SELECT statement can show that SQL Server uses a table scan to obtain the data. Alternatively, an index scan may

have been used instead if the index was determined to be a faster method of retrieving the data from the table.

The results returned by the SHOWPLAN_TEXT and SHOWPLAN_ALL statements are a tabular representation (rows and columns) of a tree structure. The execution plan tree structure uses one row in the result set for each node in the tree, each node representing a logical or physical operator used to manipulate the data to produce expected results. SQL Query Analyzer instead graphically displays each logical and physical operator as an icon. For more information, see Logical and Physical Operators.

Optimizing Database Performance

Graphically Displaying the Execution Plan Using SQL Query Analyzer

SQL Query Analyzer is an interactive, graphical tool that enables a database administrator or developer to write queries, execute multiple queries simultaneously, view results, analyze the query plan, and receive assistance to improve the query performance. The Execution Plan options graphically display the data retrieval methods chosen by the Microsoft® SQL Server™ 2000 query optimizer. The graphical execution plan uses icons to represent the execution of specific statements and queries in SQL Server rather than the tabular representation produced by the SET SHOWPLAN_ALL or SET SHOWPLAN_TEXT statements. This is very useful for understanding the performance characteristics of a query. Additionally, SQL Query Analyzer shows suggestions for additional indexes and statistics on nonindexed columns that would improve the ability of the query optimizer to process a query efficiently. In particular, SQL Query Analyzer shows which statistics are missing, thereby forcing the query optimizer to make estimates about predicate selectivity, and then permits those missing statistics to be easily created.

The following icons displayed in the graphical execution plan represent the physical operators used by SQL Server to execute statements. For more information, see Logical and Physical Operators.

Icon	Physical operator
	Assert
	Bookmark Lookup
	Clustered Index Delete
	Clustered Index Insert
	Clustered Index Scan
	Clustered Index Seek
	Clustered Index Update
	Collapse

	Compute Scalar
	Concatenation
	Constant Scan
	Deleted Scan
	Filter (clsColumn)
	Hash Match
	Hash Match Root
	Hash Match Team
	Index Delete
	Index Insert
	Index Scan
	Index Seek
	Index Spool
	Index Update
	Inserted Scan
	Log Row Scan
	Merge Join
	Nested Loops
	Parallelism
	Parameter Table Scan
	Remote Delete
	Remote Insert
	Remote Query
	Remote Scan
	Remote Update
	Row Count Spool

	Sequence
	Sort
	Stream Aggregate
	Table Delete
	Table Insert
	Table Scan
	Table Spool
	Table Update
	Top

The following icons displayed in the graphical execution plan represent the cursor physical operators used by SQL Server to execute statements.

Icon	**Cursor physical operator**
	Dynamic
	Fetch Query
	Keyset
	Population Query
	Refresh Query
	Snapshot

Reading the Graphical Execution Plan Output

The graphical execution plan output in SQL Query Analyzer is read from right to left and from top to bottom. Each query in the batch that is analyzed is displayed, including the cost of each query as a percentage of the total cost of the batch.

- Each node in the tree structure is represented as an icon that specifies

the logical and physical operator used to execute part of the query or statement.

- Each node is related to a parent node. All nodes with the same parent are drawn in the same column. Rules with arrowheads connect each node to its parent.
- Recursive operations are shown with an iteration symbol.
- Operators are shown as symbols related to a specific parent.
- When the query contains multiple statements, multiple query execution plans are drawn.
- The parts of the tree structures are determined by the type of statement executed.

Type of statement	Tree structure element
Transact-SQL and stored procedures	If the statement is a stored procedure or Transact-SQL statement, it becomes the root of the graphical execution plan tree structure. The stored procedure can have multiple children that represent statements called by the stored procedure. Each child is a node or branch of the tree.
Data manipulation language (DML)	If the statement analyzed by the SQL Server query optimizer is a DML statement, such as SELECT, INSERT, DELETE, or UPDATE, the DML statement is the root of the tree. DML statements can have up to two children. The first child is the execution plan for the DML statement. The second child represents a trigger, if used in or by the statement.
Conditional	The graphical execution plan divides conditional

	statements such as IF...ELSE statements (if condition exists, then do the following, else do this statement instead) into three children. The IF...ELSE statement is the root of the tree. The if condition becomes a subtree node. The then and else conditions are represented as statement blocks. WHILE and DO-UNTIL statements are represented using a similar plan.
Relational operators	Operations performed by the query engine, such as table scans, joins, and aggregations, are represented as nodes on the tree.
DECLARE CURSOR	The DECLARE CURSOR statement is the root of the graphical execution plan tree, with its related statement as a child or node.

Each node displays ToolTip information when the cursor is pointed at it. The ToolTip information can include:

- The physical operator (**Physical Operation**) used, such as Hash Join or Nested Loops. Physical operators displayed in red indicate that the query optimizer has issued a warning, such as missing column statistics or missing join predicates. This can cause the query optimizer to choose a less-efficient query plan than otherwise expected. For more information about column statistics, see Statistical Information. The graphical execution plan suggests remedial action, such as creating or updating statistics, or creating an index. The missing column statistics and indexes can be immediately created or updated using the context menus of SQL Query Analyzer.

- The logical operator (**Logical Operation**) that matches the physical operator, such as the Join operator. The logical operator, if different from the physical operator, is listed after the physical operator at the top of the ToolTip and separated by a forward slash (/).

- The number of rows (**Row Count**) output by the operator.

- The estimated size of the row (**Estimated Row Size**) output by the operator.

- The estimated cost (**I/O Cost**) of all I/O activity for the operation. This value should be as low as possible.

- The estimated cost for all CPU activity (**CPU Cost**) for the operation.

- The number of times the operation was executed (**Number of executes**) during the query.

- The cost to the query optimizer (**Cost**) in executing this operation, including cost of this operation as a percentage of the total cost of the query. Because the query engine selects the most efficient operation to perform the query or execute the statement, this value should be as low as possible.

- The total cost to the query optimizer (**Subtree cost**) in executing this operation and all operations preceding it in the same subtree.

- The predicates and parameters (**Argument**) used by the query.

To create statistics

Optimizing Database Performance

Logical and Physical Operators

The logical and physical operators describe how a query or update was executed. The physical operators describe the physical implementation algorithm used to process a statement, for example, scanning a clustered index. Each step in the execution of a query or update statement involves a physical operator. The logical operators describe the relational algebraic operation used to process a statement, for example, performing an aggregation. Not all steps used to process a query or update involve logical operations.

Assert

The Assert logical and physical operator verifies a condition. For example, it validates referential integrity or check constraints, or ensures that a scalar subquery returns one row. For each input row, the Assert operator evaluates the expression in the **Argument** column. If this expression evaluates to NULL, the row is passed through the Assert operator. If this expression evaluates to a nonnull value, the appropriate error will be raised.

Aggregate

The Aggregate logical operator groups the input by a set of columns and calculates aggregate expressions (MIN, MAX, SUM, and so on).

See Also

Aggregate Functions

Bookmark Lookup

The Bookmark Lookup logical and physical operator uses a bookmark (row ID or clustering key) to look up the corresponding row in the table or clustered index. The **Argument** column contains the bookmark label used to look up the row in the table or clustered index. The **Argument** column also contains the name of the table or clustered index in which the row is looked up. If the WITH PREFETCH clause appears in the **Argument** column, then the query processor has determined that it is optimal to use asynchronous prefetching (read-ahead) when looking up bookmarks in the table or clustered index.

Clustered Index Delete

The Clustered Index Delete physical operator deletes rows from the clustered index specified in the **Argument** column. If a WHERE:() predicate is present in the **Argument** column, then only those rows that satisfy the predicate are deleted.

See Also

Using Clustered Indexes

Clustered Index Insert

The Clustered Index Insert physical operator inserts rows from its input into the clustered index specified in the **Argument** column. The **Argument** column will also contain a SET:() predicate, which indicates the value to which each column is set.

See Also

Using Clustered Indexes

Clustered Index Update

The Clustered Index Update physical operator updates input rows in the clustered index specified in the **Argument** column.

If a WHERE:() predicate is present, only those rows that satisfy this predicate are updated. If a SET:() predicate is present, it indicates the value to which each updated column is set. If a DEFINE:() predicate is present, this lists the values that this operator defines. These values may be referenced in the SET clause or elsewhere within this operator and elsewhere within this query.

See Also

Using Clustered Indexes

Clustered Index Scan

The Clustered Index Scan logical and physical operator scans the clustered index specified in the **Argument** column. When an optional WHERE:() predicate is present, only those rows that satisfy the predicate are returned. If the **Argument** column contains the ORDERED clause, the query processor has requested that the rows' output be returned in the order in which the clustered index has sorted them. If the ORDERED clause is not present, the storage engine will scan the index in the optimal way (not guaranteeing the output to be sorted).

See Also

Using Clustered Indexes

Clustered Index Seek

The Clustered Index Seek logical and physical operator uses the seeking ability of indexes to retrieve rows from a clustered index.

The **Argument** column contains the name of the clustered index being used and the SEEK:() predicate. The storage engine uses the index to process only those rows that satisfy this SEEK:() predicate. It optionally can include a WHERE:() predicate, which the storage engine evaluates against all rows satisfying the SEEK:() predicate (it does not use indexes to do this).

If the **Argument** column contains the ORDERED clause, the query processor has determined that the rows must be returned in the order in which the clustered index has sorted them. If the ORDERED clause is not present, the storage engine searches the index in the optimal way (not guaranteeing the output to be sorted). Allowing the output to retain its ordering can be less efficient than producing nonsorted output.

See Also

Using Clustered Indexes

Collapse

The Collapse logical and physical operator optimizes update processing. When an update is performed, it can be split (using the Split operator) into a delete and an insert. If the **Argument** column contains a GROUP BY:() predicate and a list of key columns being grouped, the query processor groups by the set of key columns to optimize these update operations by removing any temporary, unnecessary intermediate changes to each row.

See Also

Split

Compute Scalar

The Compute Scalar logical and physical operator evaluates an expression to produce a computed scalar value, which may be returned to the user and/or referenced elsewhere in the query, for example, in a filter predicate or join predicate.

See Also

Functions

Concatenation

The Concatenation logical and physical operator scans multiple inputs, returning each row scanned.

Constant Scan

The Constant Scan logical and physical operator introduces a constant row into a query. It will return either zero or one row, which usually contains no columns. A Compute Scalar operator is often used to add columns to the row produced by a Constant Scan.

See Also

Compute Scalar

Cross Join

The Cross Join logical operator joins each row from the first (top) input with each row from the second (bottom) input.

See Also

Using Cross Joins

Delete

The Delete logical operator deletes from an object rows that satisfy the optional predicate in the **Argument** column.

See Also

DELETE

Deleted Scan

The Deleted Scan logical and physical operator scans the deleted table within a trigger.

See Also

Using the inserted and deleted Tables

Distinct

The Distinct logical operator scans the input, removing duplicates.

See Also

Eliminating Duplicates with DISTINCT

Distinct Sort

The Distinct Sort logical operator scans the input, removing duplicates and sorting by the columns specified in the DISTINCT ORDER BY:() predicate of the **Argument** column.

See Also

Distinct

Eliminating Duplicates with DISTINCT

Distribute Streams

The Distribute Streams logical operator is used only in parallel query plans. The Distribute Streams operator consumes a single input stream of records and produces multiple output streams. The record contents and format are not changed. Each record from the input stream appears in one of the output streams. This operator automatically preserves the relative order of the input records in the output streams. Usually, hashing is used to decide to which output stream a particular input record belongs.

If the output is partitioned, then the **Argument** column contains a PARTITION COLUMNS:() predicate and the partitioning columns.

See Also

Gather Streams

Parallel Query Processing

Parallelism

Repartition Streams

Eager Spool

The Eager Spool logical operator will consume the entire input, storing each row in a hidden temporary object stored in the **tempdb** database. If the operator is rewound (for example, by a Nested Loops operator) but no rebinding is needed, the spooled data is used instead of rescanning the input. If rebinding is needed, the spooled data is discarded and the spool object is rebuilt by rescanning the (rebound) input.

The Eager Spool operator will build its spool file eagerly. When the spool's parent operator asks for the first row, the spool operator will consume all rows from its input operator and store them in the spool.

Note An alternative way of building a spool file is with the Lazy Spool operator.

Filter

The Filter logical and physical operator scans the input, returning only those rows that satisfy the filter expression (predicate) that appears in the **Argument** column.

Flow Distinct

The Flow Distinct logical operator scans the input, removing duplicates. Whereas the Distinct operator consumes all input before producing any output, the Flow Distinct operator returns each row as it is obtained from the input (unless that row is a duplicate, in which case it is discarded).

See Also

Distinct

Eliminating Duplicates with DISTINCT

Full Outer Join

The Full Outer Join logical operator returns each row satisfying the join predicate from the first (top) input joined with each row from the second (bottom) input. It also returns rows from:

- The first input that had no matches in the second input.
- The second input that had no matches in the first input.

The input that does not contain the matching values is returned as a null value.

See Also

Using Outer Joins

Gather Streams

The Gather Streams logical operator is only used in parallel query plans. The Gather Streams operator consumes several input streams and produces a single output stream of records by combining the input streams. The record contents and format are not changed. If this operator is order-preserving, then all input streams must be ordered.

If the output is ordered, then the **Argument** column contains an ORDER BY:() predicate and the names of columns being ordered.

See Also

Distribute Streams

Parallel Query Processing

Parallelism

Repartition Streams

Hash Match

The Hash Match physical operator builds a hash table by computing a hash value for each row from its build input. A HASH:() predicate with a list of columns used to create a hash value appears in the **Argument** column. Then, for each probe row (as applicable), it computes a hash value (using the same hash function) and looks in the hash table for matches. If a residual predicate is present (identified by RESIDUAL:() in the **Argument** column), that predicate must also be satisfied for rows to be considered a match. Behavior is slightly different based on the logical operation being performed:

- For any joins, use the first (top) input to build the hash table and the second (bottom) input to probe the hash table. Output matches (or nonmatches) as dictated by the join type. If multiple joins use the same join column, these operations are grouped into a hash team.

- For the distinct or aggregate operators, use the input to build the hash table (removing duplicates and computing any aggregate expressions). When the hash table is built, scan the table and output all entries.

- For the union operator, use the first input to build the hash table (removing duplicates). Use the second input (which must have no duplicates) to probe the hash table, returning all rows that have no matches, then scan the hash table and return all entries.

See Also

Distinct

Understanding Hash Joins

Hash Match Team

Union

Hash Match Root

The Hash Match Root physical operator coordinates the operation of all Hash Match Team operators directly below it. The Hash Match Root operator and all Hash Match Team operators directly below it share a common hash function and partitioning strategy. The Hash Match Root operator always returns output to an operator that is not a member of its team.

See Also

Hash Match Team

Understanding Hash Joins

Hash Match Team

The Hash Match Team physical operator is part of a team of connected hash operators sharing a common hash function and partitioning strategy.

See Also

Hash Match Root

Understanding Hash Joins

Index Delete

The Index Delete physical operator will delete input rows from the nonclustered index specified in the **Argument** column. If a WHERE:() predicate is present, only those rows that satisfy this predicate will be deleted.

See Also

DELETE

Using Nonclustered Indexes

Index Insert

The Index Insert physical operator inserts rows from its input into the nonclustered index specified in the **Argument** column. The **Argument** column will also contain a SET:() predicate, which indicates the value to which each column is set.

See Also

INSERT

Using Nonclustered Indexes

Index Scan

The Index Scan logical and physical operator retrieves all rows from the nonclustered index specified in the **Argument** column. If an optional WHERE:() predicate appears in the **Argument** column, only those rows that satisfy the predicate are returned.

If the **Argument** column must contain the ORDERED clause, the query processor has determined that the rows be returned in the order in which the nonclustered index has sorted them. If the ORDERED clause is not present, the storage engine will search the index in the optimal way (which does not guarantee that the output will be sorted).

See Also

SELECT

Using Nonclustered Indexes

Index Seek

The Index Seek logical and physical operator uses the seeking ability of indexes to retrieve rows from a nonclustered index.

The **Argument** column contains the name of the nonclustered index being used. It also contains the SEEK:() predicate. The storage engine uses the index to process only those rows that satisfy the SEEK:() predicate. It optionally may include a WHERE:() predicate, which the storage engine will evaluate against all rows that satisfy the SEEK:() predicate (it does not use the indexes to do this).

If the **Argument** column contains the ORDERED clause, the query processor has determined that the rows must be returned in the order in which the nonclustered index has sorted them. If the ORDERED clause is not present, the storage engine searches the index in the optimal way (which does not guarantee that the output will be sorted). Allowing the output to retain its ordering may be less efficient than producing nonsorted output.

See Also

SELECT

Using Nonclustered Indexes

Index Spool

The Index Spool physical operator contains a SEEK:() predicate in the **Argument** column. The Index Spool operator scans its input rows, placing a copy of each row in a hidden spool file (stored in the **tempdb** database and existing only for the lifetime of the query), and builds an index on the rows. This allows you to use the seeking capability of indexes to output only those rows that satisfy the SEEK:() predicate.

If the operator is rewound (for example, by a Nested Loops operator) but no rebinding is needed, the spooled data is used instead of rescanning the input.

See Also

Creating an Index

Index Update

The Index Update physical operator updates rows from its input in the nonclustered index specified in the **Argument** column.

If a WHERE:() predicate is present, only those rows that satisfy this predicate are updated. If a SET:() predicate is present, it indicates the value to which each updated column is set. If a DEFINE:() predicate is present, it lists the values that this operator defines. These values may be referenced in the SET clause, or elsewhere within this operator and elsewhere within this query.

See Also

UPDATE

Using Nonclustered Indexes

Inner Join

The Inner Join logical operator returns each row that satisfies the join of the first (top) input with the second (bottom) input.

See Also

Using Inner Joins

Insert

The Insert logical operator inserts each row from its input into the object specified in the **Argument** column. The physical operator will be either the Table Insert, Index Insert, or Clustered Index Insert operator.

See Also

Clustered Index Insert

Table Insert

Index Insert

Inserted Scan

The Inserted Scan logical and physical operator scans the inserted table within a trigger.

See Also

Using the inserted and deleted Tables

Lazy Spool

The Lazy Spool logical operator stores each row from its input in a hidden temporary object stored in the **tempdb** database. If the operator is rewound (for example, by a Nested Loops operator) but no rebinding is needed, the spooled data is used instead of rescanning the input. If rebinding is needed, the spooled data is discarded and the spool object is rebuilt by rescanning the (rebound) input.

The Lazy Spool operator will build its spool file in a lazy (noneager) manner. Each time the spool's parent operator asks for a row, the spool operator gets a row from its input operator and stores it in the spool.

See Also

Eager Spool

Left Anti Semi Join

The Left Anti Semi Join logical operator returns each row from the first (top) input when there is no matching row in the second (bottom) input. If no join predicate exists in the **Argument** column, each row is a matching row.

See Also

Using Joins

Left Outer Join

The Left Outer Join logical operator returns each row that satisfies the join of the first (top) input with the second (bottom) input. It also returns any rows from the first input that had no matching rows in the second input. The nonmatching rows in the second input are returned as null values. If no join predicate exists in the **Argument** column, each row is a matching row.

See Also

Using Outer Joins

Left Semi Join

The Left Semi Join logical operator returns each row from the first (top) input when there is a matching row in the second (bottom) input. If no join predicate exists in the **Argument** column, each row is a matching row.

See Also

Using Joins

Log Row Scan

The Log Row Scan logical and physical operator scans the transaction log.

See Also

Transaction Logs

Merge Interval

The Merge Interval logical and physical operator merges multiple (potentially overlapping) intervals to produce minimal, nonoverlapping intervals that are then used to seek index entries. This operator typically appears above one or more Compute Scalar operators over Constant Scan operators, which construct the intervals (represented as columns in a row) that this operator merges.

See Also

Compute Scalar

Constant Scan

Merge Join

The Merge Join physical operator performs the Inner Join, Left Outer Join, Left Semi Join, Left Anti Semi Join, Right Outer Join, Right Semi Join, Right Anti Semi Join, and Union logical operations.

In the **Argument** column, the Merge Join operator contains a MERGE:() predicate if the operation is performing a one-to-many join, or a MANY-TO-MANY MERGE:() predicate if the operation is performing a many-to-many join. The **Argument** column also includes a comma-separated list of columns used to perform the operation. The Merge Join operator requires two inputs sorted on their respective columns, possibly by inserting explicit sort operations into the query plan. Merge join is particularly effective if explicit sorting is not required, for example, if there is a suitable B-tree index in the database or if the sort order can be exploited for multiple operations, such as a merge join and grouping with roll up.

See Also

Understanding Merge Joins

Nested Loops

The Nested Loops physical operator performs the Inner Join, Left Outer Join, Left Semi Join, and Left Anti Semi Join logical operations.

Nested loops joins perform a search on the inner table for each row of the outer table, typically using an index. Microsoft® SQL Server™ 2000 decides, based on anticipated costs, whether to sort the outer input in order to improve locality of the searches on the index over the inner input.

Any rows that satisfy the (optional) predicate in the **Argument** column are returned (as applicable based on the logical operation being performed).

See Also

Understanding Nested Loops Joins

Parallelism

The Parallelism physical operator performs the Distribute Streams, Gather Streams, and Repartition Streams logical operations. The **Argument** columns can contain a PARTITION COLUMNS:() predicate with a comma-separated list of the columns being partitioned. The **Argument** columns can also contain an ORDER BY:() predicate with a list of the columns for which the sort order is preserved during partitioning.

See Also

Distribute Streams

Repartition Streams

Gather Streams

Parameter Table Scan

The Parameter Table Scan logical and physical operator scans a table that is acting as a parameter in the current query. Typically, this is used for INSERT queries within a stored procedure.

See Also

INSERT

Remote Delete

The Remote Delete logical and physical operator deletes the input rows from a remote object.

Remote Insert

The Remote Insert logical and physical operator inserts the input rows into a remote object.

Remote Query

The Remote Query logical and physical operator submits a query to a remote source. The text of the query sent to the remote server appears in the **Argument** column.

See Also

Optimizing Distributed Queries

Remote Scan

The Remote Scan logical and physical operator will scan a remote object. The name of the remote object appears in the **Argument** column.

Remote Update

The Remote Update logical and physical operator updates the input rows in a remote object.

Repartition Streams

The Repartition Streams logical operator is used only in parallel query plans. The Repartition Streams operator consumes multiple streams and produces multiple streams of records. The record contents and format are not changed. Each record from an input stream is placed into one output stream. If this operator is order-preserving, then all input streams must be ordered and merged into several ordered output streams.

If the output is partitioned, then the **Argument** column contains a PARTITION COLUMNS:() predicate and the partitioning columns.

If the output is ordered, then the **Argument** column contains an ORDER BY:() predicate and the columns being ordered.

See Also

Distribute Streams

Parallel Query Processing

Gather Streams

Right Anti Semi Join

The Right Anti Semi Join logical operator will output each row from the second (bottom) input when a matching row in the first (top) input does not exist. A matching row is defined as a row that satisfies the predicate in the **Argument** column (if no predicate exists, each row is a matching row).

See Also

Using Joins

Right Outer Join

The Right Outer Join logical operator returns each row that satisfies the join of the second (bottom) input with each matching row from the first (top) input. It will also return any rows from the second input that had no matching rows in the first input, joined with NULL. If no join predicate exists in the **Argument** column, each row is a matching row.

See Also

Using Outer Joins

Right Semi Join

The Right Semi Join logical operator returns each row from the second (bottom) input when there is a matching row in the first (top) input. If no join predicate exists in the **Argument** column, each row is a matching row.

See Also

Using Joins

Row Count Spool

The Row Count Spool physical operator scans the input, counting how many rows are present and returning that many rows without any data in them. This operator is used when it is important to check for the existence of rows rather than the data contained in the rows. For example, if a Nested Loops operator performs a left semi join operation and the join predicate applies to inner input, a row count spool may be placed at the top of the inner input of the Nested Loops operator. Then the Nested Loops operator can look at how many rows are output by the row count spool (because the actual data from the inner side is not needed) to determine whether to return the outer row.

See Also

Left Semi Join

Nested Loops

Sequence

The Sequence logical and physical operator drives wide update plans. Functionally, it executes each input in sequence (top to bottom). Each input is usually an update of a different object. It returns only those rows that come from its last (bottom) input.

Sort

The Sort logical and physical operator sorts all incoming rows. The **Argument** column contains a DISTINCT ORDER BY:() predicate if duplicates are removed by this operation or an ORDER BY:() predicate with a comma-separated list of the columns being sorted. The columns are prefixed with the value ASC if the columns are sorted in ascending order, or the value DESC if the columns are sorted in descending order.

See Also

SELECT

Split

The Split logical and physical operator is used to optimize update processing. It splits each update operation into a delete and an insert operation.

See Also

Collapse

Stream Aggregate

The Stream Aggregate physical operator optionally groups by a set of columns and calculates one or more aggregate expressions returned by the query and/or referenced elsewhere within the query. This operator requires that input is ordered by the columns within its groups.

If the Stream Aggregate operator groups by columns, a GROUP BY:() predicate and the list of columns appear in the **Argument** column. If the Stream Aggregate operator computes any aggregate expressions, a list of them will appear in the **Defined Values** column of the output from the SHOWPLAN_ALL statement or the **Argument** column of the graphical execution plan.

See Also

Aggregate Functions

Table Delete

The Table Delete physical operator deletes rows from the table specified in the argument column. If a WHERE:() predicate is present in the **Argument** column, only those rows that satisfy the predicate will be deleted.

See Also

DELETE

Table Insert

The Table Insert physical operator inserts rows from its input into the table specified in the **Argument** column. The **Argument** column also contains a SET:() predicate, which indicates the value to which each column is set.

See Also

INSERT

Table Scan

The Table Scan logical and physical operator retrieves all rows from the table specified in the **Argument** column. If a WHERE:() predicate appears in the **Argument** column, only those rows that satisfy the predicate are returned.

See Also

SELECT

Table Spool

The Table Spool physical operator scans the input and places a copy of each row in a hidden spool table (stored in the **tempdb** database and existing only for the lifetime of the query). If the operator is rewound (for example, by a Nested Loops operator) but no rebinding is needed, the spooled data is used instead of rescanning the input.

Table Update

The Table Update physical operator updates input rows in the table specified in the **Argument** column. If a WHERE:() predicate is present, only those rows that satisfy this predicate are updated. If a SET:() predicate is present, it indicates the value to which each updated column is set. If a DEFINE:() predicate is present, this lists the values that this operator defines. These values may be referenced in the SET clause or elsewhere within this operator and elsewhere within this query.

See Also

UPDATE

Top

The Top logical and physical operator will scan the input, returning only the first specified number or percent of rows. The **Argument** column can optionally contain a list of the columns that are being checked for ties. In update plans, the Top operator is used to enforce row count limits.

See Also

Limiting Result Sets Using TOP and PERCENT

SET ROWCOUNT

Union

The Union logical operator scans multiple inputs, outputting each row scanned and removing duplicates.

See Also

UNION

Update

The Update logical operator updates each row from its input in the object specified in the **Argument** column. The physical operator is Table Update, Index Update, or Clustered Index Update.

See Also

UPDATE

Cursor Logical and Physical Operators

The Cursor logical and physical operators are used to describe how a query, or update involving cursor operations, is executed. The physical operators describe the physical implementation algorithm used to process the cursor; for example, using a keyset-driven cursor. Each step in the execution of a cursor involves a physical operator. The logical operators describe a property of the cursor, such as the cursor is read only.

Logical Operators

The Cursor logical operators include:

Asynchronous

The cursor table is populated asynchronously. For more information, see Asynchronous Population.

Optimistic

This cursor uses the optimistic mode of concurrency. For more information, see Cursor Concurrency.

Primary

This is the primary fetch query for this cursor.

Read Only

This cursor uses read-only semantics for concurrency. This cursor can only read data, not insert, update, or delete it. For more information, see Cursor Concurrency.

Scroll Locks

This cursor uses scroll locks for concurrency. For more information, see Cursor Concurrency.

Secondary

This is the secondary fetch query (used if the primary fetch query fails).

Synchronous

The cursor table is populated synchronously.

Physical Operators

The Cursor physical operators include:

Dynamic

This cursor can see all changes made by others. For more information, see Dynamic Cursors.

Fetch Query

This query retrieves rows when a fetch is issued against a cursor.

Keyset

This cursor can see updates made by others, but not inserts. For more information, see Keyset-driven Cursors.

Population Query

This query populates a cursor's work table when the cursor is opened.

Refresh Query

This query fetches current data for rows in the cursor fetch buffer.

Snapshot

This cursor does not see changes made by others. For more information, see Static Cursors.

See Also

Cursors

Optimizing Database Performance

Query Tuning Recommendations

Some queries are inherently resource intensive. This is related to fundamental database and index issues. These queries are not inefficient, because the query optimizer will implement the queries in the most efficient fashion possible. However, they are resource intensive, and the set-oriented nature of Transact-SQL can make them appear inefficient. No degree of query optimizer intelligence can eliminate the inherent resource cost of these constructs. They are intrinsically costly when compared to a less complex query. Although Microsoft® SQL Server™ 2000 uses the most optimal access plan, this is limited by what is fundamentally possible. For example, the following types of queries can be resource intensive:

- Queries returning large result sets

- Highly nonunique WHERE clauses

However, recommendations for tuning queries and improving query performance include:

- Add more memory (especially if the server runs many complex queries and several of the queries execute slowly).

- Run SQL Server on a computer with more than one processor. Multiple processors allow SQL Server to make use of parallel queries. For more information, see Parallel Query Processing.

- Consider rewriting the query.
    - If the query uses cursors, determine if the cursor query could be written more efficiently using either a more efficient cursor type, such as fast forward-only, or a single query. Single queries typically outperform cursor operations. Because a set of cursor statements is typically an outer loop operation, in which each row in the outer loop is processed once using an inner statement, consider using either a GROUP BY or CASE

statement or a subquery instead.

- If an application uses a loop, consider putting the loop inside the query. Often an application will contain a loop that contains a parameterized query, which is executed many times and requires a network round trip between the computer running the application and SQL Server. Instead, create a single, more complex query using a temporary table. Only one network round trip is necessary, and the query optimizer can better optimize the single query.

- Do not use multiple aliases for a single table in the same query to simulate index intersection. This is no longer necessary because SQL Server automatically considers index intersection and can make use of multiple indexes on the same table in the same query. For example, given the sample query:

```
SELECT * FROM lineitem
WHERE partkey BETWEEN 17000 AND 17100 AND
    shipdate BETWEEN '1/1/1994' AND '1/31/1994''
```

  SQL Server can exploit indexes on both the **partkey** and **shipdate** columns, and then perform a hash match between the two subsets to obtain the index intersection.

- Make use of query hints only if necessary. Queries using hints executed against earlier versions of SQL Server should be tested without the hints specified. The hints can prevent the query optimizer from choosing a better execution plan. For more information, see SELECT.

- Make use of the **query governor** configuration option and setting. The **query governor** configuration option can be used to prevent long-running queries from executing, thus preventing system resources from being consumed. By default, the **query governor** configuration option allows all queries to execute, no matter how long they take. However, the query governor can be set to the maximum number of seconds that

all queries for all connections, or just the queries for a specific connection, are allowed to execute. Because the query governor is based on estimated query cost, rather than actual elapsed time, it does not have any run-time overhead. It also stops long-running queries before they start, rather than running them until some predefined limit is hit. For more information, see query governor cost limit Option and SET QUERY_GOVERNOR_COST_LIMIT.

See Also

CASE

Subquery Fundamentals

GROUP BY Components

Optimizing Database Performance

Advanced Query Tuning Concepts

Microsoft® SQL Server™ 2000 performs sort, intersect, union, and difference operations using in-memory sorting and hash join technology. Using this type of query plan, SQL Server supports vertical table partitioning, sometimes called columnar storage.

SQL Server employs three types of join operations:

- Nested loops joins
- Merge joins
- Hash joins

If one join input is quite small (such as fewer than 10 rows) and the other join input is fairly large and indexed on its join columns, index nested loops are the fastest join operation because they require the least I/O and the fewest comparisons. For more information about nested loops, see Understanding Nested Loops Joins.

If the two join inputs are not small but are sorted on their join column (for example, if they were obtained by scanning sorted indexes), merge join is the fastest join operation. If both join inputs are large and the two inputs are of similar sizes, merge join with prior sorting and hash join offer similar performance. However, hash join operations are often much faster if the two input sizes differ significantly from each other. For more information, see Understanding Merge Joins.

Hash joins can process large, unsorted, nonindexed inputs efficiently. They are useful for intermediate results in complex queries because:

- Intermediate results are not indexed (unless explicitly saved to disk and then indexed) and often are not produced suitably sorted for the next operation in the query plan.
- Query optimizers estimate only intermediate result sizes. Because

estimates can be an order of magnitude wrong in complex queries, algorithms to process intermediate results not only must be efficient but also must degrade gracefully if an intermediate result turns out to be much larger than anticipated.

The hash join allows reductions in the use of denormalization to occur. Denormalization is typically used to achieve better performance by reducing join operations, in spite of the dangers of redundancy, such as inconsistent updates. Hash joins reduce the need to denormalize. Hash joins allow vertical partitioning (representing groups of columns from a single table in separate files or indexes) to become a viable option for physical database design. For more information, see Understanding Hash Joins.

Optimizing Database Performance

Understanding Nested Loops Joins

The nested loops join, also called nested iteration, uses one join input as the outer input table (shown as the top input in the graphical execution plan) and one as the inner (bottom) input table. The outer loop consumes the outer input table row by row. The inner loop, executed for each outer row, searches for matching rows in the inner input table. In the simplest case, the search scans an entire table or index; this is called a naive nested loops join. If the search exploits an index, it is called an index nested loops join. If the index is built as part of the query plan (and destroyed upon completion of the query), it is called a temporary index nested loops join. All these variants are considered by the query optimizer. A nested loops join is particularly effective if the outer input is quite small and the inner input is preindexed and quite large. In many small transactions, such as those affecting only a small set of rows, index nested loops joins are far superior to both merge joins and hash joins. In large queries, however, nested loops joins are often not the optimal choice.

Optimizing Database Performance

Understanding Merge Joins

The merge join requires that both inputs be sorted on the merge columns, which are defined by the equality (WHERE) clauses of the join predicate. The query optimizer typically scans an index, if one exists on the proper set of columns, or places a sort operator below the merge join. In rare cases, there may be multiple equality clauses, but the merge columns are taken from only some of the available equality clauses.

Because each input is sorted, the Merge Join operator gets a row from each input and compares them. For example, for inner join operations, the rows are returned if they are equal. If they are not equal, whichever row has the lower value is discarded and another row is obtained from that input. This process repeats until all rows have been processed.

The merge join operation may be either a regular or a many-to-many operation. A many-to-many merge join uses a temporary table to store rows. If there are duplicate values from each input, one of the inputs will have to rewind to the start of the duplicates as each duplicate from the other input is processed.

If a residual predicate is present, all rows that satisfy the merge predicate will evaluate the residual predicate, and only those rows that satisfy it will be returned.

Merge join itself is very fast, but it can be an expensive choice if sort operations are required. However, if the data volume is large and the desired data can be obtained presorted from existing B-tree indexes, merge join is often the fastest available join algorithm.

Optimizing Database Performance

Understanding Hash Joins

The hash join has two inputs: the build input and probe input. The query optimizer assigns these roles so that the smaller of the two inputs is the build input.

Hash joins are used for many types of set-matching operations: inner join; left, right, and full outer join; left and right semi-join; intersection; union; and difference. Moreover, a variant of the hash join can do duplicate removal and grouping (such as SUM(salary) GROUP BY department). These modifications use only one input for both the build and probe roles.

Similar to a merge join, a hash join can be used only if there is at least one equality (WHERE) clause in the join predicate. However, because joins are typically used to reassemble relationships, expressed with an equality predicate between a primary key and a foreign key, most joins have at least one equality clause. The set of columns in the equality predicate is called the hash key, because these are the columns that contribute to the hash function. Additional predicates are possible and are evaluated as residual predicates separate from the comparison of hash values. The hash key can be an expression, as long as it can be computed exclusively from columns in a single row. In grouping operations, the columns of the group by list are the hash key. In set operations such as intersection, as well as in the removal of duplicates, the hash key consists of all columns.

In-Memory Hash Join

The hash join first scans or computes the entire build input and then builds a hash table in memory. Each row is inserted into a hash bucket depending on the hash value computed for the hash key. If the entire build input is smaller than the available memory, all rows can be inserted into the hash table. This build phase is followed by the probe phase. The entire probe input is scanned or computed one row at a time, and for each probe row, the hash key's value is computed, the corresponding hash bucket is scanned, and the matches are produced.

Grace Hash Join

If the build input does not fit in memory, a hash join proceeds in several steps. Each step has a build phase and probe phase. Initially, the entire build and probe inputs are consumed and partitioned (using a hash function on the hash keys) into multiple files. The number of such files is called the partitioning fan-out. Using the hash function on the hash keys guarantees that any two joining records must be in the same pair of files. Therefore, the task of joining two large inputs has been reduced to multiple, but smaller, instances of the same tasks. The hash join is then applied to each pair of partitioned files.

Recursive Hash Join

If the build input is so large that inputs for a standard external merge sorts would require multiple merge levels, multiple partitioning steps and multiple partitioning levels are required. If only some of the partitions are large, additional partitioning steps are used for only those specific partitions. In order to make all partitioning steps as fast as possible, large, asynchronous I/O operations are used so that a single thread can keep multiple disk drives busy.

Note If the build input is larger but not a lot larger than the available memory, elements of in-memory hash join and grace hash join are combined in a single step, producing a hybrid hash join.

It is not always possible during optimization to determine which hash join will be used. Therefore, Microsoft® SQL Server™ 2000 starts using an in-memory hash join and gradually transitions to grace hash join, and recursive hash join, depending on the size of the build input.

If the optimizer anticipates wrongly which of the two inputs is smaller and, therefore, should have been the build input, the build and probe roles are reversed dynamically. The hash join makes sure that it uses the smaller overflow file as build input. This technique is called role reversal.

Optimizing Database Performance

Application Design

Application design plays a pivotal role in determining the performance of a system using Microsoft® SQL Server™ 2000. Consider the client the controlling entity rather than the database server. The client determines the type of queries, when they are submitted, and how the results are processed. This in turn has a major effect on the type and duration of locks, amount of I/O, and processing (CPU) load on the server, and hence on whether performance is generally good or bad.

For this reason, it is important to make the correct decisions during the application design phase. However, even if a performance problem occurs using a turn-key application, where changes to the client application seem impossible, this does not change the fundamental factors that affect performance: The client plays a dominant role and many performance problems cannot be resolved without making client changes. A well-designed application allows SQL Server to support thousands of concurrent users. Conversely, a poorly designed application prevents even the most powerful server platform from handling more than a few users.

Guidelines for client-application design include:

- Eliminate excessive network traffic.

  Network roundtrips between the client and SQL Server are usually the main reason for poor performance in a database application, an even greater factor than the amount of data transferred between server and client. Network roundtrips describe the conversational traffic sent between the client application and SQL Server for every batch and result set. By making use of stored procedures, you can minimize network roundtrips. For example, if your application takes different actions based on data values received from SQL Server, make those decisions directly in the stored procedure whenever possible, thus eliminating network traffic.

  If a stored procedure has multiple statements, then by default SQL Server sends a message to the client application at the completion of each statement and details the number of rows affected for each

statement. Most applications do not need these messages. If you are confident that your applications do not need them, you can disable these messages, which can improve performance on a slow network. Use the SET NOCOUNT session setting to disable these messages for the application. For more information, see SET NOCOUNT.

- Use small result sets.

  Retrieving needlessly large result sets (for example, thousands of rows) for browsing on the client adds CPU and network I/O load, makes the application less capable of remote use, and limits multiuser scalability. It is better to design the application to prompt the user for sufficient input so queries are submitted that generate modest result sets. For more information, see Optimizing Application Performance Using Efficient Data Retrieval.

  Application design techniques that facilitate this include exercising control over wildcards when building queries, mandating certain input fields, not allowing ad hoc queries, and using the TOP, PERCENT, or SET ROWCOUNT Transact-SQL statements to limit the number of rows returned by a query. For more information, see Limiting Result Sets Using TOP and PERCENT and SET ROWCOUNT.

- Allow cancellation of a query in progress when the user needs to regain control of the application.

  An application should never force the user to restart the client computer to cancel a query. Ignoring this can lead to irresolvable performance problems. When a query is canceled by an application, for example, using the open database connectivity (ODBC) **sqlcancel** function, proper care should be exercised regarding transaction level. Canceling a query, for example, does not commit or roll back a user-defined transaction. All locks acquired within the transaction are retained after the query is canceled. Therefore, after canceling a query, always either commit or roll back the transaction. The same issues apply to DB-Library and other application programming interfaces (APIs) that can be used to cancel queries.

- Always implement a query or lock time-out.

Do not allow queries to run indefinitely. Make the appropriate API call to set a query time-out. For example, use the ODBC **SQLSetStmtOption** function.

For more information about setting a query time-out, see the ODBC API documentation.

For more information about setting a lock time-out, see Customizing the Lock Time-out.

- Do not use application development tools that do not allow explicit control over the SQL statements sent to SQL Server.

  Do not use a tool that transparently generates Transact-SQL statements based on higher-level objects if it does not provide crucial features such as query cancellation, query time-out, and complete transactional control. It is often not possible to maintain good performance or to resolve a performance problem if the application generates transparent SQL statements, because this does not allow explicit control over transactional and locking issues, which are critical to the performance picture.

- Do not intermix decision support and online transaction processing (OLTP) queries. For more information, see Online Transaction Processing vs. Decision Support.

- Do not use cursors more than necessary.

  Cursors are a useful tool in relational databases; however, it is almost always more expensive to use a cursor than to use a set-oriented SQL statement to accomplish a task.

  In set-oriented SQL statements, the client application tells the server to update the set of records that meet specified criteria. The server figures out how to accomplish the update as a single unit of work. When updating through a cursor, the client application requires the server to maintain row locks or version information for every row, just in case the client asks to update the row after it has been fetched.

  Also, using a cursor implies that the server is maintaining client state

information, such as the user's current rowset at the server, usually in temporary storage. Maintaining this state for a large number of clients is an expensive use of server resources. A better strategy with a relational database is for the client application to get in and out quickly, maintaining no client state at the server between calls. Set-oriented SQL statements support this strategy.

However, if the query uses cursors, determine if the cursor query could be written more efficiently either by using a more-efficient cursor type, such as fast forward-only, or a single query. For more information, see Optimizing Application Performance Using Efficient Data Retrieval.

- Keep transactions as short as possible. For more information, see Effects of Transactions and Batches on Application Performance.

- Use stored procedures. For more information, see Effects of Stored Procedures on Application Performance.

- Use prepared execution to execute a parameterized SQL statement. For more information, see Prepared Execution (ODBC).

- Always process all results to completion.

  Do not design an application or use an application that stops processing result rows without canceling the query. Doing so will usually lead to blocking and slow performance. For more information, see Understanding and Avoiding Blocking.

- Ensure that your application is designed to avoid deadlocks. For more information, see Minimizing Deadlocks.

- Ensure that all the appropriate options for optimizing the performance of distributed queries have been set. For more information, see Optimizing Distributed Queries.

See Also

Deadlocking

Locking

Dynamic Locking

Transactions

Optimizing Database Performance

Networking and Performance

One key characteristic of client/server databases is the limited amount of network traffic involved. Many applications using Microsoft® SQL Server™ 2000 allow the user to log in over a modem link and still use the application effectively. Although network I/O performance is reduced by a factor of 1,000 compared to a local area network (LAN), you often see little performance degradation. However, if large amounts of data are transferred between the client and the server, network performance can be affected.

It is a good idea to monitor the traffic between your client applications and the server. An application designed and tuned for slow networks works great on a fast network, but the opposite is not true. If you use a higher-level development tool that generates Transact-SQL statements and issues queries and updates on your behalf, it is especially important to keep on eye on what is going across the network.

Optimizing Database Performance

Named Pipes vs. TCP/IP Sockets

In a fast local area network (LAN) environment, Transmission Control Protocol/Internet Protocol (TCP/IP) Sockets and Named Pipes clients are comparable in terms of performance. However, the performance difference between the TCP/IP Sockets and Named Pipes clients becomes apparent with slower networks, such as across wide area networks (WANs) or dial-up networks. This is because of the different ways the interprocess communication (IPC) mechanisms communicate between peers.

For named pipes, network communications are typically more interactive. A peer does not send data until another peer asks for it using a read command. A network read typically involves a series of peek named pipes messages before it begins to read the data. These can be very costly in a slow network and cause excessive network traffic, which in turn affects other network clients.

It is also important to clarify if you are talking about local pipes or network pipes. If the server application is running locally on the computer running an instance of Microsoft® SQL Server™ 2000, the local Named Pipes protocol is an option. Local named pipes runs in kernel mode and is extremely fast.

For TCP/IP Sockets, data transmissions are more streamlined and have less overhead. Data transmissions can also take advantage of TCP/IP Sockets performance enhancement mechanisms such as windowing, delayed acknowledgements, and so on, which can be very beneficial in a slow network. Depending on the type of applications, such performance differences can be significant.

TCP/IP Sockets also support a backlog queue, which can provide a limited smoothing effect compared to named pipes that may lead to pipe busy errors when you are attempting to connect to SQL Server.

In general, sockets are preferred in a slow LAN, WAN, or dial-up network, whereas named pipes can be a better choice when network speed is not the issue, as it offers more functionality, ease of use, and configuration options.

For more information about TCP/IP, see the Microsoft Windows NT® documentation.

See Also

Client Net-Libraries and Network Protocols

Optimizing Database Performance

Optimizing Application Performance Using Efficient Data Retrieval

One of the capabilities of the SQL language is its ability to filter data at the server so that only the minimum data required is returned to the client. Using these facilities minimizes expensive network traffic between the server and client. This means that WHERE clauses must be restrictive enough to retrieve only the data that is required by the application.

It is always more efficient to filter data at the server than to send it to the client and filter it in the application. This also applies to columns requested from the server. An application that issues a SELECT * FROM... statement requires the server to return all column data to the client, whether or not the client application has bound these columns for use in program variables. Selecting only the necessary columns by name avoids unnecessary network traffic. This also makes your application more robust in the event of table definition changes, because newly added columns are not returned to the client application.

Performance also depends on how your application requests a result set from the server. In an application using Open Database Connectivity (ODBC), statement options set prior to executing a query determine how the application requests a result set from the server. When you leave the statement options at default values, Microsoft® SQL Server™ 2000 sends the result set the most efficient way.

SQL Server assumes that your application will fetch all the rows from a default result set immediately. Therefore, your application must buffer any rows that are not used immediately but may be needed later. This buffering requirement makes it especially important for you to specify (by using Transact-SQL) only the data you need.

It may seem economical to request a default result set and fetch rows only as your application logic or your application user needs them, but this is false economy. Unfetched rows from a default result set can tie up your connection to the server, blocking other work in the same transaction. Additionally, unfetched rows from a default result set can cause SQL Server to hold locks at the server, possibly preventing other users from updating. This concurrency problem may

not show up in small-scale testing, but it can appear later when the application is deployed. Therefore, immediately fetch all rows from a default result set. For more information, see Understanding and Avoiding Blocking.

Some applications cannot buffer all the data they request from the server. For example, an application that queries a large table and allows the user to specify the selection criteria may return no rows or millions of rows. The user is unlikely to want to see millions of rows. Instead, the user is more likely to reexecute the query with narrower selection criteria. In this case, fetching and buffering millions of rows only to have them thrown away by the user wastes time and resources.

For these applications, SQL Server offers server cursors that allow an application to fetch a small subset or block of rows from an arbitrarily large result set. If the user wants to see other records from the same result set, a server cursor allows the application to fetch any other block of rows from the result set, including the next *n* rows, the previous *n* rows, or *n* rows starting at a certain row number in the result set. SQL Server does the work to fulfill each block fetch request only as needed, and SQL Server does not normally hold locks between block fetches on server cursors.

Server cursors also allow an application to do a positioned update or delete of a fetched row without having to figure out the source table and primary key of the row. If the row data changes between the time it is fetched and the time the update is requested, SQL Server detects the problem and prevents a lost update.

However, the features of server cursors come at a cost. If all the results from a given query are going to be used in your application, a server cursor is always going to be more expensive than a default result set. A default result set always requires only one roundtrip between client and server, whereas each call to fetch a block of rows from a server cursor results in a roundtrip. Moreover, server cursors consume resources on the server, and there are restrictions on the SELECT statements that can be used with some types of cursor. For example, KEYSET cursors are restricted to using tables with unique indexes only, while KEYSET and STATIC cursors make heavy use of temporary storage at the server. For these reasons, only use server cursors when your application needs their features. If a particular task requests a single row by primary key, use a default result set. If another task requires an unpredictably large or updatable result set, use a server cursor and fetch rows in reasonably sized blocks (for

example, one screen of rows at a time). Additionally, where possible, make use of Fast Forward-only cursors with auto-fetch. These cursors can be used to retrieve small result sets with only one roundtrip between the client and server, similar to a default result set. For more information, see Fast Forward-only Cursors.

See Also

Cursors

SELECT

Optimizing Database Performance

Effects of Transactions and Batches on Application Performance

A primary goal of using Transact-SQL appropriately is to reduce the amount of data transferred between server and client. Reducing the amount of data transferred will usually reduce the time it takes to accomplish a logical task or transaction. Long-running transactions can be fine for a single user, but they scale poorly to multiple users. To support transactional consistency, the database must hold locks on shared resources from the time they are first acquired within the transaction until the transaction commits. If other users need access to the same resources, they must wait. As individual transactions get longer, the queue and other users waiting for locks gets longer and system throughput decreases. Long transactions also increase the chances of a deadlock, which occurs when two or more users are simultaneously waiting on locks held by each other. For more information, see Deadlocking.

Techniques you can use to reduce transaction duration include:

- Committing transactional changes as soon as possible within the requirements of the application.

  Applications often perform large batch jobs, such as month-end summary calculations, as a single unit of work (and thus one transaction). With many of these applications, individual steps of the job can be committed without compromising database consistency. Committing changes as quickly as possible means that locks are released as quickly as possible.

- Taking advantage of Microsoft® SQL Server™ 2000 statement batches.

  Statement batches are a way of sending multiple Transact-SQL statements from the client to SQL Server at one time, thereby reducing the number of network roundtrips to the server. If the statement batch contains multiple SELECT statements, the server will return multiple result sets to the client in a single data stream.

- Using parameter arrays for repeated operations.

  For example, the Open Database Connectivity (ODBC)

SQLParamOptions function allows multiple parameter sets for a single Transact-SQL statement to be sent to the server in a batch, again reducing the number of roundtrips.

SQL Profiler can be used to monitor, filter, and capture all calls sent from client applications to SQL Server. It will often reveal unexpected application overhead due to unnecessary calls to the server. SQL Profiler can also reveal opportunities for placing statements that are currently being sent separately to the server in batches. For more information, see Monitoring with SQL Profiler.

See Also

Batches

Coding Efficient Transactions

Optimizing Database Performance

Effects of Stored Procedures on Application Performance

All well-designed Microsoft® SQL Server™ 2000 applications should use stored procedures. This is true whether or not the business logic of the application is written into stored procedures. Even standard Transact-SQL statements with no business logic component gain a performance advantage when packaged as stored procedures with parameters. Transact-SQL statements compiled into stored procedures can save a significant amount of processing at execution time. For more information, see Stored Procedures.

Another advantage of stored procedures is that client execution requests use the network more efficiently than equivalent Transact-SQL statements sent to the server. For example, suppose an application needs to insert a large binary value into an **image** data column. To send the data in an INSERT statement, the application must convert the binary value to a character string (doubling its size), and then send it to the server. The server then converts the value back into a binary format for storage in the **image** column. In contrast, the application can create a stored procedure of the form:

```
CREATE PROCEDURE P(@p1 image) AS INSERT T VALUES (@p
```

When the client application requests an execution of procedure **P**, the **image** parameter value will stay in binary format all the way to the server, thereby saving processing time and network traffic.

SQL Server stored procedures can provide even greater performance gains when they include business services logic because it moves the processing to the data, rather than moving the data to the processing.

Optimizing Database Performance

Understanding and Avoiding Blocking

Blocking happens when one connection from an application holds a lock and a second connection requires a conflicting lock type. This forces the second connection to wait, blocked on the first. One connection can block another connection, regardless of whether they emanate from the same application or separate applications on different client computers.

Note Some of the actions needing locking protection may not be obvious, for example, locks on system catalog tables and indexes.

Most blocking problems happen because a single process holds locks for an extended period of time, causing a chain of blocked processes, all waiting on other processes for locks.

Common blocking scenarios include:

- Submitting queries with long execution times.

  A long-running query can block other queries. For example, a DELETE or UPDATE operation that affects many rows can acquire many locks that, whether or not they escalate to a table lock, block other queries. For this reason, you generally do not want to intermix long-running decision support queries and online transaction processing (OLTP) queries on the same database. The solution is to look for ways to optimize the query, by changing indexes, breaking a large, complex query into simpler queries, or running the query during off hours or on a separate computer.

  One reason queries can be long-running, and hence cause blocking, is if they inappropriately use cursors. Cursors can be a convenient method for navigating through a result set, but using them may be slower than set-oriented queries.

- Canceling queries that were not committed or rolled back.

  This can happen if the application cancels a query; for example, using the Open Database Connectivity (ODBC) **sqlcancel** function without also issuing the required number of ROLLBACK and COMMIT statements. Canceling the query does not automatically roll back or

commit the transaction. All locks acquired within the transaction are retained after the query is canceled. Applications must properly manage transaction nesting levels by committing or rolling back canceled transactions.

- Applications that are not processing all results to completion.

  After sending a query to the server, all applications must immediately fetch all result rows to completion. If an application does not fetch all result rows, locks may be left on the tables, blocking other users. If you are using an application that transparently submits Transact-SQL statements to the server, the application must fetch all result rows. If it does not (and if it cannot be configured to do so), you may be unable to resolve the blocking problem. To avoid the problem, you can restrict these applications to a reporting or decision-support database.

- Distributed client/server deadlocks.

  Unlike a conventional deadlock, a distributed deadlock cannot be automatically detected by Microsoft® SQL Server™ 2000. A distributed client/server deadlock may occur if the application opens more than one connection to SQL Server and submits a query asynchronously.

  For example, a single client application thread has two open connections. It asynchronously starts a transaction and issues a query on the first connection. The application then starts another transaction, issues a query on another connection, and waits for the results. When SQL Server returns results for one of the connections, the application starts to process them. The application processes the results until no more results are available because the query generating the results is blocked by the query executed on the other connection. At this point, the first connection is blocked, waiting indefinitely for more results to process. The second connection is not blocked on a lock, but tries to return results to the application. However, because the application is blocked, waiting for results on the first connection, the results for the second connection are not processed.

  To avoid this problem, use either:

- A query time-out for each query.

- A lock time-out for each query. For more information, see Customizing the Lock Time-out.

- A bound connection. For more information, see Using Bound Connections.

SQL Server is essentially a puppet of the client application. The client application has almost total control over (and responsibility for) the locks acquired on the server. Although the SQL Server lock manager automatically uses locks to protect transactions, this is directly instigated by the query type sent from the client application and the way the results are processed. Therefore, resolution of most blocking problems involves inspecting the client application.

A blocking problem frequently requires both the inspection of the exact SQL statements submitted by the application and the exact behavior of the application regarding connection management, processing of all result rows, and so on. If the development tool does not allow explicit control over connection management, query time-out, processing of results, and so on, blocking problems may not be resolvable.

Guidelines for designing applications to avoid blocking include:

- Do not use or design an application that allows users to fill in edit boxes that generate a long-running query. For example, do not use or design an application that prompts the user for inputs but rather allows certain fields to be left blank or a wildcard to be entered. This may cause the application to submit a query with an excessive running time, thereby causing a blocking problem.

- Do not use or design an application that allows user input within a transaction.

- Allow for query cancellation.

- Use a query or lock time out to prevent a runaway query and avoid distributed deadlocks.
- Immediately fetch all result rows to completion.
- Keep transactions as short as possible.
- Explicitly control connection management.
- Stress test the application at the full projected concurrent user load.

See Also

Deadlocking

Locking

Optimizing Database Performance

Optimizing Distributed Queries

Microsoft® SQL Server™ 2000 distributed queries allow users to reference remote tables and rowsets as though they are local tables by using SELECT, INSERT, UPDATE, and DELETE statements. Distributed queries cause data to be retrieved across the network when data sources are located on remote computers. Therefore, SQL Server performs two types of optimization specific to distributed queries to improve performance:

- Remote query execution used with OLE DB SQL Command Providers.
- Indexed access used with OLE DB Index Providers.

An OLE DB provider is considered to be a SQL Command Provider if the OLE DB provider meets the following minimum requirements:

- Supports the **Command** object and all of its mandatory interfaces.
- Supports DBPROPVAL SQL SUBMINIMUM Syntax, or SQL-92 at Entry level or higher, or ODBC at Core level or higher. The provider should expose this dialect level through the DBPROP_SQLSUPPORT OLE DB property.

An OLE DB Provider is considered to be an Index Provider if the OLE DB provider meets the following minimum requirements:

- Supports the **IDBSchemaRowset** interface with the TABLES, COLUMNS and INDEXES schema rowsets.
- Supports opening a rowset on an index using **IOpenRowset** by specifying the index name and the corresponding base table name.
- The Index object should support all its mandatory interfaces: **IRowset**, **IRowsetIndex**, **IAccessor**, **IColumnsInfo**, **IRowsetInfo**, and **IConvertTypes**.

- Rowsets opened against the indexed base table (using **IOpenRowset**) should support the **IRowsetLocate** interface for positioning on a row based off a bookmark retrieved from the index.

Remote Query Execution

SQL Server attempts to delegate as much of the evaluation of a distributed query to the SQL Command Provider as possible. An SQL query that accesses only the remote tables stored in the provider's data source is extracted from the original distributed query and executed against the provider. This reduces the number of rows returned from the provider and allows the provider to use its indexes in evaluating the query.

Considerations that affect how much of the original distributed query gets delegated to the SQL Command Provider include:

- The dialect level supported by the SQL Command Provider

  SQL Server delegates operations only if they are supported by the specific dialect level. The dialect levels from highest to lowest are: SQL Server, SQL-92 Entry level, ODBC core, and Jet. The higher the dialect level, the more operations SQL Server can delegate to the provider.

  Note The SQL Server dialect level is used when the provider corresponds to a SQL Server linked server.

  Each dialect level is a superset of the lower levels. Therefore, if an operation is delegated to a particular level, then it is also delegated to all higher levels.

  Queries involving the following are never delegated to a provider and are always evaluated locally:

  - **bit**

  - **uniqueidentifier**

  The following operations/syntactic elements are delegated to the dialect

level indicated (and all higher levels):

- SQL Server: Outer join, CUBE, ROLLUP, modulo operator (%), bit-wise operators, string functions, and arithmetic system functions.

- SQL-92 Entry Level: UNION, and UNION ALL.

- ODBC Core: Aggregation functions with DISTINCT, and string constants.

- Jet: Aggregate functions without DISTINCT, sorting (ORDER BY), inner joins, predicates, subquery operators (EXISTS, ALL, SOME, IN), DISTINCT, arithmetic operators not mentioned in higher levels, constants not mentioned in higher levels, and all logical operators.

For example, all operations except those involving CUBE, ROLLUP, outer join, modulo operator (%), bit-wise operators, string functions, and arithmetic system functions are delegated to a SQL-92 Entry level provider that is not also SQL Server.

- Collation compatibility

  For a distributed query, the comparison semantics for all character data is defined by the character set and sort order of the local SQL Server. Microsoft SQL Server 2000 supports multiple collations, which can be different for each column; each character value has an associated collation property. SQL Server 2000 interprets the collation property of character data from a remote data source and treats it accordingly. For more information on the collation of remote columns, see Collations in Distributed Queries.

  SQL Server can delegate comparisons and ORDER BY operations on character columns to a provider only if it can determine that:

  - The underlying data source uses the collation sequence and character set of the column.

    - The character comparison semantics follow the SQL-92 (and SQL Server) standard.
- Following the table in the Collations in Distributed Queries topic, SQL Server will determine a collation for each column. If the remote data source supports that collation, then the provider is considered collation compatible.

- Other SQL support considerations

  The following SQL syntax elements are not dictated by the SQL dialect levels:

    - Nested query support

      If the provider supports nested queries (subqueries), then SQL Server can delegate these operations to the provider. Because nested query support cannot be automatically determined from OLE DB properties, the system administrator should set the **NestedQueries** provider option to indicate to SQL Server that the provider supports nested queries.

    - Parameter marker support

      If the provider supports parameterized query execution by using the ? parameter marker in a query, then SQL Server can delegate parameterized query execution to the provider. Because nested query support cannot be automatically determined from OLE DB properties, the system administrator should set the **DynamicParameters** provider option to indicate to SQL Server that the provider supports nested queries.

Indexed Access

SQL Server can use execution strategies that involve using the indexes of the Index provider to evaluate predicates and perform sorting operations against remote tables. Set the **IndexAsAccessPath** provider option to enable indexed access against a provider.

Additionally, when using indexes involving character columns, set the **collation compatible** linked server configuration option to **true** for the corresponding linked server. For more information, see sp_serveroption.

Note Graphically display the execution plan using SQL Query Analyzer to determine the execution plan for a given distributed query. When remote query execution is employed in the execution plan, it is represented using the Remote Query logical and physical operator. The argument of this operator shows the remotely executed query.

See Also

Configuring OLE DB Providers for Distributed Queries

Subquery Fundamentals

Optimizing Database Performance

Optimizing Utility and Tool Performance

Three operations performed on a production database that can benefit from optimal performance include:

- Backup and restore operations.
- Bulk copying data into a table.
- Performing database console command (DBCC) operations.

Generally, these operations do not need to be optimized. However, in situations where performance is critical, techniques can be used to fine-tune performance.

Optimizing Database Performance

Optimizing Backup and Restore Performance

Microsoft® SQL Server™ 2000 offers several methods for increasing the speed of backup and restore operations:

- Using multiple backup devices allows backups to be written to all devices in parallel. Similarly, the backup can be restored from multiple devices in parallel. Backup device speed is one potential bottleneck in backup throughput. Using multiple devices can increase throughput in proportion to the number of devices used. For more information, see Using Multiple Media or Devices.

- Using a combination of database, differential database, and transaction log backups to minimize the time necessary to recover from a failure. Differential database backups reduce the amount of transaction log that must be applied to recover the database. This is normally faster than creating a full database backup. For more information, see Logged and Minimally Logged Bulk Copy Operations.

- Logged and Minimally Logged Bulk Copy Operations

Optimizing Database, Differential Database, and File Backup Performance

Creating a database backup comprises two steps:

- Copying the data from the database files to the backup devices.

- Copying the portion of the transaction log needed to roll forward the database to a consistent state to the same backup devices.

Creating a differential database backup comprises the same two steps as creating a database backup, except only the data that has changed is copied (although all database pages need to be read to determine this).

Backing up a database file consists of one step: Copying the data from the database file to the backup devices.

The database files used to store the database are sorted by a disk device, and a reader thread is assigned to each device. The reader thread reads the data from the database files. A writer thread is assigned to each backup device. The writer thread writes data to the backup device. Parallel read operations can be increased by spreading the database files among more logical drives. Similarly, parallel write operations can be increased by using more backup devices.

Generally, the bottleneck will be either the database files or the backup devices. If the total read throughput is greater than the total backup device throughput, then the bottleneck is on the backup device side. Adding more backup devices (and SCSI controllers, as necessary) can improve performance. However, if the total backup throughput is greater than the total read throughput, then increase the read throughput by adding, for example, more database files on devices or by using more disks in the RAID (redundant array of independent disks) device.

Optimizing Transaction Log Backup Performance

Creating a transaction log backup comprises only a single step: copying the portion of the log not yet backed up to the backup devices. Even though there may be multiple transaction log files, the transaction log is logically one stream read sequentially by one thread.

A reader/writer thread is assigned to each backup device. Higher performance is achieved by adding more backup devices.

The bottleneck can be either the disk device containing the transaction log files or the backup device, depending on their relative speed and the number of backup devices used. Adding more backup devices will scale linearly until the capacity of the disk device containing the transaction log files is reached, whereupon no further gains are possible without increasing the speed of the disk devices containing the transaction log, for example, by using disk striping.

Optimizing Restore Performance

Restoring a database or differential database backup comprises four steps:

- Creating the database and transaction log files if they do not already

exist.

- Copying the data from the backup devices to the database files.
- Copying the transaction log from the transaction log files.
- Rolling forward the transaction log, and then restarting recovery if necessary.

Applying a transaction log backup comprises two steps:

- Copying data from the backup devices to the transaction log file.
- Rolling forward the transaction log.

Restoring a database file comprises two steps:

- Creating any missing database files.
- Copying the data from the backup devices to the database files.

If the database and transaction log files do not already exist, they must be created before data can be restored to them. The database and transaction log files are created and the file contents initialized to zero. Separate worker threads create and initialize the files in parallel. The database and transaction log files are sorted by disk device, and a separate worker thread is assigned to each disk device. Because creating files and initializing them requires very high throughput, spreading the files evenly across the available logical drives yields the highest performance.

Copying the data and transaction log from the backup devices to the database and transaction log files is performed by reader/writer threads; one thread is assigned to each backup device. Performance is limited by either the ability of the backup devices to deliver the data or the ability of the database and transaction log files to accept the data. Therefore, performance increases linearly with the number of backup devices added, until the ability of the database or

transaction log files to accept the data is reached.

The performance of rolling forward a transaction log is fixed and cannot be further optimized apart from using a faster computer.

Optimizing Tape Backup Device Performance

There are four variables that affect tape backup device performance and that allow SQL Server backup and restore performance operations to roughly scale linearly as more tape devices are added:

- Software data block size
- Number of tape devices that share a small computer system interface (SCSI) bus
- Tape device type

The software data block size is computed for optimal performance by SQL Server and should not be altered.

Many high-speed tape drives perform better if they have a dedicated SCSI bus for each tape drive used. Drives whose native transfer rate exceeds 50 percent of the SCSI bus speed must be on a dedicated SCSI bus.

For more information about settings that affect tape drive performance, see the tape drive vendor's documentation.

IMPORTANT Never place a tape drive on the same SCSI bus as disks or a CD-ROM drive. The error-handling actions for these devices are mutually incompatible.

Optimizing Disk Backup Device Performance

Raw I/O speed of the disk backup device affects disk backup device performance and allows SQL Server backup and restore performance operations to roughly scale linearly as multiple disk devices are added.

The use of RAID (redundant array of independent disks) for a disk backup device needs to be carefully considered. For example, RAID 5 has low write

performance, approximately the same speed as for a single disk (due to having to maintain parity information). Additionally, the raw speed of appending data to a file is significantly slower than the raw device write speed.

If the backup device is heavily striped, such that the maximum write speed to the backup device greatly exceeds the speed at which it can append data to a file, then it can be appropriate to place several logical backup devices on the same stripe set. In other words, backup performance can be increased by placing several backup media families on the same logical drive. However, an empirical approach is required to determine if this is a gain or a loss for each environment. Usually, it is better to place each backup device on a separate disk device.

Generally, on a SCSI bus, only a few disks can be operated at maximum speed, although Ultra-wide and Ultra-2 buses can handle more. However, careful configuration of the hardware is likely to be needed to obtain optimal performance.

For more information about settings that affect disk performance, see the disk vendor's documentation.

Data Compression

Modern tape drives have built-in hardware data compression that can significantly increase the effective transfer rate of data to the drive. Data compression increases the effective transfer rate to the tape drives over what can be achieved with hardware compression disabled. The compressibility of the real data in the database depends both on the data itself and on the tape drives used. Typical data compression ratios range from 1.2:1 to 2:1 for a wide range of databases. This compression ratio is typical of data in a wide variety of business applications, although some databases can have higher or lower compression ratios. For example, a database consisting largely of images that are already compressed will not be compressed further by the tape drives. For more information about data compression, see the tape-drive vendor's documentation.

By default, SQL Server supports hardware compression, although this procedure can be disabled by using the 3205 trace flag. Disabling hardware compression can, in rare circumstances, improve backup performance. For example, if the data is already fully compressed, disabling hardware compression prevents the tape device from wasting time trying to compress the data further.

For more information about trace flags, see Trace Flags.

Amount of Data Transferred to Tape

Creating a database backup captures only the portion of the database containing real data; unused space is not backed up. The result is faster backup operations.

Although SQL Server 2000 databases can be configured to grow automatically as needed, you can continue to reserve space within the database to guarantee that this space is available. Reserving space within the database does not adversely affect backup throughput or the overall time needed to back up the database.

See Also

Handling Large Mission-Critical Environments

SQL Server: Backup Device Object

SQL Server: Databases Object

Optimizing Database Performance

Optimizing Bulk Copy Performance

To bulk copy data as fast as possible, several options are available to specify how data should be bulk copied into Microsoft® SQL Server™ 2000 using the **bcp** utility or BULK INSERT statement, including:

- Using logged and nonlogged bulk copies.
- Using the **bcp** utility for parallel data loading.
- Controlling the locking behavior.
- Using batches.
- Ordering data files.

For more information, see Bulk Copy Performance Considerations.

Note If possible, use the BULK INSERT statement rather than the **bcp** utility to bulk copy data into SQL Server. The BULK INSERT statement is faster than the **bcp** utility.

Two factors determine which of these options can or should be used to increase the performance of bulk-copy operations:

- Amount of existing data in the table compared to the amount of data to be copied into the table.
- Number and type of indexes on the table.

Additionally, these factors depend on whether data is bulk copied into a table from a single client or in parallel from multiple clients.

Loading Data into an Empty Table from a Single Client

When you are loading data into an empty table from a single client, it is recommended that you specify:

- The **TABLOCK** hint. This causes a table-level lock to be taken for the duration of the bulk-copy operation.

- A large, batch size, using the **ROWS_PER_BATCH** hint. A single batch representing the size of the entire file is recommended.

- A nonlogged bulk-copy operation. Transaction log backups should not be created after performing a nonlogged operation. For more information, see Logged and Nonlogged Bulk Copy Operations.

Additionally, if your table has a clustered index and the data in the data file is ordered to match the clustered index key columns, bulk copy the data into the table with the clustered index already in place and specify the **ORDER** hint. This is significantly faster than creating the clustered index after the data is copied into the table.

If nonclustered indexes are also present on the table, drop these before copying data into the table. It is generally faster to bulk copy data into a table without nonclustered indexes, and then to re-create the nonclustered indexes, rather than bulk copy data into a table with the nonclustered indexes in place.

Loading Data into a Nonempty Table from a Single Client

When you are copying data into a table that has existing data, the recommendation to perform the bulk copy operation with the indexes in place depends on the amount of data to be copied into the table compared to the amount of existing data already in the table. As the percentage of data to be copied into the table increases (based on the amount of existing data in the table), the faster it is to drop all indexes on the table, perform the bulk copy operation, and then re-create the indexes after the data is loaded.

As a general guide, the following table shows suggested figures for the amount of data to be added to a table for various types of indexes. If you exceed these percentages, you may find it faster to drop and re-create the indexes.

Indexes	Amount of data added
Clustered index only	30%
Clustered and one nonclustered index	25%
Clustered and two nonclustered indexes	25%
Single nonclustered index only	100%
Two nonclustered indexes	60%

Loading Data in Parallel from Multiple Clients

If SQL Server is running on a computer with more than one processor and the data to be bulk copied into the table can be partitioned into separate data files, then it is recommended that data be loaded into the same table from multiple clients in parallel, thereby improving the performance of the bulk-copy operation. For example, when bulk copy loading from eight clients into one table, each client must have one input data file containing a portion of the partitioned data. To achieve maximum performance, the batch size specified for each client should be the same as the size of the client data file.

When copying data into a table from multiple clients, consider that:

- All indexes on the table must be dropped first, and then re-created on the table. Consider re-creating the secondary indexes in parallel by creating each secondary index from a separate client at the same time.
- Using ordered data and the **ORDER** hint will not affect performance because the clustered index is not present during the load.
- The data must be partitioned into multiple input files, one file per client.

As with bulk-copy operations from a single client, specify:

- The **TABLOCK** hint. This causes a table-level lock to be taken for the duration of the bulk-copy operation.
- A large, batch size, using the **ROWS_PER_BATCH** hint. A single

batch representing the size of the entire client file is recommended for each client.

- Set the database option **select into/bulkcopy** to true to enable nonlogged operations.

Copying Data Between Computers Running SQL Server

If data is being copied from one computer running an instance of SQL Server to another, perform all bulk-copy operations using either native or Unicode native format. For more information, see Using Native, Character, and Unicode Formats.

If the source table has a clustered index or if you intend to bulk copy the data into a table with a clustered index:

1. Bulk copy the data out of the source table specifying a SELECT statement and an appropriate ORDER BY clause to create an ordered data file.

2. Use the **ORDER** hint when bulk copying the data into SQL Server. For more information, see Ordered Data Files.

See Also

Using bcp and BULK INSERT

SQL Server: Databases Object

Optimizing Database Performance

Optimizing DBCC Performance

The database console command (DBCC) tends to be both CPU and disk intensive because DBCC must read each data page that requires checking from disk into memory (unless the data page is already cached in memory). Running DBCC when there is a lot of activity on the system, such as intensive query processing, impairs DBCC performance because less memory is available and Microsoft® SQL Server™ 2000 is forced to spool data pages to the **tempdb** database. Therefore, DBCC statements execute faster if more memory is made available for DBCC processing because more of the database can be cached.

Because the **tempdb** database resides on disk, the bottleneck from I/O operations as data is written to and from disk impairs performance. Regardless of system activity, running DBCC against large databases (relative to the size of available memory) causes spooling to the **tempdb** database. Therefore, it is recommended that the **tempdb** database be placed on a separate fast disk or disks, such as a RAID (redundant array of independent disks) device, from user databases. For more information, see ALTER DATABASE and RAID.

Note Executing DBCC CHECKDB automatically executes DBCC CHECKTABLE for each table in the database and DBCC CHECKALLOC, eliminating the need to run them separately.

See Also

DBCC

SQL Server: Databases Object

Optimizing Database Performance

Optimizing Server Performance

Microsoft® SQL Server™ 2000 automatically tunes many of the server configuration options, therefore requiring little, if any, tuning by a system administrator. Although these configuration options can be modified by the system administrator, it is generally recommended that these options be left at their default values, allowing SQL Server to automatically tune itself based on run-time conditions.

However, if necessary, the following components can be configured to optimize server performance:

- SQL Server Memory
- I/O subsystem
- Microsoft Windows NT® options

Optimizing Database Performance

Optimizing Server Performance Using Memory Configuration Options

The memory manager component of Microsoft® SQL Server™ 2000 eliminates the need for manual management of the memory available to SQL Server. When SQL Server starts, it dynamically determines how much memory to allocate based on how much memory the operating system and other applications are currently using. As the load on the computer and SQL Server changes, so does the memory allocated. For more information, see Memory Architecture.

The following server configuration options can be used to configure memory usage and affect server performance:

- **min server memory**
- **max server memory**
- **max worker threads**
- **index create memory**
- **min memory per query**

The **min server memory** server configuration option can be used to ensure that SQL Server starts with at least the minimum amount of allocated memory and does not release memory below this value. This configuration option can be set to a specific value based on the size and activity of your SQL Server. Always set the **min server memory** server configuration option to some reasonable value to ensure that the operating system does not request too much memory from SQL Server, affecting SQL Server performance.

The **max server memory** server configuration option can be used to specify the maximum amount of memory SQL Server can allocate when it starts and while it runs. This configuration option can be set to a specific value if you know there are multiple applications running at the same time as SQL Server and you want to guarantee that these applications have sufficient memory to run. If these other

applications, such as Web or e-mail servers, request memory only as needed, then do not set the **max server memory** server configuration option, because SQL Server will release memory to them as needed. However, applications often use whatever memory is available when they start and do not request more if needed. If an application that behaves in this manner runs on the same computer at the same time as SQL Server, set the **max server memory** server configuration option to a value that guarantees that the memory required by the application is not allocated by SQL Server.

Do not set **min server memory** and **max server memory** server configuration options to the same value, thereby fixing the amount of memory allocated to SQL Server. Dynamic memory allocation will give you the best overall performance over time. For more information, see Server Memory Options.

The **max worker threads** server configuration option can be used to specify the number of threads used to support the users connected to SQL Server. The default setting of 255 can be slightly too high for some configurations, depending on the number of concurrent users. Because each worker thread is allocated, even if it is not being used (because there are fewer concurrent connections than allocated worker threads), memory resources that can be better utilized by other operations, such as the buffer cache, can be unused. Generally, this configuration value should be set to the number of concurrent connections, but cannot exceed 1,024. For more information, see max worker threads Option.

Note The max worker threads server configuration option has no effect when SQL Server is running on Microsoft Windows® 95 or Microsoft Windows 98.

The **index create memory** server configuration option controls the amount of memory used by sort operations during index creation. Creating an index on a production system is usually an infrequently performed task, often scheduled as a job to execute during off-peak time. Therefore, when creating indexes infrequently and during off-peak time, increasing this number can improve the performance of index creation. Keep the **min memory per query** configuration option at a lower number, however, so the index creation job will still start even if all the requested memory is not available. For more information, see index create memory Option.

The **min memory per query** server configuration option can be used to specify the minimum amount of memory that will be allocated for the execution of a query. When there are many queries executing concurrently in a system,

increasing the value of the **min memory per query** can help improve the performance of memory-intensive queries, such as substantial sort and hash operations. However, do not set the **min memory per query** server configuration option too high, especially on very busy systems, because the query will have to wait until it can secure the minimum memory requested or until the value specified in the **query wait** server configuration option is exceeded. If more memory is available than the specified minimum value required to execute the query, the query is allowed to make use of the additional memory, provided that the memory can be used effectively by the query. For more information, see min memory per query Option and query wait Option.

See Also

Monitoring Memory Usage

Optimizing Database Performance

Optimizing Server Performance Using I/O Configuration Options

The following server configuration option can be used to configure I/O usage and affect server performance:

- **recovery interval**

The **recovery interval** server configuration option controls when Microsoft® SQL Server™ 2000 issues a checkpoint in each database. By default, SQL Server determines the best time to perform checkpoint operations. However, to determine if this is the appropriate setting, monitor disk write activity on the database files using Windows NT Performance Monitor. Spikes of activity that cause disk utilization to reach 100 percent can affect performance. Changing this parameter to cause the checkpoint process to occur less often can improve overall performance in this situation. However, continue to monitor performance to determine if the new value has had a positive effect on performance. For more information, see recovery interval Option.

See Also

Monitoring Disk Activity

Optimizing Database Performance

Optimizing Server Performance Using Windows NT Options

You can set Microsoft® Windows NT® or Windows® 2000 options on the server to:

- Maximize throughput.
- Configure server tasking.
- Configure virtual memory.

Maximizing Throughput

SQL Server Setup automatically configures Microsoft® Windows NT® to maximize throughput for network applications. This enables the server to accommodate more connections. Although maximizing throughput for network applications is recommended for Microsoft SQL Server™ 2000, you can change this setting.

If the Full-Text Search feature is installed, the Windows NT Server or Windows 2000 configuration must be set for maximizing throughout for network applications and must not be changed.

Note The Windows NT Server configuration setting does not apply to computers running Windows NT Workstation. For more information, see the Windows NT or Windows 2000 documentation.

Configuring Server Tasking

If you plan to connect to Microsoft® SQL Server™ 2000 from a local client (a client running on the same computer as the server), you can improve processing time by setting up the server to run foreground and background applications with equal priority. SQL Server, which runs as a background application, then runs at equal priority to other applications running in the foreground. For more information, see the Windows NT® or Windows 2000 documentation.

Note When you run SQL Server Setup, server tasking is set to **none** in Microsoft Windows NT 4.0 and **background services** in Microsoft Windows 2000 (the SQL Server default), which gives foreground and background programs equal processor time. You can set the server tasking setting to **maximum** (the Microsoft Windows NT default) or **applications** (the Microsoft Windows 2000 default), which gives foreground applications the most processor time.

Configuring Virtual Memory

Microsoft® Windows NT® or Windows 2000 virtual memory size should be configured based on the services concurrently running on the computer. When you are running Microsoft SQL Server™ 2000, consider setting the virtual memory size to 1.5 times the physical memory installed in the computer.

If you have additionally installed the Full-Text Search feature and plan to run the Microsoft Search service so that you can do full-text indexing and querying, consider configuring:

- The virtual memory size to at least 3 times the physical memory installed in the computer.
- The SQL Server **max server memory** server configuration option to 1.5 times the physical memory (half the virtual memory size setting).

If the virtual memory setting is configured too low, then the following Windows NT error can occur:

```
Your system is running low on virtual memory. Please close some appl
```

Note For more information, see the Windows NT or Windows 2000 documentation.