SQL Server Architecture Overview

Microsoft® SQL Server[™] 2000 is a set of components that work together to meet the data storage and analysis needs of the largest Web sites and enterprise data processing systems. The topics in SQL Server Architecture describe how the various components work together to manage data effectively.

Торіс	Description
Features of SQL Server	Highlights the features of Microsoft SQL Server
2000	2000.
Relational Database	Describes the main relational database
<u>Components</u>	components of SQL Server 2000, including the
	database engine itself and the components
	involved in communications between applications
	and the database engine.
Database Architecture	Describes the logical components defined in SQL
	Server databases and how they are physically
	implemented in database files.
<u>Relational Database</u>	Describes the features of the server engine that
Engine Architecture	make it efficient at processing large numbers of
	concurrent requests for data from many users.
<u>Administration</u>	Describes how the easy-to-use tools provided with
<u>Architecture</u>	SQL Server 2000 and the dynamic configuration
	capabilities of SQL Server minimize routine
	administrative tasks.
Replication Architecture	Describes the replication components of SQL
	Server 2000 and how they can be used to
	distribute data between databases.
Data Warehousing and	Describes Data Transformation Services (DTS)
<u>Online Analytical</u>	and Microsoft SQL Server 2000 Analysis
Processing	Services, and how they help in building and
	analyzing a data warehouse or data mart.
<u>Application</u>	Describes how SQL Server 2000 supports the
<u>Development</u>	various database programming APIs, which allow
<u>Architecture</u>	users to build robust database applications.

Implementation Details	Provides implementation details, such as the
	maximum capacities of Transact-SQL statements,
	the ranges of SQL Server configuration options,
	memory usage of SQL Server objects, and the
	differences among the editions of Microsoft SQL
	Server.

Fundamentals of SQL Server 2000 Architecture

Microsoft® SQL Server[™] 2000 is a family of products that meet the data storage requirements of the largest data processing systems and commercial Web sites, yet at the same time can provide easy-to-use data storage services to an individual or small business.

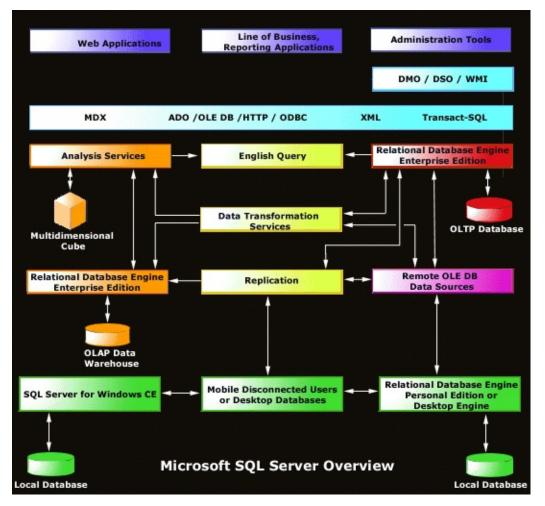
The data storage needs of a modern corporation or government organization are very complex. Some examples are:

- Online Transaction Processing (OLTP) systems must be capable of handling thousands of orders placed at the same time.
- Increasing numbers of corporations are implementing large Web sites as a mechanism for their customers to enter orders, contact the service department, get information about products, and for many other tasks that previously required contact with employees. These sites require data storage that is secure, yet tightly integrated with the Web.
- Organizations are implementing off-the-shelf software packages for critical services such as human resources planning, manufacturing resources planning, and inventory control. These systems require databases capable of storing large amounts of data and supporting large numbers of users.
- Organizations have many users who must continue working when they do not have access to the network. Examples are mobile disconnected users, such as traveling sales representatives or regional inspectors. These users must synchronize the data on a notebook or laptop with the current data in the corporate system, disconnect from the network, record the results of their work while in the field, and then finally reconnect with the corporate network and merge the results of their fieldwork into the corporate data store.

- Managers and marketing personnel need increasingly sophisticated analysis of trends recorded in corporate data. They need robust Online Analytical Processing (OLAP) systems easily built from OLTP data and support sophisticated data analysis.
- Independent Software Vendors (ISVs) must be able to distribute data storage capabilities with applications targeted at individuals or small workgroups. This means the data storage mechanism must be transparent to the users who purchase the application. This requires a data storage system that can be configured by the application, and then tune itself automatically so that the users do not need to dedicate database administrators to constantly monitor and tune the application.

SQL Server 2000 Component Overview

This diagram is an illustration of the relationships between the major components of Microsoft® SQL Server[™] 2000.



SQL Server 2000 provides two fundamental services to applications in a Windows® DNA environment:

• The SQL Server 2000 relational database engine is a modern, highly scalable, highly reliable engine for storing data. The database engine stores data in tables. Each table represents some object of interest to the organization, such as vehicles, employees, or customers. Each table has columns that represent an attribute of the object modeled by the table (such as weight, name, or cost), and rows that represent a single occurrence of the type of object modeled by the table (such as the car

with license plate number ABC-123, or the employee with ID 123456). Applications can submit Structured Query Language (SQL) statements to the database engine, which returns the results to the application in the form of a tabular result set. The specific dialect of SQL supported by SQL Server is called Transact-SQL. Applications can also submit either SQL statements or XPath queries and request that the database engine return the results in the form of an XML document.

The relational database engine is highly scalable. The SQL Server 2000, Enterprise Edition can support groups of database servers that cooperate to form terabyte-sized databases accessed by thousands of users at the same time. The engine is capable of handling the traffic of any Web site in the world. The database engine also tunes itself, dynamically acquiring resources as more users connect to the database, and then freeing the resources as the users log off. This means that the smaller editions of SQL Server can be used for individuals or small workgroups that do not have dedicated database administrators. SQL Server for Windows CE even extends the SQL Server programming model to Windows CE devices used by mobile, disconnected users. Even large Enterprise Edition database servers running in production are easy to administer using the graphical user interface (GUI) administration utilities that are a part of the product.

The relational database engine is highly reliable and capable of running for long periods without down time. Administrative actions that required stopping and starting in earlier versions of the database engine can now be performed while the engine is running, increasing availability. The integration of the database engine with Windows 2000 and Windows NT® failover clustering allows you to define virtual servers that keep running even if one of the physical servers in the node fails. Where appropriate, log shipping can be used to maintain a warm standby server that can replace a production server within minutes of a failure.

The relational database engine is also highly secure. Login authentication can be integrated with Windows Authentication, so that no passwords are stored in SQL Server or sent across the network where they could be read by network sniffers. Sites can set up C2-level auditing of all users accessing a database, and can use Secure Sockets Layer (SSL) encryption to encrypt all data transferred between applications and the database.

The distributed query feature of the database engine allows you to access data from any source of data that can be accessed using OLE DB. The tables of the remote OLE DB data source can be referenced in Transact-SQL statements just like tables that actually reside in a SQL Server database. In addition, the full-text search feature allows you to perform sophisticated pattern matches against textual data stored in SQL Server databases or Windows files.

The relational database engine is capable of storing detailed records of all the transactions generated by the top online transaction processing (OLTP) systems. The database engine can also support the demanding processing requirements for fact tables and dimension tables in the largest online analytical (OLAP) data warehouses.

For more information about the SQL Server 2000 relational database component, see <u>Relational Database Components</u>

 Microsoft SQL Server 2000 Analysis Services provides tools for analyzing the data stored in data warehouses and data marts. Certain analytical processes, such as getting a summary of the monthly sales by product of all the stores in a district, take a long time if run against all the detail records of an OLTP system. To speed up these types of analytical processes, data from an OLTP system is periodically summarized and stored in fact and dimension tables in a data warehouse or data mart. Analysis Services presents the data from these fact and dimension tables as multidimensional cubes that can be analyzed for trends and other information that is important for planning future work. Processing OLAP queries on multidimensional Analysis Services cubes is substantially faster than attempting the same queries on the detail data recorded in OLTP databases. For more information about Analysis Services, see <u>Data Warehousing and Online Analytical Processing</u>.

Application Support

Both the relational database engine and Analysis Services provide native support for the common Windows DNA or Win32 data access interfaces, such as ActiveX® Data Objects (ADO), OLE DB, and Open Database Connectivity (ODBC). Applications can use any of these application programming interfaces (APIs) to send SQL or XML statements to the relational database engine using a native OLE DB provider or ODBC driver. SQL Server 2000 also introduces the ability to use HTTP to send SQL or XML statements to the relational database engine. Applications can use the multidimensional extensions of either ADO or OLE DB to send Multidimensional Expressions (MDX) queries to Analysis Services. Because SQL Server uses the standard Windows DNA data access APIs, the development of SQL Server applications is well supported by the Microsoft application development environments. In addition, interactive query tools, such as Query Analyzer, provide templates, interactive debuggers, and interactive test environments that speed the ability of your programmers to deliver SQL Server applications.

In addition to supporting the data storage and OLAP processing needs of applications, SQL Server 2000 provides a full set of easy to use, graphical administration tools and wizards for creating, configuring, and maintaining databases, data warehouses, and data marts. SQL Server also documents the administration APIs used by the SQL Server tools, giving you the ability to incorporate SQL Server administration functionality directly into your own applications. The SQL Server administration APIs include:

- SQL Distributed Management Objects (SQL-DMO), a set of COM objects that encapsulates the administration functions for all of the entities in the relational database engine and databases.
- Decision Support Objects (DSO), a set of COM objects that encapsulates the administration functions for all of the entities in Analysis Services engine and multidimensional cubes.
- Windows Management Instrumentation (WMI), SQL Server 2000 provides a SQL Server WMI provider that lets WMI applications get information on SQL Server databases and instances.

For more information about developing SQL Server applications, see <u>Application Development Architecture</u>, and <u>SQL Server and XML Support</u>.

Additional Components

SQL Server 2000 provides several components that support important requirements of modern data storage systems. The data storage needs of today's large enterprises are very complex, and go beyond having a single OLTP system integrated with a single data warehouse or data mart. Increasing numbers of field personnel need to load sets of data, disconnect from the network, record their work autonomously during the day, then plug back in to the network and merge their records into the central data store at the end of the day. OLTP systems have to support the needs of both internal employees operating through an intranet and hundreds of thousands of customers placing orders through your Web portal. Keeping data close to the workgroups or even individuals who primarily work on the data, and then replicating the data to a primary data store may minimize the overall processing load of your system.

- SQL Server 2000 replication allows sites to maintain multiple copies of data on different computers in order to improve overall system performance while at the same time making sure the different copies of data are kept synchronized. For example, a department could maintain the department sales data on a departmental server, but use replication to update the sales data in the corporate computer. Several mobile disconnected users can disconnect from the network, work throughout the day, and at the end of the day use merge replication to merge their work records back into the main database. These workers can be using SQL Server Personal Edition on notebook or laptop computers, or using SQL Server replication. SQL Server replication also supports replicating data to data warehouses, and can replicate data to or from any data source that supports OLE DB access. For more information, see <u>Replication Architecture</u>.
- SQL Server 2000 Data Transformation Services (DTS) greatly improves the process of building OLAP data warehouses. Large OLTP databases are finely tuned to support the entry of thousands of business transactions at the same time. OLTP databases are also structured to record the details of every transaction. Trying to perform sophisticated analysis to discover trends in sales over a number of months and years

would require scanning huge numbers of records, and the heavy processing load would drag down the performance of the OLTP databases. Data warehouses and data marts are built from the data in one or more OLTP systems that is extracted and transformed into something more useful for OLAP processing. OLTP detail rows are periodically pulled into a staging database, where they are summarized and the summary data is stored in a data warehouse or data mart. Data Transformation Services supports extracting data from one source of data, performing sometimes complex transformations of the data, and then storing the summarized, transformed data in another data source. The component greatly simplifies the process of extracting data from multiple OLTP systems and building it into an OLAP data warehouse or data mart. For more information, see <u>Transforming OLTP Data to</u> <u>OLAP Data Warehouses</u>.

DTS is not limited to being used to build data warehouses. It can be used any time you have to retrieve data from one data source, perform complex transformations on the data, and then store it in another data source. DTS is also not limited to working with SQL Server databases or Analysis Services cubes, DTS can work with any data source that can be accessed using OLE DB.

- SQL Server 2000 English Query allows you to build applications that can customize themselves to ad hoc user questions. An English Query administrator defines for the English Query engine all of the logical relationships between the tables and columns of a database or the cubes in a data warehouse or data mart. An application can then present the user with a box where she can enter a character string with a question (written in English) about the data in the database or data warehouse. The application passes the string to the English Query engine, which analyzes the string against the relationships defined between the tables or cubes. English Query then returns to the application a SQL statement or MDX (multidimensional expression) query that will return the answer to the user's question. For more information, see <u>SQL Server and English Query</u>.
- Meta Data Services provides facilities for storing, viewing, and

retrieving descriptions of the objects in your applications and system. Meta Data Services supports the MDC Open Information Model (OIM) specification defining a common format for storing descriptions of entities such as tables, views, cubes, or transformations, as well as the relationships between these entities. Application development tools that support OIM can use these descriptions to facilitate rapid development and interchange with other tools and applications. SQL Server components, such as Data Transformation Services packages and Analysis Services databases, can also be stored in the Meta Data Services repository. For more information, see <u>SQL Server 2000 Data</u> <u>Warehouse and OLAP Components</u>.

Using SQL Server 2000

An organization may use the SQL Server 2000 components to perform various tasks, for example:

- Each department might have a departmental SQL Server database server. Each of these servers periodically replicate their data into a central database server that serves the entire organization.
- The organization may have another central database computer that services the organization's Web site, sometimes servicing thousands of queries at once. Some of the Web applications use English Query to allow customers to tailor requests for the data in the Web site database.
- Several employees may be running individual copies of a shrinkwrapped software product that installed a copy of SQL Server Desktop Engine as its data storage component.
- Several other employees in the service department are operating as mobile disconnected users, where they use replication each morning to load their daily schedules into notebook computers or Microsoft Windows CE devices, work in the field all day, then use merge replication at the end of the day to enter their work items back into the central computer.

- Periodically, detailed OLTP data is extracted from the central databases by Data Transformation Services packages that scrub the data and build it into summary data that is then loaded into a data warehouse.
- The senior managers and marketing personnel use Analysis Services to analyze the data warehouse for business trends that indicate possible opportunities that could be exploited or risks that must be minimized.

Features of SQL Server 2000

Microsoft[®] SQL Server[™] 2000 features include:

• Internet Integration.

The SQL Server 2000 database engine includes integrated XML support. It also has the scalability, availability, and security features required to operate as the data storage component of the largest Web sites. The SQL Server 2000 programming model is integrated with the Windows DNA architecture for developing Web applications, and SQL Server 2000 supports features such as English Query and the Microsoft Search Service to incorporate user-friendly queries and powerful search capabilities in Web applications.

• Scalability and Availability.

The same database engine can be used across platforms ranging from laptop computers running Microsoft Windows® 98 through large, multiprocessor servers running Microsoft Windows 2000 Data Center Edition. SQL Server 2000 Enterprise Edition supports features such as federated servers, indexed views, and large memory support that allow it to scale to the performance levels required by the largest Web sites.

• Enterprise-Level Database Features.

The SQL Server 2000 relational database engine supports the features required to support demanding data processing environments. The database engine protects data integrity while minimizing the overhead of managing thousands of users concurrently modifying the database. SQL Server 2000 distributed queries allow you to reference data from multiple sources as if it were a part of a SQL Server 2000 database, while at the same time, the distributed transaction support protects the integrity of any updates of the distributed data. Replication allows you to also maintain multiple copies of data, while ensuring that the separate copies remain synchronized. You can replicate a set of data to multiple, mobile, disconnected users, have them work autonomously, and then merge their modifications back to the publisher.

• Ease of installation, deployment, and use.

SQL Server 2000 includes a set of administrative and development tools that improve upon the process of installing, deploying, managing, and using SQL Server across several sites. SQL Server 2000 also supports a standards-based programming model integrated with the Windows DNA, making the use of SQL Server databases and data warehouses a seamless part of building powerful and scalable systems. These features allow you to rapidly deliver SQL Server applications that customers can implement with a minimum of installation and administrative overhead.

• Data warehousing.

SQL Server 2000 includes tools for extracting and analyzing summary data for online analytical processing. SQL Server also includes tools for visually designing databases and analyzing data using English-based questions.

Integrated with the Internet

The Microsoft® SQL Server[™] 2000 relational database engine includes native support for XML:

- Transact-SQL results can be returned as XML documents to Web or line of business applications using the OLE DB and ADO APIs.
- You can define annotated XDR schemas that represent a logical view of the tables in your database. Web applications can then reference these schemas in XPath queries to build XML documents.
- The SQL Server 2000 includes an ISAPI DLL that allows you to define virtual roots in Microsoft Internet Information Services (IIS) associated with an instance of SQL Server 2000. Internet applications can then compose URL strings that reference a SQL Server 2000 virtual root and contains a Transact-SQL statement. The Transact-SQL statement is sent to the instance of SQL Server 2000 associated with the virtual root, and the result is returned as an XML document.
- XML documents can be added to SQL Server 2000 databases. The OPENXML function can be used to expose the data from an XML document in a rowset, which can be referenced by Transact-SQL statements, such as SELECT, INSERT, or UPDATE.

SQL Server 2000 works with other products to form a stable and secure data store for Internet and intranet networks:

- SQL Server 2000 works with Microsoft Windows® 2000 Server and Microsoft Windows NT® Server security and encryption facilities to implement secure data storage.
- SQL Server 2000 forms a high-performance data storage service for Web applications running under IIS, or accessing the database through a

firewall.

- SQL Server 2000 can be used with Site Server to build and maintain large, sophisticated e-commerce Web sites.
- The SQL Server 2000 TCP/IP Sockets communications support can be integrated with Microsoft Proxy Server to implement secure Internet and intranet communications.

Analysis Services includes features that support the functionality required in many Business to Business, or Business to Consumer Web applications:

- An integrated data mining engine supports data mining analysis of both relational databases and OLAP cubes. The data mining engine is extensible through OLE DB for Data Mining, allowing you to incorporate algorithms from Independent Software Vendors (ISVs) to support extended data mining features.
- Features such as distinct count and OLAP alerts allow you perform actions such as analyzing Web site click-streams to evaluate the effectiveness of your Web interface.

English Query allows Web applications to support users of any skill level entering English language questions about data in either a relational database or OLAP cube. English Query will match the question against a model of the database or cube, and return either a SQL or MDX query to retrieve the proper results.

All of these SQL Server 2000 features are also supported from your line of business applications, allowing you to more easily integrate your Web and line of business applications.

See Also

Communication Components Managing Security SQL Server and XML Support

Scalability and Availability

The same Microsoft® SQL Server[™] 2000 database engine operates on Microsoft Windows® 2000 Professional, Microsoft Windows 2000 Server, Microsoft Windows 2000 Advanced Server, Windows 98, and Windows Millennium Edition. It also runs on all editions of Microsoft Windows NT® version 4.0. The database engine is a robust server that can manage terabytesized databases accessed by thousands of users. Additionally, when running at its default settings, SQL Server 2000 has features such as dynamic self-tuning that let it work effectively on laptops and desktops without burdening users with administrative tasks. SQL Server 2000 Windows CE Edition extends the SQL Server 2000 programming model to mobile Windows CE devices and is easily integrated into SQL Server 2000 environments.

SQL Server 2000 works with Windows NT and Windows 2000 failover clustering to support immediate failover to a backup server in continuous operation. SQL Server 2000 also introduces log shipping, which allows you to maintain a warm standby server in environments with lower availability requirements.

Same Server Across Windows 2000, Windows NT, Windows 98, and Windows Millennium Edition Platforms

The same programming model is shared in all environments, because the SQL Server 2000 database engine runs on Windows NT Workstation, Windows NT Server, Windows 2000 Professional, Windows 2000 Server, Windows 2000 Advanced Server, Windows 2000 Datacenter Server, Windows 98, and Windows Millennium Edition.

In general, an application written for an instance of SQL Server 2000 operating in one environment works on any other instance of SQL Server 2000. The Microsoft Search service is not available on the Windows NT Workstation, Windows 2000 Professional, Windows Millennium Edition, or Windows 98 operating systems. SQL Server databases on those platforms do not support fulltext catalogs and indexes. Applications running on these operating systems can, however, make use of the full-text capabilities if they connect to an instance of SQL Server 2000 on a different computer that supports them. The differences in the behavior of SQL Server 2000 when running on the different operating systems are due mainly to features not supported by Windows Millennium Edition or Windows 98. Generally, these features, such as asynchronous I/O and scatter/gather I/O, do not affect the data or responses given to applications. They just prevent instances of SQL Server running on Windows Millennium or Windows 98 from supporting the same levels of performance as are possible for instances of SQL Server on Windows NT or Windows 2000. Instances of SQL Server on Windows Millennium Edition or Windows 98, however, do not support failover clustering and cannot publish transactional replications.

Federated Database Servers

SQL Server 2000 introduces support for updatable, distributed partitioned views. These views can be used to partition subsets of the rows in a table across a set of instances of SQL Server, while having each instance of SQL Server operate as if it had a full copy of the original table. These partitioned views can be used to spread the processing of one table across multiple instances of SQL Server, each on a separate server. By partitioning all, or many, of the tables in a database, this feature can be used to spread the database processing of a single Web site across multiple servers running SQL Server 2000. The servers do not form a cluster because each server is administered separately from the others. Collections of such autonomous servers are called federations of servers. Federations of servers running SQL Server 2000 are capable of supporting the growth needs of the largest Web sites or enterprise database systems that exist today.

To improve the performance and scalability of federated servers, SQL Server 2000 supports high-speed system area networks such as GigaNet.

Very Large Database Improvements

SQL Server 2000 has high-speed optimizations that support very large database environments. SQL Server version 6.5 and earlier can support databases from 200 GB through 300 GB. SQL Server 2000 and SQL Server version 7.0 can effectively support terabyte-sized databases.

The Transact-SQL BACKUP and RESTORE statements are optimized to read through a database serially and write in parallel to multiple backup devices. Sites can also reduce the amount of data to be backed up by performing differential backups that back up only the data changed after the last backup, or by backing up individual files or file groups. In SQL Server 2000, the time required to run a differential backup has been improved, making it proportional to the amount of data modified since the last backup.

Multiple bulk copy operations can be performed concurrently against a single table to speed data entry. The database console command utility statements are implemented with reduced locking requirements and support for parallel operations on computers with multiple processors, greatly improving their speed.

Operations that create multiple indexes on a table can create them concurrently.

SQL Server 2000 databases map directly to Windows files, simplifying the creation and administration of databases. The database page size is 8-KB, and the size of extents increases to 64 KB, which results in improved I/O.

Improved Query Optimizer

The SQL Server 2000 query optimizer has new access methods to increase the speed of query processing. These improved access methods are often matched to improvements and simplifications in the on-disk data structures in the database:

- The query optimizer uses serial, read-ahead I/O when scanning tables and indexes for improved performance. The optimizer also uses merge and hash algorithms for performing joins.
- The query optimizer natively supports the prepare/execute model of executing SQL statements. When an application executes an SQL statement, the optimizer has efficient algorithms for determining if the same statement has already been executed by any application. If the optimizer finds an existing execution plan for the statement, it saves processing resources by reusing the existing plan instead of compiling a new plan. In systems where many users are running the same application, this can reduce the resources needed to compile SQL statements into execution plans.

Intra-Query Parallelism

When running on servers with multiple multiprocessors, or CPUs, SQL Server

2000 can build parallel execution plans that split the processing of a SQL statement into several parts. Each part can be run on a different CPU and the complete result set built more quickly than if the different parts were executed serially.

Large Memory Support

SQL Server 2000 Enterprise Edition uses the Microsoft Windows 2000 Address Windowing Extensions API to support memory approaching 64 GB of RAM. This allows SQL Server 2000 Enterprise Edition to cache large number of rows in memory, which reduces overhead and speeds its ability to process queries.

Indexed Views

The SQL Server 2000 relational database engine supports creating indexes on views. The result set of the index is materialized at the time the index is created, and is maintained as the underlying base data is modified. Creating an index on a view that performs complex calculations on large amounts of data can speed subsequent queries by orders of magnitude. The performance benefits are not limited to queries that specify the indexed view in their FROM clause, the performance benefits apply to any query that references data covered by the indexed view. This means existing queries can realize performance gains from using the view without having to be recoded to explicitly reference the indexed view. Indexed views substantially improve the performance of large, complex reporting applications that access SQL Server databases.

High Availability

SQL Server 2000 can maintain the extremely high levels of availability required by large Web sites and enterprise systems.

SQL Server 2000 carries forward the SQL Server 7.0 architecture, which has proven to be robust in high-volume Web sites and enterprise systems.

SQL Server 2000 has improved support for Windows NT and Windows 2000 failover clustering. Support for setting up failover clustering is now implemented as a Setup option that is much easier to use than earlier versions of Microsoft SQL Server. SQL Server 2000 also supports up to four nodes in a failover cluster.

SQL Server 2000 introduces log shipping for Web sites and enterprise systems that do not require immediate failover support and can potentially lose some updates. You can create a production database, copy it to a warm standby server, and then use log shipping to feed transaction logs from the production server to the standby at set intervals, such as every 10 minutes. By restoring the logs on the standby, you create a server that can replace the production server in case of a problem. The only data that might be lost would be any modifications made since the last set of logs shipped to the warm standby server. Log shipping can also be used to copy data from a production server to one or more read-only reporting servers, assuming the reporting systems do not have to be kept exactly synchronized with the production server.

See Also

Designing Federated Database Servers Relational Database Engine Architecture Overview Server Scalability Log Shipping Creating a Failover Cluster Using AWE Memory on Windows 2000

Enterprise-Level Database Features

Microsoft® SQL Server[™] 2000 includes several features that support the complex data storage needs of large Web sites and modern, enterprise data processing systems.

Distributed Query

SQL Server 2000 supports referencing heterogeneous OLE DB data sources directly in Transact-SQL statements. Distributed queries allow you to integrate data from several sources with the data in a SQL Server 2000 database.

OLE DB providers return their results as rowsets in a tabular form. SQL Server 2000 supports functions, such as OPENQUERY and OPENDATASOURCE, that return rowsets from OLE DB data sources. These functions can be used in place of a table reference in a Transact-SQL statement. You can also define linked server names that reference an OLE DB data source, and then reference tables from that data source in the FROM clause of Transact-SQL statements, just as you would reference any SQL Server table.

The distributed query capability of SQL Server 2000 supports referencing the OLE DB rowsets in data modification statements such as INSERT, UPDATE, and DELETE, if the OLE DB provider supports updates. The OLE DB rowset modifications are protected by distributed transactions if the OLE DB provider supports the required interfaces.

SQL Server 2000 can also take advantage of OLE DB providers that publish statistics regarding the distribution of data values in the rowsets exposed by the provider. SQL Server 2000 uses this information to build intelligent queries that minimize the numbers of rows the OLE DB provider must return to SQL Server. This improves the speed of distributed query processing.

Dynamic Row-Level Locking

SQL Server 2000 dynamically adjusts the granularity of locking to the appropriate level for each table referenced by a query. When a query references a small number of rows scattered in a large table, the best way to maximize concurrent access to data is to use fine-grained locks such as row locks.

However, if a query references most or all of the rows in a table, the best way to maximize concurrency may be to lock the whole table to minimize the locking overhead and finish the query as quickly as possible.

SQL Server 2000 maximizes overall concurrent access to data by choosing the appropriate locking level for each table in each query. For one query, the database engine may use row-level locking for a large table where few rows are referenced; page-level locking for another large table where many rows on a few pages are referenced; and table-level locking for a small table in which all the rows are referenced.

Full Integrity Protection

SQL Server 2000 fully protects the integrity of its databases. All data modifications are performed in transactions, and each transaction is either wholly committed if it reaches a state of consistency, or completely rolled back if it encounters errors. If a server fails, all uncompleted transactions are automatically rolled back from all SQL Server 2000 databases when the server is restarted.

Distributed Transactions

SQL Server 2000 databases can participate in distributed transactions managed by an X/Open XA compliant transaction manager. This includes distributed transactions spanning multiple SQL Server 2000 databases, and also distributed transactions spanning heterogeneous resource managers. The OLE DB Provider for SQL Server 2000 and the SQL Server 2000 ODBC Driver both support enlistment in distributed transactions.

Transact-SQL scripts and applications can have their local transactions escalated dynamically to distributed transactions if they reference objects on other SQL Server 2000 systems or heterogeneous OLE DB data sources. SQL Server 2000 manages these distributed transactions transparently using the Microsoft Distributed Transaction Coordinator.

Replication

SQL Server 2000 replication allows you to maintain copies of data in multiple sites, sometimes hundreds of sites, using a publish-subscribe metaphor. This

allows sites to locate data close to the users who most frequently access it, while keeping it synchronized with copies in other locations.

SQL Server 2000 supports three types of replication. Snapshot replication copies data or database objects as they exist at a particular time. In transactional replication, Publishers and Subscribers first synchronize their data (typically using a snapshot) and then, as data is modified on the Publisher, the modifications are transmitted to the Subscribers. Merge replication lets multiple Subscribers work autonomously with copies of a set of data, and then later merge their updated versions back to the Publisher. Merge replication supports several methods for resolving conflicts in how different Subscribers modify the same data.

Replication in SQL Server 2000 supports queued updating, which allows transactional and snapshot replication subscribers to modify published data without requiring an active network connection.

SQL Server 2000 Replication introduces transformable subscriptions, which allow subscriptions to use the flexibility and power of Data Transformation Services to map, transform, and filter replicated data.

The usability of replication has been further enhanced, making it very easy to administer. Transactional replication can now be synchronized with backing up and restoring databases, eliminating the need to reconfigure transactional replication. You can browse the Windows 2000 Active Directory for publications, subject to proper permissions. SQL Server 2000 introduces new, improved replication wizards, and supports more centralized recording of Publications and Subscriptions.

See Also

Distributed Query Architecture

Relational Database Engine Architecture Overview

Transactions Architecture

Replication Architecture

Ease of Installation, Deployment, and Use

Many databases capable of supporting all of the processing needs of an enterprise are complex and difficult to administer. Microsoft® SQL Server[™] 2000 includes many tools and features that simplify the process of installing, deploying, managing, and using databases. SQL Server 2000 provides database administrators with all the tools required to fine-tune SQL Server 2000 installations running production online systems. SQL Server 2000 is also capable of operating efficiently on a small, single-user system with minimal administrative overhead.

Dynamic Self-Management

SQL Server 2000 reconfigures itself automatically and dynamically while running. As more users connect to SQL Server 2000, it can dynamically acquire additional resources, such as memory. As the workload falls, SQL Server 2000 frees the resources back to the system. If other applications are started on the server, SQL Server 2000 will detect the additional allocations of virtual memory to those applications, and reduce its use of virtual memory to reduce paging overhead. SQL Server 2000 can also increase or decrease the size of a database automatically as data is inserted or deleted.

Database administrators can control the amount of dynamic reconfiguration in each instance of SQL Server 2000. A small database used by someone not familiar with databases can run with the default configuration settings, in which case it will configure itself dynamically. A large production database monitored by experienced database administrators can be set up to give the administrators full control of configuration.

Complete Administrative Tool Set

SQL Server 2000 offers database administrators several tools for managing their systems:

• SQL Server Enterprise Manager is a snap-in component for Microsoft Management Console (MMC).

MMC supports the management of multiple types of servers from a

single console, such as Microsoft Windows® 2000 Services, Microsoft Internet Information Servers, Microsoft SNA Servers, and instances of SQL Server 2000. An administrator at a single console has the ability to manage all the servers on a worldwide network. SQL Server Enterprise Manager shares a subset of the MMC user interface for Web administration. It presents all SQL Server objects in a hierarchical console tree with an easy-to-use graphical user interface.

• SQL Server Agent allows the definition and scheduling of tasks that run on a scheduled or recurring basis.

It also alerts administrators when certain warning conditions occur, and can even be programmed to take corrective action.

• SQL Profiler offers administrators a sophisticated tool for monitoring and analyzing network traffic to and from a server running SQL Server 2000.

It also profiles server events such as the acquisition of locks.

- SQL Server Performance Monitor integrates SQL Server counters into the Windows Performance Monitor, allowing administrators to monitor and graph the performance of SQL Server with the same tool used to monitor Microsoft Windows NT® Servers.
- The Index Tuning Wizard analyzes how a SQL statement, or group of statements, uses the existing indexes on a set of tables.

The wizard makes recommendations on index changes that would speed up the SQL statements.

Programmable Administration

Administering SQL Server 2000 can be highly automated, freeing database administrators to design new databases and applications.

SQL Distributed Management Objects (SQL-DMO) is a set of Automation objects that can be used to code applications with the logic to administer an instance of SQL Server 2000. This gives application packages the ability to transparently embed SQL Server 2000 into their applications. Experienced

database administrators can also use SQL-DMO to build applications for many of the common administrative tasks unique to their site. SQL Server 2000 also includes support for the Windows Management Instrumentation (WMI) API. The WMI support maps over the SQL-DMO API.

Routine, recurring tasks can be implemented as automatically scheduled jobs that run without constant supervision by an operator. For example, after a database administrator has designed a backup procedure for a server, the backups can be implemented as a set of automatic jobs.

SQL Server 2000 can also be programmed to raise alerts when specific events occur. The actions taken by alerts can take several forms:

• E-mail, paging messages, or Windows 2000 **net send** messages can be sent to the affected parties.

For example, if the number of Full Scans (a scan of an entire table or index) in a server exceeds a specific number, an e-mail can be sent to the database administrator for investigation.

• A predefined job can be executed to address the problem (if it is relatively routine and can be addressed programmatically).

Installation and Upgrade

The SQL Server 2000 compact disc has an autorun application that enables users to make several choices, such as:

- Install a new instance of SQL Server 2000.
- Upgrade an existing instance of Microsoft SQL Server version 7.0 or earlier.
- Install prerequisite software.
- Install only the documentation from the CD so that it can be reviewed before the product is installed.

• View an evaluation guide explaining the benefits of SQL Server 2000 features.

The installation or upgrade of SQL Server 2000 is driven by a graphical user interface (GUI) application that guides users through the information required by SQL Server 2000 Setup. The Setup program itself detects automatically if an earlier version of SQL Server is present and, after SQL Server 2000 is installed, asks users if they want to launch the SQL Server 2000 Upgrade Wizard to quickly guide them through the upgrade process. The entire installation or upgrade process is accomplished quickly and with minimal input from the users.

Sites needing to install SQL Server 2000 on many servers can take advantage of the SQL Server unattended installation feature to install SQL Server with the appropriate configuration on all the servers.

Building SQL Server 2000 Applications

SQL Server 2000 has several advantages in building applications:

• Full integration in the Windows DNA architecture by providing native support for the Windows DNA data access APIs, including ADO, OLE DB, and the MDX (multi-dimensional) OLAP extensions to these APIs

These APIs include powerful, low-level APIs, such as ODBC and OLE DB, that allow programmers control over the interaction between the application and database. They also include APIs such as ADO that support Rapid Application Development.

• SQL-DMO, SQL-DTS, and replication components

These are Automation objects used to write customized applications to administer a server running SQL Server.

• SQL Query Analyzer

This component enables programmers to develop and test Transact-SQL statements interactively. It includes aids such as a graphical display of the execution plan and performance statistics of a Transact-SQL statement. It color-codes the different syntax elements to increase the readability of Transact-SQL statements, and includes an integrated Transact-SQL debugger. It also has an Object Browser that determines

the attributes of the tables, views, stored procedures, and other objects in a database, and supports templates used to speed the building of complex statements.

• Analysis Services, Meta Data Services, and English Query programming

Analysis Services and Meta Data Services supply OLE DB Providers that support the online analytical process (OLAP) extensions to OLE DB and ADO. These allow the easy integration of OLAP and meta data processing in applications using the Microsoft data-access APIs. English Query also supports an object-model API that allows the easy integration of English Query functionality into applications accessing SQL Server 2000 databases and Analysis Services cubes through OLE DB or ADO.

• Transact-SQL programmability improvements

SQL Server 2000 introduces several items that improve the power and flexibility of Transact-SQL, as well as increasing programmer productivity. Cascading referential integrity actions can replace the need to develop triggers to enforce referential integrity actions when you update or delete rows. INSTEAD OF triggers can be used to greatly extend the types of update actions that views can support, and you can now specify which AFTER triggers fire first or last. User defined functions can be used to introduce new functionality to Transact-SQL statements.

Security Integrated with Windows NT and Windows 2000 Security

SQL Server supports using Windows NT and Windows 2000 user and domain accounts as SQL Server 2000 login accounts. This is called Windows Authentication. Users are validated by Windows 2000 when they connect to the network. When a connection is formed to SQL Server, the SQL Server client software requests a trusted connection, which can be granted only if validated by Windows 2000. SQL Server then does not have to validate the user separately. Users do not have to have separate logins and passwords for each SQL Server system to which they connect. With Windows Authentication, no passwords are transmitted to the server running SQL Server, eliminating a security concern. Also, SQL Server 2000 supports the use of Secure Sockets Layer encryption of all network traffic between their client computer and an instance of SQL Server.

SQL Server 2000 also provides auditing, which allows you to trace and record the activity in an instance of SQL Server. SQL Server 2000 auditing can support the C2 level of security defined by the United States government. For more information, see the Trusted Facilities Manual.

See Also

Administration Architecture Application Development Architecture Overview of Installing SQL Server 2000 SQL Server Architecture

Data Warehousing

Microsoft® SQL Server[™] 2000 includes several components you can use to build data warehouses that effectively support your decision support processing needs.

Data Warehousing Framework

The Data Warehousing Framework is a set of components and APIs that implement the data warehousing features of SQL Server 2000. It provides a common interface to be used by various components seeking to build and use a data warehouse or data mart.

Data Transformation Services

Data Transformation Services (DTS) provides a set of services used to build a data warehouse or data mart. Decision support systems analyze data to find trends of interest to the database users. Online transaction processing databases store large numbers of records covering the details of each transaction, and online analytical processing (OLAP) systems aggregate and summarize the information to speed analysis of the trends exhibited in the data.

DTS offers support for extracting data from heterogeneous OLE DB data sources and the summarizing or aggregating of data to build a data warehouse.

Online Analytical Processing Support

Microsoft SQL Server 2000 Analysis Services allows you to build flexible, powerful business intelligence applications for Web sites and large enterprise systems.

Microsoft SQL Server 2000 Analysis Services provides OLAP processing capabilities against heterogeneous OLE DB data sources. It has efficient algorithms for defining and building multidimensional cubes that can be referenced by applications using the OLE DB 2.0 OLAP extensions or the Microsoft ActiveX® Data Objects Multidimensional extensions. Analysis Services is an excellent tool for multidimensional analysis of data in SQL Server 2000 databases.

Analysis Services supports multidimensional queries against cubes with hundreds of millions of dimensions. You can control cube security down to the level of cells and members. You can create custom rollup functions that tailor the types of aggregations and processing that can be performed in multidimensional cubes.

Data Mining Support

Data mining allows you to define models containing grouping and predictive rules that can be applied to data in either a relational database or multidimensional OLAP cubes. These predictive models are then used to automatically perform sophisticated analysis of the data to find trends that help you identify new opportunities and chose the ones that have a winning outcome. SQL Server 2000 Analysis Services includes support for data mining models, including API support of the OLE DB for Data Mining specification. Through the OLE DB for Data Mining API, Analysis Services supports integration with third-party data mining providers.

English Query

English Query makes a definition of the entities and relationships defined in a SQL Server 2000 database. Given this definition, an application can use an Automation API to pass English Query a string containing a natural-language question about the data in the database. English Query returns a SQL statement that the application can use to extract the necessary data.

Meta Data Services

SQL Server 2000 includes Microsoft Meta Data Services, which consists of a set of Microsoft ActiveX® interfaces and information models that define database schema and data transformations as defined by the Microsoft Data Warehousing Framework. A goal of the Microsoft Data Warehousing Framework is to provide meaningful integration of multiple products through shared meta data. It combines business and technical meta data to provide an industry standard method for storing the schema of production data sources and destinations.

Meta Data Services is the preferred means of storing DTS packages in a data

warehousing scenario because it is the only method of providing data lineage for packages. DTS also uses Meta Data Services storage to allow transformations, queries, and ActiveX scripts to be reused by heterogeneous applications.

See Also

Data Warehousing and Online Analytical Processing

SQL Server Architecture

Relational Database Components

The database component of Microsoft® SQL Server[™] 2000 is a Structured Query Language (SQL)–based, scalable, relational database with integrated Extensible Markup Language (XML) support for Internet applications. Each of the following terms describes a fundamental part of the architecture of the SQL Server 2000 database component:

Database

A database is similar to a data file in that it is a storage place for data. Like a data file, a database does not present information directly to a user; the user runs an application that accesses data from the database and presents it to the user in an understandable format.

Database systems are more powerful than data files in that data is more highly organized. In a well-designed database, there are no duplicate pieces of data that the user or application must update at the same time. Related pieces of data are grouped together in a single structure or record, and relationships can be defined between these structures and records.

When working with data files, an application must be coded to work with the specific structure of each data file. In contrast, a database contains a catalog that applications use to determine how data is organized. Generic database applications can use the catalog to present users with data from different databases dynamically, without being tied to a specific data format.

A database typically has two main parts: first, the files holding the physical database and second, the database management system (DBMS) software that applications use to access data. The DBMS is responsible for enforcing the database structure, including:

- Maintaining relationships between data in the database.
- Ensuring that data is stored correctly, and that the rules defining data relationships are not violated.

• Recovering all data to a point of known consistency in case of system failures.

Relational Database

Although there are different ways to organize data in a database, relational databases are one of the most effective. Relational database systems are an application of mathematical set theory to the problem of effectively organizing data. In a relational database, data is collected into tables (called relations in relational theory).

A table represents some class of objects that are important to an organization. For example, a company may have a database with a table for employees, another table for customers, and another for stores. Each table is built of columns and rows (called attributes and tuples in relational theory). Each column represents some attribute of the object represented by the table. For example, an **Employee** table would typically have columns for attributes such as first name, last name, employee ID, department, pay grade, and job title. Each row represents an instance of the object represented by the table. For example, one row in the **Employee** table represents the employee who has employee ID 12345.

When organizing data into tables, you can usually find many different ways to define tables. Relational database theory defines a process called normalization, which ensures that the set of tables you define will organize your data effectively.

Scalable

SQL Server 2000 supports having a wide range of users access it at the same time. An instance of SQL Server 2000 includes the files that make up a set of databases and a copy of the DBMS software. Applications running on separate computers use a SQL Server 2000 communications component to transmit commands over a network to the SQL Server 2000 instance. When an application connects to an instance of SQL Server 2000, it can reference any of the databases in that instance that the user is authorized to access. The communication component also allows communication between an instance of SQL Server 2000 and an application running on the same computer. You can run multiple instances of SQL Server 2000 on a single computer. SQL Server 2000 is designed to support the traffic of the largest Web sites or enterprise data processing systems. Instances of SQL Server 2000 running on large, multiprocessor servers are capable of supporting connections to thousands of users at the same time. The data in SQL Server tables can be partitioned across multiple servers, so that several multiprocessor computers can cooperate to support the database processing requirements of extremely large systems. These groups of database servers are called federations.

Although SQL Server 2000 is designed to work as the data storage engine for thousands of concurrent users who connect over a network, it is also capable of working as a stand-alone database directly on the same computer as an application. The scalability and ease-of-use features of SQL Server 2000 allow it to work efficiently on a single computer without consuming too many resources or requiring administrative work by the stand-alone user. The same features allow SQL Server 2000 to dynamically acquire the resources required to support thousands of users, while minimizing database administration and tuning. The SQL Server 2000 relational database engine dynamically tunes itself to acquire or free the appropriate computer resources required to support a varying load of users accessing an instance of SQL Server 2000 at any specific time. The SQL Server 2000 relational database engine has features to prevent the logical problems that occur if a user tries to read or modify data currently used by others.

Structured Query Language

To work with data in a database, you have to use a set of commands and statements (language) defined by the DBMS software. Several different languages can be used with relational databases; the most common is SQL. The American National Standards Institute (ANSI) and the International Standards Organization (ISO) define software standards, including standards for the SQL language. SQL Server 2000 supports the Entry Level of SQL-92, the SQL standard published by ANSI and ISO in 1992. The dialect of SQL supported by Microsoft SQL Server is called Transact-SQL (T-SQL). T-SQL is the primary language used by Microsoft SQL Server applications.

Extensible Markup Language

XML is the emerging Internet standard for data. XML is a set of tags that can be used to define the structure of a hypertext document. XML documents can be easily processed by the Hypertext Markup Language, which is the most important language for displaying Web pages.

Although most SQL statements return their results in a relational, or tabular, result set, the SQL Server 2000 database component supports a FOR XML clause that returns results as an XML document. SQL Server 2000 also supports XPath queries from Internet and intranet applications. XML documents can be added to SQL Server databases, and the OPENXML clause can be used to expose data from an XML document as a relational result set.

SQL Server Architecture

Database Applications and Servers

Microsoft[®] SQL Server[™] 2000 is designed to work effectively as:

- A central database on a server shared by many users who connect to it over a network. The number of users can range from a handful in one workgroup, to thousands of employees in a large enterprise, to hundreds of thousands of Web users.
- A desktop database that services only applications running on the same desktop.

Server Database Systems

Server-based systems are constructed so that a database on a central computer, known as a server, is shared among multiple users. Users access the server through an application:

- In a multitier system, such as Windows® DNA, the client application logic is run in two or more locations:
 - A thin client is run on the user's local computer and is focused on displaying results to the user.
 - The business logic is located in server applications running on a server. Thin clients request functions from the server application, which is itself a multithreaded application capable of working with many concurrent users. The server application is the one that opens connections to the database server. The server application can be running on the same server as the database, or it can connect across the network to a separate server operating as a database server. In complex systems, the business logic may be implemented in several interconnected server applications, or in multiple layers of server applications.

This is a typical scenario for an Internet application. For

example, a multithreaded server application can run on a Microsoft® Internet Information Services (IIS) server and service thousands of thin clients running on the Internet or an intranet. The server application uses a pool of connections to communicate with one or more instances of SQL Server 2000. The instances of SQL Server 2000 can be on the same computer as IIS, or they can be on separate servers in the network.

• In a two-tier client/server system, users run an application on their local computer, known as a <u>client application</u>, that connects over a network to an instance of SQL Server 2000 running on a server computer. The client application runs both business logic and the code to display output to the user, so this is sometimes referred to as a thick client.

Advantages of Server Database System

Having data stored and managed in a central location offers several advantages:

• Each data item is stored in a central location where all users can work with it.

Separate copies of the item are not stored on each client, which eliminates problems with users having to ensure they are all working with the same information. Their system does not need to ensure that all copies of the data are updated with the current values, because there is only one copy in the central location.

• Business and security rules can be defined one time on the server and enforced equally among all users.

Rule enforcement can be done in a database through the use of constraints, stored procedures, and triggers. Rules can also be enforced in a server application, since these applications are also central resources accessed by many thin clients.

• A relational database server optimizes network traffic by returning only the data an application needs.

For example, if an application working with a file server needs to

display a list of the names of sales representatives in Oregon, it must retrieve the entire employee file. If the application is working with a relational database server, it sends this command:

```
SELECT first_name, last_name
FROM employees
WHERE emp_title = 'Sales Representative'
AND emp_state = 'OR'
```

The relational database sends back only the names of the sales representatives in Oregon, not all of the information about all employees.

• Hardware costs can be minimized.

Because the data is not stored on each client, clients do not have to dedicate disk space to storing data. The clients also do not need the processing capacity to manage data locally, and the server does not need to dedicate processing power to displaying data.

The server can be configured to optimize the disk I/O capacities needed to retrieve data, and clients can be configured to optimize the formatting and display of data retrieved from the server.

The server can be stored in a relatively secure location and equipped with devices such as an Uninterruptable Power Supply more economically than fully protecting each client.

• Maintenance tasks such as backing up and restoring data are simplified because they can focus on the central server.

Advantages of SQL Server 2000 as a Database Server

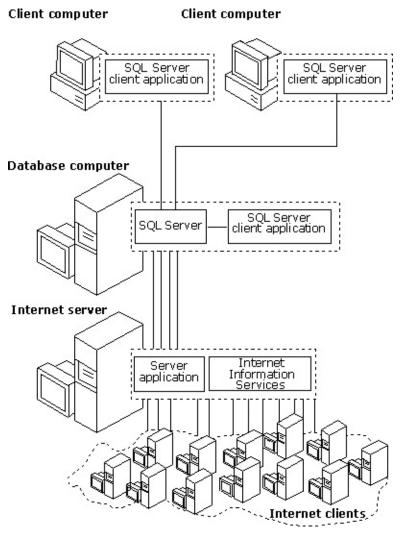
Microsoft SQL Server 2000 is capable of supplying the database services needed by extremely large systems. Large servers may have thousands of users connected to an instance of SQL Server 2000 at the same time. SQL Server 2000 has full protection for these environments, with safeguards that prevent problems, such as having multiple users trying to update the same piece of data at the same time. SQL Server 2000 also allocates the available resources effectively, such as memory, network bandwidth, and disk I/O, among the multiple users.

Extremely large Internet sites can partition their data across multiple servers, spreading the processing load across many computers, and allowing the site to serve thousands of concurrent users.

Multiple instances of SQL Server 2000 can be run on a single computer. For example, an organization that provides database services to many other organizations can run a separate instance of SQL Server 2000 for each customer organization, all on one computer. This isolates the data for each customer organization, while allowing the service organization to reduce costs by only having to administer one server computer.

SQL Server 2000 applications can run on the same computer as SQL Server 2000. The application connects to SQL Server 2000 using Windows Interprocess Communications (IPC) components, such as shared memory, instead of a network. This allows SQL Server 2000 to be used on small systems where an application must store its data locally.

The illustration shows an instance of SQL Server 2000 operating as the database server for both a large Web site and a legacy client/server system.



The largest Web sites and enterprise-level data processing systems often generate more database processing than can be supported on a single computer. In these large systems, the database services are supplied by a group of database servers that form a database services tier. SQL Server 2000 does not support a load-balancing form of clustering for building a database services tier, but it does support a mechanism that can be used to partition data across a group of autonomous servers. Although each server is administered individually, the servers cooperate to spread the database-processing load across the group. A group of autonomous servers that share a workload is called a federation of servers. For more information, see <u>Designing Federated Database Servers</u>.

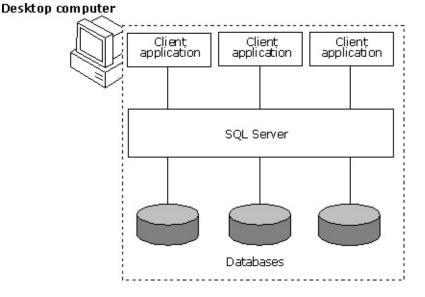
Desktop Database Systems

Although SQL Server 2000 works effectively as a powerful database server, the

same database engine can also be used in applications that need stand-alone databases stored locally on the client. SQL Server 2000 can configure itself dynamically to run efficiently with the resources available on a client desktop or laptop computer, without the need to dedicate a database administrator to each client. Application vendors can also embed SQL Server 2000 as the data storage component of their applications.

When clients use local SQL Server 2000 databases, applications connect to local instances of the database engine in much the same way they connect across the network to a database engine running on a remote server. The primary difference is that local connections are made through local IPCs such as shared memory, and remote connections must go through a network.

The illustration shows using SQL Server 2000 in a desktop database system.



SQL Server Architecture

Logins

To connect to an instance of Microsoft® SQL Server[™] 2000, you typically give an application only two or three pieces of information:

- The network name of the computer on which the SQL Server instance is running.
- The name of the instance (optional, required only if you are connecting to a named instance).
- Your login identifier (ID).

A login ID is the account identifier that controls access to any SQL Server 2000 system. SQL Server 2000 does not complete a connection unless it has first verified that the login ID specified is valid. Verification of the login is called authentication.

One of the properties of a login is the default database. When a login connects to SQL Server, this default database becomes the current database for the connection, unless the connection request specifies that another database be made the current database.

A login ID only enables you to connect to an instance of SQL Server. Permissions within specific databases are controlled by user accounts. The database administrator maps your login account to a user account in any database you are authorized to access. For more information, see <u>Logins, Users,</u> <u>Roles, and Groups</u>.

Authenticating Logins

Instances of SQL Server must verify that the login ID supplied on each connection request is authorized to access the instance. This process is called authentication. SQL Server 2000 uses two types of authentication: Windows Authentication and SQL Server Authentication. Each has a different class of login ID.

Windows Authentication

A member of the SQL Server 2000 **sysadmin** fixed server role must first specify to SQL Server 2000 all the Microsoft Windows NT® or Microsoft Windows® 2000 accounts or groups that can connect to SQL Server 2000. When using Windows Authentication, you do not have to specify a login ID or password when you connect to SQL Server 2000. Your access to SQL Server 2000 is controlled by your Windows NT or Windows 2000 account or group, which is authenticated when you log on to the Windows operating system on the client.

When you connect, the SQL Server 2000 client software requests a Windows *trusted connection* to SQL Server 2000. Windows does not open a trusted connection unless the client has logged on successfully using a valid Windows account. The properties of a trusted connection include the Windows NT and Windows 2000 group and user accounts of the client that opened the connection. SQL Server 2000 gets the user account information from the trusted connection properties and matches them against the Windows accounts defined as valid SQL Server 2000 logins. If SQL Server 2000 finds a match, it accepts the connection. When you connect to SQL Server 2000 using Windows 2000 Authentication, your identification is your Windows NT or Windows 2000 group or user account.

The Microsoft Windows 98 operating system does not support the server side of the trusted connection API. When SQL Server is running on Windows 98, it does not support Windows Authentication. Users must supply a SQL Server login when they connect. When SQL Server is running on Windows NT or Windows 2000, Windows 95 and Windows 98 clients can connect to it using Windows 2000 Authentication.

SQL Server Authentication

A member of the **sysadmin** fixed server role first specifies to SQL Server 2000 all the valid SQL Server 2000 login accounts and passwords. These are not related to your Microsoft Windows account or network account. You must supply both the SQL Server 2000 login and password when you connect to SQL Server 2000. You are identified in SQL Server 2000 by your SQL Server 2000 login.

SQL Server Authentication Modes

When SQL Server 2000 is running on Windows NT or Windows 2000, members of the **sysadmin** fixed server role can specify one of two authentication modes:

• Windows Authentication Mode

Only Windows Authentication is allowed. Users cannot specify a SQL Server 2000 login ID. This is the default authentication mode for SQL Server 2000. You cannot specify Windows Authentication Mode for an instance of SQL Server running on Windows 98, because the operating system does not support Windows Authentication.

• Mixed Mode

If users supply a SQL Server 2000 login ID when they log on, they are authenticated using SQL Server Authentication. If they do not supply a SQL Server 2000 login ID, or request Windows Authentication, they are authenticated using Windows Authentication.

These modes are specified during setup or with SQL Server Enterprise Manager.

Login Delegation

If you use Windows Authentication to log on to an instance of SQL Server 2000 running on Windows 2000, and the computer has Kerberos support enabled, SQL Server 2000 can pass your Windows login credentials to other instances of SQL Server. Delegation of your credentials from one instance to another is sometimes called impersonation, typically when both instances of SQL Server are running on the same computer.

For example, if Instance A and Instance B are running on separate computers using Windows 2000, you can connect to Instance A and execute a distributed query that references tables on Instance B. When Instance A connects to Instance B to retrieve the required data, Instance A can use your Windows account credentials for the connection. Instance B has visibility to your specific account, and can validate your individual permissions to access the data requested.

Without delegation, administrators have to specify the login that Instance A uses to connect to Instance B (or any other instance). This login is used regardless of which user executes a distributed query on Instance A, and prevents Instance B

from having any knowledge of the actual user executing the query. The administrators of Instance B cannot define permissions specific to individual users coming in from Instance A, they must define a global set of permissions for the login account used by Instance A. The administrators also cannot audit which specific users perform actions in Instance B. Using delegation with Windows Authentication on Windows 2000 allows administrators greater control over user permissions and gives auditors greater visibility to the actions of individual users.

Connections that use delegation are authenticated using a Kerberos ticket. Each ticket has a timeout period defined by the Windows 2000 security administrator. If a connection remains idle for a long period and the Kerberos ticket times out, all subsequent attempts to execute a distributed query will fail until the user disconnects and reconnects.

See Also

Managing Security
Security Account Delegation

SQL Server Architecture

Client Components

Clients do not access Microsoft® SQL Server[™] 2000 directly; instead, clients use applications written to access the data in SQL Server. These can include utilities that come with SQL Server 2000, third-party applications that access SQL Server 2000, in-house applications developed by programmers at the SQL Server 2000 site, or Web pages. SQL Server 2000 can also be accessed through COM, Microsoft ActiveX®, or Windows® DNA components.

SQL Server 2000 supports two main classes of applications:

- Relational database applications that send Transact-SQL statements to the database engine; results are returned as relational result sets.
- Internet applications that send either Transact-SQL statements or XPath queries to the database engine; results are returned as XML documents.

Relational Database APIs

Relational database applications are written to access SQL Server 2000 through a database application programming interface (API). A database API contains two parts:

• The language statements passed to the database.

The language by relational SQL Server 2000 applications is Transact-SQL. Transact-SQL supports all SQL-92 Entry Level SQL statements and many additional SQL-92 features. It also supports the ODBC extensions to SQL-92 and other extensions specific to Transact-SQL.

• A set of functions or object-oriented interfaces and methods used to send the language statements to the database and process the results returned by the database.

Native API Support

Native API support means the API function calls are mapped directly to the

network protocol sent to the server. There is no intermediate translation to another API needed. SQL Server 2000 provides native support for two main classes of database APIs:

• OLE DB

SQL Server 2000 includes a native OLE DB provider. The provider supports applications written using OLE DB, or other APIs that use OLE DB, such as ActiveX Data Objects (ADO). Through the native provider, SQL Server 2000 also supports objects or components using OLE DB, such as ActiveX, ADO, or Windows DNA applications.

• ODBC

SQL Server 2000 includes a native ODBC driver. The driver supports applications or components written using ODBC, or other APIs using ODBC, such as DAO, RDO, and the Microsoft Foundation Classes (MFC) database classes.

An example of nonnative support for an API would be a database that does not have an OLE DB provider, but does have an ODBC driver. An OLE DB application could use the OLE DB provider for ODBC to connect to the database through an ODBC driver. This provider maps the OLE DB API function calls from the application to ODBC function calls it sends to the ODBC driver.

Additional SQL Server API Support

SQL Server 2000 also supports:

• DB-Library

DB-Library is an API specific to SQL Server 2000 and Microsoft SQL Server. SQL Server 2000 supports DB-Library applications written in C. DB-Library has not been extended beyond the functionality it had in Microsoft SQL Server version 6.5. Existing DB-Library applications developed against earlier versions of Microsoft SQL Server can be run against SQL Server 2000, but many features introduced in SQL Server 2000 and SQL Server version 7.0 are not available to DB-Library applications.

• Embedded SQL

SQL Server 2000 includes a C precompiler for the Embedded SQL API. Embedded SQL applications use the DB-Library DLL to access SQL Server 2000.

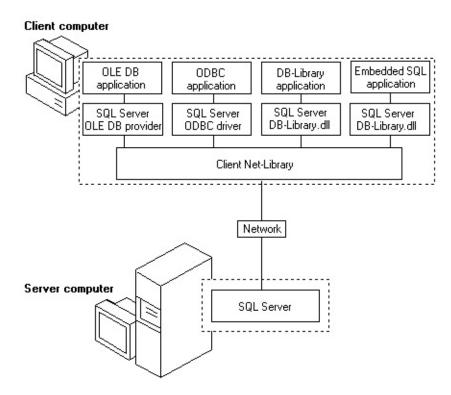
XML Access

Internet applications retrieve results in the form of XML documents rather than relational result sets. The applications execute either XPath queries or Transact-SQL statements that use the FOR XML clause to specify that results be returned as XML documents. If you define a virtual root on a Microsoft Internet Information Server (IIS) that points to an instance of SQL Server 2000, IIS applications can use three mechanisms for executing XPath queries or Transact-SQL statements:

- Execute a Uniform Resource Locator (URL) that references the virtual root and contains an XPath query or Transact-SQL statement with FOR XML.
- Use the ADO API to execute an XPath query to Transact-SQL statement with FOR XML.
- Use the OLE DB API to execute an XPath query to Transact-SQL statement with FOR XML.

Client Communications

The Microsoft OLE DB Provider for SQL Server 2000, the SQL Server 2000 ODBC driver, and DB-Library are each implemented as a DLL that communicates to SQL Server 2000 through a component called a client Net-Library.



See Also

Application Development Architecture

Overview of Building SQL Server Applications

SQL Server Architecture

Communication Components

Microsoft® SQL Server[™] 2000 supports several methods of communicating between client applications and the server. When the application is on the same computer as an instance of SQL Server 2000, Windows Interprocess Communication (IPC) components, such as local named pipes or shared memory, are used. When the application is on a separate client, a network IPC is used to communicate with SQL Server.

An IPC has two components:

• Application Programming Interface (API)

The API is a definition of the set of functions software uses to send requests to and retrieve results from the IPC.

• Protocol

The protocol defines the format of the information sent between any two components communicating through the IPC. In the case of a network IPC, the protocol defines the format of the packets sent between two computers using the IPC.

Some network APIs can be used over multiple protocols. For example, the Named Pipes API and the Microsoft Win32® RPC API can both be used with several protocols. Other network APIs, such as the Banyan VINES API, can be used with only one protocol.

The SQL Server 2000 client communication components require little or no administration when they connect to SQL Server 2000. Although the actual implementation of the communication components is more complex than in earlier versions of SQL Server, SQL Server 2000 users are shielded from this when connecting to instances of SQL Server 2000. The SQL Server 2000 client software dynamically determines the network address needed to communicate with any instance of SQL Server 2000. All the client software needs is the network name of the computer on which the SQL Server 2000 instance is running, and the name of the instance if connecting to a named instance. There are very few reasons for SQL Server 2000 users to manage the client communications components using the Client Network Utility.

System Area Networks

SQL Server 2000 Enterprise Edition introduces support for System Area Network (SAN) protocols built using the Virtual Interface Architecture (VIA). A SAN is a high-speed, highly reliable network for interconnecting servers or clusters of servers. A multi-tier, distributed system can generate extremely high levels of network traffic between servers. Gaining high performance in such a system is possible only if message transmissions are fast enough to minimize the time the servers spend processing messages and waiting for replies. Compared to local area networks (LANs) or wide area networks (WANs), SANs support high levels of messaging traffic by lowering CPU loads and message latency. SANs are also more reliable than LANs or WANs, and are implemented in groups or clusters of servers that are located close together, such as in the same computer room.

Compaq®, Intel®, Microsoft, and other companies have defined Virtual Interface Architecture (VIA) as a generic definition of a SAN that allows many possible hardware implementations. The Virtual Interface Architecture allows a VIA provider to implement a flexible, scalable, robust messaging component built at low cost using standard components. VIA SANs can support the intense messaging requirements of large Web servers.

The Virtual Interface Architecture defines both an API and a protocol. The API is referred to as the VIA API, and protocol is referred to as the VIA protocol.

SANs are well suited for these uses with SQL Server 2000:

- The application servers forming the business services tier can use the SAN for high-speed communications with the data services tier. This is done when the application servers and database servers are at the same physical location.
- SQL Server 2000 servers can use the SAN to improve the performance of distributed queries, distributed transactions, and data replication between database servers at the same location. A SAN can improve the distributed queries needed to support the distributed views used to implement federations of computers running SQL Server.

SQL Server 2000 supports the Giganet VIA SAN implementation. Because

SANs are intended to support the high communications bandwidth between servers, SQL Server 2000 only supports the VIA Net-Libraries on the Windows NT® Server, Windows 2000 Data Center, Advanced Server, and Server operating systems. SQL Server Architecture

Client and Server Net-Libraries

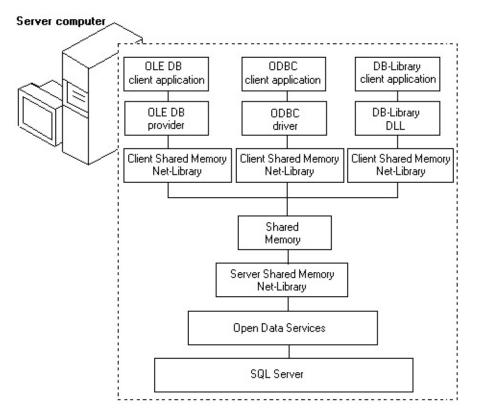
Microsoft® SQL Server[™] 2000 uses components called client Net-Libraries to shield the OLE DB Provider for SQL Server 2000, the SQL Server 2000 ODBC driver, and the DB-Library DLL, from the details of communicating with different Interprocess Communication (IPC) components. Server Net-Libraries perform the same function for the database engine.

The following components manage communications between SQL Server 2000 and its clients in this sequence:

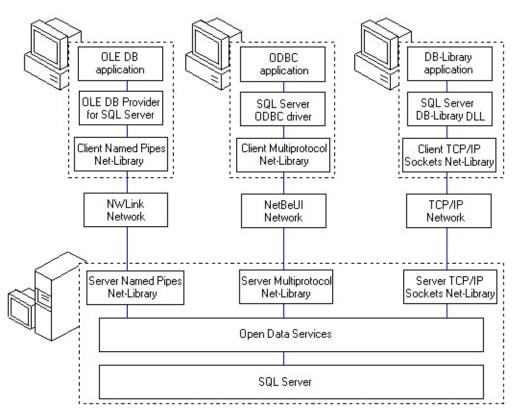
- 1. The client application calls the OLE DB, ODBC, DB-Library, or Embedded SQL API. This causes the OLE DB provider, ODBC driver, or DB-Library DLL to be used for SQL Server communications.
- 2. The OLE DB provider, ODBC driver, or DB-Library DLL calls a client Net-Library. The client Net-Library calls an IPC API.
- 3. The client calls to the IPC API are transmitted to a server Net-Library by the underlying IPC. If it is a local IPC, calls are transmitted using a Windows operating IPC such as shared memory or local named pipes. If it is a network IPC, the network protocol stack on the client uses the network to communicate with the network protocol stack on the server.
- 4. The server Net-Library passes the requests coming from the client to the instance of SQL Server 2000.

Replies from SQL Server 2000 to the client follow the reverse sequence.

This illustration shows the communication path when a SQL Server application runs on the same computer as an instance of SQL Server.



This is a simplified illustration of the communication path when a SQL Server application connects through a LAN or WAN to an instance of SQL Server 2000 on a separate computer. Although the illustration shows the OLE DB Provider for SQL Server 2000, SQL Server 2000 ODBC driver, and DB-Library DLL using specific Net-Libraries, there is nothing that limits these components to these Net-Libraries. The provider, driver, and DB-Library can each use any of the SQL Server Net-Libraries.

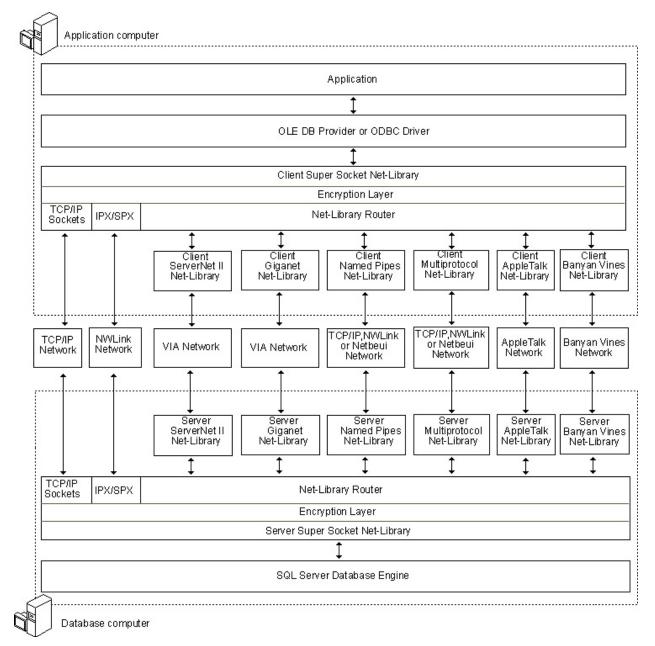


SQL Server 2000 classifies the Net-Libraries as primary or secondary Net-Libraries. The OLE DB Provider for SQL Server 2000, the SQL Server 2000 ODBC driver, the DB-Library DLL, and the database engine communicate directly with only the two primary Net-Libraries:

- By default, local connections between an application and an instance of SQL Server 2000 on the same computer use the Shared Memory primary Net-Library. This path is shown in the illustration above.
- Intercomputer connections communicate through the Super Socket primary Net-Library. The Super Socket Net-Library has two communication paths:
 - If you choose a TCP/IP Sockets connection or an NWLINK IPX/SPX connection, the Super Socket Net-Library directly calls the Windows Socket 2 API for the communication between the application and the instance of SQL Server 2000.
 - If a Named Pipes, Virtual Interface Architecture (VIA) SAN,

Multiprotocol, AppleTalk, or Banyan VINES connection is chosen, a subcomponent of the Super Socket Net-Library, called the Net-Library router, loads the secondary Net-Library for the chosen protocol and routes all Net-Library calls to it.

This illustration shows in more detail the communication paths through the client and server Net-Libraries for network connections between a computer running the SQL Server 2000 client components and an instance of SQL Server 2000.



The server Super Socket Net-Library is implemented as Ssnetlib.dll, and the client Super Socket Net-Library is implemented as Dbnetlib.dll.

This table shows how the Net-Libraries relate to the IPC APIs and protocols used to make connections.

Protocol specified in network utilities		Server Net- Library used	IPC API called by Net- Library	Protocols supporting the IPC API
TCP/IP Sockets	Dbnetlib.dll	Ssnetlib.dll	Windows Socket 2	TCP/IP
Named Pipes	routes to	Ssnetlib.dll routes to Ssnmpn70.dll (Microsoft Windows NT® and Windows® 2000 only)	Windows Named Pipes	File system (local) TCP/IP NetBEUI NWLink
NWLink IPX/SPX	Dbnetlib.dll	Ssnetlib.dll	Windows Socket 2	NWLink
VIA GigaNet SAN	routes to Dbmsgnet.dll (Microsoft Windows NT	Ssnetlib.dll routes to Dbmsgnet.dll (Microsoft Windows NT and Windows 2000 only)	Virtual Interface Architecture (VIA)	Virtual Interface Architecture (VIA)
Multiprotocol	Dbnetlib.dll routes to Dbmsrpcn.dll	Ssnetlib.dll routes to Ssmsrpc.dll (default instance only)	Windows RPC	File system (local) TCP/IP NetBEUI NWLink
AppleTalk	Dbnetlib.dll routes to	Ssnetlib.dll routes to	AppleTalk ADSP	AppleTalk

	Ssmsad70.dll (default instance only)		
routes to Dbmsvinn.dll	routes to	Banyan VINES SPP	Banyan VINES

Instances of SQL Server 2000 running on Microsoft Windows® 98 do not support the server Named Pipes and Banyan VINES Net-Libraries, because the Windows 98 operating system does not support the server part of these APIs. SQL Server 2000 also does not support the server NWLink IPX/SPX Net-Library on Windows 98. SQL Server 2000 does support the client side of these Net-Libraries on Windows 98; therefore, applications running on Windows 98 can use the Net-Libraries to connect to instances of SQL Server on Microsoft Windows NT or Microsoft Windows 2000. Applications running on Windows 95 can also make connections using the client side of these Net-Libraries.

The AppleTalk Net-Library does not run on computers running Windows 95 or Windows 98.

VIA networks are designed to support the high levels of messaging traffic between servers in the same data center, such as in a Web site implemented as one or more Internet Information Services application servers connected to one or more database servers running SQL Server. VIA networks are not used to connect individual workstations. Both the client and server SQL Server VIA Net-Libraries are supported only on Windows NT Server and Advanced Server, and Windows 2000 Server, Advanced Server, and Data Center.

Named instances of SQL Server 2000 support only the Named Pipes, TCP/IP Sockets, NWLink IPX/SPX, and Shared Memory Net-Libraries. Named instances do not support the Multiprotocol, AppleTalk, or Banyan VINES Net-Libraries. To maintain compatibility with earlier versions of SQL Server, default instances support all server Net-Libraries.

Some of the Net-Libraries support only one type of protocol stack. For example,

the AppleTalk Net-Library requires an AppleTalk protocol stack. Other Net-Libraries, such as the Named Pipes and Multiprotocol Net-Libraries support several protocol stacks.

The Microsoft SQL Server Net-Libraries have been tested intensively with the Microsoft protocol stacks and are supported with these stacks. Protocol stacks from other vendors should work, provided that the stacks fully support the APIs used by the Microsoft SQL Server Net-Libraries.

When the Named Pipes or Multiprotocol Net-Libraries are used to connect an application to an instance of SQL Server on the same computer, and the computer does not have a protocol stack, the IPC APIs are implemented by the file system.

Controlling Net-Libraries and Communications Addresses

After installing Microsoft® SQL Server[™] 2000, you define the behaviors of the client Net-Libraries by using the Client Network Utility and server Net-Libraries by using the Server Network Utility.

Each instance of SQL Server 2000 can be listening on any combination of the server Net-Libraries at one time. There is one set of server Net-Libraries for each set of database engine executable files. The server Net-Libraries are installed in: C:\Program Files\Microsoft SQL Server\MSSQL\$*n*, where *n* is the number associated with this set of database engine executable files.

All of the server Net-Libraries are installed during the server portion of SQL Server Setup, but some of them may not be active. The person running the Setup program can choose which combination of Net-Libraries is active for the instance being installed. The table shows the default server Net-Libraries that are activated by SQL Server Setup for the Microsoft Windows NT®, Microsoft Windows® 2000, and Microsoft Windows 98 operating systems.

Windows NT and Windows 2000	Windows 98
TCP/IP Sockets	TCP/IP Sockets
Shared Memory	Shared Memory
Named Pipes	

Disabling and Enabling Net-Libraries

After setup, you can disable and enable individual server Net-Libraries for each instance of SQL Server on a database computer using the Server Network Utility. When a server Net-Library is disabled for a specific instance, the database engine for the instance does not load the server Net-Library and does not accept connections using that Net-Library. The server Net-Library remains installed and can be enabled for other instances sharing the same set of executable files. For more information, see <u>SQL Server Network Utility</u>.

There is always one set of the client Net-Library DLLs installed on any computer running SQL Server 2000 client components. The client Net-Library

DLLs are installed in the C:\Windows\System32 or C:\Windows\System directory. All of the client Net-Libraries are installed when you install the SQL Server 2000 client utilities. You can enable and disable the various client Net-Libraries using the Client Network Utility. When a client Net-Library is disabled it remains installed but is not considered for any connections. You can:

- Specify the sequence in which client Net-Libraries are considered for all connections except those that use a server alias.
- Enable or disable specific client Net-Libraries.
- As a compatibility option, define server aliases that define specific Net-Libraries and connection parameters to use when connecting to instances of SQL Server version 7.0 or earlier.

For more information, see <u>Configuring Client Net-Libraries</u>.

Connecting to SQL Server 2000

For a client to connect to a server running SQL Server 2000, the client must use a client Net-Library that matches one of the server Net-Libraries the server is currently listening on. Also, both the client and server must be running a protocol stack supporting the network API called by the Net-Library being used for the connection. For example, if the client tries using the client Multiprotocol Net-Library, and the server is listening on the server Multiprotocol Net-Library, but the server is running with the TCP/IP protocol while the client computer is running only with the IPX/SPX protocol stack, the client cannot connect to the server. Both the client and the server must be using the same Net-Library and running the same protocol stack.

Each instance of SQL Server on a computer must listen on different network addresses so that applications can connect to specific instances. Default instances of SQL Server 2000 listen on the same default network addresses as earlier versions of SQL Server so that existing client computers can continue to connect to the default instance. The table shows the default network addresses that instances of SQL Server 2000 listen on.

Net-	Default instance network	
Library	address	Named instance network address
TCP/IP	TCP Port 1433	A TCP port is chosen dynamically the fi
Sockets		MSSQL\$ <i>instancename</i> service is started
Named	\\ <i>computername</i> \pipe\sql\query	\\computername\pipe\MSSQL\$instance
Pipes		
NWLink	Port 33854	First available port after 33854 for each
IPX/SPX		
VIA	VIA Port 0:1433	VIA Port 0:1433
Giganet		
SAN		

The VIA server Net-Libraries assign the same default address to both default and named instances. The system administrator must use the Server Network Utility to assign unique port addresses to each instance on a computer.

You can use the SQL Server 2000 Server Network Utility to find out what specific set of network address each instance of SQL Server is listening on for client connections.

When the SQL Server 2000 client Net-Libraries connect to an instance of SQL Server 2000, only the network name of the computer running the instance and the instance name are required. When an application requests a connection to a remote computer, Dbnetlib.dll opens a connection to UDP port 1434 on the computer network name specified in the connection. All computers running an instance of SQL Server 2000 listen on this port. When a client Dbnetlib.dll connects to this port, the server returns a packet listing all the instances running on the server. For each instance, the packet reports the server Net-Libraries and network addresses the instance is listening on. After the Dbnetlib.dll on the application computer receives this packet, it chooses a Net-Library that is enabled on both the application computer and on the instance of SQL Server, and makes a connection to the address listed for that Net-Library in the packet. The connection attempt fails only if:

• The requested instance of SQL Server 2000 is not running.

• None of the Net-Libraries that the instance of SQL Server 2000 is listening on is active on the application computer.

When Dbnetlib.dll compares the network protocols enabled on the application computer against those enabled on the instance of SQL Server 2000, the sequence of the comparison is specified using the Client Network Utility on the application computer. For example, assume an application computer has three client Net-Libraries enabled and specifies that the comparison sequence is TCP/IP Sockets first, NWLink IPX/SPX second, and named pipes third. If the application computer attempts a connection to an instance of SQL Server 2000 that has enabled only the NWLink IPX/SPX, named pipes and Multiprotocol server Net-Libraries, the connection is made using NWLink IPX/SPX. For more information about configuring the comparison sequence, see <u>Configuring Client</u> <u>Net-Libraries</u>.

You cannot assign UDP port 1434 to an application other than SQL Server on computers running instances of SQL Server 2000. Network administrators managing network filters must allow communications on UDP port 1434 to enable SQL Server 2000 connections to pass through the filter.

When running an application on the same computer as a default instance of SQL Server, you can use these names to reference the default instance.

Windows NT and Windows 2000	Windows 98 and Windows 95
Computer name	Computer name
(local)*	(local)*
•	

*Where "(local)" is the word local in parentheses and "." is a period, or dot. "." is valid only in SQL Server utilities, such as SQL Query Analyzer and **osql**; it cannot be specified in API connection requests.

Do not use either (local) or . to connect to a virtual server implemented using failover clustering.

Using the computer name is recommended. These connections will be made with the Shared Memory Net-Library. DB-Library does not support using (local).

Connecting to Earlier Instances of SQL Server

When applications using the SQL Server 2000 client components connect to

instances of SQL Server version 7.0 or earlier, the communications between the instance and the application function the same as they did in the earlier versions of SQL Server. Applications using SQL Server version 7.0 or earlier client components to connect to default instances of SQL Server 2000 also communicate as they did in earlier versions of SQL Server. In both of these cases you must administer the network addresses the way they were administered in earlier versions of SQL Server. For more information about configuring a client in earlier versions of SQL Server, see <u>Managing Clients</u>.

SQL Server version 6.5 and earlier supported Windows Authentication (called Integrated Security in those versions) only on the Named Pipes and Multiprotocol Net-Libraries. SQL Server 2000 and SQL Server version 7.0 support Windows Authentication on all Net-Libraries. Existing SQL Server version 6.5 or 7.0 applications that use the default Named Pipes Net-Library can be used to open Windows Authentication connections to instances of SQL Server version 6.5. However, if you upgrade the SQL Server client utilities on the application computer to SQL Server 2000, the default Net-Library changes to TCP/IP, and any attempt to open a Windows Authentication connection to instances of SQL Server version 6.5 fails. To resolve this, you can use the Client Network Utility to put the Named Pipes Net-Library at the top of the Net-Library list, thereby establishing it as the default Net-Library.

See Also

Managing Clients

Managing Servers

Tabular Data Stream Protocol

Microsoft® SQL Server[™] 2000 uses an application-level protocol called Tabular Data Stream (TDS) for communication between client applications and SQL Server. The TDS packets are encapsulated in the packets built for the protocol stack used by the Net-Libraries. For example, if you are using the TCP/IP Sockets Net-Library, then the TDS packets are encapsulated in the TCP/IP packets of the underlying protocol.

The contents of the packets that send result sets back to the application depends on whether FOR XML is specified in the Transact-SQL statement transmitted to the database engine:

- If FOR XML is not specified, the database engine sends a relational result set back to the application. The TDS packets contain the rows of the result set, with each row comprised of one or more columns, as specified in the select list of the SELECT statement.
- If FOR XML is specified, the database engine streams an XML document back to the application. The XML document is formatting in the TDS packets as if it were a single, long Unicode value, with each packet being approximately 4 KB in size.

You can configure the SQL Server packet size, which is the size of the TDS packets. The size of the TDS packets defaults to 4 KB on most clients (DB-Library applications default to 512 bytes), which testing has shown to be the optimal TDS packet size in almost all scenarios. The size of the TDS packets can be larger than the size of the packets in the underlying protocol. If this is the case, the protocol stack on the sending computer disassembles the TDS packets automatically into units that fit into the protocol packets, and the protocol stack on the client computer reassembles the TDS packets on the receiving computer.

Net-Library Encryption

Microsoft® SQL Server[™] 2000 can use the Secure Sockets Layer (SSL) to encrypt all data transmitted between an application computer and a SQL Server instance on a database computer. The SSL encryption is performed within the Super Socket Net-Library (Dbnetlib.dll and Ssnetlib.dll) and applies to all intercomputer protocols supported by SQL Server 2000. When SSL encryption is active, the Super Socket Net-Library performs the SSL encryption before calling:

- The Windows Socket 2 API to transmit TCP/IP Sockets or NWLink IPX/SPX packets.
- The Net-Library router to send a packet to the Named Pipe, Multiprotocol, AppleTalk, or Banyan VINES Net-Libraries.

SSL encryption works only with instances of SQL Server 2000 running on a computer that has been assigned a certificate from a public certification authority. The computer on which the application is running must also have a root CA certificate from the same authority.

The Net-Library encryption is implemented using the Secure Sockets Layer API. The level of encryption, 40-bit or 128-bit, depends on the version of the Microsoft Windows® operating system that is running on the application and database computers.

Enabling encryption slows the performance of the Net-Libraries. Encryption forces these actions in addition to all of the work for an unencrypted connection:

- An extra network round trip is required at connect time.
- All packets sent from the application to the instance of SQL Server must be encrypted by the client Net-Library and decrypted by the server Net-Library.
- All packets sent from the SQL Server instance to the application must

be encrypted by the server Net-Library and decrypted by the client Net-Library.

Shared memory Net-Library communications are inherently secure without the need for encryption. The shared memory Net-Library never participates in intercomputer communications. The area of memory shared between the application process and the database engine process cannot be accessed from any other Windows process.

For compatibility with earlier versions of SQL Server, the Multiprotocol Net-Library continues to support its own encryption. This encryption is specified independently of the SSL encryption and is implemented by calling the Windows RPC encryption API. It does not require the use of certificates. The level of RPC encryption, 40-bit or 128-bit, depends on the version of the Windows operating system that is running on the application and database computers. The Multiprotocol Net-Library is not supported by named instances.

Server Components

In addition to the server Net-Libraries, Microsoft® SQL Server[™] 2000 incorporates these main server components:

- SQL Server database engine (MSSQLServer service)
- SQL Server Agent (SQLServerAgent service)
- Microsoft Search service
- Microsoft Distributed Transaction Coordinator (MS DTC service)

The server components are supported on computers running the Microsoft Windows NT®, Windows® 2000, and Windows 98 operating systems. The server components are not supported on computers running Microsoft Windows 95. When SQL Server is running on Windows NT or Windows 2000, the SQL Server database engine, SQL Server Agent, and MS DTC are implemented as Windows NT or Windows 2000 services. On Windows 98, the server components are not implemented as services because the operating system does not support services. The Microsoft Search service is not available on Windows 95 or Windows 98.

The server components can be stopped and started several ways:

- Windows NT and Windows 2000 can start each service automatically when the operating system is starting.
- Use SQL Server Service Manager to start or stop the service.
- Use SQL Server Enterprise Manager to start or stop the service.
- On Windows NT or Windows 2000, use the **net start** and **net stop** command prompt commands to stop or start each service (except for a

virtual server in a failover cluster).

SQL Server 2000 supports multiple instances of SQL Server on computers running Windows NT or Windows 2000. Each instance has its own copy of the SQL Server service and the SQL Server Agent Service. There are only single copies of the Microsoft Search service or the MS DTC service, whose services are shared among the multiple instances of SQL Server running on the computer.

SQL Server Service

The Microsoft® SQL Server[™] 2000 database engine runs as a service on the Microsoft Windows NT® or Microsoft Windows® 2000 operating systems. It does not run as a service on Microsoft Windows 98 because this operating system does not support services. SQL Server can also run as an executable file on Windows NT and Windows 2000, although it is usually run as a service.

When multiple instances of SQL Server are run on the same computer, each instance has its own SQL Server service. The service name for the default instance is named MSSQLServer, the service name for named instances is MSSQL\$InstanceName. For more information, see <u>Multiple Instances of SQL</u> <u>Server</u>.

The SQL Server service manages all of the files that comprise the databases owned by an instance of SQL Server. It is the component that processes all Transact-SQL statements sent from SQL Server client applications. SQL Server also supports distributed queries that retrieve data from multiple sources, not only SQL Server.

The SQL Server service allocates computer resources effectively between multiple concurrent users. It also enforces business rules defined in stored procedures and triggers, ensures the consistency of the data, and prevents logical problems such as having two people trying to update the same data at the same time.

SQL Server Agent Service

SQL Server Agent supports features allowing the scheduling of periodic activities on Microsoft® SQL Server[™] 2000, or the notification to system administrators of problems that have occurred with the server. The SQL Server Agent components that implement this capability are:

• Jobs

Defined objects consisting of one or more steps to be performed. The steps are Transact-SQL statements that can be executed. Jobs can be scheduled, for example, to execute at specific times or recurring intervals.

• Alerts

Actions to be taken when specific events occur, such as a specific error, errors of certain severities, or a database reaching a defined limit of free space available. The alert can be defined to take such actions as sending an e-mail, paging an operator, or running a job to address the problem.

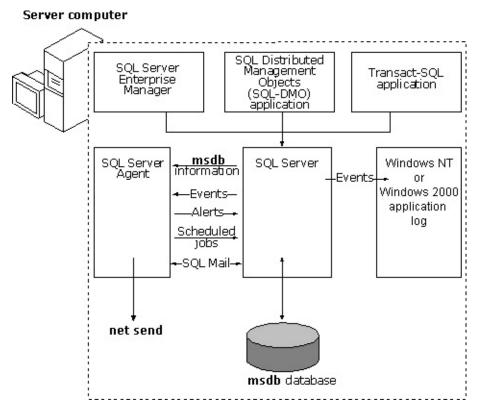
• Operators

People identified through their network account or e-mail identifier (ID) who can address problems with the server. They can be the targets of alerts, either through e-mail, a pager, or a **net send** network command.

The service name of SQLServerAgent applies only to the Agent service associated with a default instance. SQL Server Agent services associated with named instances are named SQLAgent\$InstanceName.

Managing Scheduled Operations

The illustration shows the primary components that are used in the definition and operation of jobs, alerts, and operators.



- Jobs, alerts, and operators are specified using:
 - SQL Server Enterprise Manager.
 - Applications that use SQL Distributed Management Objects (SQL-DMO).
 - Applications that use Transact-SQL and a standard database API.
- The definitions are stored by SQL Server in the **msdb** system database.
- When the SQLServerAgent service is started, it queries the system tables in the **msdb** database to determine what jobs and alerts to enable.
- SQL Server Agent executes jobs at their scheduled time.
- SQL Server passes any events that occur to the SQL Server Agent.

• SQL Server Agent executes any alerts, or sends SQL Mail requests to SQL Server, or sends **net send** commands to Windows.

SQL Server 2000 is more highly automated than SQL Server version 6.5 and earlier, and more efficiently tunes itself to meet processing demands. These features lower the potential for exception conditions that would trigger alerts. Scheduled jobs remain a good feature for implementing recurring tasks such as backup procedures.

See Also

Automating Administrative Tasks

Microsoft Search Service

The Microsoft Search service is a full-text indexing and search engine.

The SQL-92 standard defines only basic character-search capabilities:

- For a character value equal to, less than, or greater than a character constant.
- For a character value containing a string pattern.

Using the Microsoft Search service allows Microsoft® SQL Server[™] 2000 and SQL Server version 7.0 to support more sophisticated searches on character string columns.

The Microsoft Search service has two roles:

• Indexing support

Implements the full-text catalogs and indexes defined for a database. Accepts definitions of full-text catalogs, and the tables and columns comprising the indexes in each catalog. Implements requests to populate the full-text indexes.

• Querying support

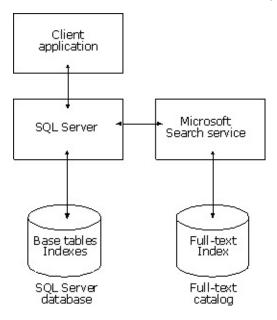
Processes full-text search queries. Determines which entries in the index meet the full-text selection criteria. For each entry that meets the selection criteria, it returns the identity of the row plus a ranking value to the SQL Server service, where this information is used to construct the query result set. The types of queries supported include searching for:

- Words or phrases.
- Words in close proximity to each other.
- Inflectional forms of verbs and nouns.

The full-text engine runs as a service named Microsoft Search on Microsoft Windows NT® or Microsoft Windows® 2000. It is installed when the Full-Text Search feature is selected during custom installation. The Microsoft Search service itself is not installed on Microsoft Windows 95 or Microsoft Windows 98, although Windows 95 and Windows 98 clients can make use of the service when connected to a SQL Server installation running on Windows NT or Windows 2000.

The Microsoft Search service runs in the context of the local system account. During setup, SQL Server adds itself as an administrator of the Microsoft Search service. To ensure this relationship is maintained correctly, all changes to the SQL Server service account information must be made using the **Properties** tab of the **SQL Server Properties** dialog box in SQL Server Enterprise Manager.

The full-text catalogs and indexes are not stored in a SQL Server database. They are stored in separate files managed by the Microsoft Search service. The full-text catalog files are accessible only to the Microsoft Search service and the Windows NT or Windows 2000 system administrator.



See Also

<u>Full-Text Catalogs and Indexes</u> <u>Full-Text Query Architecture</u>

MSSQLServerADHelper Service

The **MSSQLServerADHelper** service performs two functions:

- It adds and removes the objects used to register instances of Microsoft® SQL Server[™] 2000 relational database engine or Analysis server in the Microsoft Windows® 2000 Active Directory[™].
- It ensures that the Windows account under which a SQL Server service is running has permissions to update all of the Active Directory objects for the instance, as well as any replication publications and databases for that instance.

The service is dynamically started by an instance of SQL Server or the Analysis Manager when needed. The service is stopped as soon as it has completed its work.

Active Directory objects in a computer container can be created or removed only by programs that have been assigned either domain administration rights or that are running under the **localsystem** Windows account. Few sites run their SQL Server service under either of these types of accounts. A service application that does not perform network administration, such as SQL Server, is rarely granted full domain administration rights. The **localsystem** account cannot be given any privileges on remote computers; therefore, running SQL Server under this account would prevent much of the SQL Server distributed functionality from working. The **MSSQLServerADHelper** service is run under the **localsystem** account so that it can add and remove objects registering SQL Server entities in the Active Directory.

There is only one **MSSQLServerADHelper** service on a computer. The single service handles the Active Directory objects for all instances of the SQL Server relational database engine and all Analysis Manager applications running on the computer.

Registering SQL Server Analysis Servers

Analysis servers are registered from the Analysis Manager, which is a Microsoft

Management Console (MMC) application. When users of Analysis Manager request that an Analysis server be registered in the Active Directory, the application dynamically starts the **MSSQLServerADHelper** service and requests that it create an **MS-SQL-OLAPServer** object in the Active Directory. The helper service is stopped after the object has been completed, and the Analysis Manager finishes filling in the information for the object. For more information, see <u>Using Active Directory with Analysis Services</u>.

Registering SQL Server Relational Components

All management of the registrations of instances of SQL Server, and the databases and replication publications in each instance, are made using system stored procedures on the instance of SQL Server. SQL Server Enterprise Manager calls the system stored procedures when users specify Active Directory actions in the user interface. The procedures used are:

- **sp_ActiveDirectory_SCP.** Manages the registration of an instance of the relational database engine.
- **sp_addpublication**, **sp_addmergepublication**, **sp_changepublication**, or **sp_changemergepublication**. Manage the registration of replication publications.
- **sp_ActiveDirectory_Obj.** Manages the registration of a database.

Each of these system stored procedures internally call an internal component that use the Active Directory Services Interface (ADSI) to manage the objects. When an **MS-SQL-SQLServer** object must be added or removed from the Active Directory, or permissions granted, the SQL Server ADSI component calls the **MSSQLServerADHelper** service to perform the task. The SQL Server service uses the SQL Server ADSI component to dynamically start the **MSSQLServerADHelper** service as needed.

The SQL Server service dynamically calls the **MSSQLServerADHelper** service at these times:

• When an **MS-SQL-SQLServer** object must be created in the Active Directory to register an instance of SQL Server, the SQL Server service

calls **MSSQLServerADHelper** to create the object. **MSSQLServerADHelper** creates the object and gives update permissions to the Windows account under which the SQL Server service is running, and then **MSSQLServerADHelper** stops. The SQL Server service now has the permissions needed to maintain the object until it is removed. These permissions include creating **MS-SQL-SQLPublication** and **MS-SQL-SQLDatabase** objects as children of the **MS-SQL-SQLServer** object.

- If an administrator changes the Windows account under which the SQL Server service runs, the SQL Server service detects this the next time it attempts to update any information in objects that existed in the Active Directory before the account change. The SQL Server service automatically starts **MSSQLServerADHelper**. That service reassigns update permissions on the all the objects related to the current instance of SQL Server to the new Windows account.
- When a request is made to delete an **MS-SQL-SQLServer** object, the SQL Server ADSI component calls the **MSSQLServerADHelper** service to delete the object and any children that are still present.

The SQL Server service must be run under a Windows account that has permissions to start the **MSSQLServerADHelper** service. By default, members of the local Power Users and local Administrator's groups have this permission.

MS DTC Service

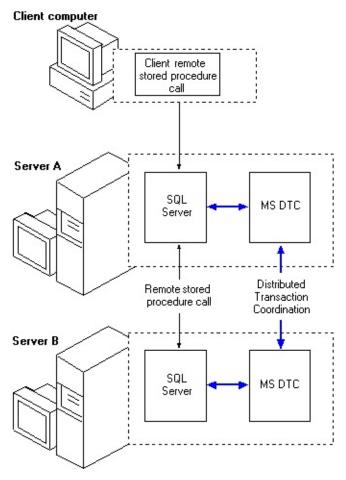
The Microsoft Distributed Transaction Coordinator (MS DTC) is a transaction manager that allows client applications to include several different sources of data in one transaction. MS DTC coordinates committing the distributed transaction across all the servers enlisted in the transaction.

An installation of Microsoft® SQL Server[™] can participate in a distributed transaction by:

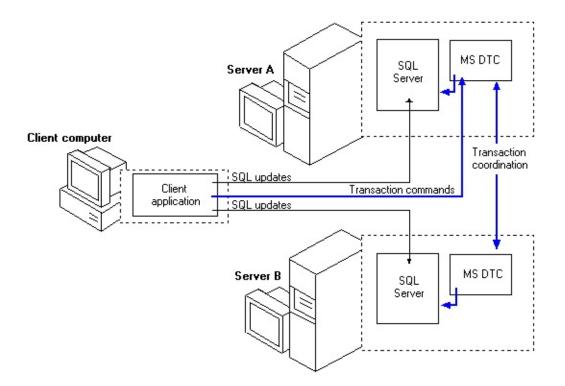
- Calling stored procedures on remote servers running SQL Server.
- Automatically or explicitly promoting the local transaction to a distributed transaction and enlist remote servers in the transaction.
- Making distributed updates that update data on multiple OLE DB data sources.

If these OLE DB data sources support the OLE DB distributed transaction interface, SQL Server can also enlist them in the distributed transaction.

The MS DTC service coordinates the proper completion of the distributed transaction to ensure that either all of the updates on all the servers are made permanent, or, in the case of errors, all erased.



SQL Server applications can also call MS DTC directly to start a distributed transaction explicitly. One or more servers running SQL Server can then be instructed to enlist in the distributed transaction and coordinate the proper completion of the transaction with MS DTC.



See Also

Distributed Transactions

SQL Server Architecture

Multiple Instances of SQL Server

Microsoft® SQL Server[™] 2000 supports multiple instances of the SQL Server database engine running concurrently on the same computer. Each instance of the SQL Server database engine has its own set of system and user databases that are not shared between instances. Applications can connect to each SQL Server database engine instance on a computer in much the same way they connect to SQL Server database engines running on different computers.

There are two types of instances of SQL Server:

Default Instances

The default instance of the SQL Server 2000 database engine operates the same way as the database engines in earlier versions of SQL Server. The default instance is identified solely by the name of the computer on which the instance is running, it does not have a separate instance name. When applications specify only the computer name in their requests to connect to SQL Server, the SQL Server client components attempt to connect to the default instance of the database engine on that computer. This preserves compatibility with existing SQL Server applications.

There can only be one default instance on any computer, the default instance can be any version of SQL Server.

Named Instances

All instances of the database engine other than the default instance are identified by an instance name specified during installation of the instance. Applications must provide both the computer name and the instance name of any named instance to which they are attempting to connect. The computer name and instance name are specified in the format *computer_name\instance_name*.

There can be multiple named instances running on a computer, but only the SQL Server 2000 database engine can operate as a named instance. The database engines from earlier versions of SQL Server cannot operate as a named instance.

Instances apply primarily to the database engine and its supporting components,

not to the client tools. When you install multiple instances, each instance gets a unique set of:

- System and user databases.
- The SQL Server and SQL Server Agent services. For default instances, the names of the services remain MSSQLServer and SQLServerAgent. For named instances, the names of the services are changed to MSSQL\$*instancename* and SQLAgent\$*instancename*, allowing them to be started and stopped independently of the other instances on the server. The database engines for the different instances are started and stopped using the associated SQL Server service. The SQL Server Agent services manage scheduled events for the associated instances of the database engine.
- The registry keys associated with the database engine and the SQL Server and SQL Server Agent services.
- Network connection addresses so that applications can connect to specific instances.

Shared Components

The following components are shared between all of the instances running on the same computer:

• There is only one SQL Server 2000 program group (Microsoft SQL Server) on the computer, and only one copy of the utility represented by each icon in the program group. There is only one copy of SQL Server Books Online.

The versions of the utilities in the program group are from the first version of SQL Server 2000 installed on the computer. For example, if you install the French version of SQL Server 2000 as a default instance and then the U.S. English version of SQL Server 2000 as a named instance, there is one SQL Server 2000 program group. All of the utility icons and the SQL Server Books Online icon in the program group start the French versions of the tools.

All of the SQL Server 2000 utilities work with multiple instances. You can start and stop each of the instances from a single copy of the SQL Server 2000 Service Manager. You can use a single copy of the SQL Server 2000 SQL Server Enterprise Manager to control objects in all instances on the computer, and use a single copy of the SQL Server 2000 Server Network Manager to manage the network addresses with which all of the instances on the computer communicate.

- There is only one copy of the MSSearchService that manages full-text searches against all of the instances of SQL Server on the computer.
- There is only one copy each of the English Query and Microsoft SQL Server 2000 Analysis Services servers.
- The registry keys associated with the client software are not duplicated between instances.
- There is only one copy of the SQL Server development libraries (include and .lib files) and sample applications.

Default Instances

Configurations that can operate as a default instance include:

- A default instance of SQL Server 2000.
- An installation of SQL Server version 7.0 operates as a default instance.
- An installation of SQL Server version 6.5 operates as a default instance.
- A default instance of SQL Server 2000 that can be version switched with an installation of SQL Server version 6.5 using the SQL Server 2000 **vswitch** utility.

• An installation of SQL Server version 7.0 that can be version switched with an installation of SQL Server version 6.5 using the SQL Server version 7.0 **vswitch** utility.

Note You must apply SQL Server 6.5 Service Pack 5 to any instance of SQL Server 6.5 before installing instances of SQL Server 2000 on the same computer.

Switching Between Versions of SQL Server

You cannot version switch between an installation of SQL Server version 7.0 and a default instance of SQL Server 2000.

You can have any number of named instances of SQL Server 2000 in addition to the default instance. You are not required to run a default instance on a computer before you can run named instances. You can run named instances on a computer that has no default instance. SQL Server version 6.5 and SQL Server 7.0 cannot operate as named instances, only as default instances.

Microsoft does not support more than 16 instances on a single computer or failover cluster.

If you run SQL Server version 6.5 as a default instance and run one or more named instances of SQL Server 2000 on a single computer, the computer has two SQL Server program groups instead of one SQL Server program group:

- A SQL Server 2000 program group executes the SQL Server 2000 tools.
- A SQL Server version 6.5 program group runs the SQL Server 6.5 tools.

If you are running SQL Server version 7.0 with SQL Server 2000, the icons in the SQL Server 7.0 program group will execute the SQL Server 2000 tools.

Note You must apply SQL Server 6.5 Service Pack 5 to any instance of SQL Server 6.5 before installing instances of SQL Server 2000 on the same computer.

Multiple Instances of SQL Server on a Failover Cluster

You can run only one instance of SQL Server on each virtual server of a SQL Server failover cluster, although you can install up to 16 virtual servers on a failover cluster. The instance can be either a default instance or a named instance. The virtual server looks like a single computer to applications connecting to that instance of SQL Server. When applications connect to the virtual server, they use the same convention as when connecting to any instance of SQL Server; they specify the virtual server name of the cluster and the optional instance name (only needed for named instances): *virtualservername\instancename*. For more information about clustering, see Failover Clustering Architecture.

Communicating with Multiple Instances

Each instance of Microsoft® SQL Server[™] 2000 listens on a unique set of network address so that applications can connect to different instances. SQL Server 2000 clients do not have to be configured to connect to an instance of SQL Server 2000. The SQL Server 2000 client components query a computer running instances of SQL Server 2000 to determine the Net-Libraries and network addresses for each instance. The client components then transparently choose a supported Net-Library and address for the connection without having to be configured on the client. The only information the application must supply is the computer name and instance name. For more information, see <u>Controlling Net-Libraries and Communications Addresses</u>.

A default instance of SQL Server 2000 listens on the same network addresses as earlier versions of SQL Server; therefore, applications using the client connectivity components of SQL Server version 7.0 or earlier can continue to connect to the default instance with no change. Named instances listen on alternative network addresses, and client computers using the client connectivity components of SQL Server version 7.0 or earlier must be set up to connect to the alternative addresses.

Using Multiple Instances

Although running multiple instances of Microsoft® SQL Server[™] 2000 on a single computer expands the capabilities of SQL Server, the recommended configuration for most production databases servers is to use a single instance of SQL Server with multiple databases.

Using a single instance of SQL Server on a production server offers these benefits:

- Only one instance needs to be administered.
- There is no duplication of components or processing overhead, such as having to run multiple database engines on the same computer. This means that the overall performance of a server with a single instance may be higher than a server running multiple instances.
- A single instance of SQL Server 2000 is capable of handling the processing growth requirements of the largest Web sites and enterprise data-processing systems, especially when it is part of a federation of database servers. For more information, see <u>Federated SQL Server 2000</u> <u>Database Servers</u>.

Running multiple instances of SQL Server on a single computer is best:

- When you must support different systems that have to be securely isolated from each other, such as when a service bureau has a large server and must create a separate instance of SQL Server for each customer.
- When you need to support multiple test and development databases, and the most economical configuration is to run these as separate instances of SQL Server on a single large server.

• When you need to run multiple applications on a desktop, and each application installs a separate instance of SQL Server 2000 Desktop Engine.

Working with Multiple Instances

Although multiple instances of Microsoft® SQL Server[™] 2000 can run on a single computer, there is no direct connection between instances. Each instance operates in many ways as if it is on a separate server. An application connected to one instance cannot access objects in databases created in another instance, except through distributed queries. Databases and database files cannot be shared between instances.

Named instances of SQL Server 2000 database engines have almost the same behaviors as default instances. The main difference is that you must supply both the computer name and instance name to identify a named instance. When you specify only *computername*, you work with the default instance. When you specify *computername*\instancename you work with the named instance.

• Service Manager.

When you specify only *computername* in Service Manager, you can stop and start the default instance. When you specify *computername\instancename* you can stop and start the named instance. When a specific instance is started, any database created in that instance is available to any application that connects to the instance using an authorization ID that has permissions to access the database.

• SQL Server Enterprise Manager.

Using SQL Server Enterprise Manager you can register each instance for which you have permissions. After an instance is registered, you can create, edit, and drop objects in the databases associated with that instance, subject to the permissions granted to you. You can also create, edit, and drop Data Transformation Services, Replication, and SQL Server Agent objects for that instance.

• Applications.

In an application, when you specify *computername* as the server name parameter in a connection request, you are connected to the default instance on the computer. You can access any databases in the default instance that you have permissions to access. If you specify *computername\instancename* as the server name parameter, you are connected to the named instance. You can access any databases in that named instance that you have permissions to access. When you are connected to a specific instance, objects in databases in other instances can be accessed only through distributed queries, just as objects in databases on other servers can be accessed only through distributed queries. Applications specify the instance name in different ways:

- ADO applications specify "Server=*computername**instancename*" in the provider string. For more information, see <u>Connecting to Multiple Instances of</u> <u>SQL Server</u>.
- OLE DB applications specify
 "Server=computername\instancename" in the provider string.
 They can alternatively set DBPROP_INIT_DATASOURCE to
 computername\\instancename (the backslash must be escaped
 with a second backslash). For more information, see
 <u>Establishing a Connection to a Data Source</u>.
- ODBC applications specify

"Server=*computername**instancename*" in the connection string specified on **SQLDriverConnect**. They can alternatively specify *computername**instancename* for the *ServerName* parameter on **SQLConnect**, or connect through a data source that has *computername**instancename* specified for the server name. For more information, see Support for <u>SQLDriverConnect</u> and <u>SQLConfigDataSource</u>.

- SQL DMO applications can manage instances of SQL Server 2000 using the **SQLServer2** object. For more information, see <u>SQLServer2 Object</u>.
- DB-Library and Embedded SQL for C do not support multiple

instances.

• Distributed queries and linked servers.

Distributed queries and linked server definitions use *computername**instancename* to identify named instances and *computername* to identify default instances. For more information, see <u>Distributed Queries on Multiple Instances of SQL Server</u>.

• Command prompt utilities.

When you use the command prompt utilities, you can use the Server switch to specify an instance by using *computername*\instancename, for example:

osql -E -Scomputer1\instance1 sqlservr /Sinstance1

The **isql** utility does not support named instances.

• SQL Server 2000 client components.

Applications using SQL Server 2000 client components can enumerate the instances available for connections:

- The OLE DB Provider for SQL Server 2000 returns instance names using **ISourcesRowset::GetSourcesRowset**. The names of named instances are returned as the data source name in the format *computername\instancename*, where *computername* can be either the name of a single computer or the virtual server name of a failover cluster. The names of default instances are returned as the data source name in the format *computername*, with no instance name.
- The SQL Server 2000 ODBC driver supports extensions to **SQLBrowseConnect** and **SQLSetConnectAttr** that allow applications to enumerate instances on a server. ODBC applications can also determine whether the *computername* is the name of a single computer or a virtual server name for a failover cluster. For more information, see

SQLBrowseConnect.

- SQL-DMO applications can enumerate instances using the **SQLServer2** object. The **SQLServer2** object also presents information such as the names of the SQL Server and SQL Server Agent services for the instance, or whether the instance is running on a single computer or a failover cluster. For more information, see <u>SQLServer2 Object</u>.
- DB-Library and Embedded SQL for C do not support named instances.

Identifying Instances

Performance Monitor counters, Profiler events, and Windows events in the Event Viewer Application Log all identify the instance of SQL Server with which they are associated.

The string returned by the @@SERVERNAME function identifies the name of the instance in the form *servername**instancename* if you are connected to a named instance. If connected to a default instance @@SERVERNAME returns only *servername*. For more information, see <u>@@SERVERNAME</u>.

The SERVERPROPERTY function INSTANCENAME property reports the instance name of the instance to which you are connected. INSTANCENAME returns NULL if connected to a default instance. In addition, the SERVERNAME property returns the same format string returned by @@SERVERNAME and will have the format *servername\instancename* when connected to a named instance. For more information, see <u>SERVERPROPERTY</u>.

Although the strings reported by @@SERVERNAME and SERVERNAME use the same format, the information they report can be different, for example:

• The string returned by @@SERVERNAME is affected by the actions of **sp_addserver** and **sp_dropserver**, and the string reported by SERVERNAME is not.

• SERVERNAME automatically reports changes in the network name of the computer, and @@SERVERNAME does not, unless **sp_dropserver** and **sp_addserver** are used to change the name it reports.

SQL Server Architecture

Federated SQL Server 2000 Servers

Microsoft® SQL Server[™] 2000 databases can be spread across a group of autonomous database servers capable of supporting the processing growth requirements of the largest Web sites and enterprise data-processing systems built with Microsoft Windows® DNA.

Windows DNA divides the processing units of a data processing system into logical tiers:

• User services tier

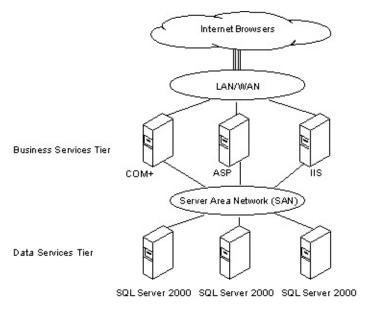
Presents the interface seen by the users, and typically calls the second tier for business logic processing.

• Business services tier

Contains the business logic that controls the operation of the Web site, and uses the persistent data storage provided by the third tier.

• Data services tier

Stores the persistent data required to run the Web site.



Scaling refers to the process of adding resources to a tier so that it can handle increased workloads. Scaling can be done in one of two ways:

• Scale up

Increases the processing power of a server by using a more powerful computer.

• Scale out

Increases the processing power of a system designed in a modular fashion, such as becoming a cluster of computers, by adding one or more additional computers, or nodes, to the system.

The growth requirements of the largest Web sites generate processing loads that exceed the capacity of large individual servers. In these cases, scaling out may be the best option for increasing the processing capacity of the system.

Microsoft Windows 2000 COM+ components are designed to be used in clusters of Windows 2000 application servers to form a clustered business services tier. Each server has identical sets of COM+ components, and Windows 2000 balances the cluster processing load by sending new requests to the server that has the least processing load. This forms an easily administered cluster that can quickly scale out by simply adding a new server.

SQL Server 2000 does not support this type of clustering. However, SQL Server 2000 does support updatable distributed partitioned views used to transparently partition data horizontally across a group of servers. Although these servers cooperate in managing the partitioned data, they operate autonomously. Each server is managed independently, has separate operational rules, and can support independent processes and data. A group of autonomous servers that cooperate to process a workload is known as a federation. Although SQL Server 2000 delivers very impressive performance when scaled up on servers with eight or more processors, it can support huge processing loads when partitioned across a federation. A federation of servers running SQL Server 2000 is capable of supporting the growth requirements of any Web site, or of the largest enterprise systems.

Partitioning Data

The first step in building a set of federated database servers is to horizontally partition the data in a set of tables across multiple servers. Horizontally partitioning a table refers to dividing a table into multiple smaller tables, called member tables. Each member table has the same format as the original table, but only part of the rows. Each table is placed on a separate resource (files or servers) to spread the processing load across the resources. For example, a company assigns customer identifiers (IDs) from 1 through 9999999. The **Customers** table may be partitioned into three member tables, with each member table having an equal customer ID range.

If used without views, horizontal partitioning would require applications to have logic to determine which member tables have the data requested by the user and dynamically build SQL statements referencing the tables. The application would require complex queries joining the member tables. Changing the member tables would also involve recoding the application. Views solve the problem by making the member tables look like one table. The SQL UNION operator combines result sets with identical formats into one. Because all the member tables have the same format, the result of SELECT * statements for each table have the same format, and can be combined using the UNION clause to form a single result set that operates similarly to the original table. For example, the **Customers** table has been partitioned across three servers (**Server1**, **Server2**, and **Server3**). The distributed partitioned view defined on **Server1** is:

```
CREATE VIEW Customers
AS
SELECT * FROM Customers_33
UNION ALL
SELECT * FROM Server2.CustomerDB.dbo.Customers_66
UNION ALL
SELECT * FROM Server3.CustomerDB.dbo.Customers_99
```

This view makes the actual location of the data transparent to an application. When a SQL statement is executed on **Server1** that references the **Customers**

partitioned view, the application has no visibility to where the data is located. If some of the rows required to complete the SQL statement reside on **Server2** or **Server3**, the instance of SQL Server on **Server1** automatically generates a distributed query that pulls in the required rows from the other servers. This transparency allows database administrators to repartition tables without recoding applications. If the **Customers** view is updatable, the behavior of the view is the same as a table named **Customers**.

Local partitioned views reference member tables on one server. Distributed partitioned views reference member tables on multiple servers. A server containing a member table is called a member server, and a database containing a member table is called a member database. Each member server contains one member table and a distributed partitioned view. An application that references the partitioned view on any of the servers gets the same results as if a complete copy of the original table were present on each server.

Microsoft SQL Server 2000 and Microsoft SQL Server version 7.0 support partitioned views; however, SQL Server 2000 introduces key features that allow the views to scale out and form federations of database servers:

- SQL Server 2000 partitioned views are updatable. This is crucial for distributing data so that the location of the data is transparent to the application. Updatable views support the full behavior of the original table; nonupdatable views are like read-only copies.
- The SQL Server 2000 query optimizer supports new optimizations that minimize the amount of distributed data that has to be transferred. The distributed execution plans generated by SQL Server 2000 result in good performance for a larger set of queries than the plans generated by SQL Server version 7.0.

SQL Server 2000 partitioned views are best suited for the types of SQL statements generated by Web sites and online transaction processing (OLTP) systems.

Partitioning a Database

To build an effective federation of database servers:

- Create multiple databases, each on a different member server running an instance of SQL Server 2000.
- Partition the individual tables in the original database so that most related data is placed together on a member server. This may require different methods of distributing the data in the various tables across all the member databases; partitioning some tables; making complete copies of other tables in each member database; and leaving some tables intact on the original server.
- Devise data routing rules that can be incorporated in the business services tier, so that applications can send each SQL statement to the member server that stores most of the data required by the statement.

The most important goal is to minimize distributed processing in such a system. You must be able to collocate related data on the same member server, and then route each SQL statement to a member server that contains most, if not all, of the data required to process the statement. For example, you may find that all the sales, customer, sales personnel, and inventory tables in a database can be partitioned by sales region, and that most SQL statements only reference data in a single region. You can then create member servers where each server has the horizontally partitioned data for one or more regions. If applications can identify the region currently referenced in the user's input, the application can submit any generated SQL statement to the member server containing the data for that region. The only SQL statements that will generate distributed queries are those that reference data from multiple regions. SQL Server Architecture

Failover Clustering Architecture

Microsoft® SQL Server[™] 2000 failover clustering increases server availability by allowing a system to automatically switch the processing for an instance of SQL Server from a failed server to a working server. For example, an instance of SQL Server can quickly restore database services to a Web site or enterprise network even if the server running the instance fails. SQL Server 2000 implements failover clustering based on the failover clustering features of the Microsoft Clustering Service (MSCS) in Windows NT® 4.0 and Windows® 2000.

The type of MSCS failover cluster used by SQL Server 2000 consists of multiple server computers (two on Windows NT 4.0, up to four on Windows 2000 Datacenter Server) that share a common set of cluster resources, such as disk drives. Each server in the cluster is called a node. Each server, or node, is connected to the network, and each node can communicate with each other node. Each node runs the same version of MSCS.

The shared resources in the failover cluster are collected into cluster groups. For example, if a failover cluster has four clustered disk drives, two of the drives can be collected in one cluster group and the other two in a second cluster group. Each cluster group is owned by one of the nodes in the failover cluster, although the ownership can be transferred between nodes.

Applications can be installed on the nodes in the failover cluster. These applications are typically server applications or distributed COM objects that users access through network connections. The application executables and other resources are typically stored in one or more of the cluster groups owned by the node. Each node can have multiple applications installed on it.

The failover cluster nodes periodically send each other network messages called heartbeat messages. If the MSCS software detects the loss of a heartbeat signal from one of the nodes in the cluster, it treats the server as a failed server. MSCS then automatically transfers the cluster groups and application resources of that node to the other nodes in the network. The cluster administrator specifies the alternate nodes to which cluster groups are transferred when any given node fails. The other nodes then continue processing user network requests for the applications transferred from the failed server. For more information about MSCS, see the Windows NT Server, Windows 2000 Server, Windows 2000 Advanced Server, or Windows 2000 Datacenter documentation.

SQL Server 2000 Failover Clusters

You can install up to 16 instances of Microsoft® SQL Server[™] 2000 in a Microsoft Clustering Service (MSCS) failover cluster.

You install an instance of SQL Server 2000 by running SQL Server Setup on one of the nodes of the cluster. The Setup program installs the instance on the nodes of the failover cluster that you specify during setup. The SQL Server 2000 executable files are installed on the local disk drives of each node in the failover cluster. This means that each node must have a local hard drive that is assigned the same drive letter as on all the other nodes, and that drive letter must be in the path of the location you specify for the SQL Server executable files during setup. For example, if you specify C:\Program Files\Microsoft SQL Server as the location in which to install the SQL Server executables, each node in the cluster must have drive letter C mapped to a local drive. The registry information for the instance is also stored in the registry of each node in the failover cluster.

An MSCS cluster group is a collection of clustered resources, such as clustered disk drives, which are owned by one of the failover cluster nodes. The ownership of the group can be transferred from one node to another, but each group can only be owned by one node at a time. The database files for an instance of SQL Server 2000 are placed in a single MSCS cluster group owned by the node on which you install the instance. If a node running an instance of SQL Server fails, MSCS switches the cluster group containing the data files for that instance to another node. Since the new node already has the executable files and registry information for that instance of SQL Server on its local disk drive, it can start up the instance of SQL Server and start accepting connection requests for that instance.

Because the executable files and registry information for each instance of SQL Server 2000 is stored in each node, the SQL Server 2000 limit of 16 instances per computer also applies to each failover cluster. Each instance in the failover cluster must either have a unique instance name or be a default instance. There can only be one default instance per failover cluster.

The MSCS cluster group that holds the database files for an instance is associated with a SQL Server virtual server name during SQL Server setup.

There can only be one instance per virtual server, which also means that there can only be one instance associated with any cluster group.

When an application attempts to connect to an instance of SQL Server 2000 running on a failover cluster, the application must specify both the virtual server name and the instance name. The application does not have to specify an instance name only if the instance associated with the virtual server is a default instance that does not have a name.

For example:

- A Windows cluster administrator creates a failover cluster with two nodes: **NodeA** and **NodeB**. Each node maps the drive letter C to a local hard drive.
- There is one shared disk in the cluster. The cluster administrator creates **ClusterGroupA** to hold the drive, and assigns it to **NodeA**.
- The SQL Server system administrator runs the Setup program to install a default instance of SQL Server on **NodeA**. During setup, the administrator specifies a SQL Server virtual server name of **VirtualServerX**, and specifies that the database files be placed on the drive in **ClusterGroupA**. Setup installs the SQL Server executable files on the local drives of both **NodeA** and **NodeB**, and places the database files in **ClusterGroupA**.
- Applications attempting to connect to the default instance only need to specify the virtual server name **VirtualServerA**. The default instance normally runs on **NodeA**. Should **NodeA** fail, however, the MSCS clustering will transfer ownership of **ClusterGroupA** to **NodeB** and will restart the default instance on **NodeB**. Applications will still connect to the default instance by specifying the virtual server name **VirtualServerX**.

See Also

Failover Clustering

Installing a Virtual Server Configuration

SQL Server Architecture

Active Directory Integration

The Microsoft® Windows® 2000 Active Directory[™] operates as a secure central resource for storing information about the users, devices, and services available on a Windows 2000 network. Microsoft SQL Server[™] 2000 supports registering instances of the SQL Server relational engine, databases, replication publications, and Analysis servers in the Active Directory. The SQL Server tools also provide a dialog box that supports browsing for replication publications registered in the Active Directory.

SQL Server Objects in the Active Directory Hierarchy

The Active Directory uses a hierarchy to represent the relationships between network entities such as users, services, and devices (such as computers, scanners, or printers). The hierarchy starts from a single root node at the top and branches down to leaf nodes representing individual entities in the network. The intermediate nodes in the hierarchy are containers that hold references to multiple entities. For example, several Windows users can be collected into a group for administrative purposes. Each node is implemented as an Active Directory object that represents the specific entity for that node.

When you register an instance of the SQL Server relational database engine in the Active Directory, an **MS-SQL-SQLServer** object is added as a Service Connection Point (SCP) object in the container for the computer on which the instance is running. An SCP is the type of Active Directory object that represents services available on the network. An SCP object records information about the service, such as connection information. An Analysis server is also registered as an SCP of the computer on which the Analysis server is running.

After registering an instance of the SQL Server relational database engine in the Active Directory, you can also register the replication publications that reside in the instance. The publications are registered as children of the instance. After registering replication publications in the Active Directory, the Create Pull Subscription Wizard supports a dialog box that allows users to search for registered publications in the Active Directory. For more information, see <u>Active Directory Services</u>.

After registering an instance of the relational database engine in the Active Directory, you can also register any databases in that instance. In SQL Server Enterprise Manager, right-click the database and select **Properties**. The **Options** tab has a check box at the bottom that controls whether the database is registered in the Active Directory. When you select the checkbox, the database is registered in the Active Directory when you close the **Properties** dialog box. After the check box is selected, the database object in the Active Directory is refreshed each time you close the **Properties** dialog box, provided the check box is selected when you open the **Properties** dialog box and remains checked when you click **OK** to close the dialog box. You can also use the **sp_ActiveDirectory_Obj** stored procedure to register databases from Transact-SQL scripts or applications.

You can register Analysis servers in the Active Directory. For more information, see <u>Using Active Directory with Analysis Services</u>. The SQL Server 2000 tools do not provide any facilities for browsing the Active Directory for instances of the relational database engine, Analysis servers, or relational databases. Applications can be coded to browse the Active Directory for the objects used to register these SQL Server entities.

The Active Directory class objects supported by SQL Server 2000 are defined in the Windows 2000 Active Directory schema:

Active Directory Object Name	SQL Server Entity
MS-SQL-SQLServer	An instance of SQL Server
MS-SQL-SQLPublication	A replication publication defined in an instance of SQL Server.
MS-SQL-SQLDatabase	A database in an instance of SQL Server.
MS-SQL-OLAPServer	An instance of the SQL Server Analysis server.

SQL Server 2000 makes no extensions to the definitions of these objects; SQL Server uses the objects as defined in the Windows 2000 Active Directory schema. Users can also code Active Directory Service Interfaces (ADSI) applications that browse the Active Directory for registered instances of SQL Server, Analysis servers, publications, and databases, For more information about ADSI and the structure of Active Directory schema objects, see the MSDN® Library at Microsoft Web site.

Note SQL Server 2000 does not use the **MS-SQL-OLAPCube**, **MS-SQL-OLAPDatabase**, or **MS-SQL-SQLRepository** class objects defined in the Windows 2000 Active Directory schema.

See Also

MSSQLServerADHelper Service_mssqlserveradhelper_service

sp_ActiveDirectory_SCP_sp_activedirectory_scp

SQL Server Architecture

SQL Server and XML Support

Extensible Markup Language (XML) is a hypertext programming language used to describe the contents of a set of data and how the data should be output to a device or displayed in a Web page. Markup languages originated as ways for publishers to indicate to printers how the content of a newspaper, magazine, or book should be organized. Markup languages for electronic data perform the same function for electronic documents that can be displayed on different types of electronic gear.

Both XML and the Hypertext Markup Language (HTML) are derived from Standard Generalized Markup Language (SGML). SGML is a very large, complex language that is difficult to fully use for publishing data on the Web. HTML is a more simple, specialized markup language than SGML, but has a number of limitations when working with data on the Web. XML is smaller than SGML and more robust than HTML, so is becoming an increasingly important language in the exchange of electronic data through the Web or intracompany networks.

In a relational database such as Microsoft® SQL Server[™] 2000, all operations on the tables in the database produce a result in the form of a table. The result set of a SELECT statement is in the form of a table. Traditional client/server applications that execute a SELECT statement process the results by fetching one row or block of rows from the tabular result set at a time and mapping the column values into program variables. Web application programmers, on the other hand, are more familiar with working with hierarchical representations of data in XML or HTML documents.

SQL Server 2000 introduces support for XML. These new features include:

- The ability to access SQL Server through a URL.
- Support for XML-Data schemas and the ability to specify XPath queries against these schemas.
- The ability to retrieve and write XML data:

- Retrieve XML data using the SELECT statement and the FOR XML clause.
- Write XML data using the OpenXML rowset provider.
- Enhancements to the Microsoft SQL Server 2000 OLE DB provider (SQLOLEDB) that allow XML documents to be set as command text and to return result sets as a stream.

See Also

XML and Internet Support Overview

Accessing SQL Server Using a URL

Creating XML Views Using Annotated Schemas

Using XPath Queries

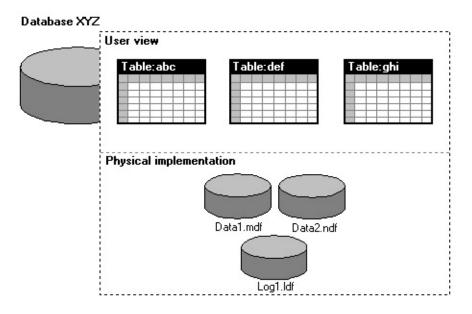
Retrieving and Writing XML Data

SQL Server Architecture

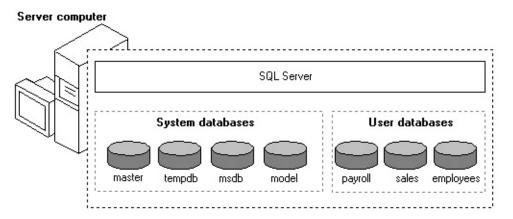
Database Architecture

Microsoft® SQL Server[™] 2000 data is stored in databases. The data in a database is organized into the logical components visible to users. A database is also physically implemented as two or more files on disk.

When using a database, you work primarily with the logical components such as tables, views, procedures, and users. The physical implementation of files is largely transparent. Typically, only the database administrator needs to work with the physical implementation.



Each instance of SQL Server has four system databases (**master**, **model**, **tempdb**, and **msdb**) and one or more user databases. Some organizations have only one user database, containing all the data for their organization. Some organizations have different databases for each group in their organization, and sometimes a database used by a single application. For example, an organization could have one database for sales, one for payroll, one for a document management application, and so on. Sometimes an application uses only one database; other applications may access several databases.



It is not necessary to run multiple copies of the SQL Server database engine to allow multiple users to access the databases on a server. An instance of the SQL Server Standard or Enterprise Edition is capable of handling thousands of users working in multiple databases at the same time. Each instance of SQL Server makes all databases in the instance available to all users that connect to the instance, subject to the defined security permissions.

When connecting to an instance of SQL Server, your connection is associated with a particular database on the server. This database is called the current database. You are usually connected to a database defined as your default database by the system administrator, although you can use connection options in the database APIs to specify another database. You can switch from one database to another using either the Transact-SQL USE *database_name* statement, or an API function that changes your current database context.

SQL Server 2000 allows you to detach databases from an instance of SQL Server, then reattach them to another instance, or even attach the database back to the same instance. If you have a SQL Server database file, you can tell SQL Server when you connect to attach that database file with a specific database name.

See Also

Database Design Considerations

Logical Database Components

The data in a Microsoft® SQL Server[™] 2000 database is organized into several different objects. These objects are what a user can see when they connect to the database.

In SQL Server 2000, these components are defined as objects:

Constraints	Tables
Defaults	Triggers
Indexes	User-defined data types
Keys	User-defined functions
Stored procedures	Views

Data Types and Table Structures

All the data in Microsoft® SQL Server[™] 2000 databases is contained in objects called tables. Each table represents some type of object meaningful to the users. For example, in a school database you would find tables such as a class table, an instructor table, and a student table.

SQL Server tables have two main components:

• Columns

Each column represents some attribute of the object modeled by the table, such as a parts table having columns for ID, color, and weight.

• Rows

Each row represents an individual occurrence of the object modeled by the table. For example, the parts table would have one row for each part carried by the company.

parts table		
ID	color	weight
AB123	Blue	10.5
CD 456	Red	8.0
EF789	Green	9.25
GH012	Yellow	8.0
IJ341	Blue	1.0

Data Types

Because each column represents one attribute of an object, the data in each occurrence of the column is similar. One of the properties of a column is called its data type, which defines the type of data the column can hold. SQL Server has several base data types that can be specified for columns:

binary	Bigint	bit	Char	datetime
decimal	Float	image	Int	Money

nchar	Ntext	nvarchar	Numeric	Real
smalldatetime	smallint	smallmoney	sql_variant	sysname
text	timestamp	tinyint	varbinary	varchar
uniqueidentifier				

SQL Server 2000 also supports a **table** base data type, which can be used to store the result set of an SQL statement. The **table** data type cannot be used for columns in a table. It can only be used for Transact-SQL variables and the return values of user-defined functions. For more information, see <u>Using Special Data</u>.

Users can also create their own user-defined data types, for example:

-- Create a birthday data type that allows nulls. EXEC sp_addtype birthday, datetime, 'NULL' GO -- Create a table using the new data type. CREATE TABLE employee (emp_id char(5), emp_first_name char(30), emp_last_name char(40), emp_birthday birthday)

A user-defined data type makes a table structure more meaningful to programmers and helps ensure that columns holding similar classes of data have the same base data type.

SQL Server provides several data type synonyms to help support SQL-92 data type names not included as base data types, such as national character and character varying. When a synonym is specified in a CREATE TABLE statement, the column is assigned the base data type associated with the synonym. For more information, see Data Type Synonyms.

A <u>domain</u> is the set of all allowable values in a column. It includes not only the concept of enforcing data types, but also the values allowed in the column. For example, a part color domain would include both the data type, such as **char(6)**, and the character strings allowed in the column, such as Red, Blue, Green, Yellow, Brown, Black, White, Teal, Grey, and Silver. Domain values can be enforced through mechanisms such as CHECK constraints and triggers.

When a column has been assigned a data type, all values placed into the column must be of that data type. SQL statements can specify that values of different data types be used as the source value only if SQL Server can implicitly convert the source value data type to the data type of the column. For example, SQL Server supports the implicit conversion of **int** values to **decimal**; therefore, SQL statements can specify **int** values as the value to be assigned to a **decimal** column.

The SQL Server 2000 **sql_variant** data type is a special data type that allows you to store values of multiple base data types in the same column. For example, you can store **nchar** values, **int** values, and **decimal** values in the same column. For more information, see <u>Using sql_variant Data</u>.

Null Values

Columns can either accept or reject null values. NULL is a special value in databases that represents the concept of an unknown value. NULL is not the same as a blank character or 0. Blank is actually a valid character, and 0 is a valid number. NULL simply represents the idea that we do not know what this value is. NULL is also different from a zero-length string. If a column definition contains the NOT NULL clause, you cannot insert rows having the value NULL for that row. If the column definition has only the NULL keyword, it accepts NULL values.

Allowing NULL values in a column can increase the complexity of any logical comparisons using the column. The SQL-92 standard states that any comparison against a NULL value does not evaluate to TRUE or FALSE, it evaluates to UNKNOWN. This introduces three-value logic to comparison operators, which can be difficult to manage correctly.

System Tables

SQL Server stores the data defining the configuration of the server and all its tables in a special set of tables known as system tables. Users should not query or update the system tables directly unless there is no other way to get the data required by the application. Only SQL Server should reference the system tables in response to administration commands issued by users. The system tables can change from version to version; applications referencing system tables directly may have to be rewritten before they can be upgraded to a newer version of SQL

Server with a different version of the system tables. SQL Server exposes most of the information from the system tables through other means. For more information, see <u>System Tables</u>.

Temporary Tables

SQL Server supports temporary tables. These tables have names that start with a number sign (#). If a temporary table is not dropped when a user disconnects, SQL Server automatically drops the temporary table. Temporary tables are not stored in the current database; they are stored in the **tempdb** system database.

There are two types of temporary tables:

• Local temporary tables

The names of these tables begin with one number sign (#). These tables are visible only to the connection that created them.

• Global temporary tables

The names of these tables begin with two number signs (##). These tables are visible to all connections. If the tables are not dropped explicitly before the connection that created them disconnects, they are dropped as soon as all other tasks stop referencing them. No new tasks can reference a global temporary table after the connection that created it disconnects. The association between a task and a table is always dropped when the current statement completes executing; therefore, global temporary tables are usually dropped soon after the connection that created them disconnects.

Many traditional uses of temporary tables can now be replaced with variables that have the **table** data type.

Working with Tables

Users work with the data in tables using data manipulation language (DML) SQL statements:

-- Get a list of all employees named Smith: SELECT emp_first_name, emp_last_name

FROM employee WHERE emp_last_name = 'Smith'

-- Delete an employee who quit: DELETE employee WHERE emp_id = 'OP123'

-- Add a new employee: INSERT INTO employee VALUES ('OP456', 'Dean', 'Straight', '01/01/1960')

-- Change an employee name: UPDATE employee SET emp_last_name = 'Smith' WHERE emp_id = 'OP456'

See Also

Specifying a Column Data Type

<u>Tables</u>

SQL Views

A view can be thought of as either a virtual table or a stored query. The data accessible through a view is not stored in the database as a distinct object. What is stored in the database is a SELECT statement. The result set of the SELECT statement forms the virtual table returned by the view. A user can use this virtual table by referencing the view name in Transact-SQL statements the same way a table is referenced. A view is used to do any or all of these functions:

• Restrict a user to specific rows in a table.

For example, allow an employee to see only the rows recording his or her work in a labor-tracking table.

• Restrict a user to specific columns.

For example, allow employees who do not work in payroll to see the name, office, work phone, and department columns in an employee table, but do not allow them to see any columns with salary information or personal information.

- Join columns from multiple tables so that they look like a single table.
- Aggregate information instead of supplying details.

For example, present the sum of a column, or the maximum or minimum value from a column.

Views are created by defining the SELECT statement that retrieves the data to be presented by the view. The data tables referenced by the SELECT statement are known as the base tables for the view. In this example, **titleview** in the **pubs** database is a view that selects data from three base tables to present a virtual table of commonly needed data:

CREATE VIEW titleview AS SELECT title, au_ord, au_lname, price, ytd_sales, pub_id FROM authors AS a JOIN titleauthor AS ta ON (a.au_id = ta.au_id) JOIN titles AS t ON (t.title_id = ta.title_id)

You can then reference **titleview** in statements in the same way you would reference a table:

SELECT * FROM titleview

A view can reference another view. For example, **titleview** presents information that is useful for managers, but a company typically discloses year-to-date figures only in quarterly or annual financial statements. A view can be built that selects all the **titleview** columns except **au_ord** and **ytd_sales**. This new view can be used by customers to get lists of available books without seeing the financial information:

CREATE VIEW Cust_titleview AS SELECT title, au_lname, price, pub_id FROM titleview

Views can be used to partition data across multiple databases or instances of Microsoft® SQL Server[™] 2000. Partitioned views can be used to distribute database processing across a group of servers. The group of servers has the same performance benefits as a cluster of servers, and can be used to support the processing needs of the largest Web sites or corporate data centers. An original table is subdivided into several member tables, each of which has a subset of the rows from the original table. Each member table can be placed in databases on separate servers. Each server also gets a partitioned view. The partitioned view uses the Transact-SQL UNION operator to combine the results of selects against all the member tables into a single result set that behaves exactly like a copy of the full original table. For example, a table is partitioned across three servers. On the first server you define a partitioned view similar to this:

CREATE VIEW PartitionedView AS SELECT *

FROM MyDatabase.dbo.PartitionTable1

UNION ALL SELECT * FROM Server2.MyDatabase.dbo.PartitionTable2 UNION ALL SELECT * FROM Server3.MyDatabase.dbo.PartitionTable3

You define similar partitioned views on each of the other servers. With these three views, any Transact-SQL statements on any of the three servers that reference **PartitionedView** will see the same behavior as from the original table. It is as if a copy of the original table exists on each server, when in fact there is only one member table and a partitioned view on each table. For more information, see <u>Scenarios for Using Views</u>.

Views in all versions of SQL Server are updatable (can be the target of UPDATE, DELETE, or INSERT statements), as long as the modification affects only one of the base tables referenced by the view, for example:

```
-- Increase the prices for publisher '0736' by 10%.
UPDATE titleview
SET price = price * 1.10
WHERE pub_id = '0736'
GO
```

SQL Server 2000 supports more complex types of INSERT, UPDATE, and DELETE statements that reference views. INSTEAD OF triggers can be defined on a view to specify the individual updates that must be performed against the base tables to support the INSERT, UPDATE, or DELETE statement. Also, partitioned views support INSERT, UDPATE, and DELETE statements that modify multiple member tables referenced by the view.

Indexed views are a SQL Server 2000 feature that greatly improves the performance of complex views of the type usually found in data warehouses or other decision support systems.

Views are called virtual tables because the result set of a view is us not usually saved in the database The result set for a view is dynamically incorporated into the logic of the statement and the result set is built dynamically at run time. For more information, see <u>View Resolution</u>.

Complex queries, such as those in decision support systems, can reference large numbers of rows in base tables, and aggregate large amounts of information into relatively concise aggregates such as sums or averages. SQL Server 2000 supports creating a clustered index on a view that implements such a complex query. When the CREATE INDEX statement is executed the result set of the view SELECT is stored permanently in the database. Future SQL statements that reference the view will have substantially better response times. Modifications to the base data are automatically reflected in the view.

The SQL Server 2000 CREATE VIEW statement supports a SCHEMABINDING option that prevents the tables referenced by the view being changed without adjusting the view. You must specify SCHEMABINDING for any view on which you create an index.

See Also

<u>CREATE INDEX</u> <u>CREATE TRIGGER</u> <u>CREATE VIEW</u> <u>Designing an Indexed View</u> Views

SQL Stored Procedures

A <u>stored procedure</u> is a group of Transact-SQL statements compiled into a single execution plan.

Microsoft® SQL Server[™] 2000 stored procedures return data in four ways:

- Output parameters, which can return either data (such as an integer or character value) or a cursor variable (cursors are result sets that can be retrieved one row at a time).
- Return codes, which are always an integer value.
- A result set for each SELECT statement contained in the stored procedure or any other stored procedures called by the stored procedure.
- A global cursor that can be referenced outside the stored procedure.

Stored procedures assist in achieving a consistent implementation of logic across applications. The SQL statements and logic needed to perform a commonly performed task can be designed, coded, and tested once in a stored procedure. Each application needing to perform that task can then simply execute the stored procedure. Coding business logic into a single stored procedure also offers a single point of control for ensuring that business rules are correctly enforced.

Stored procedures can also improve performance. Many tasks are implemented as a series of SQL statements. Conditional logic applied to the results of the first SQL statements determines which subsequent SQL statements are executed. If these SQL statements and conditional logic are written into a stored procedure, they become part of a single execution plan on the server. The results do not have to be returned to the client to have the conditional logic applied; all of the work is done on the server. The IF statement in this example shows embedding conditional logic in a procedure to keep from sending a result set to the application:

IF (@QuantityOrdered < (SELECT QuantityOnHand

```
FROM Inventory
WHERE PartID = @PartOrdered))
```

BEGIN

-- SQL statements to update tables and process order.

END

ELSE

BEGIN

-- SELECT statement to retrieve the IDs of alternate items

```
-- to suggest as replacements to the customer.
```

END

Applications do not need to transmit all of the SQL statements in the procedure: they have to transmit only an EXECUTE or CALL statement containing the name of the procedure and the values of the parameters.

Stored procedures can also shield users from needing to know the details of the tables in the database. If a set of stored procedures supports all of the business functions users need to perform, users never need to access the tables directly; they can just execute the stored procedures that model the business processes with which they are familiar.

An illustration of this use of stored procedures is the SQL Server system stored procedures used to insulate users from the system tables. SQL Server includes a set of system stored procedures whose names usually start with **sp_**. These system stored procedures support all of the administrative tasks required to run a SQL Server system. You can administer a SQL Server system using the Transact-SQL administration-related statements (such as CREATE TABLE) or the system stored procedures, and never need to directly update the system tables.

Stored Procedures and Execution Plans

In SQL Server version 6.5 and earlier, stored procedures were a way to partially precompile an execution plan. At the time the stored procedure was created, a partially compiled execution plan was stored in a system table. Executing a stored procedure was more efficient than executing an SQL statement because SQL Server did not have to compile an execution plan completely, it only had to

finish optimizing the stored plan for the procedure. Also, the fully compiled execution plan for the stored procedure was retained in the SQL Server procedure cache, meaning that subsequent executions of the stored procedure could use the precompiled execution plan.

SQL Server 2000 and SQL Server version 7.0 incorporate a number of changes to statement processing that extend many of the performance benefits of stored procedures to all SQL statements. SQL Server 2000 and SQL Server 7.0 do not save a partially compiled plan for stored procedures when they are created. A stored procedure is compiled at execution time, like any other Transact-SQL statement. SQL Server 2000 and SQL Server 7.0 retain execution plans for all SQL statements in the procedure cache, not just stored procedure execution plans. The database engine uses an efficient algorithm for comparing new Transact-SQL statements with the Transact-SQL statements of existing execution plans. If the database engine determines that a new Transact-SQL statement matches the Transact-SQL statement of an existing execution plan, it reuses the plan. This reduces the relative performance benefit of precompiling stored procedures by extending execution plan reuse to all SQL statements.

SQL Server 2000 and SQL Server version 7.0 offer new alternatives for processing SQL statements. For more information, see <u>Query Processor</u> <u>Architecture</u>.

Temporary Stored Procedures

SQL Server 2000 also supports temporary stored procedures that, like temporary tables, are dropped automatically when you disconnect. Temporary stored procedures are stored in **tempdb** and are useful when connected to earlier versions of SQL Server. Temporary stored procedures can be used when an application builds dynamic Transact-SQL statements that are executed several times. Rather than have the Transact-SQL statements recompiled each time, you can create a temporary stored procedure that is compiled on the first execution, and then execute the precompiled plan multiple times. Heavy use of temporary stored procedures, however, can lead to contention on the system tables in **tempdb**.

Two features of SQL Server 2000 and SQL Server 7.0 eliminate the need for using temporary stored procedures:

- Execution plans from prior SQL statements can be reused. This is especially powerful when coupled with the use of the new **sp_executesql** system stored procedure.
- Natively support for the prepare/execute model of OLE DB and ODBC without using any stored procedures.

For more information about alternatives to using temporary stored procedures, see <u>Execution Plan Caching and Reuse</u>.

Stored Procedure Example

This simple stored procedure example illustrates three ways stored procedures can return data:

- 1. It first issues a SELECT statement that returns a result set summarizing the order activity for the stores in the **sales** table.
- 2. It then issues a SELECT statement that fills an output parameter.
- 3. Finally, it has a RETURN statement with a SELECT statement that returns an integer. Return codes are generally used to pass back error checking information. This procedure runs without errors, so it returns another value to illustrate how returned codes are filled.

USE Northwind

GO

DROP PROCEDURE OrderSummary

GO

CREATE PROCEDURE OrderSummary @MaxQuantity INT OUTPU -- SELECT to return a result set summarizing

-- employee sales.

SELECT Ord.EmployeeID, SummSales = SUM(OrDet.UnitPrice * Or FROM Orders AS Ord

JOIN [Order Details] AS OrDet ON (Ord.OrderID = OrDet.OrderII

GROUP BY Ord.EmployeeID ORDER BY Ord.EmployeeID

-- SELECT to fill the output parameter with the -- maximum quantity from Order Details. SELECT @MaxQuantity = MAX(Quantity) FROM [Order Details]

-- Return the number of all items ordered. RETURN (SELECT SUM(Quantity) FROM [Order Details]) GO

-- Test the stored procedure.

-- DECLARE variables to hold the return code -- and output parameter. DECLARE @OrderSum INT DECLARE @LargestOrder INT

-- Execute the procedure, which returns

-- the result set from the first SELECT.

EXEC @OrderSum = OrderSummary @MaxQuantity = @LargestOrd

-- Use the return code and output parameter. PRINT 'The size of the largest single order was: ' + CONVERT(CHAR(6), @LargestOrder) PRINT 'The sum of the quantities ordered was: ' + CONVERT(CHAR(6), @OrderSum) GO

The output from running this sample is:

EmployeeID SummSales

1 202,143.71

2	177,749.26
	,

- 3 213,051.30
- 4 250,187.45
- 5 75,567.75
- 6 78,198.10
- 7 141,295.99
- 8 133,301.03
- 9 82,964.00

The size of the largest single order was: 130 The sum of the quantities ordered was: 51317

See Also

Stored Procedures

SQL User-Defined Functions

Functions in programming languages are subroutines used to encapsulate frequently performed logic. Any code that must perform the logic incorporated in a function can call the function rather than having to repeat all of the function logic.

Microsoft[®] SQL Server[™] 2000 supports two types of functions:

• Built-in functions

Operate as defined in the Transact-SQL Reference and cannot be modified. The functions can be referenced only in Transact-SQL statements using the syntax defined in the Transact-SQL Reference. For more information about these built-in functions, see <u>Using Functions</u>.

• User-defined functions

Allow you to define your own Transact-SQL functions using the CREATE FUNCTION statement. For more information about these built-in functions, see <u>User-defined Functions</u>.

User-defined functions take zero or more input parameters, and return a single value. Some user-defined functions return a single, scalar data value, such as an **int**, **char**, or **decimal** value.

For example, this statement creates a simple function that returns a decimal:

CREATE FUNCTION CubicVolume

-- Input dimensions in centimeters.

(@CubeLength decimal(4,1), @CubeWidth decimal(4,1),

@CubeHeight decimal(4,1))

RETURNS decimal(12,3) -- Cubic Centimeters.

AS

BEGIN

RETURN (@CubeLength * @CubeWidth * @CubeHeight) END This function can then be used anywhere an integer expression is allowed, such as in a computed column for a table:

```
CREATE TABLE Bricks
 (
  BrickPartNmbr int PRIMARY KEY,
  BrickColor
               nchar(20),
  BrickHeight
                decimal(4,1),
                decimal(4,1),
  BrickLength
  BrickWidth
                decimal(4,1),
  BrickVolume AS
        (
        dbo.CubicVolume(BrickHeight,
              BrickLength, BrickWidth)
        )
 )
```

SQL Server 2000 also supports user-defined functions that return a **table** data type:

- A function can declare an internal **table** variable, insert rows into the variable, and then return the variable as its return value.
- A class of user-defined functions known as in-line functions, return the result set of a SELECT statement as a variable of type **table**.

These functions can be used in places where table expressions can be specified. For more information about the **table** data type, see <u>Using Special Data</u>.

User-defined functions that return a **table** can be powerful alternatives to views. A user-defined function that returns a **table** can be used where table or view expressions are allowed in Transact-SQL queries. Views are limited to a single SELECT statement; however, user-defined functions can contain additional statements that allow more powerful logic than is possible in views.

A user-defined function that returns a **table** can also replace stored procedures that return a single result set. The **table** returned by a user-defined function can

be referenced in the FROM clause of a Transact-SQL statement, whereas stored procedures that return result sets cannot. For example, **fn_EmployeesInDept** is a user-defined function that returns a **table** and can be invoked by a SELECT statement:

```
SELECT *
FROM tb_Employees AS E,
dbo.fn_EmployeesInDept('shipping') AS EID
WHERE E.EmployeeID = EID.EmployeeID
```

This is an example of a statement that creates a function in the **Northwind** database that will return a **table**:

```
CREATE FUNCTION LargeOrderShippers ( @FreightParm money )
RETURNS @OrderShipperTab TABLE
```

```
(
  ShipperID
             int,
  ShipperName nvarchar(80),
  OrderID
             int,
  ShippedDate datetime,
  Freight
            money
 )
AS
BEGIN
 INSERT @OrderShipperTab
    SELECT S.ShipperID, S.CompanyName,
        O.OrderID, O.ShippedDate, O.Freight
    FROM Shippers AS S
       INNER JOIN Orders AS O ON (S.ShipperID = O.ShipVia)
    WHERE O.Freight > @FreightParm
 RETURN
END
```

In this function, the local return variable name is **@OrderShipperTab**. Statements in the function build the **table** result returned by the function by inserting rows into the variable **@OrderShipperTab**. External statements invoke the function to reference the **table** returned by the function:

SELECT *
FROM LargeOrderShippers(\$500)

Constraints, Rules, Defaults, and Triggers

Table columns have properties other than data type and size. These other properties are an important part in ensuring the integrity of data in a database:

• Data integrity refers to each occurrence of a column having a correct data value.

The data values must be of the right data type and in the correct domain.

• Referential integrity indicates that the relationships between tables have been properly maintained.

Data in one table should only point to existing rows in another table; it should not point to rows that do not exist.

Objects used to maintain both types of integrity include:

- Constraints
- Rules
- Defaults
- Triggers

Constraints

Constraints allow you to define the way Microsoft® SQL Server[™] 2000 automatically enforces the integrity of a database. Constraints define rules regarding the values allowed in columns and are the standard mechanism for enforcing integrity. Using constraints is preferred to using triggers, rules, and defaults. The query optimizer also uses constraint definitions to build high-performance query execution plans.

Classes of Constraints

SQL Server 2000 supports five classes of constraints.

- NOT NULL specifies that the column does not accept NULL values.
- CHECK constraints enforce domain integrity by limiting the values that can be placed in a column.

A CHECK constraint specifies a Boolean (evaluates to TRUE or FALSE) search condition that is applied to all values entered for the column; all values that do not evaluate to TRUE are rejected. You can specify multiple CHECK constraints for each column. This sample shows the creation of a named constraint, **chk_id**, that further enforces the domain of the primary key by ensuring that only numbers within a specified range are entered for the key.

CREATE TABLE cust_sample

```
(

cust_id int PRIMARY KEY,

cust_name char(50),

cust_address char(50),

cust_credit_limit money,

CONSTRAINT chk_id CHECK (cust_id BETWEEN 0 and 1)

)
```

• UNIQUE constraints enforce the uniqueness of the values in a set of columns.

No two rows in the table are allowed to have the same not null values for the columns in a UNIQUE constraint. Primary keys also enforce uniqueness, but primary keys do not allow null values. A UNIQUE constraint is preferred over a unique index.

• PRIMARY KEY constraints identify the column or set of columns whose values uniquely identify a row in a table.

No two rows in a table can have the same primary key value. You cannot enter a NULL for any column in a primary key. NULL is a special value in databases that represents an unknown value, which is distinct from a blank or 0 value. Using a small, integer column as a primary key is recommended. Each table should have a primary key.

A table may have more than one combination of columns that could uniquely identify the rows in a table; each combination is a candidate key. The database administrator picks one of the candidate keys to be the primary key. For example, in the **part_sample** table both **part_nmbr** and **part_name** could be candidate keys, but only **part_nmbr** is chosen as a primary key.

CREATE TABLE part_sample

1	— I	
(part_nmbr	int	PRIMARY KEY,
part_name	char(30),	
part_weight	decimal	(6,2),
part_color	char(15))	

• FOREIGN KEY constraints identify the relationships between tables.

A foreign key in one table points to a candidate key in another table. Foreign keys prevent actions that would leave rows with foreign key values when there are no candidate keys with that value. In the following sample, the **order_part** table establishes a foreign key referencing the **part_sample** table defined earlier. Usually, **order_part** would also have a foreign key against an order table, but this is a simple example.

```
CREATE TABLE order_part
(order_nmbr int,
part_nmbr int
FOREIGN KEY REFERENCES part_sample(part_nmb
ON DELETE NO ACTION,
qty_ordered int)
GO
```

You cannot insert a row with a foreign key value (except NULL) if there is no candidate key with that value. The ON DELETE clause controls what actions are taken if you attempt to delete a row to which existing foreign keys point. The ON DELETE clause has two options:

- NO ACTION specifies that the deletion fails with an error.
- CASCADE specifies that all the rows with foreign keys pointing to the deleted row are also deleted.

The ON UPDATE clause defines the actions that are taken if you attempt to update a candidate key value to which existing foreign keys point. It also supports the NO ACTION and CASCADE options.

Column and Table Constraints

Constraints can be column constraints or table constraints:

- A column constraint is specified as part of a column definition and applies only to that column (the constraints in the earlier samples are column constraints).
- A table constraint is declared independently from a column definition and can apply to more than one column in a table.

Table constraints must be used when more than one column must be included in a constraint.

For example, if a table has two or more columns in the primary key, you must

use a table constraint to include both columns in the primary key. Consider a table that records events happening in a computer in a factory. Assume that events of several types can happen at the same time, but that no two events happening at the same time can be of the same type. This can be enforced in the table by including both the **type** and **time** columns in a two-column primary key.

CREATE TABLE factory_process

(event_type int, event_time datetime, event_site char(50), event_desc char(1024), CONSTRAINT event_key PRIMARY KEY (event_type, event_time))

See Also

CREATE TABLE

Creating and Modifying a Table

Rules

Rules are a backward-compatibility feature that perform some of the same functions as CHECK constraints. CHECK constraints are the preferred, standard way to restrict the values in a column. CHECK constraints are also more concise than rules; there can only be one rule applied to a column, but multiple CHECK constraints can be applied. CHECK constraints are specified as part of the CREATE TABLE statement, while rules are created as separate objects and then bound to the column.

This example creates a rule that performs the same function as the CHECK constraint example in the preceding topic. The CHECK constraint is the preferred method to use in Microsoft® SQL Server[™] 2000.

```
CREATE RULE id chk AS @id BETWEEN 0 and 10000
GO
CREATE TABLE cust_sample
 (
 cust id
              int
 PRIMARY KEY,
                char(50),
 cust name
                  char(50),
 cust address
 cust_credit_limit money,
 )
GO
sp_bindrule id_chk, 'cust_sample.cust_id'
GO
```

See Also

CREATE TABLE

Creating and Modifying a Table

Defaults

Defaults specify what values are used in a column if you do not specify a value for the column when inserting a row. Defaults can be anything that evaluates to a constant, such as:

- Constant
- Built-in function
- Mathematical expression

There are two ways to apply defaults:

• Create a default definition using the DEFAULT keyword in CREATE TABLE to assign a constant expression as a default on a column.

This is the preferred, standard method. It is also the more concise way to specify a default.

• Create a default object using the CREATE DEFAULT statement and bind it to columns using the **sp_bindefault** system stored procedure.

This is a backward compatibility feature.

This example creates a table using one of each type of default. It creates a default object to assign a default to one column, and binds the default object to the column. It then does a test insert without specifying values for the columns with defaults and retrieves the test row to verify the defaults were applied.

```
USE pubs
GO
CREATE TABLE test_defaults
(keycol smallint,
process_id smallint DEFAULT @@SPID, --Preferred default defin
date_ins datetime DEFAULT getdate(), --Preferred default definiti
```

smallint DEFAULT 10 * 2, --Preferred default definition mathcol char1 char(3), char(3) DEFAULT 'xyz') -- Preferred default definition char2 GO /* Illustration only, use DEFAULT definitions instead.*/ CREATE DEFAULT abc_const AS 'abc' GO sp_bindefault abc_const, 'test_defaults.char1' GO INSERT INTO test_defaults(keycol) VALUES (1) GO SELECT * FROM test_defaults GO

The output of this sample is:

Default bound to column.

(1 row(s) affected)

keycol process_id date_insmathcol char1 char217Oct 16 1997 8:34PM20abcxyz

(1 row(s) affected)

See Also

CREATE TABLE

Creating and Modifying a Table

Triggers

Microsoft® SQL Server[™] 2000 triggers are a special class of stored procedure defined to execute automatically when an UPDATE, INSERT, or DELETE statement is issued against a table or view. Triggers are powerful tools that sites can use to enforce their business rules automatically when data is modified. Triggers can extend the integrity checking logic of SQL Server constraints, defaults, and rules, although constraints and defaults should be used instead whenever they provide all the needed functionality.

Tables can have multiple triggers. The CREATE TRIGGER statement can be defined with the FOR UPDATE, FOR INSERT, or FOR DELETE clauses to target a trigger to a specific class of data modification actions. When FOR UPDATE is specified, the IF UPDATE (*column_name*) clause can be used to target a trigger to updates affecting a particular column.

Triggers can automate the processing for a company. In an inventory system, update triggers can detect when a stock level reaches a reorder point and generate an order to the supplier automatically. In a database recording the processes in a factory, triggers can e-mail or page operators when a process exceeds defined safety limits.

The following trigger generates an e-mail whenever a new title is added in the **pubs** database:

CREATE TRIGGER reminder ON titles FOR INSERT AS EXEC master..xp_sendmail 'MaryM', 'New title, mention in the next report to distributors.'

Triggers contain Transact-SQL statements, much the same as stored procedures. Triggers, like stored procedures, return the result set generated by any SELECT statements in the trigger. Including SELECT statements in triggers, except statements that only fill parameters, is not recommended. This is because users do not expect to see any result sets returned by an UPDATE, INSERT, or DELETE statement.

You can use the FOR clause to specify when a trigger is executed:

• AFTER

The trigger executes after the statement that triggered it completes. If the statement fails with an error, such as a constraint violation or syntax error, the trigger is not executed. AFTER triggers cannot be specified for views, they can only be specified for tables. You can specify multiple AFTER triggers for each triggering action (INSERT, UPDATE, or DELETE). If you have multiple AFTER triggers for a table, you can use **sp_settriggerorder** to define which AFTER trigger fires first and which fires last. All other AFTER triggers besides the first and last fire in an undefined order which you cannot control.

AFTER is the default in SQL Server 2000. You could not specify AFTER or INSTEAD OF in SQL Server version 7.0 or earlier, all triggers in those versions operated as AFTER triggers.

• INSTEAD OF

The trigger executes in place of the triggering action. INSTEAD OF triggers can be specified on both tables and views. You can define only one INSTEAD OF trigger for each triggering action (INSERT, UPDATE, and DELETE). INSTEAD OF triggers can be used to perform enhance integrity checks on the data values supplied in INSERT and UPDATE statements. INSTEAD OF triggers also let you specify actions that allow views, which would normally not support updates, to be updatable.

See Also

Enforcing Business Rules with Triggers

SQL Server Architecture

Collations

The physical storage of character strings in Microsoft® SQL Server[™] 2000 is controlled by collations. A collation specifies the bit patterns that represent each character and the rules by which characters are sorted and compared.

SQL Server 2000 supports objects that have different collations being stored in a single database. Separate SQL Server 2000 collations can be specified down to the level of columns. Each column in a table can be assigned different collations. Earlier versions of SQL Server support only one collation for each instance of SQL Server. All databases and database objects created in an instance of SQL Server 7.0 or earlier have the same collation.

How Character Data Is Stored

In a computer, characters are represented by different patterns of bits being either ON or OFF. There are 8 bits in a byte, and the 8 bits can be turned ON and OFF in 256 different patterns. A program that uses 1 byte to store each character can therefore represent up to 256 different characters by assigning a character to each of the bit patterns. There are 16 bits in 2 bytes, and 16 bits can be turned ON and OFF in 65,536 unique patterns. A program that uses 2 bytes to represent each character can represent up to 65,536 characters.

Single-byte code pages are definitions of the characters mapped to each of the 256 bit patterns possible in a byte. Code pages define bit patterns for uppercase and lowercase characters, digits, symbols, and special characters such as !, @, #, or %. Each European language, such as German or Spanish, has its own single-byte code page. Although the bit patterns used to represent the Latin alphabet characters A through Z are the same for all the code pages, the bit patterns used to represent accented characters such as 'é' and 'á' vary from one code page to the next. If data is exchanged between computers running different code pages, all character data must be converted from the code page of the sending computer to the code page of the receiving computer. If the source data has extended characters that are not defined in the code page of the receiving computer, data is lost. When a database serves clients from many different countries, it is difficult to pick a code page for the database that contains all the extended characters required by all the client computers. Also, there is a lot of processing time spent

doing the constant conversions from one code page to another.

Single-byte character sets are also inadequate to store all the characters used by many languages. For example, some Asian languages have thousands of characters, so must use two bytes per character. Double-byte character sets have been defined for these languages. Still, each of these languages have their own code page, and there are difficulties in transferring data from a computer running one double-byte code page to a computer running another.

Code page	Description
1258	Vietnamese
1257	Baltic
1256	Arabic
1255	Hebrew
1254	Turkish
1253	Greek
1252	Latin1 (ANSI)
1251	Cyrillic
1250	Central European
950	Chinese (Traditional)
949	Korean
936	Chinese (Simplified)
932	Japanese
874	Thai
850	Multilingual (MS-DOS Latin1)
437	MS-DOS U.S. English

SQL Server 2000 supports these code pages.

To address the character conversion and interpretation problems that occur when trying to support multiple code pages in a network, the ISO standards organization and a group called the Unicode Consortium defined the Unicode standard. Unicode uses two bytes to store each character. Because 65,536 characters are enough to cover all the commonly used characters from all the languages of the world, all major languages are covered by the Unicode standard. If all the computers and programs in a network use Unicode, there is no need for any character conversions, each user will see exactly the same

characters as all other users, and no loss of characters will occur.

On computers running Microsoft Windows® operating systems, the code page used by the operating system and Windows applications is defined by the Windows locale. The locale is selected when the operating system is installed. Windows applications interpret character data using the code page defined by the Windows locale. Windows applications also support wide character, or Unicode, data.

SQL Server 2000 supports two categories of character data types:

- The Unicode data types **nchar**, **nvarchar**, and **ntext**. These data types use the Unicode character representation. Code pages do not apply to these data types.
- The non-Unicode character data types **char**, **varchar**, and **text**. These data types use the character representation scheme defined in a single or double-byte code page.

For more information about how character data is stored and the operation of code pages, Unicode, and sort orders, see Developing International Software for Windows 95 and Windows NT 4.0 in the MSDN® page at http://msdn.microsoft.com.

International Data and Unicode

Storing data in multiple languages within one database is difficult to manage when using only character data and code pages. It is difficult to find one code page for the database that can store all the required language-specific characters. It is also difficult to ensure the proper translation of special characters when being read or updated by different clients running various code pages. Databases that support international clients should always use Unicode data types instead of non-Unicode data types.

For example, a database of customers in North America has to handle three major languages:

• Spanish names and addresses for Mexico.

- French names and addresses for Quebec.
- English names and addresses for the rest of Canada and the United States.

When you use only character columns and code pages, care has to be taken to ensure the database is installed with a code page that will handle the characters of all three languages. More care must be taken to ensure the proper translation of characters from one of the languages when read by clients running a code page for another language.

With the growth of the Internet, it is becoming more important than ever before to support many client computers running different locales. It is difficult to pick a code page for character data types that will support all of the characters required by a worldwide audience.

The easiest way to manage character data in international databases is to always use the Unicode **nchar**, **nvarchar**, and **ntext** data types in place of their non-Unicode equivalents (**char**, **varchar**, and **text**). If all the applications that work with international databases also use Unicode variables instead of non-Unicode variables, character translations do not have to be performed anywhere in the system. All clients will see exactly the same characters in data as all other clients.

For systems that could use single-byte code pages, the fact that Unicode data needs twice as much storage space as non-Unicode character data is at least partially offset by eliminating the need to convert extended characters between code pages. Systems using double-byte code pages do not have this issue.

SQL Server 2000 stores all textual system catalog data in columns having Unicode data types. The names of database objects such as tables, views, and stored procedures are stored in Unicode columns. This allows applications to be developed using only Unicode, which avoids all issues with code page conversions.

Sort Order

A sort order specifies the rules used by SQL Server to interpret, collate, compare, and present character data. For example, a sort order defines whether 'a' is less

than, equal to, or greater than 'b'. A sort order defines whether the collation is case-sensitive, for example whether 'm' is equal or not equal to 'M'. It also defines if the collation is accent-sensitive, for example whether 'á' is equal or not equal to 'ä'.

SQL Server 2000 uses two sort orders with each collation, one for Unicode data and another for the character code page.

Many SQL Server collations use the same code page, but have a different sort order for the code page. This allows sites to choose:

- Whether characters will simply be sorted based on the numeric value represented by their bit patterns. Binary sorting is fastest because SQL Server does not have to make any adjustments and can use fast, simple sorting algorithms. Binary sort orders are always case-sensitive. Because the bit patterns in a code page may not be arranged in the same sequence as defined by the dictionary rules for a specific language, binary sorting sometimes does not sort characters in a sequence users who speak that language might expect.
- Between case-sensitive or case-insensitive behavior.
- Between accent-sensitive or accent-insensitive behavior.

See Also

Collation Options for International Support SQL Server Collation Fundamentals Unicode Data Using Unicode Data

SQL Server Collation Fundamentals

Microsoft® SQL Server[™] 2000 supports several collations. A collation encodes the rules governing the proper use of characters for either a language, such as Macedonian or Polish, or an alphabet, such as Latin1_General (the Latin alphabet used by western European languages).

Each SQL Server collation specifies three properties:

- The sort order to use for Unicode data types (**nchar**, **nvarchar**, and **ntext**). A sort order defines the sequence in which characters are sorted, and the way characters are evaluated in comparison operations.
- The sort order to use for non-Unicode character data types (char, varchar, and text).
- The code page used to store non-Unicode character data.

Note You cannot specify the equivalent of a code page for the Unicode data types (**nchar**, **nvarchar**, and **ntext**). The double-byte bit patterns used for Unicode characters are defined by the Unicode standard and cannot be changed.

SQL Server 2000 collations can be specified at any level. When you install an instance of SQL Server 2000, you specify the default collation for that instance. Each time you create a database, you can specify the default collation used for the database. If you do not specify a collation, the default collation for the database is the default collation for the instance. Whenever you define a character column, variable, or parameter, you can specify the collation of the object. If you do not specify a collation, the object is created with the default collation of the database.

If all of the users of your instance of SQL Server speak the same language, you should pick the collation that supports that language. For example, if all of the users speak French, choose the French collation.

If the users of your instance of SQL Server speak multiple languages, you should pick a collation that best supports the requirements of the various languages. For example, if the users generally speak western European languages, choose the Latin1_General collation. When you support users who speak multiple languages, it is most important to use the Unicode data types, **nchar**, **nvarchar**, and **ntext**, for all character data. Unicode was designed to eliminate the code page conversion difficulties of the non-Unicode **char**, **varchar**, and **text** data types. Collation still makes a difference when you implement all columns using Unicode data types because it defines the sort order for comparisons and sorts of Unicode characters. Even when you store your character data using Unicode data types you should pick a collation that supports most of the users in case a column or variable is implemented using the non-Unicode data types.

A SQL Server collation defines how the database engine stores and operates on character and Unicode data. After data has been moved into an application, however, character sorts and comparisons done in the application are controlled by the Windows locale selected on the computer. The collation used for character data by applications is one of the items controlled by the Windows locale also defines other items, such as number, time, date, and currency formats). For Microsoft Windows NT® 4.0, Microsoft Windows® 98, and Microsoft Windows 95, the Windows locale is specified using the Regional Settings application in Control Panel. For Microsoft Windows 2000, the locale is specified using the Regional Options application in Control Panel. For more information about Windows NT 4.0 in the MSDN® page at Microsoft Web site.

Multiple collations can use the same code page for non-Unicode data. For example, the 1251 code page defines a set of Cyrillic characters. This code page is used by several collations, such as Cyrillic_General, Ukrainian, and Macedonian. Although all of these collations use the same set of bits to represent non-Unicode character data, the sorting and comparison rules they apply are slightly different to handle the dictionary definitions of the correct sequence of characters in the language or alphabet associated with the collation.

Because SQL Server 2000 collations control both the Unicode and non-Unicode sort orders, you do not encounter problems caused by specifying different sorting rules for Unicode and non-Unicode data. In earlier versions of SQL Server, the code page number, the character sort order, and the Unicode collation are specified separately. Earlier versions of SQL Server also support varying numbers of sort orders for each code pages, and for some code pages support sort orders not available in Windows locales. In SQL Server 7.0, it is also possible to specify a Unicode sort order that is different from the sort order chosen for non-Unicode data. This can cause ordering and comparison operations to return different results when working with Unicode data as opposed to non-Unicode data.

See Also

COLLATE Collation Options for International Support Collations Unicode Data Using Unicode Data

Selecting Collations

In Microsoft® SQL Server[™] 2000, you specify a single collation name that controls all three collation attributes: the Unicode sort order, the non-Unicode code page, and the non-Unicode sort order. None of the SQL Server 2000 collations allow different comparison and sorting rules for Unicode and non-Unicode character data. There are two groups of SQL Server 2000 collations: Windows collations and SQL collations.

Windows Collations

Windows collations are collations defined for SQL Server to support Microsoft Windows® locales. By specifying a Windows collation for SQL Server, the instance of SQL Server uses the same code pages and sorting and comparison rules as an application running on a computer for which you have specified the associated Windows locale. For example, the French Windows collation for SQL Server matches the collation attributes of the French locale for Windows.

There are more Windows locales than there are SQL Server Windows collations. The names of Windows locales are based on a language and territory, for example French (Canada). Several languages, however, share common alphabets and rules for sorting and comparing characters. For example, 33 Windows locales, including all of the Portuguese, and English Windows locales, use the Latin1 code page (1252) and follow a common set of rules for sorting and comparing characters. The SQL Server Windows collation based on the Latin1_General code page and sorting rules supports all 33 of these Windows locales. Also, Windows locales specify attributes not covered by SQL Server Windows collations, such as currency, date, and time formats. Because countries such as Great Britain and the United States have different currency, date, and time formats, they require different Windows collations. They do not require different SQL Server collations because they have the same alphabet and rules for sorting and comparing characters.

SQL Collations

SQL collations are a compatibility option to match the attributes of common

combinations of code page number and sort orders that have been specified in earlier versions of SQL Server. For example, for mapping a SQL Server 2000 SQL collation to what is specified in earlier versions of SQL Server, the SQL Server 2000 SQL collation SQL_Latin1_General_CP1_CI_AS matches the SQL Server version 7.0 default specification of:

- The ISO code page 1252.
- The dictionary order, case-insensitive character sort order.
- The General Unicode collation.

The SQL collations available in SQL Server 2000 do not match all combinations that can be specified in earlier versions of SQL Server. For example, no SQL Server 2000 SQL collation supports a case-sensitive sort order for non-Unicode data and case-insensitive sort order for Unicode data. The earlier SQL collations that cannot be exactly specified in SQL Server 2000 are called obsolescent SQL collations.

In SQL Server 2000, you should primarily use Windows collations. You should use SQL collations only to maintain compatibility with existing instances of earlier versions of SQL Server, or to maintain compatibility in applications developed using SQL collations in earlier versions of SQL Server.

Collation Comparison and Ordering Rules

Most of the comparison and ordering rules defined in a collation are governed by the dictionary definition of the correct sequence of characters for the alphabet or language. The attributes you can control are whether comparisons and sorts of character and Unicode data should be:

- Based on the dictionary conventions that define the correct sequence of characters in the language or alphabet associated with the collation, or based on the sequence of the binary bit patterns representing the different characters.
- Case-sensitive or case-insensitive. For example, defining whether 'a' is

equal or not equal to 'A'. If you choose case-insensitive, comparisons always ignore case, so the uppercase version of a character evaluates to being equal to the lowercase version of the character. When you choose case-insensitivity, the relative sequence in which uppercase and lowercase are sorted is undefined unless you also specify uppercase preference. Uppercase preference affects only sort operations and specifies that uppercase versions of a character come earlier in the sort sequence than lowercase versions of the same character. Uppercase preference has no affect on comparisons, so 'A' still evaluates to being equal to 'a' when uppercase preference is on. Uppercase preference can be specified only in SQL collations, not in Windows collations.

 Sensitive or insensitive to accented characters, also known as extended characters. Accented characters are those characters that have a diacritical mark, such as the German umlaut (ë) or the Spanish tilde (~). For example, accent sensitivity defines whether 'a' is equal or not equal to 'ä'.

When you choose a collation, you can specify if you want binary behavior, or dictionary sorting that is sensitive or insensitive to case and accents:

In binary collations, comparisons and sorting are based strictly on the bit pattern of the characters. This is the fastest option. Because uppercase characters are stored with different bit patterns than their corresponding lowercase characters, and accented characters have different bit patterns than characters without accents, binary sort orders are always case-sensitive and accent sensitive. Binary collations also ignore dictionary sequences that have been defined for specific languages. They simply order the characters based on the relative value of the bit patterns that represent each character. While the bit patterns defined for Latin characters, such as 'A' or 'z', are such that binary sorting yields the correct results, the bit patterns for some extended characters in some code pages may be different than the ordering sequence defined in dictionaries for the language associated with a collation. This can lead to occasional ordering and comparison results that are different than what a speaker of the language might expect.

• If you do not specify a binary collation, SQL Server uses the dictionary ordering of the collation you have chosen. Dictionary order means characters are not sorted or compared based only on their bit patterns. The collation follows the conventions of the associated language regarding the proper sequence for characters. For example, case-insensitive sort orders must use dictionary rules to determine which lowercase and uppercase bit patterns are equal.

Although the bit patterns in a code page generally yield the correct comparison and ordering results for any language that uses the code page, the conventions for some of the languages may require different results than are generated for the bit patterns of a small number of characters. For example, the Czech, Hungarian, and Polish collations use the same code page, 1250, which was designed for the Slavic languages. Each of these languages, however, use slightly different conventions for the sequence in which accented characters should be sorted.

If you do not specify binary sorting, all SQL Server operations follow the dictionary conventions for sorting and comparing characters. When the dictionary order is used, you can specify whether you want the collation to be sensitive or insensitive to both case and accented characters.

Case-sensitivity applies to SQL identifiers and passwords as well as to data. If you specify a binary or case-sensitive default sort order for an instance of SQL Server or database, all references to objects must use the same case with which they were created. For example, consider this table:

CREATE TABLE MyTable (PrimaryKey int PRIMARY KEY, CharCo

If the CREATE TABLE statement is executed on an instance of SQL Server or database that has a case-sensitive or binary sort order, all references to the table must use the same case that was specified in the CREATE TABLE statement:

-- Object not found error because case is not correct: SELECT * FROM MYTABLE

- -- Invalid column name error because case is not correct
- -- for the WHERE clause reference to the PrimaryKey column.

SELECT * FROM MyTable WHERE PRIMARYKEY = 123 -- Correct statement: SELECT CharColumn FROM MyTable WHERE PrimaryKey = 123

See Also

Collation Options for International Support Specifying Collations Unicode Data Using Unicode Data

Specifying Collations

Microsoft[®] SQL Server[™] 2000 collations can be specified at several levels, including the following:

- When you install an instance of SQL Server, you can specify the default collation for that instance during setup. The default collation for the instance also becomes the default collation of the system databases: **master**, **model**, **tempdb**, **msdb**, and **Distribution**.
- When you create a database, you can use the COLLATE clause of the CREATE DATABASE statement to specify the default collation of the database. You can also specify a collation when you create a database using SQL Server Enterprise Manager. If you do not specify a collation, the database is assigned the default collation of the **model** database. The default collation of the **model** database is the same as the default collation of the instance of SQL Server.
- When you create a table, you can specify collations for each character string column using the COLLATE clause of the CREATE TABLE statement. You can also specify a collation when you create a table using SQL Server Enterprise Manager. If you do not specify a collation, the column is assigned the default collation of the database.

You can also use the database_default option in the COLLATE clause to specify that a column in a temporary table use the collation default of the current user database for the connection instead of **tempdb**.

- When you specify a literal string, you can use the COLLATE clause to specify the collation. If you do not specify a collation, the literal is assigned the database default collation.
- In SQL-DMO you can use the **Collation** property to specify collations for instances, databases, and columns. For more information, see

Collation Property.

• Parameters for stored procedures or functions, user-defined data types, and variables are assigned the default collation of the database:

The collation of an identifier depends on the level at which it is defined. Identifiers of instance-level objects, such as logins and database names, are assigned the default collation of the instance. Identifiers of objects within a database, such as tables, views, and column names, are assigned the default collation of the database. Variables, GOTO labels, temporary stored procedures, and temporary tables can be created when the connection context is associated with one database, and then referenced when the context has been switched to another database. Because of this, the identifiers for variables, GOTO labels, and temporary tables are in the default collation of the instance.

Specifying collations for columns or literals can be done only for the **char**, **varchar**, **text**, **nchar**, **nvarchar**, and **ntext** data types.

Collations are generally identified by a collation name. There are two classes of names: Windows collation names for the new collations aligned with Windows locales, and SQL collation names for the compatibility mode collations that result when upgrading from earlier versions of SQL Server. For more information, see <u>Windows Collation Name</u>), and <u>SQL Collation Name</u>.

The exception to specifying collation names is in Setup:

- You do not specify a collation name for Windows collations, but instead specify the collation designator, and then select check boxes to specify binary sorting or dictionary sorting that is either sensitive or insensitive to either case or accents.
- You do not specify SQL collation names, but instead select a collation based on a longer, more human-readable display name.

You can execute the system function **fn_helpcollations** to retrieve a list of all the valid collation names for Windows collations and SQL collations, for example:

SELECT *

FROM ::fn_helpcollations()

You can also use the SQL-DMO **ListCollations** method to get a list of the valid collation names. For more information, see <u>ListCollations Method</u>.

The system catalog stored procedures have been enhanced to report the collation of all SQL Server objects that have a collation.

SQL Server can support only code pages that are supported by the underlying operating system. When you perform an action that depends on collations, the SQL Server collation used by the referenced object must use a code page supported by the operating system running on the computer. These actions can include:

- Specifying a default collation for an instance of SQL Server.
- Specifying a default collation for a database when you create the database.
- Restoring a database backup. Windows must support the code page of the default collation used by the database.
- Attaching a database. Windows must support the code page of the default collation used by the database.
- Specifying a collation for a column when creating a table.
- Specifying a collation when creating a user-defined data type.
- Specifying a collation when declaring a character-string constant.

If the collation specified or the collation used by the referenced object, uses a code page not supported by the Microsoft Windows® operating systems, SQL Server issues error 2775:

"Code page *codepagenumber* is not supported by the system."

Your response to this message depends on the version of the Windows operating system installed on the computer:

- Microsoft Windows 2000 supports all of the code pages used by SQL Server collations, so the error message will not occur.
- Microsoft Windows NT® 4.0 may require that you install a language pack to support some code pages. For more information about installing a Windows NT language pack, see the Windows NT Help.
- Microsoft Windows 98 supports only one code page on a computer. You must choose a SQL Server collation that uses the same code page used by Windows 98.

See Also

ALTER TABLE Collation Options for International Support Collations Constants CREATE DATABASE CREATE TABLE DECLARE @local_variable table

Using Unicode Data

Specifying the Default Collation for an Instance of SQL Server

The default collation for an instance of Microsoft® SQL Server[™] 2000 is defined during setup. If you choose the minimal or typical setup options, then Setup installs these collations:

- If you upgrade a default instance of SQL Server version 6.5 or SQL Server version 7.0 to SQL Server 2000, or if you install a default instance of SQL Server 2000 that will be version switched with a default instance of SQL Server version 6.5, SQL Server Setup carries forward the same collation used in the existing instance of SQL Server version 6.5 or SQL Server version 7.0, including obsolescent collations.
- In all other cases, Setup chooses the Windows collation that supports the Windows locale of the computer on which the instance of SQL Server 2000 is being installed.

Note The Setup program does not set the instance default collation to the Windows collation Latin1_General_CI_AS if the computer is using the U.S. English locale. Instead, it sets the instance default collation to the SQL collation SQL_Latin1_General_Cp1_CI_AS. This may change in a future release.

If you choose the **Custom** setup option, Setup uses the same logic as in the minimal and typical options to set the collation that is selected when the **Character Set / Sort Order / Windows Collation** window is displayed. You should not use the selected collation in these cases:

- If the instance will be included in a replication scheme, all instances of SQL Server involved in the replication scheme (Publishers, Subscribers, and Distributors) should use the same code page. You should make sure the collation selected by Setup uses the same code page as the other instances of SQL Server in the replication scheme.
- If the primary language that the instance must support is different than

the Windows locale of the computer on which the instance is being installed.

For a table showing which collation designator to specify for a Windows locale, see <u>Windows Collation Names Table</u>.

During setup, the **master**, **model**, **tempdb**, **msdb**, and **Distribution** system databases are assigned the same default collation as the default collation chosen for the instance.

See Also

Collation Options for International Support

Mixed Collation Environments

Compatibility issues can have an impact on organizations that use multiple collations to store their data. Most organizations use the same collation for all of their Microsoft® SQL Server[™] 2000 databases, thereby eliminating all collation compatibility issues. Other organizations, however, must store data viewed by users who speak various languages and want to do so with a minimum of collation compatibility issues.

All character and Unicode objects (such as columns, variables, and constants) have a collation. Whenever you work with objects that have different collations and code pages, you must code your queries to comply with the rules of collation coercion. When you code a complex expression that uses operators to combine multiple simple expressions that have different collations, all of the collations must be implicitly convertible, or explicitly converted using the COLLATE clause. For more information about collation coercion, see <u>Collation Precedence</u>.

If you do not specify a collation in a character or Unicode expression, the default collation may vary depending on the current database setting for the connection. For example, if you do not specify a COLLATE clause on a character or Unicode constant, the constant is assigned the default collation of the current database. This means that the result of a Transact-SQL statement may have different collations when executed in the context of different databases.

If you are setting up replication, all of the databases involved in a replication network, including Publishers, Subscribers, and Distributors, must have the same code page.

The bulk copy functions, BULK INSERT, and the **bcp** command prompt utility support column collations. For more information, see <u>Copying Data Between</u> <u>Different Collations</u>.

Minimizing Collation Issues

If you must store character data that reflects multiple languages, you can minimize collation compatibility issues by always using the Unicode **nchar**, **nvarchar**, and **ntext** data types instead of the **char**, **varchar**, **text** data types.

Using the Unicode data types eliminates code page conversion issues.

Another recommendation that minimizes collation compatibility issues is to standardize your site as either sensitive or insensitive to case and accented characters. If you always choose collations with the same case and accent sensitivity, end users experience consistent behavior across all systems. Most SQL Server 2000 sites choose to be case-insensitive and accent-sensitive. Case sensitivity also applies to the names of SQL Server objects; therefore, if you specify case-sensitive collations, all users must specify the correct case when querying the database. For example, if you have a case-sensitive server and create a table named **Employees**, all queries must refer to the table as **Employees**. References that do not use the correct case, such as **EMPLOYEES** or **employees**, are invalid.

Collations and tempdb

The **tempdb** database is built each time SQL Server is started, and has the same default collation as the **model** database, which is typically the same as the default collation of the instance. If you create a user database and specify a different default collation than **model**, the user database has a different default collation than **model**, the user database has a different default collation than **tempdb**. All temporary stored procedures or temporary tables are created and stored in **tempdb**, which means that all implicit columns in temporary tables and all coercible-default constants, variables, and parameters in temporary stored procedures have different collations than comparable objects created in permanent tables and stored procedures.

This can lead to problems with the **text** data type, which does not support code page conversions. For example, an instance of SQL Server 2000 defaults to the Latin1_General_CS_AS collation, and you execute these statements:

CREATE DATABASE TestDB COLLATE Estonian_CS_AS USE TestDB CREATE TABLE TestPermTab (PrimaryKey int PRIMARY KEY, Tex

In this system, the **tempdb** database uses the Latin1_General_CS_AS collation with code page 1252, and **TestDB** and **TestPermTab.TextCol** use the Estonian_CS_AS collation with code page 1257. If you then execute:

USE TestDB

GO

-- Create a temporary table with the same column declarations

-- as TestPermTab

CREATE TABLE #TestTempTab (PrimaryKey int PRIMARY KEY, Te

-- This statement gets an code page conversion not allowed error

-- because the temporary table is created in tempdb, which has a -- different default collation than TestDB.

INSERT INTO #TestTempTab

```
SELECT * FROM TestPermTab
```

GO

To eliminate the error you can use one of these alternatives:

- Use the Unicode data type **ntext** instead of **text** for the two **TextCol** columns.
- Specify that the temporary table column use the default collation of the user database, not **tempdb**. This allows the temporary table to work with similarly formatted tables in multiple databases, if that is a requirement of your system.

CREATE TABLE #TestTempTab (PrimaryKey int PRIMARY KEY, TextCol text COLLATE database_default)

 Specify the correct collation for the #TestTempTab column: CREATE TABLE #TestTempTab (PrimaryKey int PRIMARY KEY, TextCol text COLLATE Estonian_CS_AS)

Collations in BACKUP and RESTORE

If you restore a database, RESTORE uses the collation of the source database that was recorded in the backup file. The restored database has the same

collation as the original database that was backed up. Individual objects within the database that have different collations also retain their original collation. The database can be restored even if the instance on which you run restore has a different default collation than the instance on which BACKUP was run.

If there is already a database with the same name on the target server, the only way to restore from the backup is to specify REPLACE on the RESTORE statement. If you specify REPLACE, the existing database is completely replaced with the contents of the database on the backup file, and the restored version of the database will have the same collation recorded in the backup file.

If you are restoring log backups, the destination database must have the same collation as the source database.

Collations and text column

If you create a table with a **text** column that has a different code page than the code page of the database's default collation, there are only two ways you can specify data values to be inserted into the column, or update existing values. You can:

- Specify a Unicode constant.
- Select a value from another column with the same code page.

Assume the following database and table:

 -- Create a database with a default of code page 1252.
 CREATE DATABASE TestDB COLLATE Latin1_General_CS_AS
 -- Create a table with a different code page, 1253.
 CREATE TABLE TestTab

 (PrimaryKey int PRIMARY KEY, TextCol text COLLATE Greek_CS_AS
)

-- This INSERT statement successfully inserts a Unicode string. INSERT INTO TestTab VALUES (1, N'abc')

-- This INSERT statement successfully inserts data by selecting

-- from a similarly formatted table in another database that uses

-- uses the Greek 1253 code page as its default.

INSERT INTO TestTab

SELECT * FROM GreekDatabase.dbo.TestTab

SQL Server Architecture

Changing Collations

You can change the collation of a column by using the ALTER TABLE statement:

```
CREATE TABLE MyTable
(PrimaryKey int PRIMARY KEY,
CharCol varchar(10) COLLATE French_CI_AS NOT NULL
)
GO
ALTER TABLE MyTable ALTER COLUMN CharCol
varchar(10)COLLATE Latin1_General_CI_AS NOT NULL
GO
```

You cannot alter the collation of a column that is currently referenced by:

- A computed column.
- An index.
- Distribution statistics, either generated automatically or by the CREATE STATISTICS statement.
- A CHECK constraint.
- A FOREIGN KEY constraint.

You can also use the COLLATE clause on an ALTER DATABASE to change the default collation of the database:

ALTER DATABASE MyDatabase COLLATE French_CI_AS

Altering the default collation of a database does not change the collations of the

columns in any existing user-defined tables. These can be changed with ALTER TABLE. The COLLATE CLAUSE on an ALTER DATABASE statement changes:

- The default collation for the database. This new default collation is applied to all columns, user-defined data types, variables, and parameters subsequently created in the database. It is also used when resolving the object identifiers specified in SQL statements against the objects defined in the database.
- Any **char**, **varchar**, **text**, **nchar**, **nvarchar**, or **ntext** columns in system tables to the new collation.
- All existing **char**, **varchar**, **text**, **nchar**, **nvarchar**, or **ntext** parameters and scalar return values for stored procedures and user-defined functions to the new collation.
- The **char**, **varchar**, **text**, **nchar**, **nvarchar**, or **ntext** system data types, and all user-defined data types based on these system data types, to the new default collation.

After a collation has been assigned to any object other than a column or database, you cannot change the collation except by dropping and re-creating the object. This can be a complex operation. To change the default collation for an instance of Microsoft® SQL ServerTM 2000 you must:

- Make sure you have all of the information or scripts needed to re-create your user databases and all of the objects in them.
- Export all of your data using a tool such as bulk copy.
- Drop all of the user databases.
- Rebuild the **master** database specifying the new collation.

- Create all of the databases and all of the objects in them.
- Import all of your data.

Note Instead of changing the default collation of an instance of SQL Server 2000, you can specify a default collation for each new database you create.

SQL Server Architecture

SQL Indexes

A Microsoft® SQL Server[™] 2000 index is a structure associated with a table or view that speeds retrieval of rows from the table or view. An index contains keys built from one or more columns in the table or view. These keys are stored in a structure that allows SQL Server to find the row or rows associated with the key values quickly and efficiently.

SQL Server Architecture

Table Indexes

Microsoft® SQL Server[™] 2000 supports indexes defined on any column in a table, including computed columns.

If a table is created with no indexes, the data rows are not stored in any particular order. This structure is called a heap.

The two types of SQL Server indexes are:

• Clustered

Clustered indexes sort and store the data rows in the table based on their key values. Because the data rows are stored in sorted order on the clustered index key, clustered indexes are efficient for finding rows. There can only be one clustered index per table, because the data rows themselves can only be sorted in one order. The data rows themselves form the lowest level of the clustered index.

The only time the data rows in a table are stored in sorted order is when the table contains a clustered index. If a table has no clustered index, its data rows are stored in a heap.

• Nonclustered

Nonclustered indexes have a structure completely separate from the data rows. The lowest rows of a nonclustered index contain the nonclustered index key values and each key value entry has pointers to the data rows containing the key value. The data rows are not stored in order based on the nonclustered key.

The pointer from an index row in a nonclustered index to a data row is called a row locator. The structure of the row locator depends on whether the data pages are stored in a heap or are clustered. For a heap, a row locator is a pointer to the row. For a table with a clustered index, the row locator is the clustered index key.

The only time the rows in a table are stored in any specific sequence is when a clustered index is created on the table. The rows are then stored in sequence on

the clustered index key. If a table only has nonclustered indexes, its data rows are stored in a unordered heap.

Indexes can be unique, which means no two rows can have the same value for the index key. Otherwise, the index is not unique and multiple rows can share the same key value.

There are two ways to define indexes in SQL Server. The CREATE INDEX statement creates and names an index. The CREATE TABLE statement supports the following constraints that create indexes:

- PRIMARY KEY creates a unique index to enforce the primary key.
- UNIQUE creates a unique index.
- CLUSTERED creates a clustered index.
- NONCLUSTERED creates a nonclustered index.

When you create an index on SQL Server 2000, you can specify whether the keys are stored in ascending or descending order.

SQL Server 2000 supports indexes defined on computed columns, as long as the expression defined for the column meets certain restrictions, such as only referencing columns from the table containing the computed column, and being deterministic.

A fill factor is a property of a SQL Server index that controls how densely the index is packed when created. The default fill factor usually delivers good performance, but in some cases it may be beneficial to change the fill factor. If the table is going to have many updates and inserts, create an index with a low fill factor to leave more room for future keys. If the table is a read-only table that will not change, create the index with a high fill factor to reduce the physical size of the index, which lowers the number of disk reads SQL Server uses to navigate through the index. Fill factors are only applied when the index is created. As keys are inserted and deleted, the index will eventually stabilize at a certain density.

Indexes not only speed up the retrieval of rows for selects, they also usually increase the speed of updates and deletes. This is because SQL Server must first find a row before it can update or delete the row. The increased efficiency of using the index to locate the row usually offsets the extra overhead needed to update the indexes, unless the table has a lot of indexes.

This example shows the Transact-SQL syntax for creating indexes on a table.

```
USE pubs
GO
CREATE TABLE emp_sample
(emp_id int PRIMARY KEY CLUSTERED,
emp_name char(50),
emp_address char(50),
emp_title char(25) UNIQUE NONCLUSTERED )
GO
CREATE NONCLUSTERED INDEX sample_nonclust ON emp_samp
GO
```

Deciding which particular set of indexes will optimize performance depends on the mix of queries in the system. Consider the clustered index on **emp_sample.emp_id**. This works well if most queries referencing **emp_sample** have equality or range comparisons on **emp_id** in their WHERE clauses. If the WHERE clauses of most queries reference **emp_name** instead of **emp_id**, performance could be improved by instead making the index on **emp_name** the clustered index.

Many applications have a complex mix of queries that is difficult to estimate by interviewing users and programmers. SQL Server 2000 provides an Index Tuning Wizard to help design indexes in a database. The easiest way to design indexes for large schemas with complex access patterns is to use the Index Tuning Wizard.

You provide the Index Tuning Wizard with a set of SQL statements. This could be a script of statements you build to reflect a typical mix of statements in the system, but it is usually a SQL Profiler trace of the actual SQL statements processed on the system during a period of time that reflects the typical load on the system. The Index Tuning Wizard analyzes the workload and the database, and then recommends an index configuration that will improve the performance of the workload. You can choose to either replace the existing index configuration, or to keep the existing index configuration and implement new indexes to improve the performance of a slow-running subset of the queries.

See Also

Indexes

Parallel Operations Creating Indexes

SQL Server Architecture

View Indexes

Microsoft® SQL Server[™] 2000 supports defining indexes on views. Views are sometimes called virtual tables because the result set returned by the view has the same general form as a table with columns and rows, and views can be referenced the same way as tables in SQL statements. The result set of a non-indexed view is not stored permanently in the database. Each time a query references the view, SQL Server dynamically merges the logic needed to build the view result set into the logic needed to build the complete query result set from the data in the base tables. The process of building the view results is called materializing the view. For more information, see <u>View Resolution</u>.

For a nonindexed view, the overhead of dynamically building the result set for each query that references a view can be substantial for views that involve complex processing of large numbers of rows. Examples include views that aggregate large amounts of data, or join many rows. If such views are frequently referenced in queries, you can improve performance by creating a unique clustered index on the view. When a unique clustered index is created on a view, the view is executed and the result set is stored in the database in the same way a table with a clustered index is stored. For more information about the structure used to store clustered indexes, see <u>Clustered Indexes</u>.

Another benefit of creating an index on a view is that the optimizer starts using the view index in queries that do not directly name the view in the FROM clause. Existing queries can benefit from the improved efficiency of retrieving data from the indexed view without having to be recoded.

Creating a clustered index on a view stores the result set built at the time the index is created. An indexed view also automatically reflects modifications made to the data in the base tables after the index is created, the same way an index created on a base table does. As modifications are made to the data in the base tables, the data modifications are also reflected in the data stored in the indexed view. The requirement that the view's clustered index be unique improves the efficiency with which SQL Server can find the rows in the index that are affected by any data modification.

You must have set specific SET options before you can create an index on a

view. The query optimizer will not consider the index for any subsequent SQL statements unless the connection executing the statement has the same option settings. For more information, see <u>SET Options That Affect Results</u>.

Indexed views can be more complex to maintain than indexes on base tables. You should create indexes only on views where the improved speed in retrieving results outweighs the increased overhead of making modifications. This usually occurs for views mapped over relatively static data, that process many rows, and are referenced by many queries.

The first index created on a view must be a unique clustered index. After the unique clustered index has been created, you can create additional nonclustered indexes. The naming conventions for indexes on views are the same as for indexes on tables. The only difference is that the table name is replaced with a view name.

All indexes on a view are dropped if the view is dropped. All nonclustered indexes on the view are dropped if the clustered index is dropped. Nonclustered indexes can be dropped individually. Dropping the clustered index on the view removes the stored result set, and the optimizer returns to processing the view like a standard view.

Although only the columns that make up the clustered index key are specified in the CREATE UNIQUE CLUSTERED INDEX statement, the complete result set of the view is stored in the database. As in a clustered index on a base table, the b-tree structure of the clustered index contains only the key columns, but the data rows contain all of the columns in the view result set.

See Also

CREATE INDEX
Creating an Indexed View

Using Indexes on Views

Maximum Size of Index Keys

Microsoft® SQL Server[™] 2000 retains the 900-byte limit for the maximum size of an index key but changes the algorithm used by CREATE INDEX to check if the specified index key exceeds the maximum allowable key size of 900 bytes. The new CREATE INDEX algorithm is similar to the row size algorithm used for CREATE TABLE.

Microsoft SQL Server version 7.0 and earlier always used the maximum size of variable columns when checking whether the key specified in a CREATE INDEX statement exceeded 900 bytes, for example:

CREATE TABLE TestTable (PrimaryKey int PRIMARY KEY, VarCharCol1 varchar(500), VarCharCol2 varchar(500))

-- This statement fails because the maximum sizes

-- of the two columns exceeds 900 bytes:

CREATE INDEX TestIdx ON TestTable(VarCharCol1, VarCharCol2)

In SQL Server 2000, the preceding CREATE INDEX statement succeeds with a warning message, unless one or more rows of data will generate a key whose value exceeds 900 bytes.

The SQL Server 2000 CREATE INDEX statement uses these algorithms:

- If the size of all fixed columns plus the maximum size of all variable columns specified in the CREATE INDEX statement is less than 900 bytes, the CREATE INDEX statement completes successfully with no warnings or errors.
- If the size of all fixed columns plus the maximum size of all variable columns exceeds 900, but the size of all fixed columns plus the minimums of the variable columns is less than 900, the CREATE

INDEX statement succeeds with a warning that a subsequent INSERT or UPDATE statement may fail if it specifies values that generates a key value larger than 900 bytes. The CREATE INDEX statement fails if existing data rows in the table have values that generate a key larger than 900 bytes. A subsequent INSERT or UPDATE statement that specifies data values that generates a key value longer than 900 bytes fails.

• The CREATE INDEX statement fails if the size of all fixed columns plus the minimum size of all variable columns specified in the CREATE INDEX statement exceeds 900 bytes.

This table shows the results of creating indexes where the keys contain only fixed or only variable-length columns.

Index Columns					
Size of the fixed-data column(s)	length	MAX of the SUM of the index key column lengths*	Index created	Message	INSERT or UPDATE run- time error due to oversized index key value
> 900 bytes	None	Not relevant	No	Error	No index present to generate error.
< = 900 bytes	None	Not relevant	Yes	None	No
None	< = 900 bytes	Not relevant	Yes	None	No
None	> 900 bytes	> 900 bytes	No	Error	No index present to generate error.
None	> 900 bytes	< = 900 bytes	Yes	Warning	Only if the sum of current lengths of all index columns is greater than 900 bytes.

* None of the rows in the table at time the CREATE INDEX statement is executed can have index key values whose total lengths exceed 900 bytes.

This table shows the results of creating indexes where the keys contain a mixture of fixed and variable-length columns.

Index Colum	ns				
Minimum size of variable- length column(s) + Size of the fixed-data column(s)	Maximum size of variable- length column(s) + Size of the fixed-data column(s)	MAX of the SUM of the index key column lengths *	Index created	Message	INSERT or UPDATE run- time error due to oversized index key value
> 900 bytes	Not relevant	Not relevant	No	Error	No index present to generate error.
< = 900 bytes	< = 900 bytes	Not relevant	Yes	None	No.
< = 900 bytes	> 900 bytes	< = 900 bytes	Yes	Warning	Only if the sum of current lengths of all index columns is greater than 900 bytes.
<= 900 bytes	> 900 bytes	> 900 bytes	No	Error	No index present to generate error.

* None of the rows in the table at time the CREATE INDEX statement is executed can have index key values whose total lengths exceed 900 bytes.

See Also

CREATE INDEX

Property Management

Microsoft® SQL Server[™] 2000 introduces extended properties that users can define on various objects in a database. These extended properties can be used to store application-specific or site-specific information about the database objects. Because the property is stored in the database, all applications reading the property can evaluate the object in the same way. This helps enforce consistency in how data is treated by all of the programs in the system.

Each extended property has a user-defined name and value. The value of an extended property is a **sql_variant** that can contain up to 7500 bytes of data. Individual database objects can have multiple extended properties.

Extended properties are managed using three system stored procedures: **sp_addextendedproperty**, **sp_updateextendedproperty**, and **sp_dropextendedproperty**. You can read the value of an existing extended property using the system function **FN_LISTEXTENDEDPROPERTY**.

There is no convention or standard for defining extended properties. The database designer sets the rules specifying the property names and contents when the database is designed, and then the applications accessing the database have to be coded to follow those rules or conventions.

See Also

Using Extended Properties on Database Objects

fn_listextendedproperty

sp_addextendedproperty

sp_dropextendedproperty

sp_updateextendedproperty

Full-Text Catalogs and Indexes

A Microsoft® SQL ServerTM 2000 full-text index provides efficient support for sophisticated word searches in character string data. The full-text index stores information about significant words and their location within a given column. This information is used to quickly complete full-text queries that search for rows with particular words or combinations of words.

Full-text indexes are contained in full-text catalogs. Each database can contain one or more full-text catalogs. A catalog cannot belong to multiple databases and each catalog can contain full-text indexes for one or more tables. A table can only have one full-text index, so each table with a full-text index belongs to only one full-text catalog.

Full-text catalogs and indexes are not stored in the database to which they belong. The catalogs and indexes are managed separately by the Microsoft Search service.

A full-text index must be defined on a base table; it cannot be defined on a view, system table, or temporary table. A full-text index definition includes:

- A column that uniquely identifies each row in the table (primary or candidate key) and does not allow NULLs.
- One or more character string columns covered by the index.

The full-text index is populated with the key values. The entry for each key has information about the significant words (noise-words or stop-words are stripped out) that are associated with the key, the column they are in, and their location in the column.

Formatted text strings, such as Microsoft® Word[™] document files or HTML files, cannot be stored in character string or Unicode columns because many of the bytes in these files contain data structures that do not form valid characters. Database applications may still have a need to access this data and apply full-text searches to it. Many sites store this type of data in **image** columns, because **image** columns do not require that each byte form a valid character. SQL Server 2000 introduces the ability to perform full-text searches against these types of

data stored in **image** columns. SQL Server 2000 supplies filters that allow it to extract the textual data from Microsoft Office[™] files (.doc, .xls, and .ppt files), text files (.txt files), and HTML files (.htm files). When you design the table, in addition to the **image** column that holds the data, you include a binding column to hold the file extension for the format of data stored in the **image** column. You can create a full-text index that references both the **image** column and the binding column to enable full-text searches on the textual information stored in the **image** column. The SQL Server 2000 full-text search engine uses the file extension information from the binding column to select the proper filter to extract the textual data from the column.

Full-text indexing is the component that implements two Transact-SQL predicates for testing rows against a full-text search condition:

- CONTAINS
- FREETEXT

Transact-SQL also has two functions that return a set of rows that match a full-text search condition:

- CONTAINSTABLE
- FREETEXTTABLE

Internally, SQL Server sends the search condition to the Microsoft Search service. The Microsoft Search service finds all the keys that match the full-text search condition and returns them to SQL Server. SQL Server then uses the list of keys to determine which table rows are to be processed.

See Also

Full-text IndexesFull-Text Query ArchitectureFull-text Querying SQL Server DataMicrosoft Search Service

Logins, Users, Roles, and Groups

Logins, users, roles, and groups are the foundation for the security mechanisms of Microsoft® SQL Server[™] 2000. Users that connect to SQL Server must identify themselves using a specific login identifier (ID). Users can then only see the tables and views they are authorized to see, and can only execute the stored procedures and administrative functions they are authorized to execute. This system of security is based on the IDs used to identify users.

See Also

Managing Security

Logins

Login identifiers (Ids) are associated with users when they connect to Microsoft® SQL ServerTM 2000. Login IDs are the accounts that control access to the SQL Server system. A user cannot connect to SQL Server without first specifying a valid login ID. Members of the **sysadmin** fixed server role define login IDs.

sp_grantlogin authorizes a Microsoft Windows® network account (either a group or a user account) to be used as a SQL Server login for connecting to SQL Server using Windows Authentication. **sp_addlogin** defines a login account for SQL Server connections using SQL Server Authentication.

See Also

<u>Logins</u> <u>sp_addlogin</u> <u>sp_grantlogin</u>

Users

A user identifier (ID) identifies a user within a database. All permissions and ownership of objects in the database are controlled by the user account. User accounts are specific to a database; the **xyz** user account in the **sales** database is different from the **xyz** user account in the **inventory** database, even though both accounts have the same ID. User IDs are defined by members of the **db_owner** fixed database role.

A login ID by itself does not give a user permissions to access objects in any databases. A login ID must be associated with a user ID in each database before anyone connecting with that login ID can access objects in the databases. If a login ID has not been explicitly associated with any user ID in a database, it is associated with the **guest** user ID. If a database has no **guest** user account, a login cannot access the database unless it has been associated with a valid user account.

When a user ID is defined, it is associated with a login ID. For example, a member of the **db_owner** role can associate the Microsoft® Windows® 2000 login **NETDOMAIN\Joe** with user ID **abc** in the **sales** database and user ID **def** in the **employee** database. The default is for the login ID and user ID to be the same.

This example shows giving a Windows 2000 account access to a database and associating the login with a user in the database:

```
USE master
GO
sp_grantlogin 'NETDOMAIN\Sue'
GO
sp_defaultdb @loginame = 'NETDOMAIN\Sue', defdb = 'sales'
GO
USE sales
GO
sp_grantdbaccess 'NETDOMAIN\Sue', 'Sue'
```

GO

In the **sp_grantlogin** statement, the Windows 2000 user **NETDOMAIN\Sue** is given access to Microsoft SQL Server[™] 2000. The **sp_defaultdb** statement makes the **sales** database her default database. The **sp_grantdbaccess** statement gives the login **NETDOMAIN\Sue** access to the **sales** database and sets her user ID within **sales** to **Sue**.

This example shows defining a SQL Server login, assigning a default database, and associating the login with a user in the database:

```
USE master
GO
sp_addlogin @loginame = 'TempWorker', @password = 'fff', defdb = 's
GO
USE sales
GO
sp_grantdbaccess 'TempWorker'
GO
```

The **sp_addlogin** statement defines a SQL Server login that will be used by various temporary workers. The statement also specifies the **sales** database as the default database for the login. The **sp_grantdbaccess** statement grants the **TempWorker** login access to the **sales** database; because no username is specified, it defaults to **TempWorker**.

A user in a database is identified by their user ID, not their login ID. For example, **sa** is a login account mapped to the special user account **dbo** (database owner) in every database. All the security-related Transact-SQL statements use the user ID as the *security_name* parameter. The administration and understanding of permissions is less confusing if the members of the **sysadmin** fixed server role and the **db_owner** fixed database role set up the system such that the login ID and user ID of each user are the same, but it is not a requirement.

The **guest** account is a special user account in SQL Server databases. If a user enters a USE *database* statement to access a database in which they are not associated with a user account, they are instead associated with the **guest** user.

See Also

<u>guest User</u>

<u>sp_addlogin</u>

<u>sp_defaultdb</u>

<u>sp_grantdbaccess</u>

<u>sp_grantlogin</u>

Roles

Roles are a powerful tool that allow you to collect users into a single unit against which you can apply permissions. Permissions granted to, denied to, or revoked from a role also apply to any members of the role. You can establish a role that represents a job performed by a class of workers in your organization and grant the appropriate permissions to that role. As workers rotate into the job, you simply add them as a member of the role; as they rotate out of the job, remove them from the role. You do not have to repeatedly grant, deny, and revoke permissions to or from each person as they accept or leave the job. The permissions are applied automatically when the users become members of the role.

Microsoft® Windows NT® and Windows® 2000 groups can be used in much the same way as roles. For more information, see <u>Groups</u>.

It is easy to manage the permissions in a database if you define a set of roles based on job functions and assign each role the permissions that apply to that job. You can then simply move users between roles rather than having to manage the permissions for each individual user. If the function of a job changes, it is easier to simply change the permissions once for the role and have the changes applied automatically to all members of the role.

In Microsoft® SQL Server[™] 2000 and SQL Server version 7.0, users can belong to multiple roles.

The following script shows adding a few logins, users, and roles, and granting permissions to the roles.

```
USE master
GO
sp_grantlogin 'NETDOMAIN\John'
GO
sp_defaultdb 'NETDOMAIN\John', 'courses'
GO
sp_grantlogin 'NETDOMAIN\Sarah'
```

GO

sp_defaultdb 'NETDOMAIN\Sarah', 'courses' GO

sp_grantlogin 'NETDOMAIN\Betty'

GO

sp_defaultdb 'NETDOMAIN\Betty', 'courses' GO

sp_grantlogin 'NETDOMAIN\Ralph'

GO

sp_defaultdb 'NETDOMAIN\Ralph', 'courses' GO

sp_grantlogin 'NETDOMAIN\Diane'

GO

sp_defaultdb 'NETDOMAIN\Diane', 'courses'

USE courses

GO

sp_grantdbaccess 'NETDOMAIN\John' GO

sp_grantdbaccess 'NETDOMAIN\Sarah'

sp_grantdbaccess 'NETDOMAIN\Betty' GO

sp_grantdbaccess 'NETDOMAIN\Ralph' GO

sp_grantdbaccess 'NETDOMAIN\Diane'

GO

sp_addrole 'Professor'

GO

sp_addrole 'Student'

GO

sp_addrolemember 'Professor', 'NETDOMAIN\John' GO

sp_addrolemember 'Professor', 'NETDOMAIN\Sarah' GO

sp_addrolemember 'Professor', 'NETDOMAIN\Diane' GO

sp_addrolemember 'Student', 'NETDOMAIN\Betty' GO

sp_addrolemember 'Student', 'NETDOMAIN\Ralph' GO

sp_addrolemember 'Student', 'NETDOMAIN\Diane' GO

GRANT SELECT ON StudentGradeView TO Student GO

GRANT SELECT, UPDATE ON ProfessorGradeView TO Professor GO

This script gives the professors John and Sarah permission to update students' grades, while the students Betty and Ralph can only select their grades. Diane has been added to both roles because she is teaching one class while taking another. The view **ProfessorGradeView** should restrict professors to the rows for students in their classes, while **StudentGradeView** should restrict students to selecting only their own grades.

There are several fixed roles defined in SQL Server 2000 and SQL Server version 7.0 during setup. Users can be added to these roles to pick up the associated administration permissions. These are server-wide roles.

Fixed server role	Description
sysadmin	Can perform any activity in SQL Server.
serveradmin	Can set serverwide configuration options, shut down the server.
setupadmin	Can manage linked servers and startup procedures.
securityadmin	Can manage logins and CREATE DATABASE permissions, also read error logs and change passwords.

processadmin Can manage processes running in SQL Serve	
dbcreator Can create, alter, and drop databases.	
diskadmin	Can manage disk files.
bulkadmin	Can execute BULK INSERT statements.

You can get a list of the fixed server roles from **sp_helpsrvrole**, and get the specific permissions for each role from **sp_srvrolepermission**.

Each database has a set of fixed database roles. While roles with the same names exist in each database, the scope of an individual role is only within a specific database. For example, if **Database1** and **Database2** both have user IDs named **UserX**, adding **UserX** in **Database1** to the **db_owner** fixed database role for **Database1** has no effect on whether **UserX** in **Database2** is a member of the **db_owner** role for **Database2**.

Fixed database role	Description	
db_owner	Has all permissions in the database.	
db_accessadmin	Can add or remove user IDs.	
db_securityadmin	Can manage all permissions, object ownerships, roles and role memberships.	
db_ddladmin	Can issue ALL DDL, but cannot issue GRANT, REVOKE, or DENY statements.	
db_backupoperator	Can issue DBCC, CHECKPOINT, and BACKUP statements.	
db_datareader	Can select all data from any user table in the database.	
db_datawriter	Can modify any data in any user table in the database.	
db_denydatareader	Cannot select any data from any user table in the database.	
db_denydatawriter	Cannot modify any data in any user table in the database.	

You can get a list of the fixed database roles from **sp_helpdbfixedrole**, and get the specific permissions for each role from **sp_dbfixedrolepermission**.

Every user in a database belongs to the **public** database role. If you want everyone in a database to be able to have a specific permission, assign the permission to the **public** role. If a user has not been specifically granted permissions on an object, they use the permissions assigned to **public**.

See Also

Adding a Member to a Predefined Role sp_dbfixedrolepermission sp_helpdbfixedrole sp_helpsrvrole sp_srvrolepermission

Groups

There are no groups in Microsoft® SQL Server[™] 2000 or SQL Server version 7.0. You can, however, manage SQL Server security at the level of an entire Microsoft Windows NT® or Microsoft Windows® 2000 group.

If you use **sp_grantlogin** and specify the name of a Windows NT or Windows 2000 group, all members of the group can then connect to SQL Server using Windows Authentication.

After the group has been authorized to connect, you can use **sp_grantdbaccess** to associate the group members with a user identifier (ID) in each database they need to access. You can use two methods:

• Associate the group with a user ID in the database.

In this case, all members of the group will be associated with that user ID when they reference the database.

• Associate an individual user account in the Windows NT or Windows 2000 group with a user ID in the database.

This individual will be associated with the user ID when they reference the database. None of the other individuals in the group will be associated with the user ID. They will be assigned the user ID associated with the group login.

Consider a Windows NT or Windows 2000 group **NETDOMAIN\Managers** with three members: **NETDOMAIN\Sue**, **NETDOMAIN\Fred**, and **NETDOMAIN\Mary**. The following Transact-SQL statements add the Windows NT or Windows 2000 group as both a login and a user in the **sales** database, and then associate **NETDOMAIN\Sue** with a specific user ID:

USE master

GO

- -- Authorize all members of NETDOMAIN\Managers to connect
- -- using Windows Authentication.
- sp_grantlogin 'NETDOMAIN\Managers'

GO

-- Make sales the default database for all members. sp_dbdefault 'NETDOMAIN\Managers', 'sales' USE sales

GO

-- Grant all members of the group access to sales

-- No user ID is specified, so SQL Server creates

-- one named 'NETDOMAIN\Managers'

sp_grantdbaccess 'NETDOMAIN\Managers' GO

-- Grant a specific member of the group access to

-- sales with a specific user.

sp_grantdbaccess 'NETDOMAIN\Sue', 'Sue'

Permissions can now be granted to either user **NETDOMAIN\Managers** or user **Sue:**

USE sales GO GRANT SELECT ON SalesTable TO NETDOMAIN\Managers GO GRANT UPDATE ON SalesTable to NETDOMAIN\Sue

The permissions applied to **NETDOMAIN****Sue** are the union of the permissions granted, revoked, or denied to both the **NETDOMAIN****Managers** or **Sue** users. Any DENY permission overrides any corresponding GRANT permissions.

Unless their Windows NT or Windows 2000 account has been associated with a specific user, members of a group are subject to the permissions assigned to the user associated with the group. If a member of the group creates an object, however, the owner name of the object is their Windows NT or Windows 2000 account name, not the group name. Consider the **NETDOMAIN\Manager** account. If **NETDOMAIN\Fred** connects to the **sales** database, he can see all tables for which **NETDOMAIN\Managers** has been granted SELECT permission. If **NETDOMAIN\Fred** executes the following statement, the table is created as **sales.NETDOMAIN\Fred.TableX**, not

sales.NETDOMAIN\Managers.TableX:

CREATE TableX (cola INT PRIMARY KEY, colb CHARACTER(200

See Also

sp_grantdbaccess

<u>sp_grantlogin</u>

Owners and Permissions

Every object in Microsoft® SQL Server[™] 2000 is owned by a user. The owner is identified by a database user identifier (ID). When an object is first created, the only user ID that can access the object is the user ID of the owner or creator. For any other user to access the object, the owner must grant permissions to that user. If the owner wants only specific users to access the object, the object, the owner can grant permissions to those specific users.

For tables and views, the owner can grant INSERT, UPDATE, DELETE, SELECT, and REFERENCES permissions, or ALL permissions. A user must have INSERT, UPDATE, DELETE, or SELECT permissions on a table before they can specify it in INSERT, UPDATE, DELETE, or SELECT statements. The REFERENCES permission lets the owner of another table use columns in your table as the target of a REFERENCES FOREIGN KEY constraint from their table. The following example illustrates granting SELECT permissions to a group named **Teachers** and REFERENCES permissions to another development user:

GRANT SELECT ON MyTable TO Teachers GRANT REFERENCES (PrimaryKeyCol) ON MyTable to DevUser1

The owner of a stored procedure can grant EXECUTE permissions for the stored procedure. If the owner of a base table wants to prevent users from accessing the table directly, they can grant permissions on views or stored procedures referencing the table, but not grant any permissions on the table itself. This is the foundation of the SQL Server mechanisms to ensure that users do not see data they are not authorized to access.

Users can also be granted statement permissions. Some statements, such as CREATE TABLE and CREATE VIEW, can only be executed by certain users (in this case, the **dbo** user). If the **dbo** wants another user to be able to create tables or views, they must grant the permission to execute these statements to that user.

Session Context Information

Microsoft® SQL Server[™] 2000 introduces the ability to programmatically associate up to 128 bytes of binary information with the current session or connection. Session context information enables applications to set binary values that can be referenced in multiple batches, stored procedures, triggers, or user-defined functions operating on the same session, or connection. You can set a session context by using the new SET CONTEXT_INFO statement, and then you can retrieve the context string from the new **context_info** column in the **master.dbo.sysprocesses** table.

Session context information differs from Transact-SQL variables, whose scope is limited to the current batch, stored procedure, trigger, or function. Session context information can be used to store information specific to each user or the current state of the application, which can then be used to control the logic in Transact-SQL statements.

The SET CONTEXT_INFO statement supports:

- A constant, with a maximum of 128 bytes, that is either binary or a data type that can be implicitly converted to binary.
- The name of a **varbinary(128)** or **binary(128)** variable.

SET CONTEXT_INFO cannot be specified in a user-defined function. You cannot supply a null value to SET CONTEXT_INFO because the **sysprocesses** table, where the information is stored, does not allow null values.

To get the current session context for the current connection, select the **context_info** column from the **master.dbo.sysprocesses** row whose SQL Server Process ID (SPID) is equal to the SPID for the connection. The SPID for the current connection is returned by the @@SPID function:

SELECT context_info FROM master.dbo.sysprocesses WHERE spid = @@SPID The value in the **context_info** column is initialized to 128 bytes of binary zeros if SET CONTEXT_INFO has not yet been executed for the current connection. If SET CONTEXT_INFO has been executed, the **context_info** column contains the value set by the last execution of SET CONTEXT_INFO for the current connection. The **context_info** column is a **varbinary(128)** column.

This is an example of using session context information:

```
-- Set context information at start.
SET CONTEXT INFO 0x1256698456
GO
-- Perform several non-related batches.
sp_who
GO
USE Northwind
GO
SELECT CustomerID
FROM Customers
WHERE City = 'London'
GO
-- Select context information set several batches earlier.
SELECT context info
FROM master.dbo.sysprocesses
WHERE spid = @@spid
GO
```

SET CONTEXT_INFO does not support referencing expressions other than constants or variable names, such as functions. If you need to set the context information to the result of a function call, you must first place the function call result in a **binary** or **varbinary** variable:

DECLARE @BinVar varbinary(128) SET @BinVar = CAST(REPLICATE(0x20, 128) AS varbinary(128) SET CONTEXT_INFO @BinVar

See Also

SET CONTEXT_INFO

<u>sysprocesses</u>

System Databases and Data

Microsoft[®] SQL Server[™] 2000 systems have four system databases:

• master

The **master** database records all of the system level information for a SQL Server system. It records all login accounts and all system configuration settings. **master** is the database that records the existence of all other databases, including the location of the database files. **master** records the initialization information for SQL Server; always have a recent backup of **master** available.

• tempdb

tempdb holds all temporary tables and temporary stored procedures. It also fills any other temporary storage needs such as work tables generated by SQL Server. **tempdb** is a global resource; the temporary tables and stored procedures for all users connected to the system are stored there. **tempdb** is re-created every time SQL Server is started so the system starts with a clean copy of the database. Because temporary tables and stored procedures are dropped automatically on disconnect, and no connections are active when the system is shut down, there is never anything in **tempdb** to be saved from one session of SQL Server to another.

By default, **tempdb** autogrows as needed while SQL Server is running. Unlike other databases, however, it is reset to its initial size each time the database engine is started. If the size defined for **tempdb** is small, part of your system processing load may be taken up with autogrowing **tempdb** to the size needed to support your workload each time to restart SQL Server. You can avoid this overhead by using ALTER DATABASE to increase the size of **tempdb**.

• model

The **model** database is used as the template for all databases created on a system. When a CREATE DATABASE statement is issued, the first part of the database is created by copying in the contents of the **model** database, then the remainder of the new database is filled with empty pages. Because **tempdb** is created every time SQL Server is started, the **model** database must always exist on a SQL Server system.

• msdb

The **msdb** database is used by SQL Server Agent for scheduling alerts and jobs, and recording operators.

In SQL Server 2000 and SQL Server version 7.0, every database, including the system databases, has its own set of files and does not share those files with other databases.

		Default size, typical
Database file	Physical file name	setup
master primary data	Master.mdf	11.0 MB
master log	Mastlog.ldf	1.25 MB
tempdb primary data	Tempdb.mdf	8.0 MB
tempdb log	Templog.ldf	0.5 MB
model primary data	Model.mdf	0.75 MB
model log	Modellog.ldf	0.75 MB
msdb primary data	Msdbdata.mdf	12.0 MB
msdb log	Msdblog.ldf	2.25 MB

The sizes of these files may vary slightly for different editions of SQL Server 2000. For more information about default locations of these files, see <u>Directories</u> and <u>File Locations</u>.

Each database in SQL Server 2000 contains system tables recording the data needed by the SQL Server components. The successful operation of SQL Server depends on the integrity of information in the system tables; therefore, Microsoft does not support users directly updating the information in the system tables.

Microsoft provides a complete set of administrative tools that allow users to fully administer their system and manage all users and objects in a database. Users can use the administration utilities, such as SQL Server Enterprise Manager, to directly manage the system. Programmers can use the SQL-DMO API to include complete functionality for administering SQL Server in their applications. Programmers building Transact-SQL scripts and stored procedures can use the system stored procedures and Transact-SQL DDL statements to support all administrative functions in their systems.

An important function of SQL-DMO, system stored procedures, and data definition languare (DDL) statements is to shield applications from changes in the system tables. Microsoft sometimes needs to change the system tables in new versions of SQL Server to support new functionality being added in that version. Applications issuing SELECT statements that directly reference system tables are frequently dependent on the old format of the system tables. Sites may not be able to upgrade to a new version of SQL Server until they have rewritten applications that are selecting from system tables. Microsoft considers the system stored procedures, DDL, and SQL-DMO published interfaces, and seeks to maintain the backward compatibility of these interfaces.

Microsoft does not support triggers defined on the system tables; they may alter the operation of the system.

Another important tool for querying the SQL Server catalog is the set of Information Schema Views. These views comply with the information schema defined in the SQL-92 standard. These views provide applications a standardsbased component for querying the SQL Server catalog.

You should not code Transact-SQL statements that directly query the system tables unless that is the only way to obtain the information required by the application. In most cases applications should obtain catalog and system information from:

- The SQL-92 Information Schema Views.
- SQL-DMO.
- The catalog functions, methods, attributes, or properties of the data API used in the application, such as ADO, OLE DB, or ODBC.
- Transact-SQL system stored procedures, catalog statements, and built-in functions.

Physical Database Architecture

The topics in this section describe the way Microsoft® SQL Server[™] 2000 files and databases are organized. The organization of SQL Server 2000 and SQL Server version 7.0 is different from the organization of data in SQL Server version 6.5 or earlier.

Pages and Extents

The fundamental unit of data storage in Microsoft® SQL Server[™] is the <u>page</u>. In SQL Server 2000, the page size is 8 KB. This means SQL Server 2000 databases have 128 pages per megabyte.

The start of each page is a 96-byte header used to store system information, such as the type of page, the amount of free space on the page, and the object ID of the object owning the page.

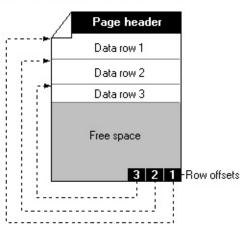
The table shows eight types of pages in the data files of a SQL Server 2000 database.

Page type	Contents
Data	Data rows with all data except text , ntext , and image
	data.
Index	Index entries.
Text/Image	Text, ntext, and image data.
Global Allocation	Information about allocated extents.
Map, Secondary	
Global Allocation	
Мар	
Page Free Space	Information about free space available on pages.
Index Allocation	Information about extents used by a table or index.
Мар	
Bulk Changed Map	Information about extents modified by bulk
	operations since the last BACKUP LOG statement.
Differential Changed	Information about extents that have changed since
Мар	the last BACKUP DATABASE statement.

Log files do not contain pages; they contain a series of log records.

Data pages contain all the data in data rows except **text**, **ntext**, and **image** data, which is stored in separate pages. Data rows are placed serially on the page starting immediately after the header. A row offset table starts at the end of the page. The row offset table contains one entry for each row on the page and each entry records how far the first byte of the row is from the start of the page. The

entries in the row offset table are in reverse sequence from the sequence of the rows on the page.



Microsoft SQL Server Data Page

Rows cannot span pages in SQL Server. In SQL Server 2000, the maximum amount of data contained in a single row is 8060 bytes, not including **text**, **ntext**, and **image** data.

Extents are the basic unit in which space is allocated to tables and indexes. An extent is 8 contiguous pages, or 64 KB. This means SQL Server 2000 databases have 16 extents per megabyte.

To make its space allocation efficient, SQL Server 2000 does not allocate entire extents to tables with small amounts of data. SQL Server 2000 has two types of extents:

- Uniform extents are owned by a single object; all eight pages in the extent can only be used by the owning object.
- Mixed extents are shared by up to eight objects.

A new table or index is usually allocated pages from mixed extents. When the table or index grows to the point that it has eight pages, it is switched to uniform extents. If you create an index on an existing table that has enough rows to generate eight pages in the index, all allocations to the index are in uniform extents.

Mixed extent

table 2	index 1	index 2	table 2	table 3	index 3	table 2	table 3	
Uniform extent								
table 1	table 1	table 1	table 1	table 1	table 1	table 1	table 1	

SQL Server Architecture

Physical Database Files and Filegroups

Microsoft® SQL Server[™] 2000 maps a database over a set of operating-system files. Data and log information are never mixed on the same file, and individual files are used only by one database.

SQL Server 2000 databases have three types of files:

• Primary data files

The primary data file is the starting point of the database and points to the other files in the database. Every database has one primary data file. The recommended file name extension for primary data files is .mdf.

• Secondary data files

Secondary data files comprise all of the data files other than the primary data file. Some databases may not have any secondary data files, while others have multiple secondary data files. The recommended file name extension for secondary data files is .ndf.

• Log files

Log files hold all of the log information used to recover the database. There must be at least one log file for each database, although there can be more than one. The recommended file name extension for log files is .ldf.

SQL Server 2000 does not enforce the .mdf, .ndf, and .ldf file name extensions, but these extensions are recommended to help identify the use of the file.

In SQL Server 2000, the locations of all the files in a database are recorded in both the **master** database and the primary file for the database. Most of the time the database engine uses the file location information from the **master** database. For some operations, however, the database engine uses the file location information from the primary file to initialize the file location entries in the **master** database:

• When attaching a database using the **sp_attach_db** system stored procedure.

- When upgrading from SQL Server version 7.0 to SQL Server 2000.
- When restoring the **master** database.

SQL Server 2000 files have two names:

• *logical_file_name* is a name used to refer to the file in all Transact-SQL statements.

The logical file name must conform to the rules for SQL Server identifiers and must be unique to the database.

• *os_file_name* is the name of the physical file.

It must follow the rules for Microsoft Windows NT® or Microsoft Windows® 98, and Microsoft Windows 95 file names.

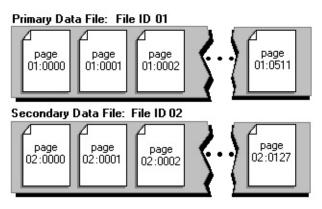
These are examples of the logical file names and physical file names of a database created on a default instance of SQL Server 2000:

MyDB primary c:\Program Files\Microsoft SQL Server\MSSQL\Data\MyData1.mdf Primary data file MyDB_secondary1 c:\Program Files\Microsoft SQL Server\MSSQL\Data\MyData2.ndf Secondary data file MyDB_secondary2 c:\Program Files\Microsoft SQL Server\MSSQL\Data\MyData3.ndf Secondary data file -----MyDB_log1 c:\Program Files\Microsoft SQL Server\MSSQL\Data\MyLog4.ldf Log file MvDB log2 c:\Program Files\Microsoft SQL Server\MSSQL\Data\MyLog5.ldf Log file

SQL Server data and log files can be placed on either FAT or NTFS file systems, but cannot be placed on compressed file systems.

Pages in a SQL Server 2000 data file are numbered sequentially starting with 0

for the first page in the file. Each file has a file ID number. Uniquely identifying a page in a database requires both the file ID and page number. The following example shows the page numbers in a database that has a 4-MB primary data file and a 1-MB secondary data file.



The first page in each file is a file header page containing information about the attributes of the file. Several of the other pages at the start of the file also contain system information, such as allocation maps. One of the system pages stored in both the primary data file and the first log file is a database boot page containing information about the attributes of the database.

SQL Server 2000 files can grow automatically from their originally specified size. When you define a file, you can specify a growth increment. Each time the file fills, it increases its size by the growth increment. If there are multiple files in a filegroup, they do not autogrow until all the files are full. Growth then occurs using a round-robin algorithm.

Each file can also have a maximum size specified. If a maximum size is not specified, the file can continue to grow until it has used all available space on the disk. This feature is especially useful when SQL Server is used as a database embedded in an application where the user does not have ready access to a system administrator. The user can let the files autogrow as needed to lessen the administrative burden of monitoring the amount of free space in the database and allocating additional space manually.

When multiple instances of SQL Server are run on a single computer, each instance gets a different default directory to hold the files for the databases created in the instance. For more information, see <u>Directories and File Locations</u>.

Database Filegroups

Database files can be grouped together in filegroups for allocation and administration purposes. Some systems can improve their performance by controlling the placement of data and indexes onto specific disk drives. Filegroups can aid this process. The system administrator can create filegroups for each disk drive, then assign specific tables, indexes, or the **text**, **ntext**, or **image** data from a table, to specific filegroups.

No file can be a member of more than one filegroup. Tables, indexes, and **text**, **ntext**, and **image** data can be associated with a filegroup, in which case all their pages will be allocated in that filegroup.

Log files are never a part of a filegroup. Log space is managed separately from data space.

Files in a filegroup will not autogrow unless there is no space available on any of the files in the filegroup.

There are two types of filegroups:

• Primary

The primary filegroup contains the primary data file and any other files not specifically assigned to another filegroup. All pages for the system tables are allocated in the primary filegroup.

• User-defined

User-defined filegroups are any filegroups specified using the FILEGROUP keyword in a CREATE DATABASE or ALTER DATABASE statement.

One filegroup in each database operates as the default filegroup. When SQL Server allocates a page to a table or index for which no filegroup was specified when they were created, the pages are allocated from the default filegroup. Only one filegroup at a time can be the default filegroup. Members of the **db_owner** fixed database role can switch the default filegroup from one filegroup to another. If no default filegroup is specified, the primary filegroup is the default filegroup.

SQL Server 2000 can work quite effectively without filegroups, so many systems will not need to specify user-defined filegroups. In this case, all files are included in the primary filegroup and SQL Server 2000 can allocate data

anywhere in the database. Filegroups are not the only method that can be used to distribute I/O across multiple drives.

Members of the **db_owner** fixed database role can back up and restore individual files or filegroups instead of backing up or restoring an entire database.

The following example creates a database on a default instance of SQL Server 2000. The database has a primary data file, a user-defined filegroup, and a log file. The primary data file is in the primary filegroup and the user-defined filegroup has two secondary data files. An ALTER DATABASE statement makes the user-defined filegroup the default. A table is then created specifying the user-defined filegroup.

USE master

GO

```
-- Create the database with the default data
```

```
-- filegroup and the log file. Specify the
```

```
-- growth increment and the max size for the
```

```
-- primary data file.
```

```
CREATE DATABASE MyDB
```

ON PRIMARY

```
( NAME='MyDB_Primary',
```

```
FILE NAME=
```

```
'c:\Program Files\Microsoft SQL Server\MSSQL\data\MyDB_Prm SIZE=4,
```

```
MAXSIZE=10,
```

```
FILEGROWTH=1),
```

```
FILEGROUP MyDB_FG1
```

```
(NAME = 'MyDB_FG1_Dat1',
```

```
FILE NAME =
```

```
'c:\Program Files\Microsoft SQL Server\MSSQL\data\MyDB_FG1
SIZE = 1MB,
MAXSIZE=10,
FILEGROWTH=1),
( NAME = 'MyDB FG1 Dat2',
```

```
FILE NAME =
   'c:\Program Files\Microsoft SQL Server\MSSQL\data\MyDB_FG1
 SIZE = 1MB,
 MAXSIZE=10,
 FILEGROWTH=1)
LOG ON
(NAME='MyDB_log',
 FILE NAME =
  'c:\Program Files\Microsoft SQL Server\MSSQL\data\MyDB.ldf',
 SIZE=1,
 MAXSIZE=10,
 FILEGROWTH=1)
GO
ALTER DATABASE MyDB
MODIFY FILEGROUP MyDB_FG1 DEFAULT
GO
```

```
-- Create a table in the user-defined filegroup.
USE MyDB
CREATE TABLE MyTable
( cola int PRIMARY KEY,
 colb char(8) )
ON MyDB_FG1
GO
```

Database: MyDB
Primary file group c:\Program Files\Microsoft SQL Server\MSSQL\Data\MyDB_Prm.mdf
4 MB
Log file
c:\Program Files\Microsoft SQL Server\MSSQL\Data\MyDB.ldf 1 MB
MyDB_FG1 file group
c:\Program Files\Microsoft SQL Server\MSSQL\Data\MyDB_FG1_1.ndf
1 MB
c:\Program Files\Microsoft SQL Server\MSSQL\Data\MyDB_FG1_2.ndf
1 MB

User filegroups can be made read-only. The data cannot be altered, but the catalog can still be modified to allow work such as permissions management.

SQL Server 2000 databases can be detached from a server and reattached to either another server or the same server. This is especially useful in making databases distributed for use on a customer's local SQL Server installation. For example, a company could create a database containing their current product catalog. The company could create this database on a writable compact disc drive and make the database read-only. They could then copy the compact disc and send copies to all of their field sales representatives equipped with a catalog application and SQL Server on Windows 95 laptops. The sales representatives would then have the latest catalog information. SQL Server Architecture

Space Allocation and Reuse

Microsoft® SQL Server[™] 2000 is effective at quickly allocating pages to objects and reusing space freed up by deleted rows. These operations are internal to the system and use data structures not visible to users, yet these processes and structures are occasionally referenced in SQL Server messages. This topic is an overview of the space allocation algorithms and data structures to give users and administrators the knowledge needed to understand references to the terms in messages generated by SQL Server.

Managing Extent Allocations and Free Space

The Microsoft® SQL Server[™] 2000 data structures that track free space have a relatively simple structure. This has two benefits:

• The free space information is densely packed, so there are relatively few pages containing this information.

This increases speed by reducing the amount of disk reads necessary to retrieve allocation information, and increasing the chance the allocation pages will remain in memory, eliminating even more reads.

• Most of the allocation information is not chained together, which simplifies the maintenance of the allocation information.

Each page allocation or deallocation can be performed quickly, decreasing the contention between concurrent tasks needing to allocate or free pages.

SQL Server uses two types of allocation maps to record the allocation of extents:

• Global Allocation Map (GAM)

GAM pages record what extents have been allocated. Each GAM covers 64,000 extents, or nearly 4 GB of data. The GAM has one bit for each extent in the interval it covers. If the bit is 1, the extent is free; if the bit is 0, the extent is allocated.

• Shared Global Allocation Map (SGAM)

SGAM pages record what extents are currently used as mixed extents and have at least one unused page. Each SGAM covers 64,000 extents, or nearly 4 GB of data. The SGAM has one bit for each extent in the interval it covers. If the bit is 1, the extent is being used as a mixed extent and has free pages; if the bit is 0, the extent is not used as a mixed extent, or it is a mixed extent whose pages are all in use.

Each extent has the following bit patterns set in the GAM and SGAM based on its current use.

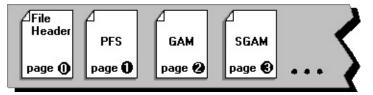
Current use of extent	GAM bit setting	SGAM bit setting
Free, not in use	1	0
Uniform extent, or full mixed	0	0
extent		
Mixed extent with free pages	0	1

This results in simple extent management algorithms. To allocate a uniform extent, SQL Server searches the GAM for a 1 bit and sets it to 0. To find a mixed extent with free pages, SQL Server searches the SGAM for a 1 bit. To allocate a mixed extent, SQL Server searches the GAM for a 1 bit, sets it to 0, and then also sets the corresponding bit in the SGAM to 1. To free an extent, SQL Server ensures the GAM bit is set to 1 and the SGAM bit is set to 0. The algorithms actually used internally by SQL Server are more sophisticated than what is stated here (SQL Server distributes data evenly in a database), but even the real algorithms are simplified by not having to manage chains of extent allocation information.

Page Free Space (PFS) pages record whether an individual page in a heap or an **ntext**, **text**, or **image** column has been allocated, and the amount of space free on each page. Each PFS page covers approximately 8,000 pages. For each page, the PFS has a bitmap recording whether the page is empty, 1-50% full, 51-80% full, 81-95% full, or 96-100% full.

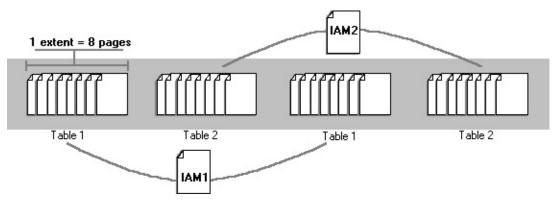
After an extent has been allocated to an object, SQL Server uses the PFS pages to record which pages in the extent are allocated or free, and how much free space is available for use. This information is used when SQL Server has to allocate a new page, or when it needs to find a page with free space available to hold a newly inserted row.

A PFS page is the first page after the file header page in a data file (with page number 1). Next comes a GAM (with page number 2) followed by an SGAM (page 3). There is a PFS page approximately 8,000 pages after the first. There is another GAM each 64,000 extents after the first GAM on page 2, and another SGAM each 64,000 extents after the first SGAM on page 3.

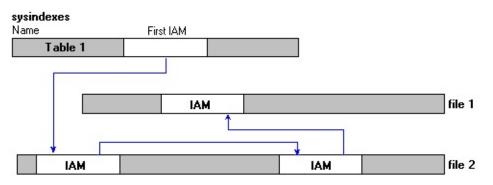


Managing Space Used by Objects

Index Allocation Map (IAM) pages map the extents in a database file used by a heap or index. IAM pages also map the extents allocated to the **ntext**, **text**, and **image** page chain for any table that has columns of these types. Each of these objects has a chain of one or more IAM pages recording all the extents allocated to it. Each object has at least one IAM for each file on which it has extents. They may have more than one IAM on a file if the range of the extents on the file allocated to the object exceeds the range that an IAM can record.



IAM pages are allocated as needed for each object and are located randomly in the file. **sysindexes.dbo.FirstIAM** points to the first IAM page for an object, and all the IAM pages for that object are linked in a chain.



An IAM page has a header indicating the starting extent of the range of extents mapped by the IAM. The IAM also has a large bitmap in which each bit represents one extent. The first bit in the map represents the first extent in the range, the second bit represents the second extent, and so on. If a bit is 0, the extent it represents is not allocated to the object owning the IAM. If the bit is 1, the extent it represents is allocated to the object owning the IAM page.

When Microsoft® SQL Server[™] 2000 needs to insert a new row and no space is available in the current page, it uses the IAM and PFS pages to find a page with enough space to hold the row. SQL Server uses the IAM pages to find the extents allocated to the object. For each extent, SQL Server searches the PFS pages to see if there is a page with enough free space to hold the row. Each IAM and PFS page covers a large number of data pages, so there are few IAM and PFS pages in a database. This means that the IAM and PFS pages are generally in memory in the SQL Server buffer pool, so they can be searched quickly.

SQL Server allocates a new extent to an object only when it cannot quickly find a page in an existing extent with enough space to hold the row being inserted. SQL Server allocates extents from those available in the filegroup using a proportional allocation algorithm. If a filegroup has two files, one of which has twice the free space of the other, two pages will be allocated from the file with more empty space for every one page allocated from the other file. This means that every file in a filegroup should have a similar percentage of space used.

Tracking Modified Extents

SQL Server 2000 introduces two new internal data structures to track extents modified by bulk copy operations or modified since the last full backup. These new data structures greatly speed differential backups and logging bulk copy operations when a database is using the bulk-logged recovery model. Like the Global Allocation Map (GAM) and Secondary Global Allocation Map (SGAM) pages, these new structures are bitmaps where each bit represents a single extent.

• Differential Changed Map (DCM)

Tracks the extents that have changed since the last BACKUP DATABASE statement. If the bit for an extent is 1, the extent has been modified since the last BACKUP DATABASE statement. If the bit is 0, the extent has not been modified.

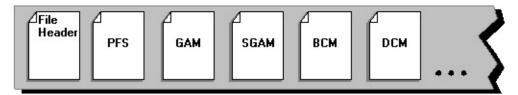
Differential backups can now read just the DCM pages to find out which extents have been modified. This greatly reduces the number of pages that a differential backup must scan. The length of time a differential backup runs is now proportional to the number of extents modified since the last BACKUP DATABASE statement, not the overall size of the database.

• Bulk Changed Map (BCM)

Tracks the extents that have been modified by bulk logged operations since the last BACKUP LOG statement. If the bit for an extent is 1, the extent has been modified by a bulk logged operation after the last BACKUP LOG statement. If the bit is 0, the extent has not been modified by bulk logged operations.

BCM pages are only relevant when the database is using the bulklogged recovery model. In this recovery model, when a BACKUP LOG is performed, the backup process scans the BCMs for extents that have been modified and includes those extents in the log backup. This allows the bulk logged operations to be recovered if the database is restored from a database backup and a sequence of transaction log backups. BCM pages are not relevant in a database is using the simple recovery model because no bulk logged operations are logged. They are not relevant in a database using the full recovery model because that recovery model treats bulk logged operations as fully logged operations.

The interval between DCM pages and BCM pages is the same as the interval between GAM and SGAM pages; 64,000 extents. The DCM and BCM pages are located behind the GAM and SGAM pages in a physical file:



See Also

Managing Extent Allocations and Free Space

Shrinking Databases

SQL Server 2000 autoshrinks databases that have a large amount of free space. Only those databases where the **autoshrink** option has been set to true are candidates for this process. The server checks the space usage in each database periodically. If a database is found with a lot of empty space and it has the **autoshrink** option set to true, SQL Server reduces the size of the files in the database. You can also use SQL Server Enterprise Manager or the DBCC SHRINKDATABASE and DBCC SHRINKFILE statements to shrink the files of a database manually.

Files are always shrunk from the end. For example, if you have a 5 GB file and specify 4GB as the *target_size* in a DBCC SHRINKDB statement, SQL Server will free as much space as it can from the last 1 GB of the file. If there are used pages in the part of the file being released, SQL Server first relocates the pages to the part being retained. You can only shrink a database to the point where it has no free space remaining. For example, if a 5GB database has 4 GB of data and you specify 3 GB as the *target_size* of a DBCC SHRINKDATABASE statement, only 1 GB will be freed.

If a DBCC SHRINKDATABASE or DBCC SHRINKFILE statement cannot reclaim all the space in a log file, the statement will issue an informational message indicating what action you must perform to make more space eligible to be freed. For more information about shrinking log files, see <u>Shrinking the Transaction Log</u>.

SQL Server Architecture

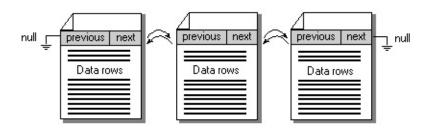
Table and Index Architecture

Objects in a Microsoft® SQL Server[™] 2000 database are stored as a collection of 8-KB pages. This topic describes the way the pages for tables and indexes are organized.

SQL Server 2000 supports indexes on views. The first index allowed on a view is a clustered index. At the time a CREATE INDEX statement is executed on a view, the result set for the view is materialized and stored in the database with the same structure as a table that has a clustered index. The result set that is stored is the same as that which is produced by this statement.

SELECT * FROM ViewName

The data rows for each table or indexed view are stored in a collection of 8-KB data pages. Each data page has a 96-byte header containing system information such as the identifier (ID) of the table that owns the page. The page header also includes pointers to the next and previous pages that are used if the pages are linked in a list. A row offset table is at the end of the page. Data rows fill the rest of the page.



Organization of Data Pages

SQL Server 2000 tables use one of two methods to organize their data pages:

• Clustered tables are tables that have a clustered index.

The data rows are stored in order based on the clustered index key. The index is implemented as a B-tree index structure that supports fast retrieval of the rows based on their clustered index key values. The pages in each level of the index, including the data pages in the leaf level, are linked in a doubly-linked list, but navigation from one level to

another is done using key values.

• Heaps are tables that have no clustered index.

The data rows are not stored in any particular order, and there is no particular order to the sequence of the data pages. The data pages are not linked in a linked list.

Indexed views have the same storage structure as clustered tables.

SQL Server also supports up to 249 nonclustered indexes on each table or indexed view. The nonclustered indexes have a B-tree index structure similar to the one in clustered indexes. The difference is that nonclustered indexes have no effect on the order of the data rows. Clustered tables and indexed views keep their data rows in order based on the clustered index key. The collection of data pages for a heap is not affected if nonclustered indexes are defined for the table. The data pages remain in a heap unless a clustered index is defined.

The pages holding **text**, **ntext**, and **image** data are managed as a single unit for each table. All of the **text**, **ntext**, and **image** data for a table is stored in one collection of pages.

All of the page collections for tables, indexes and indexed views are anchored by page pointers in the **sysindexes** table. Every table and indexed view has one collection of data pages, plus additional collections of pages to implement each index defined for the table or view.

Each table, index and indexed view has a row in **sysindexes** uniquely identified by the combination of the object identifier (**id**) column and the index identifier (**indid**) column. The allocation of pages to tables, indexes, and indexed views is managed by a chain of IAM pages. The column **sysindexes.FirstIAM** points to first IAM page in the chain of IAM pages managing the space allocated to the table, index or indexed view.

Each table has a set of rows in **sysindexes**:

• A heap has a row in **sysindexes** with **indid** = 0.

The **FirstIAM** column points to the IAM chain for the collection of data pages for the table. The server uses the IAM pages to find the pages in the data page collection because they are not linked together.

• A clustered index on a table or view has a row in **sysindexes** with **indid** = 1.

The **root** column points to the top of the clustered index B-tree. The server uses the index B-tree to find the data pages.

• Each nonclustered index created for a table or view has a row in **sysindexes**.

The values for **indid** in the rows for each nonclustered index range from 2 through 250. The **root** column points to the top of the nonclustered index B-tree.

• Each table that has at least one **text**, **ntext**, or **image** column also has a row in **sysindexes** with **indid** = 255.

The column **FirstIAM** points to the chain of IAM pages that manage the **text**, **ntext**, and **image** pages.

In SQL Server version 6.5 and earlier, **sysindexes.first** always points to the start of a heap, the start of the leaf level of an index, or the start of a chain of **text** and **image** pages. In SQL Server version 7.0 and later, **sysindexes.first** is largely unused. In SQL Server version 6.5 and earlier, **sysindexes.root** in a row with **indid** = 0 points to the last page in a heap. In SQL Server version 7.0 and later, **sysindexes.root** in a row with **indid** = 0 is unused.

Distribution Statistics

All indexes have distribution statistics that describe the selectivity and distribution of the key values in the index. Selectivity is a property that relates to how many rows are typically identified by a key value. A unique key has high selectivity; a key value found in 1,000 rows has poor selectivity. The selectivity and distribution statistics are used by Microsoft® SQL Server[™] 2000 to optimize its navigation through tables and indexed views when processing Transact-SQL statements. The distribution statistics are used to estimate how efficient an index would be in retrieving data associated with a key value or range specified in the query. The statistics for each index are not limited to a single page but are stored as a long string of bits across multiple pages in the same way **image** data is stored. The column **sysindexes.statblob** points to this distribution data. You can use the DBCC SHOW_STATISTICS statement to get a report on the distribution statistics for an index.

Distribution statistics may also be maintained for unindexed columns. These can be defined manually using the CREATE STATISTICS statement or created automatically by the query optimizer. Statistics on unindexed columns count against the limit of 249 nonclustered indexes allowed on a table.

To be useful to query optimizer, distribution statistics must be kept reasonably current. The distribution statistics should be refreshed anytime significant numbers of changes to keys occur in the index. Distribution statistics can be updated manually using the UPDATE STATISTICS statement. SQL Server 2000 can also detect when distribution statistics are out of date and update the statistics automatically. This update is performed by the task that detected that the statistics needed to be updated. The update is performed using a complex sampling method that minimizes the effect of the update on transaction throughput.

See Also

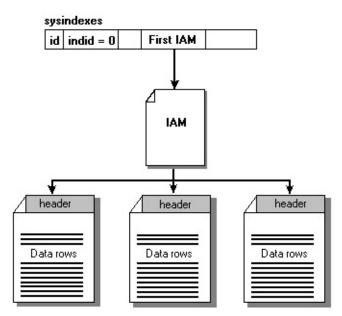
Statistical Information

SQL Server Architecture

Heap Structures

Heaps have one row in **sysindexes** with **indid** = 0. The column **sysindexes.FirstIAM** points to the first IAM page in the chain of IAM pages that manage the space allocated to the heap. Microsoft® SQL ServerTM 2000 uses the IAM pages to navigate through the heap. The data pages and the rows within them are not in any specific order, and are not linked together. The only logical connection between data pages is that recorded in the IAM pages.

Table scans or serial reads of a heap can be done by scanning the IAM pages to find the extents holding pages for the heap. Because the IAM represents extents in the same order they exist in the data files, this means that serial heap scans progress uniformly down each file. This is more efficient than following the data page chains used in earlier versions of SQL Server, in which the data page chain often takes a somewhat random path through the files of a database. Using the IAM pages to set the scan sequence also means that rows from the heap are not typically returned in the order in which they were inserted.



SQL Server Architecture

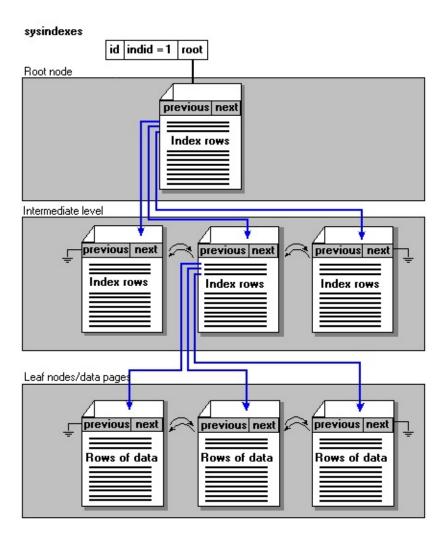
Clustered Indexes

Clustered indexes have one row in **sysindexes** with **indid** = 1. The pages in the data chain and the rows in them are ordered on the value of the clustered index key. All inserts are made at the point the key value in the inserted row fits in the ordering sequence.

Microsoft® SQL Server[™] 2000 indexes are organized as B-trees. Each page in an index holds a page header followed by index rows. Each index row contains a key value and a pointer to either a lower-level page or a data row. Each page in an index is called an index node. The top node of the B-tree is called the root node. The bottom layer of nodes in the index are called the leaf nodes. The pages in each level of the index are linked together in a doubly-linked list. In a clustered index, the data pages make up the leaf nodes. Any index levels between the root and the leaves are collectively known as intermediate levels.

For a clustered index, **sysindexes.root** points to the top of the clustered index. SQL Server navigates down the index to find the row corresponding to a clustered index key. To find a range of keys, SQL Server navigates through the index to find the starting key value in the range, and then scans through the data pages using the previous or next pointers. To find the first page in the chain of data pages, SQL Server follows the leftmost pointers from the root node of the index.

This illustration shows the structure of a clustered index.



Nonclustered Indexes

Nonclustered indexes have the same B-tree structure as clustered indexes, with two significant differences:

- The data rows are not sorted and stored in order based on their nonclustered keys.
- The leaf layer of a nonclustered index does not consist of the data pages.

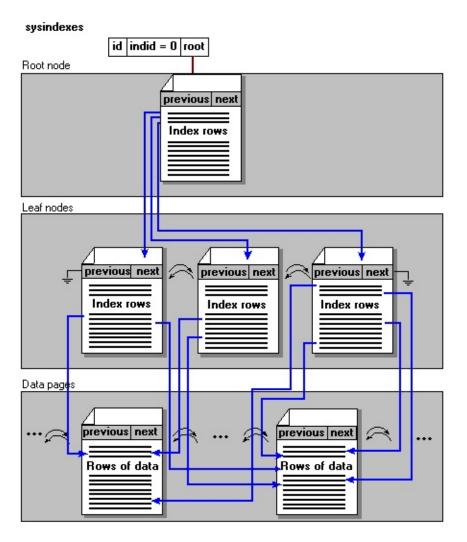
Instead, the leaf nodes contain index rows. Each index row contains the nonclustered key value and one or more row locators that point to the data row (or rows if the index is not unique) having the key value.

Nonclustered indexes can be defined on a table with a clustered index, a heap, or an indexed view. In Microsoft® SQL ServerTM 2000, the row locators in nonclustered index rows have two forms:

- If the table is a heap (does not have a clustered index), the row locator is a pointer to the row. The pointer is built from the file identifier (ID), page number, and number of the row on the page. The entire pointer is known as a Row ID.
- If the table does have a clustered index, or the index is on an indexed view, the row locator is the clustered index key for the row. If the clustered index is not a unique index, SQL Server 2000 makes duplicate keys unique by adding an internally generated value. This value is not visible to users; it is used to make the key unique for use in nonclustered indexes. SQL Server retrieves the data row by searching the clustered index using the clustered index key stored in the leaf row of the nonclustered index.

Because nonclustered indexes store clustered index keys as their row locators, it is important to keep clustered index keys as small as possible. Do not choose

large columns as the keys to clustered indexes if a table also has nonclustered indexes.



tempdb and Index Creation

When you create an index, you can specify WITH SORT_IN_TEMPDB option, which directs the database engine to use **tempdb** to store the intermediate sort results used to build the index. Although this option increases the amount of disk space used to create an index, it reduces the time it takes to create an index when **tempdb** is on a different set of disks than the user database.

As the database engine builds an index, it goes through two phases:

- The database engine first scans the data pages to retrieve key values and builds a index leaf row for each data row. When the internal sort buffers have been filled with leaf index entries, the entries are sorted and written to disk as an intermediate sort run. The database engine then resumes the data page scan until the sort buffers are again filled. This pattern of scanning multiple data pages followed by sorting and writing a sort run continues until all the rows of the base table have been processed. In a clustered index, the leaf rows of the index are the data rows. In a nonclustered index, the leaf rows do not contain values from nonkey columns, so are generally smaller. A nonclustered sort run can be large, however, if the index keys are large.
- The database engine merges the sorted runs of index leaf rows into a single, sorted stream. The sort merge component of the engine starts with the first page of each sort run, finds the lowest key in all the pages, and passes that leaf row to the index create component. The next lowest key is then processed, then the next, and so on. When the last leaf index row is extracted from a sort run page, the process shifts to the next page from that sort run. When all the pages in a sort run extent have been processed, the extent is freed. As each leaf index row is passed to the index create component, it is placed in a leaf index page in the buffer. Each leaf page is written as it is filled. As leaf pages are written, the database engine also builds the upper levels of the index. Each upper level index page is written when it is filled.

If you create a clustered index on a table that has existing nonclustered indexes, the general process is:

- The nonclustered indexes are deallocated, but the definitions of the indexes are retained. The space is not available for use until the end of the transaction containing the CREATE INDEX statement, so that the old index pages are still available if they have to be restored during a rollback of the transaction.
- The clustered index is created.
- The nonclustered indexes are re-created.

When SORT_IN_TEMPDB is not specified, the sort runs are stored in the destination filegroup. During the first phase of creating the index, the alternating reads of the base table pages and writes of the sort runs move the disk read-write heads from one area of the disk to another. The heads are in the data page area as the data pages are scanned. They move to an area of free space when the sort buffers fill and the current sort run has to be written to disk, then move back to the data page area as the table page scan is resumed. The read-write head movement is higher in the second phase. At that time the sort process is typically alternating reads from each sort run area. Both the sort runs and the new index pages are built in the destination filegroup, meaning that at the same time the database engine is spreading reads across the sort runs, it has to periodically jump to the index extents to write new index pages as they are filled.

If the SORT_IN_TEMPDB option is specified and **tempdb** is on a separate set of disks from the destination filegroup, then during the first phase the reads of the data pages occur on a different disk than the writes to the sort work area in **tempdb**. This means the disk reads of the data keys tend to proceed more serially across the disk, and the writes to the **tempdb** disk also tend to be serial, as do the writes to build the final index. Even if other users are using the database and accessing separate disk addresses, the overall pattern of reads and writes are more efficient when SORT_IN_TEMPDB is specified than when it is not.

The SORT_IN_TEMPDB option may improve the contiguity of index extents, especially if the CREATE INDEX is not being processed in parallel. The sort

work area extents are freed on a somewhat random basis with respect to their location in the database. If the sort work areas are contained in the destination filegroup, then as the sort work extents are freed, they can be acquired by the requests for extents to hold the index structure as it is built. This can randomize the locations of the index extents to a certain degree. If the sort extents are held separately in **tempdb**, the sequence in which they are freed has no bearing on the location of the index extents. Also, when the intermediate sort runs are stored in **tempdb** instead of the destination filegroup, there is more space available in the destination filegroup, which increases the chances that index extents will be contiguous.

The SORT_IN_TEMPDB option affects only the current statement. No meta data records that the index was or was not sorted in **tempdb**. For example, if you create a nonclustered index using the SORT_IN_TEMPDB option, and later create a clustered index without specifying the option, the database engine does not use the option when it re-creates the nonclustered index.

Free Space Requirements

When you specify the SORT_IN_TEMPDB option, you must have sufficient free space available in **tempdb** to hold the intermediate sort runs, and enough free space in the destination filegroup to hold the new index. The CREATE INDEX statement fails if there is not enough free space and there is some reason the databases cannot autogrow to acquire more space (such as no space on the disk, or autogrow turned off).

If SORT_IN_TEMPDB is not specified, the available free space in the destination filegroup must be roughly the size of the final index. During the first phase, the sort runs are built and require about the same amount of space as the final index. During the second phase, each sort run extent is freed after it has been processed. This means that sort run extents are freed at about the same rate at which extents are acquired to hold the final index pages, so the overall space requirements do not greatly exceed the size of the final index. One side effect of this is that if the amount of free space is very close to the size of the final index, the database engine will tend to reuse the sort run extents very quickly after they are freed. Because the sort run extents are freed in a somewhat random manner, this reduces the continuity of the index extents in this scenario. If SORT_IN_TEMPDB is not specified, the continuity of the index extents is

improved if there is enough free space available in the destination filegroup that the index extents can be allocated from a contiguous pool rather than from the freshly deallocated sort run extents.

At the time you execute the CREATE INDEX statement, you must have available as free space:

- When you create a nonclustered index:
 - If SORT_IN_TEMPDB is specified, there must be enough free space in **tempdb** to store the sort runs, and enough free space in the destination filegroup to store the final index structure. The sort runs contain the leaf rows of the index.
 - If SORT_IN_TEMPDB is not specified, the free space in the destination filegroup must be large enough to store the final index structure. The continuity of the index extends may be improved if more free space is available.
- When you create a clustered index on a table that does not have nonclustered indexes:
 - If SORT_IN_TEMPDB is specified, there must be enough free space in **tempdb** to store the sort runs, which include the data rows of the table. There must be enough free space in the destination filegroup to store the final index structure, including the data rows of the table and the index B-tree. A rough estimate is 1.2 times the size of the original table, although you may need to adjust the estimate for factors such as having a large key size or a fillfactor with a low value.
 - If SORT_IN_TEMPDB is not specified, the free space in the destination filegroup must be large enough to store the final table, including the index structure. The continuity of the table and index extents may be improved if more free space is available.
- When you create a clustered index on a table that has nonclustered indexes:

- If SORT_IN_TEMPDB is specified, there must be enough free space in **tempdb** to store the collection of sort runs for the largest index (typically the clustered index), and enough free space in the destination filegroup to store the final structures of all the indexes, including the clustered index that contains the data rows of the table.
- If SORT_IN_TEMPDB is not specified, the free space in the destination filegroup must be large enough to store the final table, including the structures of all the indexes. The continuity of the table and index extents may be improved if more free space is available.

See Also

CREATE INDEX

text, ntext, and image Data

Individual **text**, **ntext**, and **image** values can be a maximum of 2-GB, which is too long to store in a single data row. In Microsoft® SQL Server[™] 2000, small **text**, **ntext**, or **image** values can be stored directly in the row, but values too large to fit in the row are stored in a collection of pages separate from the pages holding the data for the other columns of the row.

The administrator uses the **text in row** option in **sp_tableoption** to specify whether small **text**, **ntext**, or **image** values are stored directly in a row:

- When **text in row** is OFF, SQL Server 2000 has the same **ntext**, **text**, and **image** behavior as SQL Server version 7.0. For each **text**, **ntext**, or **image** value, all that is stored in the data row is a 16-byte pointer. For each row, this pointer points to the location of the **text**, **ntext**, or **image** data. A row containing multiple **text**, **ntext**, or **image** columns has one pointer for each **text**, **ntext**, or **image** column.
- When **text in row** is ON, SQL Server 2000 stores small **text**, **ntext**, and **image** values in the data row. Only **text**, **ntext**, or **image** values that cannot fit in the row are stored in a separate collection of pages.

Each table has only one collection of pages to hold **text**, **ntext**, and **image** data. The **sysindexes** row that has **indid** = 255 is the anchor for the collection. The **text**, **ntext**, and **image** data for all the rows in the table is interleaved in this collection of **text** and **image** pages.

In SQL Server 2000, individual **text**, **ntext**, and **image** pages are not limited to holding data for only one occurrence of a **text**, **ntext**, or **image** column. A **text**, **ntext**, or **image** page can hold data from multiple rows; the page can even have a mix of **text**, **ntext**, and **image** data.

Although the user always works with **text**, **ntext**, and **image** data as if it is a single long string of bytes, the data is not stored in that format. The data is stored in a collection of 8-KB pages that are not necessarily located next to each other. In SQL Server 2000, the pages are organized logically in a B-tree structure, and

in SQL Server version 6.5 and earlier they are linked in a page chain. The advantage of the method used by SQL Server 2000 is that operations starting in the middle of the string are more efficient. SQL Server 2000 can quickly navigate the B-tree, and SQL Server version 6.5 must scan through the page chain.

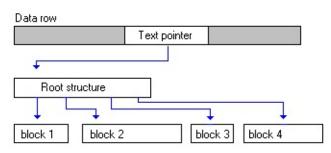
See Also

<u>sp_tableoption</u>

ntext, text, and image Data When text in row Is Set to OFF

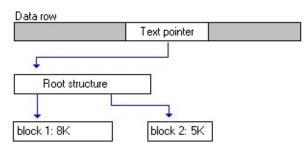
The structure of the B-tree used to store **text**, **ntext**, or **image** data when the **text in row** option of **sp_tableoption** is set to OFF differs slightly if there is less than 32 KB of data than if there is more.

If there is less than 32 KB of data, the 16-byte text pointer in the data row points to an 84-byte text root structure. This forms the root node of the B-tree structure. The root node points to the blocks of **text**, **ntext**, or **image** data.



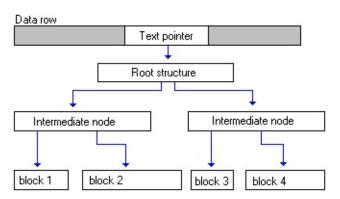
Although the data for **text**, **ntext**, and **image** columns is arranged logically in a B-tree, both the root node and the individual blocks of data are spread throughout the chain of **text**, **ntext**, and **image** pages for the table. They are placed wherever there is space available. The size of each block of data is determined by the size written by an application. Small blocks of data will be combined to fill a page. If there is less than 64 bytes of data, it is all stored in the root structure.

For example, if an application first writes 1 KB of **image** data, this is stored as the first 1-KB block of **image** data for the row. If the application then writes 12 KB of **image** data, then 7 KB is combined with the first 1-KB block so the first block becomes 8 KB. The remaining 5 KB forms the second block of **image** data. (The actual capacity of each **ntext**, **text**, or **image** page is 8080 bytes of data.)



Because the blocks of **text**, **ntext**, or **image** data and the root structures can all share space on the same **text**, **ntext**, or **image** pages, SQL Server 7.0 uses less space with small amounts of **text**, **ntext**, or **image** data than earlier versions of SQL Server. For example, if you insert 20 rows that each have 200 bytes of data in a **text** column, the data and all the root structures can all fit on the same 8-KB page.

If the amount of data for one occurrence of a **text**, **ntext**, or **image** column exceeds 32 KB, SQL Server starts building intermediate nodes between the data blocks and the root node.



The root structure and the data blocks are interleaved throughout the **text**, **ntext**, or **image** pages in the same manner as described earlier. The intermediate nodes, however, are stored in pages not shared between occurrences of **text**, **ntext**, or **image** columns. A page storing intermediate nodes contains only intermediate nodes for one **ntext**, **text**, or **image** data value in one data row.

See Also

sp_tableoption

text, ntext, and image Data When text in row Is Set to ON

You enable the **text in row** option for a table by using **sp_tableoption**. With the **text in row** option set to ON, Microsoft® SQL Server[™] 2000 stores **text**, **ntext**, or **image** strings directly in the data row if:

- The length of the string is shorter than the specified limit.
- There is enough space available in the data row to hold the string.

When the **text**, **ntext**, or **image** string is stored in the data row, SQL Server does not have to access a separate page or set of pages to read or write the string. This makes reading and writing the **text**, **ntext**, or **image** in-row strings about as fast as reading or writing **varchar**, **nvarchar**, or **varbinary** strings.

If a **text**, **ntext**, or **image** string is longer than the **text in row** option limit or the available space in the row, the set of pointers otherwise stored in the root node of the pointer tree are stored in the row. Moving the root node to the row itself allows SQL Server to eliminate a page access each time it references the string value, which speeds processing.

A full root structure placed in a data row requires 72 bytes to hold five pointers. If the **text in row** option limit is less than 72 bytes, or if there are fewer than 72 bytes available in the row, SQL Server puts as many pointers as it can in the row. The lowest limit is 24 bytes, which holds a root node with only one pointer.

Reducing the number of pointers in the root structure truncates the top level of the tree structure used to store the **text**, **ntext**, or **image** string. For example, if the root structure has only three pointers, the top level of the tree structure can only contain three nodes, not five. Reducing the size of the root structure can introduce extra layers in the tree structure. Setting the **text in row** option limit under 72 can also cause the top level to be truncated.

When **text**, **ntext**, or **image** strings are stored in the row, they are stored similarly to variable-length strings. For example, if the **text in row** option limit

is 500 bytes and you store a 200-byte string in a row, SQL Server uses only the number of bytes needed to store the string. If a string longer than 500 bytes is inserted, so that pointers are stored in the row, SQL Server uses only enough space to hold the pointers and not the entire 500 bytes.

If a table has multiple **text**, **ntext**, or **image** columns, and you attempt to insert multiple **text**, **ntext**, or **image** strings, SQL Server assigns space to the strings one at a time in sequence based on column ID. For example, assume you have a table containing four **text** columns and you have set the **text in row** option limit to 1000. You then insert a row where with a 900-byte string for each **text** column, and enough data for all of the other columns in the table so there is only 3,000 bytes of free space in the row to hold the **text** strings. The strings for the first three **text** columns are stored in the row, using 2,700 bytes of the 3,000 bytes available. The string for the fourth **text** column is not stored in the row, but the pointers from the root node are stored in the row.

Setting the text in row option on has several side effects in regards to processing **text**, **ntext**, or **image** data. For more information, see <u>Managing ntext</u>, text, and <u>image Data</u>.

See Also

sp_tableoption

SQL Server Architecture

Transaction Log Architecture

Every Microsoft® SQL Server[™] 2000 database has a transaction log that records all transactions and the database modifications made by each transaction. This record of transactions and their modifications supports three operations:

• Recovery of individual transactions.

If an application issues a ROLLBACK statement, or if SQL Server detects an error such as the loss of communication with a client, the log records are used to roll back the modifications made by an incomplete transaction.

• Recovery of all incomplete transactions when SQL Server is started.

If a server running SQL Server fails, the databases may be left in a state where some modifications were never written from the buffer cache to the data files, and there may be some modifications from incomplete transactions in the data files. When a copy of SQL Server is started, it runs a recovery of each database. Every modification recorded in the log which may not have been written to the data files is rolled forward. Every incomplete transaction found in the transaction log is then rolled back to ensure the integrity of the database is preserved.

• Rolling a restored database forward to the point of failure.

After the loss of a database, as is possible if a hard drive fails on a server that does not have RAID drives, you can restore the database to the point of failure. You first restore the last full or differential database backup, and then restore the sequence of transaction log backups to the point of failure. As you restore each log backup, SQL Server reapplies all the modifications recorded in the log to roll forward all the transactions. When the last log backup is restored, SQL Server then uses the log information to roll back all transactions that were not complete at that point.

The characteristics of the SQL Server 2000 transaction log are:

• The transaction log is not implemented as a table but as a separate file

or set of files in the database. The log cache is managed separately from the buffer cache for data pages, resulting in simple, fast, and robust code within the database engine.

- The format of log records and pages is not constrained to follow the format of data pages.
- The transaction log can be implemented on several files. The files can be defined to autogrow as required. This reduces the potential of running out of space in the transaction log, while at the same time reducing administrative overhead.
- The mechanism to truncate unused parts of the log is quick and has minimal effect on transaction throughput.

SQL Server Architecture

Write-Ahead Transaction Log

Microsoft[®] SQL Server[™] 2000, like many relational databases, uses a writeahead log. A write-ahead log ensures that no data modifications are written to disk before the associated log record.

SQL Server maintains a buffer cache into which it reads data pages when data must be retrieved. Data modifications are not made directly to disk, but are instead made to the copy of the page in the buffer cache. The modification is not written to disk until either the database is checkpointed, or the modifications must be written to disk so the buffer can be used to hold a new page. Writing a modified data page from the buffer cache to disk is called flushing the page. A page modified in the cache but not yet written to disk is called a dirty page.

At the time a modification is made to a page in the buffer, a log record is built in the log cache recording the modification. This log record must be written to disk before the associated dirty page is flushed from the buffer cache to disk. If the dirty page were flushed before the log record, it would create a modification on disk that could not be rolled back if the server failed before the log record were written to disk. SQL Server has logic that prevents a dirty page from being flushed before the associated log record. Because log records are always written ahead of the associated data pages, the log is called a write-ahead log.

See Also

Backup/Restore Architecture

Transactions Architecture

SQL Server Architecture

Transaction Log Logical Architecture

The Microsoft® SQL Server[™] 2000 transaction log operates logically as if it is a serial string of log records. Each log record is identified by a log sequence number (LSN). Each new log record is written to the logical end of the log with an LSN higher than the LSN of the record before it.

Log records for data modifications record either the logical operation performed or before and after images of the modified data. A before image is a copy of the data before the operation is performed; an after image is a copy of the data after the operation has been performed. The steps to recover an operation depend on the type of log record:

- Logical operation logged.
 - To roll the logical operation forward, it is performed again.
 - To roll the logical operation back, the reverse logical operation is performed.
- Before and after image logged.
 - To roll the operation forward, the after image is applied.
 - To roll the operation back, the before image is applied.

Many types of operations are recorded in the transaction log, including:

- The start and end of each transaction.
- Every data modification (insert, update, or delete). This includes changes to system tables made by system stored procedures or data definition language (DDL) statements.
- Every extent allocation or deallocation.

• The creation or dropping of a table or index.

Log records are stored in a serial sequence as they are created. Each log record is stamped with the ID of the transaction to which it belongs. For each transaction, all log records associated with the transaction are singly-linked in a chain using backward pointers that speed the rollback of the transaction.

Rollback statements are also logged. Each transaction reserves space on the transaction log to ensure enough log space exists to support a rollback if an error is encountered. This reserve space is freed when the transaction completes. The amount of space reserved depends on the operations performed in the transaction, but is generally equal to the amount of space used to log each operation.

Checkpoints and the Active Portion of the Log

Checkpoints minimize the portion of the log that must be processed during a full recovery of a database. During a full recovery, two types of actions must be performed:

- The log may contain records of modifications not flushed to disk before the system stopped. These modifications must be rolled forward.
- All the modifications associated with incomplete transactions (transactions for which there is no COMMIT or ROLLBACK log record) must be rolled back.

Checkpoints flush dirty data and log pages from the buffer cache of the current database, minimizing the number of modifications that have to be rolled forward during a recovery.

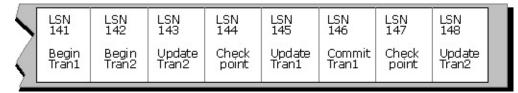
A SQL Server 2000 checkpoint performs these processes in the current database:

- Writes to the log file a record marking the start of the checkpoint.
- Stores information recorded for the checkpoint in a chain of checkpoint log records. The LSN of the start of this chain is written to the database boot page.
- One piece of information recorded in the checkpoint records is the LSN of the first log image that must be present for a successful database-wide rollback. This LSN is called the Minimum Recovery LSN (MinLSN) and is the minimum of:
 - The LSN of the start of the checkpoint.
 - The LSN of the start of the oldest active transaction.

- The LSN of the start of the oldest replication transaction that has not yet replicated to all subscribers.
- Another piece of information recorded in the checkpoint records is a list of all outstanding, active transactions.
- Deletes all log records before the new MinLSN, if the database is using the simple recovery model.
- Writes to disk all dirty log and data pages.
- Writes to the log file a record marking the end of the checkpoint.

The portion of the log file from the MinLSN to the last-written log record is called the active portion of the log. This is the portion of the log required to do a full recovery of the database. No part of the active log can ever be truncated. All log truncation must be done from the parts of the log before the MinLSN.

This is a simplified version of the end of a transaction log with two active transactions. Checkpoint records have been compacted to a single record.



LSN 148 is the last record in the transaction log. At the time the checkpoint recorded at LSN 147 was processed, Tran 1 had been committed and Tran 2 was the only active transaction. That makes the first log record for Tran 2 the oldest log record for a transaction active at the time of the last checkpoint. This makes LSN 142, the begin transaction record for Tran 2, the MinLSN.

Checkpoints occur:

- When a CHECKPOINT statement is executed. The current database for the connection is checkpointed.
- When ALTER DATABASE is used to change a database option.

ALTER DATABASE checkpoints the database when database options are changed.

- When an instance of SQL Server is stopped by:
 - Executing a SHUTDOWN statement.
 - Using the SQL Server Service Control Manager to stop the service running an instance of the database engine.

Either of these methods checkpoints each database in the instance of SQL Server.

• When an instance SQL Server periodically generates automatic checkpoints in each database to reduce the amount of time the instance would take to recover the database.

Automatic Checkpoints

SQL Server 2000 always generates automatic checkpoints. The interval between automatic checkpoints is based on the number of records in the log, not time. The time interval between automatic checkpoints can be highly variable. The time interval between automatic checkpoints is long if few modifications are made in the database. Automatic checkpoints occur frequently if a lot of data is modified.

The interval between automatic checkpoints is calculated from the **recovery interval** server configuration option. This option specifies the maximum time SQL Server should use to recover a database during a system restart. SQL Server estimates how many log records it can process in the **recovery interval** during a recovery operation. The interval between automatic checkpoints also depends on whether or not the database is using the simple recovery model.

• If the database is using either the full or bulk-logged recovery model, an automatic checkpoint is generated whenever the number of log records reaches the number SQL Server estimates it can process during the time specified in the **recovery interval** option.

- If the database is using the simple recovery model, an automatic checkpoint is generated whenever the number of log records reaches the lesser of these two values:
 - The log becomes 70 percent full.
 - The number of log records reaches the number SQL Server estimates it can process during the time specified in the **recovery interval** option.

Automatic checkpoints truncate the unused portion of the transaction log if the database is using the simple recovery model. The log is not truncated by automatic checkpoints if the database is using the full or bulk-logged recovery models. For more information, see <u>Truncating the Transaction Log</u>.

Long-Running Transactions

The active portion of the log must include every part of all uncommitted transactions. An application that starts a transaction and does not commit it or roll it back prevents SQL Server from advancing the MinLSN. This can cause two types of problems:

- If the system is shut down after the transaction has performed many uncommitted modifications, the recovery phase of the subsequent restart can take considerably longer than the amount of time specified in the **recovery interval** option.
- The log may grow very large because the log cannot be truncated past the MinLSN. This happens even if the database is using the simple recovery model, in which the transaction log is normally truncated on each automatic checkpoint.

Replication Transactions

The active portion of the log must also contain all transactions marked for replication, but that have not yet been replicated to a subscriber. If these transactions are not replicated in a timely manner, they can also prevent

truncation of the log.

See Also

Backup/Restore Architecture <u>CHECKPOINT</u> <u>Freeing and Writing Buffer Pages</u> <u>Transaction Recovery</u>

Truncating the Transaction Log

If log records were never deleted from the transaction log, the logical log would grow until it filled all the available space on the disks holding the physical log files. At some point in time, old log records no longer necessary for recovering or restoring a database must be deleted to make way for new log records. The process of deleting these log records to reduce the size of the logical log is called truncating the log.

The active portion of the transaction log can never be truncated. The active portion of the log is the part of the log needed to recover the database at any time, so must have the log images needed to roll back all incomplete transactions. It must always be present in the database in case the server fails because it will be required to recover the database when the server is restarted. The record at the start of the active portion of the log is identified by the minimum recovery log sequence number (MinLSN).

The recovery model chosen for a database determines how much of the transaction log in front of the active portion must be retained in the database. Although the log records in front of the MinLSN play no role in recovering active transactions, they are required to roll forward modifications when using log backups to restore a database to the point of failure. If you lose a database for some reason, you can recover the data by restoring the last database backup, and then restoring every log backup since the database backup. This means that the sequence of log backups must contain every log record that was written since the database backup. When you are maintaining a sequence of transaction log backups, no log record can be truncated until after it has been written to a log backup.

The log records before the MinLSN are only needed to maintain a sequence of transaction log backups.

 In the simple recovery model, a sequence of transaction logs is not being maintained. All log records before the MinLSN can be truncated at any time, except while a BACKUP statement is being processed. NO_LOG and TRUNCATE_ONLY are the only BACKUP LOG options that are valid for a database that is using the simple recovery model.

Note The **tempdb** database always uses the simple recovery model, it cannot be switched to another recovery model. Log truncation always occurs on a checkpoint in **tempdb**.

• In the full and bulk-logged recovery models, a sequence of transaction log backups is being maintained. The part of the logical log before the MinLSN cannot be truncated until those log records have been copied to a log backup.

Log truncation occurs at these points:

- At the completion of any BACKUP LOG statement.
- Every time a checkpoint is processed, provided the database is using the simple recovery model. This includes both explicit checkpoints resulting from a CHECKPOINT statement and implicit checkpoints generated by the system. The exception is that the log is not truncated if the checkpoint occurs when a BACKUP statement is still active. For more information about the interval between automatic checkpoints, see <u>Checkpoints and the Active Portion of the Log</u>.

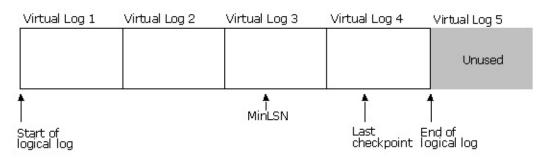
Transaction logs are divided internally into sections called virtual log files. Virtual log files are the unit of truncation. When a transaction log is truncated, all log records before the start of the virtual log file containing the MinLSN are deleted. For more information about virtual log files, see <u>Transaction Log</u> <u>Physical Architecture</u>.

The size of a transaction log is therefore controlled in one of these ways:

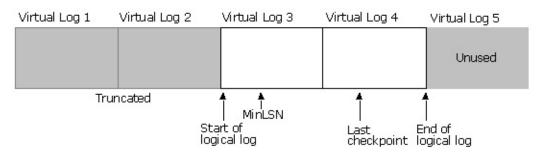
- When a log backup sequence is being maintained, schedule BACKUP LOG statements to occur at intervals that will keep the transaction log from growing past the desired size.
- When a log backup sequence is not maintained, specify the simple recovery model.

This illustration shows a transaction log that has four virtual logs. The log has

not been truncated after the database was created. The logical log starts at the beginning of the first virtual log and the part of virtual log 4 beyond the end of the logical file has never been used.



This illustration shows how the log looks after truncation. The rows before the start of the virtual log containing the MinLSN record have been truncated.



Truncation does not reduce the size of a physical log file, it reduces the size of the logical log file. For information on shrinking the size of a physical log file, see Shrinking the Transaction Log.

See Also

BACKUP

Setting Database Options

Transaction Log Backups

Truncate Method

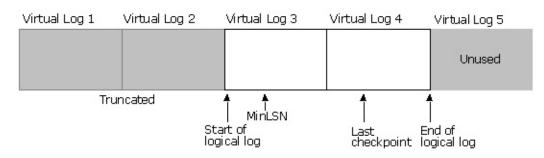
Transaction Log Physical Architecture

The transaction log in a database maps over one or more physical files. Conceptually, the log file is a serial string of log records. Physically, the sequence of log records must be stored efficiently in the set of physical files that implement the transaction log.

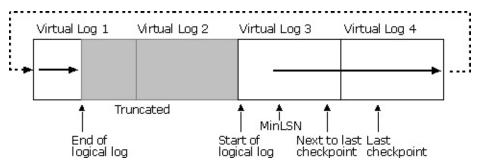
Microsoft® SQL Server[™] 2000 segments each physical log file internally into a number of virtual log files. Virtual log files have no fixed size, and there is no fixed number of virtual log files for a physical log file. SQL Server chooses the size of the virtual log files dynamically while creating or extending log files. SQL Server tries to maintain a small number of virtual files. The size of the virtual file name extension is based on the size of the existing log and the size of the new file increment. The size or number of virtual log files cannot be configured or set by administrators; it is determined dynamically by the SQL Server code.

The only time virtual log files affect system performance is if the log files are defined with small *size* and *growth_increment* values. If these log files grow to a large size through many small increments, they will have a lot of virtual log files, which can slow down recovery. It is recommended that log files be defined with a *size* value close to the final size needed, and also have a relatively large *growth_increment* value.

The transaction log is a wrap-around log file. For example, consider a database with one physical log file divided into four virtual log files. When the database is created, the logical log file begins at the start of the physical log file. New log records are added at the end of the logical log, which grows toward the end of the physical log. As truncation operations occur, the records in the virtual logs before the minimum recovery log sequence number (MinLSN) are deleted. The log in the example database would look like the one in the illustration.



When the end of the logical log reaches the end of the physical log file, the new log records wrap around to the start of the physical log file.



This cycle repeats endlessly, as long as the end of the logical log never reaches the beginning of the logical log. If the old log records are truncated often enough to always leave enough room for all the new log records created through the next checkpoint, the log never fills. If the end of the logical log does reach the start of the logical log, however, one of two things happens:

- If autogrow is enabled for the log and space is available on the disk, the file is extended by the amount specified in *growth_increment* and the new log records are added to the extension.
- If autogrow is not enabled, or the disk holding the log file has less free space than the amount specified in *growth_increment*, an 1105 error is generated.

If the log contains multiple physical log files, then the logical log will move through all of the physical log files before it wraps back to the start of the first physical log file.

See Also

Transaction Log Backups

Transaction Logs

Shrinking the Transaction Log

The size of the log files are physically reduced when:

- A DBCC SHRINKDATABASE statement is executed.
- A DBCC SHRINKFILE statement referencing a log file is executed.
- An autoshrink operation occurs.

Shrinking a log is dependent on first truncating the log. Log truncation does not reduce the size of a physical log file, it reduces the size of the logical log and marks as inactive the virtual logs that do not hold any part of the logical log. A log shrink operation removes enough inactive virtual logs to reduce the log file to the requested size.

The unit of size reduction is a virtual log. For example, if you have a 600 MB log file that has been divided into six 100 MB virtual logs, the size of the log file can only be reduced in 100 MB increments. The file size can be reduced to sizes such as 500 MB or 400 MB, but it cannot be reduced to sizes such as 433 MB or 525 MB.

Virtual logs that hold part of the logical log cannot be freed. If all the virtual logs in a log file hold parts of the logical log, the file cannot be shrink until a truncation marks one or more of the virtual logs at the end of the physical log as inactive.

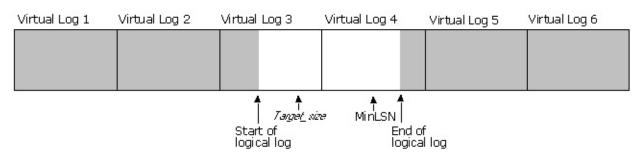
When any file is shrunk, the space freed must come from the end of the file. When a transaction log file is shrunk, enough virtual logs from the end of the file are freed to reduce the log to the size requested by the user. The *target_size* specified by the user is rounded to the next highest virtual log boundary. For example, if a user specifies a *target_size* of 325 MB for our sample 600 MB file with 100 MB virtual log files, the last two virtual log files are removed and the new file size is 400 MB.

In SQL Server 2000, a DBCC SHRINKDATABASE or DBCC SHRINKFILE

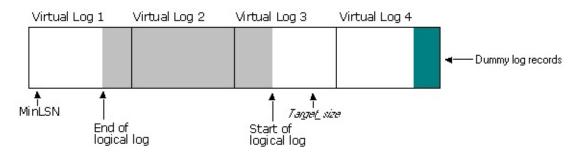
operation attempts to shrink the physical log file to the requested size (subject to rounding) immediately:

- If no part of the logical log is in the virtual logs beyond the *target_size* mark, the virtual logs after the *target_size* mark are freed and the successful DBCC statement completes with no messages.
- If part of the logical log is in the virtual logs beyond the *target_size* mark, SQL Server 2000 frees as much space as possible and issues an informational message. The message tells you what actions you need to perform to get the logical log out of the virtual logs at the end of the file. After you perform this action, you can then reissue the DBCC statement to free the remaining space.

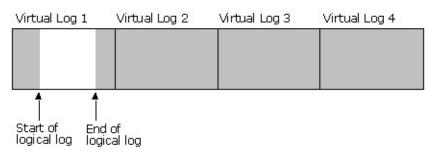
For example, assume that a 600 MB log file with six virtual logs has a logical log starting in virtual log 3 and ending in virtual log 4, when you execute a DBCC SHRINKFILE statement with a *target_size* of 275 MB:



Virtual logs 5 and 6 are freed immediately because they hold no portion of the logical log. To meet the specified *target_size*, however, virtual log 4 should also be freed, but cannot because it holds the end portion of the logical log. After freeing virtual logs 5 and 6, SQL Server 2000 fills the remaining part of virtual log 4 with dummy records. This forces the end of the log file to virtual log 1. In most systems, all transactions starting in virtual log 4 will be committed within seconds, meaning that all of the active portion of the log moves to virtual log 1, and the log file now looks like this:



The DBCC SHRINKFILE statement also issues an informational message that it could not free all the space requested, and indicate that you can execute a BACKUP LOG statement to make it possible to free the remaining space. Once the active portion of the log moves to virtual log 1, a BACKUP LOG statement will truncate the entire logical log that is in virtual log 4:



Because virtual log 4 no longer holds any portion of the logical log, if you now execute the same DBCC SHRINKFILE statement with a *target_size* of 275 MB, virtual log 4 will be freed and the size of the physical log file reduced to the size requested.

See Also

BACKUP

Setting Database Options

Space Allocation and Reuse

Transaction Log Backups

Truncating the Transaction Log

SQL Server Architecture

Relational Database Engine Architecture

The server is the component of Microsoft® SQL ServerTM 2000 that receives SQL statements from clients and performs all the actions necessary to complete the statements. This section discusses:

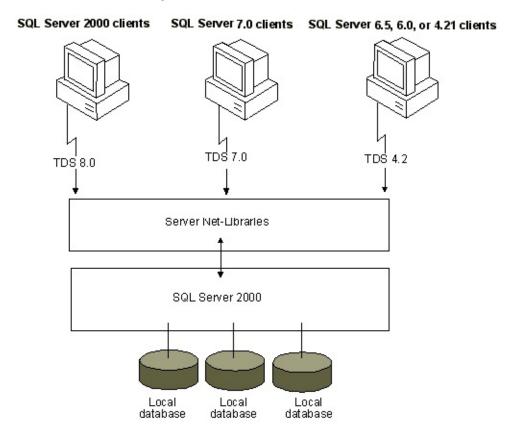
- An overview of the components that make up the server.
- How the server compiles each batch of SQL statements into an execution plan that tells the server how to process the statement.
- How the server manages Microsoft Windows® resources such as memory, threads, and tasks.
- How the server determines what part of a distributed query references a linked server and what request to transmit to the server to obtain the needed data.
- How the server transmits remote stored procedure calls to remote servers.
- How the server manages concurrency and transaction issues.
- How the server implements server cursors.
- The features that allow SQL Server to scale from small laptop computers to large servers that provide the primary data storage for large enterprises.
- How the SQL Mail component integrates SQL Server with e-mail servers to allow the server to send e-mail and pages when specified

events occur.

SQL Server Architecture

Relational Database Engine Architecture Overview

The server components of Microsoft® SQL Server 2000TM receive SQL statements from clients and process those SQL statements. This illustration shows the major components involved with processing an SQL statement received from a SQL Server client.



Tabular Data Stream

SQL statements are sent from clients by using an application-level protocol specific to SQL Server called Tabular Data Stream (TDS). SQL Server 2000 accepts the following versions of TDS:

• TDS 8.0 sent by clients running versions of the SQL Server client components from SQL Server 2000. TDS 8.0 clients support all the features of SQL Server 2000.

- TDS 7.0 sent by clients running versions of the SQL Server client components from SQL Server version 7.0. TDS 4.2 clients do not support features introduced in SQL Server 2000, and the server sometimes has to adjust the data it sends back to the clients using TDS 7.0. For example, TDS 7.0 clients do not support the sql_variant data type, so SQL Server 2000 must convert any sql_variant data to Unicode.
- TDS 4.2 sent by clients running SQL Server client components from SQL Server 6.5, 6.0, and 4.21a. TDS 4.2 clients do not support features introduced in either SQL Server 2000 or SQL Server 7.0, and the server sometimes has to adjust the data it sends back to clients using TDS 4.2. For example, TDS 4.2 clients do not support Unicode data types, so SQL Server 2000 must convert any Unicode data to character data before sending it to the client, with possible loss of extended characters. TDS 4.2 clients also do not support char, varchar, binary, or varbinary values longer than 255 bytes, so SQL Server 2000 must truncate any values longer than 255 before sending them to the client.

Server Net-Libraries

TDS packets are built by the Microsoft OLE DB Provider for SQL Server, the SQL Server ODBC driver, or the DB-Library DLL. The TDS packets are then passed to a SQL Server client Net-Library, which encapsulates the TDS packets into network protocol packets. On the server, the network protocol packets are received by a server Net-Library that extracts the TDS packet and passes it to the relational database server.

This process is reversed when results are returned to the client.

Each server can be listening simultaneously on several network protocols and will be running one server Net-Library for each protocol on which it is listening.

Relational Database Engine

The database server processes all requests passed to it from the server Net-Libraries. It compiles all the SQL statements into execution plans, and then uses the plans to access the requested data and build the result set returned to the client.

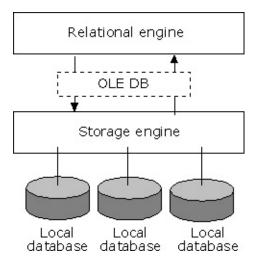
See Also

Relational Database Components

SQL Server Architecture

Database Engine Components

The relational database server of Microsoft® SQL Server[™] 2000 has two main parts: the relational engine and the storage engine. One of the most important architectural changes made in SQL Server version 7.0 was to strictly separate the relational and storage engine components within the server and to have them use the OLE DB API to communicate with each other.



The processing for a SELECT statement that references only tables in local databases can be summarized as:

1. The relational engine compiles the SELECT statement into an optimized execution plan. The execution plan defines a series of operations against basic rowsets from the individual tables or indexes referenced in the SELECT statement.

A rowset is the OLE DB term for a result set. The rowsets requested by the relational engine return the amount of data needed from a table or index to perform one of the operations used to build the SELECT result set. For example, this SELECT statement requires a table scan if it references a table with no indexes:

SELECT * FROM ScanTable

The relational engine implements the table scan by requesting one rowset containing all the rows from **ScanTable**.

This SELECT statement only needs information available in an index:

SELECT DISTINCT LastName FROM Northwind.dbo.Employees

The relational engine implements the index scan by requesting one rowset containing the leaf rows from the index built on the **LastName** column.

This SELECT statement needs information from two indexes:

```
SELECT CompanyName, OrderID, ShippedDate
FROM Northwind.dbo.Customers AS Cst
JOIN Northwind.dbo.Orders AS Ord
ON (Cst.CustomerID = Ord.CustomerID)
```

The relational engine requests two rowsets, one for the clustered index on **Customers** and the other on one of the nonclustered indexes in **Orders**.

- 2. The relational engine uses the OLE DB API to request that the storage engine open the rowsets.
- 3. As the relational engine works through the steps of the execution plan and needs data, it uses OLE DB to fetch the individual rows from the rowsets it requested the storage engine to open. The storage engine transfers the data from the data buffers to the relational engine.
- 4. The relational engine combines the data from the storage engine rowsets into the final result set transmitted back to the user.

Relational Engine

The main responsibilities of the relational engine are:

• Parsing the SQL statements.

The parser scans an SQL statement and breaks it down into the logical units, such as keywords, parameters, operators, and identifiers. The parser also breaks down the overall SQL statement into a series of smaller logical operations.

• Optimizing the execution plans.

Typically, there are many ways that the server could use data from the source tables to build the result set. The query optimizer determines what these various series of steps are, estimates the cost of each series (primarily in terms of file I/O), and chooses the series of steps that has the lowest cost. It then combines the specific steps with the query tree to produce an optimized execution plan.

• Executing the series of logical operations defined in the execution plan.

After the query optimizer has defined the logical operations required to complete a statement, the relational engine steps through these operations in the sequence specified in the optimized execution plan.

• Processing Data Definition Language (DDL) and other statements.

These statements are not the typical SELECT, INSERT, UPDATE, or DELETE statements; these statements have special processing needs. Examples are the SET statements to set connection options, and the CREATE statements to create objects in a database.

• Formatting results.

The relational engine formats the results returned to the client. The results are formatted as either a traditional, tabular result set or as an XML document. The results are then encapsulated in one or more TDS packets and returned to the application.

SQL Server Architecture

Storage Engine

The main responsibilities of the storage engine include:

- Managing the files on which the database is stored and managing the use of space in the files.
- Building and reading the physical pages used to store data.
- Managing the data buffers and all I/O to the physical files.
- Controlling concurrency. Managing transactions and using locking to control concurrent user access to rows in the database.
- Logging and recovery.
- Implementing utility functions such as the BACKUP, RESTORE, and DBCC statements and bulk copy.

SQL Server Language Support

Microsoft[®] SQL Server[™] 2000 is installed with 33 natural languages defined on the server. The definitions for each language establish how date data is interpreted:

- The formats in which dates are presented:
 - dmy (day, month, year)
 - mdy (month, day, year)
 - ymd (year, month, day)
- Short and long names for each month.
- Names for each day.
- Which day is considered the first day of the week.

These language definitions are stored in **master.dbo.syslanguages** and a language identifier (ID) identifies each language.

Each instance of SQL Server uses a default language for all connections to the server. For more information about configuring the setting, see <u>default language</u> <u>Option</u>.

Most connections use the default language configured for the server, but each connection can individually set a SQL Server language to be used for the connection:

- Microsoft ActiveX[®] Data Object and OLE DB applications can include the Language keyword in a provider string specified when they connect.
- OLE DB applications can also set the provider-specific property

SSPROP_INIT_CURRENTLANGUAGE before connecting.

- Open Database Connectivity (ODBC) applications can include the LANGUAGE keyword in a connection string specified on SQLDriverConnect. ODBC applications can also specify the language setting in a SQL Server ODBC data source definition.
- DB-Library applications can use **dblogin** to allocate a loginrec, and then use the DBSETNATLANG macro to specify a language setting before calling **dbopen** to connect.
- Any application can use the SET LANGUAGE statement to specify the SQL Server language.

SQL Server supports having multiple, language-specific copies of the error messages stored in **master.dbo.sysmessages**. All instances of SQL Server contain the set of English messages. SQL Server is localized, or translated, into French, German, Spanish, and Japanese versions. Installations of localized versions of SQL Server install the translated set of messages in addition to the English set. When SQL Server sends a message to a connection, it uses the localized message if the language ID of the connection matches one of the language IDs present in **sysmessages**. If there is no message in **sysmessages** with the same language ID, the English version of the message is sent.

Query Processor Architecture

SQL statements are the only commands sent from applications to Microsoft® SQL Server[™] 2000. All of the work done by an instance of SQL Server is the result of accepting, interpreting, and executing SQL statements. The processes by which SQL statements are executed by SQL Server include:

- Single SQL statement processing.
- Batch processing.
- Stored procedure and trigger execution.
- Execution plan caching and reuse.
- Parallel query processing.

Single SQL Statement Processing

Processing a single SQL statement is the most basic way that Microsoft® SQL Server[™] 2000 executes SQL statements. The steps used to process a single SELECT statement that references only local base tables (no views or remote tables) illustrates the basic process.

Optimizing SELECT Statements

A SELECT statement is nonprocedural; it does not state the exact steps the database server should use to retrieve the requested data. This means the database server must analyze the statement to determine the most efficient way to extract the requested data. This is called optimizing the SELECT statement, and the component that does this is called the query optimizer.

A SELECT statement defines only:

- The format of the result set. This is specified mostly in the select list, although other clauses such as ORDER BY and GROUP BY also affect the final form of the result set.
- The tables containing the source data. This is specified in the FROM clause.
- How the tables are logically related for the purposes of the SELECT statement. This is defined in the join specifications.
- What conditions the rows in the source tables must satisfy to qualify for the SELECT statement. These are specified in the WHERE and HAVING clauses.

A query execution plan is a definition of:

• The sequence in which the source tables are accessed.

Typically, there are many sequences in which the database server can

access the base tables to build the result set. For example, if the SELECT statement references three tables, the database server could first access **TableA**, use the data from **TableA** to extract matching rows from **TableB**, and then use the data from **TableB** to extract data from **TableC**. The other sequences in which the database server could access the tables are: TableC, TableB, TableA; or **TableB**, **TableA**, **TableA**, **TableC**; or **TableB**, **TableC**, **TableA**; or **TableA**, **TableB**.

• The methods used to extract data from each table.

Usually, there are different methods for accessing the data in each table. If only a few rows with specific key values are needed, the database server can use an index. If all the rows in the table are needed, the database server can ignore the indexes and do a table scan. If all the rows in a table are needed, but there is an index whose key columns are in an ORDER BY, performing an index scan instead of a table scan may save a separate sort of the result set. If a table is very small, table scans may be the most efficient method for almost all access to the table.

The process of choosing one execution plan out of several possible plans is called optimization. The query optimizer is one of the most important components of a SQL database system. While some overhead is used by the query optimizer to analyze the query and choose a plan, this overhead is saved several-fold when the query optimizer picks an efficient execution plan. For example, two construction companies can be given identical blueprints for a house. If one company spends a few days at the start to plan how they will build the house, and the other company starts building without planning, the company that takes the time to plan their project will most likely finish first.

The SQL Server query optimizer is a cost-based optimizer. Each possible execution plan has an associated cost in terms of the amount of computing resources used. The query optimizer must analyze the possible plans and choose the one with the lowest estimated cost. Some complex SELECT statements have thousands of possible execution plans. In these cases, the query optimizer does not analyze all possible combinations. Instead, it uses complex algorithms to find an execution plan that has a cost reasonably close to the theoretical minimum.

The SQL Server query optimizer does not choose only the execution plan with

the lowest resource cost; it chooses the plan that returns results to the user with a reasonable cost in resources and returns the results the fastest. For example, processing a query in parallel typically uses more resources than processing it serially, but completes the query faster. The SQL Server optimizer will use a parallel execution plan to return results if the load on the server will not be adversely affected.

Query optimizer relies on distribution statistics when estimating the resource costs of different methods of extracting information from a table or index. Distribution statistics are kept for columns and indexes. They indicate the selectivity of the values in a particular index or column. For example, in a table representing cars, many cars have the same manufacturer, but each car has a unique vehicle identification number (VIN). An index on the VIN is more selective than an index on the manufacturer. If the index statistics are not current, the query optimizer may not make the best choice for the current state of the table. For more information about keeping index statistics current, see <u>Statistical Information</u>.

The query optimizer is important because it enables the database server adjust dynamically to changing conditions in the database without requiring input from a programmer or database administrator. This enables programmers to focus on describing the final result of the query. They can trust that the query optimizer will always build an efficient execution plan for the state of the database each time the statement is run.

Processing a SELECT Statement

The basic steps that SQL Server uses to process a single SELECT statement are:

- 1. The parser scans the SELECT statement and breaks it into logical units such as keywords, expressions, operators, and identifiers.
- 2. A query tree, sometimes called a sequence tree, is built describing the logical steps needed to transform the source data into the format needed by the result set.
- 3. The query optimizer analyzes all the ways the source tables can be accessed and selects the series of steps that returns the results fastest

while consuming fewer resources. The query tree is updated to record this exact series of steps, and the final, optimized version of the query tree is called the execution plan.

- 4. The relational engine begins executing the execution plan. As steps that need data from the base tables are processed, the relational engine uses OLE DB to request that the storage engine pass up data from the rowsets requested from the relational engine.
- 5. The relational engine processes the data returned from the storage engine into the format defined for the result set, and returns the result set to the client.

Processing Other Statements

The basic steps described for processing a SELECT statement apply to other SQL statements such as INSERT, UPDATE, and DELETE. UPDATE and DELETE statements both have to target the set of rows to be modified or deleted; the process of identifying these rows is the same process used to identify the source rows that contribute to the result set of a SELECT statement. The UPDATE and INSERT statements may both contain embedded SELECT statements that provide the data values to be updated or inserted.

Even Data Definition Language (DDL) statements such as CREATE PROCEDURE or ALTER TABLE are ultimately resolved to a series of relational operations on the system catalog tables and sometimes (such as ALTER TABLE ADD COLUMN) against the data tables.

View Resolution

The Microsoft® SQL Server[™] 2000 query processor treats indexed and nonindexed views differently:

- Indexed views are stored in the database in the same format as a table. The query processor treats indexed views the same way it treats base tables.
- Only the source of a nonindexed view is stored. The query optimizer incorporates the logic from the view source into the execution plan it builds for the SQL statement that references the nonindexed view.

The logic used by the SQL Server query optimizer to decide when to use an indexed view is similar to the logic used to decide when to use an index on a table. If the data in the indexed view covers the SQL statement, and the query optimizer determines that an index on the view is the low-cost access path, the query optimizer will choose the index regardless of whether the view is referenced in the WHERE clause. For more information, see <u>Resolving Indexes on Views</u>.

When an SQL statement references a nonindexed view, the parser and query optimizer analyze the source of both the SQL statement and the view, and resolve them into a single execution plan. There is not one plan for the SQL statement and a separate plan for the view.

For example, consider the following view:

```
USE Northwind
GO
CREATE VIEW EmployeeName AS
SELECT EmployeeID, LastName, FirstName
FROM Northwind.dbo.Employees
GO
```

Given this view, both of these SQL statements perform the same operations on the base tables and produce the same results:

```
/* SELECT referencing the EmployeeName view. */
SELECT LastName AS EmployeeLastName,
    OrderID, OrderDate
FROM Northwind.dbo.Orders AS Ord
    JOIN Northwind.dbo.EmployeeName as EmpN
    ON (Ord.EmployeeID = EmpN.EmployeeID)
WHERE OrderDate > '31 May, 1996'
```

```
/* SELECT referencing the Employees table directly. */
SELECT LastName AS EmployeeLastName,
    OrderID, OrderDate
FROM Northwind.dbo.Orders AS Ord
    JOIN Northwind.dbo.Employees as Emp
    ON (Ord.EmployeeID = Emp.EmployeeID)
WHERE OrderDate > '31 May, 1996'
```

The SQL Query Analyzer showplan feature shows that the relational engine builds the same execution plan for both of these SELECT statements.

Resolving Indexes on Views

The Microsoft® SQL Server[™] 2000 query optimizer determines whether a given query will benefit from using any indexes defined in the database. This includes both indexed views and indexes on base tables. The SQL Server query optimizer uses an indexed view when these conditions are met:

- These session options are set to ON:
 - ANSI_NULLS
 - ANSI_PADDING
 - ANSI_WARNINGS
 - ARITHABORT
 - CONCAT_NULL_YIELDS_NULL
 - QUOTED_IDENTIFIERS
- The NUMERIC_ROUNDABORT session option is set to OFF.
- The query optimizer finds a match between the view index columns and elements in the query, such as:
 - Search condition predicates in the WHERE clause.
 - Join operations.
 - Aggregate functions.

• The estimated cost for using the index has the lowest cost of any access mechanisms considered by the query optimizer.

Other than the requirements for the SET options, these are the same rules the query optimizer uses to determine if a table index covers a query. Nothing has to be specified in the query to make use of an indexed view.

A query does not have to explicitly reference an indexed view in the FROM clause for the query optimizer to use the indexed view. If the query contains references to columns in the base tables that are also present in the indexed view, and the query optimizer estimates that using the indexed view provides the lowest cost access mechanism, the query optimizer chooses the indexed view, similar to the way it chooses base table indexes when they are not directly referenced in a query. The query optimizer may choose the view when it contains columns that are not referenced by the query, as long as the view offers the lowest cost option for covering one or more of the columns specified in the query.

You can prevent view indexes from being used for a query by using the EXPAND VIEWS option. You can use the NOEXPAND view hint to force the use of an index for an indexed view specified in the FROM clause of a query. It is recommended, however, to let the query optimizer dynamically determine the best access methods to use for each individual query. Limit your use of EXPAND and NOEXPAND to specific cases where testing has shown they improve performance significantly.

The EXPAND VIEWS option specifies that the query optimizer not use any view indexes for the entire query.

- The query optimizer does not use any indexed views unless the view is specified in the FROM clause. The query optimizer ignores all view indexes when estimating the low-cost method for covering columns referenced in the query.
- The query optimizer treats an indexed view referenced in the FROM clause as a standard view. The query optimizer incorporates the logic of the view into the query execution plan and dynamically builds the result set from the base tables. The query optimizer ignores indexes defined on the view.

When NOEXPAND is specified for a view, the query optimizer considers the use of any indexes defined on the view. NOEXPAND specified with the optional INDEX() clause forces the query optimizer to use the specified indexes. NOEXPAND can be specified only for an indexed view and cannot be specified for a view not indexed.

Resolving Distributed Partitioned Views

The Microsoft® SQL Server 2000[™] query processor is enhanced to optimize the performance of distributed partitioned views. The most important aspect of distributed partitioned view performance is minimizing the amount of data transferred between member servers.

SQL Server 2000 builds intelligent, dynamic plans that make efficient use of distributed queries to access data from remote member tables:

- The query processor first uses OLE DB to retrieve the CHECK constraint definitions from each member table. This allows the query processor to map the distribution of key values across the member tables.
- The query processor compares the key ranges specified in an SQL statement WHERE clause to the map showing how the rows are distributed in the member tables. The query processor then builds a query execution plan that uses distributed queries to retrieve only those remote rows needed to complete the SQL statement. The execution plan is also built in such a way that any access to remote member tables, for either data or meta data, are delayed until the information is required.

For example, consider a system where a customers table is partitioned across **Server1 (CustomerID** from 1 through 3299999), **Server2 (CustomerID** from 3300000 through 6599999), and **Server3 (CustomerID** from 6600000 through 999999).

Consider the execution plan built for this query executed on **Server1**:

SELECT * FROM CompanyData.dbo.Customers WHERE CustomerID BETWEEN 3200000 AND 3400000

The execution plan for this query extracts the rows with **CustomerID** key values from 3200000 through 3299999 from the local member table, and issues a

distributed query to retrieve the rows with key values from 3300000 through 3400000 from **Server2**.

The SQL Server 2000 query processor can also build dynamic logic into query execution plans for SQL statements where the key values are not known when the plan must be built. For example, consider this stored procedure:

CREATE PROCEDURE GetCustomer @CustomerIDParameter INT AS SELECT * FROM CompanyData.dbo.Customers WHERE CustomerID = @CustomerIDParameter

SQL Server 2000 cannot predict what key value will be supplied by the **@CustomerIDParameter** parameter each time the procedure is executed. Because the key value cannot be predicted, the query processor also cannot predict which member table will have to be accessed. To handle this case, SQL Server builds an execution plan that has conditional logic, called dynamic filters, to control which member table is accessed based on the input parameter value. Assuming the **GetCustomer** stored procedure was executed on **Server1**, the execution plan logic can be represented as:

IF @CustomerIDParameter BETWEEN 1 and 3299999 Retrieve row from local table CustomerData.dbo.Customer_33 ELSEIF @CustomerIDParameter BETWEEN 3300000 and 6599999 Retrieve row from linked table Server2.CustomerData.dbo.Customer ELSEIF @CustomerIDParameter BETWEEN 6600000 and 9999999 Retrieve row from linked table Server3.CustomerData.dbo.Customer

SQL Server 2000 sometimes builds these types of dynamic execution plans even for queries that are not parameterized. The optimizer may auto-parameterize a query so that the execution plan can be reused. If the optimizer autoparameterizes a query referencing a partitioned view, then the optimizer can no longer assume the required rows will come from a specified base table, and it will have to use dynamic filters in the execution plan. For more information, see <u>Auto-Parameterization</u>.

Worktables

The relational engine may need to build a worktable to perform a logical operation specified in an SQL statement. Worktables are typically generated for certain GROUP BY, ORDER BY, or UNION queries. For example, if an ORDER BY clause references columns not covered by any indexes, the relational engine may need to generate a worktable to sort the result set into the order requested.

Worktables are built in **tempdb** and are dropped automatically at the end of the statement.

Batch Processing

A batch is a collection of one or more SQL statements sent in one unit by the client. Each batch is compiled into a single execution plan. If the batch contains multiple SQL statements, all of the optimized steps needed to perform all the statements are built into a single execution plan.

There are several ways to specify a batch:

- All the SQL statements sent in a single execution unit from an application comprise a single batch and generate a single execution plan. For more information about how an application specifies a batch, see <u>Batches</u>.
- All the statements in a stored procedure or trigger comprise a single batch. Each stored procedure or trigger is compiled into a single execution plan.
- The string executed by an EXECUTE statement is a batch compiled into a single execution plan.
- The string executed by an **sp_executesql** system stored procedure is a batch compiled into a single execution plan.

When a batch sent from an application contains an EXECUTE statement, the execution plan for the executed string or stored procedure is executed separately from the execution plan containing the EXECUTE statement. The execution plan generated for the string executed by an **sp_executesql** stored procedure also remains separate from the execution plan for the batch containing the **sp_executesql** call. If a statement in a batch invokes a trigger, the trigger execution plan executes separately from the original batch.

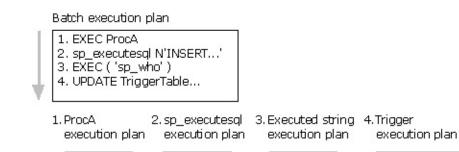
For example, a batch that contains these four statements uses five execution plans:

• An EXECUTE statement executing a stored procedure.

- An **sp_executesql** call executing a string.
- An EXECUTE statement executing a string.
- An UPDATE statement referencing a table that has an update trigger.

sp_who...

RAISERROR



INSERT ...

SELECT ...

Stored Procedure and Trigger Execution

Microsoft® SQL Server[™] 2000 stores only the source for stored procedures and triggers. When a stored procedure or trigger is first executed, the source is compiled into an execution plan. If the stored procedure or trigger is again executed before the execution plan is aged from memory, the relational engine detects the existing plan and reuses it. If the plan has aged out of memory, a new plan is built. This process is similar to the process SQL Server 2000 follows for all SQL statements. The main performance advantage that stored procedures and triggers have in SQL Server 2000 is that their SQL statements are always the same; therefore, the relational engine matches them with any existing execution plans.

Stored procedures had a more pronounced performance advantage over other SQL statements in earlier versions of SQL Server. Earlier versions of SQL Server did not attempt to reuse execution plans for batches that were not stored procedures or triggers. The only way to reuse execution plans was to encode the SQL statements in stored procedures.

The execution plan for stored procedures and triggers is executed separately from the execution plan for the batch calling the stored procedure or firing the trigger. This allows for greater reuse of the stored procedure and trigger execution plans. For more information, see <u>Batch Processing</u>.

Execution Plan Caching and Reuse

Microsoft® SQL Server[™] 2000 has a pool of memory used to store both execution plans and data buffers. The percentage of the pool allocated to either execution plans or data buffers fluctuates dynamically depending on the state of the system. The part of the memory pool used to store execution plans is called the procedure cache.

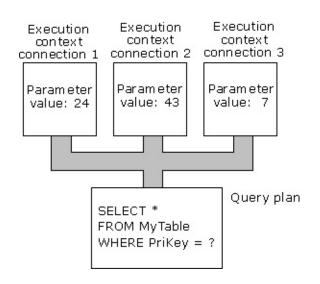
SQL Server 2000 execution plans have the following main components:

• Query plan

The bulk of the execution plan is a reentrant, read-only data structure used by any number of users. This is called the query plan. No user context is stored in the query plan. There are never more than one or two copies of the query plan in memory: one copy for all serial executions and another for all parallel executions. The parallel copy covers all parallel executions, regardless of their degree of parallelism.

• Execution context

Each user currently executing the query has a data structure that holds the data specific to their execution, such as parameter values. This data structure is called the execution context. The execution context data structures are reused. If a user executes a query and one of the structures is not in use, it is reinitialized with the context for the new user.



When any SQL statement is executed in SQL Server 2000, the relational engine first looks through the procedure cache to verify that an existing execution plan for the same SQL statement exists. SQL Server 2000 reuses any existing plan it finds, saving the overhead of recompiling the SQL statement. If no existing execution plan exists, SQL Server 2000 generates a new execution plan for the query.

SQL Server 2000 has an efficient algorithm to find any existing execution plans for any given SQL statement. In most systems, the minimal resources used by this scan are less than the resources saved by being able to reuse existing plans instead of compiling every SQL statement.

The algorithms to match new SQL statements to existing, unused execution plans in the cache require that all object references be fully qualified. For example, the first of these SELECT statements is not matched with an existing plan, and the second is matched:

SELECT * FROM Employees

SELECT * FROM Northwind.dbo.Employees

There is a higher probability that individual execution plans will be reused in an instance of SQL Server 2000 than in SQL Server version 6.5 and earlier.

Aging Execution Plans

After an execution plan is generated, it stays in the procedure cache. SQL Server 2000 ages old, unused plans out of the cache only when space is needed. Each query plan and execution context has an associated cost factor that indicates how expensive the structure is to compile. These data structures also have an age field. Each time the object is referenced by a connection, the age field is incremented by the compilation cost factor. For example, if a query plan has a cost factor of 8 and is referenced twice, its age becomes 16. The lazywriter process periodically scans the list of objects in the procedure cache. The lazywriter decrements the age field of each object by 1 on each scan. The age of our sample query plan is decremented to 0 after 16 scans of the procedure cache, unless another user references the plan. The lazywriter process deallocates an object if these conditions are met:

- The memory manager requires memory and all available memory is currently in use.
- The age field for the object is 0.
- The object is not currently referenced by a connection.

Because the age field is incremented each time an object is referenced, frequently referenced objects do not have their age fields decremented to 0 and are not aged from the cache. Objects infrequently referenced are soon eligible for deallocation, but are not actually deallocated unless memory is required for other objects.

Recompiling Execution Plans

Certain changes in a database can cause an execution plan to be either inefficient or invalid, given the new state of the database. SQL Server detects the changes that invalidate an execution plan, and marks the plan as invalid. A new plan must then be recompiled for the next connection that executes the query. The conditions that cause a plan to be invalidated include:

- Any structural changes made to a table or view referenced by the query (ALTER TABLE and ALTER VIEW).
- New distribution statistics generated either explicitly from a statement such as UPDATE STATISTICS or automatically.
- Dropping an index used by the execution plan.
- An explicit call to **sp_recompile**.
- Large numbers of changes to keys (generated by INSERT or DELETE statements from other users that modify a table referenced by the query).

• For tables with triggers, if the number of rows in the **inserted** or **deleted** tables grows significantly.

Parameters and Execution Plan Reuse

The use of parameters, including parameter markers in ADO, OLE DB, and ODBC applications, can increase the reuse of execution plans.

The only difference between the following two SELECT statements are the values compared in the WHERE clause:

SELECT * FROM Northwind.dbo.Products WHERE CategoryID = 1

SELECT * FROM Northwind.dbo.Products WHERE CategoryID = 4

The only difference between the execution plans for these queries is the value stored for the comparison against the **CategoryID** column. While the goal is for SQL Server 2000 to always recognize that the statements generate essentially the same plan and reuse the plans, SQL Server sometimes does not detect this in complex SQL statements.

Separating constants from the SQL statement by using parameters helps the relational engine recognize duplicate plans. You can use parameters in the following ways:

 In Transact-SQL, use sp_executesql: DECLARE @MyIntParm INT SET @MyIntParm = 1 EXEC sp_executesql N'SELECT * FROM Northwind.dbo.Products WHERE Categ N'@Parm INT', @MyIntParm

This method is recommended for Transact-SQL scripts, stored procedures, or triggers that generate SQL statements dynamically. For more information, see <u>Building Statements at Run Time</u>.

• ADO, OLE DB, and ODBC use parameter markers. Parameter markers are question marks (?) that replace a constant in an SQL statement and

are bound to a program variable. For example, in an ODBC application:

- Use **SQLBindParameter** to bind an integer variable to the first parameter marker in an SQL statement.
- Place the integer value in the variable.
- Execute the statement, specifying the parameter marker (?): SQLExecDirect(hstmt, "SELECT * FROM Northwind.dbo.Products WHERE SQL_NTS);

The Microsoft OLE DB Provider for SQL Server and the SQL Server ODBC driver that are included with SQL Server 2000 use **sp_executesql** to send statements to SQL Server 2000 when parameter markers are used in applications.

See Also

sp_executesql

Using Parameters

Command Parameters

Using Statement Parameters

Auto-Parameterization

In Microsoft® SQL Server[™] 2000, using parameters or parameter markers in Transact-SQL statements increases the ability of the relational engine to match new SQL statements with existing, unused execution plans. If an SQL statement is executed without parameters, SQL Server 2000 parameterizes the statement internally to increase the possibility of matching it against an existing execution plan.

Consider this statement:

SELECT * FROM Northwind.dbo.Products WHERE CategoryID = 1

The value 1 at the end of the statement can be specified as a parameter. The relational engine builds the execution plan for this batch as if a parameter had been specified in place of the value 1. Because of this auto-parameterization, SQL Server 2000 recognizes that the following two statements generate essentially the same execution plan and reuses the first plan for the second statement:

SELECT * FROM Northwind.dbo.Products WHERE CategoryID = 1

SELECT * FROM Northwind.dbo.Products WHERE CategoryID = 4

When processing complex SQL statements, the relational engine may have difficulty determining which expressions can be auto-parameterized. To increase the ability of the relational engine to match complex SQL statements to existing, unused execution plans, explicitly specify the parameters using either **sp_executesql** or parameter markers. For more information, see <u>Parameters and Execution Plan Reuse</u>.

Preparing SQL Statements

The Microsoft® SQL Server[™] 2000 relational engine introduces full support for preparing SQL statements before they are executed. If an application needs to execute an SQL statement several times, using the database API it can:

- Prepare the statement once. This compiles the SQL statement into an execution plan.
- Execute the precompiled execution plan each time it needs to execute the statement. This saves recompiling the SQL statement on each execution after the first.

Preparing and executing statements is controlled by API functions and methods. It is not a part of the Transact-SQL language. The prepare/execute model of executing SQL statements is supported by the Microsoft OLE DB Provider for SQL Server and the SQL Server ODBC driver. On a prepare request, either the provider or the driver sends the statement to SQL Server with a request to prepare the statement. SQL Server compiles an execution plan and returns a handle to that plan to the provider or driver. On an execute request, either the provider or the driver sends the server a request to execute the plan associated with the handle.

Prepared statements cannot be used to create temporary objects on SQL Server 2000 or SQL Server version 7.0. Prepared statements cannot reference system stored procedures that create temporary objects, such as temporary tables. These procedures must be executed directly.

Excess use of the prepare/execute model can degrade performance. If a statement is executed only once, a direct execution requires only one network round trip to the server. Preparing and executing an SQL statement executed only one time requires an extra network round-trip; one trip to prepare the statement and one trip to execute it.

Preparing a statement is more effective if parameter markers are used. For

example, assume an application is asked occasionally to retrieve product information from the **Northwind** sample database. There are two methods for how the application can do this.

In the first method, the application could execute a separate query for each product requested:

SELECT * FROM Northwind.dbo.Products WHERE ProductID = 63

An alternative would be for the application to:

- Prepare a statement containing a parameter marker (?): SELECT * FROM Northwind.dbo.Products WHERE ProductID = ?
- 2. Bind a program variable to the parameter marker.
- 3. Each time product information is needed, fill the bound variable with the key value and execute the statement.

The second method is more efficient when the statement is executed more than three times.

In SQL Server 2000, the prepare/execute model has little performance advantage over direct execution because of the way SQL Server 2000 reuses execution plans. SQL Server 2000 has efficient algorithms for matching current SQL statements with execution plans generated for prior executions of the same SQL statement. If an application executes an SQL statement with parameter markers multiple times, SQL Server 2000 will reuse the execution plan from the first execution for the second and subsequent executions (unless the plan ages from the procedure cache). The prepare/execute model still offers these benefits:

- Finding an execution plan by an identifying handle is more efficient than the algorithms used to match an SQL statement to existing execution plans.
- The application can control when the execution plan is created and

when reused.

• The prepare/execute model is portable to other databases, including earlier versions of SQL Server.

Prepare and Execute in Earlier Versions of SQL Server

SQL Server version 6.5 and earlier did not support the prepare/execute model directly. The SQL Server ODBC driver, however, supported the prepare/execute model by using stored procedures:

- When an application requested that an SQL statement be prepared, the ODBC driver would wrap the SQL statement in a CREATE PROCEDURE statement and send it to SQL Server.
- On an execute request, the ODBC driver would request that SQL Server execute the generated stored procedure.

In SQL Server 6.5 and SQL Server 6.0, the generated stored procedures were temporary stored procedures stored in **tempdb**. SQL Server version 4.21a and earlier did not support temporary stored procedures, so the driver generated regular stored procedures stored in the current database. The Microsoft OLE DB Provider for SQL Server and the SQL Server ODBC driver included with SQL Server 2000 follows these behaviors when connected to SQL Server version 6.5, SQL Server version 6.0, and SQL Server version 4.21a.

See Also

Execution Plan Caching and Reuse Parameters and Execution Plan Reuse Executing Prepared Statements Preparing Commands Prepared Execution

Parallel Query Processing

Microsoft® SQL Server[™] 2000 provides parallel queries to optimize query execution for computers having more than one microprocessor. By allowing SQL Server to perform a query in parallel by using several operating system threads, SQL Server completes complex queries with large amounts of data quickly and efficiently.

During query optimization, SQL Server looks for queries that might benefit from parallel execution. For these queries, SQL Server inserts exchange operators into the query execution plan to prepare the query for parallel execution. An exchange operator is an operator in a query execution plan that provides process management, data redistribution, and flow control. After exchange operators are inserted, the result is a parallel query execution plan. A parallel query execution plan can use more than one thread, whereas a serial execution plan, used by a nonparallel query, uses only a single thread for its execution. The actual number of threads used by a parallel query is determined at query plan execution initialization and is called the degree of parallelism.

Degree of Parallelism

Microsoft® SQL Server[™] 2000 detects the best degree of parallelism for each instance of a parallel query execution automatically by considering:

1. Is SQL Server running on a computer with more than one microprocessor or CPU, such as a symmetric multiprocessing computer (SMP)?

Only computers with more than one CPU can use parallel queries.

2. What is the number of concurrent users active on the SQL Server installation at this moment?

SQL Server monitors CPU usage and adjusts the degree of parallelism at the query startup time. Lower degrees of parallelism are chosen if CPU usage is high.

3. Is there sufficient memory available for parallel query execution?

Each query requires a certain amount of memory to execute. Executing a parallel query requires more memory than a nonparallel query. The amount of memory required for executing a parallel query increases with the degree of parallelism. If the memory requirement of the parallel plan for a given degree of parallelism cannot be satisfied, SQL Server decreases the degree of parallelism automatically or completely abandons the parallel plan for the query in the given workload context and executes the serial plan.

4. What is the type of query executed?

Queries heavily consuming CPU cycles are the best candidates for a parallel query. For example, joins of large tables, substantial aggregations, and sorting of large result sets are good candidates. Simple queries, often found in transaction processing applications, find the additional coordination required to execute a query in parallel outweigh the potential performance boost. To distinguish between queries that benefit from parallelism and those that do not benefit, SQL Server compares the estimated cost of executing the query with the **cost threshold for parallelism** value. Although not recommended, users can change the default value of 5 using **sp_configure**.

5. Is there a sufficient amount of rows processed in the given stream?

If the query optimizer determines the number of rows in a stream is too low, it does not introduce exchange operators to distribute the stream. Consequently, the operators in this stream are executed serially. Executing the operators in a serial plan avoids scenarios when the startup, distribution, and coordination cost exceeds the gains achieved by parallel operator execution.

The INSERT, UPDATE, and DELETE operators are executed serially; however, the WHERE clause of either an UPDATE or DELETE, or SELECT portion of an INSERT statement may be executed in parallel. The actual data changes are then serially applied to the database.

Static and keyset cursors can be populated by parallel execution plans. However, the behavior of dynamic cursors can be provided only by serial execution. The query optimizer always generates a serial execution plan for a query that is part of a dynamic cursor.

At execution time, SQL Server determines if the current system workload and configuration information allow for parallel query execution. If parallel query execution is warranted, SQL Server determines the optimal number of threads and spreads the execution of the parallel query across those threads. When a query starts executing on multiple threads for parallel execution, the query uses the same number of threads until completion. SQL Server reexamines the optimal number of thread decisions each time a query execution plan is retrieved from the procedure cache. For example, one execution of a query can result in use of a serial plan, a later execution of the same query can result in a parallel plan using three threads, and a third execution can result in a parallel plan using four threads.

Use SQL Profiler to monitor the degree of parallelism for individual statements. Use the Degree Of Parallelism event class in the Performance event category. For more information, see <u>Performance Event Category</u>.

The showplan output for every parallel query will have at least one of these

logical operators:

- Distribute Streams
- Gather Streams
- Repartition Streams

See Also

Setting Configuration Options <u>sp_configure</u> <u>System Stored Procedures</u>

Parallel Query Example

The following query counts the number of orders placed in a given quarter starting on April 1, 2000 in which at least one line item of the order was received by the customer later than the committed date. This query lists the count of such orders grouped by each order priority and sorted in ascending priority order.

This example uses theoretical table and column names.

```
SELECT o_orderpriority, COUNT(*) AS Order_Count
FROM orders
WHERE o_orderdate >= '2000/04/01'
AND o_orderdate < DATEADD (mm, 3, '2000/04/01')
AND EXISTS
    (
        SELECT *
        FROM lineitem
        WHERE l_orderkey = o_orderkey
        AND l_commitdate < l_receiptdate
    )
    GROUP BY o_orderpriority
    ORDER BY o_orderpriority</pre>
```

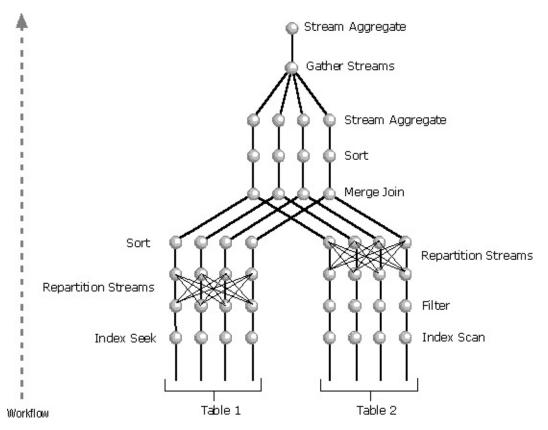
Assume the following indexes are defined on the **lineitem** and **orders** tables:

```
CREATE INDEX l_order_dates_idx
ON lineitem
(l_orderkey, l_receiptdate, l_commitdate, l_shipdate)
```

```
CREATE UNIQUE INDEX o_datkeyopr_idx
ON ORDERS
(o_orderdate, o_orderkey, o_custkey, o_orderpriority)
```

Here is one possible parallel plan generated for the query shown earlier:

|--Stream Aggregate(GROUP BY:([ORDERS].[o_orderpriority]) DEFINE:([Expr1005]=COUNT(*))) --Parallelism(Gather Streams, ORDER BY: ([ORDERS].[o_orderpriority] ASC)) --Stream Aggregate(GROUP BY: ([ORDERS].[o_orderpriority]) DEFINE:([Expr1005]=Count(*))) |--Sort(ORDER BY:([ORDERS].[o_orderpriority] ASC)) --Merge Join(Left Semi Join, MERGE: ([ORDERS].[o_orderkey])= ([LINEITEM].[l_orderkey]), RESIDUAL:([ORDERS].[o orderkey]= [LINEITEM].[l_orderkey])) |--Sort(ORDER BY:([ORDERS].[o_orderkey] ASC)) |--Parallelism(Repartition Streams, **PARTITION COLUMNS:** ([ORDERS].[o_orderkey])) |--Index Seek(OBJECT: ([tpcd1G].[dbo].[ORDERS].[O_DATKEYOPR_IDX]), SEEK:([ORDERS].[o_orderdate] >= Apr 1 2000 12:00AM AND [ORDERS].[o_orderdate] < Jul 1 2000 12:00AM) ORDERED) |--Parallelism(Repartition Streams, **PARTITION COLUMNS:** ([LINEITEM].[l_orderkey]), ORDER BY:([LINEITEM].[l_orderkey] ASC)) |--Filter(WHERE: ([LINEITEM].[l_commitdate]< [LINEITEM].[l_receiptdate])) |--Index Scan(OBJECT: ([tpcd1G].[dbo].[LINEITEM].[L ORDER DATES IDX]), ORE



The illustration shows a query optimizer plan executed with a degree of parallelism equal to 4 and involving a two-table join.

The parallel plan contains three Parallelism operators. Both the Index Seek operator of the **o_datkey_ptr** index and the Index Scan operator of the **l_order_dates_idx** index are performed in parallel, producing several exclusive streams. This can be determined from the nearest Parallelism operators above the Index Scan and Index Seek operators, respectively. They are both repartitioning the type of exchange; they are merely reshuffling data among the streams producing the same number of streams on their output as they have on input. This number of streams is equal to the degree of parallelism.

The Parallelism operator above the **l_order_dates_idx** Index Scan operator is repartitioning its input streams using the value of L_ORDERKEY as a key so the same values of L_ORDERKEY end up in the same output stream. At the same time, output streams maintain the order on the L_ORDERKEY column to meet the input requirement of the Merge Join operator.

The Parallelism operator above the Index Seek operator is repartitioning its input streams using the value of O_ORDERKEY. Because its input is not sorted on the O_ORDERKEY column values and this is the join column in the Merge Join

operator, the Sort operator between the Parallelism and Merge Join operators ensure the input is sorted for the Merge Join operator on the join columns. The Sort operator, like the Merge Join operator, is performed in parallel.

The topmost Parallelism operator gathers results from several streams into a single stream. Partial aggregations performed by the Stream Aggregate operator below the Parallelism operator are then accumulated into a single SUM value for each different value of the O_ORDERPRIORITY in the Stream Aggregate operator above the Parallelism operator.

See Also

Logical and Physical Operators

Parallel Operations Creating Indexes

The query plans built for the creation of indexes allow parallel, multi-threaded index create operations on computers with multiple microprocessors.

Microsoft® SQL Server[™] 2000 uses the same algorithms to determine the degree of parallelism (the total number of separate threads to run) for create index operations as it does for other Transact-SQL statements. The only difference is that the CREATE INDEX, CREATE TABLE, or ALTER TABLE statements that create indexes do not support the MAXDOP query hint. The maximum degree of parallelism for an index creation is subject to the **max degree of parallelism** server configuration option, but you cannot set a different MAXDOP value for individual index creation operations.

When SQL Server 2000 builds a create index query plan, the number of parallel operations is set to the lowest value of:

- The number of microprocessors, or CPUs in the computer.
- The number specified in the **max degree of parallelism** server configuration option.
- The number of CPUs not already over a threshold of work performed for SQL Server threads.

For example, on a computer with eight CPUs, but where **max degree of parallelism** is set to 6, no more than six parallel threads are generated for an index creation. If five of the CPUs in the computer exceed the threshold of SQL Server work when an index creation execution plan is built, the execution plan specifies only three parallel threads.

The main phases of parallel index creation include:

• A coordinating thread quickly and randomly scans the table to estimate the distribution of the index keys. The coordinating thread establishes the key boundaries that will create a number of key ranges equal to the

degree of parallel operations, where each key range is estimated to cover similar numbers of rows. For example, if there are 4 million rows in the table, and the degree of parallelism is 4, the coordinating thread will determine the key values that delimit 4 sets of rows with 1 million rows in each set.

- The coordinating thread dispatches a number of threads equal to the degree of parallel operations, and waits for these threads to complete their work. Each thread scans the base table using a filter that retrieves only rows with key values within the range assigned to the thread. Each thread builds an index structure for the rows in its key range. For more information about how an index is built, see <u>tempdb and Index</u> <u>Creation</u>.
- After all the parallel threads have completed, the coordinating thread connects the index subunits into a single index.

Individual CREATE TABLE or ALTER TABLE statements can have multiple constraints that require the creation of an index. These multiple index creation operations are performed in series, although each individual index creation operation may be a parallel operation on a computer with multiple CPUs.

See Also

tempdb and Index Creation

Memory Architecture

Microsoft® SQL Server[™] 2000 dynamically acquires and frees memory as needed. It is typically not necessary for an administrator to specify how much memory should be allocated to SQL Server, although the option still exists and is required in some environments. When running multiple instances of SQL Server on a computer, each instance can dynamically acquire and free memory to adjust for changes in the workload of the instance.

SQL Server 2000 Enterprise Edition introduces support for using Microsoft Windows® 2000 Address Windowing Extensions (AWE) to address approximately 8GB of memory for instances running on Windows 2000 Advanced Server, and approximately 64GB for instances running on Windows 2000 Data Center. Each instance using this extended memory, however, must statically allocate the memory it needs.

Virtual Memory and the Database Engine

Virtual memory is a method of extending the available physical memory on a computer. In a virtual memory system, the operating system creates a pagefile, or swapfile, and divides memory into units called pages. Recently referenced pages are located in physical memory, or RAM. If a page of memory is not referenced for a while, it is written to the pagefile. This is called swapping or paging out memory. If that piece of memory is later referenced by an application, the operating system reads the memory page back from the pagefile into physical memory, also called swapping or paging in memory. The total amount of memory available to applications is the amount of physical memory in the computer plus the size of the pagefile. If a computer has 256 MB of RAM and a 256 MB pagefile, the total memory available to applications is 512 MB. Operating systems such as Microsoft Windows NT®, Windows 2000, Windows 95, and Windows 98 support virtual memory.

One of the primary design goals of all database software is to minimize disk I/O because disk reads and writes are among the most resource-intensive operations. SQL Server builds a buffer cache in memory to hold pages read from the database. Much of the code in SQL Server is dedicated to minimizing the number of physical reads and writes between the disk and the buffer cache. The

larger the buffer cache is, the less I/O SQL Server has to do to the database files. However, if the buffer cache causes SQL Server memory requirements to exceed the available physical memory on the server, the operating system starts swapping memory to and from the pagefile. All that has happened is that the physical I/O to the database files has been traded for physical I/O to the swap file.

Having a lot of physical I/O to the database files is an inherent factor of database software. By default, SQL Server tries to reach a balance between two goals:

- Minimizing or eliminating pagefile I/O to concentrate I/O resources for reads and writes of the database files.
- Minimizing physical I/O to the database files by maximizing the size of the buffer cache.

By default, the SQL Server 2000 editions dynamically manage the size of the address space for each instance. There are differences in the way Windows NT, Windows 2000, Windows 95, and Windows 98 report virtual memory usage to applications. Because of this, SQL Server 2000 uses different algorithms to manage memory on these operating systems.

SQL Server 2000 Enterprise Edition does not default to dynamic memory management if you are using Windows 2000 AWE to support large address spaces.

Dynamically Managing Memory on Windows NT and Windows 2000

When running on Microsoft® Windows NT® or Windows® 2000, the default memory management behavior of the SQL Server database engine is not to acquire a specific amount of memory, but to acquire as much memory as it can without generating excess paging I/O. The database engine does this by acquiring as much memory as is available, while leaving enough memory free to prevent the operating system from swapping memory.

When an instance of SQL Server starts, it typically acquires 8 to 12 MB of memory to complete the initialization process. After the instance has finished initializing, it acquires no more memory until users connect to it and start generating a workload. The instance then keeps acquiring memory as required to support the workload. As more users connect and run queries, SQL Server acquires the additional memory required to support the demand. The instance will keep acquiring memory until it reaches its memory allocation target, it will not free any memory until it reaches the lower limit of the target.

To acquire as much memory as possible without generating excess paging I/O, each instance of SQL Server sets a target of acquiring memory until free physical memory on the computer is in the range of 4 MB to 10 MB. This range was chosen because testing has shown that Windows NT and Windows 2000 have minimal memory swapping until the memory allocations equal the available physical memory minus 4 MB. An instance of SQL Server that is processing a heavy workload keeps the free physical memory at the lower end (4 MB) of the range; an instance that is processing a light workload keeps the free memory at the higher end of the range (10 MB).

An instance of SQL Server will vary its target as the workload changes. As more users connect and generate more work, the instance will tend to acquire more memory to keep the available free memory down at the 4 MB limit. As the workload lightens, the instance will adjust its target towards 10 MB of free space, and will free memory to the operating system. Keeping the amount of free space between 10 MB and 4 MB keeps Windows NT or Windows 2000 from paging excessively, while at the same time allowing SQL Server to have the largest buffer cache possible that will not cause extra swapping.

The target memory setting for an instance is related to the demand for pages in the database buffer pool relative to the size of the available pool. At any point in time, the overall demand for buffer pages is determined by the number of data pages required to satisfy all of the currently executing queries. If the demand for data pages is large relative to the number of pages in the buffer cache, then each page currently in the buffer is likely to be replaced by a new page in a relatively short time. This is measured by the **page life expectancy** performance counter of the **Buffer Manager** object. Having a high demand against a relatively small buffer generates a short life expectancy, the net effect is that I/O is increased because pages tend to be overwritten before they can be referenced by multiple logical reads. The database engine can alleviate this by acquiring more memory to increase the size of the buffer cache. The database engine will target free memory at the high end of the target (10 MB) when the page life expectancy is long, and at the low end of the target range (4 MB) when the page life expectancy is short.

As other applications are started on a computer running an instance of SQL Server, they consume memory and the amount of free physical memory drops below the SQL Server target. The instance of SQL Server then frees enough memory from its address space to raise the amount of free memory back to the SQL Server target. If another application is stopped and more memory becomes available, the instance of SQL Server increases the size of its memory allocation. SQL Server can free and acquire several megabytes of memory each second, allowing it to quickly adjust to memory allocation changes.

Effects of min and max server memory

The **min server memory** and **max server memory** configuration options establish upper and lower limits to the amount of memory used by the SQL Server database engine. The database engine does not immediately acquire the amount of memory specified in **min server memory**. The database engine starts with only the memory required to initialize. As the database engine workload increases, it keeps acquiring the memory required to support the workload. The database engine will not free any of the acquired memory until it reaches the amount specified in **min server memory**. Once **min server memory** is reached, the database engine then uses the standard algorithm (keeping the operating system's free memory within 4 MB to 10 MB) to acquire and free memory as needed. The only difference is that the database engine never drops its memory allocation below the level specified in **min server memory**.

The amount of memory acquired by the database engine is entirely dependent on the workload placed on the instance. A SQL Server instance that is not processing many requests may never reach **min server memory**.

If the same value is specified for both **min server memory** and **max server memory**, then once the memory allocated to the database engine reaches that value, the database engine stops dynamically freeing and acquiring memory.

If an instance of SQL Server is running on a computer where other applications are frequently stopped or started, the allocation and deallocation of memory by the instance of SQL Server may slow the startup times of other applications. Also, if SQL Server is one of several server applications running on a single computer, the system administrators may need to control the amount of memory allocated to SQL Server. In these cases, you can use the **min server memory** and **max server memory** options to control how much memory SQL Server can use. For more information, see <u>Server Memory Options</u>.

Dynamically Managing Memory Between Multiple Instances

When multiple instances of SQL Server are running on the same computer, each instance independently uses the standard dynamic memory management algorithm. There is no need for the instances to communicate with each other to cooperatively manage memory. When all but 4 MB to 10 MB of the memory on a computer is allocated, the amount of memory allocated to each specific instance of the database engine is driven by the relative workload of each instance. The instances with higher workloads acquire more memory, while instances processing lighter workloads acquire less memory. Regardless of the number of instances of SQL Server on a computer, the algorithm ensures:

- The overall amount of allocated memory remains under the level that would generate Windows NT® or Windows® 2000 page I/Os.
- The computer memory is efficiently distributed between the instances of SQL Server based on their relative workloads.
- The memory allocations are dynamic and can immediately adjust to changes in the workloads of individual instances of SQL Server.

The interactions can be illustrated on a computer running two instances, but the same principles apply when several instances are running on the same computer. Consider a computer with 512MB of physical memory running two instances named **Instance1** and **Instance2**.

When both instances are first started, they typically acquire 8 MB to 12 MB of memory. As users connect to the instances, each instance acquires enough memory to satisfy its current workload.

Once the amount of memory reaches the point where only 4 MB to 10 MB is free, the instances begin competing with each other for memory. Assume that **Instance1** has a long page life expectancy and a free memory target of 10 MB, **Instance2** has a short page life expectancy with a free memory target of 4 MB.

Assume 506 MB of memory have been allocated, leaving only 6 MB free. Because 6 MB free memory is below the 10 MB target of **Instance1**, **Instance1** begins freeing memory. **Instance2** keeps acquiring memory because the amount of free memory is over its target of 4 MB. It does not matter how much memory either instance actually has. What is important is that the current buffer pool of **Instance2** is small relative to the demand for its data pages, while the buffer pool of **Instance1** is large relative to the demand for its data pages. So long as this is true, **Instance1** will have a free memory target of 10 MB and **Instance2** will have a free memory target of 4 MB, driving **Instance1** to free memory that is taken up by **Instance2**.

As **Instance1** frees memory, it reduces the size of its buffer cache. Eventually, **Instance1** reaches a point where the reduced size of the buffer cache starts decreasing the page life expectancy of the instance. As this happens, **Instance1** starts lowering its free memory target from 10 MB. At the same time, **Instance2** is using the memory it has acquired from **Instance1** to increase the size of the **Instance2** buffer cache. This increases the page life expectancy of **Instance2**, and so **Instance2** begins raising its free memory target from 4 MB. At some point, **Instance1** will have transferred enough memory to **Instance2** that both instances have the same free memory target. As soon as the amount of free memory reaches the level that is now the target of both instances, **Instance1** stops freeing memory, **Instance2** stops acquiring memory, and the system reaches a state of equilibrium.

The state of equilibrium lasts only as long as the relative workload of both instances remains constant. As soon as the workload on one or the other of the instances changes, either increases or decreases, the instance will change its free memory target. The instance with the higher free memory target will then start freeing memory and the instance with the lower free memory target will start acquiring memory until a new equilibrium is reached.

The same mechanism operates with more than two instances on a computer. All of the instances will keep freeing or acquiring memory until all of them reach the same free memory target. Once the amount free memory on the computer reaches the common target, the instances are in equilibrium.

Dynamically Managing Memory on Windows 95 and Windows 98

When running on Microsoft® Windows® 95 and Windows 98, each instance of Microsoft SQL Server[™] 2000 uses a demand-driven algorithm for allocating memory. As more Transact-SQL statements are processed and demand for cached database pages rises, the instance of SQL Server requests more virtual memory. When the demands on the instance of SQL Server go down, such as when fewer Transact-SQL statements are being processed, the instance frees memory back to the operating system.

Using AWE Memory on Windows 2000

Microsoft® SQL Server[™] 2000 Enterprise Edition uses the Microsoft Windows® 2000 Address Windowing Extensions (AWE) API to support very large amounts of physical memory. SQL Server 2000 Enterprise Edition can access amounts of memory approaching 8 GB on Windows 2000 Advanced Server and approaching 64 GB on Windows 2000 Data Center.

Standard 32-bit addresses can map a maximum of 4 GB of memory. The standard address spaces of 32-bit Microsoft Windows NT® 4.0 and Windows 2000 processes are therefore limited to 4-GB. By default, 2 GB is reserved for the operating system, and 2 GB is made available to the application. If you specify a /3GB switch in the Boot.ini file of Windows NT Enterprise Edition or Windows 2000 Advanced Server, the operating system reserves only 1 GB of the address space, and the application can access up to 3 GB. For more information about the /3GB switch, see Windows NT Enterprise Edition or Windows 2000 Advanced Server Help.

AWE is a set of extensions to the memory management functions of the Microsoft Win32® API that allow applications to address more memory than the 4 GB that is available through standard 32-bit addressing. AWE lets applications acquire physical memory as nonpaged memory, and then dynamically map views of the nonpaged memory to the 32-bit address space. Although the 32-bit address space is limited to 4 GB, the nonpaged memory can be much larger. This enables memory-intensive applications, such as large database systems, address more memory than can be supported in a 32-bit address space. For more information about AWE, see the MSDN® page at Microsoft Web site.

Enabling AWE Memory

You must specifically enable the use of AWE memory by an instance of SQL Server 2000 Enterprise Edition by using the **sp_configure** option **awe enabled**.

• When **awe enabled** is set to **0**, AWE memory is not used, and the instance defaults to using dynamic memory in standard 32-bit virtual address spaces.

• When **awe enabled** is set to **1**, AWE memory is used, and the instance can access up to 8 GB of physical memory on Windows 2000 Advanced Server and 64 GB on Windows 2000 Data Center.

When an instance of SQL Server 2000 Enterprise Edition is run with **awe enabled** set to **1**:

- The instance does not dynamically manage the size of the address space.
- The instance holds all memory acquired at startup until it is shut down.
- The memory pages for the instance come from the Windows nonpageable pool, meaning that none of the memory of the instance can be swapped out.

You must carefully manage the memory used by an instance of SQL Server when **awe enabled** is set to **1**. If the instance acquires most of the available physical memory as nonpaged memory, other applications or system processes may not be able to get the memory they need to run. Use the **max server memory** configuration setting to control how much memory is used by each instance of SQL Server that uses AWE memory. For more information, see Managing AWE Memory on Windows 2000.

SQL Server Memory Pool

In Microsoft® Windows NT®, Windows® 2000, Windows 95, and Windows 98, the total amount of virtual memory available to an application forms the set of valid memory addresses for the application. The total virtual memory allocation for an application is known as its address space.

Each instance of Microsoft SQL Server[™] 2000 has an address space with two main components, each of which has several subcomponents:

• Executable code

The number and size of the executable files and dynamic link libraries (DLLs) used by an instance of SQL Server varies over time. In addition to the executable files and DLLs used by Open Data Services, the SQL Server engine, and server Net-Libraries, the following components load in their own DLLs, and these DLLs can allocate memory themselves:

- Distributed queries can load an OLE DB Provider DLL on the server running the instance of SQL Server.
- Extended stored procedures are implemented as DLLs that are loaded into the address space of the instance of SQL Server.
- The OLE Automation system stored procedures are used to create instances of OLE Automation objects. Each class of OLE Automation object loads its own code into the address space of the instance of SQL Server.
- Memory pool

The memory pool is the main unit of memory for an instance of SQL Server. Almost all data structures that use memory in an instance of SQL Server are allocated in the memory pool. The main types of objects allocated in the memory pool are:

• System-level data structures

These are data structures that hold data global to the instance, such as database descriptors and the lock table.

• Buffer cache

This is the pool of buffer pages into which data pages are read.

• Procedure cache

This is a pool of pages containing the execution plans for all Transact-SQL statements currently executing in the instance.

• Log caches

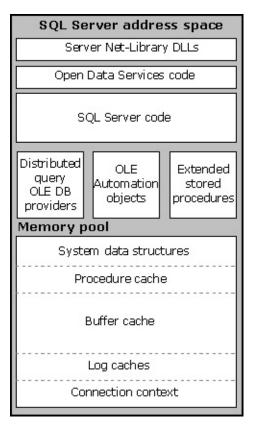
Each log has a cache of buffer pages used to read and write log pages. The log caches are managed separately from the buffer cache to reduce the synchronization between log and data buffers. This results in fast, robust code.

• Connection context

Each connection has a set of data structures that record the current state of the connection. These data structures hold items such as parameter values for queries and stored procedures, cursor positioning information, and tables currently being referenced.

• Stack space

Windows allocates stack space for each thread started by SQL Server. The default size for the stack space is 512K.



The size of the memory pool used by an instance of SQL Server 2000 can be very dynamic, especially on computers running other applications or other instances of SQL Server. By default, SQL Server seeks to keep the amount of virtual memory allocations on the computer at 4 to 10 MB less than the physical memory. The only way an instance of SQL Server can do this is by varying the size of its address space. The only variable component in the address space for an instance of SQL Server is the memory pool. The other variable components in the SQL Server address space, such as the number and size of OLE DB providers, OLE Automation objects, and extended stored procedures, are all controlled by application requests. If an application executes a distributed query, SQL Server must load the associated OLE DB provider. This means that if a SQL Server component is loaded, or another application starts up, the only mechanism an instance of SQL Server can use to release the memory needed by the new component or application is to reduce the size of the memory pool. SQL Server administrators can set limits on how much the size of the memory pool varies through the **min server memory** and **max server memory** configuration options.

The regions within the memory pool are also highly dynamic. The SQL Server code constantly adjusts the amounts of the memory pool assigned to the various

areas to optimize performance. Within the memory pool, the areas used to store connection context and system data structures are controlled by user requests. As new connections are made, SQL Server has to allocate data structures to store their context. As new databases are defined, SQL Server has to allocate data structures to define the attributes of the database. As tables and views are referenced, SQL Server allocates data structures describing their structure and attributes. This leaves the buffer cache, procedure cache, and log caches as the memory units whose size is controlled by SQL Server. SQL Server adjusts the sizes of these areas dynamically as needed to optimize performance.

For more information about the sizes of the various system and connection context data structures, see <u>Memory Used by SQL Server Objects Specifications</u>.

SQL Server 2000 is very efficient in the way it stores the context and state information for each connection, typically using less than 24 KB for each connection.

Thread and Task Architecture

Complex applications may have many tasks that could be performed at the same time. Threads are an operating system feature that lets application logic be separated into several concurrent execution paths.

When an operating system executes an instance of an application, it creates a unit called a process to manage the instance. The process has a thread of execution, which is the series of programming instructions performed by the application code. In a simple application with a single set of instructions that can be performed serially, there is just one execution path, or thread, through the application. More complex applications may have several tasks that could be performed in tandem, not serially. The application could do this by starting separate processes for each task, but starting a process is a resource-intensive operation. Instead, an application can start separate threads, which are relatively less resource-intensive. Each thread can be scheduled for execution independently from the other threads associated with a process. Each thread stores the data unique to it in an area of memory called a stack.

Threads allow complex applications to make more effective use of a CPU even on computers with a single CPU. With one CPU, only one thread can execute at a time. If one thread executes a long running operation that does not use the CPU, such as a disk read or write, another one of the threads can execute until the first operation completes. Being able to execute threads while other threads are waiting for an operation to complete allows the application to maximize its use of the CPU. This is especially true for multiuser, disk I/O intensive applications such as a database server.

Computers with multiple microprocessors, or CPUs can execute one thread per CPU at the same time. If a computer has eight CPUs, it can concurrently execute eight threads.

Windows NT Fibers

The Microsoft® Windows® operating system code that manages threads is in the kernel. Switching threads requires switches between the user mode of the application code and the kernel mode of the thread manager, which is a

moderately expensive operation. Microsoft Windows NT® fibers are a subcomponent of threads managed by code running in user mode. Switching fibers does not require the user-mode to kernel-mode transition needed to switch threads. The scheduling of fibers is managed by the application, and Windows manages the scheduling of threads. Each thread can have multiple fibers.

SQL Server Task Scheduling

Each instance of Microsoft® SQL ServerTM 2000 is a separate operating system process. Each instance has to handle potentially thousands of concurrent requests from users. Instances of SQL Server 2000 use Microsoft Windows® threads, and sometimes fibers, to manage these concurrent tasks efficiently. Each instance of SQL Server 2000 always runs several threads for system processes: one or more threads for each server Net-Library, a network thread to handle login requests, and a signal thread for communicating with the service control manager.

Each instance of SQL Server has an internal layer that implements an environment similar to an operating system for scheduling and synchronizing concurrent tasks without having to call the Windows kernel. This internal layer can schedule fibers as effectively as it works with threads. Each instance of SQL Server maintains a pool of either threads or fibers for user connections. The maximum size of this pool is controlled by the **max worker threads** server configuration option.

The server configuration **lightweight pooling** option controls whether an instance of SQL Server 2000 uses threads or fibers. The default is for **lightweight pooling** to be set to 0, in which case the instance of SQL Server schedules a thread per concurrent user command, up to the value of **max worker threads**. If **lightweight pooling** is set to 1, SQL Server then uses fibers instead of threads. This is called running in fiber mode. In fiber mode, an instance of SQL Server allocates one thread per CPU, and then allocates a fiber per concurrent user command, up to the **max worker threads** value. An instance of SQL Server uses the same algorithms to schedule and synchronize tasks when using either threads or fibers. SQL Server 2000 Personal Edition and SQL Server 2000 Desktop Engine do not support fibers.

A SQL batch is a set of one or more Transact-SQL statements sent from a client to an instance of SQL Server for execution as a unit. As an instance of SQL Server receives batches from clients, it associates each batch with an available free thread or fiber from the worker pool. If there are no free threads or fibers and the **max worker threads** value has not been reached, the instance of SQL Server allocates a new thread or fiber for the new batch. If there are no free threads or fibers available and the **max worker threads** value has already been reached, the instance blocks the new batch until a thread is freed. After a thread or fiber is associated with a batch, it remains associated with the batch until the last of the result sets generated by the batch has been returned to the client. At that time, the thread or fiber is freed and can be scheduled to the next available batch.

While threads and fibers are lightweight in their use of resources, they still consume resources. In systems with hundreds or thousands of user connections, having one thread or fiber per connection could consume enough resources to reduce the efficiency of SQL Server. Allocating a thread or fiber for each user connection is also not necessary because most connections actually spend much of their time waiting for batches to be received from the client. The pool of worker threads for an instance of SQL Server only needs to be large enough to service the number of user connections that are actively executing batches at the same time in that instance. Leaving **max worker threads** at its default value of 255 lets the instance of SQL Server effectively map user connections over a number of threads or fibers that do not consume too many resources.

Allocating Threads to a CPU

By default, each instance of Microsoft® SQL Server[™] 2000 starts each thread, and then Microsoft Windows NT® or Windows® 2000 assigns each thread to a specific CPU. Windows NT or Windows 2000 distribute threads from instances of SQL Server evenly among the microprocessors, or CPUs on a computer. At times, Windows NT or Windows 2000 can also move a thread from one CPU with heavy usage to another CPU.

SQL Server administrators can use the **affinity mask** configuration option to exclude one or more CPUs from being eligible to run threads from a specific instance of SQL Server. The **affinity mask** value specifies a bit pattern that indicates the CPUs that are used to run threads from that instance of SQL Server. For example, the **affinity mask** value 13 represents the bit pattern 1101. On a computer with four CPUs, this indicates threads from that instance of SQL Server can be scheduled on CPUs 0, 2, and 3, but not on CPU 1. If **affinity mask** is specified, the instance of SQL Server allocates threads evenly among the CPUs that have not been masked off. Another effect of **affinity mask** is that Windows NT and Windows 2000 do not move threads from one CPU to another. **affinity mask** is rarely used; most systems get optimal performance by letting Windows NT or Windows 2000 schedule the threads among the available CPUs.

Using the lightweight pooling Option

The overhead of switching thread contexts is not very large. Most instances of Microsoft® SQL Server[™] will not see any performance difference between setting the **lightweight pooling** option to 0 or 1. The only instances likely to benefit from **lightweight pooling** are those running on a computer characterized as:

- A large multi-CPU server.
- All of the CPUs are running near maximum capacity.
- There is a high level of context switching.

These systems may see a slight increase in performance by setting the **lightweight pooling** value to 1.

See Also

lightweight pooling Option

Thread and Fiber Execution

Microsoft® Windows® uses a numeric priority ranging from 1 through 31 (0 is reserved for operating system use) to schedule threads for execution. When several threads are waiting to execute, Windows dispatches the thread with the highest priority.

Each instance of Microsoft SQL Server[™] 2000 defaults to a priority of 7, which is called the normal priority. This gives SQL Server threads a high enough priority to get adequate CPU resources without adversely affecting other applications. The **priority boost** configuration option can be used to increase the priority of the threads from an instance of SQL Server to 13, which is called high priority. This setting gives SQL Server threads a higher priority than most other applications. Thus, SQL Server threads will tend to be dispatched whenever they are ready to run and will not be preempted by threads from other applications. This can improve performance when a server is running only instances of SQL Server and no other applications. If a memory-intensive operation occurs in SQL Server, however, other applications are not likely to have a high-enough priority to preempt the SQL Server thread. If you are running multiple instances of SQL Server on a computer, and turn on priority boost for only some of the instances, the performance of any instances running at normal priority can be adversely affected. The performance of other applications and components on the server can be degraded if **priority boost** is turned on, so it should only be used under tightly controlled conditions.

Some Transact-SQL statements require large amounts of memory for operations, such as sorts. If there is not enough memory available, the thread waits for memory to be freed. The **query wait** option limits how long a thread can wait for memory.

See Also

query wait Option

I/O Architecture

The primary purpose of a database is to store and retrieve data, so performing a lot of disk reads and writes is one of the inherent attributes of a database engine. Disk I/O operations consume many resources and take a relatively long time to complete. Much of the logic in relational database software concerns making the pattern of I/O usage highly efficient.

Microsoft® SQL Server[™] 2000 allocates much of its virtual memory to a buffer cache and uses the cache to reduce physical I/O. Each instance of SQL Server 2000 has its own buffer cache. Data is read from the database disk files into the buffer cache. Multiple logical reads of the data can be satisfied without requiring that the data be physically read again. The data remains in the cache until it has not been referenced for some time and the database needs the buffer area to read in more data. Data is written back to disk only if it is modified. Data can be changed multiple times by logical writes before a physical write transfers the new data back to disk.

The data in a SQL Server 2000 database is stored in 8-KB pages. Each group of eight contiguous pages is a 64-KB extent. The buffer cache is also divided into 8-KB pages.

The I/O from an instance of SQL Server is divided into logical and physical I/O. A logical read occurs every time the database engine requests a page from the buffer cache. If the page is not currently in the buffer cache, a physical read is then performed to read the page into the buffer cache. If the page is currently in the cache, no physical read is generated; the buffer cache simply uses the page already in memory. A logical write occurs when data is modified in a page in memory. A physical write occurs when the page is written to disk. It is possible for a page to remain in memory long enough to have more than one logical write made before it is physically written to disk.

One of the basic performance optimization tasks for an instance of SQL Server involves sizing the SQL Server memory. The goal is to make the buffer cache large enough to maximize the ratio of logical reads to physical reads, but not so large that excessive memory swapping starts generating physical I/O to the pagefile. Instances of SQL Server 2000 do this automatically under the default configuration settings.

By maintaining a relatively large buffer cache in virtual memory, an instance of SQL Server can significantly reduce the number of physical disk reads it requires. After a frequently referenced page has been read into the buffer cache, it is likely to remain there, eliminating further reads.

SQL Server 2000 uses two Microsoft Windows NT® and Windows® 2000 features to improve its disk I/O performance:

• Scatter-gather I/O

Before scatter-gather I/O was introduced in Windows NT version 4.0 Service Pack 2, all of the data for a disk read or write on Windows NT had to be in a contiguous area of memory. If a read transferred in 64 KB of data, the read request had to specify the address of a contiguous area of 64 KB of memory. Scatter-gather I/O allows a read or write to transfer data in to or out of discontiguous areas of memory. Windows 2000 also supports scatter-gather I/O.

If an instance of SQL Server 2000 reads in a 64 KB extent, it does not have to allocate a single 64 KB area and then copy the individual pages to buffer cache pages. It can locate eight buffer pages, and then do a single scatter-gather I/O specifying the address of the eight buffer pages. Windows NT or Windows 2000 places the eight pages directly into the buffer pages, eliminating the need for the instance of SQL Server to do a separate memory copy.

• Asynchronous I/O

In an asynchronous I/O, an application requests a read or write operation from Windows NT or Windows 2000. Windows NT or Windows 2000 immediately returns control to the application. The application can then perform additional work, and later test to see if the read or write has completed. By contrast, in a synchronous I/O, the operating system does not return control to the application until the read or write completes. Using asynchronous I/O allows instances of SQL Server to maximize the work done by individual threads while they are processing a batch.

SQL Server supports multiple concurrent asynchronous I/O operations against each file. SQL Server 2000 dynamically determines the maximum number of I/O

operations an instance can issue for any file.

Reading Pages

The read requests generated by an instance of Microsoft® SQL Server[™] 2000 are controlled by the relational engine and further optimized by the storage engine. The access method used to read pages from a table, such as a table scan, an index scan, or a keyed read, determines the general pattern of reads that will be performed. The relational engine determines the most effective access method. This request is then given to the storage engine, which optimizes the reads required to implement the access method. The thread executing the batch schedules the reads.

Table scans are extremely efficient in SQL Server 2000. The IAM pages in a SQL Server 2000 database list the extents used by a table or index. The storage engine can read the IAM to build a sorted list of the disk addresses that must be read. This allows SQL Server 2000 to optimize its I/Os as large sequential reads that are done in sequence based on their location on the disk. SQL Server 2000 issues multiple serial read-ahead reads at once for each file involved in the scan. This takes advantage of striped disk sets. SQL Server 2000 Enterprise Edition dynamically adjusts the maximum number of read ahead pages based on the amount of memory present; it is fixed in all other editions of SQL Server 2000.

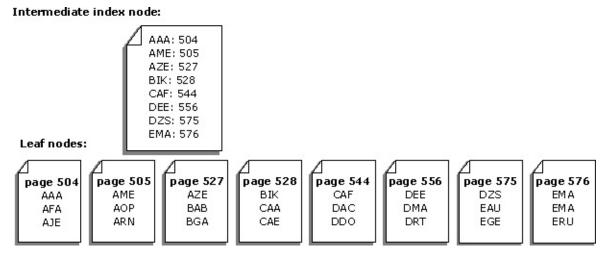
One part of the SQL Server 2000 Enterprise Edition advanced scan feature allows multiple tasks to share full table scans. If the execution plan of a SQL statement calls for a scan of the data pages in a table, and the relational database engine detects that the table is already being scanned for another execution plan, the database engine joins the second scan to the first, at the current location of the second scan. The database engine reads each page once and passes the rows from each page to both execution plans. This continues until the end of the table is reached. At that point, the first execution plan has the complete results of a scan, but the second execution plan must still retrieve the data pages that occur before the point at which it joined the in-progress scan. The scan for second execution plan then wraps back to the first scan. Any number of scans can be combined in this way, the database engine will keep looping through the data pages until it has completed all the scans.

For example, assume that you have a table with 500,000 pages. UserA executes

a SQL statement that requires a scan of the table. When that scan has processed 100,000 pages, **UserB** executes another SQL statement that scans the same table. The database engine will schedule one set of read requests for pages after 100,001, and passes the rows from each page back to both scans. When the scan reaches the 200,000th page, **UserC** executes another SQL statement that scans the same table. Starting with page 200,001, the database engine passes the rows from each page it reads back to all three scans. After reading the 500,000th row, the scan for **UserA** is complete, and the scans for **UserB** and **UserC** wrap back and start reading pages starting with page 1. When the database engine gets to page 100,000, the scan for **UserB** is complete. The scan for **Userc** then keeps going alone until it reads page 200,000, at which point all the scans have been completed.

Reading Index Pages

SQL Server 2000 reads index pages serially in key order. For example, this illustration shows a simplified representation of a set of leaf pages containing a set of keys and the intermediate index node mapping the leaf pages.



SQL Server 2000 uses the information in the intermediate index page above the leaf level to schedule serial read-ahead I/Os for the pages containing the keys. If a request is made for all the keys from 'ABC' to 'DEF', the instance of SQL Server 2000 first reads the index page above the leaf page. It does not, however, simply read each individual data page in sequence from page 504 to page 556, the last one with keys in the desired range. Instead, the storage engine scans the intermediate index page and builds a list of the leaf pages that must be read. The storage engine then schedules all the I/Os in key order. The storage engine also

recognizes that pages 504/505 and 527/528 are contiguous, and performs a single scatter-gather read to retrieve the adjacent pages in one operation. When there are many pages to be retrieved in a serial operation, SQL Server schedules a block of reads at a time. When a subset of these reads is completed, SQL Server schedules an equal number of new reads until all the needed reads have been scheduled.

SQL Server 2000 uses pre-fetching to speed the processing of non-clustered indexes. The leaf rows of a non-clustered index contain pointers to the data rows containing each specific key value. As the database engine reads through the leaf pages of the non-clustered index, it also starts scheduling asynchronous reads for the data rows whose pointers have already been retrieved. This allows the database engine to start retrieving rows before it has completed the scan of the non-clustered index. This process is followed regardless of whether or not the table has a clustered index. SQL Server 2000 Enterprise Edition uses more prefetching than other editions of SQL Server, and the level of pre-fetching is not configurable in any edition.

Freeing and Writing Buffer Pages

In Microsoft[®] SQL Server[™] 2000, one system is responsible for:

- Writing modified buffer pages to disk.
- Marking as free those pages that have not been referenced for some time.

SQL Server 2000 has a singly linked list containing the addresses of free buffer pages. Any thread needing a buffer page uses the first page in the free buffer list.

The buffer cache is an in-memory structure. Each buffer page has a header that contains a reference counter and an indicator of whether the page is dirty, which means the page contains modifications that have not yet been written to disk. The reference counter is incremented by 1 each time a SQL statement references the buffer page. The buffer cache is periodically scanned from the start to the end. Because the buffer cache is all in memory, these scans are very quick and require no I/O. During the scan, the reference counter in each buffer page header is divided by 4 and the remainder discarded. When the reference counter goes to 0, the dirty page indicator is checked. If the page is dirty, a write is scheduled to write the modifications to disk. Instances of SQL Server use a write-ahead log, so the write of the dirty data page is blocked while the log page recording the modification is first written to disk. After the modified page has been flushed to disk, or if the page and the data page it contains is removed and the buffer is placed on the free list.

Using this process, frequently referenced pages remain in memory while buffers holding pages not referenced eventually return to the free buffer list. The instance of SQL Server determines internally the size of the free buffer list, based on the size of the buffer cache. The size cannot be configured.

When an instance of SQL Server is running on Microsoft Windows NT® or Windows® 2000, the work of scanning the buffer, writing dirty pages, and populating the free buffer list is mostly done by the individual worker threads. The worker threads perform their scans in the interval of time after they have

scheduled an asynchronous read and the read completes. A thread gets the address of the next section of the buffer pool that needs to be scanned from a central data structure, then scans that section of the buffer pool while the read I/O processes asynchronously. If a write must be performed, it is also scheduled asynchronously and does not interfere with the thread's ability to process the completion of its own read.

Each instance also has a separate lazywriter thread that scans through the buffer cache. The lazywriter process sleeps for an interval of time. When it is restarted, it checks the size of the free buffer list. If the free buffer list is below a certain point (dependent on the size of the cache) the lazywriter process scans the buffer cache to reclaim unused pages and write dirty pages that have a reference count of 0. On the Windows NT and Windows 2000 operating systems, most of the work populating the free buffer list and writing dirty pages is done by the individual threads and the lazywriter thread typically finds little to do. Windows 95 and Windows 98 do not support asynchronous writes, so the lazywriter thread does the work of populating the free buffer list and writing dirty pages.

The checkpoint process also scans the buffer cache periodically and writes any dirty log or data pages to disk. The difference is that the checkpoint process does not place the buffer page back on the free list. The work of the checkpoint process is intended to minimize the number of dirty pages in memory to reduce the length of a recovery if the server fails, not to populate the free buffer list. Checkpoints typically find few dirty pages to write to disk because most dirty pages are written to disk by the worker threads or lazywriter thread in the period between two checkpoints.

Writes of log records are usually scheduled asynchronously by a logwriter thread. The exceptions are when:

- A commit forces all pending log records for a transaction to disk.
- A checkpoint forces all pending log records for all transactions to disk.

Distributed Query Architecture

Microsoft® SQL Server[™] 2000 supports two methods for referencing heterogeneous OLE DB data sources in Transact-SQL statements:

• Linked server names

The system stored procedures **sp_addlinkedserver** and **sp_addlinkedsrvlogin** are used to give a server name to an OLE DB data source. Objects in these linked servers can be referenced in Transact-SQL statements using four-part names. For example, if a linked server name of **DeptSQLSrvr** is defined against another copy of SQL Server 2000, the following statement references a table on that server:

SELECT * FROM DeptSQLSrvr.Northwind.dbo.Employees

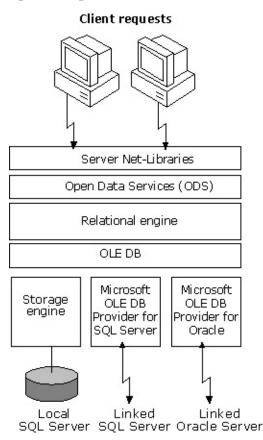
The linked server name can also be specified in an OPENQUERY statement to open a rowset from the OLE DB data source. This rowset can then be referenced like a table in Transact-SQL statements.

• Ad hoc connector names

For infrequent references to a data source, the OPENROWSET or OPENDATASOURCE functions are specified with the information needed to connect to the linked server. The rowset can then be referenced the same way a table is referenced in Transact-SQL statements:

SELECT * FROM OPENROWSET('Microsoft.Jet.OLEDB.4.0', 'c:\MSOffice\Access\Samples\Northwind.mdb';'Admin';"; Employees)

SQL Server 2000 uses OLE DB to communicate between the relational engine and the storage engine. The relational engine breaks down each Transact-SQL statement into a series of operations on simple OLE DB rowsets opened by the storage engine from the base tables. This means the relational engine can also open simple OLE DB rowsets on any OLE DB data source.



The relational engine uses the OLE DB API to open the rowsets on linked servers, to fetch the rows, and to manage transactions.

For each OLE DB data source accessed as a linked server, an OLE DB provider must be present on the server running SQL Server. The set of Transact-SQL operations that can be used against a specific OLE DB data source depends on the capabilities of the OLE DB provider. For more information, see <u>OLE DB</u> <u>Provider Reference for Distributed Queries</u>.

When possible, SQL Server pushes relational operations such as joins, restrictions, projections, sorts, and group by operations to the OLE DB data source. SQL Server does not default to scanning the base table into SQL Server and performing the relational operations itself. SQL Server queries the OLE DB provider to determine the level of SQL grammar it supports, and, based on that information, pushes as many relational operations as possible to the provider. For more information, see <u>SQL Dialect Requirements for OLE DB Providers</u>.

SQL Server 2000 specifies a mechanism for an OLE DB provider to return

statistics indicating how key values are distributed within the OLE DB data source. This lets the SQL Server query optimizer better analyze the pattern of data in the data source against the requirements of each SQL statement, increasing the ability of the query optimizer to generate optimal execution plans. For more information, see <u>Distribution Statistics Requirements for OLE DB</u> <u>Providers</u>.

See Also

Configuring Linked Servers

Distributed Queries

Full-Text Query Architecture

The Microsoft® SQL Server[™] 2000 full-text query component supports sophisticated searches on character string columns.

This capability is implemented by the Microsoft Search service, which has two roles:

• Indexing support

Implements the full-text catalogs and indexes defined for a database. Accepts definitions of full-text catalogs, and the tables and columns making up the indexes in each catalog. Implements requests to populate the full-text indexes.

• Querying support

Processes full-text search queries. Determines which entries in the index meet the full-text selection criteria. For each entry that meet the selection criteria, it returns the identity of the row plus a ranking value to the MSSQLServer service, where this information is used to construct the query result set. The types of queries supported include searching for:

- Words or phrases.
- Words in close proximity to each other.
- Inflectional forms of verbs and nouns.

The full-text engine runs as a service named Microsoft Search on Microsoft Windows NT® Server, Windows NT Advanced Server, Windows® 2000 Server, or Windows 2000 Advanced Server. It is installed when the Full-Text Search feature is selected during custom installation. The Microsoft Search service itself is not installed during an installation of SQL Server Desktop Engine. While this means that the Microsoft Search service is not installed on Microsoft Windows 95, Windows 98, Windows NT Workstation, or Windows 2000 Professional clients, these clients can make use of the service when connected to an instance

of SQL Server 2000 Standard Edition, SQL Server 2000 Developer Edition, or SQL Server 2000 Enterprise Edition.

The full-text catalogs and indexes are not stored in a SQL Server database. They are stored in separate files managed by the Microsoft Search service. The full-text catalog files are not recovered during a SQL Server recovery. They also cannot be backed up and restored using the Transact-SQL BACKUP and RESTORE statements. The full-text catalogs must be resynchronized separately after a recovery or restore operation. The full-text catalog files are accessible only to the Microsoft Search service and the Windows NT or Windows 2000 system administrator.

Communications between SQL Server and the Microsoft Search service are made through a full-text provider.

The full-text catalogs, indexes, and searches supported by the Microsoft Search service apply only to tables in SQL Server databases. The Windows NT Indexing Service and Windows 2000 Indexing Service provides similar functionality against operating system files. Indexing Service includes an OLE DB Provider for Indexing Service that can be used by OLE DB consumers. SQL Server applications can access the OLE DB Provider for Indexing Service through distributed queries. Transact-SQL statements can combine full-text searches referencing SQL Server tables with textual searches of file data by using both the full-text SQL constructs with distributed query references to the OLE DB Provider for Indexing Service. For more information, see <u>Full-text Querying of File Data</u>.

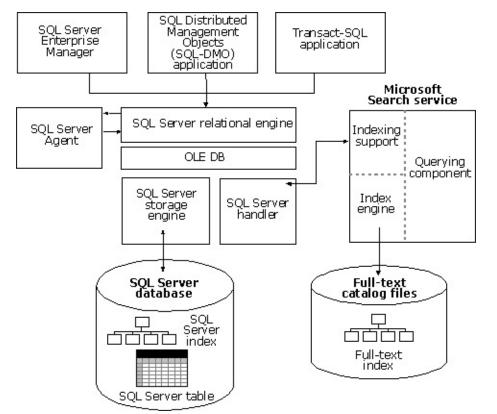
There is only one Microsoft Search service on any computer running multiple instances of SQL Server. The single instance of the full-text search engine manages the full-text indexes for all the instances of SQL Server 2000 and SQL Server version 7.0 on the computer.

See Also

<u>Full-Text Catalogs and Indexes</u> <u>Full-text Indexes</u> <u>Full-text Search</u> Microsoft Search Service

Full-Text Indexing Support

This illustration shows the components that make up the full-text indexing support. These are the components involved in defining, creating, and populating full-text indexes.



Enabling databases and tables for full-text indexing, defining, and populating the indexes is specified using:

• SQL Server Enterprise Manager.

One of the nodes of a database tree in SQL Server Enterprise Manager is used to manage the full-text catalogs in the database.

- Applications using SQL Distributed Management Objects (SQL-DMO). SQL-DMO has objects for managing full-text catalogs and indexes.
- Applications using Transact-SQL and a standard database API.

Transact-SQL has a set of system stored procedures for managing full-

text catalogs and indexes.

The other components define and populate full-text indexes in this manner:

- 1. A Microsoft® SQL Server[™] 2000 database is enabled for full-text indexing.
- 2. The full-text catalogs for the database are specified.
- 3. Individual tables are enabled for full-text indexing and associated with a catalog.
- 4. Individual columns in each table are added to the full-text index for the table. All the meta data information from Steps from 1 through 4 is stored in system tables in SQL Server databases.
- 5. The full-text indexes for each table are activated on a table-by-table basis. When a full-text table index is activated, a start seed value is sent from an instance of SQL Server to the indexing service within the Microsoft Search service. The start seed value identifies the table involved in the full text index.
- 6. Population is requested on either a catalog-by-catalog or table-by-table basis. Populating on a catalog basis allows you to populate multiple indexes in one operation; populating tables lets you populate specific indexes.

The population in Step 6 can take different forms:

• Full population

If a full population is requested for a full-text catalog, index entries are built for all the rows in all the tables covered by the catalog. If a full populates is requested for a table, index entries are built for all the rows in that table. A full population typically occurs when a catalog or index is first populated, the indexes can then be maintained using change tracking or incremental populations.

• Change tracking population

Maintains a record of the rows that have been modified in a system table, and propagates the changes to the full-text index. You start the change tracking by executing **sp_fulltext_table** and specify **start_change_tracking** for the **@action** parameter. When using change tracking, you also specify when the changes are taken from the history table and populated in the full-text index:

• Background

After starting change tracking with **start_change_tracking**, you can execute **sp_fulltext_table** specifying **start_background_updateindex** for the **@action** parameter. With this option, changes to rows in the table are propagated to the full-text index as they occur.

• On demand

In this option, all tracked changes are stored in the history, and only propagated to the full-text index when you execute **sp_fulltext_table** specifying **update_index** for the @action parameter.

• Scheduled

You can use SQL Agent to schedule periodic jobs that execute **sp_fulltext_table** specifying **update_index** for the **@action** parameter. This will propagate all outstanding tracked changes to the index.

• Incremental population

Only adjusts index entries for rows added, deleted, or modified after the last population. This feature requires that the indexed table have a column of the **timestamp** data type. If the table does not have a **timestamp** column, only full or change tracking populations can be performed. Requests for incremental populations on tables without **timestamp** columns result in a full population operation.

If a new full-text index is defined for a table not associated with the catalog before, the next catalog-level incremental population request builds all the entries for the table.

Incremental population requests are implemented as full populations if any of the meta data for the table has changed since the last population. This includes altering any column, index, or full-text index definitions.

Each population request is sent to the indexing service within the Microsoft Search service:

- The indexing service passes the appropriate start seed value to the SQL Server Handler. The start seed value contains information such as the table and index involved in the population, and the **timestamp** value (if the table has a **timestamp** column) associated with the last full or incremental population performed for the index.
- The SQL Server Handler is a driver containing logic to extract text data from the SQL Server columns involved in a full-text index. The Handler retrieves the data from SQL Server and passes it back to the index service. For a full population, the SQL Server Handler extracts all the rows in the table. For an incremental population, the SQL Server Handler only extracts information from rows whose current **timestamp** values are higher than the **timestamp** associated with the last population, which is stored in the start seed.
- The indexing service then passes an index identifier and the strings to be indexed to the index engine. The index engine eliminates noise words such as a, and, or the. It also determines the word boundaries and builds a full-text index covering the words passed down from the indexing service. This linguistic analysis differs depending on the language in which the text is written. SQL Server 2000 supports linguistic analysis for several languages; the language is specified using **sp_fulltext_column**. The full-text index is stored in the full-text catalog file.

• At the end of the population, the indexing service calculates a new start seed value that records the point at which a subsequent incremental population should start.

See Also

Full-Text Catalogs and Indexes

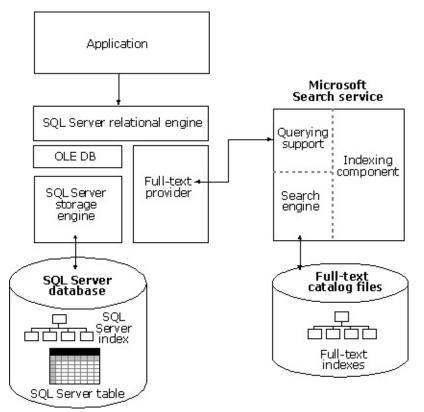
Full-text Indexes

Full-text Search

Microsoft Search Service

Full-Text Querying Support

When Microsoft® SQL Server[™] 2000 receives a Transact-SQL statement with a full-text construct, it retrieves the needed information from the Microsoft Search service using the full-text provider. Full-text constructs are the CONTAINS or FREETEXT predicates, or the CONTAINSTABLE or FREETEXTTABLE rowset functions. The full-text constructs can reference multiple columns in the full-text index if it is not known which column may contain the search conditions. The following illustration shows the flow of this process.



The steps involved in this process include:

- 1. An application sends an instance of SQL Server a Transact-SQL statement with a full-text construct.
- 2. The SQL Server relational engine validates the full-text construct by querying the system tables to determine if the column reference is covered by a full-text index. The relational engine reduces each SQL

statement to a series of rowset operations, and uses OLE DB to pass the operations to underlying components, usually the storage engine. The relational engine transforms any full-text construct into a request for a rowset from the full-text provider instead of the storage engine. The rowset requested is the set of keys satisfying the search condition and a ranking indicating how well the data for each key met the search condition criteria. The command sent with the rowset request to the full-text provider includes the full-text search condition.

- 3. The full-text provider validates the request and changes the search conditions to a form used by the querying support component of the Microsoft Search service. The request is sent to the search service.
- 4. The querying support component uses the search engine component to extract the requested data from the full-text index. This data is then passed back to the full-text provider in the form of a rowset.
- 5. The full-text provider returns the rowset to the relational engine.
- 6. The relational engine combines all the rowsets it receives from the storage engine and the full-text provider to build the final result set it sends back to the client.

See Also

<u>Full-Text Catalogs and Indexes</u> <u>Full-text Indexes</u> <u>Full-text Search</u> Microsoft Search Service

Extended Stored Procedure Architecture

Microsoft® SQL Server[™] 2000 extended stored procedures extend Transact-SQL functionality by enabling you to implement logic in functions contained in dynamic-link library (DLL) files, and call those functions from Transact-SQL statements just as you would a Transact-SQL procedure. Dynamic-link library files have the .dll file name extension. Extended stored procedures can include most of the features of Microsoft Win32® and COM applications.

A DLL file must conform to the Extended Stored Procedure API to operate as an extended stored procedure. The DLL can contain multiple functions identified to SQL Server as extended stored procedures; each function is identified by a separate extended stored procedure name. When a Transact-SQL statement references one of the extended stored procedures, the relational database engine calls the function associated with the extended stored procedure name. Extended stored procedures can open a connection back to the SQL Server instance that called them, or connect to remote SQL Server installations.

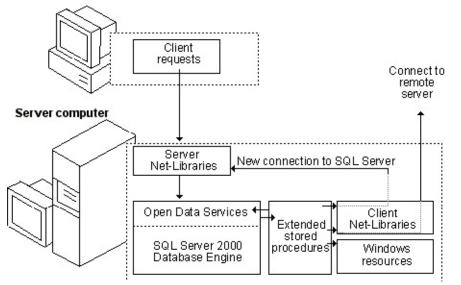
Extended stored procedures are a part of the Open Data Services layer of the relational database engine, which is the interface between the engine and the server Net-libraries. The server Net-Libraries receive client TDS packets and pass them to Open Data Services. Open Data Services transforms the TDS packets into events that it passes to other parts of the relational database engine. The database engine then uses Open Data Services to send replies back to SQL Server clients through the server Net-Libraries.

When the relational database engine determines that a Transact-SQL statement references an extended stored procedure:

- The relational database engine passes the extended stored procedure request to the Open Data Services layer.
- Open Data Services then loads the DLL containing the extended stored procedure function into the SQL Server 2000 address space, if not already loaded.

- Open Data Services passes the request to the extended stored procedure.
- Open Data Services returns the results of the operation to the database engine.





In the past, The Open Data Services API was also used to write server applications, such as gateways to other database systems. These types of applications have been replaced by newer technologies such as:

• Database APIs that support multiple different databases and other data sources, such as OLE DB and ODBC.

Applications written to the OLE DB or ODBC APIs have little need for a gateway to access different databases.

- SQL Server 2000 supports heterogeneous distributed queries, which allow Transact-SQL queries to pull data from any OLE DB data source without any need for specialized server applications.
- MS DTC, which allows distributed transactions to span multiple databases.
- Windows NT Component Services, for running midtier application

logic.

SQL Server 2000 does not support the obsolete portions of the Open Data Services API. The only part of the original Open Data Services API still supported by SQL Server 2000 are the extended stored procedure functions, so the API has been renamed to the Extended Stored Procedure API.

See Also

Programming Extended Stored Procedures

Remote Stored Procedure Architecture

Remote stored procedures are a legacy feature of Microsoft® SQL Server[™] 2000. Their functionality in Transact-SQL is limited to executing a stored procedure on a remote SQL Server installation. The distributed queries introduced in SQL Server version 7.0 support this ability along with the ability to access tables on linked, heterogeneous OLE DB data sources directly from local Transact-SQL statements. Instead of using a remote stored procedure call on SQL Server 2000, use distributed queries and an EXECUTE statement to execute a stored procedure on a remote server.

An instance of SQL Server 2000 can send and receive remote stored procedure calls to other instances of SQL Server 2000 and SQL Server version 7.0. An instance of SQL Server 2000 can also send and receive remote stored procedure calls to instances of SQL Server version 6.0 or SQL Server version 6.5. A server running SQL Server 2000 can receive remote stored procedure calls from an instance of SQL Server version 4.21a, but the instance of SQL Server 2000 cannot make remote stored procedure calls to the instance of SQL Server version 4.21a. The instance of SQL Server 4.21a cannot recognize the version of the Tabular Data Stream (TDS) used by SQL Server 2000.

Remote Stored Procedure Protocol Optimizations

The Microsoft OLE DB Provider for SQL Server and the SQL Server ODBC driver both make use of a TDS protocol performance optimization originally introduced to support remote stored procedures. The use of this optimization can be seen in SQL Profiler traces.

SQL Profiler traces events in an instance of SQL Server, such as receipt and return of the Tabular Data Stream (TDS) packets sent between applications and an instance of SQL Server. TDS is the application-level protocol defined for SQL Server client/server communications.

When an application sends a Transact-SQL batch for execution, a generic packet for executing SQL is used that shows up in the SQL Profiler trace as SQL:BatchStarting and SQL:BatchCompleted events. When one instance of SQL Server sends a request for another instance of SQL Server to execute a remote stored procedure, a specialized RPC TDS packet is used. The RPC packet is tailored to the needs of transmitting requests to execute a stored procedure. The relational engine also recognizes that this is a specialized packet and implements a number of optimizations that speeds the execution of the stored procedure. These show up in a SQL Profiler trace as RPC:Starting and RPC:Completed events.

SQL Server 2000 does not limit the use of these specialized RPC packets to server-to-server communications. The Microsoft OLE DB Provider for SQL Server and the SQL Server ODBC driver use this specialized RPC packet to increase performance in two cases:

- If an application uses the ODBC CALL syntax to execute a stored procedure.
- When the provider or driver internally generate calls to system stored procedures.

Users analyzing SQL Profiler traces from applications using the provider or driver can see these RPC TDS events.

See Also

Configuring Remote Servers

Calling a Stored Procedure

Executing Stored Procedures

Calling a Stored Procedure (OLE DB)

TSQL Event Category

Transactions Architecture

Microsoft® SQL Server[™] 2000 maintains the consistency and integrity of each database despite errors that occur in the system. Every application that updates data in a SQL Server database does so using transactions. A transaction is a logical unit of work made up of a series of statements (selects, inserts, updates, or deletes). If no errors are encountered during a transaction, all of the modifications in the transaction become a permanent part of the database. If errors are encountered, none of the modifications are made to the database.

A transaction goes through several phases:

- Before the transaction starts, the database is in a consistent state.
- The application signals the start of a transaction. This can be done explicitly with the BEGIN TRANSACTION statement. Alternatively, the application can set options to run in implicit transaction mode; the first Transact-SQL statement executed after the completion of a prior transaction starts a new transaction automatically. No record is written to the log when the transaction starts; the first record is written to the log when the application generates the first log record for a data modification.
- The application starts modifying data. These modifications are made one table at a time. As a series of modifications are made, they may leave the database in a temporarily inconsistent intermediate state.
- When the application reaches a point where all the modifications have completed successfully and the database is once again consistent, the application commits the transaction. This makes all the modifications a permanent part of the database.
- If the application encounters some error that prevents it from completing the transaction, it undoes, or rolls back, all the data

modifications. This returns the database to the point of consistency it was at before the transaction started.

SQL Server applications can also run in autocommit mode. In autocommit mode each individual Transact-SQL statement is committed automatically if it is successful and rolled back automatically if it generates an error. There is no need for an application running in autocommit mode to issue statements that specifically start or end a transaction.

All Transact-SQL statements run in a transaction: an explicit transaction, an implicit transaction, or an autocommit transaction. All SQL Server transactions that include data modifications either reach a new point of consistency and are committed, or are rolled back to the original point of consistency. Transactions are not left in an intermediate state where the database is not consistent.

See Also

Transactions

Transaction Recovery

Every Microsoft® SQL Server[™] 2000 database has a transaction log that records data modifications made in the database. The log records the start and end of every transaction and associates each modification with a transaction. An instance of SQL Server stores enough information in the log to either redo (roll forward) or undo (roll back) the data modifications that make up a transaction. Each record in the log is identified by a unique log sequence number (LSN). All of the log records for a transaction are chained together.

An instance of SQL Server records many different types of information in the transaction log. Instances of SQL Server 2000 primarily log the logical operations performed. The operation is reapplied to roll forward a modification, and the opposite of the logical operation is performed to roll back a modification.

Each instance of SQL Server controls when modifications are written from its data buffers to disk. An instance of SQL Server may cache modifications in buffers for a period of time to optimize disk writes. A buffer page that contains modifications that have not yet written to disk is known as a <u>dirty page</u>. Writing a dirty buffer page to disk is called flushing the page. When modifications are cached, care must be taken to ensure that no data modification is flushed before the corresponding log image is written to the log file. This could create a modification that could not be rolled back if necessary. To ensure that they can recover all modifications, instances of SQL Server use a write-ahead log, which means that all log images are written to disk before the corresponding data modification.

A commit operation forces all log records for a transaction to the log file so that the transaction is fully recoverable even if the server is shut down. A commit operation does not have to force all the modified data pages to disk as long as all the log records are flushed to disk. A system recovery can roll the transaction forward or backward using only the log records.

Periodically, each instance of SQL Server ensures that all dirty log and data pages are flushed. This is called a checkpoint. Checkpoints reduce the time and resources needed to recover when an instance of SQL Server is restarted. For more information on checkpoint processing, see <u>Checkpoints and the Active</u>

Rolling Back an Individual Transaction

If any errors occur during a transaction, the instance of SQL Server uses the information in the log file to roll back the transaction. This rollback does not affect the work of any other users working in the database at the same time. Usually, the error is returned to the application, and if the error indicates a possible problem with the transaction, the application issues a ROLLBACK statement. Some errors, such as a 1205 deadlock error, roll back a transaction automatically. If anything stops the communication between the client and an instance of SQL Server while a transaction is active, the instance rolls back the transaction automatically when notified of the stoppage by the network or operating system. This could happen if the client application terminates, if the client computer is shut down or restarted, or if the client network connection is broken. In all of these error conditions, any outstanding transaction is rolled back to protect the integrity of the database.

Recovery of All Outstanding Transactions at Start-up

It is possible for an instance of SQL Server to sometimes stop processing (for example, if an operator restarts the server while users are connected and working in databases). This can create two problems:

- There may be an unknown number of SQL Server transactions partially completed at the time the instance stopped. These incomplete transactions need to be rolled back.
- There may be an unknown number of data modifications recorded in the SQL Server database log files, but the corresponding modified data pages were not flushed to the data files before the server stopped. Any committed modifications must be rolled forward.

When an instance of SQL Server is started, it must find out if either of these conditions exist and address them. The following steps are taken in each SQL Server database that is in the instance:

• The LSN of the last checkpoint is read from the database boot block

along with the Minimum Recovery LSN.

- The transaction log is scanned from the Minimum Recovery LSN to the end of the log. All committed dirty pages are rolled forward by redoing the logical operation recorded in the log record.
- The instance of SQL Server then scans backward through the log file rolling back all uncompleted transactions by applying the opposite of the logical operation recorded in the log records.

The RESTORE statement also uses this type of recovery, unless a user specifies the NORECOVERY option. When restoring a sequence of database, differential, or log backups to recover a database to a point of failure, you specify NORECOVERY on all RESTORE statements except when restoring the last log backup. When the last backup in the sequence is restored, the RESTORE statement also has to ensure that all uncompleted transactions are rolled back. You specify the RECOVERY option on this RESTORE statement, in which case it uses the same logic as the startup recovery process to roll back all transactions that are still marked incomplete at the end of the last log.

Concurrency Architecture

When many people attempt to modify data in a database at the same time, a system of controls must be implemented so that modifications made by one person do not adversely affect those of another person. This is called concurrency control.

Concurrency control theory has two classifications for the methods of instituting concurrency control:

• Pessimistic concurrency control

A system of locks prevents users from modifying data in a way that affects other users. After a user performs an action that causes a lock to be applied, other users cannot perform actions that would conflict with the lock until the owner releases it. This is called pessimistic control because it is mainly used in environments where there is high contention for data, where the cost of protecting data with locks is less than the cost of rolling back transactions if concurrency conflicts occur.

• Optimistic concurrency control

In optimistic concurrency control, users do not lock data when they read it. When an update is performed, the system checks to see if another user changed the data after it was read. If another user updated the data, an error is raised. Typically, the user receiving the error rolls back the transaction and starts over. This is called optimistic because it is mainly used in environments where there is low contention for data, and where the cost of occasionally rolling back a transaction outweighs the costs of locking data when read.

Microsoft® SQL Server[™] 2000 supports a wide range of optimistic and pessimistic concurrency control mechanisms. Users specify the type of concurrency control by specifying:

- A transaction isolation level for a connection.
- Concurrency options on cursors.

These attributes can be defined using either Transact-SQL statements or through the properties and attributes of the database APIs such as ADO, OLE DB, and ODBC.

See Also

Four Concurrency Problems

Cursor Concurrency

Locking Architecture

A lock is an object used by software to indicate that a user has some dependency on a resource. The software does not allow other users to perform operations on the resource that would adversely affect the dependencies of the user owning the lock. Locks are managed internally by system software and are acquired and released based on actions taken by the user.

Microsoft® SQL Server[™] 2000 uses locks to implement pessimistic concurrency control among multiple users performing modifications in a database at the same time. By default, SQL Server manages both transactions and locks on a per connection basis. For example, if an application opens two SQL Server connections, locks acquired by one connection cannot be shared with the other connection. Neither connection can acquire locks that would conflict with locks held by the other connection. Only bound connections are not affected by this rule. For more information, see <u>Using Bound Connections</u>.

SQL Server locks are applied at various levels of granularity in the database. Locks can be acquired on rows, pages, keys, ranges of keys, indexes, tables, or databases. SQL Server dynamically determines the appropriate level at which to place locks for each Transact-SQL statement. The level at which locks are acquired can vary for different objects referenced by the same query; for example one table may be very small and have a table lock applied, while another, larger table may have row locks applied. The level at which locks are applied does not have to be specified by users and needs no configuration by administrators. Each instance of SQL Server ensures that locks granted at one level of granularity respect locks granted at another level. For example, if **UserA** attempts to acquire a share lock on a row, the instance of SQL Server also attempts to acquire intent share locks on the page and the table. If **UserB** has an exclusive lock at the page or table level, **UserA** is blocked from acquiring locks until the lock held by **UserB** is freed.

There are several lock modes: shared, update, exclusive, intent, and schema. The lock mode indicates the level of dependency the connection has on the locked object. SQL Server controls how the lock modes interact. For example, an exclusive lock cannot be obtained if other connections hold shared locks on the resource.

Locks are held for the length of time needed to protect the resource at the level requested:

- The duration of share locks used to protect reads depends on the transaction isolation levels. At the default transaction isolation level of READ COMMITTED, a share lock is held only as long as it takes to read a page. In scans, the lock is held until a lock is acquired on the next page in a scan. If the HOLDLOCK hint is specified, or the transaction isolation level is set to either REPEATABLE READ or SERIALIZABLE, the locks are held to the end of the transaction.
- Depending on the concurrency options set for a cursor, the cursor may acquire shared-mode, scroll locks to protect fetches. When scroll locks are needed, they are held until the next fetch or the closing of the cursor, whichever happens first. If HOLDLOCK is specified, however, the scroll locks are held until the end of the transaction.
- Exclusive locks used to protect updates are held until the end of the transaction.

If a connection attempts to acquire a lock that conflicts with a lock held by another connection, the connection attempting to acquire the lock is blocked until:

- The conflicting lock is freed and the connection acquires the lock it requested.
- The time-out interval for the connection expires. By default, there is no time-out interval, but some applications set a time-out interval to prevent an indefinite wait.

If several connections become blocked waiting for conflicting locks on a single resource, the locks are granted on a first-come, first-serve basis as the preceding connections free their locks.

SQL Server has an algorithm to detect deadlocks, a condition where two connections have blocked each other. If an instance of SQL Server detects a

deadlock, it will terminate one transaction, allowing the other to continue. For more information, see <u>Deadlocking</u>.

SQL Server may dynamically escalate or deescalate the granularity or type of locks. For example, if an update acquires a large number of row locks and has locked a significant percentage of a table, the row locks are escalated to a table lock. If a table lock is acquired, the row locks are released. SQL Server 2000 rarely needs to escalate locks; the query optimizer usually chooses the correct lock granularity at the time the execution plan is compiled. For more information, see Lock Escalation and Dynamic Locking.

See Also

Locking

Cursor Locking

Latching

Latches are very lightweight, short-term synchronization objects protecting actions that need not be locked for the life of a transaction. They are primarily used to protect a row when read for a connection.

When the relational engine is processing a query, each time a row is needed from a base table or index, the relational engine uses the OLE DB API to request that the storage engine return the row. While the storage engine is actively transferring the row to the relational engine, the storage engine must ensure that no other task modifies either the contents of the row or certain page structures such as the page offset table entry locating the row being read. The storage engine does this by acquiring a latch, transferring the row in memory to the relational engine, and then releasing the latch.

SQL Server Performance Monitor has a **Latches** object that indicates how many times latches could not be granted immediately and the amount of time threads spent waiting for latches to be granted.

See Also

SQL Server: Latches Object

Distributed Transactions Architecture

Distributed transactions are transactions that involve resources from two or more sources. Microsoft® SQL Server[™] 2000 supports distributed transactions, allowing users to create transactions that update multiple SQL Server databases and other sources of data.

A distributed transaction involves:

• Resource managers

The software controlling each resource involved in a distributed transaction is known as a resource manager. A distributed transaction is made up of local transactions in each individual resource manager. Each resource manager must be able to commit or roll back its local transaction in coordination with all the other resource managers in the distributed transaction. SQL Server can operate as a resource manager in a distributed transaction that complies with the X/Open XA specification for Distributed Transaction Processing.

• Transaction manager

Committing or rolling back a distributed transaction is controlled by a software component called a transaction manager. The transaction manager coordinates with each resource manager to ensure that all the local transactions making up the distributed transaction are committed or rolled back together. The Microsoft Distributed Transaction Coordinator (MS DTC) service operates as a transaction manager. MS DTC complies with the X/Open XA specification for Distributed Transaction Processing.

• Two-phase commit (2PC)

Special commit processing is required to prevent problems in managing transactions spanning multiple resource managers. A commit of a large transaction can take a relatively long time as log buffers are flushed freed. The commit process itself can also encounter errors that would force a rollback. If a transaction manager simply asked each resource manager to commit, it could get a success status back from some

resource managers and then get an error from one resource manager. This creates a conflict because all of the distributed transaction should be rolled back, but parts are already committed. Two-phase commits address this problem by dividing a commit into two phases:

• Prepare

The transaction manager sends a prepare to commit request to each resource manager. Each resource manager then performs all resource-intensive actions needed to complete the commit process, such as flushing all log buffers. The resource manager only retains the minimum locks needed to maintain the integrity of the transaction, and then returns success to the transaction manager.

• Commit

If all the resource managers return success to their prepare requests, the transaction manager then sends commit commands to each resource manager. Each resource manager then quickly records the transaction as completed and frees the last held resources. If any resource manager returns an error to the prepare request, the transaction manager then sends rollback commands to each resource manager.

There are several ways applications can include SQL Server 2000 in a distributed transaction:

- If an application has a local transaction and issues a distributed query, the local transaction is escalated to a distributed transaction.
- Issue a BEGIN DISTRIBUTED TRANSACTION statement.
- If an application has a local transaction and the option REMOTE_PROC_TRANSACTIONS is set ON, calling a remote stored procedure escalates the local transaction to a distributed transaction.
- Applications using the Microsoft OLE DB Provider for SQL Server or

the SQL Server ODBC driver can use OLE DB methods or ODBC functions to have a SQL Server connection join a distributed transaction started by the application.

See Also

Distributed Transactions

MS DTC Service

Cursor Architecture

All SQL statements operate on a set of rows. A SELECT statement returns a complete result set containing all the rows that meet the qualifications in the SELECT statement. Applications need to process the result set one row or block of rows at a time. Cursors are a logical extension to result sets that let applications work with the result set row by row.

Microsoft® SQL Server[™] 2000 supports several mechanisms for specifying cursors:

- Transact-SQL supports the SQL-92 DECLARE CURSOR, OPEN, FETCH, and CLOSE statements for managing cursors. Transact-SQL also supports cursor extensions such as:
 - A DEALLOCATE statement to allow optimizations in reusing cursors.
 - Defining a Transact-SQL variable to have a cursor data type and then using it to refer to a cursor.
 - Defining a cursor to have local or global scope.
 - Specifying the cursor types from the OLE DB and ODBC specifications (FORWARD_ONLY, STATIC, KEYSET, and DYNAMIC) in a DECLARE CURSOR statement.
- The Microsoft OLE DB Provider for SQL Server supports the cursor functionality of the ADO and OLE DB APIs.
- The Microsoft SQL Server ODBC driver supports the cursor functionality of the ODBC, RDO, DAO, and Microsoft Foundation Classes Database Classes APIs.
- Microsoft Embedded SQL for C supports the cursor functionality of the

Embedded SQL standard.

• The DB-Library API supports the same level of cursor functionality as the OLE DB and ODBC APIs.

See Also

<u>Cursors</u>

Server Scalability

Microsoft® SQL Server[™] 2000 extends the scalability of SQL Server at both ends of the performance spectrum. The SQL Server 2000 database engine that runs on Microsoft Windows NT® and Windows® 2000 includes support for items such as 64 GB of physical memory and distributed partitioned views that allow you to implement groups of database servers that can scale to meet the processing requirements of the largest Web sites or enterprise data systems. SQL Server CE adds enhanced support for mobile users by running on Microsoft Windows CE. These enhancements make SQL Server 2000 a good choice for managing databases ranging from a small, personal database on a kilobyte-sized Windows CE device to terabyte-sized databases accessed by thousands of Internet users.

SQL Server 2000 on Large Servers

One of the primary design goals for Microsoft® SQL Server[™] 2000 and SQL Server version 7.0 is to increase their ability to implement the databases supporting the largest Web sites and enterprise systems. Although earlier versions of SQL Server do well at supporting large numbers of concurrent users, the length of time it takes them to run utility, backup, and restore operations limits the size of a manageable SQL Server database to 200 through 300 GB.

SQL Server 2000 Enterprise Edition is capable of handling terabyte-sized databases with thousands of concurrent users. Some of the features that allow this are:

- SQL Server Enterprise Edition can scale effectively on up to 32 microprocessors on SMP computers running Microsoft Windows® 2000 DataCenter.
- SQL Server 2000 Enterprise Edition can use up to 64 GB of physical memory (RAM) on Windows 2000 DataCenter. For more information, see <u>Using AWE Memory on Windows 2000</u>.
- SQL Server 2000 Enterprise Edition supports distributed partitioned views, which allow groups of database servers to support the workload of a large Web site or enterprise system. Such a group, or federation, of servers must be administered separately, but provide the same level of performance as a cluster of database servers. For more information, see Federated SQL Server 2000 Servers.
- SQL Server 2000 Enterprise Edition supports indexed views. Creating an index on a view causes the view to be materialized, and its result set stored in the same format as a table. For certain types of views, this can improve performance exponentially. For more information, see <u>View</u> <u>Indexes</u>.

- The on-disk data structures that support parallel processing and serial, read-ahead scans. Table scans and index scans can now be performed serially, which is especially useful in online analytical processing (OLAP) that characterizes data warehouses. For more information, see <u>I/O Architecture</u>.
- SQL Server 2000 natively supports the prepare/execute model of executing SQL statements. It also has logic to share query execution plans between connections without requiring an application to prepare the statement. These features reduce the overhead associated with compiling and executing statements. For more information, see <u>Execution Plan Caching and Reuse</u>.
- Hash and merge join types offer improved join performance. For more information, see <u>Advanced Query Tuning Concepts</u>.
- SQL Server 2000 supports intra-query parallelism on servers that have more than one microprocessor, or CPU. Individual SQL statements can be split into two or more tasks that operate concurrently to return the results faster. For more information, see <u>Parallel Query Processing</u>.
- SQL Server 2000 evaluates an SQL statement and dynamically chooses the locking granularity (row, page, table) that will maximize concurrent throughput. For more information, see <u>Locking Architecture</u>.
- SQL Server 2000 uses Microsoft Windows NT® and Windows 2000 asynchronous I/O and scatter-gather I/O, along with buffer cache management algorithms to maximize OLTP performance. For more information, see I/O Architecture.
- The speed of the BACKUP and RESTORE statements is fast enough to run the statements during production work because they do not interfere with database activity. BACKUP and RESTORE use parallel I/Os when a backup is stored on multiple backup devices. BACKUP options, such

as differential backups, and backing up only files or filegroups, reduce size of backups and their effect on the system. For more information, see <u>Backup/Restore Architecture</u>.

- The SQL Server 2000 and SQL Server 7.0 on-disk data structures are much simpler than in earlier versions, which make the structures more robust. Also, the database engine is coded to detect errors at relatively early points in processing and terminate a task before it causes problems in the database itself (fail-fast logic). These improvements result in fewer problems with on-disk structures and reduce or eliminate the need to run database integrity checks.
- The algorithms in the database integrity check statements are much faster in SQL Server 2000 and SQL Server 7.0 than in earlier versions. The integrity check statements now make a single serial scan of the database and check objects in parallel during the scan of the database. For more information, see <u>Data Integrity Validation</u>.
- The SQL Server 2000 and version 7.0 bulk copy components now transfer data at increased speeds. The **bcp** bulk copy utility can now copy data in parallel from multiple sources into the same file concurrently. For more information, see <u>Parallel Data Loads</u>.
- SQL Server 2000 and version 7.0 now support doing bulk loads directly on the server without transferring the data through a client. This is done using the new BULK INSERT statement, and is the fastest way to get large amounts of data into a table. For more information, see <u>BULK</u><u>INSERT</u>.
- Distribution statistics indicate the selectivity of index keys and are used by the query optimizer to choose the most efficient index when compiling a query. If the statistics are out of date, the optimizer may not generate an optimal execution plan. SQL Server 2000 can be set up to generate distribution statistics automatically, which improves the

effectiveness of the query optimizer. The sampling processes that generate the statistics have also been improved; they can now generate reliable statistics after scanning less data than earlier versions of SQL Server. For more information, see <u>Statistical Information</u>.

- SQL Server 2000 defines OLE DB extensions that OLE DB providers can use to report distribution statistics to the SQL Server 2000 database engine. This allows the engine to more efficiently optimize distributed queries. The Microsoft OLE DB Provider for SQL Server 2000 supports these extensions, improving the performance of distributed queries referencing SQL Server databases. For more information, see <u>Distribution Statistics Requirements for OLE DB Providers</u>.
- SQL Server includes failover cluster support. Two to four Windows NT or Windows 2000 servers can have instances of SQL Server and all access a set of cluster disks holding SQL Server databases and each instance is identified by a single virtual server name. If the server currently processing SQL Server requests fails, one of the other Windows servers starts its SQL Server services, recovers any uncompleted transactions recorded in the database logs, and begins operating in place of the lost server. For more information, see Failover Clustering Architecture.
- SQL Server 2000 introduces log shipping, which can be used to maintain a warm standby server. The transaction logs from a production server are periodically backed up and applied to a warm standby server. If the production server fails, the warm standby server can be brought online in its place. For more information, see Log Shipping.

SQL Server 2000 Databases on the Desktop

The same Microsoft® SQL Server[™] 2000 database engine that supports thousands of concurrent users can also be installed on laptop or desktop computers running either Microsoft Windows® 98, Microsoft Windows NT® Workstation, or Windows 2000 Professional. Two versions of SQL Server 2000 that run on these operating systems are:

• SQL Server 2000 Personal Edition

An edition of SQL Server 2000 used on personal workstations or small workgroup servers. SQL Server 2000 Personal Edition includes the management tools, such as SQL Server Enterprise Manager, that come with both SQL Server 2000 Standard Edition and SQL Server 2000 Enterprise Edition.

• SQL Server 2000 Desktop Engine

A redistributable version of the SQL Server relational database engine, which third-party software developers can include in their applications that use SQL Server to store data. The SQL Server 2000 Desktop Engine is made available as a set of Windows Installer merge modules that can be included in the application setup.

The SQL Server 2000 Desktop Engine does not include graphical management tools; the application distributing the engine is usually coded to perform any needed database administration. You can manage instances of the Desktop Engine from the SQL Server 2000 graphical tools if installed with another edition of SQL Server.

The SQL Server 2000 Desktop Engine includes support for all of the programming APIs and most of the functionality of the other editions of SQL Server 2000. It also includes the SQLServerAgent service for managing scheduled tasks. Although the Desktop Engine does not include the management tools or wizards, applications can fully administer an instance of the Desktop Engine using the SQL Server administration APIs, such as SQL-DMO, the DTS and Replication programming objects, or the general database APIs (such as ADO, OLE DB, and ODBC). Applications can use the general database APIs to

access data in the Desktop Engine, and the Desktop Engine can participate alongside other editions of SQL Server 2000 in DTS transformations and replication plans (except operating as a transactional replication Publisher). For more information about the features supported by the Desktop Engine, see <u>Features Supported by</u> <u>the Editions of SQL Server 2000</u>.

The database engine included in these two versions of SQL Server 2000 is tuned to support the workloads typical of a single user or a small workgroup. The database engine provides desktop users with essentially the same functionality and features as SQL Server 2000 Standard Edition and SQL Server 2000 Enterprise Edition; however, two exceptions are:

- Certain features primarily used in large production databases, such as parallel statement processing and indexed views, are not supported. For more information about the features available in the various editions of SQL Server 2000, see <u>Features Supported by the Editions of SQL Server 2000</u>.
- A concurrent workload governor limits the performance of the database engine in these two editions. The performance of individual Transact-SQL batches is decreased when more than five batches are executed concurrently. The amount each batch is slowed down depends on how many batches over the five-batch limit are executing concurrently, and the amount of data retrieved by the individual batches. As more batches are executed concurrently, and as more data is retrieved by each batch, the more the governor slows down the individual batches. You can use the DBCC CONCURRENCYVIOLATION statement to report how often the concurrent workload governor is activated. For more information, see DBCC CONCURRENCYVIOLATION.

The ease-of-use features of the database engine allow it to run in a laptop or desktop environment with minimal configuration tuning from the user. The database engine automatically configures itself to acquire or free resources, such as memory and disk space, as needed. This means that SQL Server 2000 Personal Edition and SQL Server 2000 Desktop Engine can be run on an enduser laptop or desktop computer without requiring the user or database administrator to constantly tune the database.

SQL Server 2000 Personal Edition and SQL Server 2000 Desktop Engine support the same programming model as SQL Server 2000 Standard Edition and SQL Server 2000 Enterprise Edition. Applications use the same APIs (ADO, OLE DB, ODBC, SQL-DMO, and so on) to access the data in all the editions of SQL Server 2000. The only difference is the set of features supported in the higher-level editions, such as failover clustering or federated database servers, although most of these features are administrative or scalability features that are transparent to most applications.

The database engine used in SQL Server 2000 supports optimizations that maximize performance in small laptop or desktop systems with small amounts of memory:

- The internal data structures of the database, such as mixed extents, significantly reduce the size of small databases, or databases with many small tables.
- When running at its default configuration settings, SQL Server configures itself dynamically to the current resource usage on the computer without the need for tuning commands from the user.
- Many configuration options that had to be set manually in SQL Server version 6.5 or earlier have been replaced with internal logic in the database engine that configures these options automatically based on load.
- It is no longer necessary to update distribution statistics manually; these are updated automatically.
- Database files grow or shrink automatically depending on the amount of data.

SQL Server 2000 replication and the ability of the database engine to attach and detach databases offers good support for mobile and disconnected users with

laptops. These users can periodically connect to a regional or departmental server to resynchronize their database information with the main database through replication. Alternatively, a database can be placed on a compact disc and sent to remote users, where they can simply attach it to their server to get the latest information.

SQL Server 2000 on Windows 98

Microsoft® SQL Server[™] 2000 includes two main types of software that can be run on Microsoft Windows® 98:

• Client software

All users covered by a SQL Server client access license can install the SQL Server client software on a Microsoft Windows 98 computer. The client software can be installed from the compact disc for SQL Server 2000 Enterprise Edition, SQL Server 2000 Standard Edition, or SQL Server 2000 Professional Edition using the SQL Server Setup options **Client-Tools Only** or **Connectivity Only**. The client software is also often installed by applications that use SQL Server to store data. The SQL Server 2000 client software also runs on Microsoft Windows 95.

• Server software

Any user who has purchased SQL Server 2000 Personal Edition can install the server software from that edition on a computer running the Windows 98 operating system. Applications that install the SQL Server 2000 Desktop Edition can do so on Windows 98.

SQL Server Client Software on Windows 98 and Windows 95

SQL Server 2000 client software consists of:

- Utilities for managing SQL Server and performing ad hoc queries of SQL Server databases.
- Connectivity components such as the OLE DB Provider for SQL Server, the SQL Server ODBC driver, and the client Net-Libraries. These are used by any application that connects to an instance of SQL Server.

The SQL Server 2000 client software runs the same on a Windows 98 or Windows 95 computer as it does on a Microsoft Windows NT® or Windows 2000 computer, with the following exceptions:

- The Windows 98 and Windows 95 network redirectors do not provide computer browser support. SQL Server dialog boxes that depend on this feature to get a list of servers do not display a server list on Windows 98 or Windows 95. This includes the **Register Server** dialog box, the Register Server Wizard, and the **Query Analyzer Login** dialog box.
- The SQL Server utilities are not supported on Windows 95. They are supported on Windows 98.
- The SQL Server tools that poll for the state of a server (SQL Server Enterprise Manager, SQL Server Agent) must do so actively using a poll service state interval defined by the user.

SQL Server 2000 Server Components on Windows 98

The SQL Server 2000 Personal Edition and the SQL Server 2000 Desktop Engine are the only editions whose server components can be installed on Windows 98.

When SQL Server 2000 Personal Edition and the SQL Server 2000 Desktop Engine are running on Windows 98 computers, the following features are not available:

- The Named Pipes and Banyan VINES server Net-Libraries cannot be installed on Windows 98. The server NWLink IPX/SPX Net-Library is also not supported on Windows 98. An instance of SQL Server 2000 on a computer running the Windows 98 operating system cannot accept connections using these protocols. Although Windows 98 does not support these server Net-Libraries, it does support the client Net-Libraries. SQL Server clients running on Windows 98 computers can connect to instances of SQL Server on Windows NT or Windows 2000 computers using these protocols.
- Neither the client nor server AppleTalk Net-Libraries are supported on Windows 98 or Windows 95.

- Windows 98 does not support the server functions of the API used for Windows Authentication. Clients cannot connect to an instance of SQL Server on a Windows 98 computer using Windows Authentication.
 Windows 98 and Windows 95 do support the client functions of the API for Windows Authentication. SQL Server clients running on Windows 98 or Windows 95 computers can connect to instances of SQL Server 2000 on Windows NT or Windows 2000 computers using Windows Authentication.
- The server side of using encryption with the Multiprotocol Net-Library is not supported on Windows 98 or Windows 95. Clients cannot connect to an instance of SQL Server 2000 on a Windows 98 or Windows 95 computer using Multiprotocol encryption. Windows 98 and Windows 95 do support the client functions for Multiprotocol encryption, so SQL Server clients running on Windows 98 and Windows 95 computers can connect to instances of SQL Server on Windows NT or Windows 2000 computers using Multiprotocol encryption.
- Windows 98 does not support asynchronous I/O or scatter-gather I/O. Because of this, the database engine cannot use some of the I/O optimizations it uses on Windows NT and Windows 2000 to maximize throughput with many concurrent users.
- On Windows 98, SQL Server manages its memory requests based on the amount of database work being done instead of maintaining virtual memory at a point that minimizes swapping as it does on Windows NT and Windows 2000. For more information, see <u>Memory Architecture</u>.
- Windows 98 does not have a component that corresponds to Window NT or Windows 2000 services. The SQL Server database engine and SQL Server Agent run as executable programs on Windows 98. These SQL Server components cannot be started as services automatically. They can be started by placing a command prompt command in the Windows 98 startup group, but then they run as a separate Microsoft MS-DOS® window.

SQL Server Service Manager is installed in the Windows 98 startup group and operates with the same user interface as it does on Windows NT and Windows 2000.

- Windows 98 does not have event logs. SQL Server uses a SQL Profiler– based mechanism to launch alerts on Windows 98.
- SQL Server Performance Monitor is not available on Windows 98 computers. Performance Monitor counters cannot be implemented for instances of SQL Server 2000 running on Windows 98. Windows 98 and Windows 95 clients cannot monitor the performance counters of an instance of SQL Server running on Windows NT or Windows 2000.

SQL Server 2000 and Windows CE

Microsoft® SQL Server 2000[™] Windows® CE Edition provides a robust relational database engine for Windows CE devices:

• Optimized for Windows CE

Microsoft® SQL Server 2000[™] Windows CE Edition (SQL Server CE) is designed to run efficiently on typical Windows CE devices. The memory footprint for SQL Server CE is approximately 1 MB. SQL Server CE was designed from the ground up to balance size, RDBMS functionality, connectivity and performance.

SQL Server CE is implemented as a set of dynamic-link libraries (DLLs) that operate as an OLE DB CE provider. This allows SQL Server CE to support the ADOCE and OLE DB CE APIs, and also means that multiple applications running at the same time can share a common set of DLLs, thereby saving space.

• Integrated Development Environment

SQL Server 2000 CE is tightly integrated with the Windows CE development environment in a way that leverages the existing skills of SQL Server developers. SQL Server CE supports the ADOCE and OLE DB CE data access APIs in the Windows CE-based versions of Microsoft® Visual Basic[™] and Visual C++[™]. The SQL language and data access APIs used by SQL Server CE applications are generally upwardly compatible with SQL Server applications. Programmers already used to developing SQL Server applications using ADO or OLE DB in other Windows environments can rapidly develop data-aware Windows CE applications using SQL Server CE.

• SQL Server 2000 Interoperability

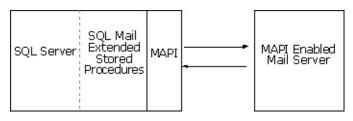
SQL Server CE can exchange data with instances of SQL Server 2000 running on other Windows platforms, giving Windows CE applications access to centrally located data. SQL Server CE supports a wide range of connectivity options to match the connectivity needs of different devices. SQL Server CE can operate as an anonymous merge replication subscriber to publications from instances of SQL Server 2000 running on other Windows platforms. This allows mobile disconnected users who must work autonomously to download data from a central database, work offline, and merge their work back into the central database. Devices that remain connected to the network can use the Remote Data Access feature to:

- Connect to instances of SQL Server on other Windows platforms.
- Execute a SQL statement and pull in the result set as a recordset.
- Optionally, modify the recordset and push the modifications back to the instance of SQL Server on the other Windows platform.

The SQL Server CE connectivity options are tailored for use on wireless networks through networking features such as data compression and messaging to reduce data transmissions, and robust recovery from lost connections.

SQL Server and Mail Integration

Microsoft® SQL Server[™] provides a set of extended stored procedures that allow SQL Server to operate as a workgroup post office for a MAPI-enabled e-mail system.



The computer running SQL Server must be set up as an e-mail client. SQL Server Enterprise Manager is used to assign an e-mail account and password to the SQL Server installation. The mail component of SQL Server can then be enabled to start automatically when the SQL Server Agent service is started. Alternatively, the mail component can be started and stopped at will using either SQL Server Enterprise Manager, or the **xp_startmail**, **xp_stopmail**, and **xp_sendmail** stored procedures.

When the mail component of SQL Server is running, it can be used to:

- Send e-mail from Transact-SQL batches, scripts, stored procedures, and triggers using **xp_send_mail**. The e-mail can be:
 - Message strings.
 - The result set of a query.
 - A Transact-SQL statement or batch to execute.
 - A page for an electronic pager.
- Read e-mail using **sp_processmail**, or a combination of **xp_findnextmessage**, **xp_readmail**, and **xp_deletemail**. The messages sent to SQL Server typically contain a Transact-SQL statement or batch to be executed. The statement is executed and the result set is returned

as a reply e-mail with an optional CC: list.

SQL Server events and alerts can be combined with SQL Mail functionality to build a system in which a server running SQL Server can e-mail or page the relevant administrators automatically if serious conditions arise.

Administration Architecture

Each new version of Microsoft® SQL Server[™] seeks to automate or eliminate some of the repetitive work performed by database administrators. Because database administrators are typically among the people most highly trained in database issues at a site, these improvements allow a valuable resource to spend more time working on database design and application data access issues.

The administration of SQL Server 2000 exhibits these characteristics:

- The SQL Server 2000 database server reduces administration work in many environments by dynamically acquiring and freeing resources. The server automatically acquires system resources such as memory and disk space when needed, and frees the resources when they are no longer required. Although large OLTP systems with critical performance needs are still monitored by trained administrators, SQL Server 2000 can also be used to implement smaller desktop or workgroup databases that do not require constant administrator attention.
- SQL Server 2000 provides a set of graphical tools that allow administrators to perform administrative tasks easily and efficiently.
- SQL Server 2000 provides a set of services that allow administrators to schedule the automatic execution of repetitive tasks.
- Administrators of SQL Server 2000 can program the server to handle exception conditions, or to at least send e-mail or pages to the on-duty administrator.
- SQL Server 2000 publishes the same administration Application Programming Interfaces (APIs) used by the SQL Server utilities. These APIs support all of the administration tasks of SQL Server. This allows developers of applications that use SQL Server 2000 as their data store

to completely shield users from the administration of SQL Server 2000.

DDL and Stored Procedures

Transact-SQL is the language used for all commands sent to Microsoft® SQL Server[™] 2000, from all applications. Transact-SQL contains statements that support all administrative work done in SQL Server. These statements fall into two main categories:

Data Definition Language (DDL)

The SQL language has two main divisions: Data Definition Language (DDL), which is used to define and manage all the objects in an SQL database, and Data Manipulation Language (DML), which is used to select, insert, update, and delete data in the objects defined using DDL. The Transact-SQL DDL used to manage objects such as databases, tables, and views is based on SQL-92 DDL statements, with extensions. For each object class, there are usually CREATE, ALTER, and DROP statements, such as CREATE TABLE, ALTER TABLE, and DROP TABLE. Permissions are controlled using the SQL-92 GRANT and REVOKE statements, and the Transact-SQL DENY statement.

System stored procedures

Administrative tasks not covered by the SQL-92 DDL are typically performed using system stored procedures. These stored procedures have names that start with **sp_** or **xp_**, and they are installed when SQL Server is installed. Some examples of system stored procedures are:

- **sp_addtype** (Defines a user-defined data type.)
- **sp_configure** (Manages the server configuration option settings.)
- **xp_sendmail** (Sends an e-mail or page.)

SQL Server 2000 also exposes the SQL-DMO, SQL-NS, DTS, and Replication Component APIs. These are all comprised of OLE Automation objects that encapsulate either DDL or system stored procedures. When an application calls one of the objects, the object actually translates the request to one or more Transact-SQL DDL or system stored procedure statements that are then sent to the server.

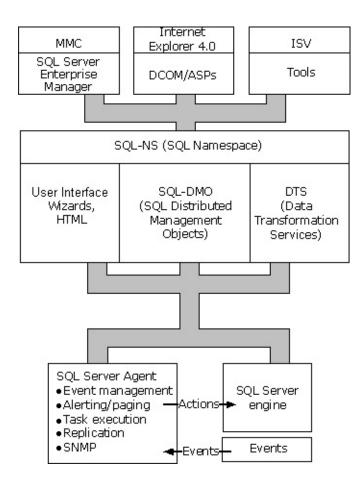
SQL Distributed Management Framework

The SQL Distributed Management Framework (SQL-DMF) is an integrated framework of objects, services, and components used to manage Microsoft® SQL Server[™] 2000. SQL-DMF provides a flexible and scalable management framework that is adaptable to the requirements of an organization. It lessens the need for user-attended maintenance tasks (such as database backup and alert notification) by providing services that interact directly with SQL Server 2000.

The key components of SQL-DMF support the proactive management of the instances of SQL Server on your network by allowing you to define:

- All SQL Server objects and their permissions.
- Repetitive administrative actions to be taken at specified intervals or times.
- Corrective actions to be taken when specific conditions are detected. The corrective actions can either be tasks defined to resolve the issue, or alerts by pages or e-mail to people who can resolve the issue.

This illustration shows the main components of SQL-DMF.



SQL-DMF Applications

There are three main classes of applications that use SQL-DMF. These applications provide the interfaces for users managing Microsoft® SQL ServerTM 2000:

SQL Server Tools

The SQL Server 2000 tools that manage SQL Server database objects use the SQL-DMO API. The primary SQL Server tool that uses SQL-DMF is SQL Server Enterprise Manager. SQL Server Enterprise Manager supplies the primary interface for users who are administering instances of SQL Server on the network. Also, the SQL Query Analyzer contains features (such as an object browser) related to listing and managing database objects. These features use SQL-DMF.

COM+ applications and Active Server Pages

The SQL-DMF APIs can be used in COM+ applications and Web applications, such as Active Server Pages (ASP).

Applications and ISV tools

Applications, written either in-house or by independent software vendors (ISVs), can use the SQL-DMF APIs to administer and configure instances of SQL Server. This allows applications to shield the administration of SQL Server from their users if the application has chosen to embed SQL Server as its data storage mechanism. ISVs who produce tools for managing server applications also use the SQL-DMF APIs to build features for managing SQL Server into their tools.

SQL-DMF APIs

Applications can use one of three APIs to access the core functionality of SQL-DMF: SQL Namespace, SQL Distributed Management Objects, and Data Transformation Services. These APIs are implemented as sets of dual-interface COM interfaces.

SQL Distributed Management Objects

The SQL Distributed Management Objects (SQL-DMO) API is composed of a set of objects that encapsulate the administrative attributes of the entities, such as tables, users, and views, found in Microsoft® SQL Server[™] databases. SQL-DMO abstracts the use of DDL, system stored procedures, registry information, and operating-system resources. SQL-DMO can be used to program all administration and configuration tasks in SQL Server.

Data Transformation Services

The Data Transformation Services (DTS) API exposes the services provided by SQL Server to aid in building data warehouses and data marts. These services provide the ability to transfer and transform data between heterogeneous OLE DB and ODBC data sources. Data from objects or the result sets of queries can be transferred at regularly scheduled times or intervals, or on an as-required basis.

Windows Management Instrumentation

The SQL Server 2000 compact disc contains support for a new API that will allow you to administer instances of SQL Server using Windows Management Instrumentation (WMI). WMI is a scalable Windows 2000 component with an object-oriented API that lets management applications and scripts monitor, configure, and control the operating system and devices, services, and applications in a Windows network. Using standard Windows security, WMI allows only properly authorized users to manage the system. WMI core components are also available for Windows NT® 4.0, Windows® 95, and Windows 98. For more information about the WMI support for these operating systems, see the MSDN® page at <u>Microsoft Web site</u>.

A component, such as SQL Server, enables WMI support by supplying a WMI

provider and defining a WMI class schema. The schema models the objects in the component that can be managed using WMI. SQL Server 2000 includes a SQL Server WMI provider and a schema class model that maps instances of SQL Server 2000 to WMI classes. The SQL Server WMI schema models objects such as databases and tables. The SQL Server WMI implementation provides management functions such as:

- Create, change, or delete managed objects. For example, create a database.
- Administer managed objects. For example, back up databases and logs.
- Enumerate managed objects. For example, list all the tables in a database.
- Retrieve information about a specific managed object. For example, determine whether full-text indexing is enabled on the **Customers** table.
- Query managed objects that meet a specific criterion. For example, list all encrypted stored procedures.
- Execute methods defined for managed objects. For example, execute a method that bulk copies data from a table.
- Generate events when a managed object is created, changed, or deleted (for example, raise an event when a database option is changed).
- Describe relationships between managed objects (for example, identify which logins are authorized to access a database).

All WMI data is available remotely and is fully scriptable. The SQL Server 2000 WMI implementation maps over the SQL-DMO API, but does not support the management of replication. The SQL Server WMI implementation can be used with SQL Server 7.0.

The SQL Server WMI support is not installed by SQL Server 2000 Setup. All of the WMI materials, including a separate setup and documentation, are included in the folder \x86\OTHER\wmi on the SQL Server 2000 compact disc.

SQL Namespace

The SQL Namespace (SQL-NS) API exposes the user interface (UI) elements of SQL Server Enterprise Manager. This allows applications to include SQL Server Enterprise Manager UI elements such as dialog boxes and wizards.

See Also

Developing SQL-DMO Applications
Programming DTS Applications
Programming SQL-NS Applications

SQL Server Agent

SQL Server Agent runs on the server running instances of Microsoft® SQL Server[™] 2000 or earlier versions of SQL Server. SQL Server Agent is responsible for:

- Running SQL Server tasks scheduled to occur at specific times or intervals.
- Detecting specific conditions for which administrators have defined an action, such as alerting someone through pages or e-mail, or a task that will address the conditions.
- Running replication tasks defined by administrators.

SQL Server Agent is similar to an auxiliary operator responsible for handling the repetitive tasks and exception handling conditions defined through the other SQL-DMF components.

See Also

DTS Overview SQL Namespace API SQL-DMO API

SQLServerAgent Service

Graphical Tools

Microsoft® SQL Server[™] 2000 includes many graphical utilities that allow users, programmers, and administrators to efficiently:

- Administer and configure SQL Server.
- Determine the catalog information in a copy of SQL Server.
- Design and test queries for retrieving data.

In addition to these tools, SQL Server contains several wizards to walk administrators and programmers through the steps needed to perform more complex administrative tasks.

SQL Server Enterprise Manager

SQL Server Enterprise Manager is the primary administrative tool for Microsoft® SQL Server[™] 2000 and provides a Microsoft Management Console (MMC)–compliant user interface that allows users to:

- Define groups of servers running SQL Server.
- Register individual servers in a group.
- Configure all SQL Server options for each registered server.
- Create and administer all SQL Server databases, objects, logins, users, and permissions in each registered server.
- Define and execute all SQL Server administrative tasks on each registered server.
- Design and test SQL statements, batches, and scripts interactively by invoking SQL Query Analyzer.
- Invoke the various wizards defined for SQL Server.

MMC is a tool that presents a common interface for managing different server applications in a Microsoft Windows® network. Server applications provide a component called an MMC snap-in that presents MMC users with a user interface for managing the server application. SQL Server Enterprise Manager is the Microsoft SQL Server 2000 MMC snap-in.

To launch SQL Server Enterprise Manager, select the Enterprise Manager icon in the Microsoft SQL Server program group. On computers running Windows 2000, you can also launch SQL Server Enterprise Manager from Computer Management in Control Panel. MMC snap-ins launched from Computer Management do not have the ability to open child windows enabled by default. You may have to enable this option to use all the SQL Server Enterprise Manager features.

Note If you register additional SQL servers in Computer Management, and then either close Computer Management or connect to another computer, the servers will no longer appear in Computer Management. The registered servers will appear in SQL Server Enterprise Manager.

See Also

<u>How to launch SQL Server Enterprise Manager in the Computer Management</u> <u>console (Windows)</u>

How to enable child windows (Enterprise Manager)

Overview of the SQL Server Tools

SQL Query Analyzer

SQL Query Analyzer is a graphical user interface for designing and testing Transact-SQL statements, batches, and scripts interactively. SQL Query Analyzer can be called from SQL Server Enterprise Manager.

SQL Query Analyzer offers:

- A Free-form text editor for keying in Transact-SQL statements.
- Color-coding of Transact-SQL syntax to improve the readability of complex statements.
- Object browser and object search tools for easily finding the objects in a database and the structure of the objects.
- Templates that can be used to speed development of the Transact-SQL statements for creating SQL Server objects. Templates are files that include the basic structure of the Transact-SQL statements needed to create objects in a database.
- An interactive debugger for analyzing stored procedures.
- Results presented in either a grid or a free-form text window.
- Graphical diagram of the showplan information showing the logical steps built into the execution plan of a Transact-SQL statement.

This allows programmers to determine what specific part of a poorly performing query is using a lot of resources. Programmers can then explore changing the query in ways that minimize the resource usage while still returning the correct data.

• Index Tuning Wizard to analyze a Transact-SQL statement and the

tables it references, to see if adding additional indexes will improve the performance of the query.

See Also

Analyzing a Query

Index Tuning Wizard

Overview of SQL Query Analyzer

Windows 2000 System Monitor

The Windows 2000 System Monitor (Windows NT Performance Monitor) is a tool for monitoring resource usage on a computer running Microsoft® Windows NT® or Microsoft Windows® 2000. Users can set up charts that present resource usage data in graphical form. The Windows System Monitor has many different counters, each of which measures some resource on the computer.

The Windows System Monitor is extensible so server applications can add their own performance counters. Microsoft SQL Server[™] 2000 adds counters to Windows System Monitor to track items such as:

- SQL Server I/O.
- SQL Server memory usage.
- SQL Server user connections.
- SQL Server locking.
- Replication activity.

See Also

Monitoring with Windows Performance Monitor

Import and Export Data

The **Import and Export Data** item in the Microsoft® SQL Server[™] program group starts the Data Transformation Services (DTS) Import/Export Wizard. The wizard walks users through the DTS functions of importing, exporting, validating, and transforming data and objects between heterogeneous OLE DB and ODBC data sources.

See Also

DTS Import/Export Wizard

SQL Profiler

SQL Profiler is a tool that captures Microsoft® SQL Server[™] 2000 events from a server. The events are saved in a trace file that can later be analyzed or used to replay a specific series of steps when trying to diagnose a problem. SQL Profiler is used for activities such as:

- Stepping through problem queries to find the cause of the problem.
- Finding and diagnosing slow-running queries.
- Capturing the series of SQL statements that lead to a problem. The saved trace can then be used to replicate the problem on a test server where the problem can be diagnosed.
- Monitoring the performance of SQL Server to tune workloads.

SQL Profiler also supports auditing the actions performed on instances of SQL Server. Audits record security-related actions for later review by a security administrator. SQL Server 2000 auditing meets C2 security certification requirements.

See Also

Monitoring with SQL Profiler

Auditing SQL Server Activity

SQL Server Service Manager

SQL Server Service Manager is used to start, stop, and pause the Microsoft® SQL Server[™] 2000 components on the server. These components run as services on Microsoft Windows NT® or Microsoft Windows® 2000 and as separate executable programs on Microsoft Windows 95 and Microsoft Windows 98:

• SQL Server service

Implements the SQL Server database engine. There is one SQL Server service for each instance of SQL Server running on the computer.

• SQL Server Agent service

Implements the agent that runs scheduled SQL Server administrative tasks. There is one SQL Server Agent service for each instance of SQL Server running on the computer.

• Microsoft Search service (Windows NT and Windows 2000 only)

Implements the full-text search engine. There is only one service, regardless of the number of SQL Server instances on the computer.

• **MSDTC service** (Windows NT and Windows 2000 only)

Manages distributed transactions. There is only one service, regardless of the number of SQL Server instances on the computer.

• **MSSQLServerOlAPService service** (Windows NT and Windows 2000 only)

Implements SQL Server 2000 Analysis Services. There is only one service, regardless of the number of SQL Server instances on the computer.

Operating the SQL Server Service Manager

SQL Server Service Manager is a taskbar application and follows the standard behavior of taskbar applications. When minimized, the **SQL Server Service**

Manager icon appears in the area of the taskbar clock on the right of the taskbar. To get a menu that includes all the tasks SQL Server Service Manager supports, right-click the taskbar item.

To maximize SQL Server Service Manager, double-click the icon. When SQL Server Service Manager is maximized, clicking the close button of the SQL Server Service Manager window does not terminate the application; it only minimizes SQL Server Service Manager to the taskbar. To terminate SQL Server Service Manager, right-click the **SQL Server Service Manager** icon on the taskbar, and then select the **File/Exit** menu item.

See Also

Starting, Pausing, and Stopping SQL Server

Client Network Utility

The Client Network utility is used to manage the client Net-Libraries and define server alias names. It can also be used to set the default options used by DB-Library applications.

Most users will never need to use the Client Network utility. To connect to Microsoft® SQL Server[™] 2000, users can specify only the network name of the server on which SQL Server is running, and optionally the name of the instance of SQL Server.

In some cases, an instance of SQL Server may be configured to listen on alternate network addresses. If this is done, client applications connecting to that instance must explicitly specify the alternate address. While applications could specify the alternate addresses on each connection request, it is easier to use the Client Network utility to set up an alias specifying the alternate addresses. Applications can then specify the alias name in place of the server network name in their connection requests.

See Also

Communication Components Managing Clients

Server Network Utility

The Server Network utility is used to manage the server Net-Libraries. This utility is used to specify:

- The network protocol stacks on which an instance of Microsoft® SQL Server[™] 2000 listens for client requests.
- The sequence in which server Net-Libraries are considered when establishing connections from applications.
- New network addresses that an instance of Microsoft SQL Server 2000 listens on.

Most administrators will never need to use the Server Network utility. They will specify during setup the server Net-Libraries on which SQL Server listens.

See Also

Communication Components Configuring Network Connections

Miscellaneous Utilities

The ODBC administrator utility and Services utility are also used to manage parts of Microsoft[®] SQL Server[™] 2000:

ODBC Administrator

The ODBC Administrator utility is used to add, delete, and edit ODBC data sources for all ODBC drivers on the computer, including data sources for the SQL Server ODBC driver. It can also be used to list the versions of all the ODBC drivers installed on the computer. In Microsoft Windows NT®, Microsoft Windows® 95, and Microsoft Windows 98, the ODBC utility is in Control Panel. In Microsoft Windows 2000, the utility is named Data Source (ODBC) utility and is in the Administrative Tools folder in Control Panel.

Services (Windows NT and Windows 2000)

The Services application can be used to start, pause, and stop Microsoft Windows NT or Windows 2000 services, including the services managed by SQL Server Service Manager. In Windows NT, the Services utility is in Control Panel. In Windows 2000, the utility is in the Administrative Tools folder in Control Panel.

SQL Server also installs several command prompt utilities used when building .cmd files to work with SQL Server. For more information, see <u>Getting Started</u> <u>with Command Prompt Utilities</u>.

Automated Administration Architecture

Microsoft® SQL Server[™] 2000 provides features that allow administrators to program the server to administer itself for many repetitive actions or exception conditions. This frees the administrators to spend more time on activities such as designing databases and advising programmers on efficient database access coding techniques. Applications from any vendor can choose SQL Server as their data storage component and minimize the administrative requirements of customers by automating administrative tasks.

These automation features are not limited to database administration tasks such as scheduling backups. They can also be used to help automate the business practices that the database supports. Applications can be scheduled to run at specific times or intervals. Specific conditions detected in the system can be used to trigger these applications if they need to be executed before the next scheduled time.

The features that support the automation of administrative tasks are:

SQL Server Agent

SQL Server Agent is a separate executable program that executes administrative jobs and alerts defined by the system administrators. SQL Server Agent runs as a service named SQLServerAgent on computers running Microsoft Windows NT® or Windows® 2000, and as an executable file on computers running Microsoft Windows 95 or Microsoft Windows 98.

Jobs

A job defines an administrative task. Each job has one or more steps; each step specifies a Transact-SQL statement, Windows command, executable program, replication agent, or Microsoft ActiveX® script. Jobs can be run once, scheduled to run at periodic intervals, or specified to run when the server is idle.

Jobs enable administrators to define when administrative tasks are performed. Each job can combine various operating system commands, Transact-SQL statements, stored procedures, and applications to complete complex administrative functions. Each job step can be very complex. For example, a Windows command could be a command or batch file that contains many commands. The Transact-SQL statement executed by a step could be a stored procedure containing many Transact-SQL statements.

SQL Server Agent runs these tasks at the specified times, without the need for human intervention. Complex procedures with error-checking logic can be designed into each job to address the most likely conditions the job would encounter. These capabilities result in the ability to build complex, robust jobs that run all periodic maintenance.

Events and alerts

Each instance of SQL Server 2000 running on Windows NT or Windows 2000 records significant events in the Windows NT or Windows 2000 application log. Each entry in the log is called an event. SQL Server administrators can define alerts that specify a job to be run when a specific event occurs. SQL Server Agent compares the SQL Server events in the application log against the alerts defined by administrators. If a match is made, the job specified in the alert is executed.

Windows 95 and Windows 98 do not have event logs. Installations of SQL Server Professional edition running on Windows 95 or Windows 98 use a SQL Profiler–based mechanism to communicate events to SQL Server Agent.

SQL Server creates events for errors with a severity of 19 or higher. Events are also raised if a RAISERROR statement is executed using the WITH LOG clause, or the **xp_logevent** system stored procedure is executed. This allows Transact-SQL scripts, triggers, stored procedures, and applications to raise events that could fire a job.

Operators

Operators are e-mail and page addresses defined to SQL Server for use in alerts. An alert can be defined that either e-mails or pages a specific person. Instances of SQL Server running on Windows NT or Windows 2000 can also use the Windows NT or Windows 2000 **net send** command to send a network message to a Windows user or group.

Triggers

Triggers are used to enforce business logic. Triggers can be integrated with

automated administrative tasks by using either RAISERROR or **xp_logevent** to generate an event that fires an alert. For example, a retail company has an inventory database, and all of its suppliers accept electronic orders. Every night, a scheduled job executes an application that reviews all inventory levels and, using guidelines established by management, either places orders with preferred providers for items in short supply or prints a report for the purchasing agents. This could be backed up by a DELETE trigger on the parts table that fires a similar job for emergency orders if heavy sales deplete the inventory during the day.

See Also

Automating Administrative Tasks

Enforcing Business Rules with Triggers

Backup/Restore Architecture

The backup and restore components of Microsoft® SQL Server[™] 2000 allow you to create a copy of a database. This copy is stored in a location protected from the potential failures of the server running the instance of SQL Server. If the server running the instance of SQL Server fails, or if the database is somehow damaged, the backup copy can be used to re-create, or restore, the database.

SQL Server 2000 provides these sophisticated backup and restore capabilities:

- Options for how a database is backed up and restored:
 - A full database backup is a full copy of the database.
 - A transaction log backup copies only the transaction log.
 - A differential backup copies only the database pages modified after the last full database backup.
 - A file or filegroup restore allows the recovery of just the portion of a database that was on the failed disk.

These options allow backup and restore processes to be tailored to how critical the data in the database is. Noncritical databases that can be easily re-created from some other source may have no backups, other databases may have simple backups that can re-create the database to the night before a failure, and critical databases may have sophisticated backups that will restore the database right up to the point of failure.

• Control with the BACKUP and RESTORE statements.

Users can execute the BACKUP and RESTORE statements directly from applications, Transact-SQL scripts, stored procedures, and triggers. It is more common, however, to use SQL Server Enterprise Manager to define a backup schedule, and then let SQL Server Agent run the backups automatically according to the schedule. The Database Maintenance Plan Wizard can be used to define and schedule a full set of backups for each database. This fully automates the backup process, requiring minimal or no operator action.

• Maintenance of a set of backup history tables in the **msdb** database.

The backup history tables record the backups for each database. If a database has to be restored, the **Restore Database** dialog box in SQL Server Enterprise Manager presents the user with a list of all the backups available for the database. The **Restore Database** dialog box also has logic to display which set of the backups in the history can be used to restore the database in the shortest possible time. When the dialog box is displayed, the backups needed to restore the database are checked. If a user knows that one of the backups is not available (for example, if a tape cartridge was damaged or lost), the user can deselect that backup, and SQL Server Enterprise Manager calculates a new restore process. When the user agrees with the restore process, SQL Server Enterprise Manager restores the database, prompting for tapes as needed.

• Backups that can be performed while the database is in use, allowing backups to be made of systems that must run continuously.

The backup processing and internal data structures of SQL Server 2000 are structured so that backups maximize their rate of data transfer with minimal effect on transaction throughput.

• Fast data transfer rates for backup and restore operations, making SQL Server 2000 capable of supporting very large databases (VLDB).

The data structures in SQL Server 2000 databases and the backup and restore algorithms support high data transfer rates for backup and restore operations. SQL Server backup and restore operations can also run in parallel against multiple backup files or tape drives, which further increases the backup and restore data transfer rates.

• RESTORE statement re-creates the database automatically if necessary.

This eliminates the need to execute a separate CREATE DATABASE or CREATE DATABASE FOR LOAD statement if the database does not exist at the time the RESTORE statement is executed.

- Interrupted backup and restore operations started near the point of the interruption when restarted.
- Verification of a SQL Server 2000 backup before an attempt to restore the database. This includes verifying that the collation of the database is supported by the instance of SQL Server.

Backup and restore processes should be planned together. The administrators must first determine the criticality of the data in the database. They must determine if it is acceptable to just restore the database to a point such as the night before the failure, or if the database must be restored to a point as close as possible to the time of failure. They must also determine how long the database can be unavailable, whether it must be brought back online as quickly as possible, or if it does not need to be restored immediately.

After the restore requirements are determined, the administrators can then plan a backup process that maintains a set of backups that will meet the restore requirements. The administrators can choose the backup processes that can be performed with the minimum effect on the system as it runs, yet still meet the restore requirements. Based on the resource requirements, the administrators also choose the recovery model for the database. The recovery model balances logging overhead against the criticality of fully recovering the data. The recovery models are:

• Full

The data is critical and must be recoverable to the point of failure. All data modifications are logged. All SQL Server 2000 recovery options are available.

• Bulk-logged

Certain bulk operations (bulk copy operations, SELECT INTO, text processing) can be replayed if necessary, so these operations are not fully logged. Can only recover to the end of the last database or log backup.

• Simple

All data modifications made since last backup are expendable, or can be

redone. Lowest logging overhead, but cannot recover past the end of the last backup.

See Also

Backing Up and Restoring Databases

Backup Devices

Backups created in Microsoft® SQL Server[™] 2000 and SQL Server version 7.0 are stored using the Microsoft Tape Format (MSTF). MSTF is not specific to tapes; it can also be used for backing up to either disks or named pipes. Each time a SQL Server backup is performed, it forms a backup set. This backup set is stored in an MSTF unit called a media. MSTF media can store backup sets from different software.

Media Header	Backup Set #1 SQL Server	Backup Set #2 Windows 2000	Backup Set #3 Windows 2000	Backup Set #4 SQL Server	•••
--------------	-----------------------------	-------------------------------	-------------------------------	-----------------------------	-----

Using the MSTF format allows SQL Server to work with administrative utilities and products from other vendors that manage MSTF format backups. SQL Server backup sets can share media, such as MSTF tape drives, with backup sets from other server software. SQL Server does not compress its backup sets, but uses the compression provided on MSTF backup devices.

See Also

Using Backup Media

Types of Backup and Restore Processes

Microsoft® SQL Server[™] 2000 and SQL Server version 7.0 supports four types of backups. These can be combined to form many different types of backup and restore processes, each customized to the availability requirements of the database. The four types are:

- Database
- Transaction log
- Differential
- File and filegroup

See Also

Designing a Backup and Restore Strategy

Database Backup and Restore

A database backup creates a copy of the full database. Not all pages are copied to the backup set, only those actually containing data. Both data pages and transaction log pages are copied to the backup set.

A database backup set is used to re-create the database as it was at the time the BACKUP statement completed. If only database backups exist for a database, it can be recovered only to the time of the last database backup taken before the failure of the server or database.

See Also

Database Backups

Transaction Log Backup and Restore

A transaction log backup makes a copy of only the log file. A log file backup by itself cannot be used to restore a database. A log file is used after a database restore to recover the database to the point of the original failure. For example, a site performs a database backup on Sunday night and a log backup on each of the other nights. If one of the data disks for the database is lost at 2:30 P.M. Tuesday, the site can:

- 1. Back up the current transaction log.
- 2. Restore the database backup from Sunday night.
- 3. Restore the log backup from Monday night to roll the database forward.
- 4. Restore the log backup taken after the failure. This will roll the database forward to the time of the failure.

A transaction log recovery requires an unbroken chain of transaction log backups from the time of the database backup to the time of the failure.

See Also

Transaction Log Backups

Differential Backup and Restore

A differential backup creates a copy of all the pages in a database modified after the last database backup. Differential logs are used primarily in heavily used systems where a failed database must be brought back online quickly. Differential backups are smaller than full database backups; therefore, they have less of an effect on the system while they run.

For example, a site executes a full database backup on Sunday night. A set of transaction log backups is made every four hours during the day, with the backups from one day overwriting the backups from the day before. Each night the site makes a differential backup. If one of the data disks for the database fails at 9:12 A.M. on Thursday, the site can:

- 1. Back up the current transaction log.
- 2. Restore the database backup from Sunday night.
- 3. Restore the differential backup from Wednesday night to roll the database forward to that point.
- 4. Restore the transaction log backups from 4:00 A.M. and 8:00 A.M. to roll the database forward to 8:00 A.M.
- 5. Restore the log backup taken after the failure. This will roll the database forward to the time of the failure.

See Also

Differential Database Backups

File and Filegroup Backup and Restore

Microsoft® SQL Server[™] 2000 supports backing up or restoring individual files or file groups within a database. This is a relatively sophisticated backup and restore process usually reserved for very large databases (VLDB) with high availability requirements. If the time available for backups is not long enough to support backing up the full database, subsets of the database can be backed up at different times.

For example, it takes three hours for a site to back up a database, and backups can be performed only during a two-hour period each day. The site can back up half the files or file groups on one night and half the next. If a disk holding database files or filegroups fails, the site can restore just the lost files or filegroups. The site must also be making transaction log backups, and must restore all transaction log backups made after the file or filegroup backup.

File and filegroup restores can also be made from a full database backup set. This allows for a quicker recovery because only the damaged files or filegroups are restored in the first step, not the entire database.

See Also

Using File Backups

Fuzzy Backup and Restore Operations

Microsoft[®] SQL Server[™] 2000 and SQL Server version 7.0 use industrystandard fuzzy backup algorithms. These new algorithms provide several significant benefits for users:

- The BACKUP statement runs faster and has less effect on users modifying data while the statement is processing.
- The RESTORE statement is faster.

A RESTORE operation restores the database to the state it was in at the time the BACKUP statement finished. In SQL Server version 6.5 and earlier, a LOAD statement restored a database to the state it was in at the time the DUMP statement started.

In a SQL Server fuzzy backup and restore operation:

- Extents containing data are written to the backup set without regard to synchronizing pages being modified by users during the backup. This significantly reduces the effect the backup has on current users. It also allows the backup to copy pages serially. The elimination of any random reads speeds the backup process in heavily used systems. It does mean, however, that the pages in the backup are stored in an inconsistent, unrecovered state.
- The transaction log is copied as part of the backup.

A RESTORE statement:

- Creates the database if it does not exist, and initializes the extents in the database. This step is bypassed if the database exists when the RESTORE statement is executed.
- Copies in the extents found in the backup set. The process is fast because all the extents are in a serial sequence. Extents not found in the

backup set are ignored; they are not initialized as empty extents.

• Uses the transaction log to recover the database. The database modifications recorded in the log are rolled forward to the end of the log, and then any incomplete transactions are rolled back. This returns the database to a consistent, recovered state that corresponds to the state the database was in at the time the BACKUP statement completed.

Parallel Backup and Restore

Parallel backup and restore operations improve the capability of Microsoft® SQL Server[™] 2000 to manage very large databases. The BACKUP and RESTORE statements use parallel I/O in a number of ways:

- If a database has files on several disk devices, BACKUP uses one thread per disk device to read the extents from the database.
- If a backup set is stored on multiple backup devices, both the BACKUP and RESTORE statements use one thread per backup device.
- If a database is defined with files on several disk drives, and RESTORE has to create the database, RESTORE uses one thread per disk device while it is initializing the database.

Data Import/Export Architecture

Microsoft® SQL Server[™] 2000 has several components that support importing and exporting data:

Data Transformation Services

Data Transformation Services (DTS) can be used to import and export data between heterogeneous OLE DB and ODBC data sources. A DTS package is defined that specifies the source and target OLE DB data sources; the package can then be executed on an as-required basis or at scheduled times or intervals. A single DTS package can cover multiple tables. DTS packages are also not limited to transferring data straight from one table to another, as the package can specify a query as the source of the data. This allows packages to transform data, such as running a query that returns aggregate summary values instead of the raw data.

Replication

Replication is used to create copies of data in separate databases and keep these copies synchronized by replicating modifications in one copy to all the others. If it is acceptable for each site to have data that may be a minute or so out of date, replication allows the distribution of data without the overhead of requiring distributed transactions to ensure all sites have an exact copy of the current data. Replication can therefore support the distribution of data for a relatively low cost in network and computing resources.

Bulk copying

The bulk copy feature of SQL Server allows for the efficient transfer of large amounts of data. Bulk copying transfers data into or out of one table at a time. Bulk copying supports the following bulk copy transfers:

- From one SQL Server table or view to another table or view.
- From a SQL Server table or view into a data file, such as a text file or tab-delimited file.

- The result set of a query into a table, view, or data file.
- The contents of a data file into a table or view.

There are several ways the bulk copy feature can be used:

- The **bcp** command prompt utility.
- The OLE DB Provider for SQL Server has a provider-specific **IRowsetFastLoad** interface for bulk copies.
- The SQL Server ODBC Driver supports a set of bulk copy functions.
- The Transact-SQL BULK INSERT statement. This is the fastest of the bulk copy methods. The data file is accessed directly from SQL Server itself, eliminating the overhead of communicating data from a client application to the server.
- The DB-Library API supports a set of bulk copy functions.

Distributed queries

Distributed queries allow Transact-SQL statements to reference data in an OLE DB data source. The OLE DB data sources can be another instance of SQL Server, or a heterogeneous data source such as Microsoft Access or Oracle. SELECT INTO and INSERT statements can be used to:

- Export data from a SQL Server database to an OLE DB data source.
- Import data from an OLE DB data source into SQL Server.

See Also

Distributed Queries

Importing and Exporting Data

DTS Overview

Replication Overview

Data Integrity Validation

Transact-SQL has a set of DBCC statements used to verify the integrity of a database. The DBCC statements in Microsoft® SQL Server[™] 2000 and SQL Server version 7.0 contain several improvements to the DBCC statements used in SQL Server version 6.5:

- The need to run the statements is reduced significantly. Two architectural changes in SQL Server have improved the robustness of the databases to the point that you do not have to verify their integrity:
 - The database engine has fail-fast logic to detect potential errors closer to the time they originate. This means errors are less likely to persist long enough to cause problems in a database.
 - The data structures in the database are simpler. This means they are easier to manage and less likely to have errors.
- It is not necessary to run DBCC validation statements as part of your normal backup or maintenance procedures. You should run them as part of a system check before major changes, such as before a hardware or software upgrade, or after a hardware failure. You should also run them if you suspect any problems with the system.
- SQL Server 2000 introduces a new PHYSICAL_ONLY option that allows a DBCC statement to run faster by only checking for the types of problems likely to be generated by a hardware problem. Run a DBCC check with PHYSICAL_ONLY if you suspect a hardware problem on your database server.
- The DBCC statements themselves also run significantly faster. Checks of complex databases typically run 8 to 10 times faster, and checks of some individual objects have run more than 300 times faster. In SQL Server 6.5, DBCC CHECKDB processed the tables serially. For each table, it first checked the structure of the underlying data and then

checked each index individually. This resulted in a very random pattern of reads. In SQL Server 2000, DBCC CHECKDB performs a serial scan of the database while performing parallel checks of multiple objects as it proceeds. SQL Server 2000 also takes advantage of multiple processors when running parallel DBCC statements.

- The level of locks required by SQL Server 2000 DBCC statements are much lower than in SQL Server 7.0. DBCC statements can now be run concurrently with data modification statements, significantly lowering their impact on users working in the database.
- The SQL Server 2000 DBCC statements can repair minor problems they might encounter. The statements have the option to repair certain errors in the B-tree structures of indexes, or errors in some of the allocation structures.

See Also

DBCC

Optimizing DBCC Performance

Replication Architecture

Replication is a set of technologies that allows you to keep copies of the same data on multiple sites, sometimes covering hundreds of sites.

Replication uses a publish-subscribe model for distributing data:

- A Publisher is a server that is the source of data to be replicated. The Publisher defines an article for each table or other database object to be used as a replication source. One or more related articles from the same database are organized into a publication. Publications are convenient ways to group related data and objects that you want to replicate together.
- A Subscriber is a server that receives the data replicated by the publisher. The Subscriber defines a subscription to a particular publication. The subscription specifies when the Subscriber receives the publication from the Publisher, and maps the articles to tables and other database objects in the Subscriber.
- A Distributor is a server that performs various tasks when moving articles from Publishers to Subscribers. The actual tasks performed depend on the type of replication performed.

Microsoft® SQL Server[™] 2000 also supports replication to and from heterogeneous data sources. OLE DB or ODBC data sources can subscribe to SQL Server publications. SQL Server can also receive data replicated from a number of data sources, including Microsoft Exchange, Microsoft Access, Oracle, and DB2.

Replication Types

SQL Server 2000 uses three types of replication:

Snapshot replication

Snapshot replication copies data or database objects exactly as they exist at

any moment. Snapshot publications are typically defined to happen on a scheduled basis. The Subscribers contain copies of the published articles as they existed at the last snapshot. Snapshot replication is used where the source data is relatively static, the Subscribers can be slightly out of date, and the amount of data to replicate is small.

Transactional replication

In transactional replication, the Subscribers are first synchronized with the Publisher, typically using a snapshot, and then, as the publication data is modified, the transactions are captured and sent to the Subscribers. Transactional integrity is maintained across the Subscribers by having all modifications be made at the Publisher, and then replicated to the Subscribers. Transactional replication is used when data must be replicated as it is modified, you must preserve the transactions, and the Publishers and Subscribers are reliably and/or frequently connected through the network.

Merge replication

Merge replication lets multiple sites work autonomously with a set of Subscribers, and then later merge the combined work back to the Publisher. The Subscribers and Publisher are synchronized with a snapshot. Changes are tracked on both the Subscribers and Publishers. At some later point, the changes are merged to form a single version of the data. During the merge, some conflicts may be found where multiple Subscribers modified the same data. Merge replication supports the definition of conflict resolvers, which are sets of rules that define how to resolve such conflicts. Custom conflict resolver scripts can be written to handle any logic that may be needed to resolve complex conflict scenarios properly. Merge replication is used when it is important for the Subscriber computers to operate autonomously (such as a mobile disconnected user), or when multiple Subscribers must update the same data.

Configuring and Managing Replication

SQL Server 2000 provides several mechanisms for defining and administering replication:

• SQL Server Enterprise Manager supports configuring and monitoring replication.

- SQL-DMO interfaces for programmatically configuring and monitoring replication.
- Programmatic interfaces for replicating data from heterogeneous data sources.
- Microsoft ActiveX[®] controls for embedding replication functionality in custom applications.
- Scripting replication using Transact-SQL system stored procedures.

See Also

Replication Overview Snapshot Replication How Snapshot Replication Works Merge Replication How Merge Replication Works Transactional Replication How Transactional Replication Works Replication and Heterogeneous Data Sources Replication Tools

Data Warehousing and Online Analytical Processing

Microsoft® SQL Server[™] 2000 provides components that can be used to build data warehouses or data marts. The data warehouses or data marts can be used for sophisticated enterprise intelligence systems that process queries required to discover trends and analyze critical factors. These systems are called online analytical processing (OLAP) systems. The data in data warehouses and data marts is organized differently than in traditional transaction processing databases.

Enterprise-level relational database management software, such as SQL Server 2000, was designed originally to centrally store the data generated by the daily transactions of large companies or government organizations. Over the decades, these databases have grown to be highly efficient systems for recording the data required to perform the daily operations of the enterprise. Because the system is based on computers and records the business transactions of the enterprise, these systems are known as online transaction processing (OLTP) systems.

OLTP Systems

The data in OLTP systems is organized primarily to support transactions, such as:

- Recording an order from a point-of-sale terminal or entered through a Web site.
- Placing an order for more supplies when inventory levels drop to a defined level.
- Tracking components as they are assembled into a final product in a manufacturing facility.
- Recording employee data.
- Recording holders of licenses, such as restaurant or driver licenses.

Individual transactions are completed quickly and access relatively small amounts of data. OLTP systems are designed and tuned to process hundreds or thousands of transactions being entered at the same time.

Although OLTP systems excel at recording the data required to support daily operations, OLTP data is not organized in a manner that easily provides the information required by managers to plan the work of their organizations. Managers need summary information from which they can analyze trends that affect their organization or team. They need to find the critical factors affecting the success of their organization, and how best to adjust those factors to improve the success of the enterprise. They need to find how the workload of their enterprise is affected by seasonal and yearly trends so that they can predict how many employees and resources will be required to perform future work.

OLAP Systems

Systems designed to handle the queries required to discover trends and critical factors are called online analytical processing (OLAP) systems. OLAP queries typically require large amounts of data. For example, the head of a government motor vehicle licensing department could ask for a report that shows the number of each make and model of vehicle registered by the department each year for the past 20 years. Running this type of query against the original detail data in an OLTP system has two effects:

- The query takes a long time to aggregate (sum) all of the detail records for the last 20 years, so the report is not ready in a timely manner.
- The query generates a very heavy workload that at least slows down the normal users of the system from recording transactions at their normal pace.

Another issue is that many large enterprises do not have only one OLTP system that records all the transaction data. Most large enterprises have multiple OLTP systems, many of which were developed at different times and use different software and hardware. In many cases, the codes and names used to identify items in one system are different from the codes and names used in another. Managers running OLAP queries generally need to be able to reference the data from several of these OLTP systems. OLAP data is organized into multidimensional cubes. The structure of data in multidimensional cubes gives better performance for OLAP queries than data organized in relational tables. The basic unit of a multidimensional cube is called a measure. Measures are the units of data that are being analyzed. For example, a corporation that operates hardware stores wants to analyze revenue and discounts for the different products it sells. The measures are the number of units sold, revenue, and the sum of any discounts. The measures are organized along dimensions. In this example, a three dimensional cube could have these dimensions: time, store, and products. Think of these dimensions as forming the logical x, y, and z axis of a three-dimensional, virtual cube.

Each dimension is divided into units called members. The members of a dimension are typically organized into a hierarchy. Similar members are grouped together as a level of the hierarchy. For example, the top hierarchy level of a time dimension can be years, with months at the next level, then weeks, days, and finally hours at the bottom level of the hierarchy. At each intersection of the three dimensions, the values for the measures that match those three dimension values are recorded. For example, suppose that the hour starting at 1:00 P.M. Saturday, Feb. 19, 2000 is a time dimension member, Store #2 of Albany, New York is a store dimension member, and Easy-Clean Mops are a product dimension member. Where these three dimensions meet, the cell records that 10 mops were sold for revenues of \$90.00 and an average discount of \$1.00.

The specific dimensions and measures defined for the cubes in any particular OLAP system depend on the kinds of analysis important to the enterprise. Transforming OLTP data from relational tables into OLAP cubes, and the design of the cubes, is a complex area that is the subject of many third-party books.

OLAP systems operate on OLAP data in data warehouses or data marts. A data warehouse stores enterprise-level OLAP data, while a data mart is smaller and typically covers a single function in an organization.

See Also

<u>Creating and Maintaining Databases Overview</u> <u>Creating and Using Data Warehouses Overview</u>

Transforming OLTP Data to OLAP Data Warehouses

The transformation of OLTP data so that it gives acceptable performance in an OLAP system requires these processes:

Merge Data

You must be able to merge all the data related to specific items (products, customers, employees) from multiple OLTP systems into a single OLAP system. The merge process must resolve differences in encoding between the different OLTP systems. For example, one system may assign an ID to each employee, and the other systems have no employee IDs. The merge process must be able to match common employee data from both systems, perhaps by comparing employee names and addresses. The merge process must also be able to convert data stored using different data types in each OLTP system to a single data type used in the OLAP system. You must also select which columns in the OLTP system are not relevant to an OLAP system, and exclude these columns from the merge process.

The systems providing input data for an OLAP system are not strictly limited to traditional, centrally located OLTP systems. Valuable information may be stored in various legacy locations, even in some cases including relatively small sources such as Microsoft® Excel spreadsheets stored on a file share.

Scrub Data

Merging the OLTP data into a data warehouse gives you an opportunity to scrub data. You may find that various OLTP systems spell items differently, or the merge process may uncover previously unknown spelling errors. You may find other inconsistencies, such as having different addresses for the same store, employee, or customer. These inconsistencies have to be addressed before the data can be loaded into the data warehouse for use by the OLAP system.

Aggregate Data

OLTP data records all transaction details. OLAP queries typically need summary data, or data aggregated in some fashion. For example, a query to retrieve the monthly sales totals for each product over the last year runs much faster if the database only has summary rows showing the daily or hourly sales for each product, than if the query must scan every transaction detail record for the last year.

The degree to which you aggregate the data in a data warehouse depends on a number of design factors, such as the speed requirements of your OLAP queries and the level of granularity required for your analysis. For example, if you aggregate sales details into daily summaries instead of hourly summaries, your OLAP queries would run faster, but you could only do this if you had no need to analyze sales on an hourly basis.

Organize Data in Cubes

Relational OLTP data is organized in a way that makes some analysis processing difficult and time-consuming. When OLTP data is moved into a data warehouse, it must be transformed into an organization that better supports decision support analysis. The process of building a data warehouse involves reorganizing OLTP data stored in relational tables into OLAP data stored in multidimensional cubes.

Transformation Stages and Data Warehousing Components

The process of making data available through OLAP applications typically goes through three phases:

- 1. Extract the data from OLTP or legacy data sources into a staging area.
- 2. Transform the data into a form usable in an OLAP system. This involves actions such as data scrubbing and aggregation.
- 3. Load the data into a data warehouse or data mart.

The process of extracting the data from the OLTP and legacy data sources and transforming it into the warehouse servers is called the ETL process, and is typically run on a periodic basis, such as once a week or once a month.

Once the data is loaded into a data warehouse, an important part of an OLAP system is to provide facilities for decision makers to access and analyze the data in the data warehouses and data marts.

The illustration shows the general categories of components that OLAP systems use to provide these services.

Data Sources

The OLTP databases and other legacy sources of data that contain the data that must be transformed into the OLAP data in data warehouses and data marts.

Intermediate Data Stores

The combined data storage areas and processes that stage, cleanse, and transform the OLTP data into useful OLAP data.

Warehouse Servers

Warehouse servers are the computers running the relational databases that contain the data for data warehouses and data marts, and the servers that manage the OLAP data.

Business Intelligence

The sets of tools and applications that query the OLAP data and provide reports and information to the enterprise decision makers.

Meta Data

Models the organization of data and applications in the different OLAP components. Meta data describes objects such as tables in OLTP databases, cubes in data warehouses and data marts, and also records which applications reference the various pieces of data.

See Also

Creating and Using Data Warehouses Overview

SQL Server 2000 Data Warehouse and OLAP Components

Microsoft® SQL Server[™] 2000 provides several components (as shown in the illustration) that allow you to transform OLTP data into OLAP data, and make the OLAP information available to decision makers.

Extensible Markup Language and OLE DB

Extensible Markup Language (XML) is a standard that defines a formatting and data representation language independent of specific data stores or applications. It is becoming an increasing important standard in the transmission of data between applications and across the Web. SQL Server 2000 is enabled to return the result sets of queries as XML documents, and also to extract the data from XML documents and store them in the relevant tables in a database.

OLE DB is a common data access specification defined by Microsoft. Many data storage products, such as spreadsheets, databases, or other server applications, supply OLE DB providers that can be used by an OLE DB application to access the data. Applications using the OLE DB API can access any data for which there is an OLE DB provider. OLE DB can present its data as XML documents. OLE DB 2.5 also includes multidimensional extensions that let OLE DB providers expose information from multidimensional cubes.

XML and OLE DB are important mechanisms for communicating data between the various SQL Server 2000 data warehousing components. The definitions of some of the conceptual models used by some components are based on XML.

Microsoft ActiveX® Data Objects (ADO) is an object API that maps over OLE DB, but is more concise and easier to code. Like OLE DB, ADO can return its data as XML documents and also supports multi-dimensional extensions. Many applications use ADO as their API for accessing OLTP data.

SQL Server 2000 Relational Database Engine

The SQL Server 2000 database engine is used primarily in the OLTP systems, and also to store the intermediate data stores used when transforming OLTP data for storage in the data warehouse or data mart, and to store and manage the data in a data warehouse or data mart.

Data Transformation Services

Data Transformation Services (DTS) is a component built to take data from one OLE DB data source, perform operations, such as aggregating the data (SUM, MIN, MAX, AVG), and storing it in a destination OLE DB data source. DTS consists of packages, which define a particular set of work that forms a logical work item. Packages contain multiple connections to data sources, tasks to be performed, and workflows connecting connections and tasks. Examples of tasks include copying data from source to destination connections, transforming data from a source connection and placing the transformed data in the destination connection, executing a set of Microsoft ActiveX scripts or Transact-SQL statements against a connection.

DTS transforms OLTP data stored in relational tables into a different organization that can be used as the foundation for multidimensional cubes. Although the data in OLTP databases is stored in entity and relationship tables, data in an OLAP data warehouse is stored in fact and dimension tables. Fact tables store the measures exposed in multidimensional cubes, and dimension tables stores information about dimension members.

DTS is a powerful tool for any system that must repeatedly access data in one format and transform it into another format. The use of DTS is not limited to building data warehouses, but the power and capabilities of the component are excellently suited to the work of transforming OLTP data into OLAP data warehouse data. For more information, see <u>DTS Overview</u>.

Analysis Services and Data Mining

Analysis Services is an easy-to-use, integrated, and scalable set of components that enables you to build multidimensional cubes and provide the application programs with access to the cubes. Analysis Services is very flexible in the types of storage mechanisms it supports for the cubes. The cubes can be stored in relational databases (ROLAP), as separate, highperformance multidimensional data structures (MOLAP), or hybrid combinations of both (HOLAP). Analysis Services support wizards that ease tasks such as defining dimensions and cubes. For more information, see <u>Analysis Services Architecture</u>.

Analysis Services exposes the data in the multidimensional cubes to applications through an OLE DB provider. The Analysis Services provider supports multi-dimensional extensions defined as part of OLE DB 2.5, and the ActiveX Data Objects (Multidimensional) (ADO MD) API. For more information, see <u>Programming Analysis Services Applications</u>.

Analysis Services also supports industry-standard data mining algorithms. Data mining supports new and sophisticated tools for discovering trends in data and predicting future results. For more information, see <u>Data Mining</u> <u>Models</u>.

English Query

English Query allows end users to pose English language questions about information stored in SQL Server 2000 databases, or data warehouses, and OLAP cubes. An English Query administrator defines the logical and semantic relationships between the various tables and columns in a database or cubes, dimensions, and measures in a data warehouse. An application can be coded to ask the end user to type in an English query into a character field on a form. The character string is then passed to the English Query engine. The engine analysis the question against the logical definitions of the data provided by the administrator. When querying OLAP cubes, the English Query engine returns to the application an SQL statement that extracts the requested information from the database. When querying a data warehouse or data mart, the English Query engine returns an MDX query. The application executes the SQL statement or MDX query and returns the results to the end user. For more information, see <u>English Query Overview</u>.

Meta Data Services

SQL Server 2000 Meta Data Services stores a model that maps the organization of data in SQL Server 2000 databases and data warehouses. This information is primarily used by third-party rapid-development tools that can either prototype applications or provide application templates based on the information in the Meta Data Services model. For more information, see <u>Meta Data Services Overview</u>.

See Also

<u>Analysis Services Overview</u>

Creating and Maintaining Databases Overview

Creating and Using Data Warehouses Overview

DTS Overview

English Query Overview

Meta Data Services Overview

Application Development Architecture

Applications use two components to access a database:

• An application programming interface (API) or Uniform Resource Locator (URL).

A database API defines how to code an application to connect to a database and pass commands to the database. An object model API is usually language independent and defines a set of objects, properties, and interfaces, and a C or Microsoft® Visual Basic® API defines a set of functions for applications written in C, C++, or Visual Basic.

A Uniform Resource Locator is a string, or stream, that an Internet application can use to access resources on the Internet or an intranet. Microsoft SQL Server[™] 2000 provides an ISAPI dynamic-link library (DLL) that Microsoft Internet Information Services (IIS) applications use to build URLs that reference instances of SQL Server 2000.

• Database language.

A database language defines the syntax of the commands sent to the database. The commands sent through the API allow the application to access and modify data. They also allow the application to create and modify objects in the database. All commands are subject to the permissions granted to the user. SQL Server 2000 supports two languages:

- Internet applications running on IIS can use XPath queries with mapping schemas.
- The Transact-SQL language.
- The topics in this section provide information about the APIs supported by SQL Server 2000 and the issues to consider when choosing which API to use in an application.

Transact-SQL

Transact-SQL is the database language supported by SQL Server 2000. Transact-SQL complies with the Entry Level of the SQL-92 standard, but also supports several features from the Intermediate and Full Levels. Transact-SQL also supports some powerful extensions to the SQL-92 standard. For more information, see <u>Transact-SQL Overview</u>.

The ODBC specification defines extensions to the SQL defined in the SQL-92 standard. The ODBC SQL extensions are also supported by OLE DB. Transact-SQL supports the ODBC extensions from applications using the Microsoft ActiveX® Data Objects (ADO), OLE DB, or ODBC APIs, or the APIs that layer over ODBC. The ODBC SQL extensions are not supported from applications that use the DB-Library or Embedded SQL APIs.

XPath

SQL Server 2000 supports a subset of the XPath language defined by the World Wide Web Consortium (W3C). XPath is a graph navigation language used to select nodes from XML documents. You first use a mapping schema to define an XML-based view of the data in one or more SQL Server tables and views. You can then use XPath queries to retrieve data from that mapping schema.

You usually use XPath queries in either URLs or the ADO API, XPath queries are also supported by the OLE DB API.

APIs Supported by SQL Server

SQL Server supports a number of APIs for building general-purpose database applications, such as:

- These open APIs with publicly defined specifications supported by several database vendors:
 - ActiveX Data Objects (ADO)
 - OLE DB
 - Open Database Connectivity (ODBC) and the object APIs built over ODBC: Remote Data Objects (RDO) and Data Access

Objects (DAO)

- Embedded SQL for C (ESQL)
- The legacy DB-Library for C API that was developed specifically to be used with earlier versions of SQL Server that predate the SQL-92 standard.

Internet applications can also use URLs that specify IIS virtual roots that reference an instance of SQL Server. The URL can contain an XPath query, a Transact-SQL statement, or a template. In addition to using URLs, Internet applications can also use ADO or OLE DB to work with data in the form of XML documents.

Choosing an API

The general-purpose application programming interfaces (APIs) recommended for use in new applications that use Microsoft® SQL Server[™] 2000 are:

- Microsoft ActiveX® Data Objects (ADO) for most database applications. ADO supports rapid development of robust applications and has access to most SQL Server features. The SQL Server features needed by most applications are supported by ADO when using the Microsoft OLE DB Provider for SQL Server.
- URLs in Internet applications such as HTML or ASP pages.
- OLE DB for COM-based tools and utilities, or COM-based systemlevel development requiring either top performance or access to SQL Server features not exposed through ADO. The OLE DB Provider for SQL Server uses provider-specific properties, interfaces, and methods to expose SQL Server features not covered by the OLE DB specification. Most of these provider-specific features are not exposed through ADO.
- ODBC for the same class of applications as are listed above for OLE DB, but which are not based on COM.

Selecting a General-Purpose API

Several factors should be considered when you select a general-purpose API to use in a SQL Server application:

- Maturity of the API specification.
 - Existing

Existing API specifications are mature, stable specifications. Supplementary information about the API is readily available in third-party books and classes. There is an existing pool of programmers familiar with the API. • Emerging

Emerging API specifications are recent and may be evolving rapidly. Supplementary information about the latest version of the API may be scarce. There are relatively few programmers available who have used the API, although programmers familiar with a similar API can be retrained quickly.

• Legacy

Legacy API specifications are stable but unchanging. They may not support new features, and are likely to be discontinued at a future date. Information about the API is readily available, but the pool of programmers familiar with the API may be shrinking.

- Overhead.
 - Native APIs

Native APIs are low-level APIs implemented with providers or drivers that communicate directly to SQL Server using the Tabular Data Stream (TDS) protocol. They are relatively complex APIs, but offer the best performance because they have the least overhead.

• Object model APIs

Object model APIs use a relatively simple object model to encapsulate a native API. They are less efficient than native APIs because they must map their objects to the underlying native API, but their performance is acceptable for almost all applications. Applications using an object model API are simple to program and maintain. The object model API may not support all of the features of the underlying native API.

• Hosted APIs

Hosted APIs also encapsulate a native API, but do not use an object model. The efficiency, ease-of-use, and feature-set issues for hosted APIs are similar to those for object model APIs.

• Degree of developer control.

APIs vary in their overall feature set. Simple APIs such as ADO are easy to learn, program, and maintain, but they do not support all of the capabilities of the more complex APIs such as OLE DB and ODBC. You can take advantage of the ease-of-use advantages of the APIs such as ADO, RDO, and ESQL if they provide the functionality the application needs.

• Access to SQL Server features.

Some APIs have limitations on the numbers or types of SQL Server features they can use.

• Access to Microsoft SQL Server 2000 Analysis Services features.

Analysis Services, ADO MD, and OLE DB for OLAP offer support for online analytical processing. These services can be integrated with ADO and OLE DB applications using the OLE DB Provider for SQL Server.

• Programming language and tool support for the API.

The following table maps the general-purpose database APIs supported by SQL Server to the factors presented in the preceding list.

API	Maturity		Degree of developer control	Server 2000 feature	2000 XML	OLAP Services feature support	Language
ADO	Existing	Object Model over OLE DB	Moderate	Most	Yes		Microsoft Visual Basic® Microsoft Visual C++® Microsoft Visual J++®

URL	Emerging	Streams over OLE DB	Low	Limited	Yes	No	HTML Active Server Pages (ASP)
OLE DB	Existing	Native	High	All	Yes	Yes	Visual C++
ODBC	Existing	Native	High	All	No	No	Visual C++
RDO	Existing	Object Model over ODBC	Moderate	Most	No	No	Visual Basic Visual J++
DAO	Legacy	Object Model over ODBC	Low	Limited	No	No	Visual Basic Visual C++
ESQL	Legacy	Hosted over DB- Library	Low	Limited	No	No	Visual C++ COBOL
DB- Library for C	Legacy	Native	High	Limited	No	No	Visual C++

Additional APIs

SQL Server also supports a number of interfaces that allow applications to make full use of all SQL Server features:

• SQL Distributed Management Objects (<u>SQL-DMO API</u>)

A set of COM interfaces for managing and administering SQL Server.

• Replication components (<u>Replication Component Programming API</u>)

A set of COM interfaces for defining and managing replication between

SQL Server databases. You can also replicate data from heterogeneous databases to SQL Server.

• Data Transformation Services (Data Transformation Services API)

A set of COM interfaces (based on OLE DB) for defining and executing complex data transformations between OLE DB data providers.

• Extended Stored Procedure API (Extended Stored Procedure API)

A C language API for writing SQL Server extended stored procedures.

• English Query API (<u>SQL Server and English Query</u>)

An Automation API for evaluating strings that contain user questions against the information in SQL Server databases or OLAP cubes. The English Query server returns the SQL statement or MDX query that will retrieve the answer to the user question.

• Analysis Services APIs (<u>Programming Analysis Services Applications</u>)

Analysis Services exposes multiple APIs: Decision Support Objects to manage OLAP and data mining objects, ADO Multidimensional (ADO MD) and OLE DB OLAP extensions for accessing OLAP cubes, and data mining functionality.

• Meta Data Services (<u>Programming Meta Data Services Applications</u>)

An Automation API that gives applications and programming tools access to a model of the meta data in SQL Server databases.

See Also

Building SQL Server Applications Overview

SQL Server and ADO

Microsoft® ActiveX® Data Objects are a set of Automation objects that consume the OLE DB API and allow applications to consume data from OLE DB data sources. This includes data stored in many different formats, not only SQL databases. The ActiveX Data Object (ADO) API can be used from applications written in any automation-enabled language, such as Microsoft Visual Basic®, Microsoft Visual C++®, Microsoft Visual J++®, and Microsoft Visual FoxPro®.

ADO applications access data through OLE DB providers. Microsoft SQL Server[™] 2000 includes a native Microsoft OLE DB Provider for SQL Server used by ADO applications to access the data in SQL Server. In SQL Server version 6.5 and earlier, ADO applications had to use the OLE DB Provider for ODBC layered over the Microsoft SQL Server ODBC driver. Although ADO applications can still use the OLE DB Provider for ODBC with the SQL Server ODBC driver, it is more efficient to only use the OLE DB Provider for SQL Server.

ADO is the API most recommended for general-purpose data access to SQL Server for these reasons:

- ADO is easy to learn and program.
- ADO has the feature set required by most general-purpose applications.
- ADO enables programmers to quickly produce robust applications.

The core capabilities of the OLE DB specification provide all the data access functionality needed by most applications. In addition, OLE DB allows individual providers to define provider-specific mechanisms to support additional features of the data engine accessed by the provider. ADO exposes the core capabilities of OLE DB, but does not expose provider-specific features. ADO applications cannot access a few SQL Server features exposed through provider-specific features of the OLE DB Provider for SQL Server, such as the **IRowsetFastLoad** bulk copy methods, SQL Server-specific extended diagnostic information, and auto-fetch cursors.

ADO also supports the XML functionality of SQL Server 2000. This provides an easy migration path for Internet applications coded to use ADO to retrieve a rowset and then convert the rowset into an XML document. The application can instead use ADO to execute an XPath query or a SELECT statement with a FOR XML clause, in which case the result set is built as an XML document on the server rather than having to be converted on the application computer.

ADO has evolved from the earlier, ODBC-based Remote Data Objects (RDO) and Data Access Objects (DAO) APIs. RDO and DAO applications can be converted to ADO, and RDO and DAO application programmers quickly learn ADO. ADO is used extensively in Active Server Pages (ASP).

See Also

Programming ADO SQL Server Applications

SQL Server and Universal Resource Locators

Uniform Resource Locators (URLs) are formatted strings or streams that an Internet application can use to reference resources on the Internet or an intranet. Microsoft® SQL Server[™] 2000 supports URLs that work with data in SQL Server databases and return the results as XML documents. The URLs can execute XPath queries referencing mapping schemas that provide an XML-based view of the data in SQL Server tables. The URLs can also execute Transact-SQL statements or templates.

Although not strictly an API, URLs are the recommended mechanism for accessing SQL Server data from Web applications running on Microsoft Internet Information Services (IIS). URLs are easily integrated in HTML and XMLbased environments, such as HTML pages or Active Server Pages (ASPs). Specifying a URL that executes an XPath query or Transact-SQL statement that returns an XML document is a relatively simple way to integrate SQL Server results into a Web application.

See Also

XML and Internet Support Overview

URL Access

SQL Server and OLE DB

OLE DB is an API that allows COM applications to consume data from OLE DB data sources. OLE DB data sources include data stored in many different formats, not only SQL databases. An application uses an OLE DB provider to access an OLE DB data source. An OLE DB provider is a COM component that accepts calls to the OLE DB API and does whatever is necessary to process that request against the data source.

Microsoft® SQL Server[™] 2000 includes a native Microsoft OLE DB Provider for SQL Server used by OLE DB applications to access the data in SQL Server. The OLE DB Provider for SQL Server complies with the OLE DB 2.0 specification. Each OLE DB provider supports a command language; the OLE DB Provider for SQL Server accepts the command syntax specified as DBGUID_SQL. DBGUID_SQL syntax is primarily SQL-92 syntax with ODBC escape sequences.

In SQL Server version 6.5 and earlier, OLE DB applications had to use the OLE DB Provider for ODBC layered over the Microsoft SQL Server ODBC driver. While OLE DB applications can still use the OLE DB Provider for ODBC with the SQL Server ODBC driver, it is more efficient to use only the OLE DB Provider for SQL Server.

OLE DB is the API recommended for tools, utilities, or system level development needing either top performance or access to SQL Server features not exposed through ADO. The core capabilities of the OLE DB specification provide all the data access functionality needed by most applications. In addition, OLE DB allows individual providers to define provider-specific mechanisms to support additional features of the data engine accessed by the provider. ADO applications cannot access some SQL Server features exposed through provider-specific features of the OLE DB Provider for SQL Server, so applications needing to use the provider-specific features of the OLE DB Provider for SQL Server must use the OLE DB API. These features include:

• An **IRowsetFastLoad** interface to the SQL Server bulk copy component.

- An **ISQLServerErrorInfo** interface to get SQL Server-specific information from messages and errors.
- A LINKEDSERVERS rowset that exposes catalog information from the linked servers used in SQL Server distributed queries.
- Various provider-specific properties to control SQL Server-specific behaviors.

OLE DB also supports the XML functionality of SQL Server 2000. This provides an easy migration path for Internet applications coded to use OLE DB to retrieve a rowset and then convert it into an XML document. The application can instead use OLE DB to execute an XPath query or a SELECT statement with a FOR XML clause, in which case the result set is built as an XML document on the server rather than having to be converted on the application computer. Most application working with XML are written in ADO or use URLs, which are less complex than OLE DB.

See Also

Programming OLE DB SQL Server Applications

SQL Server and ODBC

Open Database Connectivity (ODBC) is a Call-Level Interface (CLI) that allows C and C++ applications to access data from ODBC data sources. A CLI is an API consisting of functions an application calls to obtain a set of services. ODBC data sources include data stored in different formats, not just SQL databases. An application uses an ODBC driver to access a data source. An ODBC driver is a dynamic-link library (DLL) that accepts calls to the ODBC API functions and does whatever is necessary to process that request against the data source.

ODBC is aligned with these specifications and standards defining a CLI for data access:

- The X/Open CAE Specification "Data Management: SQL Call-Level Interface (CLI)"
- ISO/IEC 9075-3:1995(E) Call-Level Interface (SQL/CLI)

ODBC has been widely accepted by database programmers, and several database vendors or third-party companies supply ODBC drivers. Several other Microsoft data access APIs were defined as simplified object models over ODBC, such as:

- Remote Data Objects (RDO)
- Data Access Objects (DAO)
- Microsoft Foundation Classes (MFC) Database Classes

Microsoft® SQL Server[™] 2000 includes a native Microsoft SQL Server ODBC driver used by ODBC applications to access the data in SQL Server. The SQL Server ODBC Driver complies with Level 2 of the ODBC 3.51 specification and exposes all the features of SQL Server. In SQL Server 2000 all of the SQL Server utilities except **isql** use the ODBC API and the SQL Server ODBC Driver.

ODBC can be used in tools, utilities, or system level development needing either top performance or access to SQL Server features, and which are not COM applications. ODBC, like OLE DB, allows individual drivers to define driverspecific mechanisms to support additional features of the data engine accessed by the driver. These features include:

- A set of bulk copy functions based on the earlier DB-Library bulk copy functions.
- Extensions to the ODBC diagnostic functions and records to get SQL Server-specific information from messages and errors.
- A set of functions that exposes catalog information from the linked servers used in SQL Server distributed queries.
- Various driver-specific attributes and connection string keywords to control SQL Server–specific behaviors.

See Also

Programming ODBC SQL Server Applications

SQL-DMO API

SQL Distributed Management Objects (SQL-DMO) encapsulate the objects found in Microsoft® SQL Server[™] 2000 databases. SQL-DMO allows applications written in languages that support Automation or COM to administer all parts of a SQL Server installation. SQL-DMO is the application programming interface (API) used by SQL Server Enterprise Manager in SQL Server 2000; therefore, applications using SQL-DMO can perform all functions performed by SQL Server Enterprise Manager.

SQL-DMO is intended for any Automation or COM application that must incorporate SQL Server administration, for example:

- Applications that encapsulate SQL Server as their data store and want to shield users from as much SQL Server administration as possible.
- Applications that have specialized administrative logic incorporated the application itself.
- Applications that want to integrate SQL Server administrative tasks in their own user interface.

Windows Management Instrumentation

The SQL Server 2000 compact disc contains support for a new API that will allow you to administer instances of SQL Server using Windows Management Instrumentation (WMI). WMI is a scalable Windows® 2000 component with an object-oriented API that lets management applications and scripts monitor, configure, and control the operating system and devices, services, and applications in a Windows network. Using standard Windows security, WMI allows only properly authorized users to manage the system. WMI core components are also available for Windows NT® 4.0, Windows 95, and Windows 98. For more information about the WMI support for these operating systems, see the MSDN® page at <u>Microsoft Web site</u>.

A component, such as SQL Server, enables WMI support by supplying a WMI

provider and defining a WMI class schema. The schema models the objects in the component that can be managed using WMI. SQL Server 2000 includes a SQL Server WMI provider and a schema class model that maps instances of SQL Server 2000 to WMI classes. The SQL Server WMI schema models objects such as databases and tables. The SQL Server WMI implementation provides management functions such as:

- Create, change, or delete managed objects. For example, create a database.
- Administer managed objects. For example, back up databases and logs.
- Enumerate managed objects. For example, list all the tables in a database.
- Retrieve information on a specific managed object. For example, determine whether full-text indexing is enabled on the **Customers** table.
- Query managed objects that meet a specific criterion. For example, list all encrypted stored procedures.
- Execute methods defined for managed objects. For example, execute a method that bulk copies data from a table.
- Generate events when a managed object is created, changed, or deleted (for example, send a event when a database option is changed).
- Describe relationships between managed objects (for example, identify which logins are authorized to access a database).

All WMI data is available remotely and is fully scriptable. The SQL Server 2000 WMI implementation maps over the SQL-DMO API, but does not support the management of replication. The SQL Server WMI implementation can be used with SQL Server 7.0.

The SQL Server WMI support is not installed by SQL Server 2000 Setup. All of the WMI materials, including a separate setup and documentation, are included in the folder \x86\OTHER\wmi on the SQL Server 2000 compact disc.

See Also

Administration Architecture

Developing SQL-DMO Applications

SQL Namespace API

The SQL Namespace (SQL-NS) application programming interface (API) is a set of objects that encapsulate the SQL Server Enterprise Manager user interface. SQL-NS allows applications written in languages that support Automation or COM to include parts of the SQL Server Enterprise Manager user interface in their own user interface.

Using SQL-NS, an application can incorporate the following SQL Server Enterprise Manager elements into its user interface:

- Wizards
- Dialog boxes (including property dialog boxes)

When an application uses the SQL-NS objects, SQL Server Enterprise Manager must be installed on any client that attempts to run the SQL-NS application.

See Also

Administration Architecture

Programming SQL-NS Applications

Replication Component Programming API

Microsoft® SQL Server[™] 2000 includes a set of replication objects in addition to the replication objects found in SQL-DMO. These objects include:

- The Replication Distributor Interface, which enables you to programmatically implement and manage heterogeneous transactional replication, in conjunction with third-party programs that can perform change tracking at heterogeneous data sources.
- Microsoft ActiveX[®] controls that allow you to provide the functionality of the Distribution Agent or the Merge Agent in custom programs.

See Also

Administration Architecture Developing SQL-DMO Applications Replication Overview Developing Replication Applications Using ActiveX Controls Programming Replication from Heterogeneous Data Sources

Data Transformation Services API

The Data Transformation Services (DTS) application programming interface (API) is a set of objects encapsulating services that assist with building a data warehouse. DTS can be used in applications written in languages that support Automation or COM:

- DTS transfers data between heterogeneous OLE DB data sources.
- DTS performs customized transformations that can convert detailed online transaction processing (OLTP) data to a summarized form for easy analysis of trend information.

See Also

DTS Overview

Programming DTS Applications

SQL Server and English Query

English Query provides an Automation API that lets users resolve naturallanguage questions about the information in a Microsoft® SQL Server[™] 2000 database.

Given a definition of the entities and relationships associated with a SQL Server database, English Query translates a natural-language question about data in the database to a set of SQL SELECT statements that can then be executed against the SQL Server database to get the answer.

For example, given a car sales database, an application can send English Query a string containing the question, "How many blue Fords were sold in 1996?"

English Query returns to the application an SQL statement such as:

```
SELECT COUNT(*)
FROM CarSales
WHERE Make = 'Ford'
AND Color = 'Blue'
AND DATEPART(yy, SalesDate) = '1996'
```

The application can then execute the SQL statement against the SQL Server database to get a number it can return to the user.

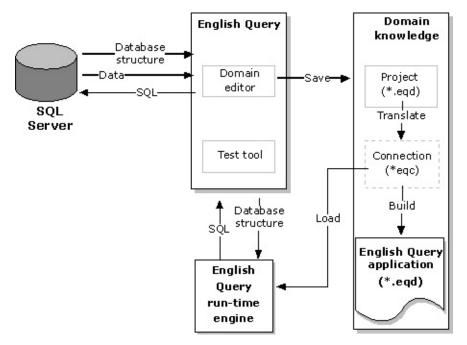
English Query works best with a normalized database. There are two parts to using English Query in an application:

1. An administrator defines an English Query project for the database and uses that to compile what is called an English Query application file. The English Query model is what defines the structure of the database to the English Query run-time engine.

An English Query model contains:

• Definitions of entities, which are usually associated with tables and columns of the database.

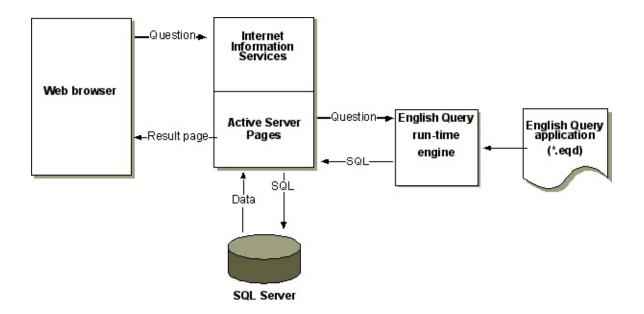
- Definitions of the relationships between the entities.
- 2. The model is defined with a Model Editor and tested with a test tool. The result is saved as an English Query project (.eqp) file. This file is compiled to form the English Query application (.eqd) file.



English Query uses full-text search to generate powerful queries designed to extract data from SQL Server database columns covered by full-text indexes. For more information, see <u>Full-Text Search</u>.

English Query also generates queries to extract data from the OLAP cubes stored in Analysis Services. When used as a front end for Analysis Services, English Query generates the MDX statements required to extract the data from the OLAP cubes. For more information, see <u>Analysis Services in English Query</u>.

English Query is an Automation server that can be called from any Automation application. The Automation server, or run-time engine, exposes an object model API. An Automation application uses the API to load the English Query application file and then send the run-time engine natural-language questions. The run-time engine usually returns an SQL statement or batch that will retrieve the required information from the SQL Server database. Sometimes the run-time engine can provide the answer directly, request clarification, or return an error.



See Also

Installing English Query

Extended Stored Procedure API

The Extended Stored Procedure application programming interface (API) is a server-based API specific to Microsoft® SQL ServerTM 2000. It can be used to produce extended stored procedures. An extended stored procedure is a C or C++ dynamic-link library (DLL) that can be called from Transact-SQL using the same syntax as calling a Transact-SQL stored procedure. Extended stored procedures are a way to extend the capabilities of Transact-SQL to include any resources or services available to Microsoft Win32® applications.

See Also

Programming Extended Stored Procedures

SQL Server and Embedded SQL

Embedded SQL (ESQL) is a SQL-92 standard application programming interface (API) for SQL database access. ESQL requires a two-step compilation process:

- 1. A precompiler translates Embedded SQL statements into commands in the programming language used to write the application. The generated statements are specific to the database that supplied the precompiler, so although the original source is generic to ESQL, the generated statements and the final executable file are specific to one database vendor.
- 2. The source generated by the precompiler is then compiled using the compiler for the application programming language.

Embedded SQL has a simpler syntax than COM APIs such as OLE DB or Call Level Interfaces such as ODBC, so it is easier to learn and program. It is less flexible than OLE DB or ODBC, where well-written applications can switch from one DBMS to another by simply switching drivers or providers. OLE DB and ODBC are also better at dealing with environments where the SQL statements are not known when the application is compiled, such as when developing as-required query tools.

Microsoft® SQL Server[™] 2000 provides an Embedded SQL precompiler for C applications. The SQL Server precompiler translates Embedded SQL statements as calls to the appropriate DB-Library API functions. The Microsoft implementation of ESQL has the same restrictions as DB-Library applications.

SQL Server is designed such that it can support COBOL Embedded-SQL applications compiled with third-party Embedded SQL precompilers that support Microsoft SQL Server.

See Also

Programming Embedded SQL for C

DB-Library API

DB-Library is a Call Level Interface that allows C applications to access Microsoft® SQL Server[™] 2000. DB-Library was the original application programming interface (API) that allowed applications to access SQL Server, and remains specific to SQL Server.

The DB-Library API has not been enhanced beyond the level of SQL Server version 6.5. All DB-Library applications can work with SQL Server 2000, but only as 6.5 level clients. Features introduced in SQL Server 2000 and SQL Server version 7.0 are not supported for DB-Library applications.

SQL Server 2000 does not include a programming environment for DB-Library for Microsoft Visual Basic®. Existing DB-Library for Visual Basic applications can run against SQL Server 2000, but must be maintained using the software development tools from SQL Server version 6.5. All development of new Visual Basic applications that access SQL Server should use the Visual Basic data APIs such as Microsoft ActiveX® Data Objects (ADO) and Remote Data Objects (RDO).

See Also

DB-Library for C Reference

SQL Syntax Recommendations

The Microsoft® SQL Server[™] 2000 Transact-SQL version complies with the Entry level of the SQL-92 standard, and supports many additional features from the Intermediate and Full levels of the standard.

The OLE DB and ODBC application programming interfaces (APIs) were developed with the understanding that applications would use:

- SQL-92 syntax when it provides the functionality needed by the application. Because the SQL dialects of most databases now comply with the Entry level of SQL-92 and support many features in the Intermediate and Full levels, this means many OLE DB providers and ODBC drivers can simply pass through most SQL-92 syntax without having to transform it to something accepted by the database.
- Use the ODBC extensions to SQL-92 when they provide functionality needed by the application that SQL-92 does not support.
- Use the native SQL syntax of the database engine when it provides functionality needed by the application that SQL-92 and the ODBC extensions do not support.

This approach minimizes the overhead of OLE DB providers and ODBC drivers. The providers and drivers only have to parse incoming SQL statements for ODBC escape sequences or SQL-92 syntax not accepted by the database. Any ODBC escape sequences and unsupported SQL-92 syntax are transformed into the corresponding SQL syntax accepted by the database engine. All other SQL syntax is passed through to the database engine.

SQL Server 2000 applications using OLE DB, ODBC, or one of the other APIs that encapsulate these two, should follow these guidelines:

• Use SQL-92 syntax when it provides the functionality required by the application.

- Use ODBC escape sequences when they provide functionality needed by the application but not provided by SQL-92.
- Use Transact-SQL syntax when it provides functionality required by the application but not provided by SQL-92 or the ODBC escape sequences.

Using SQL with DB-Library and Embedded SQL

DB-Library supports only Transact-SQL. DB-Library does not support the ODBC escape sequences or XML functionality.

Embedded SQL for C supports only the SQL syntax defined in Embedded SQL for C and Microsoft® SQL ServerTM.

DB-Library has not been extended after SQL Server version 6.5. It operates as a 6.5-level client and cannot use some new features introduced in Microsoft SQL Server 2000 and Microsoft SQL Server version 7.0. Embedded SQL uses DB-Library to communicate with SQL Server, so it also has the same restrictions. For more information, see <u>Connecting Early Version Clients to SQL Server</u> 2000.

Implementation Details

The topics in this section provide information about the editions of Microsoft® SQL Server[™] 2000 and the environments that support these editions. Information about the maximum capacities and memory usage of SQL Server 2000 objects is also provided.

Editions of SQL Server 2000

Microsoft[®] SQL Server[™] 2000 is available in these editions:

SQL Server 2000 Enterprise Edition

Used as a production database server. Supports all features available in SQL Server 2000, and scales to the performance levels required to support the largest Web sites and enterprise online transaction processing (OLTP) and data warehousing systems.

SQL Server 2000 Standard Edition

Used as a database server for a small workgroup or department.

SQL Server 2000 Personal Edition

Used by mobile users who spend some of their time disconnected from the network but run applications that require SQL Server data storage. Also used when running a stand-alone application that requires local SQL Server data storage on a client computer.

SQL Server 2000 Developer Edition

Used by programmers developing applications that use SQL Server 2000 as their data store. Although the Developer Edition supports all the features of the Enterprise Edition that allow developers to write and test applications that can use the features, the Developer Edition is licensed for use only as a development and test system, not a production server.

SQL Server 2000 Windows CE Edition

Microsoft® SQL Server 2000[™] Windows® CE Edition (SQL Server CE) is used as the data store on Windows CE devices. Capable of replicating data with any edition of SQL Server 2000 to keep Windows CE data synchronized with the primary database.

SQL Server 2000 Enterprise Evaluation Edition

Full-featured version available by a free download from the Web. Intended only for use in evaluating the features of SQL Server; this version will stop running 120 days after downloading. In addition to these editions of SQL Server 2000, the SQL Server 2000 Desktop Engine is a component that allows application developers to distribute a copy of the SQL Server 2000 relational database engine with their applications. While functionality of the database engine in the SQL Server 2000 Desktop Engine is similar to the database engine in the SQL Server Editions, the size of Desktop Engine databases cannot exceed 2 GB.

Both the SQL Server 2000 Personal Edition and SQL Server 2000 Desktop Engine have a concurrent workload governor that limits the performance of the database engine when more than 5 batches are executed concurrently. For more information about the concurrent workload governor, see <u>SQL Server 2000</u> <u>Databases on the Desktop</u>.

Upgrading From One Edition to Another

These are the supported upgrade paths between the editions and versions of SQL Server 2000:

- SQL Server 2000 Personal Edition to either SQL Server 2000 Enterprise Edition or SQL Server 2000 Standard Edition.
- SQL Server 2000 Standard Edition to SQL Server 2000 Enterprise Edition.
- SQL Server 2000 Desktop Engine to SQL Server 2000 Enterprise Edition or SQL Server 2000 Standard Edition.

Operating Systems Supported by the Editions of SQL Server 2000

This table shows the operating systems supported for running the server software from each Microsoft® SQL Server[™] 2000 edition.

Operating System	Enterprise Edition		Personal Edition	Developer Edition	Desktop Engine	SQL Server CE	Ei Ev E(
Microsoft Windows® 2000 DataCenter	Supported	Supported	Supported	Supported	Supported	N/A	St
Windows 2000 Advanced Server	Supported	Supported	Supported	Supported	Supported	N/A	St
Windows 2000 Server	Supported	Supported	Supported	Supported	Supported	N/A	St
Windows 2000 Professional	N/A	N/A	Supported	Supported	Supported	N/A	St
Microsoft Windows NT® 4.0 Server, Enterprise Edition	Supported	Supported	Supported	Supported	Supported	N/A	St
Windows NT 4.0 Server	Supported	Supported	Supported	Supported	Supported	N/A	Sτ
Windows NT 4.0	N/A	N/A	Supported	Supported	Supported	N/A	St

Workstation							
Microsoft	N/A	N/A	Supported	N/A	Supported	N/A	N۷
Windows							
98							
Microsoft	N/A	N/A	N/A	N/A	N/A	Supported	N/
Windows							
CE							

Note The client software from all SQL Server 2000 editions, except SQL Server CE, runs on any version of Microsoft Windows NT, Microsoft Windows 2000, and Microsoft Windows 98. Only the server components, such as the database engine and the Analysis server, are limited to specific versions of the operating systems. For example, although the database engine for SQL Server 2000 Enterprise Edition does not run on Windows 2000 Professional, Windows NT Workstation, or Windows 98, the SQL Server 2000 Enterprise Edition compact disc can be used to install the client software on any of these operating systems.

All of the software from SQL Server CE runs exclusively on the Windows CE operating system.

Features Supported by the Editions of SQL Server 2000

This topic summarizes the features that the different editions of Microsoft® SQL Server[™] 2000 support.

For more information about the amount of physical memory SQL Server 2000 can address, and the number of CPUs each edition supports in symmetric multiprocessor (SMP) computers, see <u>Maximum Capacity Specifications</u>.

Database Engine Features Supported by the Editions of SQL Server 2000

This table shows the database engine features and the editions of SQL Server 2000 that support them.

Database Engine Feature	Enterprise Edition			Developer Edition	Desktop Engine	Server	Enterp Evalua Editior
Multiple Instance Support	Supported	Supported	Supported	Supported	Supported	N/A	Suppor
Failover Clustering (up to four nodes)	Supported	N/A	N/A	Supported	N/A	N/A	Suppor
Failover Support in SQL Server Enterprise Manager	11	N/A	N/A	Supported	N/A	N/A	Suppor
Log	Supported	N/A	N/A	Supported	N/A	N/A	Suppor

Shipping							
Parallel DBCC	Supported	N/A	N/A	Supported	N/A	N/A	Suppor
Parallel CREATE INDEX	Supported	N/A	N/A	Supported	N/A	N/A	Suppor
Enhanced Read- ahead and Scan	Supported	N/A	N/A	Supported	N/A	N/A	Suppor
Indexed Views	Supported	N/A	N/A	Supported	N/A	N/A	Suppor
Federated Database Server	Supported	N/A	N/A	Supported	N/A	N/A	Suppor
System Area Network (SAN) Support	Supported	N/A	N/A	Supported	N/A	N/A	Suppor
	Supported	Supported	Supported	Supported	N/A	N/A	Suppor
Graphical Utilities Support for Language Settings	Supported	N/A	N/A	N/A	N/A	N/A	N/A
Full-Text Search	Supported	Supported	Supported (except on Windows 98)	Supported	N/A	N/A	Suppor

Replication Features Supported by the Editions of SQL Server 2000

This table shows the replication features and the editions of SQL Server 2000 that support them.

Replication						
Publisher	Enterprise	Standard	Personal	Developer	Desktop	SQL
Feature	Edition	Edition	Edition	Edition	Engine	Server CE
Snapshot Replication	Supported	Supported	Supported	Supported	Supported	N/A
Transactional Replication	Supported	Supported	Subscriber only	Supported	Subscriber only	N/A
Merge Replication	Supported	Supported	Supported	Supported	Supported	Anonymou Subscriber only
Immediate Updating Subscriptions		Supported	Supported	Supported	Supported	N/A
Queued Updating Subscribers	Supported	Supported	Supported	Supported	Supported	N/A

Analysis Services Features Supported by the Editions of SQL Server 2000

This table shows the Analysis Services features and the editions of SQL Server 2000 that support them.

Analysis SQL	Enterp
--------------	--------

Services Feature	Enterprise Edition	Standard Edition		Developer Edition	Desktop Engine	Server CE	Evalua Editior
Analysis Services	Supported	Supported	Supported	Supported	N/A	N/A	Suppor
User- defined OLAP Partitions	Supported	N/A	N/A	Supported	N/A	N/A	Suppor
Partition Wizard	Supported	N/A	N/A	Supported	N/A	N/A	Suppor
Linked OLAP Cubes	Supported	N/A	N/A	Supported	N/A	N/A	Suppor
ROLAP Dimension Support	Supported	N/A	N/A	Supported	N/A	N/A	Suppor
HTTP Internet Support	Supported	N/A	N/A	Supported	N/A	N/A	Suppor
Custom Rollups	Supported	Supported	Supported	Supported	N/A	N/A	Suppor
Calculated Cells	Supported	N/A	N/A	Supported	N/A	N/A	Suppor
Writeback to Dimensions		N/A	N/A	Supported	N/A	N/A	Suppor
Very Large Dimension Support	Supported	N/A	N/A	Supported	N/A	N/A	Suppor
Actions	Supported	Supported	Supported	Supported	N/A	N/A	Suppor
Real-time OLAP	Supported	N/A	N/A	Supported	N/A	N/A	Suppor
Distributed Partitioned Cubes	Supported	N/A	N/A	Supported	N/A	N/A	Suppor

Data	Supported	Supported	Supported	Supported	N/A	N/A	Suppor
Mining							

Data Transformation and Decision Support Query Features Supported by the Editions of SQL Server 2000

This table shows the data transformation and decision support query features and the editions of SQL Server 2000 that support them.

Analysis Services	Entornuico	Standard	Derconal	Davalanav		SQL Server
	Enterprise Edition			Developer Edition	-	CE
Data Transformation Services				Supported	Deployment only	N/A
Integrated Data Mining	Supported	Supported	Supported	Supported	N/A	N/A
English Query	Supported	Supported	Supported	Supported	N/A	N/A

Maximum Capacity Specifications

The first table specifies maximum capacities that are the same for all editions of Microsoft® SQL Server[™] 2000. The second and third tables specify capacities that vary by edition of SQL Server 2000 and the operating system.

This table specifies the maximum sizes and numbers of various objects defined in Microsoft SQL Server databases, or referenced in Transact-SQL statements. The table does not include Microsoft® SQL Server 2000[™] Windows® CE Edition.

	Maximum sizes/numbers			
Object	SQL Server 7.0	SQL Server 2000		
Batch size	65,536 * Network Packet Size ¹	65,536 * Network Packet Size ¹		
Bytes per short string column	8,000	8,000		
Bytes per text, ntext , or image column	2 GB-2	2 GB-2		
Bytes per GROUP BY, ORDER BY	8,060			
Bytes per index	900	9002		
Bytes per foreign key	900	900		
Bytes per primary key	900	900		
Bytes per row	8,060	8,060		
Bytes in source text of a stored procedure	Lesser of batch size or 250 MB	Lesser of batch size or 250 MB		
Clustered indexes per table	1	1		
Columns in GROUP BY, ORDER BY	Limited only by number of bytes			
Columns or expressions in a GROUP BY WITH CUBE or WITH ROLLUP statement	10			
Columns per index	16	16		
Columns per foreign key	16	16		

Columns per primary key	16	16
Columns per base table	1,024	1,024
Columns per SELECT statement	4,096	4,096
Columns per INSERT statement	1,024	1,024
Connections per client	Maximum value of configured connections	Maximum value of configured connections
Database size	1,048,516 TB ³	1,048,516 TB ³
Databases per instance of SQL Server	32,767	32,767
Filegroups per database	256	256
Files per database	32,767	32,767
File size (data)	32 TB	32 TB
File size (log)	4 TB	32 TB
Foreign key table references per table	253	253
Identifier length (in characters)	128	128
Instances per computer	N/A	16
Length of a string containing SQL statements (batch size)	65,536 * Network packet size ¹	65,536 * Network packet size ¹
Locks per connection	Max. locks per server	Max. locks per server
Locks per instance of SQL Server	2,147,483,647 (static) 40% of SQL Server memory (dynamic)	2,147,483,647 (static) 40% of SQL Server memory (dynamic)
Nested stored procedure levels	32	32
Nested subqueries	32	32
Nested trigger levels	32	32
Nonclustered indexes per table	249	249
Objects concurrently open in	2,147,483,647 (or	2,147,483,647 (or
an instance of SQL Server ⁴	available memory)	available memory)

Objects in a database	2,147,483,6474	2,147,483,6474
Parameters per stored procedure	1,024	1,024
REFERENCES per table	253	253
Rows per table	Limited by available storage	Limited by available storage
Tables per database	Limited by number of objects in a database ⁴	Limited by number of objects in a database ⁴
Tables per SELECT statement	256	256
Triggers per table	Limited by number of objects in a database ⁴	Limited by number of objects in a database ⁴
UNIQUE indexes or constraints per table	249 nonclustered and 1 clustered	249 nonclustered and 1 clustered

¹ Network Packet Size is the size of the tabular data scheme (TDS) packets used to communicate between applications and the relational database engine. The default packet size is 4 KB, and is controlled by the **network packet size** configuration option.

² The maximum number of bytes in any key cannot exceed 900 in SQL Server 2000. You can define a key using variable-length columns whose maximum sizes add up to more than 900, provided no row is ever inserted with more than 900 bytes of data in those columns. For more information, see <u>Maximum Size of Index Keys</u>.

³ The size of a database cannot exceed 2 GB when using the SQL Server 2000 Desktop Engine or the Microsoft Data Engine (MSDE) 1.0.

⁴ Database objects include all tables, views, stored procedures, extended stored procedures, triggers, rules, defaults, and constraints. The sum of the number of all these objects in a database cannot exceed 2,147,483,647.

Maximum Numbers of Processors Supported by the Editions of SQL Server 2000

This table shows the number of processors that the database engine in each SQL Server 2000 edition can support on symmetric multiprocessing (SMP) computers.



Operating	Enterprise	Standard	Dersonal	Developer	Deckton	-	Enterpr Evaluat
System		Edition		Edition	Engine	CE	Edition
Microsoft Windows® 2000 DataCenter	32	4	2	32	-	N/A	32
Windows 2000 Advanced Server	8	4	2	8	2	N/A	8
Windows 2000 Server	4	4	2	4	2	N/A	4
Windows 2000 Professional		N/A	2	2	2	N/A	2
Microsoft Windows NT® 4.0 Server, Enterprise Edition	8	8	2	8	2	N/A	8
Windows NT 4.0 Server	4	4	2	4	2	N/A	4
Windows NT 4.0 Workstation		N/A	2	2	2	N/A	2
Microsoft Windows 98	N/A	N/A	1	Use Desktop Engine	1	N/A	N/A
Microsoft Windows CE	N/A	N/A	N/A	N/A	N/A	1	N/A

Maximum Amount of Physical Memory Supported by the Editions of SQL Server 2000

This table shows the maximum amount of physical memory, or RAM, that the database engine in each SQL Server 2000 edition can support.

Operating System	Enterprise Edition	Standard Edition		Developer Edition	Desktop Engine	SQL Server CE	Enterpr Evaluat Edition
Windows 2000 DataCenter	64 GB	2 GB	2 GB	64 GB	2 GB	N/A	64 GB
Windows 2000 Advanced Server	8 GB	2 GB	2 GB	8 GB	2 GB	N/A	8 GB
Windows 2000 Server	4 GB	2 GB	2 GB	4 GB	2 GB	N/A	4 GB
Windows 2000 Professional	N/A	N/A	2 GB	2 GB	2 GB	N/A	2 GB
Windows NT 4.0 Server, Enterprise Edition	3 GB	2 GB	2 GB	3 GB	2 GB	N/A	3 GB
Windows NT 4.0 Server	2 GB	2 GB	2 GB	2 GB	2 GB	N/A	2 GB
Windows NT 4.0 Workstation	N/A	N/A	2 GB	2 GB	2 GB	N/A	2 GB

Configuration Option Specifications

Microsoft® SQL Server[™] 2000 contains improved algorithms for controlling computer resources. Many of the options that must be configured manually in earlier versions of SQL Server are managed dynamically in SQL Server 2000. These configuration options are not applicable in SQL Server 7.0 and are marked N/A in this table.

Several configuration options are still specified in SQL Server 2000; however, instead of specifying the size of a static allocation, the options now specify the upper limit for the number of objects allocated dynamically as needed. These options are marked with an asterisk (*) in this table. The information in this table does not pertain to Microsoft® SQL Server 2000TM Windows® CE Edition.

	SQL Server version 7.0		SQL Server 2000	
Configuration				
values	Minimum	Maximum	Minimum	Maximum
affinity mask	0	2,147,483,647	0	2,147,483,647
allow updates	0	1	0	1
AWE enabled	N/A	N/A	0	1
c2 audit mode	N/A	N/A	0	1
cost threshold for	0	32,767	0	32,767
parallelism				
cursor threshold	-1	2,147,483,647	-1	2,147,483,647
default full-text	N/A	N/A	0	2,147,483,647
language				
default language	0	9,999	0	9,999
default sort order	0	255	N/A	N/A
id				
extended memory	0	2,147,483,647	N/A	N/A
size (MB)				
fill factor (%)	0	100	0	100
index create	704	1,600,000	704	2,147,483,647
memory (K)				

language in cache	3	100	N/A	N/A
language neutral	0	1	N/A	N/A
full-text indexing				
lightweight	0	1	0	1
pooling				
locks	5,000*	2,147,483,647*	5,000*	2,147,483,647*
max async IO	1	255	N/A	N/A
max degree of parallelism	0	32	0	32
max server memory (MB)	4*	2,147,483,647*	4*	2,147,483,647*
max text repl size	0	2,147,483,647	0	2,147,483,647
max worker threads	10	1,024	10	32,767
media retention	0	365	0	365
min memory per query (K)	512	2,147,483,647	512	2,147,483,647
min server memory (MB)	0*	2,147,483,647*	0*	2,147,483,647*
nested triggers (bytes)	0	1	0	1
network packet size	512	65,535	512	65,532
open objects	0*	2,147,483,647*	0*	2,147,483,647*
priority boost	0	1	0	1
query governor cost limit	0	2,147,483,647	0	2,147,483,647
query wait (sec)	-1	2,147,483,647	-1	2,147,483,647
recovery interval (min)	0	32,767	0	32,767
remote access	0	1	0	1
remote login timeout (sec)	0	2,147,483,647	0	2,147,483,647

remote proc trans	0	1	0	1
remote query	0	2,147,483,647	0	2,147,483,647
timeout (sec)				
resource timeout	5	2,147,483,647	N/A	N/A
scan for startup	0	1	0	1
procs				
set working set	0	1	0	1
size				
show advanced	0	1	0	1
options				
spin counter	1	2,147,483,647	N/A	N/A
time slice	50	1,000	N/A	N/A
two digit year	1,752	9,999	1,752	9,999
cutoff				
unicode	0	2,147,483,647	N/A	N/A
comparison style				
unicode locale id	0	2,147,483,647	N/A	N/A
user connections	0*	32,767	0*	32,767
		(server)* ¹		(instance)* ¹
USEV:OPTIONS limit for	Q bjects allocat	4.0955 mically.	0	16,383

1 The concurrent workload governor in SQL Server 2000 Personal Edition and SQL Server 2000 Desktop Engine limits performance when more than 5 batches are executed concurrently.

Memory Used by SQL Server Objects Specifications

This table lists the amount of memory used by different objects in Microsoft® SQL ServerTM. The information in this table does not pertain to Microsoft® SQL Server 2000TM Windows® CE Edition.

	Object Size		
Object	SQL Server 7.0	SQL Server 2000	
Lock	96 bytes	64 bytes plus 32 bytes per owner.	
Open database	2,880 bytes	3924 bytes plus 1640 bytes per file and 336 bytes per filegroup.	
Open object ¹	276 bytes	256 bytes plus 1724 bytes per index opened on the object ² .	
User connection	12 KB + (3 * Network Packet Size) ³ .	12 KB + (3 * Network Packet Size) ³ .	

¹ Open objects include all tables, views, stored procedures, extended stored procedures, triggers, rules, defaults, and constraints.

² Indexes can be opened on tables or views.

³ Network Packet Size is the size of the tabular data scheme (TDS) packets used to communicate between applications and the relational database engine. The default packet size is 4 KB, and is controlled by the **network packet size** configuration option.