Creating and Maintaining Databases Overview

A client/server database system comprises two components:

- Programs that provide an interface for client-based users to access data.
- The database structure that manages and stores the data on the server.

For example, if you use Microsoft® SQL Server[™] 2000 to create a checking account application, you must set up a database structure to manage the account transaction data and an application that acts as the user interface to the database, allowing users to access checking account information.

Creating a database to serve your business needs requires an understanding of how to design, create, and maintain each of these components to ensure that your database performs optimally.

Торіс	Description
<u>Databases</u>	Describes how databases are used to represent, manage, and access data.
<u>Tables</u>	Describes how tables are used to store rows of data and define the relationships between multiple tables.
<u>Indexes</u>	Describes how indexes are used to increase the speed of accessing the data in the table.
<u>Views</u>	Describes views and their usefulness in providing an alternative way of looking at the data in one or more tables.
Stored Procedures	Describes how these Transact-SQL programs centralize business rules, tasks, and processes within the server.
<u>Enforcing Business</u> <u>Rules with Triggers</u>	Describes the function of triggers as special types of stored procedures executed only when data in a table is modified.
<u>Full-Text Indexes</u>	Describes how full-text indexes facilitate the

querying of data stored in character-based
columns, such as varchar and text .

Databases

A database in Microsoft® SQL ServerTM 2000 consists of a collection of tables that contain data and other objects, such as <u>views</u>, <u>indexes</u>, <u>stored procedures</u>, and <u>triggers</u>, defined to support activities performed with the data. The data stored in a database is usually related to a particular subject or process, such as inventory information for a manufacturing warehouse.

SQL Server can support many databases. Each database can store either interrelated orunrelated data from other databases. For example, a server can have one database that stores personnel data and another that stores product-related data. Alternatively, one database can store current customer order data, and another related database can store historical customer orders used for yearly reporting.

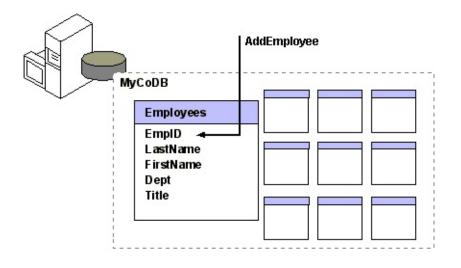
Before you create a database, it is important to understand the parts of a database and how to design these parts to ensure that the database performs well after it is implemented.

IMPORTANT It is recommended that you do not create any user objects, such as tables, views, stored procedures, or triggers, in the **master** database. The **master** database contains the system tables that store the system information used by SQL Server, such as configuration option settings.

Parts of a Database

A database in Microsoft® SQL Server[™] 2000 consists of a collection of tables that stores a specific set of structured data. A table contains a collection of rows (referred to as records or tuples) and columns (referred to as attributes). Each column in the table is designed to store a certain type of information (for example, dates, names, dollar amounts, or numbers). Tables have several types of controls (constraints, rules, triggers, defaults, and customized user data types) that ensure the validity of the data. Tables can have indexes similar to those in books that allow rows to be found quickly. Declarative referential integrity (DRI) constraints can be added to the tables to ensure that interrelated data in different tables remains consistent. A database can also store procedures that use Transact-SQL programming code to perform operations with the data in the database, such as storing views that provide customized access to table data.

For example, you create a database named **MyCoDB** to manage the data in your company. In the **MyCoDb** database, you create a table named **Employees** to store information about each employee, and the table contains columns named **EmpId**, **LastName**, **FirstName**, **Dept**, and **Title**. To ensure that no two employees share the same **EmpId** and that the **Dept** column contains only valid numbers for the departments in your company, you must add constraints to the table. Because you want to be able to quickly find the data for an employee, based on the employee ID or last name, you define indexes. You will have to add a row of data to the **Employees** table for each employee, so you create a procedure named **AddEmployee**, which is customized to accept the data values for a new employee and performs the operation of adding the row to the **Employees** table. You may need a departmental summary of employees, in which case you define a view called **DeptEmps** that combines data from the **Departments** and **Employees** tables and produces the output. This illustration shows the parts of the **MyCoDB** that is created.



Files and Filegroups

Microsoft® SQL Server[™] 2000 maps a database using a set of operating-system files. All data and objects in the database, such as tables, stored procedures, triggers, and views, are stored within these operating-system files:

• Primary

This file contains the startup information for the database and is used to store data. Every database has one primary data file.

• Secondary

These files hold all of the data that does not fit in the primary data file. If the primary file can hold all of the data in the database, databases do not need to have secondary data files. Some databases may be large enough to need multiple secondary data files or to use secondary files on separate disk drives to spread data across multiple disks.

• Transaction Log

These files hold the log information used to recover the database. There must be at least one log file for each database.

For example, a simple database, **sales**, can be created with one primary file that contains all data and objects and a log file that contains the transaction log information. Alternatively, a more complex database, **orders**, can be created with one primary file and five secondary files; the data and objects within the database spread across all six files, and four additional log files contain the transaction log information.

Filegroups allow files to be grouped together for administrative and data allocation/placement purposes. For example, three files (Data1.ndf, Data2.ndf, and Data3.ndf) can be created on three disk drives, respectively, and assigned to the filegroup **fgroup1**. A table can then be created specifically on the filegroup **fgroup1**. Queries for data from the table will be spread across the three disks, thereby improving performance. The same performance improvement can be accomplished with a single file created on a RAID (redundant array of independent disks) stripe set. Files and filegroups, however, allow you to easily

add new files on new disks. Additionally, if your database exceeds the maximum size for a single Microsoft Windows NT® file, you can use secondary data files to allow your database to continue to grow.

Rules for Designing Files and Filegroups

Rules for designing files and filegroups include:

- A file or filegroup cannot be used by more than one database. For example, file sales.mdf and sales.ndf, which contain data and objects from the **sales** database, cannot be used by any other database.
- A file can be a member of only one filegroup.
- Data and transaction log information cannot be part of the same file or filegroup.
- Transaction log files are never part of any filegroups.

See Also

CREATE DATABASE

Physical Database Files and Filegroups

Placing Tables on Filegroups

Transaction Logs

Default Filegroups

A database comprises a primary filegroup and any user-defined filegroups.

The filegroup that contains the primary file is the primary filegroup. When a database is created, the primary filegroup contains the primary data file and any other files that are not put into another filegroup. All system tables are allocated in the primary filegroup. If the primary filegroup runs out of space, no new catalog information can be added to the system tables. The primary filegroup only fills if either autogrow is turned off or all the disks holding the files in the primary filegroup run out of space. If this happens, either turn autogrow back on, or move other files off the disks to free more space.

User-defined filegroups are any filegroups that are specifically created by the user when first creating or later altering the database. If a user-defined filegroup fills up, only the user tables specifically allocated to that filegroup would be affected.

At any time, exactly one filegroup is designated as the DEFAULT filegroup. When objects are created in the database without specifying to which filegroup they belong, they are assigned to the default filegroup. The default filegroup must be large enough to hold any objects not allocated to a user-defined filegroup. Initially, the primary filegroup is the default filegroup.

The default filegroup can be changed using the ALTER DATABASE statement. By changing the default filegroup, any objects that do not have a filegroup specified when they are created are allocated to the data files in the new default filegroup. However, allocation for the system objects and tables remains within the PRIMARY filegroup, not the new default filegroup.

Changing the default filegroups prevents user objects that are not specifically created on a user-defined filegroup from competing with the system objects and tables for data space.

See Also

ALTER DATABASE

Using Files and Filegroups

Filegroups use a proportional fill strategy across all the files within each filegroup. As data is written to the filegroup, Microsoft® SQL ServerTM 2000 writes an amount proportional to the free space in the file to each file within the filegroup, rather than writing all the data to the first file until full, and then writing to the next file. For example, if file **f1** has 100 megabytes (MB) free and file **f2** has 200 MB free, one extent is allocated from file **f1**, two extents from file **f2**, and so on. This way both files become full at about the same time, and simple striping is achieved.

As soon as all the files in a filegroup are full, SQL Server automatically expands one file at a time in a round-robin fashion to accommodate more data (provided that the database is set to grow automatically). For example, a filegroup comprises three files, all set to automatically grow. When space in all files in the filegroup is exhausted, only the first file is expanded. When the first file is full, and no more data can be written to the filegroup, the second file is expanded. When the second file is full, and no more data can be written to the filegroup, the third file is expanded. If the third file becomes full, and no more data can be written to the filegroup, the first file is expanded again, and so on.

Using files and filegroups improves database performance by allowing a database to be created across multiple disks, multiple disk controllers, or RAID (redundant array of independent disks) systems. For example, if your computer has four disks, you can create a database that comprises three data files and one log file, with one file on each disk. As data is accessed, four read/write heads can simultaneously access the data in parallel, which speeds up database operations.

Additionally, files and filegroups allow data placement because a table can be created in a specific filegroup. This improves performance because all I/O for a specific table can be directed at a specific disk. For example, a heavily used table can be placed on one file in one filegroup, located on one disk, and the other, less heavily accessed tables in the database can be placed on the other files in another filegroup, located on a second disk.

Recommendations

These are some general recommendations for files and filegroups:

- Most databases will work well with a single data file and a single transaction log file.
- If you use multiple files, create a second filegroup for the additional file and make that filegroup the default filegroup. This way, the primary file will contain only system tables and objects.
- To maximize performance, create files or filegroups on as many different available local physical disks as possible, and place objects that compete heavily for space in different filegroups.
- Use filegroups to allow placement of objects on specific physical disks.
- Place different tables used in the same join queries in different filegroups. This will improve performance, due to parallel disk I/O searching for joined data.
- Place heavily accessed tables and the nonclustered indexes belonging to those tables on different filegroups. This will improve performance, due to parallel I/O if the files are located on different physical disks.
- Do not place the transaction log file or files on the same physical disk with the other files and filegroups.

See Also

CREATE DATABASE

Physical Database Files and Filegroups

Placing Tables on Filegroups

Transaction Logs

Using Files and Filegroups to Manage Database Growth

When creating a database using files and filegroups, you must specify an initial size for the file. Microsoft® SQL Server[™] 2000 creates the data files based on the size you provide. As data is added to the database, these files become full. However, you must consider whether and how the database will grow beyond the initial space you allocate if more data is added to the database than will fit in the files.

By default, SQL Server allows the data files to grow as much as necessary until disk space is exhausted. Therefore, if you do not want the database files to be allowed to grow any larger than when they were initially created, this must be specified at database creation time using SQL Server Enterprise Manager or the CREATE DATABASE statement.

Alternatively, SQL Server allows you to create data files that are allowed to grow automatically when they fill with data, but only to a predefined maximum size. This can prevent the disk drives from running out of disk space completely.

Recommendations

When you create a database, make the data files as large as possible, based on the maximum amount of data you expect in the database. Permit the data files to grow automatically but place a limit on the growth by specifying a maximum data file growth size that leaves some available space on the hard disk. This allows the database to grow if more data is added than expected, but does not fill up the disk drive. If the initial data file size is exceeded and the file starts to grow automatically, reevaluate the expected maximum database size and plan accordingly by adding more disk space (if necessary) and creating and adding more files or filegroups to the database.

However, if the database is not supposed to expand beyond its initial size, set the maximum growth size of the database to zero. This prevents the database files from growing. If the database files fill with data, no more data is added until more data files are added to the database or existing files are expanded.

Fragmentation of Files

Allowing files to grow automatically can cause fragmentation of those files if a large number of files share the same disk. Therefore, it is recommended that files or filegroups be created on as many different available local physical disks as possible. Place objects that compete heavily for space in different filegroups.

See Also

ALTER DATABASE

CREATE DATABASE

Read-Only Filegroups

Microsoft® SQL Server[™] 2000 allows filegroups to be marked as read-only. Any existing filegroup, except the primary filegroup, can be marked as read-only. A filegroup marked read-only cannot be modified in any way.

Place tables that must not be modified, such as historical data, on filegroups, and then mark the filegroup as read-only. This prevents accidental updates. The read-only filegroup can then be backed up and restored on another instance of SQL Server without concern for recovery of transaction logs.

See Also

ALTER DATABASE

Transaction Logs

A database in Microsoft® SQL Server[™] 2000 has at least one data file and one transaction log file. Data and transaction log information is never mixed on the same file, and individual files are used by only one database.

SQL Server uses the transaction log of each database to recover transactions. The transaction log is a serial record of all modifications that have occurred in the database as well as the transaction that performed each modification. The transaction log records the start of each transaction. It records the changes to the data and enough information to undo the modifications (if necessary later) made during each transaction. For some large operations, such as CREATE INDEX, the transaction log instead records the fact that the operation took place. The log grows continuously as logged operations occur in the database.

The transaction log records the allocation and deallocation of pages and the commit or rollback of each transaction. This allows SQL Server either to apply (roll forward) or back out (roll back) each transaction in the following ways:

- A transaction is rolled forward when you apply a transaction log. SQL Server copies the after image of every modification to the database or reruns statements such as CREATE INDEX. These actions are applied in the same sequence in which they originally occurred. At the end of this process, the database is in the same state it was in at the time the transaction log was backed up.
- A transaction is rolled back when you back out an incomplete transaction. SQL Server copies the before images of all modifications to the database since the BEGIN TRANSACTION. If it encounters transaction log records indicating that a CREATE INDEX was performed, it performs operations that logically reverse the statement. These before images and CREATE INDEX reversals are applied in the reverse of their original sequence.

At a checkpoint, SQL Server ensures that all transaction log records and database pages modified are written to disk. During the recovery process of each database that occurs when SQL Server is restarted, a transaction must be rolled

forward only when it is not known whether all the data modifications in the transaction were actually written from the SQL Server buffer cache to disk. Because a checkpoint forces all modified pages to disk, it represents the point at which the startup recovery must start rolling forward transactions. Because all pages modified before the checkpoint are guaranteed to be on disk, there is no need to roll forward anything done before the checkpoint.

Transaction log backups enable you to recover the database to a specific point in time (for example, prior to entering unwanted data), or to the point of failure. Transaction log backups should be a consideration in your media recovery strategy. For more information, see <u>Selecting a Recovery Model</u>.

See Also

CREATE DATABASE

Transactions

Transaction Log Backups

Virtual Log Files

Each transaction log file is divided logically into smaller segments called virtual log files. Virtual log files are the unit of truncation for the transaction log. When a virtual log file no longer contains log records for active transactions, it can be truncated and the space becomes available to log new transactions.

The smallest size for a virtual log file is 256 kilobytes (KB). The minimum size for a transaction log is 512 KB, which provides two 256-KB virtual log files. The number and size of the virtual log files in a transaction log increase as the size of the log file increases. A small log file can have a small number of small virtual log files (for example, a 5-MB log file that comprises five 1-MB virtual log files). A large log file can have larger virtual log files (for example, a 500-MB log file that comprises ten 50-MB virtual log files).

Microsoft® SQL Server[™] 2000 tries to avoid having many small virtual log files. The number of virtual log files grows much more slowly than the size. If a log file grows in small increments, it tends to have many small virtual log files. If the log file grows in larger increments, SQL Server creates a smaller number of larger virtual log files. For example, if the transaction log is growing by 1-MB increments, the virtual log files are smaller and more numerous compared to a transaction log growing at 50-MB increments. A large number of virtual log files can increase the time taken to perform database recovery.

As records are written to the log, the end of the log grows from one virtual log file to the next. If there is more than one physical log file for a database, the end of the log grows through each virtual log file in each physical file before circling back to the first virtual log file in the first physical file. Only when all log files are full will the log begin to grow automatically.

See Also

Shrinking a Database

Database Design Considerations

Designing a database requires an understanding of both the business functions you want to model and the database concepts and features used to represent those business functions.

It is important to accurately design a database to model the business because it can be time consuming to change the design of a database significantly once implemented. A well-designed database also performs better.

When designing a database, consider:

- The purpose of the database and how it affects the design. Create a database plan to fit your purpose.
- Database normalization rules that prevent mistakes in the database design.
- Protection of your data integrity.
- Security requirements of the database and user permissions.
- Performance needs of the application. You must ensure that the database design takes advantage of Microsoft® SQL Server[™] 2000 features that improve performance. Achieving a balance between the size of the database and the hardware configuration is also important for performance.
- Maintenance.
- Estimating the size of a database.

Creating a Database Plan

The first step in creating a database is creating a plan that serves both as a guide to be used when implementing the database and as a functional specification for the database after it has been implemented. The complexity and detail of a database design is dictated by the complexity and size of the database application as well as the user population.

The nature and complexity of a database application, as well as the process of planning it, can vary greatly. A database can be relatively simple and designed for use by a single person, or it can be large and complex and designed, for example, to handle all the banking transactions for hundreds of thousands of clients. In the first case, the database design may be little more than a few notes on some scratch paper. In the latter case, the design may be a formal document with hundreds of pages that contain every possible detail about the database.

In planning the database, regardless of its size and complexity, use these basic steps:

- Gather information.
- Identify the objects.
- Model the objects.
- Identify the types of information for each object.
- Identify the relationships between objects.

Gathering Information

Before creating a database, you must have a good understanding of the job the database is expected to perform. If the database is to replace a paper-based or manually performed information system, the existing system will give you most of the information you need. It is important to interview everyone involved in

the system to find out what they do and what they need from the database. It is also important to identify what they want the new system to do, as well as to identify the problems, limitations, and bottlenecks of any existing system. Collect copies of customer statements, inventory lists, management reports, and any other documents that are part of the existing system, because these will be useful to you in designing the database and the interfaces.

Identifying the Objects

During the process of gathering information, you must identify the key objects or entities that will be managed by the database. The object can be a tangible thing, such as a person or a product, or it can be a more intangible item, such as a business transaction, a department in a company, or a payroll period. There are usually a few primary objects, and after these are identified, the related items become apparent. Each distinct item in your database should have a corresponding table.

The primary object in the **pubs** sample database included with Microsoft® SQL ServerTM 2000 is a book. The objects related to books within this company's business are the authors who write the books, the publishers who manufacture the books, the stores which sell them, and the sales transactions performed with the stores. Each of these objects is a table in the database.

Modeling the Objects

As the objects in the system are identified, it is important to record them in a way that represents the system visually. You can use your database model as a reference during implementation of the database.

For this purpose, database developers use tools that range in technical complexity from pencils and scratch paper to word processing or spreadsheet programs, and even to software programs specifically dedicated to the job of data modeling for database designs. Whatever tool you decide to use, it is important that you keep it up-to-date.

SQL Server Enterprise Manager includes visual design tools such as the Database Designer that can be used to design and create objects in the database. For more information see, <u>Database Designer</u>.

Identifying the Types of Information for Each Object

After the primary objects in the database have been identified as candidates for tables, the next step is to identify the types of information that must be stored for each object. These are the columns in the object's table. The columns in a database table contain a few common types of information:

• Raw data columns

These columns store tangible pieces of information, such as names, determined by a source external to the database.

• Categorical columns

These columns classify or group the data and store a limited selection of data such as true/false, married/single, VP/Director/Group Manager, and so on.

• Identifier columns

These columns provide a mechanism to identify each item stored in the table. These columns often have id or number in their names (for example, **employee_id**, **invoice_number**, and **publisher_id**). The identifier column is the primary component used by both users and internal database processing for gaining access to a row of data in the table. Sometimes the object has a tangible form of ID used in the table (for example, a social security number), but in most situations you can define the table so that a reliable, artificial ID can be created for the row.

• Relational or referential columns

These columns establish a link between information in one table and related information in another table. For example, a table that tracks sales transactions will commonly have a link to the **customers** table so that the complete customer information can be associated with the sales transaction.

Identifying the Relationships Between Objects

One of the strengths of a relational database is the ability to relate or associate

information about various items in the database. Isolated types of information can be stored separately, but the database engine can combine data when necessary. Identifying the relationships between objects in the design process requires looking at the tables, determining how they are logically related, and adding relational columns that establish a link from one table to another.

For example, the designer of the **pubs** database has created tables for titles and publishers in the database. The **titles** table contains information for each book: an identifier column named **title_id**; raw data columns for the title, the price of the book, and the publishing date; and some columns with sales information for the book. The table contains a categorical column named **type**, which allows the books to be grouped by the type of content in the book. Each book also has a publisher, but the publisher information is in another table; therefore, the **titles** table has a **pub_id** column to store just the ID of the publisher. When a row of data is added for a book, the publisher ID is stored with the rest of the book information.

Online Transaction Processing vs. Decision Support

Many applications fall into two main categories of database applications:

- <u>Online transaction processing (OLTP)</u>
- <u>Decision support</u>

The characteristics of these application types have a dramatic effect on the design considerations for a database.

Online Transaction Processing

Online Transaction processing database applications are optimal for managing changing data, and usually have a large number of users who will be simultaneously performing transactions that change real-time data. Although individual requests by users for data tend to reference few records, many of these requests are being made at the same time. Common examples of these types of databases are airline ticketing systems and banking transaction systems. The primary concerns in this type of application are concurrency and atomicity.

Concurrency controls in a database system ensure that two users cannot change the same data, or that one user cannot change a piece of data before another user is done with it. For example, if you are talking to an airline ticket agent to reserve the last available seat on a flight and the agent begins the process of reserving the seat in your name, another agent should not be able to tell another passenger that the seat is available.

Atomicity ensures that all of the steps involved in a transaction complete successfully as a group. If any step fails, no other steps should be completed. For example, a banking transaction may involve two steps: taking funds out of your checking account and placing them into your savings account. If the step that removes the funds from your checking account succeeds, you want to make sure that the funds are placed into your savings account or put back into your checking account.

Online Transaction Processing Design Considerations

Transaction processing system databases should be designed to promote:

• Good data placement.

I/O bottlenecks are a big concern for OLTP systems due to the number of users modifying data all over the database. Determine the likely access patterns of the data and place frequently accessed data together. Use filegroups and RAID (redundant array of independent disks) systems to assist in this.

• Short transactions to minimize long-term locks and improve concurrency.

Avoid user interaction during transactions. Whenever possible, execute a single stored procedure to process the entire transaction. The order in which you reference tables within your transactions can affect concurrency. Place references to frequently accessed tables at the end of the transaction to minimize the duration that locks are held.

• Online backup.

OLTP systems are often characterized by continuous operations (24 hours a day, 7 days a week) for which downtime is kept to an absolute minimum. Although Microsoft® SQL Server[™] 2000 can back up a database while it is being used, schedule the backup process to occur during times of low activity to minimize effects on users.

• High normalization of the database.

Reduce redundant information as much as possible to increase the speed of updates and hence improve concurrency. Reducing data also improves the speed of backups because less data needs to be backed up.

• Little or no historical or aggregated data.

Data that is rarely referenced can be archived into separate databases, or moved out of the heavily updated tables into tables containing only historical data. This keeps tables as small as possible, improving backup times and query performance.

• Careful use of indexes.

Indexes must be updated each time a row is added or modified. To

avoid over-indexing heavily updated tables, keep indexes narrow. Use the Index Tuning Wizard to design your indexes.

• Optimum hardware configuration to handle the large numbers of concurrent users and quick response times required by an OLTP system.

Decision Support

Decision-support database applications are optimal for data queries that do not change data. For example, a company can periodically summarize its sales data by date, sales region, or product and store this information in a separate database to be used for analysis by senior management. To make business decisions, users need to be able to determine trends in sales quickly by querying the data based on various criteria. However, they do not need to change this data. The tables in a decision-support database are heavily indexed, and the raw data is often preprocessed and organized to support the various types of queries to be used. Because the users are not changing data, concurrency and atomicity issues are not a concern; the data is changed only by periodic, bulk updates made during off-hour, low-traffic times in the database.

Decision Support Design Considerations

Decision-support system databases should be designed to promote:

• Heavy indexing.

Decision-support systems have low update requirements but large volumes of data. Use many indexes to improve query performance.

• Denormalization of the database.

Introduce preaggregated or summarized data to satisfy common query requirements and improve query response times.

• Use of a star or snowflake schema to organize the data within the database.

See Also

Creating a Data Warehouse

Parts of a Data Warehouse

Normalization

The logical design of the database, including the tables and the relationships between them, is the core of an optimized relational database. A good logical database design can lay the foundation for optimal database and application performance. A poor logical database design can impair the performance of the entire system.

Normalizing a logical database design involves using formal methods to separate the data into multiple, related tables. A greater number of narrow tables (with fewer columns) is characteristic of a normalized database. A few wide tables (with more columns) is characteristic of an nonnomalized database.

Reasonable normalization often improves performance. When useful indexes are available, the Microsoft® SQL ServerTM 2000 query optimizer is efficient at selecting rapid, efficient joins between tables.

Some of the benefits of normalization include:

- Faster sorting and index creation.
- A larger number of clustered indexes. For more information, see <u>Clustered Indexes</u>.
- Narrower and more compact indexes.
- Fewer indexes per table, which improves the performance of INSERT, UPDATE, and DELETE statements.
- Fewer null values and less opportunity for inconsistency, which increase database compactness.

As normalization increases, so do the number and complexity of joins required to retrieve data. Too many complex relational joins between too many tables can hinder performance. Reasonable normalization often includes few regularly executed queries that use joins involving more than four tables.

Sometimes the logical database design is already fixed and total redesign is not feasible. Even then, however, it might be possible to normalize a large table selectively into several smaller tables. If the database is accessed through stored procedures, this schema change could take place without affecting applications. If not, it might be possible to create a view that hides the schema change from the applications.

Achieving a Well-Designed Database

In relational-database design theory, normalization rules identify certain attributes that must be present or absent in a well-designed database. A complete discussion of normalization rules goes well beyond the scope of this topic. However, there are a few rules that can help you achieve a sound database design:

• A table should have an identifier.

The fundamental rule of database design theory is that each table should have a unique row identifier, a column or set of columns used to distinguish any single record from every other record in the table. Each table should have an ID column, and no two records can share the same ID value. The column or columns serving as the unique row identifier for a table is the primary key of the table.

• A table should store only data for a single type of entity.

Attempting to store too much information in a table can prevent the efficient and reliable management of the data in the table. In the **pubs** database in SQL Server 2000, the titles and publishers information is stored in two separate tables. Although it is possible to have columns that contain information for both the book and the publisher in the **titles** table, this design leads to several problems. The publisher information must be added and stored redundantly for each book published by a publisher. This uses extra storage space in the database. If the address for the publisher changes, the change must be made for each book. And if the last book for a publisher is removed from the title table, the information for that publisher is lost.

In the **pubs** database, with the information for books and publishers stored in the **titles** and **publishers** tables, the information about the publisher has to be entered only once and then linked to each book. Therefore, if the publisher information is changed, it must be changed in only one place, and the publisher information will be there even if the publisher has no books in the database.

• A table should avoid nullable columns.

Tables can have columns defined to allow null values. A null value indicates that there is no value. Although it can be useful to allow null values in isolated cases, it is best to use them sparingly because they require special handling that increases the complexity of data operations. If you have a table with several nullable columns and several of the rows have null values in the columns, you should consider placing these columns in another table linked to the primary table. Storing the data in two separate tables allows the primary table to be simple in design but able to accommodate the occasional need for storing this information.

• A table should not have repeating values or columns.

The table for an item in the database should not contain a list of values for a specific piece of information. For example, a book in the **pubs** database might be coauthored. If there is a column in the **titles** table for the name of the author, this presents a problem. One solution is to store the name of both authors in the column, but this makes it difficult to show a list of the individual authors. Another solution is to change the structure of the table to add another column for the name of the second author, but this accommodates only two authors. Yet another column must be added if a book has three authors.

If you find that you need to store a list of values in a single column, or if you have multiple columns for a single piece of data (**au_lname1**, **au_lname2**, and so on), you should consider placing the duplicated data in another table with a link back to the primary table. The **pubs** database has a table for book information and another table that stores only the ID values for the books and the IDs of the authors of the books. This design allows any number of authors for a book without modifying

the definition of the table and allocates no unused storage space for books with a single author.

Data Integrity

Enforcing <u>data integrity</u> ensures the quality of the data in the database. For example, if an employee is entered with an **employee_id** value of **123**, the database should not allow another employee to have an ID with the same value. If you have an **employee_rating** column intended to have values ranging from **1** to **5**, the database should not accept a value of **6**. If the table has a **dept_id** column that stores the department number for the employee, the database should allow only values that are valid for the department numbers in the company.

Two important steps in planning tables are to identify valid values for a column and to decide how to enforce the integrity of the data in the column. Data integrity falls into these categories:

- Entity integrity
- Domain integrity
- Referential integrity
- User-defined integrity

Entity Integrity

Entity integrity defines a row as a unique entity for a particular table. Entity integrity enforces the integrity of the identifier column(s) or the primary key of a table (through indexes, UNIQUE constraints, PRIMARY KEY constraints, or IDENTITY properties).

Domain Integrity

Domain integrity is the validity of entries for a given column. You can enforce domain integrity by restricting the type (through data types), the format (through CHECK constraints and rules), or the range of possible values (through FOREIGN KEY constraints, CHECK constraints, DEFAULT definitions, NOT NULL definitions, and rules).

Referential Integrity

Referential integrity preserves the defined relationships between tables when records are entered or deleted. In Microsoft® SQL Server[™] 2000, referential integrity is based on relationships between foreign keys and primary keys or between foreign keys and unique keys (through FOREIGN KEY and CHECK constraints). Referential integrity ensures that key values are consistent across tables. Such consistency requires that there be no references to nonexistent values and that if a key value changes, all references to it change consistently throughout the database.

When you enforce referential integrity, SQL Server prevents users from:

- Adding records to a related table if there is no associated record in the primary table.
- Changing values in a primary table that result in orphaned records in a related table.
- Deleting records from a primary table if there are matching related records.

For example, with the **sales** and **titles** tables in the **pubs** database, referential integrity is based on the relationship between the foreign key (**title_id**) in the **sales** table and the primary key (**title_id**) in the **titles** table.

User-Defined Integrity

User-defined integrity allows you to define specific business rules that do not fall into one of the other integrity categories. All of the integrity categories support user-defined integrity (all column- and table-level constraints in CREATE TABLE, stored procedures, and triggers).

See Also

Specifying a Column Data Type

Using Constraints, Defaults, and Null Values

Data Security

One of the functions of a database is to protect the data by preventing certain users from seeing or changing highly sensitive data and preventing all users from making costly mistakes. The security system in Microsoft® SQL ServerTM 2000 controls user- access to the data, and user-permissions to perform activities in the database.

See Also

Setting Up Security Accounts

Database Performance

When you design a database, you must ensure that the database performs all the important functions correctly and quickly. Some performance issues can be resolved after the database is in production, but other performance issues may be the result of a poor database design and can be addressed only by changing the structure and design of the database.

When you design and implement a database, you should identify the large tables in the database and the more complex processes that the database will perform, and give special consideration to performance when designing these tables. Also consider the effect on performance of increasing the number of users who can access the database.

Examples of design changes that improve performance include:

- If a table containing hundreds of thousands of rows must be summarized for a daily report, you can add a column or columns to the table that contains preaggregated data to be used only for the report.
- Databases can be overnormalized, which means the database is defined with numerous, small, interrelated tables. When the database is processing the data in these tables, it has to perform a great deal of extra work to combine the related data. This extra processing can reduce the performance of the database. In these situations, denormalizing the database slightly to simplify complex processes can improve performance.

In conjunction with correct database design, correct use of indexes, RAID (redundant array of independent disks), and filegroups is important for achieving good performance.

Hardware Considerations

Generally, the larger the database, the greater the hardware requirements. But there are other determining factors: the number of concurrent users/sessions, transaction throughput, and the types of operations within the database. For example, a database containing infrequently updated data for a school library would generally have lower hardware requirements than a 1-terabyte (TB) data warehouse containing frequently analyzed sales, product, and customer information of a large corporation. Aside from the disk storage requirements, more memory and faster processors would be needed for the data warehouse to enable more of the data to be cached in memory and queries referencing large amounts of data to be processed quickly.

See Also

Database Design Indexes Physical Database Files and Filegroups RAID

Maintenance

After a database has been created and all objects and data have been added and are in use, there will be times when maintenance must be performed. For example, it is important to back up the database regularly. You may also need to create some new indexes to improve performance. These issues should be taken into consideration when you design the database to minimize the effect on users, the time taken to perform the task, and the effort involved.

Maintenance design guidelines include:

• Designing the database to be as small as possible and to exclude redundant information.

Normalizing your database can help you achieve this. For example, reducing the size of the database can help reduce the time taken to back up or, more importantly, restore a database. This is especially important during a restore operation because the database is unavailable while it is being restored.

• Designing partitioned tables rather than a single table, if the table will contain a large number of rows.

For example, a table containing every credit card transaction received by a bank could be split into multiple tables, with each table holding data for a single month. This can ease index maintenance if new indexes would otherwise have to be added to improve query performance. It may be necessary to create the index only on data from the last three months because older data is no longer referenced. The larger the table, the longer it takes to create new indexes.

Microsoft® SQL Server[™] 2000 provides the Database Maintenance Plan Wizard for automating many of these tasks, thereby reducing or removing the work involved in database maintenance.

See Also

Database Maintenance Plan Wizard

Estimating the Size of a Database

When designing a database, you may need to estimate how big the database will be when filled with data. Estimating the size of the database can help you determine the hardware configuration you will need for:

- Achieving the performance required by your applications. For more information, see Hardware Considerations in <u>Database Performance</u>.
- Ensuring the appropriate physical amount of disk space to store the data and indexes.

Estimating the size of a database can also lead you to determine whether the database design needs refining. For example, you may determine that the estimated size of the database is too large to implement in your organization and that more normalization is required. Conversely, the estimated size may be smaller than expected, allowing you to denormalize the database to improve query performance.

To estimate the size of a database, estimate the size of each table individually, and then add the values obtained. The size of a table depends on whether the table has indexes, and if so, what type of indexes.

See Also

Designing Tables Indexes Query Tuning Table and Index Architecture

Estimating the Size of a Table

The following steps can be used to estimate the amount of space required to store the data in a table:

1. Specify the number of rows present in the table:

Number of rows in the table = Num_Rows

2. If there are fixed-length and variable-length columns in the table definition, calculate the space that each of these groups of columns occupies within the data row. The size of a column depends on the data type and length specification. For more information, see <u>Data Types</u>.

```
Number of columns = Num_Cols
```

Sum of bytes in all fixed-length columns = Fixed_Data_Size

Number of variable-length columns = Num_Variable_Cols

Maximum size of all variable-length columns = Max_Var_Size

3. If there are fixed-length columns in the table, a portion of the row, known as the null bitmap, is reserved to manage column nullability. Calculate its size:

Null Bitmap (Null_Bitmap) = $2 + ((Num_Cols + 7) / 8)$

Only the integer portion of the above expression should be used; discard any remainder.

4. If there are variable-length columns in the table, determine how much space is used to store the columns within the row:

Total size of variable-length columns (Variable_Data_Size) = 2 + (Num_Variable_Cols x 2) + Max_Var_Size

If there are no variable-length columns, set Variable_Data_Size to 0.

This formula assumes that all variable-length columns are 100 percent full. If you anticipate that a lower percentage of the variable-length

column storage space will be used, you can adjust the result by that percentage to yield a more accurate estimate of the overall table size.

5. Calculate the row size:

Total row size (Row_Size) = Fixed_Data_Size + Variable_Data_Size + Null_Bitmap +4

The final value of 4 represents the data row header.

6. Calculate the number of rows per page (8096 free bytes per page):

Number of rows per page (Rows_Per_Page) = (8096) / (Row_Size + 2)

Because rows do not span pages, the number of rows per page should be rounded down to the nearest whole row.

7. If a clustered index is to be created on the table, calculate the number of reserved free rows per page, based on the fill factor specified. For more information, see <u>Fill Factor</u>. If no clustered index is to be created, specify Fill_Factor as 100.

Number of free rows per page (Free_Rows_Per_Page) = 8096 x ((100 - Fill_Factor) / 100) / (Row_Size + 2)

The fill factor used in the calculation is an integer value rather than a percentage.

Because rows do not span pages, the number of rows per page should be rounded down to the nearest whole row. As the fill factor grows, more data will be stored on each page and there will be fewer pages.

8. Calculate the number of pages required to store all the rows:

Number of pages (Num_Pages) = Num_Rows / (Rows_Per_Page - Free_Rows_Per_Page)

The number of pages estimated should be rounded up to the nearest whole page.

9. Calculate the amount of space required to store the data in a table (8192 total bytes per page):

Table size (bytes) = 8192 x Num_Pages

See Also

Designing Tables

Estimating the Size of a Table Without a Clustered Index

The following steps can be used to estimate the amount of space required to store the data and any additional nonclustered indexes on a table that does not have a clustered index:

- 1. Calculate the space used to store data.
- 2. Calculate the space used to store each additional nonclustered index.
- 3. Sum the values calculated.

For each calculation, specify the number of rows that will be present in the table. The number of rows in the table will have a direct effect on the size of the table:

Number of rows in the table = Num_Rows

Calculate the Space Used to Store Data

To calculate the space used to store data, see **Estimating the Size of a Table**.

Note the value calculated:

```
Space used to store data = Data_Space_Used
```

Calculate the Space Used to Store Each Additional Nonclustered Index

The followings steps can be used to estimate the size of a single nonclustered index on a table that does not have a clustered index:

1. If the index definition includes fixed-length and variable-length columns, calculate the space each of these groups of columns occupies within the index row. The size of a column depends on the data type and length specification. For more information, see <u>Data Types</u>.

Number of columns in index key = Num_Key_Cols

Sum of bytes in all fixed-length key columns = Fixed_Key_Size

Number of variable-length columns in index key = Num_Variable_Key_Cols

Maximum size of all variable-length key columns = Max_Var_Key_Size

2. If there are fixed-length columns in the index, a portion of the index row is reserved for the null bitmap. Calculate its size:

Index Null Bitmap (Index_Null_Bitmap) = 2 + ((Num_Key_Cols + 7)
/ 8)

Only the integer portion of the above expression should be used; discard any remainder.

3. If there are variable-length columns in the index, determine how much space is used to store the columns within the index row:

Total size of variable-length columns (Variable_Key_Size) = 2 + (Num_Variable_Key_Cols x 2) + Max_Var_Key_Size

If there are no variable-length columns, set Variable_Key_Size to 0.

This formula assumes that all variable-length key columns are 100 percent full. If you anticipate that a lower percentage of the variable-length key column storage space will be used, you can adjust the result by that percentage to yield a more accurate estimate of the overall index size.

4. Calculate the index row size:

Total index row size (Index_Row_Size) = Fixed_Key_Size + Variable_Key_Size + Index_Null_Bitmap + 1 + 8

5. Calculate the number of index rows per page (8096 free bytes per page):

Number of index rows per page (Index_Rows_Per_Page) = (8096) / (Index_Row_Size + 2)

Because index rows do not span pages, the number of index rows per page should be rounded down to the nearest whole row.

6. Calculate the number of reserved free index rows per leaf page, based on the fill factor specified for the nonclustered index. For more information, see <u>Fill Factor</u>.

Number of free index rows per leaf page (Free_Index_Rows_Per_Page) = 8096 x ((100 - Fill_Factor) / 100) / Index_Row_Size

The fill factor used in the calculation is an integer value rather than a percentage.

Because index rows do not span pages, the number of index rows per page should be rounded down to the nearest whole row.

7. Calculate the number of pages required to store all the index rows at each level of the index:

Number of pages (level 0) (Num_Pages_Level_0) = Num_Rows / (Index_Rows_Per_Page - Free_Index_Rows_Per_Page)

Number of pages (level 1) (Num_Pages_Level_1) = Num_Pages_Level_0 / Index_Rows_Per_Page

Repeat the second calculation, dividing the number of pages calculated from the previous level *n* by Index_Rows_Per_Page until the number of pages for a given level *n* (Num_Pages_Level_*n*) equals one (root page). For example, to calculate the number of pages required for the second index level:

Number of pages (level 2) (Num_Pages_Level_2) = Num_Pages_Level_1 / Index_Rows_Per_Page

For each level, the number of pages estimated should be rounded up to the nearest whole page.

Sum the number of pages required to store each level of the index:

Total number of pages (Num_Index_Pages) = Num_Pages_Level_0 + Num_Pages_Level_1 + Num_Pages_Level_2 + ... + Num_Pages_Level_n 8. Calculate the size of the index (8192 total bytes per page):Nonclustered index size (bytes) = 8192 x Num_Index_Pages

Calculate the Size of the Table

Calculate the size of the table:

Total table size (bytes) = Data_Space_Used + Nonclustered index size + ...*n*

See Also

<u>Creating an Index</u> <u>Nonclustered Indexes</u>

Estimating the Size of a Table with a Clustered Index

The following steps can be used to estimate the amount of space required to store the data and any additional nonclustered indexes on a table that has a clustered index:

- 1. Calculate the space used to store data.
- 2. Calculate the space used to store the clustered index.
- 3. Calculate the space used to store each additional nonclustered index.
- 4. Sum the values calculated.

For each calculation, specify the number of rows that will be present in the table. The number of rows in your table will have a direct effect on the size of your table:

Number of rows in the table = Num_Rows

Calculate the Space Used to Store Data

For more information about how to calculate the space used to store data, see <u>Estimating the Size of a Table</u>.

Note the value calculated:

Space used to store data = Data_Space_Used

Calculate the Space Used to Store the Clustered Index

The following steps can be used to estimate the amount of space required to store the clustered index:

1. A clustered index definition can include fixed-length and variablelength columns. To estimate the size of the clustered index, you must specify the space each of these groups of columns occupies within the index row:

Number of columns in index key = Num_CKey_Cols

Sum of bytes in all fixed-length key columns = Fixed_CKey_Size

Number of variable-length columns in index key = Num_Variable_CKey_Cols

Maximum size of all variable-length key columns = Max_Var_CKey_Size

2. If there are fixed-length columns in the clustered index, a portion of the index row is reserved for the null bitmap. Calculate its size:

Index Null Bitmap (CIndex_Null_Bitmap) = 2 + ((Num_CKey_Cols + 7) / 8)

Only the integer portion of the above expression should be used; discard any remainder.

3. If there are variable-length columns in the index, determine how much space is used to store the columns within the index row:

Total size of variable length columns (Variable_CKey_Size) = 2 + (Num_Variable_CKey_Cols x 2) + Max_Var_CKey_Size

If there are no variable-length columns, set Variable_CKey_Size to 0.

This formula assumes that all variable-length key columns are 100 percent full. If you anticipate that a lower percentage of the variable-length key column storage space will be used, you can adjust the result by that percentage to yield a more accurate estimate of the overall index size.

4. Calculate the index row size:

Total index row size (CIndex_Row_Size) = Fixed_CKey_Size + Variable_CKey_Size + CIndex_Null_Bitmap + 1 + 8

5. Calculate the number of index rows per page (8096 free bytes per page):

Number of index rows per page (CIndex_Rows_Per_Page) = (8096) / (CIndex_Row_Size + 2)

Because index rows do not span pages, the number of index rows per page should be rounded down to the nearest whole row.

6. Calculate the number of pages required to store all the index rows at each level of the index.

Number of pages (level 0) (Num_Pages_CLevel_0) = (Data_Space_Used / 8192) / CIndex_Rows_Per_Page

Number of pages (level 1) (Num_Pages_CLevel_1) = Num_Pages_CLevel_0 / CIndex_Rows_Per_Page

Repeat the second calculation, dividing the number of pages calculated from the previous level *n* by CIndex_Rows_Per_Page until the number of pages for a given level *n* (Num_Pages_CLevel_*n*) equals one (index root page). For example, to calculate the number of pages required for the second index level:

Number of pages (level 2) (Num_Pages_CLevel_2) = Num_Pages_CLevel_1 / CIndex_Rows_Per_Page

For each level, the number of pages estimated should be rounded up to the nearest whole page.

Sum the number of pages required to store each level of the index:

Total number of pages (Num_CIndex_Pages) = Num_Pages_CLevel_0 + Num_Pages_CLevel_1 + Num_Pages_CLevel_2 + ... + Num_Pages_CLevel_n

7. Calculate the size of the clustered index (8192 total bytes per page):

Clustered index size (bytes) = 8192 x Num_CIndex_Pages

Calculate the Space Used to Store Each Additional Nonclustered Index

The following steps can be used to estimate the amount of space required to store each additional nonclustered index:

1. A nonclustered index definition can include fixed-length and variablelength columns. To estimate the size of the nonclustered index, you must calculate the space each of these groups of columns occupies within the index row:

Number of columns in index key = Num_Key_Cols

Sum of bytes in all fixed-length key columns = Fixed_Key_Size

Number of variable-length columns in index key = Num_Variable_Key_Cols

Maximum size of all variable-length key columns = Max_Var_Key_Size

2. If there are fixed-length columns in the index, a portion of the index row is reserved for the null bitmap. Calculate its size:

Index Null Bitmap (Index_Null_Bitmap) = 2 + ((Num_Key_Cols + 7) / 8)

Only the integer portion of the above expression should be used; discard any remainder.

3. If there are variable-length columns in the index, determine how much space is used to store the columns within the index row:

Total size of variable length columns (Variable_Key_Size) = 2 + (Num_Variable_Key_Cols x 2) + Max_Var_Key_Size

If there are no variable-length columns, set Variable_Key_Size to 0.

This formula assumes that all variable-length key columns are 100 percent full. If you anticipate that a lower percentage of the variable-length key column storage space will be used, you can adjust the result by that percentage to yield a more accurate estimate of the overall index size.

4. Calculate the nonleaf index row size:

Total nonleaf index row size (NL_Index_Row_Size) = Fixed_Key_Size + Variable_Key_Size + Index_Null_Bitmap + 1 + 8

5. Calculate the number of nonleaf index rows per page:

Number of nonleaf index rows per page (NL_Index_Rows_Per_Page) =

```
( 8096 ) / (NL_Index_Row_Size + 2)
```

Because index rows do not span pages, the number of index rows per page should be rounded down to the nearest whole row.

6. Calculate the leaf index row size:

Total leaf index row size (Index_Row_Size) = CIndex_Row_Size + Fixed_Key_Size + Variable_Key_Size + Index_Null_Bitmap + 1

The final value of 1 represents the index row header. CIndex_Row_Size is the total index row size for the clustered index key.

7. Calculate the number of leaf level index rows per page:

Number of leaf level index rows per page (Index_Rows_Per_Page) = (8096) / (Index_Row_Size + 2)

Because index rows do not span pages, the number of index rows per page should be rounded down to the nearest whole row.

8. Calculate the number of reserved free index rows per page based on the fill factor specified for the nonclustered index. For more information, see <u>Fill Factor</u>.

Number of free index rows per page (Free_Index_Rows_Per_Page) = 8096 x ((100 - Fill_Factor) / 100) / Index_Row_Size

The fill factor used in the calculation is an integer value rather than a percentage.

Because index rows do not span pages, the number of index rows per page should be rounded down to the nearest whole row.

9. Calculate the number of pages required to store all the index rows at each level of the index:

Number of pages (level 0) (Num_Pages_Level_0) = Num_Rows / (Index_Rows_Per_Page - Free_Index_Rows_Per_Page)

Number of pages (level 1) (Num_Pages_Level_1) =

Num_Pages_Level_0 / NL_Index_Rows_Per_Page

Repeat the second calculation, dividing the number of pages calculated from the previous level *n* by NL_Index_Rows_Per_Page until the number of pages for a given level *n* (Num_Pages_Level_*n*) equals one (root page).

For example, to calculate the number of pages required for the second and third index levels:

Number of data pages (level 2) (Num_Pages_Level_2) = Num_Pages_Level_1 / NL_Index_Rows_Per_Page

Number of data pages (level 3) (Num_Pages_Level_3) = Num_Pages_Level_2 / NL_Index_Rows_Per_Page

For each level, the number of pages estimated should be rounded up to the nearest whole page.

Sum the number of pages required to store each level of the index:

Total number of pages (Num_Index_Pages) = Num_Pages_Level_0 + Num_Pages_Level_1 +Num_Pages_Level_2 + ... + Num_Pages_Level_n

10. Calculate the size of the nonclustered index:

Nonclustered index size (bytes) = 8192 x Num_Index_Pages

Calculate the Size of the Table

Calculate the size of the table:

• Total table size (bytes) = Data_Space_Used + Clustered index size + Nonclustered index size + ...*n*

See Also

<u>Clustered Indexes</u> <u>Creating an Index</u> Nonclustered Indexes

Creating a Database

To create a database determine the name of the database, its owner (the user who creates the database), its size, and the files and filegroups used to store it.

Before creating a database, consider that:

- Permission to create a database defaults to members of the **sysadmin** and **dbcreator** fixed server roles, although permissions can be granted to other users.
- The user who creates the database becomes the owner of the database.
- A maximum of 32,767 databases can be created on a server.
- The name of the database must follow the rules for identifiers.

Three types of files are used to store a database:

• Primary files

These files contain the startup information for the database. The primary files are also used to store data. Every database has one primary file.

• Secondary files

These files hold all the data that does not fit in the primary data file. Databases do not need secondary data files if the primary file is large enough to hold all the data in the database. Some databases may be large enough to need multiple secondary data files, or they may use secondary files on separate disk drives to spread the data across multiple disks.

• Transaction log

These files hold the log information used to recover the database. There must be at least one transaction log file for each database, although

there may be more than one. The minimum size for a log file is 512 kilobytes (KB).

IMPORTANT Microsoft® SQL ServerTM 2000 data and transaction log files must not be placed on compressed file systems or a remote network drive, such as a shared network directory.

When a database is created, all the files that comprise the database are filled with zeros to overwrite any existing data left on the disk by previously deleted files. Although this means that the files take longer to create, this action prevents the operating system from having to fill the files with zeros when data is written to the files for the first time during usual database operations. This improves the performance of day-to-day operations.

It is recommended that you specify a maximum size to which the file is permitted to grow. This prevents the file from growing, as data is added, until disk space is exhausted. To specify a maximum size for the file, use the MAXSIZE parameter of the CREATE DATABASE statement or the **Restrict filegrowth (MB)** option when using the **Properties** dialog box in SQL Server Enterprise Manager to create the database.

After you create a database, it is recommended that you create a backup of the **master** database.

To create a database

⊞<u>Transact-SQL</u>

Using Raw Partitions

Microsoft® SQL Server[™] 2000 supports the use of raw partitions for creating database files. Raw partitions are disk partitions that have not been formatted with a Microsoft Windows NT® file system, such as FAT and NTFS. In some cases, using databases created on raw partitions can yield a slight performance gain over NTFS or FAT. However, for most installations the preferred method is to use files created on NTFS or FAT partitions.

When creating a database file on a raw partition, you do not specify the physical names of the files comprising the database; you specify only the drive letters of the disks on which the database files should be created.

If you are using Microsoft Windows® 2000 Server, you can create mounted drives to point to raw partitions. When you mount a local drive at an empty folder, Windows 2000 assigns a drive path to the drive rather than a drive letter. Mounted drives are not subject to the 26-drive limit imposed by drive letters; therefore, you can use an unlimited number of raw partitions. When you create a database file on a mounted drive, you must end the drive path to the file name with a trailing backslash (\), for example, E:\Sample name\. For information about creating a mounted drive, see the Windows 2000 Server documentation.

There are several limitations to consider when using raw partitions:

- Only one database file can be created on each raw partition. The logical partition must be configured as a single database file, because there is no file system on the raw partition.
- Standard file-system operations such as copy, move, and delete cannot be used with raw partitions.
- Database files located on raw partitions cannot be backed up using the Windows NT Backup utility. However, SQL Server database or transaction log backups can still be created.
- Database files on raw partitions cannot be automatically expanded.

Either initially create the database at its full size, or manually expand the database files. For more information, see <u>Expanding a Database</u>.

- Only lettered partitions, such as E:, or mounted drives, such as E:\Sample name\ can be used. Numbered devices cannot be used.
- File-system services such as bad block replacement are not available with raw partitions.

See Also

CREATE DATABASE

Modifying a Database

After a database is created, changes can be made to its original definition. Changes can include:

- Expanding the data or transaction log space allocated to the database.
- Shrinking the data and transaction log space allocated to the database.
- Adding or removing data and transaction log files.
- Creating filegroups.
- Changing the default filegroup.
- Change the configuration settings for the database.
- Placing databases offline.
- Attaching new and detaching unused databases.
- Changing the name of the database.
- Changing the owner of the database.

Before changes are made to the database, it is sometimes necessary to take the database out of normal operating mode. In those situations, determine the appropriate method for terminating transactions.

Expanding a Database

Microsoft® SQL Server[™] 2000 can automatically expand a database according to growth parameters defined when the database was created. You can also manually expand a database by allocating additional file space on an existing database file or allocating space on another new file. You may need to expand the data or transaction log space if the existing files are becoming full. If a database has already exhausted the space allocated to it and it cannot grow automatically, Error 1105 is raised.

When expanding a database, you must increase the size of the database by at least 1 megabyte (MB). Permission for expanding a database defaults to the database owner and is automatically transferred with database ownership. When a database is expanded, the new space is immediately made available to either the data or transaction log file, depending on which file was expanded.

If the transaction log is not set up to expand automatically, it can run out of space if certain types of activity occur in the database. The transaction log is purged only of inactive (committed) transactions when it is backed up, or at each checkpoint when the database is using the simple recovery model. SQL Server can then reuse this truncated, unused portion of the transaction log. For more information about truncating the transaction log, see <u>Truncating the Transaction Log</u>.

SQL Server does not truncate the transaction log when backing up the database.

When you expand a database, it is recommended that you specify a maximum size to which the file is permitted to grow. This prevents the file from growing until disk space is exhausted. To specify a maximum size for the file, use the MAXSIZE parameter of the ALTER DATABASE statement or the **Restrict filegrowth (MB)** option when using the **Properties** dialog box in SQL Server Enterprise Manager to expand the database.

Expanding a database to increase space for data or the transaction log follows the same process.

Expanding tempdb

By default, the **tempdb** database automatically grows as space is needed because the MAXSIZE of the files is set to UNLIMITED. Therefore, **tempdb** can continue growing until space on the disk that contains **tempdb** is exhausted. To prevent **tempdb** from growing without limits, set a MAXSIZE for **tempdb** by using the ALTER DATABASE statement or SQL Server Enterprise Manager.

Conversely, if **tempdb** has been set at a MAXSIZE, and you want to increase the size of **tempdb**, you must do one of the following:

- Increase the size of the files in the default filegroup currently used by **tempdb**.
- Add a new file to the default filegroup.
- Allow the files used by **tempdb** to grow automatically.

IMPORTANT User-defined filegroups cannot be used with **tempdb**. They can be used only with the default filegroup.

Moving tempdb

To change the physical location of the **tempdb** database:

- 1. Alter the **tempdb** database, using the ALTER DATABASE statement and MODIFY FILE clause, to change the physical file names of each file in **tempdb** to reference the new physical location, such as the new disk.
- 2. Stop and restart SQL Server.
- 3. Delete the old **tempdb** database files from the original location.

To increase the size of a database

⊞ <u>Transact-SQL</u>

Shrinking a Database

Microsoft® SQL Server[™] 2000 allows each file within a database to be shrunk to remove unused pages. Both data and transaction log files can be shrunk. The database files can be shrunk manually, either as a group or individually. The database can be set to shrink automatically at given intervals. This activity occurs in the background and does not affect any user activity within the database.

When the database is set to shrink automatically using the ALTER DATABASE AUTO_SHRINK option (or the **sp_dboption** system stored procedure), shrinking occurs when a significant amount of free space is available in the database. However, if the percentage of free space to be removed cannot be configured, as much free space as possible is removed. To configure the amount of free space to be removed, such as only 50 percent of the current free space in the database, use the **Properties** dialog box in SQL Server Enterprise Manager to shrink the database.

You cannot shrink an entire database to be smaller than its original size. Therefore, if a database was created with a size of 10 megabytes (MB) and grew to 100 MB, the smallest the database could be shrunk to, assuming all the data in the database has been deleted, is 10 MB.

However, you can shrink the individual database files smaller than their initial size by using the DBCC SHRINKFILE statement. You must shrink each file individually, rather than attempting to shrink the entire database.

There are fixed boundaries from which a transaction log file can be shrunk. The size of the virtual log determines the possible reduction in size. Therefore, the log file can never be shrunk to a size less than the virtual log file. In addition, the log file is shrunk in increments equal to the size of the virtual log file. For example, a transaction log file of 1 gigabyte (GB) may comprise five virtual log files of 200 MB each. Shrinking the transaction log file deletes unused virtual log files, but leaves at least one virtual log file. Because each virtual log file in this example is 200 MB, the transaction log can shrink only to a minimum of 200 MB and can shrink only in increments of 200 MB. To allow a transaction log file to shrink to a smaller size, create a smaller transaction log file.

In SQL Server 2000, a DBCC SHRINKDATABASE or DBCC SHRINKFILE operation attempts to shrink a transaction log file to the requested size (subject to rounding) immediately. You should truncate the log file prior to shrinking the file to reduce the size of the logical log and mark as inactive virtual logs that do not hold any part of the logical log. For more information, see <u>Shrinking the Transaction Log</u>.

Note It is not possible to shrink the database or transaction log while the database or transaction log is being backed up. Conversely, it is not possible to create a database or transaction log backup while the database or transaction log is being shrunk.

To shrink a database

⊞ <u>Transact-SQL</u>

Adding and Deleting Data and Transaction Log Files

Data and transaction log files can be added to expand a database or deleted to shrink a database. When a file is added, the file is available immediately for use by the database.

IMPORTANT Microsoft[®] SQL Server[™] 2000 data and transaction log files must not be placed on compressed file systems.

SQL Server uses a proportional fill strategy across all the files within each filegroup, writing an amount of data proportional to the free space in the file and allowing the new file starts to be used immediately. This way all files tend to become full at about the same time. Transaction log files, however, cannot be part of a filegroup; they are separate from one another. As the transaction log grows, the first log file fills, then the second, and so on, using a fill-and-go strategy rather than a proportional fill strategy. Therefore, when a log file is added, it cannot be used by the transaction log until the other files have been filled first.

When adding files to the database, you can specify the size of the file (default is 1 MB), the maximum size to which the file should grow if space within the file is exhausted, the amount by which the file grows each time it needs to grow (default is 10 percent), and the filegroup to which the file belongs, as appropriate.

Deleting a data or transaction log file removes the file from the database. It is not possible to remove a file from the database unless there is no existing data or transaction log information on the file; the file must be completely empty before it can be removed. To migrate data from a data file to other files in the same filegroup, use the DBCC SHRINKFILE statement and specify the EMPTYFILE clause. SQL Server no longer allows data to be placed on the file, thereby allowing it to be deleted by using the ALTER DATABASE statement or the property page within SQL Server Enterprise Manager.

It is not possible to migrate the transaction log data from one log file to another to delete a transaction log file. To purge inactive transactions from a transaction log file, the transaction log must be truncated or backed up. When the transaction log file no longer contains any active or inactive transactions, the log file can be removed from the database.

IMPORTANT After you add or delete files, create a database backup immediately. A transaction log backup should not be created until after a full database backup is created.

To add data or transaction log files to a database

∃ <u>Transact-SQL</u>

Creating Filegroups

<u>Filegroups</u> can be created when the database is first created or later when more files are added to the database. However, it is not possible to move files to a different filegroup after the files have been added to the database.

A file cannot be a member of more than one filegroup. Tables, indexes, and **text**, **ntext**, and **image** data can be associated with a specific filegroup. This means that all their pages are allocated from the files in that filegroup.

There are three types of filegroups:

• Primary filegroup

This filegroup contains the primary data file and any other files not placed into another filegroup. All pages for the system tables are allocated from the primary filegroup.

• User-defined filegroup

This filegroup is any filegroup specified using the FILEGROUP keyword in a CREATE DATABASE or ALTER DATABASE statement, or on the **Properties** dialog box within SQL Server Enterprise Manager.

• Default filegroup

The default filegroup contains the pages for all tables and indexes that do not have a filegroup specified when they are created. In each database, only one filegroup at a time can be the default filegroup. If no default filegroup is specified, the default is the primary filegroup.

A maximum of 256 filegroups can be created for each database. Filegroups can contain only data files. Transaction log files cannot be part of a filegroup.

Note Filegroups cannot be created independently of database files. The filegroup is an administrative mechanism of grouping files within the database.

To add a filegroup when creating a database

⊞ <u>Transact-SQL</u>

Changing the Default Filegroup

When you change the default filegroup, any objects for which no filegroups have been initially specified are allocated to the data files in the new default filegroup. Changing the default filegroup prevents user objects that are not specifically created on a user-defined filegroup from competing with the system objects and tables for data space.

To change the default filegroup

⊞ <u>Transact-SQL</u>

∃ <u>SQL-DMO</u>

See Also

Default Filegroups

Setting Database Options

A number of database-level options that determine the characteristics of the database can be set for each database. Only the system administrator, database owner, members of the **sysadmin** and **dbcreator** fixed server roles and **db_owner** fixed database roles can modify these options. These options are unique to each database and do not affect other databases. The database options can be set by using the SET clause of the ALTER DATABASE statement, the **sp_dboption** system stored procedure or, in some cases, SQL Server Enterprise Manager.

Note Server-wide settings are set using the **sp_configure** system stored procedure or SQL Server Enterprise Manager. For more information, see <u>Setting</u> <u>Configuration Options</u>. Connection-level settings are specified by using SET statements. For more information, see <u>SET Options</u>.

After you set a database option, a checkpoint is automatically issued that causes the modification to take effect immediately.

To change the default values for any of the database options for newly created databases, change the appropriate database option in the **model** database. For example, if you want the default setting of the AUTO_SHRINK database option to be ON for any new databases subsequently created, set the AUTO_SHRINK option for **model** to ON.

There are five categories of database options:

- Auto options
- Cursor options
- Recovery options
- SQL options
- State options

Auto Options

Auto options control certain automatic behaviors.

AUTO_CLOSE

When set to ON, the database is closed and shut down cleanly when the last user of the database exits and all processes in the database complete, thereby freeing any resources. By default, this option is set to ON for all databases when using Microsoft® SQL ServerTM 2000 Desktop Engine, and OFF for all other editions, regardless of operating system. The database reopens automatically when a user tries to use the database again. If the database was shut down cleanly, the database is not reopened until a user tries to use the database the next time SQL Server is restarted. When set to OFF, the database remains open even if no users are currently using the database.

The AUTO_CLOSE option is useful for desktop databases because it allows database files to be managed as normal files. They can be moved, copied to make backups, or even e-mailed to other users. The AUTO_CLOSE option should not be used for databases accessed by an application that repeatedly makes and breaks connections to SQL Server. The overhead of closing and reopening the database between each connection will impair performance.

The status of this option can be determined by examining the **IsAutoClose** property of the DATABASEPROPERTYEX function.

AUTO_CREATE_STATISTICS

When set to ON, statistics are automatically created on columns used in a predicate. Adding statistics improves query performance because the SQL Server query optimizer can better determine how to evaluate a query. If the statistics are not used, SQL Server automatically deletes them. When set to OFF, statistics are not automatically created by SQL Server; instead, statistics can be manually created. For more information, see <u>Statistical Information</u>.

By default, AUTO_CREATE_STATISTICS is ON.

The status of this option can be determined by examining the **IsAutoCreateStatistics** property of the DATABASEPROPERTYEX function.

AUTO_UPDATE_STATISTICS

When set to ON, existing statistics are automatically updated when the statistics become out-of-date because the data in the tables has changed. When set to OFF, existing statistics are not automatically updated; instead, statistics can be manually updated. For more information, see <u>Statistical Information</u>.

By default, AUTO_UPDATE_STATISTICS is set to ON.

The status of this option can be determined by examining the **IsAutoUpdateStatistics** property of the DATABASEPROPERTYEX function.

AUTO_SHRINK

When set to ON, the database files are candidates for periodic shrinking. Both data file and log files can be shrunk automatically by SQL Server. When set to OFF, the database files are not automatically shrunk during periodic checks for unused space. By default, this option is set to ON for all databases when using SQL Server Desktop Edition, and OFF for all other editions, regardless of operating system.

AUTO_SHRINK only reduces the size of the transaction log if the database is set to SIMPLE recovery model or if the log is backed up.

The AUTO_SHRINK option causes files to be shrunk when more than 25 percent of the file contains unused space. The file is shrunk to a size where 25 percent of the file is unused space, or to the size of the file when it was created, whichever is greater.

It is not possible to shrink a read-only database.

The status of this option can be determined by examining the **IsAutoShrink** property of the DATABASEPROPERTYEX function.

Cursor Options

Cursor options control cursor behavior and scope.

CURSOR_CLOSE_ON_COMMIT

When set to ON, any open cursors are closed automatically (in compliance

with SQL-92) when a transaction is committed. By default, this setting is OFF and cursors remain open across transaction boundaries, closing only when the connection is closed or when they are explicitly closed.

Connection-level settings (set using the SET statement) override the default database setting for CURSOR_CLOSE_ON_COMMIT. By default, ODBC and OLE DB clients issue a connection-level SET statement setting CURSOR_CLOSE_ON_COMMIT to OFF for the session when connecting to SQL Server. For more information, see <u>SET</u> <u>CURSOR_CLOSE_ON_COMMIT</u>.

The status of this option can be determined by examining the **IsCloseCursorsOnCommitEnabled** property of the DATABASEPROPERTYEX function.

CURSOR_DEFAULT LOCAL | GLOBAL

When CURSOR_DEFAULT LOCAL is set, and a cursor is not defined as GLOBAL when it is created, the scope of the cursor is local to the batch, stored procedure, or trigger in which the cursor was created. The cursor name is valid only within this scope. The cursor can be referenced by local cursor variables in the batch, stored procedure, or trigger, or a stored procedure OUTPUT parameter. The cursor is implicitly deallocated when the batch, stored procedure, or trigger terminates, unless it was passed back in an OUTPUT parameter. If it is passed back in an OUTPUT parameter, the cursor is deallocated when the last variable referencing it is deallocated or goes out of scope.

When CURSOR_DEFAULT GLOBAL is set, and a cursor is not defined as LOCAL when created, the scope of the cursor is global to the connection. The cursor name can be referenced in any stored procedure or batch executed by the connection. The cursor is implicitly deallocated only at disconnect. CURSOR_DEFAULT GLOBAL is the default setting. For more information, see <u>DECLARE CURSOR</u>.

The status of this option can be determined by examining the **IsLocalCursorsDefault** property of the DATABASEPROPERTYEX function.

Recovery Options

Recovery options controls the recovery model for the database.

RECOVERY FULL | BULK_LOGGED | SIMPLE

When FULL is specified, database backups and transaction log backups are used to provide full recoverability from media failure. All operations, including bulk operations such as SELECT INTO, CREATE INDEX, and bulk loading data, are fully logged. For more information, see <u>Full Recovery</u>.

When BULK_LOGGED is specified, logging for all SELECT INTO, CREATE INDEX, and bulk loading data operations is minimal and therefore requires less log space. In exchange for better performance and less log space usage, the risk of exposure to loss is greater than with full recovery. For more information, see <u>Bulk-Logged Recovery</u>.

When SIMPLE is specified, the database can be recovered only to the last full database backup or last differential backup. For more information, see <u>Simple Recovery</u>.

SIMPLE is the default setting for SQL Server Desktop Edition and the data engine, and FULL is the default for all other editions.

The status of this option can be determined by examining the **Recovery** property of the DATABASEPROPERTYEX function.

TORN_PAGE_DETECTION

This recovery option allows SQL Server to detect incomplete I/O operations caused by power failures or other system outages.

When set to ON, this option causes a bit to be reversed for each 512-byte sector in an 8-kilobyte (KB) database page when the page is written to disk. If a bit is in the wrong state when the page is later read by SQL Server, the page was written incorrectly; a torn page is detected. Torn pages are usually detected during recovery because any page that was written incorrectly is likely to be read by recovery.

Although SQL Server database pages are 8 KB, disks perform I/O operations using a 512-byte sector. Therefore, 16 sectors are written per database page. A torn page can occur if the system fails (for example, due to power failure) between the time the operating system writes the first 512-byte sector to disk and the completion of the 8-KB I/O operation. If the first sector of a database

page is successfully written before the failure, the database page on disk will appear as updated, although it may not have succeeded.

Note Using battery-backed disk caches can ensure that data is successfully written to disk or not written at all.

If a torn page is detected, an I/O error is raised and the connection is killed. If the torn page is detected during recovery, the database is also marked suspect. The database backup should be restored, and any transaction log backups applied, because it is physically inconsistent.

By default, TORN_PAGE_DETECTION is ON.

The current setting of this option can be determined by examining the **IsTornPageDetectionEnabled** property of DATABASEPROPERTYEX.

SQL Options

SQL options control ANSI compliance options.

ANSI_NULL_DEFAULT

Allows the user to control the database default nullability. When NULL or NOT NULL is not specified explicitly, a user-defined data type or a column definition uses the default setting for nullability. Nullability is determined by session and database settings. Microsoft SQL ServerTM2000 defaults to NOT NULL. For ANSI compatibility, setting the database option ANSI_NULL_DEFAULT to ON changes the database default to NULL.

When this option is set to ON, all user-defined data types or columns that are not explicitly defined as NOT NULL during a CREATE TABLE or ALTER TABLE statement default to allowing null values. Columns that are defined with constraints follow constraint rules regardless of this setting.

Connection-level settings (set using the SET statement) override the default database-level setting for ANSI_NULL_DEFAULT. By default, ODBC and OLE DB clients issue a connection-level SET statement setting ANSI_NULL_DEFAULT to ON for the session when connecting to SQL Server. For more information, see <u>SET ANSI_NULL_DFLT_ON</u>.

The status of this option can be determined by examining the

IsAnsiNullDefault property of the DATABASEPROPERTYEX function.

ANSI_NULLS

When set to ON, all comparisons to a null value evaluate to NULL (unknown). When set to OFF, comparisons of non-Unicode values to a null value evaluate to TRUE if both values are NULL. By default, the ANSI_NULLS database option is OFF.

Connection-level settings (set using the SET statement) override the default database setting for ANSI_NULLS. By default, ODBC and OLE DB clients issue a connection-level SET statement setting ANSI_NULLS to ON for the session when connecting to SQL Server. For more information, see <u>SET ANSI_NULLS</u>.

SET ANSI_NULLS also must be set to ON when you create or manipulate indexes on computed columns or indexed views.

The status of this option can be determined by examining the **IsAnsiNullsEnabled** property of the DATABASEPROPERTYEX function.

ANSI_PADDING

When set to ON, trailing blanks in character values inserted into **varchar** columns and trailing zeros in binary values inserted into **varbinary** columns are not trimmed. Values are not padded to the length of the column. When set to OFF, the trailing blanks (for **varchar**) and zeros (for **varbinary**) are trimmed. This setting affects only the definition of new columns.

Char(*n*) and **binary**(*n*) columns that allow nulls are padded to the length of the column when SET ANSI_PADDING is set to ON, but trailing blanks and zeros are trimmed when SET ANSI_PADDING is OFF. **Char**(*n*) and **binary**(*n*) columns that do not allow nulls are always padded to the length of the column.

IMPORTANT It is recommended that ANSI_PADDING always be set to ON. SET ANSI_PADDING must be ON when creating or manipulating indexes on computed columns or indexed views.

The status of this option can be determined by examining the **IsAnsiPaddingEnabled** property of the DATABASEPROPERTYEX function.

ANSI_WARNINGS

When set to ON, errors or warnings are issued when conditions such as "divide by zero" occur or null values appear in aggregate functions. When set to OFF, no warnings are raised when null values appear in aggregate functions, and null values are returned when conditions such as "divide by zero" occur. By default, ANSI_WARNINGS is OFF.

SET ANSI_WARNINGS must be set to ON when you create or manipulate indexes on computed columns or indexed views.

Connection-level settings (set using the SET statement) override the default database setting for ANSI_WARNINGS. By default, ODBC and OLE DB clients issue a connection-level SET statement setting ANSI_WARNINGS to ON for the session when connecting to SQL Server. For more information, see <u>SET ANSI_WARNINGS</u>.

The status of this option can be determined by examining the **IsAnsiWarningsEnabled** property of the DATABASEPROPERTYEX function.

ARITHABORT

When set to ON, an overflow or divide-by-zero error causes the query or batch to terminate. If the error occurs in a transaction, the transaction is rolled back. When set to OFF, a warning message is displayed if one of these errors occurs, but the query, batch, or transaction continues to process as if no error occurred.

SET ARITHABORT must be set to ON when you create or manipulate indexes on computed columns or indexed views

The status of this option can be determined by examining the **IsArithmeticAbortEnabled** property of the DATABASEPROPERTYEX function.

NUMERIC_ROUNDABORT

If set to ON, an error is generated when loss of precision occurs in an expression. When set to OFF, losses of precision do not generate error messages and the result is rounded to the precision of the column or variable storing the result.

SET NUMERIC_ROUNDABORT must be set to OFF when you create or manipulate indexes on computed columns or indexed views.

The status of this option can be determined by examining the **IsNumericRoundAbortEnabled** property of the DATABASEPROPERTYEX function.

CONCAT_NULL_YIELDS_NULL

When set to ON, if one of the operands in a concatenation operation is NULL, the result of the operation is NULL. For example, concatenating the character string "This is" and NULL results in the value NULL, rather than the value "This is".

When set to OFF, concatenating a null value with a character string yields the character string as the result; the null value is treated as an empty character string. By default, CONCAT_NULL_YIELDS_NULL is OFF.

SET CONCAT_NULL_YIELDS_NULL must be set to ON when you create or manipulate indexes on computed columns or indexed views.

Connection-level settings (set using the SET statement) override the default database setting for CONCAT_NULL_YIELDS_NULL. By default, ODBC and OLE DB clients issue a connection-level SET statement setting CONCAT_NULL_YIELDS_NULL to ON for the session when connecting to SQL Server. For more information, see <u>SET</u> <u>CONCAT_NULL_YIELDS_NULL</u>.

The status of this option can be determined by examining the **IsNullConcat** property of the DATABASEPROPERTYEX function.

QUOTED_IDENTIFIER

When set to ON, identifiers can be delimited by double quotation marks and literals must be delimited by single quotation marks. All strings delimited by double quotation marks are interpreted as object identifiers. Quoted identifiers do not have to follow the Transact-SQL rules for identifiers. They can be keywords and can include characters not generally allowed in Transact-SQL identifiers. If a single quotation mark (') is part of the literal string, it can be represented by double quotation marks ('').

When set to OFF (default), identifiers cannot be in quotation marks and must

follow all Transact-SQL rules for identifiers. Literals can be delimited by either single or double quotation marks.

SQL Server also allows identifiers to be delimited by square brackets ([]). Bracketed identifiers can always be used, regardless of the setting of QUOTED_IDENTIFIER. For more information, see <u>Delimited Identifiers</u>.

SET QUOTED_IDENTIFIER must be set to ON when you create or manipulate indexes on computed columns or indexed views.

Connection-level settings (set using the SET statement) override the default database setting for QUOTED_IDENTIFIER. By default, ODBC and OLE DB clients issue a connection-level SET statement setting QUOTED_IDENTIFIER to ON when connecting to SQL Server. For more information, see <u>SET QUOTED_IDENTIFIER</u>.

The status of this option can be determined by examining the **IsQuotedIdentifiersEnabled** property of the DATABASEPROPERTYEX function.

RECURSIVE_TRIGGERS

When set to ON, triggers are allowed to fire recursively. When set to OFF (default), triggers cannot be fired recursively.

Note Only direct recursion is prevented when RECURSIVE_TRIGGERS is set to OFF. To disable indirect recursion, you must also set the **nested triggers** server option to **0**.

The status of this option can be determined by examining the **IsRecursiveTriggersEnabled** property of the DATABASEPROPERTYEX function.

State Options

State options control whether the database is online or offline, who can connect to the database, and whether the database is in read-only mode. A termination clause can be used to control how connections are terminated when the database is transitioned from one state to another.

OFFLINE | ONLINE

When OFFLINE is specified, the database is closed and shutdown cleanly and marked offline. The database cannot be modified while the database is offline.

When ONLINE is specified, the database is open and available for use. ONLINE is the default setting.

The status of this option can be determined by examining the **Status** property of the DATABASEPROPERTYEX function.

READ_ONLY | READ_WRITE

When READ_ONLY is specified, the database is in read-only mode. Users can retrieve data from the database, but cannot modify the data. Because a read-only database does not allow data modifications:

- Automatic recovery is skipped at system startup.
- Shrinking the database is not possible.
- No locking takes place in read-only databases, which can result in faster query performance.

When READ_WRITE is specified, users can retrieve and modify data. READ_WRITE is the default setting.

The status of this option can be determined by examining the **Updateability** property of the DATABASEPROPERTYEX function.

SINGLE_USER | RESTRICTED_USER | MULTI_USER

SINGLE_USER allows one user at a time to connect to the database. All other user connections are broken. The timeframe for breaking the connection is controlled by the termination clause of the ALTER DATABASE statement. New connection attempts are refused. The database remains in SINGLE_USER mode even if the user who set the option logs off. At that point, a different user (but only one) can connect to the database.

To allow multiple connections, the database must be changed to RESTRICTED_USER or MULTI_USER mode.

RESTRICTED_USER allows only members of the **db_owner** fixed database role and **dbcreator** and **sysadmin** fixed server roles to connect to the database, but it does not limit their number. Users who are not members of these roles are disconnected in the timeframe specified by the termination clause of the ALTER DATABASE statement. Moreover, new connection attempts by unqualified users are refused.

MULTI_USER allows all users with the appropriate permissions to connect to the database. MULTI_USER is the default setting.

The status of this option can be determined by examining the **UserAccess** property of the DATABASEPROPERTYEX function.

WITH <termination>

The termination clause of the ALTER DATABASE statement specifies how to terminate incomplete transactions when the database is to be transitioned from one state to another. Transactions are terminated by breaking their connections to the database. If the termination clause is omitted, the ALTER DATABASE statement waits indefinitely, until the transactions commit or roll back on their own.

ROLLBACK AFTER integer [SECONDS]

ROLLBACK AFTER *integer* SECONDS waits for the specified number of seconds and then breaks unqualified connections. Incomplete transactions are rolled back. When the transition is to SINGLE_USER mode, unqualified connections are all connections except the one issuing the ALTER DATABASE statement. When the transition is to RESTRICTED_USER mode, unqualified connections are connections for users who are not members of the **db_owner** fixed database role and **dbcreator** and **sysadmin** fixed server roles.

ROLLBACK IMMEDIATE

ROLLBACK IMMEDIATE breaks unqualified connections immediately. All incomplete transactions are rolled back. Unqualified connections are the same as those described for ROLLBACK AFTER *integer* SECONDS.

NO_WAIT

NO_WAIT checks for connections before attempting to change the

database state and causes the ALTER DATABASE statement to fail if certain connections exist. When the transition is to SINGLE_USER mode, the ALTER DATABASE statement fails if any other connections exist. When the transition is to RESTRICTED_USER mode, the ALTER DATABASE statement fails if any unqualified connections exist.

To change database options

∃ <u>Transact-SQL</u>

Creating a Removable Database

In Microsoft® SQL Server[™] 2000, you can create a database for read-only purposes that can be distributed by way of removable media, such as CD-ROM. This can be useful for distributing large databases containing history data, such as a database containing detailed sales data for the last year.

To create a removable media database, you create the database using the **sp_create_removable** system stored procedure rather than using SQL Server Enterprise Manager or the CREATE DATABASE statement.

The **sp_create_removable** system stored procedure creates three or more files:

- One file containing the system tables
- One file containing the transaction log
- One or more files containing the data tables

Although the database itself is likely to remain on the read-only media, such as CD-ROM, the system tables and transaction log are placed in separate files on writable media so that management tasks can be accomplished, such as adding users to the database, granting permissions, and so on.

A database can use multiple removable media devices. However, all media must be available simultaneously. For example, if a database uses three compact discs, then the system must have three CD-ROM drives and have all discs available when the database is used.

After the database has been created, you can use the **sp_certify_removable** system stored procedure to ensure that the database is configured properly for distribution on removable media. If the database is configured correctly, the database is placed offline, allowing the files to be copied to the removable media. By placing the database offline, users are prevented from accessing the database, and no modifications to the database can be made until the database is placed online. To make the database available again on the same server, place the database online.

After the files have been distributed on removable media, the database can be made available by attaching the files to a different instance of SQL Server. For more information, see <u>Attaching and Detaching a Database</u>.

To place a database online or offline

⊞<u>Transact-SQL</u>

∃ <u>SQL-DMO</u>

See Also

<u>sp_certify_removable</u> <u>sp_create_removable</u>

Attaching and Detaching a Database

In Microsoft® SQL Server[™] 2000, the data and transaction log files of a database can be detached and then reattached to another server, or even to the same server. Detaching a database removes the database from SQL Server but leaves the database intact within the data and transaction log files that compose the database. These data and transaction log files can then be used to attach the database to any instance of SQL Server, including the server from which the database was detached. This makes the database available in exactly the same state it was in when it was detached.

Detaching and attaching databases is useful if you want to move a database:

- From one computer to another without having to re-create the database and then restore the database backup manually.
- To a different physical disk, for example, when the disk containing the database file has run out of disk space and you want to expand the existing file rather than add a new file to the database on the other disk.

To move a database, or database file, to another server or disk:

- 1. Detach the database.
- 2. Move the database file(s) to the other server or disk.
- 3. Attach the database specifying the new location of the moved file(s).

When you attach a database, the name and physical location of the primary data file must be specified. The primary file contains the information needed to find the other files comprising the database unless one or more of those files have changed location since the database was detached. Any files that have changed location must be specified in addition to the primary file. Otherwise, SQL Server tries to attach the files based on incorrect file location information stored in the primary file, and the database will not be successfully attached. If you attach a database to a server other than the server from which the database was detached, and the detached database was enabled for replication, you should run **sp_removedbreplication** to remove replication from the database.

Alternatively, you can remove replication from the database prior to detaching it.

Errors produced while detaching a database may prevent both the database from closing cleanly and the transaction log from being rebuilt. If you receive an error message, perform these corrective actions:

- 1. Reattach all files associated with the database, not just the primary file.
- 2. Resolve the problem that caused the error message.
- 3. Detach the database again.

To attach a database

⊞ <u>Transact-SQL</u>

Attaching a Single-File Database

If a database comprises only a single data file and a single transaction log file, the database can be attached to an instance of Microsoft® SQL Server[™] 2000 without using the transaction log file, provided the database was cleanly shut down with no users and no open transactions. When the data file is attached, SQL Server creates a new transaction log file automatically.

The database must have been successfully detached from SQL Server using the **sp_detach_db** system stored procedure.

Single-file databases are useful if you want to e-mail databases to other users. All the data is stored in a single file; attaching the single file to SQL Server automatically re-creates a transaction log so that the database can be used.

To attach a single-file database

⊞ <u>Transact-SQL</u> ⊞ <u>SQL-DMO</u>

Renaming a Database

In Microsoft® SQL Server[™] 2000, you can change the name of a database to be changed. Before you rename a database, you should make sure that no one is using the database and that the database is set to single-user mode. The name of the database can include any characters as long as they follow the rules for identifiers.

To rename a database

⊞ <u>Transact-SQL</u>

∃ <u>SQL-DMO</u>

See Also

Using Identifiers

Changing the Database Owner

In Microsoft[®] SQL Server[™] 2000, the owner of the current database can be changed. Any user (SQL Server login or Microsoft Windows NT[®] user) who has access to connect to SQL Server can become the owner of a database.

Ownership of the system databases (**master**, **model**, and **tempdb**) cannot be changed.

To change the owner of a database

⊞ <u>Transact-SQL</u>

∃ <u>SQL-DMO</u>

Transaction Termination for Changing Database States

Microsoft® SQL Server[™] 2000 includes the ability to easily shut down or otherwise change the state of a database, automatically terminating the sessions of affected users and rolling back the associated transactions. Affected sessions may be terminated immediately, or may be allowed to continue to their normal conclusion with an optional time-out.

It is often necessary to stop or restrict activity on a database to perform maintenance or other operations without taking down the server, for example:

- Single-user mode: only one user is allowed
- Restricted-user mode: only members of the **db_owner**, **dbcreator**, or **sysadmin** roles are allowed
- Offline: the database is offline
- Read-only mode: no changes are allowed

Transitioning into any of these states requires the termination of transactions and the associated sessions that do not meet the requirements of the new state.

There are three types of transaction termination:

• Normal

New transactions are prevented from starting. Incomplete transactions are allowed to commit or rollback on their own.

• Normal with time-out

New transactions are prevented from starting. Incomplete transactions are allowed to commit or roll back on their own until the time-out is reached, at which time they are rolled back.

• Immediate

An immediate termination prevents new transactions from starting, and rolls back incomplete transactions unconditionally.

The user initiating the change remains connected and able to perform further commands.

Use the ALTER DATABASE statement to specify the database state and a transaction termination type.

See Also

ALTER DATABASE

Viewing a Database

You can view the definition of a database and its configuration settings when you are troubleshooting or considering making changes to a database.

To view a database

∃ <u>Transact-SQL</u>

Displaying Database and Transaction Log Space

Microsoft® SQL Server[™] 2000 can display the number of rows, reserved disk space, and disk space used by a table in a database. SQL Server can also display the reserved disk space that is used by an entire database as well as statistics about the use of transaction log space in a database. This indicates how much data is in the database, whether the database must be expanded (if autogrow is not permitted), and how fast the database is growing (if you maintain a history of the data space that is used).

To display data space information for a database

⊞ <u>Transact-SQL</u>

Documenting and Scripting Databases

With Microsoft® SQL Server[™] 2000, you can document an existing database structure (schema) by generating one or more SQL scripts. An SQL script can be viewed in SQL Server Enterprise Manager, SQL Query Analyzer, or by using any text editor.

A schema generated as an SQL script can be used in many ways, including:

- To maintain a backup script that will allow the user to re-create all users, groups, logins, and permissions.
- To create or update database development code.
- To create a test or development environment from an existing schema.
- To train newly hired employees.

SQL scripts contain descriptions of the statements used to create a database and its objects. You can generate scripts from the objects in an existing database, and then add these objects to another database by running the scripts against that database. In effect, this re-creates the whole database structure and any individual database objects. The schema of the following objects can be generated and saved as a script.

Tables	User-defined data types
Indexes	Triggers
Views	Users, groups, and roles
Stored procedures	Logins
Defaults	Rules
Table keys/declarative referential	Object-level permissions
integrity (DRI)	
Full-text indexes	

The schema for the objects generated can be saved in a single SQL Script file, or in several files with each file containing the schema of just one object. You can also save the schema generated for a single object (or a group of objects) into one or more SQL script files. Examples of SQL script files that you can generate include:

- An entire database saved into a single SQL script file.
- Table-only schema for one, some, or all tables in a database saved into one or more SQL script files.
- Table and index schema saved into one SQL script file, stored procedures saved into another SQL script file, and defaults and rules saved into yet another SQL script file.

To generate a script

Database Maintenance Plan Wizard

The Database Maintenance Plan Wizard can be used to help you set up the core maintenance tasks necessary to ensure that your database performs well, is regularly backed up in case of system failure, and is checked for inconsistencies. The Database Maintenance Plan Wizard creates a Microsoft® SQL ServerTM 2000 job that performs these maintenance tasks automatically at scheduled intervals.

The maintenance tasks that can be scheduled to run automatically are:

- Reorganizing the data on the data and index pages by rebuilding indexes with a new fill factor. This ensures that database pages contain an equally distributed amount of data and free space, which allows future growth to be faster. For more information, see <u>Fill Factor</u>.
- Compressing data files by removing empty database pages.
- Updating index statistics to ensure the query optimizer has up-to-date information about the distribution of data values in the tables. This allows the query optimizer to make better judgments about the best way to access data because it has more information about the data stored in the database. Although index statistics are automatically updated by SQL Server periodically, this option can force the statistics to be updated immediately.
- Performing internal consistency checks of the data and data pages within the database to ensure that a system or software problem has not damaged data.
- Backing up the database and transaction log files. Database and log backups can be retained for a specified period. This allows you to create a history of backups to be used in the event that you need to restore the database to a time earlier than the last database backup.

• Setting up log shipping. Log shipping allows the transaction logs from one database (the source) to be constantly fed to another database (the destination). Keeping the destination database in synchronization with the source database allows you to have a standby server, and also provides a way to offload query processing from the main computer (source server) to read-only destination servers.

The results generated by the maintenance tasks can be written as a report to a text file, HTML file, or the **sysdbmaintplan_history** tables in the **msdb** database. The report can also be e-mailed to an operator.

To start the Database Maintenance Plan Wizard

Deleting a Database

You can delete a nonsystem database when it is no longer needed or if it is moved to another database or server. When a database is deleted, the files and their data are deleted from the disk on the server. When a database is deleted, it is permanently deleted and cannot be retrieved without using a previous backup. System databases (**msdb**, **master**, **model**, **tempdb**) cannot be deleted.

It is recommended that you back up the **master** database after a database is deleted, because deleting a database updates the system tables in **master**. If **master** needs to be restored, any database that has been deleted since the last backup of **master** will still have references in the system tables and may cause error messages to be raised.

To delete a database

∃ <u>Transact-SQL</u>

Tables

Tables are database objects that contain all the data in a database. A table definition is a collection of columns. In tables, data is organized in a row-and-column format similar to a spreadsheet. Each row represents a unique record, and each column represents a field within the record. For example, a table containing employee data for a company can contain a row for each employee and columns representing employee information such as employee number, name, address, job title, and home phone number.

Designing Tables

When you design a database, you decide what tables you need, what type of data goes in each table, who can access each table, and so on. As you create and work with tables, you continue to make more detailed decisions about them.

The most efficient way to create a table is to define everything you need in the table at one time, including its data restrictions and additional components. However, you can also create a basic table, add some data to it, and then work with it for a while. This approach gives you a chance to see what types of transactions are most common and what types of data are frequently entered before you commit to a firm design by adding constraints, indexes, defaults, rules, and other objects.

It is a good idea to outline your plans on paper before creating a table and its objects. Decisions that must be made include:

- Types of data the table will contain.
- Columns in the table and the data type (and length, if required) for each column.
- Which columns accept null values.
- Whether and where to use constraints or defaults and rules.
- Types of indexes needed, where required, and which columns are primary keys and which are foreign keys.

See Also

<u>Indexes</u>

Specifying a Column Data Type

Assigning a <u>data type</u> to each column is one of the first steps to take in designing a table. Data types define the data value allowed for each column. To assign a data type to a column, you can use Microsoft® SQL Server[™] 2000 base data types or create your own user-defined data types based on these system data types. For example, if you want to include only names in a column, you can assign a character data type to the column. Similarly, if you want a column to contain only numbers, you can assign a numeric data type. For more information about user-defined data types, see <u>Creating User-Defined Data Types</u>.

SQL Server also supports SQL-92 synonyms for several base data types. For more information, see <u>Data Type Synonyms</u>.

Enforcing Data Integrity

System and user-defined data types can be used to enforce data integrity, because the data entered or changed must conform to the type specified in the original CREATE TABLE statement. For example, you cannot store a surname in a column defined as **datetime** because a **datetime** column accepts only valid dates. For the most part, keep numeric data in numeric columns, especially if calculations must be performed on the numeric data at a later date.

Binary Data

Binary data consists of hexadecimal numbers. For example, the decimal number 245 is hexadecimal F5. Binary data is stored using the **binary**, **varbinary**, and **image** data types in Microsoft® SQL Server[™] 2000. A column assigned the **binary** data type must have the same fixed length (up to 8 KB) for each row. In a column assigned the **varbinary** data type, entries can vary in the number of hexadecimal digits (up to 8 KB) they contain. Columns of **image** data can be used to store variable-length binary data exceeding 8 KB, such as Microsoft Word documents, Microsoft Excel spreadsheets, and images that include bitmaps, Graphics Interchange Format (GIF), and Joint Photographic Experts Group (JPEG) files.

In general, use **varbinary** for storing binary data, unless the length of the data exceeds 8 KB, in which case you should use **image**. It is recommended that the defined length of a binary column be no larger than the expected maximum length of the binary data to be stored.

See Also

binary and varbinary image Using Binary Data Using Data Types

Using text and image Data

Character Data

Character data consists of any combination of letters, symbols, and numeric characters. For example, valid character data includes "928", "Johnson", and " (0*&(%B99nh jkJ". In Microsoft® SQL ServerTM 2000, character data is stored using the **char**, **varchar**, and **text** data types. Use **varchar** when the entries in a column vary in the number of characters they contain, but the length of any entry does not exceeds 8 kilobytes (KB). Use **char** when every entry for a column has the same fixed length (up to 8 KB). Columns of **text** data can be used to store ASCII characters longer than 8 KB. For example, because HTML documents are all ASCII characters and usually longer than 8 KB, they can be stored in **text** columns in SQL Server prior to being viewed in a browser.

It is recommended that the defined length of a character column be no larger than the maximum expected length of the character data to be stored.

To store international character data in SQL Server, use the **nchar**, **nvarchar**, and **ntext** data types.

See Also

<u>char and varchar</u> <u>ntext, text, and image</u> <u>Using char and varchar Data</u> <u>Using Data Types</u> <u>Using text and image Data</u> <u>Using Unicode Data</u>

Unicode Data

Traditional non-Unicode data types in Microsoft® SQL Server[™] 2000 allow the use of characters that are defined by a particular character set. A character set is chosen during SQL Server Setup and cannot be changed. Using Unicode data types, a column can store any character defined by the Unicode Standard, which includes all of the characters defined in the various character sets. Unicode data types take twice as much storage space as non-Unicode data types.

Unicode data is stored using the **nchar**, **nvarchar**, and **ntext** data types in SQL Server. Use these data types for columns that store characters from more than one character set. Use **nvarchar** when a column's entries vary in the number of Unicode characters (up to 4,000) they contain. Use **nchar** when every entry for a column has the same fixed length (up to 4,000 Unicode characters). Use **ntext** when any entry for a column is longer than 4,000 Unicode characters.

Note The SQL Server Unicode data types are based on the National Character data types in the SQL-92 standard. SQL-92 uses the prefix character **n** to identify these data types and values.

See Also

<u>Collations</u> <u>nchar and nvarchar</u> <u>ntext, text, and image</u> <u>Using Unicode Data</u>

Date and Time Data

Date and time data consists of valid date or time combinations. For example, valid date and time data includes both "4/01/98 12:15:00:00:00 PM" and "1:28:29:15:01 AM 8/17/98". Date and time data is stored using the **datetime** and **smalldatetime** data types in Microsoft® SQL Server[™] 2000. Use **datetime** to store dates in the range from January 1, 1753 through December 31, 9999 (requires 8 bytes of storage per value). Use **smalldatetime** to store dates in the range from January 1, 1900 through June 6, 2079 (requires 4 bytes of storage per value).

See Also

datetime and smalldatetime

Using Data Types

Using Date and Time Data

Numeric Data

Numeric data consists of numbers only. Numeric data includes positive and negative numbers, decimal and fractional numbers, and whole numbers (integers).

Integer Data

Integer data consists of negative or positive whole numbers, such as -15, 0, 5, and 2509. Integer data is stored using the **bigint**, **int**, **smallint**, and **tinyint** data types in Microsoft® SQL Server[™] 2000. The **bigint** data type can store a larger range of numbers than the **int** data type. The **int** data type can store a larger range of integers than **smallint**, which can store a larger range of numbers than **the int** data type.

Use the **bigint** data type to store numbers in the range from -2^63 (-9223372036854775808) through 2^63-1 (9223372036854775807). Storage size is 8 bytes.

Use the **int** data type to store numbers in the range from -2,147,483,648 through 2,147,483,647 only (requires 4 bytes of storage per value).

Use the **smallint** data type to store numbers in the range from -32,768 through 32,767 only (requires 2 bytes of storage per value), and the **tinyint** data type to store numbers in the range from 0 through 255 only (requires 1 byte of storage per value).

Decimal Data

Decimal data consists of data that is stored to the least significant digit. Decimal data is stored using **decimal** or **numeric** data types in SQL Server. The number of bytes required to store a **decimal** or **numeric** value depends on the total number of digits for the data and the number of decimal digits to the right of the decimal point. For example, more bytes are required to store the value 19283.29383 than to store the value 1.1.

In SQL Server, the **numeric** data type is equivalent to the **decimal** data type.

Approximate Numeric Data

Approximate numeric (floating-point) data consists of data preserved as accurately as the binary numbering system can offer. Approximate numeric data is stored using the **float** and **real** data types in SQL Server. For example, because the fraction one-third in decimal notation is .333333 (repeating), this value cannot be represented precisely using approximate decimal data. Therefore, the value retrieved from SQL Server may not be exactly what was stored originally in the column. Additional examples of numeric approximations are floating-point values ending in .3, .6, and .7.

See Also

decimal and numeric float and real int, bigint, smallint, and tinyint Using Data Types Using decimal, float, and real Data Using Integer Data

Monetary Data

Monetary data represents positive or negative amounts of money. In Microsoft® SQL Server[™] 2000, monetary data is stored using the **money** and **smallmoney** data types. Monetary data can be stored to an accuracy of four decimal places. Use the **money** data type to store values in the range from -922,337,203,685,477.5808 through +922,337,203,685,477.5807 (requires 8 bytes to store a value). Use the **smallmoney** data type to store values in the range from -214,748.3648 through 214,748.3647 (requires 4 bytes to store a value). If a greater number of decimal places are required, use the **decimal** data type instead.

See Also

money and smallmoney

Using Data Types

Using Monetary Data

Special Data

Special data consists of data that does not fit any of the categories of data such as binary data, character data, Unicode data, date and time data, numeric data and monetary data.

Microsoft[®] SQL Server[™] 2000 includes four types of special data:

• timestamp

Is used to indicate the sequence of SQL Server activity on a row, represented as an increasing number in a binary format. As a row is modified in a table, the timestamp is updated with the current database timestamp value obtained from the @@DBTS function. **timestamp** data is not related to the date and time of an insert or change to data. To automatically record times that data modifications take place in a table, use either a **datetime** or **smalldatetime** data type to record the events and triggers.

Note In SQL Server, **rowversion** is a synonym for **timestamp**.

• bit

Consists of either a 1 or a 0. Use the **bit** data type when representing TRUE or FALSE, or YES or NO. For example, a client questionnaire that asks if this is the client's first visit can be stored in a **bit** column.

• uniqueidentifier

Consists of a 16-byte hexadecimal number indicating a globally unique identifier (GUID). The GUID is useful when a row must be unique among many other rows. For example, use the **uniqueidentifier** data type for a customer identification number column to compile a master company customer list from multiple countries.

• sql_variant

A data type that stores values of various SQL Server–supported data types, except **text**, **ntext**, **timestamp**, **image**, and **sql_variant**.

• table

A special data type used to store a result set for later processing. The **table** data type can be used only to define local variables of type **table** or the return value of a user-defined function.

• user-defined

Allows a user-defined data type, **product_code**, for example, that is based on the **char** data type and defined as two uppercase letters followed by a five-digit supplier number.

See Also

<u>bit</u>

<u>timestamp</u>

uniqueidentifier

Using Data Types

Using Special Data

Using uniqueidentifier Data

Creating User-Defined Data Types

User-defined data types are based on the system data types in Microsoft® SQL Server[™] 2000. User-defined data types can be used when several tables must store the same type of data in a column and you must ensure that these columns have exactly the same data type, length, and nullability. For example, a user-defined data type called **postal_code** could be created based on the **char** data type.

When a user-defined data type is created, you must supply these parameters:

- Name
- System data type upon which the new data type is based
- Nullability (whether the data type allows null values)

When nullability is not explicitly defined, it will be assigned based on the ANSI null default setting for the database or connection.

Note If a user-defined data type is created in the **model** database, it exists in all new user-defined databases. However, if the data type is created in a user-defined database, the data type exists only in that user-defined database.

To create user-defined data types

Text in Row Data

Microsoft® SQL Server[™] 2000 supports the ability to store small to medium **text**, **ntext**, and **image** values in a data row. The feature is best used for tables in which the data in **text**, **ntext**, and **image** columns is usually read or written in one unit and most statements referencing the table use the **text**, **ntext**, and **image** data.

Unless the **text in row** option is specified, **text**, **ntext**, or **image** strings are large character or binary strings (up to 2 gigabytes) stored outside a data row. The data row contains only a 16-byte text pointer that points to the root node of a tree built of internal pointers that map the pages in which the string fragments are stored. For more information about the storage of **text**, **ntext**, or **image** strings, see <u>text</u>, <u>ntext</u>, and <u>image Data</u>.

You can set a **text in row** option for tables containing **text**, **ntext**, or **image** columns. You can also specify a **text in row** option limit, from 24 through 7,000 bytes. With this option set, **text**, **ntext**, or **image** strings are stored 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 that are otherwise stored in the root node of the pointer tree are stored in the row. The pointers are stored in the row if:

• The amount of space needed to store the pointers is shorter than the specified **text in row** option limit.

• There is enough space available in the data row to hold the pointers.

When pointers are moved from the root node to the row itself, SQL Server does not have to use a root node. This can eliminate a page access when reading or writing the string, which speeds processing.

When root nodes are used, they are stored as one of the string fragments in a **text**, **ntext**, or **image** page and can hold up to five internal pointers. SQL Server needs 72 bytes of space in the row to store five pointers for an in-row string. If there is not enough space in the row to hold the pointers when the **text in row** option is on, SQL Server may have to allocate an 8-K page to hold them. You should not set the text in row limit to less than 72 unless you are certain that all strings stored in the column are either short or over 3 MB.

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 is inserted that is longer than 500 bytes, 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 with a 900-byte string for each **text** column, and enough data for all of the other columns in the table, leaving 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.

Enabling and Disabling the text in row Option

You can enable the **text in row** option for a table by using **sp_tableoption**:

sp_tableoption N'MyTable', 'text in row', 'ON'

Optionally, you can specify a maximum limit, from 24 through 7,000 bytes, for

the length of a **text**, **ntext**, and **image** string that can be stored in a data row: sp_tableoption N'MyTable', 'text in row', '1000'

If you specify **on** instead of a specific limit, the limit defaults to 256 bytes. This default value allows you most of the performance benefits that can be gained from the **text in row** option. Although you generally should not set the value below 72, you also should not set the value too high, especially for tables in which most statements do not reference the **text**, **ntext**, and **image** columns, or in which there are multiple **text**, **ntext**, and **image** columns. If you set a large **text in row** limit, and many strings are stored in the row itself, you can significantly reduce the number of data rows that fit on each page. If most statements referencing the table do not access the **text**, **ntext**, or **image** columns, decreasing the rows in a page can increase the pages that must be read to process queries. Reducing the rows per page can increase the size of indexes and the pages that might need to be scanned if the optimizer finds no usable index. The **text in row** limit default value of 256 is large enough to ensure that small strings and the root text pointers can be stored in the rows, but not so large that it decreases the rows per page enough to affect performance.

You can also use **sp_tableoption** to turn the option off by specifying an option value of either **off** or **0**:

sp_tableoption N'MyTable', 'text in row', 'OFF'

Effects of the text in row Option

The **text in row** option has these effects:

- After you have turned on the **text in row** option, you cannot use the READTEXT, UPDATETEXT or WRITETEXT statements, to read or modify parts of any **text**, **ntext**, or **image** value stored in the table. In SELECT statements you can read an entire **text**, **ntext**, or **image** string, or use the SUBSTRING function to read parts of the string. All INSERT or UPDATE statements referencing the table must specify complete strings and cannot modify only a part of a **text**, **ntext**, or **image** string.
- When the **text in row** option is first enabled, existing **text**, **ntext**, or

image strings are not immediately converted to in-row strings. The strings are converted to in-row strings only if they are subsequently updated. Any **text**, **ntext**, or **image** string inserted after the **text in row** option is turned on is inserted as an in-row string.

- Turning off the **text in row** option can be a long-running, logged operation. The table is locked and all in-row **text**, **ntext**, and **image** strings are converted to regular **text**, **ntext**, and **image** strings. The length of time the command must run and the amount of data modified depends on how many **text**, **ntext**, and **image** strings must be converted from in-row strings to regular strings.
- The **text in row** option does not affect the operation of the OLE DB Provider for SQL Server or the SQL Server ODBC driver, other than to speed access to the **text**, **ntext**, and **image** data.
- The DB-Library text and image functions, such as **dbreadtext** and **dbwritetext**, cannot be used on a table after the **text in row** option has been turned on.

The **text in row** option is set to 256 automatically for:

- Variables with a table data type.
- Tables returned by user-defined functions that return a **table**.

This setting cannot be changed.

Autonumbering and Identifier Columns

For each table, a single identifier column can be created that contains systemgenerated sequential values that uniquely identify each row within the table. For example, an identifier column can generate unique customer receipt numbers for an application automatically as rows are inserted into the table. Identifier columns usually contain values unique within the table on which they are defined. This means that other tables containing identifier columns can contain the same identity values used by another table. However, this is usually not a problem because the identifier values are typically used only within the context of a single table, and the identifier columns do not relate to other identifier columns in other tables.

A single, globally unique, identifier column can be created per table that contains values unique across all networked computers in the world. A column guaranteed to contain globally unique values is often useful when similar data from multiple database systems must be merged (for example, in a customer billing system with data located in various company subsidiaries around the world). When the data is merged into the central site for consolidation and reporting, using globally unique values prevents customers in different countries from having the same billing number or customer ID.

Microsoft® SQL Server[™] 2000 uses globally unique identifier columns for merge replication to ensure that rows are uniquely identified across multiple copies of the table.

See Also

Creating and Modifying Identifier Columns Merge Replication NEWID uniqueidentifier Using Uniqueidentifier Data

Using Constraints, Defaults, and Null Values

Planning tables requires identifying valid values for a column and deciding how to enforce the integrity of the data in the column. Microsoft® SQL ServerTM 2000 provides several mechanisms to enforce the integrity of the data in a column:

- PRIMARY KEY constraints
- FOREIGN KEY constraints
- UNIQUE constraints
- CHECK constraints
- DEFAULT definitions
- Nullability

See Also

Data Integrity

PRIMARY KEY Constraints

A table usually has a column or combination of columns whose values uniquely identify each row in the table. This column (or columns) is called the <u>primary key</u> of the table and enforces the <u>entity integrity</u> of the table. You can create a primary key by defining a PRIMARY KEY constraint when you create or alter a table.

A table can have only one PRIMARY KEY constraint, and a column that participates in the PRIMARY KEY constraint cannot accept null values. Because PRIMARY KEY constraints ensure unique data, they are often defined for <u>identity column</u>.

When you specify a PRIMARY KEY constraint for a table, Microsoft® SQL Server[™] 2000 enforces data uniqueness by creating a <u>unique index</u> for the primary key columns. This index also permits fast access to data when the primary key is used in queries.

If a PRIMARY KEY constraint is defined on more than one column, values may be duplicated within one column, but each combination of values from all the columns in the PRIMARY KEY constraint definition must be unique.

As shown in the following illustration, the **au_id** and **title_id** columns in the **titleauthor** table form a composite PRIMARY KEY constraint for the **titleauthor** table, which ensures that the combination of **au_id** and **title_id** is unique.

When you work with joins, PRIMARY KEY constraints relate one table to another. For example, to determine which authors have written which books, you can use a three-way join between the **authors** table, the **titles** table, and the **titleauthor** table. Because **titleauthor** contains columns for both the **au_id** and **title_id** columns, the **titles** table can be accessed by the relationship between **titleauthor** and **titles**.

See Also

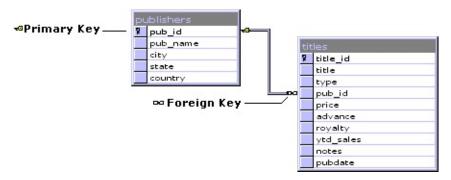
Creating and Modifying PRIMARY KEY Constraints

FOREIGN KEY Constraints

A foreign key (FK) is a column or combination of columns used to establish and enforce a link between the data in two tables. A link is created between two tables by adding the column or columns that hold one table's primary key values to the other table. This column becomes a foreign key in the second table.

You can create a foreign key by defining a FOREIGN KEY constraint when you create or alter a table.

For example, the **titles** table in the **pubs** database has a link to the **publishers** table because there is a logical relationship between books and publishers. The **pub_id** column in the **titles** table matches the primary key column of the **publishers** table. The **pub_id** column in the **titles** table is the foreign key to the **publishers** table.



A FOREIGN KEY constraint does not have to be linked only to a PRIMARY KEY constraint in another table; it can also be defined to reference the columns of a UNIQUE constraint in another table. A FOREIGN KEY constraint can contain null values; however, if any column of a composite FOREIGN KEY constraint contains null values, then verification of the FOREIGN KEY constraint will be skipped.

Note A FOREIGN KEY constraint can reference columns in tables in the same database or within the same table (self-referencing tables), for example, an employee table that contains three columns: **employee_number**, **employee_name**, and **manager_employee_number**. Because the manager is an employee too, there is a foreign key relationship from the **manager_employee_number** column to the **employee_number** column.

Although the primary purpose of a FOREIGN KEY constraint is to control the data that can be stored in the foreign key table, it also controls changes to data in the primary key table. For example, if the row for a publisher is deleted from the **publishers** table, and the publisher's ID is used for books in the **titles** table, the relational integrity between the two tables is broken; the deleted publisher's books are orphaned in the titles table without a link to the data in the publishers table. A FOREIGN KEY constraint prevents this situation. The constraint enforces referential integrity by ensuring that changes cannot be made to data in the primary key table if those changes invalidate the link to data in the foreign key table. If an attempt is made to delete the row in a primary key table or to change a primary key value, the action will fail if the deleted or changed primary key value corresponds to a value in the FOREIGN KEY constraint of another table. To change or delete a row in a FOREIGN KEY constraint successfully, you must first either delete the foreign key data in the foreign key table or change the foreign key data in the foreign key table, thereby linking the foreign key to different primary key data.

A FOREIGN KEY constraint is a candidate for an index because:

- Changes to PRIMARY KEY constraints are checked with FOREIGN KEY constraints in related tables.
- Foreign key columns are often used in join criteria when the data from related tables is combined in queries by matching the column(s) in the FOREIGN KEY constraint of one table with the primary or unique key column(s) in the other table. An index allows Microsoft® SQL Server[™] 2000 to find related data in the foreign key table quickly. However, creating this index is not a requirement. Data from two related tables can be combined even if no PRIMARY KEY or FOREIGN KEY constraints are defined between the tables, but a foreign key relationship between two tables indicates that the two tables have been optimized to be combined in a query that uses the keys as its criteria. For more information about using FOREIGN KEY constraints with joins, see Join Fundamentals.

See Also

Creating and Modifying FOREIGN KEY Constraints Indexes

Cascading Referential Integrity Constraints

Cascading referential integrity constraints allow you to define the actions Microsoft® SQL ServerTM 2000 takes when a user attempts to delete or update a key to which existing foreign keys point.

The REFERENCES clauses of the CREATE TABLE and ALTER TABLE statements support ON DELETE and ON UPDATE clauses:

- [ON DELETE { CASCADE | NO ACTION }]
- [ON UPDATE { CASCADE | NO ACTION }]

NO ACTION is the default if ON DELETE or ON UPDATE is not specified. NO ACTION specifies the same behavior that occurs in earlier versions of SQL Server.

ON DELETE NO ACTION

Specifies that if an attempt is made to delete a row with a key referenced by foreign keys in existing rows in other tables, an error is raised and the DELETE is rolled back.

ON UPDATE NO ACTION

Specifies that if an attempt is made to update a key value in a row whose key is referenced by foreign keys in existing rows in other tables, an error is raised and the UPDATE is rolled back.

CASCADE allows deletions or updates of key values to cascade through the tables defined to have foreign key relationships that can be traced back to the table on which the modification is performed. CASCADE cannot be specified for any foreign keys or primary keys that have a **timestamp** column.

ON DELETE CASCADE

Specifies that if an attempt is made to delete a row with a key referenced by foreign keys in existing rows in other tables, all rows containing those foreign keys are also deleted. If cascading referential actions have also been

defined on the target tables, the specified cascading actions are also taken for the rows deleted from those tables.

ON UPDATE CASCADE

Specifies that if an attempt is made to update a key value in a row, where the key value is referenced by foreign keys in existing rows in other tables, all of the foreign key values are also updated to the new value specified for the key. If cascading referential actions have also been defined on the target tables, the specified cascading actions are also taken for the key values updated in those tables.

Examples of cascading referential actions can be based on the **FK_Products_Suppliers** constraint on the **Products** table in **Northwind**. This constraint establishes a foreign key relationship from the **SupplierID** column in the **Products** table to the **SupplierID** primary key column in the **Suppliers** table. If ON DELETE CASCADE is specified for the constraint, deleting the row in **Suppliers** where **SupplierID** equals 1 also deletes the three rows in **Products** where **SupplierID** equals 1. If ON UPDATE CASCADE is specified for the constraint, updating the **SupplierID** value in the **Suppliers** table from 1 through 55 also updates the **SupplierID** values in the three rows in **Products** whose **SupplierID** values currently equal 1.

Cascading actions cannot be specified for a table that has an INSTEAD OF trigger. After a cascading action has been defined for a table, an INSTEAD OF trigger cannot be added to it.

Multiple Cascading Actions

Individual DELETE or UPDATE statements can start a series of cascading referential actions. For example, a database contains three tables, **TableA**, **TableB**, and **TableC**. A foreign key in **TableB** is defined with ON DELETE CASCADE against the primary key in **TableA**. A foreign key in **TableB**. If a DELETE statement deletes rows in **TableA**, the operation also deletes any rows in **TableB** that have foreign keys matching the deleted primary keys in **TableA**, and then deletes any rows in **TableB**.

The series of cascading referential actions triggered by a single DELETE or

UPDATE must form a tree containing no circular references. No table can appear more than once in the list of all cascading referential actions that result from the DELETE or UPDATE. The tree of cascading referential actions must not have more than one path to any given table. Any branch of the tree is terminated when it encounters a table for which NO ACTION has been specified or is the default.

Triggers and Cascading Referential Actions

Cascading referential actions fire AFTER triggers in this sequence:

- 1. All of the cascading referential actions directly caused by the original DELETE or UPDATE are performed first.
- 2. When the original cascading referential actions have completed, the AFTER triggers on the original table are fired, regardless of whether any rows were updated.
- 3. AFTER triggers on tables in the chain of cascaded referential actions are then fired, but only if one or more rows in the table have been updated or deleted.

If any errors are generated by any of the original set of cascading referential actions, an error is raised, no AFTER triggers are fired, and the DELETE or UPDATE is rolled back.

An AFTER trigger can execute a DELETE or UPDATE statement that starts another chain of cascading referential actions. Each secondary chain of referential actions is treated independently. These secondary chains of referential actions behave like the primary chain. All of the secondary referential actions are completed before any secondary triggers are fired. Within each independent unit, there is no defined order in which the cascading referential actions are executed and the affected triggers are fired.

A table that has an INSTEAD OF trigger cannot also have a REFERENCES clause that specifies a cascading action. An AFTER trigger on a table targeted by a cascading action, however, can execute an INSERT, UPDATE, or DELETE statement on another table or view that fires an INSTEAD OF trigger defined on that object.

Cascading Referential Constraints Catalog Information

The following catalog information is available about cascading referential constraints.

The Transact-SQL OBJECTPROPERTY function supports these new values for the *property* parameter.

Value	Object	Description
CnstIsDeleteCascade		FOREIGN KEY constraint defined with ON DELETE CASCADE
CnstIsUpdateCascade		FOREIGN KEY constraint defined with ON UPDATE CASCADE

The **REFERENTIAL_CONSTRAINTS** information schema view returns CASCADE in the **UDPATE_RULE** or **DELETE_RULE** column when either ON UPDATE CASCADE or ON DELETE CASCADE is specified. NO ACTION is returned when either ON UPDATE NO ACTION or ON DELETE NO ACTION is specified, or if ON UPDATE or ON DELETE is not specified at all.

The UPDATE_RULE and DELETE_RULE columns returned by **sp_fkeys** and **sp_foreignkeys** are set to 1 when CASCADE is specified, and return 0 when NO ACTION is specified or is the default.

When a foreign key is specified as the object of **sp_help**, the output result set contains these new columns.

Column name	Data type	Description	
delete_action	nvarchar(9)	Indicates whether the delete action is	
		CASCADE, NO ACTION, or N/A	
		(not applicable).	
update_action	nvarchar(9)	Indicates whether the update action is	
		CASCADE, NO ACTION, or N/A	
		(not applicable).	

UNIQUE Constraints

You can use UNIQUE constraints to ensure that no duplicate values are entered in specific columns that do not participate in a primary key. Although both a UNIQUE constraint and a PRIMARY KEY constraint enforce uniqueness, use a UNIQUE constraint instead of a PRIMARY KEY constraint when you want to enforce the uniqueness of:

• A column, or combination of columns, that is not the primary key.

Multiple UNIQUE constraints can be defined on a table, whereas only one PRIMARY KEY constraint can be defined on a table.

• A column that allows null values.

UNIQUE constraints can be defined on columns that allow null values, whereas PRIMARY KEY constraints can be defined only on columns that do not allow null values.

A UNIQUE constraint can also be referenced by a FOREIGN KEY constraint.

See Also

Creating and Modifying UNIQUE Constraints

CHECK Constraints

CHECK constraints enforce domain integrity by limiting the values that are accepted by a column. They are similar to FOREIGN KEY constraints in that they control the values that are placed in a column. The difference is in how they determine which values are valid: FOREIGN KEY constraints get the list of valid values from another table, and CHECK constraints determine the valid values from a logical expression that is not based on data in another column. For example, it is possible to limit the range of values for a **salary** column by creating a CHECK constraint that allows only data that ranges from \$15,000 through \$100,000. This prevents salaries from being entered beyond the normal salary range.

You can create a CHECK constraint with any logical (Boolean) expression that returns TRUE or FALSE based on the logical operators. For the previous example, the logical expression is:

salary >= 15000 AND salary <= 100000

It is possible to apply multiple CHECK constraints to a single column. These are evaluated in the order in which created. It is also possible to apply a single CHECK constraint to multiple columns by creating it at the table level. For example, a multiple-column CHECK constraint can be used to confirm that any row with a **country** column value of USA also has a two-character value in the **state** column. This allows multiple conditions to be checked in one place.

See Also

Creating and Modifying CHECK Constraints

DEFAULT Definitions

Each column in a record must contain a value, even if that value is NULL. There are situations when you need to load a row of data into a table but you do not know the value for a column, or the value does not yet exist. If the column allows null values, you can load the row with a null value. Because nullable columns may not be desirable, a better solution can be to define, where appropriate, a DEFAULT definition for the column. For example, it is common to specify zero as the default for numeric columns, or N/A as the default for string columns when no value is specified.

When you load a row into a table with a DEFAULT definition for a column, you implicitly instruct Microsoft® SQL ServerTM 2000 to load a default value in the column when you do not specify a value for the column.

Note You can also explicitly instruct SQL Server to insert the default value for the column using the DEFAULT VALUES clause of the INSERT STATEMENT.

If a column does not allow null values and does not have a DEFAULT definition, you must specify a value for the column explicitly or SQL Server will return an error indicating that the column does not allow null values.

The value inserted into a column defined by the combination of the DEFAULT definition, the nullability of the column, and the value inserted into the column can be summarized.

Column definition	No entry, no DEFAULT definition	_	Enter a null value
Allows null values			NULL
Disallows null values	Error	Default value	Error

DEFAULT Objects

A DEFAULT object is defined for a specific database and is shared by columns of different tables by being bound to each column to which the default applies. For example, if several of your tables have a **quantity** column, you can define a DEFAULT object in your database that inserts a value of 1 in the **quantity** column when the user leaves that column blank in any table.

If a DEFAULT object is bound to a column, you can specify a different default value for that column in a specific table. This unbinds the existing DEFAULT object from the column before the new default value is bound to the column.

See Also

Allowing Null Values

Creating and Modifying DEFAULT Definitions

Allowing Null Values

The nullability of a column determines if the rows in the table can contain a null value for that column. A null value, or NULL, is not the same as zero (0), blank, or a zero-length character string such as ""; NULL means that no entry has been made. The presence NULL usually implies that the value is either unknown or undefined. For example, a null value in the **price** column of the **titles** table of the **pubs** database does not mean that the book has no price; NULL means that the price is unknown or has not been set. In general, avoid permitting null values because they incur more complexity in queries and updates and because there are other column options, such as PRIMARY KEY constraints, that cannot be used with nullable columns.

If a row is inserted but no value is included for a column that allows null values, Microsoft® SQL Server[™] 2000 supplies the value NULL (unless a DEFAULT definition or object exists). A column defined with the keyword NULL also accepts an explicit entry of NULL from the user, no matter what data type it is or if it has a default associated with it. The value NULL should not be placed within quotation marks because it will be interpreted as the character string 'NULL', rather than the null value.

Specifying a column as not permitting null values can help maintain data integrity by ensuring that a column in a row always contains data. If null values are not allowed, the user entering data in the table must enter a value in the column or the table row cannot be accepted into the database.

Note Columns defined with a PRIMARY KEY constraint or IDENTITY property cannot allow null values.

See Also

Null Values
Column Properties

Creating and Modifying a Table

After you have designed the database , the tables that will store the data in the database can be created. The data is usually stored in permanent tables. Tables are stored in the database files until they are deleted and are available to any user who has the appropriate permissions.

Temporary Tables

You can also create temporary tables. Temporary tables are similar to permanent tables, except temporary tables are stored in **tempdb** and are deleted automatically when no longer in use.

The two types of temporary tables, local and global, differ from each other in their names, their visibility, and their availability. Local temporary tables have a single number sign (#) as the first character of their names; they are visible only to the current connection for the user; and they are deleted when the user disconnects from instances of Microsoft® SQL Server[™] 2000. Global temporary tables have two number signs (##) as the first characters of their names; they are visible to any user after they are created; and they are deleted when all users referencing the table disconnect from SQL Server.

For example, if you create a table named **employees**, the table can be used by any person who has the security permissions in the database to use it, until the table is deleted. If you create a local temporary table named **#employees**, you are the only person who can work with the table, and it is deleted when you disconnect. If you create a global temporary table named **##employees**, any user in the database can work with this table. If no other user works with this table after you create it, the table is deleted when you disconnect. If another user works with the table after you create it, SQL Server deletes it when both of you disconnect.

Table Properties

You can define up to 1,024 columns per table. Table and column names must follow the rules for identifiers; they must be unique within a given table, but you can use the same column name in different tables in the same database. You must

also define a data type for each column.

Although table names must be unique for each owner within a database, you can create multiple tables with the same name if you specify different owners for each. You can create two tables named **employees** and designate **Jonah** as the owner of one and **Sally** as the owner of the other. When you need to work with one of the **employees** tables, you can distinguish between the two tables by specifying the owner with the name of the table.

To create a table

- ⊞ <u>Transact-SQL</u>
- ⊞<u>Enterprise Manager</u>
- ∃ <u>SQL-DMO</u>

Modifying Tables

After a table is created, you can change many of the options that were defined for the table when it was originally created, including:

- Columns can be added, modified, or deleted. For example, the column name, length, data type, precision, scale, and nullability can all be changed, although some restrictions exist. For more information, see <u>Modifying Column Properties</u>.
- PRIMARY KEY and FOREIGN KEY constraints can be added or deleted.
- UNIQUE and CHECK constraints and DEFAULT definitions (and objects) can be added or deleted.
- An identifier column can be added or deleted using the IDENTITY or ROWGUIDCOL property. The ROWGUIDCOL property can also be added to or removed from an existing column, although only one column in a table can have the ROWGUIDCOL property at any one time.

• A table and selected columns within the table can be registered for full-text indexing.

For more information about the modifications that can be made to a table, see <u>ALTER TABLE</u>.

The name or owner of a table can also be changed. When you do this, you must also change the name of the table in any triggers, stored procedures, Transact-SQL scripts, or other programming code that uses the old name or owner of the table.

To rename a table

⊡ <u>Transact-SQL</u>

<u>■ Enterprise Manager</u>

∃ <u>SQL-DMO</u>

To change the owner of a table

⊞ <u>Transact-SQL</u>

∃ <u>SQL-DMO</u>

See Also

Specifying a Column Data Type

Using Identifiers

Placing Tables on Filegroups

Modifying Column Properties

Each column in a table has a set of properties, such as name, data type, nullability, and data length. The entire set of properties for a column makes up the definition of the column in a table.

The column properties can be specified directly in a database table by using a database diagram. Three column properties are required before you can create a table in the database:

- Column name
- Data type
- Data length

The properties of a column can be changed, for example, by renaming it, altering its length, specifying a default value, and so on.

Column Data Type

The data type of an existing column can be changed provided that the existing data in the column can be implicitly converted to the new data type. For more information, see <u>ALTER TABLE</u>.

Column Data Length

When you select a data type, length is defined automatically. You can increase or decrease the length property only for a column with a data type of **binary**, **char**, **nchar**, **varbinary**, **varchar**, or **nvarchar**. For columns with other data types, the length is derived from the data type and cannot be changed. If the new specified length is smaller than the original column length, all values in the column that exceed the new length are truncated without any warning. It is not possible to change the length of a column defined with a PRIMARY KEY or FOREIGN KEY constraint.

Note Changing the column data length re-creates the table in the database when

you save the table or database diagram using SQL Server Enterprise Manager.

Column Precision

The precision of a numeric column is the maximum number of digits used by the selected data type. The precision of a nonnumeric column generally refers to either the maximum length or the defined length of the column.

For all data types except **decimal** and **numeric**, precision is defined automatically. You can change the column precision for the **decimal** and **numeric** data types if you want to redefine the maximum number of digits these columns use. SQL Server Enterprise Manager prevents you from changing the precision of a column that does not have one of these assigned data types.

Note Changing the column precision re-creates the table in the database when you save the table or database diagram using SQL Server Enterprise Manager.

Column Scale

The scale of a **numeric** or **decimal** column is to the maximum number of digits to the right of the decimal point. When you select a data type, the column scale by default is set to 0. For columns with approximate floating point numbers, the scale is undefined because the number of digits to the right of the decimal point is not fixed. You can change the scale for a **numeric** or **decimal** column if you want to redefine the number of digits that can appear to the right of the decimal point.

Note Changing the column scale re-creates the table in the database when you save the table or diagram using SQL Server Enterprise Manager.

Column Nullability

A column can be defined to either allow or disallow null values. By default, a column permits null values. An existing column can be changed to disallow null values only if no existing null values exist in the column and there is no existing index created on the column. To disallow null values in an existing column that contains null values:

1. Add a new column with a DEFAULT definition that inserts a valid value in place of NULL.

- 2. Copy the data in the old (existing) column to the new column.
- 3. Delete the old column.

An existing column that does not allow null values can be changed to allow null values unless a PRIMARY KEY constraint is defined on the column.

Note Changing the nullability on a new, nonkey column re-creates the table in the database when you save the table or database diagram using Database Diagrams within SQL Server Enterprise Manager.

To set column properties

⊞ <u>Transact-SQL</u>

<u>■ Enterprise Manager</u>

∃ <u>SQL-DMO</u>

To view column properties

∃ <u>Transact-SQL</u>

<u>Enterprise Manager</u>

∃ <u>SQL-DMO</u>

To rename a column

∃ <u>Transact-SQL</u>

<u>■ Enterprise Manager</u>

∃ <u>SQL-DMO</u>

See Also

Working with Tables
Database Objects
Precision, Scale, and Length

Adding and Deleting Columns

Microsoft® SQL Server[™] 2000 allows columns to be added to existing tables, provided that the column allows null values or a DEFAULT constraint is created on the column. When you add a new column to a table, SQL Server inserts a value in that column for each existing row of data in the table. For this reason, it is useful to add a DEFAULT definition to the column when you add it to the table. If the new column does not have a DEFAULT definition, you must specify that the new column allows null values. SQL Server inserts null values into the column or returns an error if the new column does not allow null values.

Conversely, columns can be deleted from existing tables. However, it is not possible to delete a column that is:

- Involved in replication.
- Used in an index.
- Used in a CHECK, FOREIGN KEY, UNIQUE, or PRIMARY KEY constraint.
- Associated with a DEFAULT definition, or bound to a default object.
- Bound to a rule.
- Registered for full-text support.
- Used as a full-text key for a table.

To add or delete a column

<u>
 Transact-SQL</u>

<u>■ Enterprise Manager</u>

∃ <u>SQL-DMO</u>

To copy columns from one table to another

⊞ <u>Enterprise Manager</u>

Creating and Modifying PRIMARY KEY Constraints

A single PRIMARY KEY constraint can be:

- Created when the table is created, as part of the table definition.
- Added to an existing table, provided that no other PRIMARY KEY constraint already exists (a table can have only one PRIMARY KEY constraint).
- Modified or deleted, if it already exists. For example, you may want the PRIMARY KEY constraint of the table to reference other columns, or you may want to change the column order, index name, clustered option, or fill factor of the PRIMARY KEY constraint. It is not possible to change the length of a column defined with a PRIMARY KEY constraint.

Note To modify a PRIMARY KEY constraint using Transact-SQL or SQL-DMO, you must first delete the existing PRIMARY KEY constraint and then re-create it with the new definition.

When a PRIMARY KEY constraint is added to an existing column or columns in the table, Microsoft® SQL ServerTM 2000 checks the existing data in the columns to ensure that the existing data follows the rules for primary keys:

- No null values
- No duplicate values

If a PRIMARY KEY constraint is added to a column that has duplicate or null values, SQL Server returns an error and does not add the constraint. It is not possible to add a PRIMARY KEY constraint that violates these rules.

SQL Server automatically creates a unique index to enforce the uniqueness requirement of the PRIMARY KEY constraint. If a clustered index does not already exist on the table, or a nonclustered index is not explicitly specified, a unique, clustered index is created to enforce the PRIMARY KEY constraint.

IMPORTANT A PRIMARY KEY constraint cannot be deleted if referenced by a FOREIGN KEY constraint in another table; the FOREIGN KEY constraint must be deleted first.

To create a PRIMARY KEY constraint when creating a table

∃ <u>Transact-SQL</u>

∃ <u>SQL-DMO</u>

To create or delete a PRIMARY KEY constraint on an existing table

 \boxplus <u>Transact-SQL</u>

∃ <u>SQL-DMO</u>

To modify a PRIMARY KEY constraint

<u>■ Enterprise Manager</u>

See Also

Primary Key Constraints

Creating and Modifying FOREIGN KEY Constraints

FOREIGN KEY constraints can be:

- Created when the table is created, as part of the table definition.
- Added to an existing table provided that the FOREIGN KEY constraint is linked to an existing PRIMARY KEY constraint or UNIQUE constraint in another (or the same) table. A table can contain multiple FOREIGN KEY constraints.
- Modified or deleted if FOREIGN KEY constraints already exist. For example, you may want the table's FOREIGN KEY constraint to reference other columns. It is not possible to change the length of a column defined with a FOREIGN KEY constraint.

Note To modify a FOREIGN KEY constraint using Transact-SQL or SQL-DMO, you must first delete the existing FOREIGN KEY constraint and then re-create it with the new definition.

When a FOREIGN KEY constraint is added to an existing column or columns in the table, Microsoft® SQL Server[™] 2000 by default checks the existing data in the columns to ensure that all values, except NULL, exist in the column(s) of the referenced PRIMARY KEY or UNIQUE constraint. However, SQL Server can be prevented from checking the data in the column against the new constraint and made to add the new constraint regardless of the data in the column. This option is useful when the existing data already meets the new FOREIGN KEY constraint, or when a business rule requires the constraint to be enforced only from this point forward.

However, you should be careful when adding a constraint without checking existing data because this bypasses the controls in SQL Server that enforce the data integrity of the table.

Disabling FOREIGN KEY Constraints

Existing FOREIGN KEY constraints can be disabled for:

• INSERT and UPDATE statements

This allows data in the table to be modified without being validated by the constraints. Disable a FOREIGN KEY constraint during INSERT and UPDATE statements if new data will violate the constraint or if the constraint should apply only to the data already in the database.

• Replication processing.

Disable a FOREIGN KEY constraint during replication if the constraint is specific to the source database. When a table is replicated, the table definition and data are copied from the source database to a destination database. These two databases are usually, but not necessarily, on separate servers. If the FOREIGN KEY constraints are specific to the source database but are not disabled during replication, they may unnecessarily prevent new data from being entered in the destination database.

Delete a FOREIGN KEY constraint, thus removing the requirement, to enforce referential integrity between the foreign key columns and the related primary key (or UNIQUE constraint) columns in another table.

To create a FOREIGN KEY constraint when creating a table

⊞ <u>Transact-SQL</u>

∃ <u>SQL-DMO</u>

To create a FOREIGN KEY constraint on an existing table

⊞ <u>Transact-SQL</u>

∃ <u>SQL-DMO</u>

To prevent checking of existing data when creating a FOREIGN KEY constraint

⊞ <u>Transact-SQL</u>

⊞ Enterprise Manager

∃ <u>SQL-DMO</u>

To modify a FOREIGN KEY constraint

<u>■ Enterprise Manager</u>

To disable a FOREIGN KEY constraint for INSERT and UPDATE statements

∃ Transact-SQL

<u>Enterprise Manager</u>

∃ <u>SQL-DMO</u>

To disable a FOREIGN KEY constraint for replication

∃ <u>Transact-SQL</u>

<u>■ Enterprise Manager</u>

∃ <u>SQL-DMO</u>

To delete a FOREIGN KEY constraint

∃ <u>Transact-SQL</u>

<u>Enterprise Manager</u>

∃ <u>SQL-DMO</u>

See Also

Foreign Key Constraints

Creating and Modifying UNIQUE Constraints

UNIQUE constraints can be:

- Created when the table is created, as part of the table definition.
- Added to an existing table, provided that the column or combination of columns comprising the UNIQUE constraint contains only unique or NULL values. A table can contain multiple UNIQUE constraints.
- Modified or deleted if they already exist. For example, you may want the UNIQUE constraint of the table to reference other columns, or you may want to change the type of index clustering.

Note To modify a UNIQUE constraint using Transact-SQL or SQL-DMO, you must first delete the existing UNIQUE constraint and then re-create it with the new definition.

When a UNIQUE constraint is added to an existing column or columns in the table, Microsoft® SQL Server[™] 2000 by default checks the existing data in the columns to ensure all values, except NULL, are unique. If a UNIQUE constraint is added to a column that has duplicated values, SQL Server returns an error and does not add the constraint.

SQL Server automatically creates a UNIQUE index to enforce the uniqueness requirement of the UNIQUE constraint. Therefore, if an attempt to insert a duplicate row is made, SQL Server returns an error message that says the UNIQUE constraint has been violated and does not add the row to the table. Unless a clustered index is explicitly specified, a unique, nonclustered index is created by default to enforce the UNIQUE constraint.

Delete a UNIQUE constraint to remove the uniqueness requirement for values entered in the column or combination of columns included in the constraint. It is not possible to delete a UNIQUE constraint if the associated column is used as the full-text key of the table.

To create a UNIQUE constraint when creating a table

∃ <u>Transact-SQL</u>

<u>■ Enterprise Manager</u>

To create a UNIQUE constraint on an existing table

⊞ <u>Transact-SQL</u>

<u>■ Enterprise Manager</u>

∃ <u>SQL-DMO</u>

To modify a UNIQUE constraint

<u>■ Enterprise Manager</u>

To delete a UNIQUE constraint

∃ <u>Transact-SQL</u>

∃ <u>SQL-DMO</u>

See Also

UNIQUE Constraints

Creating and Modifying CHECK Constraints

CHECK constraints can be:

- Created when the table is created, as part of the table definition.
- Added to an existing table. Tables and columns can contain multiple CHECK constraints.
- Modified or deleted if they already exist. For example, you can modify the expression used by the CHECK constraint on a column in the table.

Note To modify a CHECK constraint using Transact-SQL or SQL-DMO, you must first delete the existing CHECK constraint and then recreate it with the new definition.

When a CHECK constraint is added to an existing table, the CHECK constraint can apply either to new data only or to existing data as well. By default, the CHECK constraint applies to existing data as well as any new data. The option of applying the constraint to new data only is useful when the existing data already meets the new CHECK constraint, or when a business rule requires the constraint to be enforced only from this point forward.

For example, an old constraint may require that postal codes be limited to five digits but a new constraint requires nine-digit postal codes. Old data with five-digit postal codes is still valid and will co-exist with new data that contains nine-digit postal codes. Therefore, only new data should be checked against the new constraint.

However, you should be careful when adding a constraint without checking existing data because this bypasses the controls in Microsoft® SQL ServerTM 2000 that enforce the integrity rules for the table.

Disabling CHECK Constraints

Existing CHECK constraints can be disabled for:

- INSERT and UPDATE statements, thereby allowing data in the table to be modified without being validated by the constraints. Disable a CHECK constraint during INSERT and UPDATE statements if new data will violate the constraint or if the constraint should apply only to the data already in the database.
- Replication processing. Disable a CHECK constraint during replication if the constraint is specific to the source database. When a table is replicated, the table definition and data are copied from the source database to a destination database. These two databases are usually, but not necessarily, on separate servers. If the CHECK constraints specific to the source database are not disabled, they may unnecessarily prevent new data from being entered in the destination database.

Delete a CHECK constraint to remove the limitations on acceptable data values in the column or columns included in the constraint expression.

To create a CHECK constraint when creating a table

∃ <u>Transact-SQL</u>

∃ <u>SQL-DMO</u>

To create a CHECK constraint on an existing table

⊞ <u>Transact-SQL</u>

⊞ <u>Enterprise Manager</u>

∃ <u>SQL-DMO</u>

To prevent checking of existing data when creating a CHECK constraint

∃ <u>Transact-SQL</u>

∃ <u>SQL-DMO</u>

To modify a CHECK constraint

<u>
 Enterprise Manager</u>

To disable a CHECK constraint for INSERT and UPDATE statements

⊞ <u>Transact-SQL</u>

<u>■ Enterprise Manager</u>

∃ <u>SQL-DMO</u>

To disable a CHECK constraint for replication

∃ <u>Transact-SQL</u>

<u>■ Enterprise Manager</u>

∃ <u>SQL-DMO</u>

To delete a CHECK constraint

∃ <u>Transact-SQL</u>

<u>■ Enterprise Manager</u>

∃ <u>SQL-DMO</u>

See Also

CHECK Constraints

Creating and Modifying DEFAULT Definitions

DEFAULT definitions can be:

- Created when the table is created, as part of the table definition.
- Added to an existing table. Each column in a table can contain a single DEFAULT definition.
- Modified or deleted if DEFAULT definitions already exist. For example, you can modify the value inserted in a column when no value is entered.

Note To modify a DEFAULT definition using Transact-SQL or SQL-DMO, you must first delete the existing DEFAULT definition and then re-create it with the new definition.

DEFAULT definitions cannot be created on columns defined with:

- A **timestamp** data type.
- An IDENTITY or ROWGUIDCOL property.
- An existing DEFAULT definition or DEFAULT object.

Note The default value must be compatible with the data type of the column to which the DEFAULT definition applies. For example, the default value for an **int** column must be an integer number, not a character string.

When a DEFAULT definition is added to an existing column in a table, Microsoft® SQL Server[™] 2000 by default applies the new default only to new rows of data added to the table; existing data inserted using the previous DEFAULT definition is unaffected. However, when adding a new column to an existing table, you can specify that SQL Server insert the default value (specified by the DEFAULT definition) rather than a null value into the new column for the existing rows in the table.

When you delete a DEFAULT definition, SQL Server inserts a null value rather than the default value when no value is inserted into the column for new rows. However, no changes are made to the existing data in the table.

To create a DEFAULT definition on a column when creating a table

Transact-SQL
Enterprise Manager
SQL-DMO
To create or delete a DEFAULT definition on a column of an existing table
Transact-SQL
Enterprise Manager
SQL-DMO
To create a DEFAULT object

 \boxplus <u>Transact-SQL</u>

Creating and Modifying Identifier Columns

Only one identifier column and one globally unique identifier column can be created for each table.

IDENTITY Property

Identifier columns can be implemented using the IDENTITY property, which allows the application developer to specify both an identity number for the first row inserted into the table (**Identity Seed** property) and an increment (**Identity Increment** property) to be added to the seed to determine successive identity numbers. When inserting values into a table with an identifier column, Microsoft® SQL Server[™] 2000 automatically generates the next identity value by adding the increment to the seed.

When you use the IDENTITY property to define an identifier column, consider that:

- A table can have only one column defined with the IDENTITY property, and that column must be defined using the **decimal**, **int**, **numeric**, **smallint**, **bigint**, or **tinyint** data type.
- The seed and increment can be specified. The default value for both is 1.
- The identifier column must not allow null values and must not contain a DEFAULT definition or object.
- The column can be referenced in a select list by using the IDENTITYCOL keyword after the IDENTITY property has been set.
- The OBJECTPROPERTY function can be used to determine if a table has an IDENTITY column, and the COLUMNPROPERTY function can be used to determine the name of the IDENTITY column.

Globally Unique Identifiers

Although the IDENTITY property automates row numbering within one table, separate tables, each with its own identifier column, can generate the same values. This is because the IDENTITY property is guaranteed to be unique only for the table on which it is used. If an application must generate an identifier column that is unique across the entire database, or every database on every networked computer in the world, use the ROWGUIDCOL property, the **uniqueidentifier** data type, and the NEWID function.

When you use the ROWGUIDCOL property to define a globally unique identifier column, consider that:

- A table can have only one ROWGUIDCOL column, and that column must be defined using the **uniqueidentifier** data type.
- SQL Server does not automatically generate values for the column. To insert a globally unique value, create a DEFAULT definition on the column that uses the NEWID function to generate a globally unique value.
- The column can be referenced in a select list by using the ROWGUIDCOL keyword after the ROWGUIDCOL property is set. This is similar to the way an IDENTITY column can be referenced using the IDENTITYCOL keyword.
- The OBJECTPROPERTY function can be used to determine if a table has a ROWGUIDCOL column, and the COLUMNPROPERTY function can be used to determine the name of the ROWGUIDCOL column.
- Because the ROWGUIDCOL property does not enforce uniqueness, the UNIQUE constraint should be used to ensure that unique values are inserted into the ROWGUIDCOL column.

Note If an identifier column exists for a table with frequent deletions, gaps can occur between identity values; deleted identity values are not reused. To avoid

such gaps, do not use the IDENTITY property. Instead, you can create a trigger that determines a new identifier value, based on existing values in the identifier column, as rows are inserted.

To create a new identifier column when creating a table

⊞ <u>Transact-SQL</u>

<u>■ Enterprise Manager</u>

∃ <u>SQL-DMO</u>

To create a new identifier column on an existing table

⊞ <u>Transact-SQL</u>

<u>■ Enterprise Manager</u>

∃ <u>SQL-DMO</u>

To delete an identifier column

⊞ <u>Transact-SQL</u>

∃ <u>SQL-DMO</u>

See Also

Autonumbering and Identifier Columns

COLUMNPROPERTY

<u>NEWID</u>

OBJECTPROPERTY

uniqueidentifier

Using Uniqueidentifier Data

Viewing a Table

After you have created the tables in a database, you may need to find information about the table properties (for example, the name or data type of a column, the nature of its indexes, and so on). Additionally, and most importantly, you will need to view the data in the table.

You can also display the dependencies of the table to determine which objects, such as views, stored procedures, and triggers, depend on the table. If you make any changes to the table, dependent objects may be affected.

To view the definition of a table

Transact-SQL
Enterprise Manager
SQL-DMO
To view the data in a table
Transact-SQL
Enterprise Manager
To view the dependencies of a table
Transact-SQL

Deleting a Table

At times you need to delete a table (for example, when you want to implement a new design or free up space in the database). When you delete a table, its structural definition, data, full-text indexes, constraints, and indexes are permanently deleted from the database, and the space formerly used to store the table and its indexes is made available for other tables. You can explicitly drop a temporary table if you do not want to wait until it is dropped automatically.

If you need to delete tables that are related through FOREIGN KEY and UNIQUE or PRIMARY KEY constraints, you must delete the tables with the FOREIGN KEY constraints first. If you need to delete a table that is referenced in a FOREIGN KEY constraint but you cannot delete the entire foreign key table, you must delete the FOREIGN KEY constraint.

To delete a table

∃ <u>Transact-SQL</u>

<u>■ Enterprise Manager</u>

∃ <u>SQL-DMO</u>

To delete a FOREIGN KEY constraint

<u>■ Enterprise Manager</u>

∃ <u>SQL-DMO</u>

Indexes

Indexes in databases are similar to indexes in books. In a book, an index allows you to find information quickly without reading the entire book. In a database, an index allows the database program to find data in a table without scanning the entire table. An index in a book is a list of words with the page numbers that contain each word. An index in a database is a list of values in a table with the storage locations of rows in the table that contain each value. Indexes can be created on either a single column or a combination of columns in a table and are implemented in the form of B-trees. An index contains an entry with one or more columns (the search key) from each row in a table. A B-tree is sorted on the search key, and can be searched efficiently on any leading subset of the search key. For example, an index on columns **A**, **B**, **C** can be searched efficiently on **A**, on **A**, **B**, and **A**, **B**, **C**.

Most books contain one general index of words, names, places, and so on. Databases contain individual indexes for selected types or columns of data: this is similar to a book that contains one index for names of people and another index for places. When you create a database and tune it for performance, you should create indexes for the columns used in queries to find data.

In the **pubs** sample database provided with Microsoft® SQL Server[™] 2000, the **employee** table has an index on the **emp_id** column. The following illustration shows how the index stores each **emp_id** value and points to the rows of data in the table with each value.

When SQL Server executes a statement to find data in the **employee** table based on a specified **emp_id** value, it recognizes the index for the **emp_id** column and uses the index to find the data. If the index is not present, it performs a full table scan starting at the beginning of the table and stepping through each row, searching for the specified **emp_id** value.

SQL Server automatically creates indexes for certain types of constraints (for example, PRIMARY KEY and UNIQUE constraints). You can further customize the table definitions by creating indexes that are independent of constraints.

The performance benefits of indexes, however, do come with a cost. Tables with

indexes require more storage space in the database. Also, commands that insert, update, or delete data can take longer and require more processing time to maintain the indexes. When you design and create indexes, you should ensure that the performance benefits outweigh the extra cost in storage space and processing resources.

See Also

Full-Text IndexesIndex Tuning WizardIndex Tuning RecommendationsTable and Index Architecture

Designing an Index

When Microsoft® SQL Server[™] 2000 executes a query, the query optimizer evaluates the costs of the available methods for retrieving the data and uses the most efficient method. SQL Server can perform a table scan, or it can use an index if one exists. When performing a table scan, SQL Server starts at the beginning of the table, steps row-by-row through all the rows in the table, and extracts the rows that meet the criteria of the query. When SQL Server uses an index, it finds the storage location of the rows needed by the query and extracts only the needed rows.

When you are considering whether to create an index on a column, consider if and how an indexed column is to be used in queries. Indexes are useful when a query:

- Searches for rows that match a specific search key value (an exact match query). An exact match comparison is one in which the query uses the WHERE statement to specify a column entry with a given value. For example:
 WHERE emp_id = 'VPA30890F'
- Searches for rows with search key values in a range of values (a range query). A range query is one in which the query specifies any entry whose value is between two values. For example:
 WHERE job_lvl BETWEEN 9 and 12

Or

WHERE job_lvl >= 9 and job_lvl <= 12

- Searches for rows in a table **T1** that match, based on a join predicate, a row in another table **T2** (an index nested loops join).
- Produces sorted query output without an explicit sort operation, in particular for sorted dynamic cursors.

- Scans rows in a sorted order to permit an order-based operation, such as merge join and stream aggregation, without an explicit sort operation.
- Scans all rows in a table with better performance than a table scan, due to the reduced column set and overall data volume to be scanned (a covering index for the query at hand).
- Searches for duplicates of new search key values in insert and update operations, to enforce PRIMARY KEY and UNIQUE constraints.
- Searches for matching rows between two tables for which a FOREIGN KEY constraint is defined.

Queries using LIKE comparisons can benefit from an index if the pattern starts with a specific character string, for example 'abc%'; but not if the pattern starts with a wildcard search, for example '%xyz'.

In many queries, the benefits of indexes can be combined. For example, an index enables a range query in addition to covering the query. SQL Server can use multiple indexes for a single table in the same query, as well as combining multiple indexes (using a join algorithm) so that the search keys together cover a query. Additionally, SQL Server automatically determines which indexes to exploit for a query and ensures that all indexes for a table are maintained when the table is modified.

Additional Guidelines for Designing Indexes

Additional guidelines to consider when designing indexes include:

• Large numbers of indexes on a table affect the performance of INSERT, UPDATE, and DELETE statements because all indexes must be adjusted appropriately as data in the table changes. Conversely, large numbers of indexes can help the performance of queries that do not modify data (SELECT statements) because SQL Server has more indexes to choose from to determine the best way to access the data as fast as possible.

- Covered queries can improve performance. Covered queries are queries where all the columns specified in the query are contained within the same index. For example, a query retrieving columns a and b from a table that has a <u>composite index</u> created on columns a, b, and c is considered covered. Creating indexes that cover a query can improve performance because all the data for the query is contained within the index itself; only the index pages, not the data pages, of the table must be referenced to retrieve the data, thereby reducing overall I/O. Although adding columns to an index to cover queries can improve performance, maintaining the extra columns in the index incurs update and storage costs.
- Indexing small tables may not be optimal because it can take SQL Server longer to traverse the index searching for data than to perform a simple table scan.
- SQL Profiler and the Index Tuning Wizard should be used to help analyze queries and determine which indexes to create. The selection of the right indexes for a database and its workload is a very complex balancing act between query speed and update cost. Narrow indexes (indexes with few columns in the search key) require less disk space and maintenance overhead. Wide indexes, on the other hand, cover more queries. There are no simple rules for determining the right set of indexes. Experienced database administrators can often design a good set of indexes, but this task is very complex, time-consuming, and errorprone even for moderately complex databases and workloads. The Index Tuning Wizard can be used to automate this task. For more information, see <u>Index Tuning Wizard</u>.
- You can specify indexes on views. For more information, see <u>Designing</u> <u>an Indexed View</u>.
- You can specify indexes on computed columns. For more information, see <u>Creating Indexes on Computed Columns</u>.

Index Characteristics

After you have determined that an index is justified for a query, you can customize the type of index that best fits your situation. Characteristics of indexes include:

- Clustered versus nonclustered
- Unique versus nonunique
- Single-column versus multicolumn
- Ascending or descending order on the columns in the index
- Covering or noncovering

You can also customize the initial storage characteristics of the index to optimize its maintenance by setting a fill factor, and customize its location using files and filegroups to optimize performance.

See Also

Designing Tables <u>Fill Factor</u> <u>Placing Indexes on Filegroups</u> <u>Query Tuning</u> <u>Understanding Merge Joins</u> <u>Understanding Nested Loops Joins</u>

Using Clustered Indexes

A clustered index determines the physical order of data in a table. A clustered index is analogous to a telephone directory, which arranges data by last name. Because the clustered index dictates the physical storage order of the data in the table, a table can contain only one clustered index. However, the index can comprise multiple columns (a composite index), like the way a telephone directory is organized by last name and first name.

A clustered index is particularly efficient on columns that are often searched for ranges of values. After the row with the first value is found using the clustered index, rows with subsequent indexed values are guaranteed to be physically adjacent. For example, if an application frequently executes a query to retrieve records between a range of dates, a clustered index can quickly locate the row containing the beginning date, and then retrieve all adjacent rows in the table until the last date is reached. This can help increase the performance of this type of query. Also, if there is a column(s) that is used frequently to sort the data retrieved from a table, it can be advantageous to cluster (physically sort) the table on that column(s) to save the cost of a sort each time the column(s) is queried.

Clustered indexes are also efficient for finding a specific row when the indexed value is unique. For example, the fastest way to find a particular employee using the unique employee ID column **emp_id** is to create a clustered index or PRIMARY KEY constraint on the **emp_id** column.

Note PRIMARY KEY constraints create clustered indexes automatically if no clustered index already exists on the table and a nonclustered index is not specified when you create the PRIMARY KEY constraint.

Alternatively, a clustered index could be created on **lname**, **fname** (last name, first name), because employee records are often grouped and queried in this way rather than by employee ID.

Considerations

It is important to define the clustered index key with as few columns as possible. If a large clustered index key is defined, any nonclustered indexes that are defined on the same table will be significantly larger because the nonclustered index entries contain the clustering key. The Index Tuning Wizard does not return an error when saving an SQL script to a disk with insufficient available space. For more information about how nonclustered indexes are implemented in Microsoft® SQL ServerTM 2000, see <u>Nonclustered Indexes</u>.

The Index Tuning Wizard can consume significant CPU and memory resources during analysis. It is recommended that tuning should be performed against a test version of the production server rather than the production server. Additionally, the wizard should be run on a separate computer from the computer running SQL Server. The wizard cannot be used to select or create indexes and statistics in databases on SQL Server version 6.5 or earlier.

Before creating clustered indexes, understand how your data will be accessed. Consider using a clustered index for:

- Columns that contain a large number of distinct values.
- Queries that return a range of values using operators such as BETWEEN, >, >=, <, and <=.
- Columns that are accessed sequentially.
- Queries that return large result sets.
- Columns that are frequently accessed by queries involving join or GROUP BY clauses; typically these are foreign key columns. An index on the column(s) specified in the ORDER BY or GROUP BY clause eliminates the need for SQL Server to sort the data because the rows are already sorted. This improves query performance.
- OLTP-type applications where very fast single row lookup is required, typically by means of the primary key. Create a clustered index on the primary key.

Clustered indexes are not a good choice for:

• Columns that undergo frequent changes

This results in the entire row moving (because SQL Server must keep the data values of a row in physical order). This is an important consideration in high-volume transaction processing systems where data tends to be volatile.

• Wide keys

The key values from the clustered index are used by all nonclustered indexes as lookup keys and therefore are stored in each nonclustered index leaf entry.

See Also

Clustered Indexes

Creating an Index

Creating and Modifying PRIMARY KEY Constraints

Using Nonclustered Indexes

A nonclustered index is analogous to an index in a textbook. The data is stored in one place, the index in another, with pointers to the storage location of the data. The items in the index are stored in the order of the index key values, but the information in the table is stored in a different order (which can be dictated by a clustered index). If no clustered index is created on the table, the rows are not guaranteed to be in any particular order.

Similar to the way you use an index in a book, Microsoft® SQL Server[™] 2000 searches for a data value by searching the nonclustered index to find the location of the data value in the table and then retrieves the data directly from that location. This makes nonclustered indexes the optimal choice for exact match queries because the index contains entries describing the exact location in the table of the data values being searched for in the queries. If the underlying table is sorted using a clustered index, the location is the clustering key value; otherwise, the location is the row ID (RID) comprised of the file number, page number, and slot number of the row. For example, to search for an employee ID (**emp_id**) in a table that has a nonclustered index on the **emp_id** column, SQL Server looks through the index to find an entry that lists the exact page and row in the table where the matching **emp_id** can be found, and then goes directly to that page and row.

Multiple Nonclustered Indexes

Some books contain multiple indexes. For example, a gardening book can contain one index for the common names of plants and another index for the scientific names because these are the two most common ways in which the readers find information. The same is true for nonclustered indexes. You can define a nonclustered index for each of the columns commonly used to find the data in the table.

Considerations

Before you create nonclustered indexes, understand how your data will be accessed. Consider using nonclustered indexes for:

- Columns that contain a large number of distinct values, such as a combination of last name and first name (if a clustered index is used for other columns). If there are very few distinct values, such as only 1 and 0, most queries will not use the index because a table scan is usually more efficient.
- Queries that do not return large result sets.
- Columns frequently involved in search conditions of a query (WHERE clause) that return exact matches.
- Decision-support-system applications for which joins and grouping are frequently required. Create multiple nonclustered indexes on columns involved in join and grouping operations, and a clustered index on any foreign key columns.
- Covering all columns from one table in a given query. This eliminates accessing the table or clustered index altogether.

See Also

Creating an Index

Index Tuning Wizard

Nonclustered Indexes

Using Unique Indexes

A unique index ensures that the indexed column contains no duplicate values. In the case of multicolumn unique indexes, the index ensures that each combination of values in the indexed column is unique. For example, if a unique index **full_name** is created on a combination of **last_name**, **first_name**, and **middle_initial** columns, no two people could have the same full name in the table.

Both clustered and nonclustered indexes can be unique. Therefore, provided that the data in the column is unique, you can create both a unique clustered index and multiple-unique nonclustered indexes on the same table.

Considerations

Specifying a unique index makes sense only when uniqueness is a characteristic of the data itself. If uniqueness must be enforced to ensure data integrity, create a UNIQUE or PRIMARY KEY constraint on the column rather than a unique index. For example, if you plan to query frequently on the Social Security number (**ssn**) column in the **employee** table (in which the primary key is **emp_id**), and you want to ensure that Social Security numbers are unique, create a UNIQUE constraint on **ssn**. If the user enters the same Social Security number for more than one employee, an error is displayed.

Note Creating a PRIMARY KEY or UNIQUE constraint automatically creates a unique index on the specified columns in the table.

Creating a unique index instead of non-unique on the same combination of columns provides additional information for the query optimizer; therefore, creating a unique index is preferred.

See Also

<u>Creating an Index</u> <u>Index Tuning Wizard</u>

Fill Factor

When you create a clustered index, the data in the table is stored in the data pages of the database according to the order of the values in the indexed columns. When new rows of data are inserted into the table or the values in the indexed columns are changed, Microsoft® SQL Server[™] 2000 may have to reorganize the storage of the data in the table to make room for the new row and maintain the ordered storage of the data. This also applies to nonclustered indexes. When data is added or changed, SQL Server may have to reorganize the storage of the data in the nonclustered index pages. When a new row is added to a full index page, SQL Server moves approximately half the rows to a new page to make room for the new row. This reorganization is known as a <u>page split</u>. Page splitting can impair performance and fragment the storage of the data in a table. For more information, see <u>Table and Index Architecture</u>.

When creating an index, you can specify a <u>fill factor</u> to leave extra gaps and reserve a percentage of free space on each <u>leaf level</u> page of the index to accommodate future expansion in the storage of the table's data and reduce the potential for page splits. The fill factor value is a percentage from 0 to 100 that specifies how much to fill the data pages after the index is created. A value of 100 means the pages will be full and will take the least amount of storage space. This setting should be used only when there will be no changes to the data, for example, on a read-only table. A lower value leaves more empty space on the data pages, which reduces the need to split data pages as indexes grow but requires more storage space. This setting is more appropriate when there will be changes to the data in the table.

The fill factor option is provided for fine-tuning performance. However, the server-wide default fill factor, specified using the **sp_configure** system stored procedure, is the best choice in the majority of situations.

Note Even for an application oriented for many insert and update operations, the number of database reads typically outnumber database writes by a factor of 5 to 10. Therefore, specifying a fill factor other than the default can degrade database read performance by an amount inversely proportional to the fill factor setting. For example, a fill factor value of 50 percent can cause database read performance to degrade by two times.

It is useful to set the fill factor option to another value only when a new index is created on a table with existing data, and then only when future changes in that data can be accurately predicted.

The fill factor is implemented only when the index is created; it is not maintained after the index is created as data is added, deleted, or updated in the table. Trying to maintain the extra space on the data pages would defeat the purpose of originally using the fill factor because SQL Server would have to perform page splits to maintain the percentage of free space, specified by the fill factor, on each page as data is entered. Therefore, if the data in the table is significantly modified and new data added, the empty space in the data pages can fill. In this situation, the index can be re-created and the fill factor specified again to redistribute the data.

See Also

<u>Creating an Index</u> <u>fill factor Option</u> <u>Table and Index Architecture</u>

Index Tuning Wizard

The Index Tuning Wizard allows you to select and create an optimal set of indexes and statistics for a Microsoft® SQL Server[™] 2000 database without requiring an expert understanding of the structure of the database, the workload, or the internals of SQL Server.

To build a recommendation of the optimal set of indexes that should be in place, the wizard requires a workload. A workload consists of an SQL script or a SQL Profiler trace saved to a file or table containing SQL batch or remote procedure call (RPC) event classes and the **Event Class** and **Text** data columns. For more information, see <u>TSQL Event Category</u>.

If you do not have an existing workload for the Index Tuning Wizard to analyze, you can create one using SQL Profiler. Either create a workload using the **Sample 1 - TSQL** trace definition or create a new trace that captures the default events and data columns. After you have determined that the trace has captured a representative sample of the normal database activity, the wizard can analyze the workload and recommend an index configuration that will improve the performance of the database.

The Index Tuning Wizard can:

- Recommend the best mix of indexes for a database given a workload, by using the query optimizer to analyze the queries in the workload.
- Analyze the effects of the proposed changes, including index usage, distribution of queries among tables, and performance of queries in the workload.
- Recommend ways to tune the database for a small set of problem queries.
- Allow you to customize the recommendation by specifying advanced options such as disk space constraints.

A recommendation consists of SQL statements that can be executed to create new, more effective indexes and, if wanted, drop existing indexes that are ineffective. Indexed views are recommended on platforms that support their use. After the Index Tuning Wizard has suggested a recommendation, it can then be:

- Implemented immediately.
- Scheduled to be implemented later by creating a SQL Server job that executes an SQL script.
- Saved to an SQL script, to be executed manually by the user at a later time or on a different server.

Considerations

The Index Tuning Wizard does not recommend indexes on:

- Tables referenced by cross-database queries that do not exist in the currently selected database.
- System tables.
- PRIMARY KEY constraints and unique indexes.

Other Index Tuning Wizard considerations include:

- The Index Tuning Wizard is limited to a maximum of 32,767 tunable queries in a workload. Additional queries in the workload will not be considered. Additionally, queries with quoted identifiers are not considered for tuning.
- The Index Tuning Wizard gathers statistics by sampling the data. Consequently, successive executions of the wizard on the same workload may result in variations in the indexes recommended as well as the improvements that result from implementing the recommendation.

- The Index Tuning Wizard cannot be used to select or create indexes and statistics in databases on SQL Server version 6.5 or earlier.
- The Index Tuning Wizard does not give an error when saving an SQL Script to a disk with insufficient available space.
- The Index Tuning Wizard can consume significant CPU and memory resources during analysis. It is recommended that tuning should be performed against a test version of the production server, rather than the production server. Additionally, the wizard should be run on a separate computer from the computer running an instance of SQL Server.

The Index Tuning Wizard may not make index suggestions if:

- There is not enough data in the tables being sampled.
- The suggested indexes do not offer enough projected improvement in query performance over existing indexes.

The queries in the workload are analyzed in the security context of the user who invokes the Index Tuning Wizard. The user must be a member of the **sysadmin** fixed server role.

To reduce the execution time of the Index Tuning Wizard:

- Ensure that **Perform thorough analysis** is not selected in the **Select Server and Database** dialog box. Performing a thorough analysis causes the Index Tuning Wizard to perform an exhaustive analysis of the queries, resulting in a longer execution time. However, selecting this option can result in a greater overall improvement in the performance of the tuned workload.
- Tune only a subset of the tables in the database.

• Reduce the size of the workload file.

The Index Tuning Wizard does not recommend that any indexes be dropped if the **Keep all existing indexes** option is selected. Only new indexes are recommended, if appropriate. Clearing this option can result in a greater overall improvement in the performance of the workload. Additionally, the Index Tuning Wizard does not recommend dropping indexes on PRIMARY KEY constraints or UNIQUE indexes. However, it may drop or replace a clustered index that is not unique or currently created on a PRIMARY KEY constraint.

The Index Tuning Wizard includes any index hint or query hint in the final recommendation, even if the index is not optimal for the table. Indexes on other tables referenced in the query may be proposed and recommended; however, all indexes specified as hints will always be part of the final recommendation. Hints can prevent the Index Tuning Wizard from choosing a better execution plan. Consider removing any index hint from queries before analyzing the workload.

Using Index Tuning Wizard in SQL Query Analyzer

Index Analysis in SQL Query Analyzer allows a single query or batch to be analyzed and a recommendation generated for the optimal set of indexes that should be in place to support the given query or batch. Only members of the **sysadmin** fixed server role can perform Index Analysis using SQL Query Analyzer.

To defer building the indexes recommended by Index Tuning Wizard, save the recommended SQL script using SQL Query Analyzer. Saving the SQL script to a file allows the Transact-SQL statements recommended by Index Analysis to be examined before being executed. The SQL script can then be edited before being executed (for example, the names of the generated indexes can be changed).

To start the Index Tuning Wizard

Creating an Index

After the design has been determined, indexes can be created on the tables in a database.

Microsoft® SQL Server[™] 2000 automatically creates unique indexes to enforce the uniqueness requirements of PRIMARY KEY and UNIQUE constraints. Unless a clustered index already exists on the table or a nonclustered index is explicitly specified, a unique, clustered index is created to enforce the PRIMARY KEY constraint. Unless a clustered index is explicitly specified, a unique, nonclustered index is created by default to enforce the UNIQUE constraint.

If you need to create an index that is independent of a constraint, you can use the CREATE INDEX statement. By default, a nonclustered index is created if the clustering option is not specified.

Additional considerations for creating an index include:

- Only the owner of the table can create indexes on the same table.
- Only one clustered index can be created per table.
- The maximum number of nonclustered indexes that can be created per table is 249 (including any indexes created by PRIMARY KEY or UNIQUE constraints).
- The maximum size of all nonvariable-length columns that comprise the index is 900 bytes. For example, a single index could not be created on three columns defined as **char(300)**, **char(300)**, and **char (301)** because the total width exceeds 900 bytes.
- The maximum number of columns that can comprise the same index is 16.

When you create indexes with the CREATE INDEX statement, you must specify the name of the index, table, and columns to which the index applies. New indexes created as part of a PRIMARY KEY or UNIQUE constraint or using SQL Server Enterprise Manager are automatically given system-defined names based on the database table name. If you create multiple indexes on a table, the index names are appended with _1, _2, and so on. The index can be renamed if necessary.

Note You cannot create an index in the current database while the current database is being backed up.

If a clustered index is created on a table with several secondary indexes, all of the secondary indexes must be rebuilt so that they contain the clustering key value instead of the row identifier (RID). Likewise, if a clustered index is deleted on a table that has several nonclustered indexes, the nonclustered indexes are all rebuilt as part of the DROP operation. This may take significant time on large tables.

The preferred way to build indexes on large tables is to start with the clustered index and then build the nonclustered indexes. When dropping all indexes, drop the nonclustered indexes first and the clustered index last. That way, no indexes need to be rebuilt.

Clustered Indexes

When you create a clustered index, the table is copied, the data in the table is sorted, and then the original table is deleted. Therefore, enough empty space must exist in the database to hold a copy of the data.

By default, the data in the table is sorted when the index is created. However, if the data is already sorted because the clustered index already exists and is being re-created using the same name and columns, the sort operation can be automatically skipped by rebuilding the index, rather than creating the index again. The rebuild operation checks that the rows are sorted while building the index. If any rows are not correctly sorted, the operations cancels and the index is not created.

Unique Indexes

Creating a unique index ensures that any attempt to duplicate key values fails. If

a single query is created that causes duplicate and nonduplicate key values to be added, SQL Server rejects all rows, including the nonduplicate key values. For example, if a single insert statement retrieves 20 rows from table **A** and inserts them into table **B**, and 10 of those rows contain duplicate key values, by default all 20 rows are rejected. However, the IGNORE_DUP_KEY clause can be specified when creating the index that causes only the duplicate key values to be rejected; the nonduplicate key values are added. In the previous example, only the 10 duplicate key values would be rejected; the other 10 nonduplicate key values would be inserted into table B.

A unique index cannot be created if there are any duplicate key values. For example, if you want to create a unique, <u>composite index</u> on columns **a** and **b**, but there are two rows in the table that contain the values 1 and 2 for **a** and **b** respectively, the unique index cannot be created.

Note You cannot create a unique index on a single column if that column contains NULL in more than one row. Similarly, you cannot create a unique index on multiple columns if the combination of columns contains NULL in more than one row. These are treated as duplicate values for indexing purposes.

To create an index when creating a table

■ <u>Transact-SQL</u>
■ <u>Enterprise Manager</u>
■ <u>SQL-DMO</u>
To create an index of

To create an index on an existing table

⊞<u>Transact-SQL</u>

⊞<u>Enterprise Manager</u>

∃ <u>SQL-DMO</u>

You can also create an index using the Create Index Wizard in SQL Server Enterprise Manager.

To create an index using the Create Index Wizard

Creating Indexes on Computed Columns

Indexes can be defined on computed columns, provided these requirements are met:

- The *computed_column_expression* must be deterministic. Expressions are deterministic if they always return the same result for a given set of inputs. *computed_column_expression* is deterministic if:
 - All functions referenced by the expression are deterministic and precise. This includes both user-defined and built-in functions. For more information, see <u>Deterministic and Nondeterministic Functions</u>.
 - All columns referenced in the expression come from the table containing the computed column.
 - No column reference pulls data from multiple rows. For example, aggregate functions such as SUM or AVG depend on data from multiple rows and would make a *computed_column_expression* nondeterministic.

The **IsDeterministic** property of the COLUMNPROPERTY function reports whether a *computed_column_expression* is deterministic.

A computed column expression is precise if:

- It is not an expression of the **float** data type
- It does not use in its definition a float data type. For example, in the following statement, column y is int and deterministic, but not precise:
 CREATE TABLE t2 (a int, b int, c int, x float, y AS CASE x WHEN 0 THEN a WHEN 1 THEN b

ELSE c END)

The **IsPrecise** property of the COLUMNPROPERTY function reports whether a *computed_column_expression* is precise.

Note Any **float** expression is considered nonprecise and cannot be a key of an index; a **float** expression can be used in an indexed view but not as a key. This is true also for computed columns. Any function, expression, user-defined function, or view definition is considered non-deterministic if it contains any **float** expressions, including logical ones (comparisons).

- The ANSI_NULL connection-level option must be set to ON when the CREATE TABLE statement is executed. The OBJECTPROPERTY function reports whether the option is on through the **IsAnsiNullsOn** property.
- The *computed_column_expression* defined for the computed column cannot evaluate to the **text**, **ntext**, or **image** data types.
- The connection on which the index is created, and all connections attempting INSERT, UPDATE, or DELETE statements that will change values in the index, must have six SET options set to ON and one option set to OFF. The optimizer ignores an index on a computed column for any SELECT statement executed by a connection that does not have these same option settings.

These options must be set to ON:

- ANSI_NULLS
- ANSI_PADDING
- ANSI_WARNINGS

- ARITHABORT
- CONCAT_NULL_YIELDS_NULL
- QUOTED_IDENTIFIER

In addition to these ON settings, the NUMERIC_ROUNDABORT option must be set to OFF. For more information, see <u>SET Options That Affect Results</u>.

Creating Indexes on Views

Indexes can be defined on views. Indexed views are a method of storing the result set of the view in the database, thereby reducing the overhead of dynamically building the result set. An indexed view also automatically reflects modifications made to the data in the base tables after the index is created.

Note You can create indexed views only if you install Microsoft® SQL Server[™] 2000 Enterprise Edition or Microsoft SQL Server 2000 Developer Edition.

Indexed views include these benefits:

- Indexed views are implemented through simple syntax extensions to the CREATE INDEX and CREATE VIEW statements.
- The data in indexed views are updated automatically as data in the base tables are updated, in much the same way that the keys in indexes on base tables are updated automatically. You do not need to synchronize the contents of the indexed view with the data in the underlying base tables.
- Indexed views are considered by the SQL Server optimizer without the need to specify special hints in queries. The optimizer considers the indexed view even if a query does not directly reference the view in the FROM clause by trying to match the query plan generated for the view with some portion of the plan generated for the query.
- To introduce indexed views in an existing database, you have to issue only the relevant CREATE VIEW and CREATE INDEX statements. Few changes have to be made to application code for SQL Server to take advantage of any indexes on views.

The Index Tuning Wizard recommends indexed views in addition to recommending indexes on base tables. Using the wizard greatly enhances an administrator's ability to determine the combination of indexes and indexed views that optimize the performance of the typical mix of queries executed against a database.

Indexed views can be more complex to maintain than indexes based on base tables. You should create indexes only on views where the improved speed in retrieving results outweighs the increased overhead of making modifications.

See Also

Designing an Indexed View Creating an Indexed View Using Indexes on Views

SET Options That Affect Results

Indexed views and indexes on computed columns involve storing results in the database for later reference. These stored results are valid only if all connections referring to the results can generate the same result set as the connection that created the stored result set.

Indexed Views

Indexed views store the result set returned by a view by creating a clustered index on the view. For complex views, the stored result set greatly speeds data retrieval. An indexed view is useful only as long as all operations referencing the view use exactly the same algorithms when building their results. Like indexes for computed columns, this includes:

- The CREATE INDEX statement that first builds the result set.
- Any subsequent INSERT, UPDATE, or DELETE statements that affect the base data used to build the view result set.
- All queries for which the optimizer must determine if the indexed view will be useful.

Indexes on Computed Columns

Indexes on computed columns must calculate the computed column values to build the keys stored in the index. An index on a computed column works only as long as all operations using the index use exactly the same algorithms to determine the key values:

- The original CREATE INDEX statement that establishes the first set of key values.
- As later INSERT, UPDATE and DELETE statements create, alter, or remove key values, the operations are not valid unless the key values

are computed with the same algorithms used by the original create index operation.

• For the index to be useful for any subsequent statement, all of the key values stored in the index must be the same as would be generated by the current settings of the connection executing the statement.

SET Option Settings

Any SET options that affect the results generated by Transact-SQL statements must have the same settings for all operations referencing the index. There are seven SET options that affect the results stored in computed columns and returned by views. All connections using indexes on computed columns or indexed views must have the same settings for these seven options:

- These six SET options must be set to ON:
 - ANSI_NULLS
 - ANSI_PADDING
 - ANSI_WARNINGS
 - ARITHABORT
 - CONCAT_NULL_YIELDS_NULL
 - QUOTED_IDENTIFIER
- The NUMERIC_ROUNDABORT option must be set to OFF.

These SET options must be set correctly for any connection that creates an index on a view or computed column. Any connection executing INSERT, UPDATE or DELETE statements that change data values stored in the indexes must have the correct settings. This includes bulk copy, Data Transformation Services (DTS), and replication operations. Microsoft® SQL Server[™] 2000 generates an error and rolls back any insert, update, or delete operation attempted by a connection that does not have the proper option settings. The optimizer does not consider using an index on a computed column or view in the execution plan of any Transact-SQL statement if the connection does not have the correct option settings.

For example, a table is defined and populated using this script:

```
CREATE TABLE Parts
 (PartID
              int PRIMARY KEY,
 PartName
                char(10),
 PartMaterial
                char(10),
 PartColor
               char(10),
 PartDescription AS PartMaterial + PartColor
 )
GO
INSERT INTO Parts VALUES (1, 'Table', 'Wood', 'Red')
INSERT INTO Parts VALUES (2, 'Chair', 'Fabric', 'Blue')
INSERT INTO Parts VALUES (3, 'Bolt', 'Steel', NULL)
GO
```

The value calculated for the **PartDescription** column for the row, where **PartID** is 3, depends on the CONCAT_NULL_YIELDS_NULL option. If CONCAT_NULL_YIELDS_NULL is set to ON, the calculated value is NULL. If CONCAT_NULL_YIELDS_NULL is set to OFF, the calculated value is the string 'Steel'. For an index on the **PartDescription** column to be properly maintained, all INSERT, UPDATE, and DELETE operations must have the same setting of CONCAT_NULL_YIELDS_NULL ON as the connection that created the index. The index is also not used by the optimizer for any connection with a different CONCAT_NULL_YIELDS_NULL setting from the connections that created the key values.

SET Option Settings for OLE DB and ODBC Connections

Six of the seven SET option settings required for indexes on computed columns and views are the default settings for the OLE DB Provider for SQL Server and

the SQL Server ODBC driver. These settings are:

- ANSI_NULLS
- ANSI_PADDING
- ANSI_WARNINGS
- CONCAT_NULL_YIELDS_NULL
- NUMERIC_ROUNDABORT
- QUOTED_IDENTIFIER

These settings also enforce the rules of the SQL-92 standard and are the recommended settings for SQL Server. Because DTS, replication, and bulk copy operations in SQL Server 2000 use OLE DB or ODBC, these options are also automatically set for these operations. Some of the SQL Server utilities set one or more of the ANSI settings to OFF to maintain compatibility with earlier versions of the utilities.

SET ARITHABORT ON is the one option that is not automatically set for connections using the OLE DB Provider for SQL Server or the SQL Server ODBC driver. OLE DB and ODBC connections do not specify an ARITHABORT setting, so connections default to the server default, which is ARITHABORT OFF. This server default is controlled by the **user options** server option. The **user options** bit that equates to 64 should be set for any server on which you implement indexes on views or computed columns. For more information about how to set this option, see <u>user options Option</u>.

Precedence for Setting Options

The settings for the SET options can be specified at several levels. The final setting for each session option for a particular connection is determined by the highest precedence operation that sets the option. The precedence of the sessionsetting operations is (with the highest precedent at the top of the list):

- Any application can explicitly override any default settings by executing a SET statement after it has connected to a server. The SET statement overrides all previous settings and can be used to turn options on and off dynamically as the application executes.
- OLE DB and ODBC applications can specify the option settings that are in effect at connection time by specifying option settings in connection strings.
- You can SET options to ON or OFF for any SQL Server ODBC data source by using the ODBC application in Control Panel, or the ODBC **SQLConfigDataSource** function. Any connection made by an ODBC application using that data source uses the specified defaults, unless the application overrides the defaults in the connect string or with SET statements after connecting.
- The OLE DB Provider for SQL Server and the SQL Server ODBC driver automatically set the seven session options to the settings required for indexed views. DB-Library and Embedded SQL for C applications do not, so systems using these APIs must either code the applications to issue the proper SET statements or change the database or server defaults to the correct settings.
- You can establish default settings for a database using ALTER DATABASE or SQL Server Enterprise Manager.
- You can establish default settings for a server by using either **sp_configure** or SQL Server Enterprise Manager to set the server configuration option named user options. For more information, see <u>user options Option</u>.

The connection option settings required for indexed views and indexes on computed columns must be active:

- For any connection that creates an index on a view or computed column.
- For any INSERT, UPDATE, or DELETE statements that attempt to modify data covered by an index on a view or computed column.
- Before the optimizer can consider using an index on a view or computed column to cover a query.
- For indexed views, the ANSI_NULLS and QUOTED_IDENTIFIER options must be set to ON when the view is created, because these two settings are stored as object properties with the view definition.

Considerations

The SET statement can change the options dynamically; therefore, issuing SET statements in a database that has indexes on views and computed columns must be done carefully. For example, an application can make a connection in which the default settings allow an indexed view to be referenced. If the connection calls a stored procedure whose first statement is SET ANSI_WARNINGS OFF, that statement overrides previous defaults or settings for ANSI_WARNINGS. The optimizer ignores all indexed views or indexes on computed columns when processing any statement in the stored procedure. Any statements in the stored procedure that attempted an INSERT, UPDATE, or DELETE that affected an indexed view or an index on a computed column generate an error.

The logic in some stored procedures or triggers originally developed in earlier versions of SQL Server depends on options such as QUOTED_IDENTIFIER or ANSI_NULLS being set to OFF. Also, DB-Library and Embedded SQL for C applications do not, by default, set any session options. Connections from these applications can create problems for other stored procedures or triggers that depend on the options being set to ON. The recommended solution has been to code SET statements at the start of either of these types of stored procedures and triggers to ensure they had the operating environment they required. In SQL Server 2000, if a stored procedure or trigger sets any of the options needed by indexes on views and computed columns to a value other than those required by

the indexes, the indexes are not used to cover any SELECT statements executed by the stored procedure or trigger. Any INSERT, UPDATE, or DELETE statements executed by these stored procedures and triggers fails if they modify data covered by an index on a view or computed column. In SQL Server 2000 instances that use indexes on views and computed columns, stored procedures and triggers should be written to work with the seven SET options needed to support these indexes. SET statements should be used only in stored procedures and triggers for these systems if they receive connections from clients using DB-Library, Embedded SQL for C, or ODBC drivers from SQL Server version 6.5 or earlier. The stored procedures and triggers should set only the options to those required by indexes on views and computed columns.

Three other session options can potentially affect the format of result sets: DATEFIRST, DATEFORMAT, and LANGUAGE. Any functions whose results would be affected by changes to these options are classified as nondeterministic and cannot be used in views or computed columns that are indexed.

See Also

CREATE INDEX Distributed Queries SET ANSI_NULLS SET ANSI_PADDING SET ANSI_WARNINGS SET ARITHABORT SET CONCAT_NULL_YIELDS_NULL SET NUMERIC_ROUNDABORT SET QUOTED_IDENTIFIER

Creating Ascending and Descending Indexes

When defining indexes, you can specify whether the data for each column is stored in ascending or descending order. If neither direction is specified, ascending is the default, which maintains compatibility with earlier versions of Microsoft® SQL ServerTM 2000.

The syntax of the CREATE TABLE, CREATE INDEX, and ALTER TABLE statements supports the keywords ASC (specifies ascending) and DESC (specifies descending) on individual columns in indexes:

```
CREATE TABLE ObjTable
(ObjID int PRIMARY KEY,
ObjName char(10),
ObjWeight decimal(9,3)
)
CREATE NONCLUSTERED INDEX DescIdx ON
ObjTable(ObjName ASC, ObjWeight DESC)
```

The INDEXKEY_PROPERTY meta data function reports whether an index column is stored in ascending or descending order. In addition, the **sp_helpindex** and **sp_helpconstraint** system stored procedures report the direction of index key columns. The descending indexed column will be listed in the result set with a minus sign (-) following its name. The default, an ascending indexed column, will be listed by its name alone.

The ability to specify the order in which key values are stored in an index is most useful in cases where most queries referencing the table have ORDER BY clauses that specify different directions for the key columns. For example, the index defined previously for the **ObjTable** can completely eliminate the need for an ORDER BY clause such as:

```
ORDER BY ObjName ASC, ObjWeight DESC
```

The internal algorithms of SQL Server can navigate equally efficiently in both directions on a single-column index, regardless of the sequence in which the keys are stored. For example, specifying DESC on a single-column index does

not make queries with an ORDER BY IndexKeyCol DESC clause run faster than if ASC was specified for the index.

Statistical Information

Microsoft® SQL Server[™] 2000 allows statistical information regarding the distribution of values in a column to be created. This statistical information can be used by the query processor to determine the optimal strategy for evaluating a query. When you create an index, SQL Server automatically stores statistical information regarding the distribution of values in the indexed column(s). The query optimizer in SQL Server uses these statistics to estimate the cost of using the index for a query. Additionally, when the AUTO_CREATE_STATISTICS database option is set to ON (default), SQL Server automatically creates statistics for columns without indexes that are used in a predicate.

As the data in a column changes, index and column statistics can become out-ofdate and cause the query optimizer to make less-than-optimal decisions on how to process a query. For example, if you create a table with an indexed column and 1,000 rows of data, all with unique values in the indexed column, the query optimizer considers the indexed column a good way to collect the data for a query. If you update the data in the column so there are many duplicated values, the column is no longer an ideal candidate for use in a query. However, the query optimizer still considers it to be a good candidate based on the index's outdated distribution statistics, which are based on the data before the update.

Note Out-of-date or missing statistics are indicated as warnings (table name in red text) when the execution plan of a query is graphically displayed using SQL Query Analyzer. For more information, see <u>Graphically Displaying the Execution Plan Using SQL Query Analyzer</u>. Additionally, monitoring the **Missing Column Statistics** event class using SQL Profiler indicates when statistics are missing. For more information, see <u>Errors and Warnings Event Category</u>.

Therefore, SQL Server automatically updates this statistical information periodically as the data in the tables changes. The sampling is random across data pages, and taken from the table or the smallest nonclustered index on the columns needed by the statistics. After a data page has been read from disk, all the rows on the data page are used to update the statistical information. The frequency at which the statistical information is updated is determined by the volume of data in the column or index and the amount of changing data. For example, the statistics for a table containing 10,000 rows may need updating when 1,000 index values have changed because 1,000 values may represent a significant percentage of the table. However, for a table containing 10 million index entries, 1,000 changing index values is less significant, and so the statistics may not be automatically updated. SQL Server, however, always ensures that a minimum number of rows are sampled; tables that are smaller than 8 megabytes (MB) are always fully scanned to gather statistics.

The cost of this automatic statistical update is minimized by sampling the data, rather than analyzing all of it. Under some circumstances, statistical sampling will not be able to accurately characterize the data in a table. You can control the amount of data that is sampled during manual statistics updates on a table-by-table basis by using the SAMPLE and FULLSCAN clauses of the UPDATE STATISTICS statement. The FULLSCAN clause specifies that all of the data in the table is scanned to gather statistics, whereas the SAMPLE clause can be used to specify either the percentage of rows to sample or the number of rows to sample.

You can also tell SQL Server not to maintain statistics for a given column or index in these ways:

- Use the **sp_autostats** system stored procedure.
- Use the STATISTICS_NORECOMPUTE clause of the CREATE INDEX statement.
- Use the NORECOMPUTE clause of the UPDATE STATISTICS statement.
- Use the NORECOMPUTE clause of the CREATE STATISTICS statement.
- Set the AUTO_CREATE_STATISTICS and AUTO_UPDATE_STATISTICS database options to OFF using the ALTER DATABASE statement. For more information, see <u>Setting</u> <u>Database Options</u>.

If you instruct SQL Server not to maintain statistics automatically, you must manually update the statistical information.

Statistics can also be created on all eligible columns in all user tables in the current database in a single statement by using the **sp_createstats** system stored procedure. Columns not eligible for statistics include nondeterministic or nonprecise computed columns, or columns of **image**, **text**, and **ntext** data types.

The statistics generated for a column can be deleted if you no longer want to retain and maintain them. Statistics created on columns by SQL Server (when the AUTO_CREATE_STATISTICS database option is set to ON) are aged and dropped automatically.

Creating statistics manually allows you to create statistics that contain multiple column densities (average number of duplicates for the combination of columns). For example, a query contains the clause:

WHERE a = 7 and b = 9

Creating manual statistics on both columns together (**a**, **b**) can allow SQL Server to make a better estimate for the query because the statistics also contain the average number of distinct values for the combination of columns **a** and **b**.

To create statistics on a column

∃ <u>Transact-SQL</u>

Rebuilding an Index

When you create an index in the database, the index information used by queries is stored in <u>index pages</u>. The sequential index pages are chained together by pointers from one page to the next. When changes are made to the data that affect the index, the information in the index can become scattered in the database. Rebuilding an index reorganizes the storage of the index data (and table data in the case of a clustered index) to remove fragmentation. This can improve disk performance by reducing the number of page reads required to obtain the requested data

In Microsoft® SQL Server[™] 2000, rebuilding an index using the DROP_EXISTING clause of the CREATE INDEX statement can be efficient if you re-create the index in a single step, rather than delete the old index and then create the same index again. This is a benefit for both clustered and nonclustered indexes.

Rebuilding a clustered index by deleting the old index and then re-creating the same index again is expensive because all the secondary indexes use the clustering key to point to the data rows. If you simply delete the clustered index and re-create it, you cause all the nonclustered indexes to be deleted and re-created twice. This occurs once when you delete the clustered index, and a second time when you re-create it. You avoid this expense by re-creating the index in one step. Re-creating the index in a single step tells SQL Server that you are reorganizing an existing index and avoids the unnecessary work of deleting and re-creating nonclustered indexes. This method also has the significant advantage of using the sorted order of the data in the existing index, thus avoiding the need to sort the data again. This is useful for both clustered and nonclustered indexes, and significantly reduces the cost of rebuilding an index. Additionally, SQL Server allows you to rebuild (in one step) one or more indexes on a table by using the DBCC DBREINDEX statement, without having to rebuild each index separately.

DBCC DBREINDEX is also useful to rebuild indexes enforcing PRIMARY KEY or UNIQUE constraints without having to delete and re-create the constraints (because an index created to enforce a PRIMARY KEY or UNIQUE constraint cannot be deleted without deleting the constraint first). For example, you may want to rebuild an index on a PRIMARY KEY constraint to reestablish a given fill factor for the index.

To delete an index

- ⊞ <u>Transact-SQL</u>
- <u>■ Enterprise Manager</u>
- ∃ <u>SQL-DMO</u>

To create an index on an existing table

- ∃ <u>Transact-SQL</u>
- <u>■ Enterprise Manager</u>

∃ <u>SQL-DMO</u>

To re-create an index in one step

⊞ <u>Transact-SQL</u>

∃ <u>SQL-DMO</u>

To rebuild one or more indexes on a table

∃ <u>Transact-SQL</u>

To modify an index

<u>■ Enterprise Manager</u>

See Also

Creating an Index

Deleting an Index

Renaming an Index

You can rename an index as long as index names are unique within the table. For example, two tables can have an index named **XPK_1**, but the same table cannot have two indexes named **XPK_1**. Renaming an index changes only the name of the index; the index is not rebuilt.

Note When you create a PRIMARY KEY or UNIQUE constraint on a table, an index with the same name as the constraint is automatically created for the table. Because index names must be unique for a table, you cannot create or rename an index to have the same name as the PRIMARY KEY or UNIQUE constraint for the table.

To rename an index

∃ <u>Transact-SQL</u>

<u>■ Enterprise Manager</u>

∃ <u>SQL-DMO</u>

To modify an index

<u>Enterprise Manager</u>

See Also

Creating an Index

Viewing an Index

After you have created indexes or PRIMARY KEY or UNIQUE constraints on tables, you may need to find information about the indexes. For example, you may need to find out the types of indexes and the columns that are indexes on a particular table or the total space in the database used by an index.

Each table registered for full-text indexing has one of its indexes selected as the full-text key. You can view the properties of an index to determine if an index is the full-text key.

To view the indexes on a table

∃ <u>Transact-SQL</u>

<u>■ Enterprise Manager</u>

∃ <u>SQL-DMO</u>

SQL Server Enterprise Manager can display the names of all the indexes in a database and the tables to which the indexes belong.

To view all indexes in a database

Deleting an Index

When you no longer need an index, you can delete it from a database and reclaim the storage space it currently uses. This reclaimed space can then be used by any object in the database.

Deleting a clustered index can take some time, because all nonclustered indexes on the same table must be rebuilt. For more information about the relationship between clustered and nonclustered indexes, see <u>Nonclustered Indexes</u>.

You cannot delete an index used by either a PRIMARY KEY or UNIQUE constraint without deleting the constraint. To delete and re-create an index used by a PRIMARY KEY or UNIQUE constraint without having to delete and re-create the constraint (for example, to reimplement the original fill factor used by the index), rebuild the index in one step. For more information about rebuilding the index, see <u>Rebuilding an Index</u>. An index specified as the full-text key for the table cannot be deleted. View index properties to determine if the index is the full-text key.

Rebuilding an index, rather than deleting and re-creating it, is also useful to recreate a clustered index, because the process of rebuilding the index can remove the need to sort the data by the index columns if the data is already in sorted order.

Indexes created on any views or tables (permanent and temporary) are automatically deleted when the view or table is deleted.

Note Only the owner of a table can delete its indexes. The owner cannot transfer the permission to other users.

To delete an index

∃ <u>Transact-SQL</u>

<u>■ Enterprise Manager</u>

∃ <u>SQL-DMO</u>

To view index properties

⊞ <u>Transact-SQL</u>

∃ <u>SQL-DMO</u>

See Also

Creating an Index

Views

A view is a virtual table whose contents are defined by a query. Like a real table, a view consists of a set of named columns and rows of data. However, a view does not exist as a stored set of data values in a database. The rows and columns of data come from tables referenced in the query defining the view and are produced dynamically when the view is referenced.

A view acts as a filter on the underlying tables referenced in the view. The query that defines the view can be from one or more tables or from other views in the current or other databases. Distributed queries can also be used to define views that use data from multiple heterogeneous sources. This is useful, for example, if you want to combine similarly structured data from different servers each of which stores data for a different region of your organization.

There are no restrictions on querying through views and few restrictions on modifying data through them.

This illustration shows a view based on two tables.

See Also

SQL Views

Scenarios for Using Views

Views are generally used to focus, simplify, and customize the perception each user has of the database. Views can be used as security mechanisms by allowing users to access data through the view, without granting the users permissions to directly access the underlying base tables of the view. Views can also be used, when copying data to and from Microsoft® SQL Server[™] 2000, to improve performance and to partition data.

To Focus on Specific Data

Views allow users to focus on specific data that interests them and on the specific tasks for which they are responsible. Unnecessary data can be left out of the view. This also increases the security of the data because users can see only the data that is defined in the view and not the data in the underlying table. For more information about using views for security purposes, see <u>Using Views as</u> <u>Security Mechanisms</u>.

To Simplify Data Manipulation

Views can simplify how users manipulate data. You can define frequently used joins, projections, UNION queries, and SELECT queries as views so that users do not have to specify all the conditions and qualifications each time an additional operation is performed on that data. For example, a complex query that is used for reporting purposes and performs subqueries, outer joins, and aggregation to retrieve data from a group of tables can be created as a view. The view simplifies access to the data because the underlying query does not have to be written or submitted each time the report is generated; the view is queried instead. For more information about manipulating data, see Query Fundamentals.

You can also create inline user-defined functions that logically operate as parameterized views, or views that have parameters in WHERE-clause search conditions. For more information, see <u>Inline User-defined Functions</u>.

To Customize Data

Views allow different users to see data in different ways, even when they are using the same data concurrently. This is particularly advantageous when users with many different interests and skill levels share the same database. For example, a view can be created that retrieves only the data for the customers with whom an account manager deals. The view can determine which data to retrieve based on the login ID of the account manager who uses the view.

To Export and Import Data

Views can be used to export data to other applications. For example, you may want to use the **stores** and **sales** tables in the **pubs** database to analyze sales data using Microsoft® Excel. To do this, you can create a view based on the **stores** and **sales** tables. You can then use the **bcp** utility to export the data defined by the view. Data can also be imported into certain views from data files using the **bcp** utility or BULK INSERT statement providing that rows can be inserted into the view using the INSERT statement. For more information about the restrictions for copying data into views, see <u>INSERT</u>. For more information afout using the **bcp** utility and BULK INSERT statement to copy data to and from a view, see <u>Copying To or From a View</u>.

To Combine Partitioned Data

The Transact-SQL UNION set operator can be used within a view to combine the results of two or more queries from separate tables into a single result set. This appears to the user as a single table called a partitioned view. For example, if one table contains sales data for Washington, and another table contains sales data for California, a view could be created from the UNION of those tables. The view represents the sales data for both regions.

To use partitioned views, you create several identical tables, specifying a constraint to determine the range of data that can be added to each table. The view is then created using these base tables. When the view is queried, SQL Server automatically determines which tables are affected by the query and references only those tables. For example, if a query specifies that only sales data for the state of Washington is required, SQL Server reads only the table containing the Washington sales data; no other tables are accessed.

Partitioned views can be based on data from multiple heterogeneous sources, such as remote servers, not just tables in the same database. For example, to

combine data from different remote servers each of which stores data for a different region of your organization, you can create distributed queries that retrieve data from each data source, and then create a view based on those distributed queries. Any queries read only data from the tables on the remote servers that contains the data requested by the query; the other servers referenced by the distributed queries in the view are not accessed.

When you partition data across multiple tables or multiple servers, queries accessing only a fraction of the data can run faster because there is less data to scan. If the tables are located on different servers, or on a computer with multiple processors, each table involved in the query can also be scanned in parallel, thereby improving query performance. Additionally, maintenance tasks, such as rebuilding indexes or backing up a table, can execute more quickly.

By using a partitioned view, the data still appears as a single table and can be queried as such without having to reference the correct underlying table manually.

Partitioned views are updatable if either of these conditions is met:

- An INSTEAD OF trigger is defined on the view with logic to support INSERT, UPDATE, and DELETE statements.
- Both the view and the INSERT, UPDATE, and DELETE statements follow the rules defined for updatable partitioned views. For more information, see <u>Creating a Partitioned View</u>.

See Also

Join Fundamentals

Using Views with Partitioned Data

Creating a View

Before you create a view, consider these guidelines:

- You can create views only in the current database. However, the tables and views referenced by the new view can exist in other databases or even other servers if the view is defined using distributed queries.
- View names must follow the rules for identifiers and must be unique for each user. Additionally, the name must not be the same as any tables owned by that user.
- You can build views on other views and on procedures that reference views. Microsoft® SQL ServerTM 2000 allows views to be nested up to 32 levels.
- You cannot associate rules or DEFAULT definitions with views.
- You cannot associate AFTER triggers with views, only INSTEAD OF triggers.
- The query defining the view cannot include the ORDER BY, COMPUTE, or COMPUTE BY clauses or the INTO keyword.
- You cannot define full-text index definitions on views.
- You cannot create temporary views, and you cannot create views on temporary tables.
- Views or tables participating in a view created with the SCHEMABINDING clause cannot be dropped, unless the view is

dropped or changed so that it no longer has schema binding. In addition, ALTER TABLE statements on tables that participate in views having schema binding will fail if these statements affect the view definition.

- You cannot issue full-text queries against a view, although a view definition can include a full-text query if the query references a table that has been configured for full-text indexing.
- You must specify the name of every column in the view if:
 - Any of the columns in the view is derived from an arithmetic expression, a built-in function, or a constant.
 - Two or more of the columns in the view would otherwise have the same name (usually because the view definition includes a join and the columns from two or more different tables have the same name).
 - You want to give any column in the view a name different from the column from which it is derived. (You can also rename columns in the view.) A view column inherits the data type of the column from which it is derived, whether or not you rename it.

Note This rule does not apply when a view is based on a query containing an outer join because columns may change from not allowing null values to allowing them.

Otherwise, you do not need to specify column names when creating the view. SQL Server gives the columns of the view the same names and data types as the columns to which the query defining the view refers. The select list can be a full or partial list of the column names in the base tables.

To create a view you must be granted permission to do so by the database owner and you must have appropriate permissions on any tables or views referenced in the view definition.

By default, as rows are added or updated through a view, they disappear from the scope of the view when they no longer fall into the criteria of the query defining the view. For example, a query can be created, defining a view that retrieves all rows from a table where the employee's salary is less than \$30,000. If the employee's salary is increased to \$32,000, then querying the view no longer displays that particular employee because his or her salary does not conform to the criteria set by the view. However, the WITH CHECK OPTION clause forces all data modification statements executed against the view to adhere to the criteria set within the SELECT statement defining the view. If you use this clause, rows cannot be modified in a way that causes them to disappear from the view. Any modification that would cause this to happen is canceled and an error is displayed.

The definition of a sensitive view can be encrypted to ensure that its definition cannot be obtained by anyone, including the owner of the view.

To create a view

⊞ <u>Transact-SQL</u>

∃ <u>SQL-DMO</u>

You can also create a view using the SQL Server Enterprise Manager Create View Wizard.

To create a view using the Create View Wizard

Designing an Indexed View

Indexed views improve the performance of some types of queries dramatically.

Note You can create indexed views only if you install Microsoft® SQL Server[™] 2000 Enterprise Edition or Microsoft SQL Server 2000 Developer Edition.

Indexed views work best when the underlying data is infrequently updated. The maintenance of an indexed view can be higher than the cost of maintaining a table index. If the underlying data is updated frequently, then the cost of maintaining the indexed view data may outweigh the performance benefits of using the indexed view.

Indexed views improve the performance of these types of queries:

- Joins and aggregations that process many rows.
- Join and aggregation operations that are frequently performed by many queries.

For example, in an online-transaction-processing (OLTP) database that is recording inventories, many queries would be expected to join the **Parts, PartSupplier**, and **Suppliers** tables. Although each query that performs this join may not process many rows, the overall join processing of hundreds of thousands of such queries can be significant. Because these relationships are not likely to be updated frequently, the overall performance of the entire system could be improved by defining an indexed view that stores the joined results.

• Decision support workloads.

Analysis systems are characterized by storing summarized, aggregated data that is infrequently updated. Further aggregating the data and joining many rows characterizes many decision support queries.

Indexed views usually do not improve the performance of these types of queries:

• OLTP systems with many writes.

- Databases with many updates.
- Queries that do not involve aggregations or joins.
- Aggregations of data with a high degree of cardinality for the key. A high degree of cardinality means the key contains many different values. A unique key has the highest possible degree of cardinality because every key has a different value. Indexed views improve performance by reducing the number of rows a query has to access. If the view result set has almost as many rows as the base table, then there is little performance benefit from using the view. For example, consider this query on a table that has 1,000 rows:
 SELECT PriKey, SUM(SalesCol)
 FROM ExampleTable
 GROUP BY PriKey

If the cardinality of the table key is 100, then an indexed view built using the result of this query would only have 100 rows. Queries using the view would on average need one tenth of the reads needed against the base table. If the key is a unique key, the cardinality of the key is 1000 and the view result set returns 1000 rows. A query has no performance gain from using this indexed view instead of directly reading the base table.

• Expanding joins, which are views whose result sets are larger than the original data in the base tables.

Combining Indexed Views with Queries

Although the restrictions on the types of views that can be indexed may prevent you from designing a view that solves a complete problem, you may be able to design multiple smaller indexed views that speed parts of the process.

Consider these examples:

- A frequently executed query aggregates data in one database, aggregates data in another database, and then joins the results. Because an indexed view cannot reference tables from more than one database, you cannot design a single view to perform the entire process. You can, however, create an indexed view in each database that does the aggregation for that database. If the optimizer can match the indexed views against existing queries, at least the aggregation processing will be speeded up without the need to recode existing queries. Although the join processing is not faster, the overall query is faster because it uses the aggregations stored in the indexed views.
- A frequently executed query aggregates data from several tables, and then uses UNION to combine the results. UNION is not allowed in an indexed view. You can once again design views to do each of the individual aggregation operations. The optimizer can then select the indexed views to speed up queries with no need to recode the queries. While the UNION processing is not improved, the individual aggregation processes are.

Design indexed views that can satisfy multiple operations. Because the optimizer can use an indexed view even when it is not specified in the FROM clause, a well-designed indexed view can speed the processing of many queries.

For example, consider creating an index on this view:

```
CREATE VIEW ExampleView (PriKey, SumColx, CountColx)
AS
SELECT PriKey, SUM(Colx), COUNT_BIG(Colx)
FROM MyTable
GROUP BY PriKey
```

Not only can this view satisfy queries that directly reference the view columns, it can also be used to satisfy queries that query the base table and contain expressions such as SUM(Colx), COUNT_BIG(Colx), COUNT(Colx), and AVG(Colx). All such queries will be faster because they only have to retrieve the small number of rows in the view rather than reading the full number of rows from the base tables.

See Also

Creating Indexes on Computed Columns Resolving Indexes on Views View Indexes

Using 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 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 optimizer.

Other than the requirements for the SET options, these are the same rules the optimizer uses to determine if an index covers a query. Nothing has to be specified in the query to make use of an indexed view.

A query also does not have to specifically reference an indexed view in the FROM clause for the 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 optimizer estimates that using the indexed view provides the lowest cost access mechanism, the optimizer chooses the indexed view, similar to the way it chooses base table indexes when they are not directly referenced in a query. The 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 usually best, however, to let the 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 significantly improve performance.

The EXPAND VIEWS option specifies that the optimizer not use any view indexes for the entire query.

- The optimizer does not use any indexed views. The optimizer ignores all view indexes when estimating the low-cost method for covering columns referenced in the query.
- The optimizer treats an indexed view referenced in the FROM clause as a standard view. The optimizer incorporates the logic of the view into the query execution plan and dynamically builds the result set from the base tables. The optimizer ignores indexes defined on the view.

When NOEXPAND is specified for a view, the optimizer considers the use of any indexes defined on the view. NOEXPAND specified with the optional INDEX() clause forces the optimizer to use the specified indexes. NOEXPAND can be specified only for an indexed view and cannot be specified for a view that has not been indexed.

See Also

<u>FROM</u>

Resolving Indexes on Views

<u>SELECT</u>

Creating an Indexed View

Views are also known as 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 standard view is not stored permanently in the database. Each time a query references the view, Microsoft® SQL Server[™] 2000 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 standard 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, such as aggregating large amounts of data, or joining 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>.

Note You can create indexed views only if you install Microsoft SQL Server 2000 Enterprise Edition or Microsoft SQL Server 2000 Developer Edition.

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. For more information, see <u>Using Indexes on Views</u>.

Creating a clustered index on a view stores the data as it exists 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 clustered index of the view be unique improves the efficiency with which SQL Server can find the rows in the index

that are affected by any data modification.

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 that are mapped over relatively static data, process many rows, and are referenced by many queries.

Requirements for the View

A view must meet these requirements before you can create a clustered index on it:

- The ANSI_NULLS and QUOTED_IDENTIFIER options must have been set to ON when the CREATE VIEW statement was executed. The OBJECTPROPERTY function reports this for views through the **ExecIsAnsiNullsOn** or **ExecIsQuotedIdentOn** properties.
- The ANSI_NULLS option must have been set to ON for the execution of all CREATE TABLE statements that create tables referenced by the view.
- The view must not reference any other views, only base tables.
- All base tables referenced by the view must be in the same database as the view and have the same owner as the view.
- The view must be created with the SCHEMABINDING option. SCHEMABINDING binds the view to the schema of the underlying base tables.
- User-defined functions referenced in the view must have been created with the SCHEMABINDING option.
- Tables and user-defined functions must be referenced by two-part

names. One-part, three-part, and four-part names are not allowed.

- All functions referenced by expressions in the view must be deterministic. The **IsDeterministic** property of the OBJECTPROPERTY function reports if a user-defined function is deterministic. For more information, see <u>Deterministic and Nondeterministic Functions</u>.
- The SELECT statement in the view cannot contain these Transact-SQL syntax elements:
 - The select list cannot use the * or *table_name*.* syntax to specify columns. Column names must be explicitly stated.
 - A table column name used as a simple expression cannot be specified in more than one view column. A column can be referenced multiple times provided all, or all but one, reference to the column is part of a complex expression or a parameter to a function. For example, this select list is invalid: SELECT ColumnA, ColumnB, ColumnA

These select lists are valid:

SELECT ColumnA, AVG(ColumnA), ColumnA + Colu

SELECT SUM(ColumnA), ColumnA % ColumnB AS]

- A derived table.
- Rowset functions.
- UNION operator.
- Subqueries.

- Outer or self joins.
- TOP clause.
- ORDER BY clause.
- DISTINCT keyword.
- COUNT(*) (COUNT_BIG(*) is allowed.)
- The AVG, MAX, MIN, STDEV, STDEVP, VAR, or VARP aggregate functions. If AVG, MAX, MIN, STDEV, STDEVP, VAR, or VARP are specified in queries referencing the indexed view, the optimizer can often calculate the needed result if the view select list contains these substitute functions.

	Substitute simple aggregate functions
AVG(X)	

SUM(*X*), COUNT_BIG(*X*) STDEV(*X*)SUM(*X*), COUNT_BIG(*X*), SUM(*X***2) STDEVP(*X*)SUM(*X*), COUNT_BIG(*X*), SUM(*X***2) VAR(*X*)SUM(*X*), COUNT_BIG(*X*), SUM(*X***2) VARP(*X*)SUM(*X*), COUNT_BIG(*X*), SUM(*X***2)

For example, an indexed view select list cannot contain the expression AVG(SomeColumn). If the view select list contains the expressions SUM(SomeColumn) and COUNT_BIG(SomeColumn), SQL Server can calculate the average for a query that references the view and specifies AVG(SomeColumn).

• A SUM function that references a nullable expression.

- The full-text predicates CONTAINS or FREETEXT.
- COMPUTE or COMPUTE BY clause.
- If GROUP BY is not specified, the view select list cannot contain aggregate expressions.
- If GROUP BY is specified, the view select list must contain a COUNT_BIG(*) expression, and the view definition cannot specify HAVING, CUBE, or ROLLUP.

• A column resulting from an expression that either evaluates to a **float** value or uses **float** expressions for its evaluation cannot be a key of an index in an indexed view or a table.

Requirements for the CREATE INDEX Statement

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. For more information, see <u>CREATE INDEX</u>.

The CREATE INDEX statement must meet these requirements in addition to the normal CREATE INDEX requirements:

- The user executing the CREATE INDEX statement must be the view owner.
- These SET options must be set to ON when the CREATE INDEX statement is executed:
 - ANSI_NULLS
 - ANSI_PADDING

- ANSI_WARNINGS
- ARITHABORT
- CONCAT_NULL_YIELDS_NULL
- QUOTED_IDENTIFIERS
- The NUMERIC_ROUNDABORT option must be set to OFF.
- The view cannot include **text**, **ntext**, or **image** columns, even if they are not referenced in the CREATE INDEX statement.
- If the SELECT statement in the view definition specifies a GROUP BY clause, the key of the unique clustered index can reference only columns specified in the GROUP BY clause.

Considerations

After the clustered index is created, any connection attempting to modify the base data for the view must also have the same option settings required to create the index. SQL Server generates an error and rolls back any INSERT, UPDATE, or DELETE statement that will affect the result set of the view if the connection executing the statement does not have the proper option settings. For more information, see <u>SET Options That Affect Results</u>.

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.

If you want to add indexes to views in an existing system, you must schema bind any view on which you want to place an index. You can:

- Drop the view and re-create it specifying WITH SCHEMABINDING.
- You can create a second view that has the same text as the existing view but a different name. The optimizer considers the indexes on the new view, even if it is not directly referenced in the FROM clause of queries.

Note Views or tables participating in a view created with the SCHEMABINDING clause cannot be dropped, unless the view is dropped or changed so that it no longer has schema binding. In addition, ALTER TABLE statements on tables that participate in views having schema binding will fail if these statements affect the view definition.

You must ensure that the new view meets all of the requirements of an indexed view. This may require you to change the ownership of the view and all base tables it references so they are all owned by the same user.

See Also

CREATE INDEX SET ANSI_NULLS SET ANSI_PADDING SET ANSI_PADDING SET ANSI_WARNINGS SET ARITHABORT SET CONCAT_NULL_YIELDS_NULL SET NUMERIC_ROUNDABORT SET QUOTED_IDENTIFIER

Creating a Partitioned View

A partitioned view joins horizontally partitioned data from a set of member tables across one or more servers, making the data appear as if from one table. Microsoft® SQL Server[™] 2000 distinguishes between local and distributed partitioned views. In a local partitioned view, all participating tables and the view reside on the same instance of SQL Server. In a distributed partitioned view, at least one of the participating tables resides on a different (remote) server. In addition, SQL Server 2000 differentiates between partitioned views that are updatable and views that are read-only copies of the underlying tables.

Distributed partitioned views can be used to implement a federation of database servers. A federation is a group of servers administered independently, but which cooperate to share the processing load of a system. Forming a federation of database servers by partitioning data is the mechanism that enables you to scale out a set of servers to support the processing requirements of large, multitiered Web sites. For more information, see <u>Designing Federated Database Servers</u>.

Before implementing a partitioned view, you must first partition a table horizontally. The original table is replaced with several smaller member tables. Each member table has the same number of columns as the original table, and each column has the same attributes (such as data type, size, collation) as the corresponding column in the original table. If you are creating a distributed partitioned view, each member table is on a separate member server. For the greatest location transparency, the name of the member databases should be the same on each member server, although this is not a requirement. For example: **Server1.CustomerDB, Server2.CustomerDB, Server3.CustomerDB**.

You design the member tables so that each table stores a horizontal slice of the original table based on a range of key values. The ranges are based on the data values in a partitioning column. The range of values in each member table is enforced by a CHECK constraint on the partitioning column, and ranges cannot overlap. For example, you cannot have one table with a range from 1 through 200000, and another with a range from 150000 through 300000 because it would not be clear which table contains the values from 150000 through 200000.

For example, you are partitioning a **Customer** table into three tables. The CHECK constraint for these tables is:

-- On Server1:

CREATE TABLE Customer_33

(CustomerID INTEGER PRIMARY KEY

CHECK (CustomerID BETWEEN 1 AND 32999),

... -- Additional column definitions)

-- On Server2:

CREATE TABLE Customer 66

(CustomerID INTEGER PRIMARY KEY

CHECK (CustomerID BETWEEN 33000 AND 65999),

... -- Additional column definitions)

-- On Server3: CREATE TABLE Customer_99 (CustomerID INTEGER PRIMARY KEY CHECK (CustomerID BETWEEN 66000 AND 99999), ... -- Additional column definitions)

After creating the member tables, you define a distributed partitioned view on each member server, with each view having the same name. This allows queries referencing the distributed partitioned view name to run on any of the member servers. The system operates as if a copy of the original table is on each member server, but each server has only a member table and a distributed partitioned view. The location of the data is transparent to the application.

You build the distributed partitioned views by:

- Adding linked server definitions on each member server containing the connection information needed to execute distributed queries on the other member servers. This gives a distributed partitioned view access to data on the other servers.
- Setting the **lazy schema validation** option, using **sp_serveroption**, for each linked server definition used in distributed partitioned views. This optimizes performance by ensuring the query processor does not request

meta data for any of the linked tables until data is actually needed from the remote member table.

• Creating a distributed partitioned view on each member server. The views use distributed SELECT statements to access data from the linked member servers, and merges the distributed rows with rows from the local member table.

To create distributed partitioned views for the preceding example, you must:

- Add a linked-server definition named **Server2** with the connection information for **Server2**, and a linked server definition named **Server3** for access to **Server3**.
- Create this distributed partitioned view: CREATE VIEW Customers AS SELECT * FROM CompanyDatabase.TableOwner.Customer UNION ALL SELECT * FROM Server2.CompanyDatabase.TableOwner.C UNION ALL SELECT * FROM Server3.CompanyDatabase.TableOwner.C
- Perform the same steps on **Server2** and **Server3**.

Updatable Partitioned Views

If a local or distributed partitioned view is not updatable, it can serve only as a read-only copy of the original table. An updatable partitioned view can exhibit all the capabilities of the original table.

A view is considered an updatable partitioned view if:

• The view is a set of SELECT statements whose individual result sets are combined into one using the UNION ALL statement. Each individual SELECT statement references one SQL Server base table. The table can be either a local table or a linked table referenced using a four-part name, the OPENROWSET function, or the OPENDATASOURCE

function (you cannot use an OPENDATASOURCE or OPENROWSET function that specifies a pass-through query).

Table Rules

Member tables are defined in the FROM clause in each SELECT statement in the view definition. Each member table must adhere to these rules:

- Member tables cannot be referenced more than once in the view.
- Member tables cannot have indexes created on any computed columns.
- Member tables must have all PRIMARY KEY constraints on an identical number of columns.
- Member tables must have the same ANSI padding setting. For more information about the ANSI padding setting, see <u>SET</u> <u>ANSI_PADDING</u>.

Column Rules

Columns are defined in the select list of each SELECT statement in the view definition. The columns must follow these rules.

- All columns in each member table must be included in the select list.
- The same column cannot be used multiple times in the select list.
- Columns cannot be referenced more than once in the select list.
- The columns must be in the same ordinal position in the select list
- The columns in the select list of each SELECT statement must be of the same type (including data type, precision, scale, and collation). For

```
example, this view definition fails because the first column in both
SELECT statements does not have the same data type:
CREATE VIEW NonUpdatable
AS
SELECT IntPrimaryKey, IntPartNmbr
FROM FirstTable
UNION ALL
SELECT NumericPrimaryKey, IntPartNmbr
FROM SecondTable
```

Partitioning Column Rules

A partitioning column exists on each member table and, through CHECK constraints, identifies the data available in that specific table. Partitioning columns must adhere to these rules:

- Each base table has a partitioning column whose key values are enforced by CHECK constraints. The key ranges of the CHECK constraints in each table do not overlap with the ranges of any other table. Any given value of the partitioning column must map to only one table. The CHECK constraints can only use these operators: BETWEEN, AND, OR, <, <=, >, >=, =.
- The partitioning column must be in the same ordinal location in the select list of each SELECT statement in the view. For example, the partitioning column is always the first column in each select list, or the second column in each select list, and so on.
- Partitioning columns cannot allow nulls.
- Partitioning columns must be a part of the primary key of the table.
- Partitioning columns cannot be computed columns.

• There must be only one constraint on the partitioning column. If there is more than one constraint, SQL Server ignores all the constraints and will not consider them when determining whether or not the view is a partitioned view.

A partitioned column that meets all these rules will support all of the optimizations that are supported by the SQL Server 2000 query optimizer. For more information, see <u>Resolving Distributed Partitioned Views</u>.

Data Modification Rules

In addition to the rules defined for updatable partitioned views, data modification statements referencing the view must adhere to the rules defined for INSERT, UPDATE and DELETE statements.

Note You can modify data through a partitioned view only if you install Microsoft SQL Server 2000 Enterprise Edition or Microsoft. SQL Server 2000 Developer Edition.

INSERT Statements

INSERT statements add data to the member tables through the partitioned view. The INSERT statements must adhere to these rules:

- All columns must be included in the INSERT statement even if the column can be NULL in the base table or has a DEFAULT constraint defined in the base table.
- The DEFAULT keyword cannot be specified in the VALUES clause of the INSERT statement.
- INSERT statements must supply a value that satisfies the logic of the CHECK constraint defined on the partitioning column for one of the member tables.
- INSERT statements are not allowed if a member table contains a column with an identity property.

- INSERT statements are not allowed if a member table contains a **timestamp** column.
- INSERT statements are not allowed if there is a self-join with the same view or any of the member table.

UPDATE Statements

UPDATE statements modify data in one or more of the member tables through the partitioned view. The UPDATE statements must adhere to these rules:

- UPDATE statements cannot specify the DEFAULT keyword as a value in the SET clause even if the column has a DEFAULT value defined in the corresponding member table
- The value of a column with an identity property cannot be changed: however, the other columns can be updated.
- The value of a PRIMARY KEY cannot be changed if the column contains **text, image** or **ntext** data.
- Updates are not allowed if a base table contains a **timestamp** column.
- Updates are not allowed if there is a self-join with the same view or any of the member tables.
- The DEFAULT keyword cannot be specified in the SET clause of the UPDATE statement.

DELETE Statements

DELETE statements remove data in one or more of the member tables through

the partitioned view. The DELETE statements must adhere to this rule:

• DELETE statements are not allowed if there is a self-join with the same view or any of the member tables.

Distributed Partition View Rules

In addition to the rules defined for partitioned views, distributed (remote) partition views have these additional conditions:

- A distributed transaction will be started to ensure atomicity across all nodes affected by the update.
- The XACT_ABORT SET option must be set to ON.
- **smallmoney** and **smalldatetime** columns in remote tables are mapped as **money** and **datetime** respectively. Consequently, the corresponding columns in the local tables should also be **money** and **datetime**.
- Any linked server cannot be a loopback linked server, that is, a linked server that points to the same instance of SQL Server.

A view that references partitioned tables without following all these rules may still be updatable if there is an INSTEAD OF trigger on the view. The query optimizer, however, may not always be able to build execution plans for a view with an INSTEAD OF trigger that are as efficient as the plans for a partitioned view that follows all of the rules.

See Also

<u>CREATE VIEW</u> <u>Designing Partitions</u> <u>Scenarios for Using Views</u> <u>Using Partitioned Views</u> •

Modifying and Renaming a View

After a view is defined, you can change its name or modify its definition without dropping and re-creating the view, thereby losing the permissions associated with the view. When you rename a view, follow these guidelines:

- The view to be renamed must be in the current database.
- The new name must follow the rules for identifiers.
- You can rename only views that you own.
- The database owner can change the name of any user's view.

Altering a view does not affect any dependent objects, such as stored procedures or triggers, unless the definition of the view changes in such a way that the dependent object is no longer valid. For example, a view **authors_view** in the **pubs** database is defined as:

```
CREATE VIEW authors_view
AS
```

SELECT au_id FROM authors

The stored procedure **authors_proc** is defined as:

CREATE PROC authors_proc AS

SELECT au_id from authors_view

authors_view is modified to retrieve the column au_lname instead of au_id:

ALTER VIEW authors_view

AS

SELECT au_lname FROM authors

authors_proc now fails when executed because the column **au_id** no longer exists in the view.

You can also modify a view to encrypt its definition, or to ensure that all data modification statements executed against the view adhere to the criteria set within the SELECT statement defining the view. For more information, see <u>Creating a View</u>.

To modify a view

∃ <u>Transact-SQL</u>

Modifying Data Through a View

You can modify data through a view in these ways:

- Use INSTEAD OF triggers with logic to support INSERT, UPDATE and DELETE statements.
- Use updatable partitioned views that modify one or more member tables.

If a view does not use an INSTEAD OF trigger or is not an updatable partitioned view, it can still be updatable provided that:

If a view does not use an INSTEAD OF trigger or is not an updatable partitioned view, it can still be updatable provided that:

- The view contains at least one table in the FROM clause of the view definition; the view cannot be based solely on an expression.
- No aggregate functions (AVG, COUNT, SUM, MIN, MAX, GROUPING, STDEV, STDEVP, VAR, VARP) or GROUP BY, UNION, DISTINCT, or TOP clauses are used in the select list. However, aggregate functions can be used within a subquery defined in the FROM clause provided that the derived values generated by the aggregate functions are not modified.

Note Partitioned views using the UNION ALL operator can be updatable.

• No derived columns are used in the select list. Derived columns are result set columns formed by anything other than a simple column reference.

Guidelines for Modifying Data Through a View

Before you modify data through a view without using an INSTEAD OF trigger or an updatable partitioned view, consider these guidelines:

- All data modification statements executed against the view must adhere to the criteria set within the SELECT statement defining the view if the WITH CHECK OPTION clause is used in the definition of the view. If the WITH CHECK OPTION clause is used, rows cannot be modified in a way that causes them to disappear from the view. Any modification that would cause this to happen is canceled and an error is displayed.
- SQL Server must be able to resolve unambiguously the modification operation to specific rows in one of the base tables referenced by the view. You cannot use data modification statements on more than one underlying table in a single statement. Therefore, the columns listed in the UPDATE or INSERT statement must belong to a single base table within the view definition.
- All the columns in the underlying table that are being updated and do not allow null values have values specified in either the INSERT statement or DEFAULT definitions. This ensures that all the columns in the underlying table that require values have them.
- The data modified in the columns in the underlying table must adhere to the restrictions on those columns, such as nullability, constraints, DEFAULT definitions and so on. For example, if a row is deleted, all the underlying FOREIGN KEY constraints in related tables must still be satisfied for the delete to succeed.
- A distributed partition view (remote view) cannot be updated using a keyset-driven cursor. This restriction can be resolved by declaring the cursor on the underlying tables and not on the view itself.

Additionally, to delete data in a view:

• Only one table can be listed in the FROM clause of the view definition.

The READTEXT and WRITETEXT statements cannot be used with **text**, **ntext**, or **image** columns in a view.

To add data through a view

⊞ <u>Transact-SQL</u>

To change data through a view

⊞ <u>Transact-SQL</u>

To delete data through a view

∃ <u>Transact-SQL</u>

Getting Information About a View

You can gain information about the definition of a view if it is not encrypted. You may need to see the definition of the view to understand how its data is derived from the source tables or to see the data defined by the view.

Views are queried the same way that ordinary tables are queried. However, any table hints used when querying the view are ignored. For more information about table hints, see <u>SELECT</u>.

If you change the name of an object referenced by a view, you must modify the view so that its text reflects the new name. Therefore, before renaming an object, display the dependencies of the object first to determine if any views are affected by the proposed change.

To get information about a view

∃ <u>Transact-SQL</u>

Deleting a View

After a view has been created, you can delete the view if it is not needed, or if you want to clear the view definition and the permissions associated with it. When a view is deleted, the tables and the data upon which it is based are not affected. Any queries that use objects that depend on the deleted view fail when they are next executed, unless a view with the same name is created. However, if the new view does not reference objects expected by any objects dependent on the new view, queries using the dependent objects fail when executed. For example, a view **my_view** that retrieves all columns from the **authors** table in the **pubs** database is deleted and replaced by a new view called **my_view** that retrieves all columns from the **titles** table instead. Any stored procedures that reference columns from the underlying **authors** table in **my_view** now fail because those columns are replaced by columns from the **titles** table instead.

To delete a view

∃ <u>Transact-SQL</u>

Stored Procedures

When you create an application with Microsoft® SQL Server[™] 2000, the Transact-SQL programming language is the primary programming interface between your applications and the SQL Server database. When you use Transact-SQL programs, two methods are available for storing and executing the programs. You can store the programs locally and create applications that send the commands to SQL Server and process the results, or you can store the programs as <u>stored procedures</u> in SQL Server and create applications that execute the stored procedures and process the results.

Stored procedures in SQL Server are similar to procedures in other programming languages in that they can:

- Accept input parameters and return multiple values in the form of output parameters to the calling procedure or batch.
- Contain programming statements that perform operations in the database, including calling other procedures.
- Return a status value to a calling procedure or batch to indicate success or failure (and the reason for failure).

You can use the Transact-SQL EXECUTE statement to run a stored procedure. Stored procedures are different from <u>functions</u> in that they do not return values in place of their names and they cannot be used directly in an expression.

The benefits of using stored procedures in SQL Server rather than Transact-SQL programs stored locally on client computers are:

• They allow modular programming.

You can create the procedure once, store it in the database, and call it any number of times in your program. Stored procedures can be created by a person who specializes in database programming, and they can be modified independently of the program source code.

• They allow faster execution.

If the operation requires a large amount of Transact-SQL code or is performed repetitively, stored procedures can be faster than batches of Transact-SQL code. They are parsed and optimized when they are created, and an in-memory version of the procedure can be used after the procedure is executed the first time. Transact-SQL statements repeatedly sent from the client each time they run are compiled and optimized every time they are executed by SQL Server.

• They can reduce network traffic.

An operation requiring hundreds of lines of Transact-SQL code can be performed through a single statement that executes the code in a procedure, rather than by sending hundreds of lines of code over the network.

• They can be used as a security mechanism.

Users can be granted permission to execute a stored procedure even if they do not have permission to execute the procedure's statements directly.

A SQL Server stored procedure is created with the Transact-SQL CREATE PROCEDURE statement and can be modified with the ALTER PROCEDURE statement. The stored procedure definition contains two primary components: the specification of the procedure name and its parameters, and the body of the procedure, which contains Transact-SQL statements that perform the procedure's operations.

See Also

Catalog Stored Procedures

System Stored Procedures

Extended Stored Procedures

Extended stored procedures allow you to create your own external routines in a programming language such as C. The extended stored procedures appear to users as normal stored procedures and are executed in the same way. Parameters can be passed to extended stored procedures, and they can return results and return status. Extended stored procedures can be used to extend the capabilities of Microsoft® SQL ServerTM 2000.

Extended stored procedures are dynamic-link libraries (DLLs) that SQL Server can dynamically load and execute. Extended stored procedures run directly in the address space of SQL Server and are programmed using the SQL Server Open Data Services API.

After an extended stored procedure has been written, members of the **sysadmin** fixed server role can register the extended stored procedure with SQL Server and then grant permission to other users to execute the procedure. Extended stored procedures can be added only to the **master** database.

Note Extended stored procedures may produce memory leaks or other problems that reduce the performance and reliability of the server. You should consider storing extended stored procedures in an instance of SQL Server separate from the instance containing the referenced data and using distributed queries to access the database. For more information, see <u>Distributed Queries</u>.

To add an extended stored procedure

⊞ <u>Transact-SQL</u>

Creating a Stored Procedure

You can create stored procedures using the CREATE PROCEDURE Transact-SQL statement. Before creating a stored procedure, consider that:

- CREATE PROCEDURE statements cannot be combined with other SQL statements in a single batch.
- Permission to create stored procedures defaults to the database owner, who can transfer it to other users.
- Stored procedures are database objects, and their names must follow the rules for identifiers.
- You can create a stored procedure only in the current database.

When creating a stored procedure, you should specify:

- Any input parameters and output parameters to the calling procedure or batch.
- The programming statements that perform operations in the database, including calling other procedures.
- The status value returned to the calling procedure or batch to indicate success or failure (and the reason for failure).

System Stored Procedures

Many of your administrative activities in Microsoft® SQL Server[™] 2000 are performed through a special kind of procedure known as a <u>system stored procedure</u>. System stored procedures are created and stored in the **master** database and have the **sp**_ prefix. System stored procedures can be executed from any database without having to qualify the stored procedure name

fully using the database name **master**.

It is strongly recommended that you do not create any stored procedures using **sp_** as a prefix. SQL Server always looks for a stored procedure beginning with **sp_** in this order:

- 1. The stored procedure in the **master** database.
- 2. The stored procedure based on any qualifiers provided (database name or owner).
- 3. The stored procedure using **dbo** as the owner, if one is not specified.

Therefore, although the user-created stored procedure prefixed with **sp_** may exist in the current database, the **master** database is always checked first, even if the stored procedure is qualified with the database name.

IMPORTANT If any user-created stored procedure has the same name as a system stored procedure, the user-created stored procedure will never be executed.

Grouping

A procedure can be created with the same name as an existing stored procedure if it is given a different identification number, which allows the procedures to be grouped logically. Grouping procedures with the same name allows them to be deleted at the same time. Procedures used in the same application are often grouped this way. For example, the procedures used with the **my_app** application might be named **my_proc;1**, **my_proc;2**, and so on. Deleting **my_proc** deletes the entire group. After procedures have been grouped, individual procedures within the group cannot be deleted.

Temporary Stored Procedures

Private and global temporary stored procedures, analogous to temporary tables, can be created with the # and ## prefixes added to the procedure name. # denotes a local temporary stored procedure; ## denotes a global temporary stored procedure. These procedures do not exist after SQL Server is shut down.

Temporary stored procedures are useful when connecting to earlier versions of

SQL Server that do not support the reuse of execution plans for Transact-SQL statements or batches. Applications connecting to SQL Server version 2000 should use the **sp_executesql** system stored procedure instead of temporary stored procedures. For more information, see <u>Execution Plan Caching and Reuse</u>.

Only the connection that created a local temporary procedure can execute it, and the procedure is automatically deleted when the connection is closed (when the user logs out of SQL Server).

Any connection can execute a global temporary stored procedure. A global temporary stored procedure exists until the connection used by the user who created the procedure is closed and any currently executing versions of the procedure by any other connections are completed. Once the connection that was used to create the procedure is closed, no further execution of the global temporary stored procedure is allowed. Only those connections that have already started executing the stored procedure are allowed to complete.

If a stored procedure not prefixed with # or ## is created directly in the **tempdb** database, the stored procedure is automatically deleted when SQL Server is shut down because **tempdb** is re-created every time SQL Server is started. Procedures created directly in **tempdb** exist even after the creating connection is terminated. As with any other object, permissions to execute the temporary stored procedure can be granted, denied, and revoked to other users.

To create a stored procedure

⊞ <u>Transact-SQL</u>

Specifying Parameters

A stored procedure communicates with the calling program through its parameters. When a program executes a stored procedure, it can pass values to the stored procedure through the parameters of the stored procedure. These values can be used as standard variables in the Transact-SQL programming language. The stored procedure can also return values to the calling program through OUTPUT parameters. A stored procedure can have as many as 2100 parameters, with each parameter having a name, data type, direction, and default value.

See Also

Parameters

Returning Data Using OUTPUT Parameters

Specifying a Name

Each stored procedure parameter must be defined with a unique name. Stored procedure names must begin with a single @ character, as in a standard Transact-SQL variable, and must follow the rules for object identifiers. The parameter name can be used in the stored procedure to obtain and change the value of the parameter.

Values can be passed to stored procedures either by explicitly naming the parameters and assigning the appropriate value or by supplying the parameter values given in the CREATE PROCEDURE statement without naming them. For example, if the stored procedure **my_proc** expects three parameters named *@***first**, *@***second**, and *@***third**, the values passed to the stored procedure can be assigned to the parameter names, such as:

EXECUTE my_proc @second = 2, @first = 1, @third = 3

Or by position without naming them:

```
EXECUTE my_proc 1, 2, 3
```

Naming the parameters when executing the stored procedure allows the parameters to be supplied in any order. If the parameters are not named, they must be supplied in the same order (left to right) as they are defined in the stored procedure. Additionally, all parameters preceding a given parameter must be supplied even if they are optional and have default values. For example, if the parameters of **my_proc** are all optional, **my_proc** could be executed by supplying values only for the first and second parameters, but not by supplying values only for the second and third parameters. This is necessary because, otherwise, Microsoft® SQL ServerTM 2000 cannot identify the parameters that are being specified.

See Also

EXECUTE

<u>Using Identifiers</u> <u>Specifying a Default Value</u>

Specifying a Data Type

Parameters in a stored procedure are defined with a data type, much as a column in a table is defined. A stored procedure parameter can be defined with any of the Microsoft® SQL Server[™] 2000 data types, including **text** and **image**. Stored procedure parameters can also be defined with user-defined data types.

Note The **cursor** data type can be used only as an OUTPUT parameter to a stored procedure. For more information about using cursor variables, see <u>Scope of Transact-SQL Cursor Names</u>.

The data type of a parameter determines the type and range of values that are accepted for the parameter. For example, if you define a parameter with a **tinyint** data type, only numeric values ranging from 0 to 255 are accepted. An error is returned if a stored procedure is executed with a value incompatible with the data type.

See Also

Creating User-Defined Data Types CREATE PROCEDURE

Data Types

Specifying the Direction of a Parameter

All procedure parameters can receive input values when the stored procedure is executed by the program that calls the stored procedure.

Examples

The following stored procedure, **get_sales_for_title**, uses an input parameter. The **@title** parameter in the stored procedure receives the input value of a title of a book specified by the calling program. The SELECT statement uses the **@title** parameter to obtain the correct **ytd_sales** value and displays the value.

```
CREATE PROCEDURE get_sales_for_title
@title varchar(80) -- This is the input parameter.
AS
```

```
-- Get the sales for the specified title.
SELECT "YTD_SALES" = ytd_sales
FROM titles
WHERE title = @title
```

RETURN GO

If you specify the OUTPUT keyword for a parameter in the stored procedure definition, the stored procedure can return the current value of the parameter to the calling program when the stored procedure exits. The calling program must also use the OUTPUT keyword when executing the stored procedure to save the parameter's value in a variable that can be used in the calling program. For more information, see <u>Returning Data Using OUTPUT Parameters</u>.

Specifying a Default Value

You can create a stored procedure with optional parameters by specifying a default value for optional parameters. When the stored procedure is executed, the default value is used if no other value has been specified.

Specifying default values is necessary because a system error is returned if a parameter does not have a default value specified in the stored procedure and the calling program does not provide a value for the parameter when the stored procedure is executed.

If no value can be specified appropriately as a default for the parameter, you can specify NULL as the default for a parameter and have the stored procedure return a customized message if the stored procedure is executed without a value for the parameter.

Note If the default value is a character string that contains embedded blanks or punctuation, or if it begins with a number (for example, 6xxx), it must be enclosed in single, straight quotation marks.

Examples

This example shows the **get_sales_for_title** procedure with special handling for cases when the stored procedure is executed without a value for the **@title** parameter:

```
CREATE PROCEDURE get_sales_for_title
@title varchar(80) = NULL, -- NULL default value
@ytd_sales int OUTPUT
AS
```

```
-- Validate the @title parameter.
IF @title IS NULL
BEGIN
PRINT 'ERROR: You must specify a title value.'
RETURN
```

END

```
-- Get the sales for the specified title and

-- assign it to the output parameter.

SELECT @ytd_sales = ytd_sales

FROM titles

WHERE title = @title
```

RETURN GO

The following example shows the **my_proc** procedure with default values for each of the three parameters **@first**, **@second**, and **@third**, and the values displayed when the stored procedure is executed with other parameter values:

```
CREATE PROCEDURE my_proc
@first int = NULL, -- NULL default value
@second int = 2, -- Default value of 2
@third int = 3 -- Default value of 3
AS
```

```
-- Display values.
SELECT @first, @second, @third
GO
```

```
EXECUTE my_proc -- No parameters supplied GO
```

Displays:

NULL 2 3

EXECUTE my_proc 10, 20, 30 -- All parameters supplied GO

Displays:

10 20 30

EXECUTE my_proc @second = 500 -- Only second parameter suppli GO

Displays:

NULL 500 3

EXECUTE my_proc 40, @third = 50 -- Only first and third parameters GO -- are supplied.

Displays:

40 2 50

See Also

EXECUTE

Programming Stored Procedures

Almost any Transact-SQL code that can be written as a batch can be used to create a stored procedure.

Rules for Programming Stored Procedures

Rules for programming stored procedures include:

• The CREATE PROCEDURE definition itself can include any number and type of SQL statements except for the following CREATE statements, which cannot be used anywhere within a stored procedure:

CREATE DEFAULT	CREATE TRIGGER
CREATE PROCEDURE	CREATE VIEW
CREATE RULE	

- Other database objects can be created within a stored procedure. You can reference an object created in the same stored procedure as long as it is created before it is referenced.
- You can reference temporary tables within a stored procedure.
- If you create a local temporary table inside a stored procedure, the temporary table exists only for the purposes of the stored procedure; it disappears when you exit the stored procedure.
- If you execute a stored procedure that calls another stored procedure, the called stored procedure can access all objects created by the first stored procedure, including temporary tables.
- If you execute a remote stored procedure that makes changes on a remote instance of Microsoft® SQL Server[™] 2000, those changes

cannot be rolled back. Remote stored procedures do not take part in transactions.

- The maximum number of parameters in a stored procedure is 2100.
- The maximum number of local variables in a stored procedure is limited only by available memory.
- Depending on available memory, the maximum size of a stored procedure is 128 megabytes (MB).

For more information about the rules for creating stored procedures, see <u>CREATE PROCEDURE</u>.

Qualifying Names Inside Stored Procedures

Inside a stored procedure, object names used with statements (for example, SELECT or INSERT) that are not user-qualified default to the owner of the stored procedure. If a user who creates a stored procedure does not qualify the name of the tables referenced in SELECT, INSERT, UPDATE, or DELETE statements within the stored procedure, access to those tables through the stored procedure is restricted by default to the creator of the procedure.

Object names used with the statements ALTER TABLE, CREATE TABLE, DROP TABLE, TRUNCATE TABLE, CREATE INDEX, DROP INDEX, UPDATE STATISTICS, and DBCC must be qualified with the name of the object owner if other users are to use of the stored procedure. For example, Mary, who owns table **marytab**, must qualify the name of her table when it is used with one of these statements if she wants other users to be able to execute the stored procedure in which the table is used.

This rule is necessary because object names are resolved when the stored procedure is run. If **marytab** is not qualified and John tries to execute the procedure, SQL Server looks for a table called **marytab** owned by John.

Encrypting Procedure Definitions

If you are creating a stored procedure and you want to make sure that the procedure definition cannot be viewed by other users, you can use the WITH ENCRYPTION clause. The procedure definition is then stored in an unreadable form.

After a stored procedure is encrypted, its definition cannot be decrypted and cannot be viewed by anyone, including the owner of the stored procedure or the system administrator.

SET Statement Options

When an ODBC application connects to SQL Server, the server automatically sets these options for the session:

- SET QUOTED_IDENTIFIER ON
- SET TEXTSIZE 2147483647
- SET ANSI_DEFAULTS ON
- SET CURSOR_CLOSE_ON_COMMIT OFF
- SET IMPLICIT_TRANSACTIONS OFF

These settings increase the portability of ODBC applications. Because DB-Library-based applications generally do not set these options, stored procedures should be tested with the SET options listed above turned both on and off. This ensures that the stored procedures work correctly regardless of the options a particular connection may have set when it invokes the stored procedure. A stored procedure that requires a particular setting for one of these options should issue a SET statement at the start of the stored procedure. This SET statement remains in effect only for the execution of the stored procedure; when the stored procedure ends, the original setting is restored.

Examples

A. Create a stored procedure that uses parameters

This example creates a stored procedure that is useful in the **pubs** database. Given the last and first name of an author, the stored procedure displays the title and publisher of each book by that author.

```
CREATE PROC au_info @lastname varchar(40), @firstname varchar(
AS
SELECT au_lname, au_fname, title, pub_name
FROM authors INNER JOIN titleauthor ON authors.au_id = titleautho
JOIN titles ON titleauthor.title_id = titles.title_id
JOIN publishers ON titles.pub_id = publishers.pub_id
WHERE au_fname = @firstname
AND au_lname = @lastname
GO
```

When a message appears stating that the command did not return any data and it did not return any rows, the stored procedure has been created.

Now execute the **au_info** stored procedure:

```
EXECUTE au_info Ringer, Anne GO
```

Here is the result set:

au_lname	au_fname	title	pub_name
Ringer	Anne	The Gourmet	Binnet & Hardley
		Microwave	
Ringer	Anne	Is Anger the Enemy?	New Moon Books

(2 row(s) affected)

B. Create a stored procedure that uses default values for parameters

This example creates a stored procedure, **pub_info2**, that displays the names of all authors who have written a book published by the publisher given as a parameter. If no publisher name is supplied, the stored procedure shows the authors published by Algodata Infosystems.

CREATE PROC pub_info2 @pubname varchar(40) = 'Algodata Infosy AS SELECT au_lname, au_fname, pub_name FROM authors a INNER JOIN titleauthor ta ON a.au_id = ta.au_id JOIN titles t ON ta.title_id = t.title_id JOIN publishers p ON t.pub_id = p.pub_id

WHERE @pubname = p.pub_name

Execute **pub_info2** with no parameter specified:

```
EXECUTE pub_info2
GO
```

Here is the result set:

au_lname	au_fname	pub_name
Green	Marjorie	Algodata Infosystems
Bennet	Abraham	Algodata Infosystems
O'Leary	Michael	Algodata Infosystems
MacFeather	Stearns	Algodata Infosystems
Straight	Dean	Algodata Infosystems
Carson	Cheryl	Algodata Infosystems
Dull	Ann	Algodata Infosystems
Hunter	Sheryl	Algodata Infosystems
Locksley	Charlene	Algodata Infosystems

(9 row(s) affected)

C. Execute a stored procedure that overrides the default value of a parameter with an explicit value

In this example, the stored procedure, **showind2**, the default value for the *@***table** parameter is **titles**.

CREATE PROC showind2 @table varchar(30) = 'titles' AS SELECT TABLE_NAME = sysobjects.name, INDEX_NAME = sysindexes.name, INDEX_ID = indid FROM sysindexes INNER JOIN sysobjects ON sysobjects.id = sysind WHERE sysobjects.name = @table

The column headings (for example, **TABLE_NAME**) make the results more readable. Here is what the stored procedure shows for the **authors** table:

```
EXECUTE showind2 authors GO
```

TABLE_NAME	INDEX_NAME	INDEX_ID
authors	UPKCL_auidind	1
authors	aunmind	2

(2 row(s) affected)

If you do not supply a value, SQL Server uses the default table, **titles**:

EXECUTE showind2 GO

Here is the result set:

TABLE_NAME	INDEX_NAME	INDEX_ID

titles	UPKCL_titleidind	1
titles	titleind	2

(2 row(s) affected)

D. Create a stored procedure using a parameter default of NULL

The parameter default can be the value NULL. In this case, if you do not supply a parameter, SQL Server executes the stored procedure according to its other statements. No error message is displayed.

The procedure definition can also specify that some other action be taken if you do not give a parameter. For example:

```
CREATE PROC showind3 @table varchar(30) = NULL
AS IF @table IS NULL
PRINT 'Give a table name'
ELSE
SELECT TABLE_NAME = sysobjects.name,
INDEX_NAME = sysindexes.name, INDEX_ID = indid
FROM sysindexes INNER JOIN sysobjects
ON sysobjects.id = sysindexes.id
WHERE sysobjects.name = @table
```

E. Create a stored procedure using a parameter default including wildcard characters

The default can include wildcard characters (%, _, [] and [^]) if the stored procedure uses the parameter with the LIKE keyword. For example, **showind** can be modified to display information about the system tables if you do not supply a parameter:

```
CREATE PROC showind4 @table varchar(30) = 'sys%'
AS SELECT TABLE_NAME = sysobjects.name,
INDEX_NAME = sysindexes.name, INDEX_ID = indid
FROM sysindexes INNER JOIN sysobjects
```

ON sysobjects.id = sysindexes.id WHERE sysobjects.name LIKE @table

The following variation of the stored procedure **au_info** has defaults with wildcard characters for both parameters:

```
CREATE PROC au_info2 @lastname varchar(30) = 'D%',
  @firstname varchar(18) = '%'
AS
SELECT au_lname, au_fname, title, pub_name
FROM authors INNER JOIN titleauthor ON authors.au_id = titleautho
JOIN titles ON titleauthor.title_id = titles.title_id
JOIN publishers ON titles.pub_id = publishers.pub_id
WHERE au_fname LIKE @firstname
AND au_lname LIKE @lastname
```

If **au_info2** is executed with no parameters, all the authors with last names beginning with the letter D are displayed:

EXECUTE au_info2 GO

Here is the result set:

au_lname	au_fname	title	pub_name
Dull	Ann	Secrets of Silicon Val	Algodata
			Infosystems
del Castillo	Innes	Silicon Val Gastrono	Binnet & Hardley
DeFrance	Michel	The Gourmet	Binnet & Hardley
		Microwave	

(3 row(s) affected)

This example omits the second parameter when defaults for two parameters have been defined, so you can find the books and publishers for all authors with the last name **Ringer**:

EXECUTE au_info2 Ringer GO

au_lname	au_fname	title	pub_name
Ringer	Anne	The Gourmet	Binnet & Hardley
		Microwave	
Ringer	Anne	Is Anger the Enemy?	New Moon Books
Ringer	Albert	Is Anger the Enemy?	New Moon Books
Ringer	Albert	Life Without Fear	New Moon Books

(4 row(s) affected)

See Also

CREATE PROCEDURE

EXECUTE

Effects of SQL-92 Options

Rollbacks in Stored Procedures and Triggers

Nesting Stored Procedures

Stored procedures are nested when one stored procedure calls another. You can nest stored procedures up to 32 levels. The nesting level increases by one when the called stored procedure begins execution and decreases by one when the called stored procedure completes execution. Attempting to exceed the maximum of 32 levels of nesting causes the whole calling stored procedure chain to fail. The current nesting level for the stored procedures in execution is stored in the @@NESTLEVEL function.

Although the nesting limit is 32 levels, Microsoft® SQL Server[™] 2000 has no limit on the number of stored procedures that can be invoked from a given stored procedure, provided that the subordinate stored procedures do not invoke other subordinate stored procedures and the maximum nesting level is never exceeded.

An error in a nested stored procedure is not necessarily fatal to the calling stored procedure. When invoking stored procedures within stored procedures, use the Transact-SQL RETURN statement to return a return code and check the return code from the calling stored procedure. In this way, you can specify the behavior of your stored procedures when errors occur. For more information about using return codes, see <u>Returning Data Using a Return Code</u>.

Stored procedures can even do a nested call to themselves, a technique known as recursion.

See Also

@@NESTLEVEL

Deferred Name Resolution and Compilation

When a stored procedure is created, the statements in the procedure are parsed for syntactical accuracy. If a syntactical error is encountered in the procedure definition, an error is returned and the stored procedure is not created. If the statements are syntactically correct, the text of the stored procedure is stored in the **syscomments** system table.

When a stored procedure is executed for the first time, the query processor reads the text of the stored procedure from the **syscomments** system table of the procedure and checks that the names of the objects used by the procedure are present. This process is called deferred name resolution because objects referenced by the stored procedure need not exist when the stored procedure is created, but only when it is executed.

In the resolution stage, Microsoft® SQL Server[™] 2000 also performs other validation activities (for example, checking the compatibility of a column data type with variables). If the objects referenced by the stored procedure are missing when the stored procedure is executed, the stored procedure stops executing when it gets to the statement that references the missing object. In this case, or if other errors are found in the resolution stage, an error is returned.

Note If an object referenced by a stored procedure is deleted or renamed, then an error is returned when the stored procedure is executed. However, if an object referenced in a stored procedure is replaced with an object of the same name, the stored procedure executes without having to be recompiled. For example, if stored procedure **proc1** references table **test1**, and **test1** is deleted and a different table called **test1** is created, **proc1** references the new table. The stored procedure does not have to be recompiled.

If procedure execution successfully passes the resolution stage, the SQL Server query optimizer analyzes the Transact-SQL statements in the stored procedure and creates an execution plan. The execution plan describes the fastest method of executing the stored procedure, based on information such as:

• The amount of data in the tables.

- The nature and presence of indexes on the tables and the distribution of data in the indexed columns.
- The comparison operators and comparison values used in WHERE clause conditions.
- The presence of joins and UNION, GROUP BY, and ORDER BY keywords.

After the query optimizer has analyzed these factors in the stored procedure, it places the execution plan in memory. The process of analyzing the stored procedure and creating an execution plan is called compilation. The optimized in-memory execution plan is used to execute the query. The execution plan stays in memory until SQL Server is restarted, or until space is needed for storage of another object.

When the stored procedure is subsequently executed, SQL Server reuses the existing execution plan if it is still in memory. If the execution plan is no longer in memory, a new execution plan is created. For more information, see <u>Stored</u> <u>Procedure and Trigger Execution</u>.

See Also

CREATE PROCEDURE

Execution Plan Caching and Reuse

Returning Data from a Stored Procedure

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

- 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). For more information about using cursor variables, see <u>Scope of Transact-SQL Cursor Names</u>.
- 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. For more information about using the SELECT statement, see <u>Query</u> <u>Fundamentals</u>.
- A global cursor that can be referenced outside the stored procedure. For more information about using cursor variables, see <u>Scope of Transact-SQL Cursor Names</u>.

Returning Data Using OUTPUT Parameters

If you specify the OUTPUT keyword for a parameter in the procedure definition, the stored procedure can return the current value of the parameter to the calling program when the stored procedure exits. To save the value of the parameter in a variable that can be used in the calling program, the calling program must use the OUTPUT keyword when executing the stored procedure.

Examples

The following example shows a stored procedure with an input and an output parameter. The first parameter in the stored procedure **@title** receives the input value specified by the calling program, and the second parameter **@ytd_sales** is used to return the value to the calling program. The SELECT statement uses the **@title** parameter to obtain the correct **ytd_sales** value, and assigns the value to the **@ytd_sales** output parameter.

```
CREATE PROCEDURE get_sales_for_title
@title varchar(80), -- This is the input parameter.
@ytd_sales int OUTPUT -- This is the output parameter.
AS
```

```
-- Get the sales for the specified title and

-- assign it to the output parameter.

SELECT @ytd_sales = ytd_sales

FROM titles

WHERE title = @title
```

RETURN GO

The following program executes the stored procedure with a value for the input parameter and saves the output value of the stored procedure in the **@ytd_sales_for_title** variable local to the calling program.

-- Declare the variable to receive the output value of the procedure. DECLARE @ytd_sales_for_title int

-- Execute the procedure with a title_id value

-- and save the output value in a variable.

EXECUTE get_sales_for_title "Sushi, Anyone?", @ytd_sales = @ytd_sales_for_title OUTPUT

-- Display the value returned by the procedure. PRINT 'Sales for "Sushi, Anyone?": ' + convert(varchar(6),@ytd_sal GO

Sales for "Sushi, Anyone?": 4095

Input values can also be specified for OUTPUT parameters when the stored procedure is executed. This allows the stored procedure to receive a value from the calling program, change it or perform operations with it, then return the new value to the calling program. In the earlier example, the **@ytd_sales_for_title** variable can be assigned a value prior to executing the stored procedure. The **@ytd_sales** variable contains the value of the parameter in the body of the stored procedure, and the value of the **@ytd_sales** variable is returned to the calling program when the stored procedure exits. This is often referred to as "pass-by-reference capability."

If you specify OUTPUT for a parameter when you execute a stored procedure and the parameter is not defined using OUTPUT in the stored procedure, you get an error message. You can execute a stored procedure with OUTPUT parameters and not specify OUTPUT when executing the stored procedure. No error is returned, but you cannot use the output value in the calling program.

See Also

EXECUTE

Scope of Transact-SQL Cursor Names

Returning Data Using a Return Code

A stored procedure can return an integer value called a return code to indicate the execution status of a procedure. You specify the return code for a stored procedure using the RETURN statement. As with OUTPUT parameters, you must save the return code in a variable when the stored procedure is executed to use the return code value in the calling program. For example, the assignment variable **@result** of data type **int** is used to store the return code from the stored procedure **my_proc**:

```
DECLARE @result int
EXECUTE @result = my_proc
```

Return codes are commonly used in control-of-flow blocks within stored procedures to set the return code value for each possible error situation. You can use the @@ERROR function after a Transact-SQL statement to detect if an error occurred during the execution of the statement.

Examples

A. Return a different return code depending on the type of error

This example shows the **get_sales_for_title** procedure with special handling that sets special return code values for various errors. The table shows the integer value assigned by the stored procedure to each possible error.

Value	Meaning
0	Successful execution.
1	Required parameter value not specified.
2	Invalid parameter value specified.
3	Error occurred getting sales value.
4	NULL sales value found for the title.

CREATE PROCEDURE get_sales_for_title

-- This is the input parameter, with a default.

```
@title varchar(80) = NULL,
-- This is the output parameter.
@ytd_sales int OUTPUT
AS
-- Validate the @title parameter.
IF @title IS NULL
BEGIN
 PRINT "ERROR: You must specify a title value."
 RETURN(1)
END
ELSE
BEGIN
 -- Make sure the title is valid.
 IF (SELECT COUNT(*) FROM titles
   WHERE title = @title) = 0
   RETURN(2)
END
-- Get the sales for the specified title and
-- assign it to the output parameter.
SELECT @ytd_sales = ytd_sales
FROM titles
WHERE title = @title
-- Check for SQL Server errors.
IF @@ERROR <> 0
BEGIN
 RETURN(3)
END
```

ELSE

BEGIN

-- Check to see if the ytd_sales value is NULL.

```
IF @ytd_sales IS NULL
RETURN(4)
ELSE
-- SUCCESS!!
RETURN(0)
END
```

GO

Using return codes in this manner allows your calling programs to detect and handle the errors that occur when the stored procedure is executed.

B. Handle the different return codes returned from a stored procedure

This example creates a program to handle the return codes returned from the **get_sales_for_title** procedure.

-- Declare the variables to receive the output value and return code -- of the procedure.

DECLARE @ytd_sales_for_title int, @ret_code INT

```
-- Execute the procedure with a title_id value
-- and save the output value and return code in variables.
EXECUTE @ret_code = get_sales_for_title
"Sushi, Anyone?",
@ytd_sales = @ytd_sales_for_title OUTPUT
```

-- Check the return codes.

```
IF @ret_code = 0
```

BEGIN

PRINT "Procedure executed successfully"

-- Display the value returned by the procedure.

PRINT 'Sales for "Sushi, Anyone?": ' + CONVERT(varchar(6),@ytd END

```
ELSE IF @ret_code = 1

PRINT "ERROR: No title_id was specified."

ELSE IF @ret_code = 2

PRINT "ERROR: An invalid title_id was specified."

ELSE IF @ret_code = 3

PRINT "ERROR: An error occurred getting the ytd_sales."
```

GO

Executing a Stored Procedure

When you have to execute a stored procedure, use the Transact-SQL EXECUTE statement. You can execute a stored procedure without using the EXECUTE keyword if the stored procedure is the first statement in the batch.

Parameter values can be supplied if a stored procedure is written to accept them.

Note If you supply parameters in the form *@parameter = value*, you can supply them in any order. You can also omit parameters for which defaults have been supplied. If you supply one parameter in the form *@parameter = value*, you must supply all subsequent parameters this way. If you do not supply parameters in the form *@parameter = value*, you must supply them in the order given in the CREATE PROCEDURE statement.

When executing a stored procedure, the server rejects any parameters that were not included with the parameter list during procedure creation. Any parameter passed by reference (explicitly passing the parameter name) is not accepted if the parameter name does not match.

Although you can omit parameters for which defaults have been supplied, you can only truncate the list of parameters. For example, if a stored procedure has five parameters, you can omit both the fourth and the fifth parameters, but you cannot skip the fourth and still include the fifth unless you supply parameters in the form *@parameter = value*.

The default value of a parameter, if defined for the parameter in the stored procedure, is used when:

- No value for the parameter is specified when the stored procedure is executed.
- The DEFAULT keyword is specified as the value for the parameter.

To execute a stored procedure that is grouped with other stored procedures of the same name, specify the identification number of the stored procedure within the group. For example, to execute the second stored procedure in the group **my_proc**, execute:

EXECUTE my_proc;2

To execute a stored procedure

⊞ <u>Transact-SQL</u>

See Also

Batches

Execution Characteristics of Extended Stored Procedures

SQL Profiler Scenarios

Specifying Parameters

Debugging an Extended Stored Procedure

SQL Stored Procedures

Returning Data Using a Return Code

Recompiling a Stored Procedure

Automatic Execution of Stored Procedures

When you mark stored procedures for automatic execution, these stored procedures are executed every time Microsoft® SQL ServerTM 2000 starts.

This is useful if you have operations that you want to perform regularly, or if you have a stored procedure that runs as a background process and is expected to be running at all times. Another use for automatic execution of stored procedures is to have the stored procedure perform system or maintenance tasks in **tempdb**, such as creating a global temporary table. This ensures that such a temporary table will always exist when **tempdb** is re-created as SQL Server starts.

A stored procedure that is automatically executed operates with the same permissions as members of the **sysadmin** fixed server role. Any error messages generated by the stored procedure are written to the SQL Server error log. Do not return any result sets from a stored procedure that is executed automatically. Because the stored procedure is being executed by SQL Server rather than a user, there is nowhere for the result sets to go.

Although stored procedures are set for automatic execution individually, the SQL Server **scan for startup procs** configuration option can be set to prevent all stored procedures from executing automatically when SQL Server starts.

To set or unset a stored procedure for automatic execution

⊞ <u>Transact-SQL</u>

∃ <u>SQL-DMO</u>

To set or unset the scan for startup procs configuration option

∃ <u>Transact-SQL</u>

See Also

scan for startup procs Option

Modifying and Renaming a Stored Procedure

If you need to change the statements or parameters in a stored procedure, you can either delete and re-create the stored procedure or alter the stored procedure in a single step. When you delete and re-create a stored procedure, all permissions associated with the stored procedure are lost. When you alter the stored procedure, the procedure or parameter definition is changed but the permissions defined for the stored procedure are retained.

You can also rename a stored procedure. The new name must follow the rules for identifiers. You can rename only the stored procedures that you own, but the database owner can change the name of any user's stored procedure. The stored procedure to be renamed must be in the current database.

A stored procedure can also be modified to encrypt the definition or cause the procedure to be recompiled each time it is executed.

Note Changing the name or definition of a stored procedure can cause any dependent objects to fail when executed if those dependent objects are not also updated to reflect the changes made to the stored procedure.

To modify a stored procedure

⊞ <u>Transact-SQL</u>

Recompiling a Stored Procedure

As a database is changed by such actions as adding indexes or changing data in indexed columns, the original query plans used to access its tables should be optimized again by recompiling them. This optimization happens automatically the first time a stored procedure is run after Microsoft® SQL Server[™] 2000 is restarted. It also occurs if an underlying table used by the stored procedure changes. But if a new index is added from which the stored procedure might benefit, optimization does not automatically happen (until the next time the stored procedure is run after SQL Server is restarted).

SQL Server provides three ways to recompile a stored procedure:

- The **sp_recompile** system stored procedure forces a recompile of a stored procedure the next time it is run.
- Creating a stored procedure that specifies the WITH RECOMPILE option in its definition indicates that SQL Server does not cache a plan for this stored procedure; the stored procedure is recompiled each time it is executed. Use the WITH RECOMPILE option when stored procedures take parameters whose values differ widely between executions of the stored procedure, resulting in different execution plans to be created each time. Use of this option is uncommon, and causes the stored procedure to execute more slowly because the stored procedure must be recompiled each time it is executed.
- You can force the stored procedure to be recompiled by specifying the WITH RECOMPILE option when you execute the stored procedure. Use this option only if the parameter you are supplying is atypical or if the data has significantly changed since the stored procedure was created.

Note If an object referenced by a stored procedure is deleted or renamed, an error is returned when the stored procedure is executed. If, however, an object referenced in a stored procedure is replaced with an object of the same name, the stored procedure executes without having to be recompiled.

To recompile a stored procedure next time it is run

⊡ <u>Transact-SQL</u>

See Also

Creating a Stored Procedure Deferred Name Resolution and Compilation Executing a Stored Procedure Programming Stored Procedures

Viewing a Stored Procedure

Several system stored procedures provide information from the system tables about stored procedures. Using these stored procedures, you can:

- See the Transact-SQL statements used to create a stored procedure. This can be useful if you do not have the Transact-SQL script files used to create the stored procedure.
- Get information about a stored procedure such as its owner, when it was created, and its parameters.
- List the objects used by the specified stored procedure, and the procedures that use the specified stored procedure. This information can be used to identify the procedures affected by the changing or removal of an object in the database.

To view the definition of a stored procedure

⊞ <u>Transact-SQL</u>

Deleting a Stored Procedure

You can delete a stored procedure when you no longer need it. If a stored procedure that has been deleted is called by another stored procedure, Microsoft® SQL Server[™] 2000 displays an error message when the calling procedure is executed. However, if a new stored procedure of the same name and the same parameters is defined to replace the one that was deleted, other procedures that reference it will still execute successfully. For example, if stored procedure **proc1** references stored procedure **proc2**, and **proc2** is deleted and a different stored procedure called **proc2** is created, **proc1** now references the new stored procedure. **proc1** does not have to be recompiled.

After stored procedures have been grouped, individual stored procedures within the group cannot be deleted. Deleting a stored procedure deletes all stored procedures in the same group.

To delete a stored procedure

∃<u>Transact-SQL</u>

Enforcing Business Rules with Triggers

Microsoft® SQL Server[™] 2000 provides two primary mechanisms for enforcing business rules and data integrity: constraints and triggers. A trigger is a special type of stored procedure that automatically takes effect when the data in a specified table is modified. A trigger is invoked in response to an INSERT, UPDATE, or DELETE statement. A trigger can query other tables and can include complex Transact-SQL statements. The trigger and the statement that fires it are treated as a single transaction, which can be rolled back from within the trigger. If a severe error is detected (for example, insufficient disk space), the entire transaction automatically rolls back.

Triggers are useful in these ways:

- Triggers can cascade changes through related tables in the database; however, these changes can be executed more efficiently using cascading referential integrity constraints.
- Triggers can enforce restrictions that are more complex than those defined with CHECK constraints.

Unlike CHECK constraints, triggers can reference columns in other tables. For example, a trigger can use a SELECT from another table to compare to the inserted or updated data and to perform additional actions, such as modify the data or display a user-defined error message.

- Triggers can also evaluate the state of a table before and after a data modification and take action(s) based on that difference.
- Multiple triggers of the same type (INSERT, UPDATE, or DELETE) on a table allow multiple, different actions to take place in response to the same modification statement.

Triggers Compared to Constraints

Constraints and triggers each have benefits that make them useful in special

situations. The primary benefit of triggers is that they can contain complex processing logic that uses Transact-SQL code. Therefore, triggers can support all of the functionality of constraints; however, triggers are not always the best method for a given feature.

Entity integrity should always be enforced at the lowest level by indexes that are part of PRIMARY KEY and UNIQUE constraints or are created independently of constraints. <u>Domain integrity</u> should be enforced through CHECK constraints, and <u>referential integrity (RI)</u> should be enforced through FOREIGN KEY constraints, assuming their features meet the functional needs of the application.

Triggers are most useful when the features supported by constraints cannot meet the functional needs of the application. For example:

- FOREIGN KEY constraints can validate a column value only with an exact match to a value in another column, unless the REFERENCES clause defines a cascading referential action.
- A CHECK constraint can validate a column value only against a logical expression or another column in the same table. If your application requires that a column value be validated against a column in another table, you must use a trigger.
- Constraints can communicate about errors only through standardized system error messages. If your application requires (or can benefit from) customized messages and more complex error handling, you must use a trigger.

Triggers can cascade changes through related tables in the database; however, these changes can be executed more efficiently through cascading referential integrity constraints.

• Triggers can disallow or roll back changes that violate referential integrity, thereby canceling the attempted data modification. Such a trigger might go into effect when you change a foreign key and the new value does not match its primary key. For example, you can create an insert trigger on **titleauthor.title_id** that rolls back an insert if the new value does not match some value in **titles.title_id**. However, FOREIGN

KEY constraints are usually used for this purpose.

• If constraints exist on the trigger table, they are checked after the INSTEAD OF trigger execution but prior to the AFTER trigger execution. If the constraints are violated, the INSTEAD OF trigger actions are rolled back and the AFTER trigger is not executed.

See Also

Cascading Referential Integrity Constraints CHECK Constraints Data Integrity Stored Procedures Using Constraints, Defaults, and Null Values

Designing Triggers

Microsoft[®] SQL Server[™] 2000 provides two options when designing triggers:

- INSTEAD OF triggers are executed in place of the usual triggering action. INSTEAD OF triggers can also be defined on views with one or more base tables, where they can extend the types of updates a view can support.
- AFTER triggers are executed after the action of the INSERT, UPDATE, or DELETE statement is performed. Specifying AFTER is the same as specifying FOR, which is the only option available in earlier versions of SQL Server. AFTER triggers can be specified only on tables.

Function	AFTER trigger	INSTEAD OF trigger	
Applicability	Tables	Tables and views	
Quantity per table or	Multiple per	One per triggering action	
view	triggering action	(UPDATE, DELETE, and	
	(UPDATE,	INSERT)	
	DELETE, and		
	INSERT)		
Cascading references	No restrictions apply	Are not allowed on tables that	
		are targets of cascaded	
		referential integrity	
		constraints.	
Execution	After:	Before:	
	Constraint	Constraint	
	processing	processing	
		In place of:	
	 Declarative referential actions 	• The triggering action	
		After:	

This table compares the functionality of the AFTER and INSTEAD OF triggers.

	 inserted and deleted tables creation The triggering action 	• inserted and deleted tables creation
Order of execution	First and last execution may be specified	Not applicable
text, ntext , and image column references in inserted and deleted tables	Not allowed	Allowed

See Also

Using the inserted and deleted Tables

Specifying When a Trigger Fires

You can specify one of two options to control when a trigger fires:

- AFTER triggers fire after the triggering action (INSERT, UPDATE, or DELETE) and after any constraints are processed. You can request AFTER triggers by specifying either the AFTER or FOR keywords. Because the FOR keyword has the same effect as AFTER, triggers with the FOR keyword are also classified as AFTER triggers.
- INSTEAD OF triggers fire in place of the triggering action and before constraints are processed.

Each table or view can have one INSTEAD OF trigger for each triggering action (UPDATE, DELETE, and INSERT). A table can have several AFTER triggers for each triggering action.

Examples

A. Use the INSTEAD OF trigger to replace the standard triggering action

CREATE TRIGGER TableAInsertTrig ON TableA INSTEAD OF INSERT AS ...

B. Use the AFTER trigger to augment the standard triggering action

CREATE TRIGGER TableBDeleteTrig ON TableB AFTER DELETE AS ...

C. Use the FOR trigger to augment the standard triggering action

-- This statement uses the FOR keyword to generate an AFTER trigger CREATE TRIGGER TableCUpdateTrig ON TableC FOR UPDATE AS ...

Trigger Execution

AFTER triggers are never executed if a constraint violation occurs; therefore, these triggers cannot be used for any processing that might prevent constraint violations.

INSTEAD OF triggers are executed instead of the triggering action. These triggers are executed after the **inserted** and **deleted** tables reflecting the changes to the base table are created, but before any other actions are taken. They are executed before any constraints, so can perform preprocessing that supplements the constraint actions.

If an INSTEAD OF trigger defined on a table executes a statement against the table that would usually fire the INSTEAD OF trigger again, the trigger is not called recursively. Instead, the statement is processed as if the table had no INSTEAD OF trigger and starts the chain of constraint operations and AFTER trigger executions. For example, if a trigger is defined as an INSTEAD OF INSERT trigger for a table, and the trigger executes an INSERT statement on the same table, the INSERT statement executed by the INSTEAD OF trigger does not call the trigger again. The INSERT executed by the trigger starts the process of performing constraint actions and firing any AFTER INSERT triggers defined for the table.

If an INSTEAD OF trigger defined on a view executes a statement against the view that would usually fire the INSTEAD OF trigger again, it is not called recursively. Instead, the statement is resolved as modifications against the base tables underlying the view. In this case, the view definition must meet all of the restrictions for an updatable view. For a definition of updatable views, see Modifying Data Through a View. For example, if a trigger is defined as an INSTEAD OF UPDATE trigger for a view, and the trigger executes an UPDATE statement referencing the same view, the UPDATE statement executed by the INSTEAD OF trigger does not call the trigger again. The UPDATE executed by the trigger is processed against the view as if the view did not have an INSTEAD OF trigger. The columns changed by the UPDATE must be resolved to a single base table. Each modification to an underlying base table starts the chain of applying constraints and firing AFTER triggers defined for the table.

Trigger performance overhead is usually low. The time involved in running a

trigger is spent mostly in referencing other tables, which can be either in memory or on the database device. The **deleted** and **inserted** tables are always in memory. The location of other tables referenced by the trigger determines the amount of time the operation requires.

Note The use of cursors in triggers is not recommended because of the potentially negative impact on performance. Use rowset-based logic rather than cursors to design a trigger that affects multiple rows.

See Also

Using the inserted and deleted Tables

Designing INSTEAD OF Triggers

The primary advantage of INSTEAD OF triggers is that they allow views that would not be updatable support updates. A view comprising multiple base tables must use an INSTEAD OF trigger to support inserts, updates and deletes that reference data in the tables. Another advantage of INSTEAD OF triggers is that they allow you to code logic that can reject parts of a batch while allowing other parts of a batch succeed.

An INSTEAD OF trigger can take actions such as:

- Ignoring parts of a batch.
- Not processing a part of a batch and logging the problem rows.
- Taking an alternative action if an error condition is encountered.

Note INSTEAD OF DELETE and INSTEAD OF UPDATE triggers cannot be defined on a table that has a foreign key defined with a DELETE or UPDATE action.

Coding this logic as part of an INSTEAD OF trigger prevents all applications accessing the data from having to reimplement the logic.

In the following sequence of Transact-SQL statements, an INSTEAD OF trigger updates two base tables from a view. In addition, two approaches to handling errors are shown:

- Duplicate inserts to the **Person** table are ignored, and the information from the insert is logged in the **PersonDuplicates** table.
- Inserts of duplicates to the **EmployeeTable** are turned into an UPDATE statement that retrieves the current information into the **EmployeeTable** without generating a duplicate key violation.

The Transact-SQL statements create two base tables, a view, a table to record errors, and the INSTEAD OF trigger on the view. These tables separate personal

and business data and are the base tables for the view:

```
CREATE TABLE Person
 (
  SSN
          char(11) PRIMARY KEY,
           nvarchar(100),
  Name
  Address
           nvarchar(100),
 Birthdate datetime
CREATE TABLE EmployeeTable
 (
 EmployeeID
               int PRIMARY KEY,
  SSN
             char(11) UNIQUE,
               nvarchar(10),
  Department
  Salary
             money,
  CONSTRAINT FKEmpPer FOREIGN KEY (SSN)
  REFERENCES Person (SSN)
 )
```

This view reports all relevant data from the two tables for a person:

```
CREATE VIEW Employee AS
SELECT P.SSN as SSN, Name, Address,
Birthdate, EmployeeID, Department, Salary
FROM Person P, EmployeeTable E
WHERE P.SSN = E.SSN
```

You can record attempts to insert rows with duplicate social security numbers. The **PersonDuplicates** table logs the inserted values, the name of the user who attempted the insert, and the time of the insert:

```
CREATE TABLE PersonDuplicates
(
SSN char(11),
```

```
Name nvarchar(100),
Address nvarchar(100),
Birthdate datetime,
InsertSNAME nchar(100),
WhenInserted datetime
)
```

The INSTEAD OF trigger inserts rows into multiple base tables from a single view. Attempts to insert rows with duplicate social security numbers are recorded in the **PersonDuplicates** table. Duplicate rows in the **EmployeeTable** are changed to update statements.

CREATE TRIGGER IO_Trig_INS_Employee ON Employee INSTEAD OF INSERT AS BEGIN SET NOCOUNT ON -- Check for duplicate Person. If no duplicate, do an insert. IF (NOT EXISTS (SELECT P.SSN FROM Person P, inserted I WHERE P.SSN = I.SSN)) **INSERT INTO Person** SELECT SSN,Name,Address,Birthdate,Comment FROM inserted ELSE -- Log attempt to insert duplicate Person row in PersonDuplicates table **INSERT INTO PersonDuplicates** SELECT SSN,Name,Address,Birthdate,SUSER_SNAME(),GETD FROM inserted -- Check for duplicate Employee. If no duplicate, do an insert. IF (NOT EXISTS (SELECT E.SSN FROM EmployeeTable E, inserted WHERE E.SSN = inserted.SSN)) **INSERT INTO EmployeeTable**

SELECT EmployeeID,SSN, Department, Salary,Comment FROM inserted ELSE --If duplicate, change to UPDATE so that there will not --be a duplicate key violation error. UPDATE EmployeeTable SET EmployeeID = I.EmployeeID, Department = I.Department, Salary = I.Salary, Comment = I.Comment FROM EmployeeTable E, inserted I WHERE E.SSN = I.SSN END

Creating a Trigger

Before you create a trigger, consider that:

- The CREATE TRIGGER statement must be the first statement in the batch. All other statements that follow in that batch are interpreted as part of the definition of the CREATE TRIGGER statement.
- Permission to create triggers defaults to the table owner, who cannot transfer it to other users.
- Triggers are database objects, and their names must follow the rules for identifiers.
- You can create a trigger only in the current database, although a trigger can reference objects outside of the current database.
- A trigger cannot be created on a temporary or system table, although triggers can reference temporary tables. System tables should not be referenced; use the Information Schema Views instead. For more information, see <u>Information Schema Views</u>.
- INSTEAD OF DELETE and INSTEAD OF UPDATE triggers cannot be defined on a table that has a foreign key defined with a DELETE or UPDATE action.
- Although a TRUNCATE TABLE statement is like a DELETE statement without a WHERE clause (it deletes all rows), it does not cause DELETE triggers to fire because the TRUNCATE TABLE statement is not logged.
- The WRITETEXT statement does not cause the INSERT or UPDATE

triggers to fire.

When you create a trigger, specify:

- The name.
- The table upon which the trigger is defined.
- When the trigger is to fire.
- The data modification statements that activate the trigger. Valid options are INSERT, UPDATE, or DELETE. More than one data modification statement can activate the same trigger. For example, a trigger can be activated by an INSERT and an UPDATE statement.
- The programming statements that perform the trigger action.

Multiple Triggers

A table can have multiple AFTER triggers of a given type provided they have different names; each trigger can perform numerous functions. However, each trigger can apply to only one table, although a single trigger can apply to any subset of three user actions (UPDATE, INSERT, and DELETE).

A table can have only one INSTEAD OF trigger of a given type.

Trigger Permissions and Ownership

CREATE TRIGGER permissions default to the table owner on which the trigger is defined, the **sysadmin** fixed server role, and members of the **db_owner** and **db_ddladmin** fixed database roles, and are not transferable.

If an INSTEAD OF trigger is created on a view, the ownership chain is broken if the view owner does not also own the base tables referenced by the view and trigger. For a base table not owned by the view owner, the table owner must separately grant the necessary permissions to anybody reading or updating the view. If the same user owns both the view and the underlying base tables, they have to grant other users permissions only on the view, not individual base tables. For more information, see <u>Using Ownership Chains</u>.

To create a trigger

 \boxplus <u>Transact-SQL</u>

Programming Triggers

Almost any Transact-SQL statement that can be written as a batch can be used to create a trigger, except for these:

ALTER DATABASE	CREATE DATABASE	DISK INIT
DISK RESIZE	DROP DATABASE	LOAD DATABASE
LOAD LOG	RECONFIGURE	RESTORE
		DATABASE
RESTORE LOG		

IMPORTANT The DISK RESIZE, DISK INIT, LOAD DATABASE, and LOAD LOG statements are included in Microsoft® SQL Server 2000[™] for backward compatibility only, and may not be supported in the future.

Encrypting Trigger Definitions

If you want to ensure that other users cannot view the trigger definition, you can use the WITH ENCRYPTION clause. The trigger definition is then stored in an unreadable form.

Once encrypted, the definition of the trigger cannot be decrypted and cannot be viewed by anyone, including the owner of the trigger or the system administrator.

SET Statement Options

When an ODBC application connects to SQL Server, the server automatically sets these options for the session:

- SET QUOTED_IDENTIFIER ON
- SET TEXTSIZE 2147483647
- SET ANSI_DEFAULTS ON

- SET CURSOR_CLOSE_ON_COMMIT OFF
- SET IMPLICIT_TRANSACTIONS OFF

These settings increase the portability of ODBC applications. Because DB-Library–based applications generally do not set these options, triggers should be tested with the SET options listed above set to both ON and OFF. This ensures that the triggers work correctly regardless of the options a particular connection may have set when it invokes the trigger. A trigger that requires a particular setting for one of these options should issue a SET statement at the start of the trigger. This SET statement remains in effect only for the execution of the trigger; when the trigger completes, the original setting is restored.

Testing for Changes to Specific Columns

The IF UPDATE (*column_name*) clause in the definition of a trigger can be used to determine if an INSERT or UPDATE statement affected a specific column in the table. The clause evaluates to TRUE whenever the column is assigned a value.

Note Because a specific value in a column cannot be deleted using the DELETE statement, the IF UPDATE clause does not apply to the DELETE statement.

Alternatively, the IF COLUMNS_UPDATED() clause can be used to check which columns in a table were updated by an INSERT or UPDATE statement. This clause uses an integer bitmask to specify the columns to test. For more information, see <u>CREATE TRIGGER</u>.

Examples

A. Use the IF UPDATE clause to test data modifications

This example creates an INSERT trigger **my_trig** on table **my_table** and tests whether column **b** was affected by any INSERT statements.

```
CREATE TABLE my_table*
(a int NULL, b int NULL)
GO
```

```
CREATE TRIGGER my_trig
ON my_table
FOR INSERT
AS
IF UPDATE(b)
PRINT 'Column b Modified'
GO
```

B. Use the COLUMNS UPDATED() clause to test data modifications

This example obtains similar results using the COLUMNS_UPDATED() clause.

```
CREATE TRIGGER my_trig2
ON my_table
FOR INSERT
AS
IF ( COLUMNS_UPDATED() & 2 = 2 )
PRINT 'Column b Modified'
GO
```

Deferred Name Resolution

Triggers can refer to tables that do not exist at trigger creation time. This is called deferred name resolution. For more information about deferred name resolution, see <u>Deferred Name Resolution and Compilation</u>.

Note If an object referenced by a trigger is deleted or renamed, an error is returned when the trigger is executed. However, if an object referenced in a trigger is replaced with an object of the same name, the trigger executes without having to be re-created. For example, if trigger **trig1** references table **test1**, and **test1** is deleted and a different table called **test1** is created, **trig1** now references the new table.

Returning Results

It is recommended that a trigger not return any results. This is because special handling for these returned results must be written into every application in which modifications to the trigger table are allowed. To prevent any results from being returned from a trigger, do not include either SELECT statements or variable assignments in the definition of the trigger. If variable assignment must occur in a trigger, use a SET NOCOUNT statement at the beginning of the trigger to eliminate the return of any result sets.

See Also

CREATE TRIGGER

<u>SELECT</u>

<u>SET</u>

Using Triggers that Include ROLLBACK TRANSACTION

When triggers that include ROLLBACK TRANSACTION statements are executed from a batch, they cancel the entire batch. In the following example, if the INSERT statement fires a trigger that includes a ROLLBACK TRANSACTION, the DELETE statement is not executed because the batch is canceled:

/* Start of Batch */ INSERT employee VALUES ('XYZ12345M', 'New', 'M', 'Employee', 1 DELETE employee WHERE emp_id = 'PMA42628M' GO

If triggers that include ROLLBACK TRANSACTION statements are fired from within a user-defined transaction, the ROLLBACK TRANSACTION rolls back the entire transaction. In this example, if the INSERT statement fires a trigger that includes a ROLLBACK TRANSACTION, the UPDATE statement is also rolled back:

/* Start of Transaction */ BEGIN TRANSACTION UPDATE employee SET hire_date = '7/1/94' WHERE emp_id = 'VPA.' INSERT employee VALUES ('XYZ12345M', 'New', 'M', 'Employee', 1

See Also

Rollbacks in Stored Procedures and Triggers

ROLLBACK TRANSACTION

Transactions

Using the inserted and deleted Tables

Two special tables are used in trigger statements: the **deleted** table and the **inserted** table. Microsoft® SQL Server[™] 2000 automatically creates and manages these tables. You can use these temporary, memory-resident tables to test the effects of certain data modifications and to set conditions for trigger actions; however, you cannot alter the data in the tables directly.

The **inserted** and **deleted** tables are used primarily in triggers to:

- Extend referential integrity between tables.
- Insert or update data in base tables underlying a view.
- Check for errors and take action based on the error.
- Find the difference between the state of a table before and after a data modification and take action(s) based on that difference.

The **deleted** table stores copies of the affected rows during DELETE and UPDATE statements. During the execution of a DELETE or UPDATE statement, rows are deleted from the trigger table and transferred to the **deleted** table. The **deleted** table and the trigger table ordinarily have no rows in common.

The **inserted** table stores copies of the affected rows during INSERT and UPDATE statements. During an insert or update transaction, new rows are added simultaneously to both the **inserted** table and the trigger table. The rows in the **inserted** table are copies of the new rows in the trigger table.

An update transaction is similar to a delete operation followed by an insert operation; the old rows are copied to the **deleted** table first, and then the new rows are copied to the trigger table and to the **inserted** table.

When you set trigger conditions, use the **inserted** and **deleted** tables appropriately for the action that fired the trigger. Although referencing the **deleted** table while testing an INSERT, or the **inserted** table while testing a DELETE does not cause any errors, these trigger test tables do not contain any rows in these cases.

Note If trigger actions depend on the number of rows a data modification effects, use tests (such as an examination of @@ROWCOUNT) for multirow data modifications (an INSERT, DELETE, or UPDATE based on a SELECT statement), and take appropriate actions.

SQL Server 2000 does not allow **text**, **ntext**, or **image** column references in the **inserted** and **deleted** tables for AFTER triggers; however, these column references are allowed for INSTEAD OF triggers. For more information, see <u>CREATE TRIGGER</u>.

Using the inserted and deleted Tables in INSTEAD OF Triggers

The **inserted** and **deleted** tables passed to INSTEAD OF triggers defined on tables follow the same rules as the **inserted** and **deleted** tables passed to AFTER triggers. The format of the **inserted** and **deleted** tables is the same as the format of the table on which the INSTEAD OF trigger is defined. Each column in the **inserted** and **deleted** tables maps directly to a column in the base table.

The rules regarding when an INSERT or UPDATE statement referencing a table with an INSTEAD OF trigger must supply values for columns are the same as if the table did not have an INSTEAD OF trigger:

- Values cannot be specified for computed columns or columns with a **timestamp** data type.
- Values cannot be specified for columns with an IDENTITY property, unless IDENTITY_INSERT is ON for that column. When IDENTITY_INSERT is ON, INSERT statements must supply a value.
- INSERT statements must supply values for all NOT NULL columns that do not have DEFAULT constraints.
- For any columns except computed, identity, or **timestamp** columns, values are optional for any column that allows nulls, or any NOT NULL column that has a DEFAULT definition.

When an INSERT, UPDATE, or DELETE statement references a view that has an INSTEAD OF trigger, the database engine calls the trigger instead of taking any direct action against any table. The trigger must use the information presented in the **inserted** and **deleted** tables to build any statements needed to implement the requested action in the base tables even when the format of the information in the **inserted** and **deleted** tables built for the view is different than the format of the data in the base tables.

The format of the **inserted** and **deleted** tables passed to an INSTEAD OF trigger defined on a view matches the select list of the SELECT statement defined for the view. For example:

CREATE VIEW EmployeeNames (EmployeeID, LName, FName) AS SELECT EmployeeID, LastName, FirstName FROM Northwind.dbo.Employees

The result set for this view has three columns: an **int** column and two **nvarchar** columns. The **inserted** and **deleted** tables passed to an INSTEAD OF trigger defined on the view also have an **int** column named **EmployeeID**, an **nvarchar** column named **LName**, and an **nvarchar** column named **FName**.

The select list of a view can also contain expressions that do not map directly to a single base table column. Some view expressions, such as a constant or function invocation, may not reference any columns and can be ignored. Complex expressions can reference multiple columns, yet the **inserted** and **deleted** tables have only one value for each inserted row. The same issues apply to simple expressions in a view if they reference a computed column that has a complex expression. An INSTEAD OF trigger on the view must handle these types of expressions. For more information, see <u>Expressions and Computed</u> <u>Columns in INSTEAD OF Triggers on Views</u>.

Multirow Considerations

An important consideration to keep in mind when writing the code for a trigger is that the statement that causes the trigger to fire can be a single statement that affects multiple rows of data, rather than a single row. This is common for UPDATE and DELETE triggers because these statements often affect multiple rows. It is less common for INSERT triggers, because the basic INSERT statement adds only a single row. However, because an INSERT trigger can be fired by an INSERT INTO (*table_name*) SELECT statement, the insertion of many rows may result in a single trigger invocation.

Multirow considerations are particularly important when the function of a trigger is to automatically recalculate summary values from one table and store the results in another for ongoing tallies.

Note The use of cursors in triggers is not recommended because of the potentially negative impact on performance. Use rowset-based logic rather than cursors to design a trigger that affects multiple rows.

Examples

The triggers in the following examples are designed to store a running total of a column in another table.

A. Store a running total for a single-row insert

The first version of the trigger works well for a single-row insert, when a row of data is loaded into the **sales** table. An INSERT statement fires the trigger, and the new row is loaded into the **inserted** table for the duration of the trigger execution. The UPDATE statement reads the **qty** column value for the row and adds it to the existing value in the **ytd_sales** column in the **titles** table. The WHERE clause ensures that the updated row in the **sales** table matches the **title_id** of the row in the **inserted** table.

-- Trigger is valid for single-row inserts. CREATE TRIGGER intrig ON sales

AFTER INSERT AS

UPDATE titles SET ytd_sales = ytd_sales + qty FROM inserted WHERE titles.title_id = inserted.title_id

B. Store a running total for a multirow or single row insert

In the case of a multirow insert, the trigger in Example A might not operate correctly; the expression to the right of an assignment expression in an UPDATE statement (**ytd_sales + qty**) can be only a single value, not a list of values. So the effect of the trigger is to obtain a value from any single row in the **inserted** table and add it to the existing **ytd_sales** value in the **titles** table for a given **title_id** value. This might not have the desired effect if a single **title_id** value occurred more than once in the **inserted** table.

To update the **titles** table properly, the trigger has to accommodate the possibility of multiple rows in the **inserted** table. This can be done with the SUM function that calculates the total **qty** for a group of rows in the **inserted** table for each **title_id**. The SUM function is placed in a <u>correlated subquery</u> (the SELECT statement in parentheses), which returns a single value for each **title_id** in the **inserted** table that matches or is correlated with a **title_id** in the **titles** table.

```
-- Trigger is valid for multirow and single-row inserts.
CREATE TRIGGER intrig
ON sales
AFTER INSERT AS
```

```
UPDATE titles
SET ytd_sales = ytd_sales +
(SELECT SUM(qty) -- Correlated subquery.
FROM inserted
WHERE titles.title_id = inserted.title_id)
WHERE titles.title_id IN
(SELECT title_id FROM inserted)
```

This trigger also works correctly in a single-row insert; the sum of the **qty** value column is the sum of a single row. However, with this trigger the correlated subquery and the IN operator used in the WHERE clause require additional processing from Microsoft® SQL Server[™] 2000, which is unnecessary for a single-row insert.

C. Store a running total based on the type of insert

You can change the trigger to use the method optimal for the number of rows. For example, the @@ROWCOUNT function can be used in the logic of the trigger to distinguish between a single and a multirow insert.

```
-- Trigger valid for multirow and single row inserts
-- and optimal for single row inserts.
CREATE TRIGGER intrig
ON sales
FOR INSERT AS
IF @@ROWCOUNT = 1
BEGIN
 UPDATE titles
 SET ytd_sales = ytd_sales + qty
 FROM inserted
 WHERE titles.title_id = inserted.title_id
END
ELSE
BEGIN
 UPDATE titles
 SET ytd_sales = ytd_sales +
 (SELECT SUM(qty)
   FROM inserted
   WHERE titles.title_id = inserted.title_id)
 WHERE titles.title id IN
   (SELECT title id FROM inserted)
END
```

Conditional INSERT Trigger

A trigger rejects or accepts each data modification transaction as a whole. However, you do not have to roll back all data modifications simply because some of them are unacceptable. Using a correlated subquery in a trigger can force the trigger to examine the modified rows one by one.

Examples

A. Use an AFTER INSERT trigger

The following example assumes the existence of a table called **newsale** in the **pubs** database. This the CREATE statement for **newsale**:

CREATE TABLE newsale

```
(stor_id char(4),
ord_num varchar(20),
date datetime,
qty smallint,
payterms varchar(12),
title_id tid)
```

If you want to examine each of the records you are trying to insert, the trigger **conditionalinsert** analyzes the insert row by row, and then deletes the rows that do not have a **title_id** in **titles**.

```
CREATE TRIGGER conditionalinsert
ON sales
AFTER INSERT AS
IF
(SELECT COUNT(*) FROM titles, inserted
WHERE titles.title_id = inserted.title_id) <> @@ROWCOUNT
BEGIN
DELETE sales FROM sales, inserted
WHERE sales.title_id = inserted.title_id AND
```

```
inserted.title_id NOT IN
(SELECT title_id
FROM titles)
PRINT 'Only sales records with matching title_ids added.'
END
```

When unacceptable titles have been inserted, the transaction is not rolled back; instead, the trigger deletes the unwanted rows. This ability to delete rows that have been inserted relies on the order in which processing occurs when triggers are fired. First, rows are inserted into the **sales** table and the **inserted** table, and then the trigger fires.

To test the trigger, insert four rows in the **newsale** table. Two of the **newsale** rows have **title_ids** that do not match any of those already in the **titles** table:

stor_id	ord_num	date	qty	payterms	title_id
7066	QA7442.3	Jul 25 1995 8:35AM	75	Net 30	PS1372
7066	QA7442.3	Jul 24 1995 8:35AM	75	Net 60	BU7832
7067	D4482	Jul 27 1995	10	Net 30	PSxxxx
		12:00AM			
7131	N914008	Jul 27 1995	20	Net 30	PSyyyy
		12:00AM			

newsale

Next, insert data from **newsale** into **sales**. The statement looks like this:

INSERT sales SELECT * FROM newsale

The **title_ids** PSxxxx and PSyyyy do not match any in the **titles** table, and the **conditionalinsert** trigger deletes these two rows from the **sales** and **inserted** tables.

Specifying First and Last Triggers

You can specify that one of the AFTER triggers associated with a table be either the first AFTER trigger or the last AFTER trigger executed for each of the INSERT, DELETE, and UPDATE triggering actions. The AFTER triggers that are fired between the first and last triggers are executed in undefined order.

To specify the order for an AFTER trigger, use the **sp_settriggerorder** stored procedure. The options available are:

• First

Specifies that the trigger is the first AFTER trigger fired for a triggering action.

• Last

Specifies that the trigger is the last AFTER trigger fired for a triggering action.

• None

Specifies that there is no specific order in which the trigger should be fired. Used mainly to reset a trigger from being either first or last.

This is an example of using **sp_settriggerorder**:

sp_settriggerorder @triggername = 'MyTrigger', @order = 'first', @stm

IMPORTANT The first and last triggers must be two different triggers.

A table may have INSERT, UPDATE, and DELETE triggers defined on it at the same time. Each statement type can have its own first and last triggers, but they cannot be the same triggers.

If the first or last trigger defined for a table does not cover a triggering action, such as not covering FOR UPDATE, FOR DELETE, or FOR INSERT, there is no first or last trigger for the missing actions.

INSTEAD OF triggers cannot be specified as first or last triggers. INSTEAD OF triggers are fired before updates are made to the underlying tables. However, if

updates are made by an INSTEAD OF trigger to underlying tables, the updates occur after triggers defined on the table, including the first trigger. For example, if an INSTEAD OF trigger on a view updates a base table and the base table contains three triggers, the three triggers in the table fire before the data is inserted by the INSTEAD OF trigger. For more information, see <u>Specifying</u> <u>When a Trigger Fires</u>.

If an ALTER TRIGGER statement changes a first or last trigger, the **First** or **Last** attribute is dropped and the order value is set to **None**; the order must be reset with **sp_settriggerorder**.

The OBJECTPROPERTY function reports whether a trigger is a first or last trigger using the properties **ExecIsFirstTrigger** and **ExecIsLastTrigger**.

Replication generates a first trigger automatically for any table that is an immediate or queued update subscriber. Replication requires that its trigger is the first trigger. Replication raises an error if you try to make a table that has a first trigger an immediate or queued update Subscriber. If you make a user-defined trigger a first trigger after a table has been made an immediate or queued update Subscriber, **sp_settriggerorder** returns an error. If you use ALTER on the replication trigger, or use **sp_settriggerorder** to change the replication trigger to a last or none trigger, the subscription does not work correctly.

See Also

OBJECTPROPERTY

<u>sp_settriggerorder</u>

Using Nested Triggers

Triggers are nested when a trigger performs an action that initiates another trigger, which can initiate another trigger, and so on. Triggers can be nested up to 32 levels, and you can control whether triggers can be nested through the **nested triggers** server configuration option.

If nested triggers are allowed and a trigger in the chain starts an infinite loop, the nesting level is exceeded and the trigger terminates.

You can use nested triggers to perform useful housekeeping functions such as storing a backup copy of rows affected by a previous trigger. For example, you can create a trigger on **titleauthor** that saves a backup copy of the **titleauthor** rows that the **delcascadetrig** trigger deleted. With the **delcascadetrig** trigger in effect, deleting **title_id** PS2091 from **titles** deletes the corresponding row or rows from **titleauthor**. To save the data, you create a DELETE trigger on **titleauthor** that saves the deleted data into another separately created table, **del_save**. For example:

```
CREATE TRIGGER savedel
ON titleauthor
FOR DELETE
AS
INSERT del_save
SELECT * FROM deleted
```

Using nested triggers in an order-dependent sequence is not recommended. Use separate triggers to cascade data modifications.

Note Because triggers execute within a transaction, a failure at any level of a set of nested triggers cancels the entire transaction, and all data modifications are rolled back. Include PRINT statements in your triggers so that you can determine where the failure occurred.

Recursive Triggers

A trigger does not call itself recursively unless the RECURSIVE_TRIGGERS

database option is set. There are two types of recursion:

• Direct recursion

Occurs when a trigger fires and performs an action that causes the same trigger to fire again. For example, an application updates table **T3**, which causes trigger **Trig3** to fire. **Trig3** updates table **T3** again, which causes trigger **Trig3** to fire again.

• Indirect recursion

Occurs when a trigger fires and performs an action that causes a trigger on another table to fire. This second trigger causes an update to occur on the original table, which causes the original trigger to fire again. For example, an application updates table **T1**, which causes trigger **Trig1** to fire. **Trig1** updates table **T2**, which causes trigger **Trig2** to fire. **Trig2** in turn updates table **T1** which causes **Trig1** to fire again.

Only direct recursion is prevented when the RECURSIVE_TRIGGERS database option is set to OFF. To disable indirect recursion, set the **nested triggers** server option to **0**, as well.

Examples

A. Use recursive triggers to solve self-referencing relationships

One use for recursive triggers is on a table with a self-referencing relationship (also known as transitive closure). For example, the table **emp_mgr** defines:

- An employee (**emp**) in a company.
- The manager for each employee (**mgr**).
- The total number of employees in the organizational tree reporting to each employee (**NoOfReports**).

A recursive UPDATE trigger can be used to keep the **NoOfReports** column upto-date as new employee records are inserted. The INSERT trigger updates the **NoOfReports** column of the manager record, which recursively updates the **NoOfReports** column of other records up the management hierarchy.

```
USE pubs
GO
-- Turn recursive triggers ON in the database.
ALTER DATABASE pubs
 SET RECURSIVE TRIGGERS ON
GO
CREATE TABLE emp_mgr (
 emp char(30) PRIMARY KEY,
  mgr char(30) NULL FOREIGN KEY REFERENCES emp_mgr(emj
 NoOfReports int DEFAULT 0
)
GO
CREATE TRIGGER emp_mgrins ON emp_mgr
FOR INSERT
AS
DECLARE @e char(30), @m char(30)
DECLARE c1 CURSOR FOR
 SELECT emp_mgr.emp
 FROM emp_mgr, inserted
 WHERE emp_mgr.emp = inserted.mgr
OPEN c1
FETCH NEXT FROM c1 INTO @e
WHILE @@fetch_status = 0
BEGIN
 UPDATE emp_mgr
 SET emp_mgr.NoOfReports = emp_mgr.NoOfReports + 1 -- Add 1 f
 WHERE emp mgr.emp = @e
                                       -- added employee.
 FETCH NEXT FROM c1 INTO @e
END
CLOSE c1
```

DEALLOCATE c1

GO

-- This recursive UPDATE trigger works assuming:

-- 1. Only singleton updates on emp_mgr.

-- 2. No inserts in the middle of the org tree.

CREATE TRIGGER emp_mgrupd ON emp_mgr FOR UPDATE AS

IF UPDATE (mgr)

BEGIN

UPDATE emp_mgr

```
SET emp_mgr.NoOfReports = emp_mgr.NoOfReports + 1 -- Increm
FROM inserted -- (no. of reports) by
```

WHERE emp_mgr.emp = inserted.mgr -- 1 for the new report.

UPDATE emp_mgr

```
SET emp_mgr.NoOfReports = emp_mgr.NoOfReports - 1 -- Decrem
FROM deleted -- (no. of reports) by 1
```

WHERE emp_mgr.emp = deleted.mgr -- for the new report. END

GO

-- Insert some test data rows.

```
INSERT emp_mgr(emp, mgr) VALUES ('Harry', NULL)
INSERT emp_mgr(emp, mgr) VALUES ('Alice', 'Harry')
INSERT emp_mgr(emp, mgr) VALUES ('Paul', 'Alice')
INSERT emp_mgr(emp, mgr) VALUES ('Joe', 'Alice')
INSERT emp_mgr(emp, mgr) VALUES ('Dave', 'Joe')
GO
SELECT * FROM emp_mgr
GO
-- Change Dave's manager from Joe to Harry
UPDATE emp_mgr SET mgr = 'Harry'
WHERE emp = 'Dave'
GO
```

SELECT * FROM emp_mgr GO

Here are the results before the update:

emp	mgr	NoOfReports		
Alice	Harry	2		
Dave	Joe	0		
Harry	NULL	1		
Joe	Alice	1		
Paul	Alice	0		
Here are the results after the update:				

emp	mgr	NoOfReports
Alice	Harry	2
Dave	Harry	0
Harry	NULL	2
Joe	Alice	0
Paul	Alice	0

To set the nested triggers option

⊞ <u>Transact-SQL</u>

Using INSTEAD OF Triggers

INSTEAD OF triggers override the standard actions of the triggering statement (INSERT, UPDATE, or DELETE). For example, an INSTEAD OF trigger can be defined to perform error or value checking on one or more columns, and then perform additional actions before inserting the record. For instance, when the value being updated in an hourly wage column in a payroll table exceeds a specified value, a trigger can be defined to either produce an error message and roll back the transaction, or insert a new record into an audit log before inserting the record into the payroll table. For more information, see <u>Designing INSTEAD</u> <u>OF Triggers</u>.

INSTEAD OF triggers can be defined on either tables or views; however, INSTEAD OF triggers are most useful for extending the types of updates a view can support. For example, INSTEAD OF triggers can provide the logic to modify multiple base tables through a view or to modify base tables that contain these columns:

- timestamp data type
- Computed columns
- Identity columns

INSTEAD OF INSERT Triggers

INSTEAD OF INSERT triggers can be defined on a view or table to replace the standard action of the INSERT statement. Usually, the INSTEAD OF INSERT trigger is defined on a view to insert data into one or more base tables.

Columns in the view select list can be nullable or not nullable. If a view column does not allow nulls, an INSERT statement must provide values for the column. View columns allow nulls if the expression defining the view column includes items such as:

- References to any base table column that allows nulls.
- Arithmetic operators.
- References to functions.
- CASE or COALESCE with a nullable subexpression.
- NULLIF.

You can use the **AllowsNull** property reported by the COLUMNPROPERTY function to determine whether a view column allows nulls. The **sp_help** stored procedure also reports which view columns allow nulls.

An INSERT statement referencing a view that has an INSTEAD OF INSERT trigger must supply values for every view column that does not allow nulls. This includes view columns that reference columns in the base table for which input values cannot be specified:

- Computed columns in the base table.
- Identity columns in the base table for which IDENTITY INSERT is OFF.

• Base table columns with the **timestamp** data type.

If the INSTEAD OF INSERT view trigger generates an INSERT against the base table using the data in the **inserted** table, it must ignore the values for these types of columns by not including the columns in the select list of the INSERT statement. The INSERT statement can generate dummy values for these types of columns.

For example, while an INSERT statement must specify a value for a view column that maps to an identity or computed column in a base table, it can supply a placeholder value. The INSTEAD OF trigger can ignore the value supplied when it forms the INSERT statement that inserts the values into the base table.

These statements create a table, view, and trigger that illustrate the process:

```
CREATE TABLE BaseTable
(PrimaryKey int IDENTITY(1,1)
Color nvarchar(10) NOT NULL,
Material nvarchar(10) NOT NULL,
ComputedCol AS (Color + Material)
)
GO
```

```
--Create a view that contains all columns from the base table.
CREATE VIEW InsteadView
AS SELECT PrimaryKey, Color, Material, ComputedCol
FROM BaseTable
GO
```

```
--Create an INSTEAD OF INSERT trigger on the view.
CREATE TRIGGER InsteadTrigger on InsteadView
INSTEAD OF INSERT
AS
BEGIN
```

```
--Build an INSERT statement ignoring inserted.PrimaryKey and

--inserted.ComputedCol.

INSERT INTO BaseTable

SELECT Color, Material

FROM inserted

END

GO
```

An INSERT statement that refers directly to **BaseTable** cannot supply a value for the **PrimaryKey** and **ComputedCol** columns. For example:

--A correct INSERT statement that skips the PrimaryKey and Compute INSERT INTO BaseTable (Color, Material) VALUES (N'Red', N'Cloth')

--View the results of the INSERT statement. SELECT PrimaryKey, Color, Material, ComputedCol FROM BaseTable

--An incorrect statement that tries to supply a value for the --PrimaryKey and ComputedCol columns. INSERT INTO BaseTable

VALUES (2, N'Green', N'Wood', N'GreenWood')

INSERT statements that refer to **InsteadView**, however, must supply a value for **PrimaryKey** and **ComputedCol**:

--A correct INSERT statement supplying dummy values for the
--PrimaryKey and ComputedCol columns.
INSERT INTO InsteadView (PrimaryKey, Color, Material, ComputedC VALUES (999, N'Blue', N'Plastic', N'XXXXX')
--View the results of the INSERT statement.
SELECT PrimaryKey, Color, Material, ComputedCol
FROM InsteadView The **inserted** table passed to **InsteadTrigger** is built with a nonnullable **PrimaryKey** and **ComputedCol** column; therefore, the INSERT statement referencing the view must supply a value for those columns. The values 999 and N'XXXXX' are passed in to **InsteadTrigger**, but the INSERT statement in the trigger does not select either **inserted.PrimaryKey** or **inserted.ComputedCol**; therefore, the values are ignored. The row actually inserted into **BaseTable** has 2 in **PrimaryKey** and N'BluePlastic' in **ComputedCol**.

The values contained in the **inserted** table for computed, identity, and **timestamp** columns are different for INSTEAD OF INSERT triggers specified on tables compared to an INSTEAD OF triggers specified on views.

Base table column	Value in inserted table in any INSERT trigger on a table	Value in inserted table in an INSTEAD OF INSERT trigger on a view
Is a computed column.	Computed expression	User-specified value or NULL
Has an IDENTITY property.	0 if IDENTITY_INSERT is OFF, specified value if IDENTITY_INSERT is ON	User-specified value or NULL
Has a timestamp data type.	Binary zeros if the column does not allow nulls, NULL if column allows nulls	User-specified value or NULL

An INSERT statement that directly references a base table does not have to supply values for a NOT NULL column that also has a DEFAULT definition. If the INSERT statement does not supply a value, the default value is used. If a NOT NULL column with a DEFAULT definition is referenced by a simple expression in a view that has an INSTEAD OF INSERT trigger, however, any INSERT statement referencing the view must supply a value for the column. This value is required to build the **inserted** table passed to the trigger. A convention is required for a value that signals to the trigger that the default value should be used. The best convention is for the INSERT statement to supply the default value.

The **deleted** table in an INSTEAD OF INSERT trigger is always empty.

See Also

<u>COLUMNPROPERTY</u>

<u>sp_help</u>

INSTEAD OF UPDATE Triggers

INSTEAD OF UPDATE triggers can be defined on a view or table to replace the standard action of the UPDATE statement. Usually, the INSTEAD OF UPDATE trigger is defined on a view to modify data in one or more base tables.

UPDATE statements that reference views with INSTEAD OF UPDATE triggers must supply values for all nonnullable view columns referenced in the SET clause. This includes view columns that reference columns in the base table for which input values cannot be specified, such as:

- Computed columns in the base table.
- Identity columns in the base table for which IDENTITY INSERT is set to OFF.
- Base table columns with the **timestamp** data type.

Usually, when an UPDATE statement that references a table attempts to set the value of a computed, identity, or **timestamp** column, an error is generated because the values for these columns must be determined by Microsoft® SQL Server[™]. These columns must be included in the UPDATE statement to meet the NOT NULL requirement of the column. However, if the UPDATE statement references a view with an INSTEAD OF UPDATE trigger, the logic defined in the trigger can bypass these columns and avoid the error. To do so, the INSTEAD OF UPDATE trigger must not try to update the values for the columns in the base table. This is done by not including the columns in the SET clause of the UPDATE statement. When a record is processed from the **inserted** table, the computed, identity, or **timestamp** column can contain a dummy value to meet the NOT NULL column requirement, but the INSTEAD OF UPDATE trigger ignores those values and the correct values are set by SQL Server.

This solution works because an INSTEAD OF UPDATE trigger does not have to process data from the **inserted** columns that are not updated. In the **inserted**

table passed to an INSTEAD OF UPDATE trigger, the columns specified in the SET clause follow the same rules as the **inserted** columns in an INSTEAD OF INSERT trigger. For columns not specified in the SET clause, the **inserted** table contains the values as they existed before the UPDATE statement was issued. The trigger can test whether a specific column has been updated using the IF UPDATED(*column*) clause. For more information, see <u>INSTEAD OF INSERT</u> <u>Triggers</u>.

INSTEAD OF UPDATE triggers should use values supplied for computed, identity, or **timestamp** columns only in WHERE clause search conditions.

The logic an INSTEAD OF UPDATE trigger on a view should use to process updated values supplied for computed, identity, **timestamp**, or default columns is the same as the logic applied to inserted values for these column types.

Note INSTEAD OF UPDATE triggers cannot be defined on a table that has a foreign key defined with an UPDATE action.

INSTEAD OF DELETE Triggers

INSTEAD OF DELETE triggers can be defined on a view or table to replace the standard action of the DELETE statement. Usually, the INSTEAD OF DELETE trigger is defined on a view to modify data in one or more base tables.

DELETE statements do not specify modifications to existing data values. DELETE statements specify only the rows that are to be deleted. The **inserted** table passed to a DELETE trigger is always empty. The **deleted** table sent to a DELETE trigger contains an image of the rows as they existed before the UPDATE statement was issued. In the case of an INSTEAD OF DELETE trigger on a view or table, the format of the **deleted** table is based on the format of the select list defined for the view.

Note INSTEAD OF DELETE triggers cannot be defined on a table that has a foreign key defined with a DELETE action.

Expressions and Computed Columns in INSTEAD OF Triggers

The select list of a view can have expressions other than simple expressions made up of only a column name. INSTEAD OF triggers on these views must have logic to correctly determine from the values specified on INSERT and UPDATE what values must be set into columns in the base table. Examples of such expressions include:

- View expressions that do not map to any column in any table, such as a constant or some types of functions.
- View expressions that map to multiple columns, such as complex expressions formed by concatenating strings from two or more columns.
- View expressions that transform the value of a single base table column, such as referencing a column in a function.

These issues also apply to view columns that are simple expressions referencing a computed column in a base table. The expression defining the computed column can have the same form as a more complex expression in the view select list.

Views can contain expressions in their select list that do not map to any base table columns, for example:

CREATE VIEW ExpressionView AS SELECT *, GETDATE() AS TodaysDate FROM Northwind.dbo.Employees

Although the **TodaysDate** column does not map to any table column, Microsoft® SQL Server[™] 2000 must build a **TodaysDate** column in the **inserted** table it passes to an INSTEAD OF trigger defined on **ExpressionView**. The **inserted.TodaysDate** column is nullable, however, so an INSERT referencing **ExpressionView** does not have to supply a value for this column. Because the expression does not map to a column in a table, the trigger can ignore any value supplied by the INSERT in this column.

The same approach should be applied to simple view expressions that reference computed columns in base tables that also build a result that is not dependent on other columns, for example:

```
CREATE TABLE ComputedExample
(
PrimaryKey int PRIMARY KEY,
ComputedCol AS SUSER_NAME()
)
```

Some complex expressions map to multiple columns:

```
CREATE TABLE SampleTable

(

PriKey int,

FirstName nvarchar(20),

LastName nvarchar(30)

)

GO

CREATE VIEW ConcatView

AS

SELECT PriKey, FirstName + ' ' + LastName AS CombinedName

FROM SampleTable
```

The expression **CombinedName** in **ConcatView** has the concatenated values of the **FirstName** and **LastName** values. If an INSTEAD OF INSERT trigger is defined on **ConcatView**, you must have a convention for how INSERT statements supply a value for the **CombinedName** column that lets the trigger determine which part of the string should be put in the **FirstName** column and which part should be put in the **LastName** column. If you choose a convention of having INSERT statements specify the value of **CombinedName** using the convention '*first_name*;*last_name*', this trigger can successfully process an

INSERT:

```
CREATE TRIGGER InsteadSample on ConcatView
INSTEAD OF INSERT
AS
BEGIN
```

```
INSERT INTO SampleTable
SELECT PriKey,
-- Pull out the first name string.
SUBSTRING(
CombinedName,
1,
(CHARINDEX(';', CombinedName) - 1)
),
-- Pull out the last name string.
SUBSTRING(
CombinedName,
(CHARINDEX(';', CombinedName) + 1),
DATALENGTH(CombinedName)
)
FROM inserted
END
```

Similar logic is needed to process view columns that are simple expressions referring to computed columns that have complex expressions.

Some view expressions can transform the value of a base table column, for example, by performing a mathematical operation or using the column as a parameter to a function. In this case, the logic in the INSTEAD OF INSERT trigger can take a couple of approaches:

• The convention can be that all INSERT statements supply the raw value to place in the base table, and the trigger logic moves the value from the **inserted** table to the base table.

• The convention can be that all INSERT statements supply the value they expect to have returned by a SELECT on the view, in which case the logic in the trigger must reverse the operation. For example: CREATE TABLE BaseTable

```
(
PrimaryKey int PRIMARY KEY,
ColumnB int,
ColumnC decimal(19,3)
)
```

```
CREATE VIEW SquareView AS
SELECT PrimaryKey, ColumnB,
-- Square the value of ColumnC
SQUARE(ColumnC) AS SquareC
FROM BaseTable
```

```
CREATE TRIGGER SquareTrigger ON SquareView
INSTEAD OF INSERT
AS
BEGIN
INSERT INTO BaseTable
SELECT PrimaryKey, ColumnB,
-- Perform logical inverse of function in view.
SQRT(SquareC)
FROM inserted
END
```

For some expressions, such as complex expressions using mathematical operations like addition and subtraction, it may not be possible for users to supply a value that the trigger can use to unambiguously build values for the destination base table columns. For example, if a view select list contains the expression IntColA + IntColB AS AddedColumns, what does a value of 10 in **inserted.AddedColumns** mean? Is 10 the result of 3 + 7, 2 + 8, or 5 + 5? There is no way to tell from the value of **inserted.AddedColumns** alone what values

should be placed in **IntColA** and **IntColB**.

In these cases, the trigger can be coded to use alternative sources of information to determine the values to set in the base table columns. For views that have INSTEAD OF triggers, the view select list must contain enough information to build values for all non-null columns in the base tables modified by the trigger. Not all data must come directly from the **inserted** table. In some cases, the values in the **inserted** table can be key values that the trigger uses to retrieve the relevant data from other base tables.

Using text, ntext, and image Data in INSTEAD OF Triggers

Data modifications may involve **text**, **ntext**, and **image** columns. In base tables, the value stored in a **text**, **ntext**, or **image** column is a text pointer pointing to the pages holding the data. For more information, see <u>text</u>, <u>ntext</u>, <u>and image Data</u>.

Although AFTER triggers do not support **text**, **ntext**, or **image** data in the **inserted** and **deleted** tables, INSTEAD OF triggers do support them. **text**, **ntext**, and **image** data is stored in the **inserted** and **deleted** tables differently from the way the data is stored in base tables. **text**, **ntext**, and **image** data is not stored as a separate chain of pages. Instead, they are stored as a continuous string within each row, which means there are no text pointers for **text**, **ntext**, or **image** columns in the **inserted** and **deleted** tables. The TEXTPTR and TEXTVALID functions and the READTEXT, UPDATETEXT, and WRITETEXT statements are not valid against **text**, **ntext**, or **image** columns are supported, such as referring to them in select lists, WHERE clause search conditions, or the SUBSTRING, PATINDEX, or CHARINDEX functions. Operations on **text**, **ntext**, or **image** data in the INSTEAD OF triggers are affected by the current SET TEXTSIZE option, which can be determined with the @@TEXTSIZE function.

The type of **text**, **ntext**, or **image** data stored in the **inserted** and **deleted** tables varies depending on the triggering action (INSERT, UPDATE, or DELETE):

- On INSERT statements, the **inserted** table contains the new value for the **text**, **ntext**, or **image** column. The **deleted** table has no rows.
- On DELETE statements, the **inserted** table has no rows and the **deleted** table rows contain the values the **text**, **ntext**, or **image** column had before the DELETE started.
- On UPDATE statements in which the **text**, **ntext**, or **image** value is not changed, both the **inserted** and **deleted** table rows contain the same values for the **text**, **ntext**, or **image** column.

• On UPDATE statements in which the **text**, **ntext**, or **image** value is changed, the **deleted** table contains the data values as they existed before the UPDATE started, and the **inserted** table contains the data with any modifications specified in the SET clause.

If an INSERT, UPDATE, or DELETE statement modifies many rows with large **text**, **ntext**, or **image** values, considerable memory can be required to hold the copies of the **text**, **ntext**, or **image** data in the **inserted** and **deleted** tables. Copying these large amounts of data can also lower performance. INSERT, UPDATE, and DELETE statements that reference views or tables that have INSTEAD OF triggers should modify one row at a time, or only a few rows at a time, whenever possible.

Activating Triggers with Implicit and Explicit Null Values

Inserting an explicit null value into a column or using the DEFAULT keyword to assign a value to a column activates the trigger as expected. Similarly, when no value is specified in the INSERT statement for a column, the trigger is still activated when:

- An implicit null value is inserted into a column because no DEFAULT definition exists.
- A default value is inserted into a column because a DEFAULT definition does exist.

Examples

A. Test trigger activation with null and default values

The following examples show how a trigger is affected by implicit and explicit null values. A small table is created to hold two integer values. One column can contain null values; the other column contains a default value. A trigger evaluates whether the both columns are modified, and displays a message when the trigger is activated. A series of INSERT statements tests trigger activation by inserting combinations of implicit and explicit null values.

CREATE TABLE t1 (a int NULL, b int NOT NULL DEFAULT 99) GO

CREATE TRIGGER t1trig ON t1 FOR INSERT, UPDATE AS IF UPDATE(a) AND UPDATE(b) PRINT 'FIRING' GO

--When two values are inserted, the UPDATE is TRUE for both colum INSERT t1 (a, b) VALUES (1, 2)

--When two values are updated, the UPDATE is TRUE for both colum UPDATE t1 SET a = 1, b = 2

--When an explicit NULL is inserted in column a, the UPDATE is TRU INSERT t1 VALUES (NULL, 2)

--When an explicit NULL is updated in column a, the UPDATE is TRU UPDATE t1 SET a = NULL, b = 2

--When an implicit NULL is inserted in column a, the UPDATE is TRU INSERT t1 (b) VALUES (2)

--When column a is updated with an implicit NULL, the UPDATE is F UPDATE t1 SET b = 2

--When the default value is implicitly inserted in column b, the UPDA INSERT t1 (a) VALUES (2)

--When column b is updated with an implicit NULL, the UPDATE is F UPDATE t1

SET a = 2

--When the default value is explicitly inserted in column b, the UPDAT INSERT t1 (a, b) VALUES (2, DEFAULT)

--When column b is updated explicitly with the default value, the UPD UPDATE t1 SET a = 2, b = DEFAULT

See Also

DEFAULT Definitions

Null Values

Modifying and Renaming a Trigger

If you must modify the definition of a trigger, you can either drop and re-create the trigger or redefine the existing trigger in a single step.

If you change the name of an object referenced by a trigger, you must modify the trigger so that its text reflects the new name. Therefore, before renaming an object, display the dependencies of the object first to determine if any triggers are affected by the proposed change.

You can also rename a trigger. The new name must follow the rules for identifiers. You can rename only the triggers that you own, but the database owner can change the name of any user's triggers. The trigger to be renamed must be in the current database.

A trigger can also be modified to encrypt its definition.

To modify a trigger

⊞<u>Transact-SQL</u>

Viewing a Trigger

In Microsoft® SQL Server[™] 2000, you can determine the types of triggers on a table, the name of the trigger, its owner, and the date it was created.

You can also:

- Gain information about the definition of a trigger if it was not encrypted when created or modified. You may need to see the definition of the trigger to see its Transact-SQL statements or to understand how it affects the table upon which it is defined.
- List the objects used by the specified trigger. This information can be used to identify the objects that affect the trigger if they are changed or deleted in the database.

The result set of **sp_helptrigger** contains the columns **isafter** and **isinsteadof** to report whether a trigger is an AFTER or INSTEAD OF trigger. The OBJECTPROPERTY function reports whether a trigger is an AFTER or INSTEAD OF trigger through the **ExecIsInsteadOfTrigger** and **ExecIsAfterTrigger** properties.

To view the types of triggers on a table

⊞ <u>Transact-SQL</u>

∃ <u>SQL-DMO</u>

To view information about a trigger

∃ <u>Transact-SQL</u>

∃ <u>SQL-DMO</u>

To view a trigger

⊞ <u>Transact-SQL</u>

Deleting a Trigger

When a trigger is no longer needed, you can delete it. When a trigger is deleted, the table and the data upon which it is based are not affected. Deleting a table automatically deletes any triggers on the table. Permissions to delete a trigger default to the owner of the table upon which the trigger is defined.

To delete a trigger

∃ <u>Transact-SQL</u>

User-Defined Functions

Functions are subroutines made up of one or more Transact-SQL statements that can be used to encapsulate code for reuse. Microsoft® SQL Server[™] 2000 does not limit users to the built-in functions defined as part of the Transact-SQL language, but allows users to create their own user-defined functions.

User-defined functions are created using the CREATE FUNCTION statement, modified using the ALTER FUNCTION statement, and removed using the DROP FUNCTION statement. Each fully qualified user-defined function name (*database_name.owner_name.function_name*) must be unique.

You must have been granted CREATE FUNCTION permissions to create, alter, or drop user-defined functions. Users other than the owner must be granted appropriate permissions on a function before they can use it in a Transact-SQL statement. To create or alter tables with references to user-defined functions in the CHECK constraint, DEFAULT clause, or computed column definition, you must also have REFERENCES permission on the functions.

Transact-SQL errors that cause a statement to be canceled and continue with the next statement in the module (such as triggers or stored procedures) are treated differently inside a function. In functions, such errors cause the execution of the function to stop. This in turn causes the statement that invoked the function to be canceled.

Types of User-Defined Functions

SQL Server 2000 supports three types of user-defined functions:

- Scalar functions
- Inline table-valued functions
- Multistatement table-valued functions

A user-defined function takes zero or more input parameters and returns either a scalar value or a table. A function can have a maximum of 1024 input

parameters. When a parameter of the function has a default value, the keyword default DEFAULT must be specified when calling the function to get the default value. This behavior is different from parameters with default values in stored procedures in which omitting the parameter also implies the default value. User-defined functions do not support output parameters.

Scalar functions return a single data value of the type defined in a RETURNS clause. All scalar data types, including **bigint** and **sql_variant**, can be used. The **timestamp** data type, user-defined data type, and nonscalar types, such as **table** or **cursor**, are not supported. The body of the function, defined in a BEGIN...END block, contains the series of Transact-SQL statements that return the value. The return type can be any data type except **text**, **ntext**, **image**, **cursor**, and **timestamp**.

Table-valued functions return a **table**. For an inline table-valued function, there is no function body; the table is the result set of a single SELECT statement. For a multistatement table-valued function, the function body, defined in a BEGIN...END block, contains the TRANSACT-SQL statements that build and insert rows into the table that will be returned. For more information about inline table-valued functions, see <u>Inline User-Defined Functions</u>. For more information about table-valued functions, see <u>User-Defined Functions</u> That Return a table <u>Data Type</u>.

The statements in a BEGIN...END block cannot have any side effects. Function side effects are any permanent changes to the state of a resource that has a scope outside the function such as a modification to a database table. The only changes that can be made by the statements in the function are changes to objects local to the function, such as local cursors or variables. Modifications to database tables, operations on cursors that are not local to the function, sending e-mail, attempting a catalog modification, and generating a result set that is returned to the user are examples of actions that cannot be performed in a function.

The types of statements that are valid in a function include:

- DECLARE statements can be used to define data variables and cursors that are local to the function.
- Assignments of values to objects local to the function, such as using SET to assign values to scalar and table local variables.

- Cursor operations that reference local cursors that are declared, opened, closed, and deallocated in the function. FETCH statements that return data to the client are not allowed. Only FETCH statements that assign values to local variables using the INTO clause are allowed.
- Control-of-flow statements.
- SELECT statements containing select lists with expressions that assign values to variables that are local to the function.
- UPDATE, INSERT, and DELETE statements modifying **table** variables that are local to the function.
- EXECUTE statements calling an extended stored procedure.

The number of times that a function specified in a query is actually executed can vary between execution plans built by the optimizer. An example is a function invoked by a subquery in a WHERE clause. The number of times the subquery and its function is executed can vary with different access paths chosen by the optimizer.

Built-in functions that can return different data on each call are not allowed in user-defined functions. The built-in functions not allowed in user-defined functions are:

@@CONNECTIONS	@@PACK_SENT	GETDATE
@@CPU_BUSY	@@PACKET_ERRORS	GetUTCDate
@@IDLE	@@TIMETICKS	NEWID
@@IO_BUSY	@@TOTAL_ERRORS	RAND
@@MAX_CONNECTIONS	@@TOTAL_READ	TEXTPTR
@@PACK_RECEIVED	@@TOTAL_WRITE	

Schema-Bound Functions

CREATE FUNCTION supports a SCHEMABINDING clause that binds the function to the schema of any objects it references, such as tables, views, and other user-defined functions. An attempt to alter or drop any object referenced by a schema-bound function fails.

These conditions must be met before you can specify SCHEMABINDING in CREATE FUNCTION:

- All views and user-defined functions referenced by the function must be schema-bound.
- All objects referenced by the function must be in the same database as the function. The objects must be referenced using either one-part or two-part names.
- You must have REFERENCES permission on all objects (tables, views, and user-defined functions) referenced in the function.

You can use ALTER FUNCTION to remove the schema binding. The ALTER FUNCTION statement should redefine the function without specifying WITH SCHEMABINDING.

Calling User-Defined Functions

When calling a scalar user-defined function, you must supply at least a two-part name:

SELECT *, MyUser.MyScalarFunction() FROM MyTable

Table-valued functions can be called by using a one-part name:

SELECT *
FROM MyTableFunction()

However, when you call SQL Server built-in functions that return a table, you

must add the prefix **::** to the name of the function:

SELECT * FROM ::fn_helpcollations()

A scalar function can be referenced any place an expression of the same data type returned by the function is allowed in a Transact-SQL statement, including computed columns and CHECK constraint definitions. 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),
BrickLength decimal(4,1),
BrickWidth decimal(4,1),
BrickVolume AS
(
dbo.CubicVolume(BrickHeight,
BrickLength, BrickWidth)
)
```

dbo.CubicVolume is an example of a user-defined function that returns a scalar value. The RETURNS clause defines a scalar data type for the value returned by the function. The BEGIN...END block contains one or more Transact-SQL statements that implement the function. Each RETURN statement in the function must have an argument that returns a data value that has the data type specified in the RETURNS clause, or a data type that can be implicitly converted to the type specified in RETURNS. The value of the RETURN argument is the value returned by the function.

Obtaining Information About Functions

Several catalog objects report information about user-defined functions:

- **sp_help** reports information about user-defined functions.
- **sp_helptext** reports the source of user-defined functions.

Three information schema views report information about user-defined functions: **ROUTINES**, **PARAMETERS**, and **ROUTINE_COLUMNS**. These information schema views also report information for stored procedures.

User-Defined Functions That Return a table Data Type

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. While views are limited to a single SELECT statement, 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, but stored procedures that return result sets cannot. For example, assume that **fn_EmployeesInDept** is a user-defined function that returns a **table** and can be invoked by a SELECT statement such as:

```
SELECT * FROM tb_Employees AS E INNER JOIN
dbo.fn_EmployeesInDept('shipping') AS EID
ON E.EmployeeID = EID.EmployeeID
```

In a user-defined function that returns a **table**:

- The RETURNS clause defines a local return variable name for the table returned by the function. The RETURNS clause also defines the format of the table. The scope of the local return variable name is local within the function.
- The Transact-SQL statements in the function body build and insert rows into the return variable defined by the RETURNS clause.
- When a RETURN statement is executed, the rows inserted into the variable are returned as the tabular output of the function. The RETURN statement cannot have an argument.

No Transact-SQL statements in a function that returns a **table** can return a result

set directly to a user. The only information the function can return to the user is the **table** returned by the function.

This example creates a function in the **Northwind** database that returns 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 body insert rows into the variable **@OrderShipperTab** to build the table result returned by the function.

This query references the table returned by the function in its FROM clause:

SELECT * FROM LargeOrderShippers(\$500)

Note The **text in row** table option is automatically set to 256 for a table returned by a user-defined function. This cannot be changed. The READTEXT,

WRITETEXT, and UPDATETEXT statements cannot be used to read or write parts of any **text**, **ntext**, or **image** columns in the table. For more information, see <u>Text in Row Data</u>.

Inline User-Defined Functions

Inline user-defined functions are a subset of user-defined functions that return a **table**. Inline functions can be used to achieve the functionality of parameterized views.

Consider this view:

```
CREATE VIEW vw_CustomerNamesInWA AS
SELECT CustomerID, CompanyName
FROM Northwind.dbo.Customers
WHERE Region = 'WA'
```

You can create a more generalized version, **vw_CustomerNamesInRegion**, by replacing the WHERE Region = 'WA' with a WHERE Region = @RegionParameter and letting users specify the region they are interested in viewing. Views, however, do not support parameters in the search conditions specified in the WHERE clause.

Inline user-defined functions can be used to support parameters in the search conditions specified in the WHERE clause. This is an example of a function that allows users to specify the region in their select:

```
CREATE FUNCTION fn_CustomerNamesInRegion
(@RegionParameter nvarchar(30))
RETURNS table
AS
RETURN (
SELECT CustomerID, CompanyName
FROM Northwind.dbo.Customers
WHERE Region = @RegionParameter
)
GO
-- Example of calling the function for a specific region
SELECT *
```

FROM fn_CustomerNamesInRegion(N'WA') GO

Inline user-defined functions follow these rules:

- The RETURNS clause contains only the keyword **table**. You do not have to define the format of a return variable because it is set by the format of the result set of the SELECT statement in the RETURN clause.
- There is no *function_body* delimited by BEGIN and END.
- The RETURN clause contains a single SELECT statement in parentheses. The result set of the SELECT statement forms the table returned by the function. The SELECT statement used in an inline function is subject to the same restrictions as SELECT statements used in views.

Inline functions can also be used to increase the power of indexed views. The indexed view itself cannot use parameters in its WHERE clause search conditions to tailor the stored result set to specific users. You can, however, define an indexed view that stores the complete set of data that matches the view, and then define an inline function over the indexed view that contains parameterized search conditions that allow users to tailor their results. If the view definition is complex, most of the work performed to build a result set involves operations such as building aggregates or joining several tables when the clustered index is created on the view. If you then create an inline function that references the view, the function can apply the user's parameterized filters to pull specific rows from the result set that was built by the CREATE INDEX statement. The complex aggregations and joins are done once, at CREATE INDEX time, and all subsequent queries referencing the inline function filter rows from the simplified, stored result set. For example:

1. You define a view **vw_QuarterlySales** that aggregates all sales data into a result set that reports summarized sales data by quarter for all stores.

- 2. You create a clustered index on **vw_QuarterlySales** to materialize a result set containing the summarized data.
- 3. You create an inline function to filter the summarized data: CREATE FUNCTION fn_QuarterlySalesByStore

```
(
@StoreID int
)
RETURNS table
AS
RETURN (
SELECT *
FROM SalesDB.dbo.vw_QuarterlySales
WHERE StoreID = @StoreID
)
```

 Users can then get the data for their specific store by selecting from the inline function: SELECT * FROM fn_QuarterlySalesByStore(14432)

Most of the work needed to satisfy the queries issued at Step 4 is to aggregate the sales data by quarter. This work is done once at Step 2. Each individual SELECT statement in Step 4 uses the function **fn_QuarterlySalesByStore** to filter out the aggregated data specific to their store.

Deterministic and Nondeterministic Functions

All functions are deterministic or nondeterministic:

- Deterministic functions always return the same result any time they are called with a specific set of input values.
- Nondeterministic functions may return different results each time they are called with a specific set of input values.

Whether a function is deterministic or nondeterministic is called the determinism of the function.

For example, the DATEADD built-in function is deterministic because it always returns the same result for any given set of argument values for its three parameters. GETDATE is not deterministic because it is always invoked with the same argument, yet the value it returns changes each time it is executed.

Earlier versions of Microsoft® SQL Server[™] have no functionality that is dependent on the determinism of functions. In Microsoft SQL Server 2000, nondeterministic functions cannot be specified in two types of Transact-SQL expressions:

- An index cannot be created on a computed column if the *computed_column_expression* references any nondeterministic functions.
- A clustered index cannot be created on a view if the view references any nondeterministic functions.

One of the properties SQL Server records for user-defined functions is whether the function is deterministic. A nondeterministic user-defined function cannot be invoked by either a view or computed column if you want to create an index on the view or computed column.

User-Defined Function Determinism

Whether a user-defined function is deterministic or nondeterministic depends on how the function is coded. User-defined functions are deterministic if:

- The function is schema-bound.
- All built-in or user-defined functions called by the user-defined function are deterministic.
- The body of the function references no database objects outside the scope of the function. For example, a deterministic function cannot reference tables other than table variables that are local to the function.
- The function does not call any extended stored procedures.

User-defined functions that do not meet these criteria are marked as nondeterministic. Built-in nondeterministic functions are not allowed in the body of user-defined functions.

Built-in Function Determinism

You cannot influence the determinism of any built-in function. Each built-in function is deterministic or nondeterministic based on how the function is implemented by Microsoft SQL Server.

All of the aggregate and string built-in functions are deterministic except the string functions CHARINDEX and PATINDEX. For a list of these functions, see <u>Aggregate Functions</u> and <u>String Functions</u>.

These built-in functions from categories of built-in functions other than aggregate and string functions are always deterministic:

ABS	DATEDIFF	PARSENAME
ACOS	DAY	POWER
ASIN	DEGREES	RADIANS
ATAN	EXP	ROUND
ATN2	FLOOR	SIGN
CEILING	ISNULL	SIN

COALESCE	ISNUMERIC	SQUARE
COS	LOG	SQRT
COT	LOG10	TAN
DATALENGTH	MONTH	YEAR
DATEADD	NULLIF	

These functions are not always deterministic but can be used in indexed views or indexes on computed columns when they are specified in a deterministic manner.

Function	Comments
CAST	Deterministic unless used with datetime , smalldatetime ,
	or sql_variant .
CONVERT	Deterministic unless used with datetime , smalldatetime ,
	or sql_variant . The datetime and smalldatetime data
	types are deterministic if the style parameter is also
	specified.
CHECKSUM	Deterministic, with the exception of CHECKSUM(*).
ISDATE	Deterministic only if used with the CONVERT function,
	the CONVERT style parameter is specified and style is
	not equal to 0, 100, 9, or 109.
RAND	RAND is deterministic only when a <i>seed</i> parameter is
	specified.

All of the configuration, cursor, meta data, security, and system statistical functions are nondeterministic. For a list of these functions, see <u>Configuration</u> <u>Functions</u>, <u>Cursor Functions</u>, <u>Meta Data Functions</u>, <u>Security Functions</u>, and <u>System Statistical Functions</u>.

These built-in functions from other categories are always nondeterministic:

@@ERROR	FORMATMESSAGE	NEWID
@@IDENTITY	GETANSINULL	PATINDEX
@@ROWCOUNT	GETDATE	PERMISSIONS
@@TRANCOUNT	GetUTCDate	SESSION_USER

APP_NAME	HOST_ID	STATS_DATE
CHARINDEX	HOST_NAME	SYSTEM_USER
CURRENT_TIMESTAMP	IDENT_INCR	TEXTPTR
CURRENT_USER	IDENT_SEED	TEXTVALID
DATENAME	IDENTITY	USER_NAME

Calling Extended Stored Procedures from Functions

Functions that call extended stored procedures are nondeterministic because the extended stored procedures can cause side effects on the database. Side effects are changes to a global state of the database, such as an update to a table, or to an external resource, such as a file or the network (for example, modifying a file or sending an e-mail message). You should not rely on returning a consistent result set when executing an extended stored procedure from a user-defined function. User-defined functions that create side effects on the database are not recommended.

When called from inside a function, the extended stored procedure cannot return result sets to the client. Any Open Data Services API that returns result sets to the client will have a return code of FAIL.

The extended stored procedure can connect back to SQL Server; however, the procedure cannot join the same transaction as the original function that invoked the extended stored procedure.

Similar to invocations from a batch or stored procedure, the extended stored procedure is executed in the context of the Microsoft Windows® security account under which SQL Server is running. The owner of the extended stored procedure should consider this when granting permissions to other users to execute the procedure.

Rewriting Stored Procedures as Functions

This topic describes how to determine whether to rewrite existing stored procedure logic as user-defined functions. For example, if you want to invoke a stored procedure directly from a query, repackage the code as a user-defined function.

In general, if the stored procedure returns a (single) result set, define a tablevalued function. If the stored procedure computes a scalar value, define a scalar function.

Criteria for Table-Valued Functions

If a stored procedure meets the following criteria, it is a good candidate for being rewritten as a table-valued function:

- The logic is expressible in a single SELECT statement but is a stored procedure, rather than a view, only because of the need for parameters. This scenario can be handled with an inline table-valued function.
- The stored procedure does not perform update operations (except to table variables).
- There is no need for dynamic EXECUTE statements
- The stored procedure returns one result set.
- The primary purpose of the stored procedure is to build intermediate results that are to be loaded into a temporary table, which is then queried in a SELECT statement. INSERT...EXEC statements can be written using table-valued functions. For example, consider the following sequence:
 INSERT #temp EXEC sp_getresults SELECT ...

FROM #temp, t1 WHERE ...

The **sp_getresults** stored procedure can be rewritten as a table-valued function, for example **fn_results()**, which means the preceding statements can be rewritten as:

SELECT ... FROM fn_results(), t1 WHERE ...

Using Extended Properties on Database Objects

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 the way 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** value that can contain up to 7,500 bytes of data. Individual database objects can have multiple extended properties.

Possible uses of extended properties include:

- Specifying a caption for a table, view, or column. All applications can then use the same caption in a user interface that displays information from that table, view, or column.
- Specifying an input mask for a column so that all applications can validate data before executing a Transact-SQL statement.
- Specifying formatting rules for displaying the data in a column.
- Recording a description of specific database objects that applications can display to users.
- Specifying the size and window location at which a column should be displayed.

For the purposes of specifying extended properties, the objects in a SQL Server 2000 database are classified into three levels (0, 1, 2). Level 0 is the highest level and 2 is the lowest level. The table lists the level-0 objects, user and user-defined data type, with their valid level-1 and level-2 objects.

Level 0	Level 1	Level 2
User	Table	Column, index, constraint,
		trigger
	View	Column, INSTEAD OF
		trigger
	Schema-bound view	Column, index, INSTEAD OF
		trigger
	Stored procedure	Parameter
	Rule	<none></none>
	Default	<none></none>
	Function	Column, parameter,
		constraint,
	Schema-bound	Column, parameter, constraint
	function	
User-defined data	<none></none>	<none></none>
type		

Extended properties are not supported on objects that are not listed as level 0, 1, or 2 objects.

References to an object in one level must be qualified with the names of the higher level objects that own or contain them. For example, when referencing a column (level 2) you must also specify the table (level 1) that contains the column and the user (level 0) who owns the table.

Extended properties provide only a named location in which to store data. All applications must be coded to query the property and take appropriate action. For example, adding a caption property to a column does not create a caption that can be displayed by an application. Each application must be coded to read the caption and display it properly.

Extended properties are managed using three system stored procedures:

• sp_addextendedproperty

Adds a new extended property to a database object.

• sp_updateextendedproperty

Updates the value of an existing extended property.

• sp_dropextendedproperty

Drops an existing extended property.

You can retrieve the value of an existing extended property using the system function **FN_LISTEXTENDEDPROPERTY**.

The following is an example of a table that has:

- Caption extended properties for the table and the columns.
- Input-mask extended properties for the columns.

```
USE Northwind
GO
CREATE TABLE TestExProp
         int PRIMARY KEY IDENTITY(1,1),
 (PriKey
 USPhoneNmbr
               char(13)
   CHECK (USPhoneNmbr LIKE
       ),
 USSocialScrty char(11)
   CHECK (USSocialScrty LIKE
       )
)
GO
sp_addextendedproperty 'Caption', 'Caption Test Table',
          'user', dbo, 'table', TestExProp
GO
sp_addextendedproperty 'Caption', 'Primary Key',
          'user', dbo, 'table', TestExProp, 'column', PriKey
GO
sp_addextendedproperty 'Input Mask', '(NNN)NNN-NNNN',
```

'user', dbo, 'table', TestExProp, 'column', USPhoneNmbr GO sp_addextendedproperty 'Caption', 'US Phone Number', user', dbo, 'table', TestExProp, 'column', USPhoneNmbr GO sp_addextendedproperty 'Input Mask', 'NNN-NN-NNNN', user', dbo, 'table', TestExProp, 'column', USSocialScrty GO sp_addextendedproperty 'Caption', 'US Social Security Number', user', dbo, 'table', TestExProp, 'column', USSocialScrty GO

This statement updates the primary-key caption property:

This statement drops the input-mask properties:

sp_dropextendedproperty 'Input Mask', 'user', dbo, 'table', TestExProp, 'column', USSocialScrty

GO

sp_dropextendedproperty 'Input Mask', 'user', dbo, 'table', TestExProp, 'column', USPhoneNmbr

GO

This statement retrieves the table-caption property:

SELECT * FROM ::FN_LISTEXTENDEDPROPERTY('Caption', 'User','dbo','tab 'TestExProp', default, default)

The example shows using CHECK constraints and an input-mask property to specify the pattern of data for each column. Most sites choose one or the other

unless:

- The CHECK constraints were used as an interim measure until all the programs dealing with this table could be changed to use the input mask properties.
- The site also supports users who can update the data through ad hoc tools that do not read the extended properties.

The advantage of the input mask over the CHECK constraint is that the logic is applied in the applications, which can generate more informative errors if a user provides improperly formatted data. The disadvantage of the input mask is that it requires a separate call to **fn_listextendedproperty** to obtain the property, and the logic to enforce the mask must be added in all programs.

See Also

fn_listextendedproperty Property Management sp_addextendedproperty sp_dropextendedproperty sp_updateextendedproperty

Full-Text Indexes

Full-text support for Microsoft[®] SQL Server[™] 2000 data involves two features: the ability to issue queries against character data, and the creation and maintenance of the underlying indexes facilitating these queries.

Regular SQL indexes	Full-text indexes
Stored under the control of the	Stored in the file system, but
database in which they are defined.	administered through the database.
Several regular indexes per table	Only one full-text index per table is
are allowed.	allowed.
Updated automatically when the	Addition of data to full-text indexes,
data upon which they are based is	called population, can be requested
inserted, updated, or deleted.	through either a schedule or a specific
	request, or can occur automatically
	with the addition of new data.
Not grouped.	Grouped within the same database into
	one or more full-text catalogs.
Created and dropped using SQL	Created, managed, and dropped using
Server Enterprise Manager,	SQL Server Enterprise Manager,
wizards, or Transact-SQL	wizards, or stored procedures.
statements.	

Full-text indexes differ from regular SQL indexes in a number of ways.

These differences make a number of administrative tasks necessary. Full-text administration is carried out at several levels:

• Server

Certain server-wide properties, such as **resource_usage**, can be set to increase and reduce the amount of system resources used by the full-text service.

Note The full-text engine runs as a service named Microsoft Search on Microsoft Windows NT® Server and Microsoft Windows® 2000 Server. The Microsoft Search service is not available for Microsoft SQL

Server Personal Edition. Although this means the Microsoft Search service is not installed on Microsoft Windows 95 or Windows 98, Windows NT Workstation, or Windows 2000 Professional clients, these clients can make use of the service when they are connected to an instance of SQL Server Standard or Enterprise edition.

• Database

A database must be enabled to use the full-text service. Meta data for one or more full-text catalogs can be created and dropped in an enabled database.

• Full-text catalog

A full-text catalog contains full-text indexes in a database. Each catalog can serve the indexing needs of one or more tables within a database. The catalog is populated with indexes using the administrative facilities described here. (Full-text catalogs must reside on a local hard drive associated with the instance of SQL Server. Removable drives, floppy disks, and network drives are not supported.) A maximum of 256 full-text catalogs can be created on each server.

Note Full-text indexing is fully supported in a Windows NT failover cluster environment. For more information, see <u>Running Full-Text</u> <u>Queries with Failover Clustering</u>.

• Table

A table must first be enabled for full-text support. Then meta data, such as the name of the table and its full-text catalog, is created for the fulltext index associated with the table. After the table is enabled, you can populate it with the data in columns enabled for full-text support. If the full-text definition for a table is changed (for example, by including a new column that will also be indexed for a full-text search), the associated full-text catalog must be repopulated to synchronize the fulltext index with the new full-text definition.

• Column

Columns that support full-text queries can be added or dropped from an inactive registered table.

At all these levels, facilities are available to retrieve meta data and status information.

Like regular SQL indexes, full-text indexes can be automatically updated as data is modified in the associated tables. Alternatively, full-text indexes can be repopulated manually at appropriate intervals. This repopulation can be timeconsuming and resource-intensive; therefore, it is an asynchronous process that usually runs in the background during periods of low database activity.

Tables with the same update characteristics (such as small number of changes versus large number of changes, or tables that change frequently during a particular time of day) should be grouped together and assigned to the same full-text catalog. By setting up full-text catalog population schedules in this way, full-text indexes stay synchronous with the tables without adversely affecting the resource usage of the database server during periods of high database activity.

It is important to plan the placement of full-text indexes for tables in full-text catalogs. When you assign a table to a full-text catalog, consider the following guidelines:

- Always select the smallest unique index available for your full-text unique key. (A 4-byte, integer-based index is optimal.) This reduces the resources required by Microsoft Search service in the file system significantly. If the primary key is large (over 100 bytes), consider choosing another unique index in the table (or creating another unique index) as the full-text unique key. Otherwise, if the full-text unique key size reaches the maximum size allowed (450 bytes), full-text population will not be able to proceed.
- If you are indexing a table that has millions of rows, assign the table to its own full-text catalog.
- Consider the amount of change occurring in the tables being full-text indexed, as well as the number of table rows. If the total number of rows being changed, together with the numbers of rows in the table present during the last full-text population, represents millions of rows, assign the table to its own full-text catalog.

See Also

<u>sp_fulltext_table</u>

Administering Full-Text Features Using SQL Enterprise Manager

You can administer full-text indexes using the Full-Text Indexing Wizard or shortcut menus in SQL Server Enterprise Manager.

To enable a database for full-text indexing

Administering Full-Text Features Using Stored Procedures and Scalar Functions

Administering full-text features using stored procedures and scalar functions is described through examples provided in these topics:

- Enabling Others to Issue Full-Text Queries
- <u>Performing Investigation and Clean-up Tasks for Full-Text Catalogs</u>
- <u>Performing Infrequent Tasks</u>

The examples in these topics use two fictitious tables, **writers** and **books**, in the **pubs** database. This table shows the **writers** table and the columns that are candidates for full-text indexing, specifically the **varchar** data type.

		Enable full-text
Column name	Data type	features?
writer_name	char(40)	No
citizenship	char(40)	No
organization	varchar(100)	Yes
royalties_ytd	decimal(7,2)	No
Royalyties_lifetime	decimal(7,2)	No
Bio	varchar(500)	Yes
writer_id	integer (with a unique value enforced by the writer_id_index index)	No
last_changed	timestamp	No

This table shows the **books** table and the columns that are candidates for full-text indexing, specifically the **varchar** and **text** data types.

		Enable full-text
Column name	Data type	features?

writer_name	char(40)	No
Title	varchar(120)	Yes
Size	smallint	No
pub_date	datetime	No
the_words	text	Yes
Abstract	varchar(500)	Yes
isbn	char(15) (with a unique value	No
	enforced by the isbn_index index)	
last_changed	timestamp	No

For more information about both querying and indexing, see <u>Example of</u> <u>Combining Full-Text Administration and Full-Text Query</u>.

Enabling Others to Issue Full-Text Queries

This example demonstrates how to use SQL Server Service Manager to enable others to issue full-text queries against the **writers** and **books** tables. The **writers** and **books** tables are fictitious tables located in the **pubs** database. For a description of the tables, see Administering Full-Text Features Using Stored Procedures and Scalar Functions_administering_full-text_features_using_stored_procedures_and_scalar_functions

1. Use SQL Server Service Manager to verify the full-text service, Microsoft Search, is running.

If necessary, the service can be started and stopped in one of these ways:

- Use the shortcut menu of the **Full-Text Search** service in SQL Server Enterprise Manager.
- Use Microsoft Search service in SQL Server Service Manager.
- Type **net start mssearch** from a command prompt.
- Find out if the **pubs** database has been enabled for full-text processing by executing this statement: SELECT DatabaseProperty ('Pubs', 'IsFulltextEnabled')

For more information, see **DATABASEPROPERTY**.

This returns 1 if full-text support has been enabled, and 0 if it has not.

3. If not enabled (which is the default for newly created databases), enable the **pubs** database for full-text processing. Using **pubs**, execute this stored procedure: sp_fulltext_database 'enable'

For more information, see sp_fulltext_database.

4. Create a full-text catalog named **PubsCatalog**, opting for the default directory, by executing this stored procedure in the **pubs** database: sp_fulltext_catalog 'PubsCatalog', 'create'

For more information, see <u>sp_fulltext_catalog</u>.

This creates meta data about a full-text catalog in the system tables of the database and builds an empty full-text catalog in the file system.

 Register the writers and books tables for full-text processing by executing this stored procedure once for each table: sp_fulltext_table 'writers', 'create', 'PubsCatalog', 'writer_id_index' sp_fulltext_table 'books', 'create', 'PubsCatalog', 'isbn_index'

For more information, see <u>sp_fulltext_table</u>.

Both tables use the **PubsCatalog** full-text catalog. These stored procedure calls create meta data about both full-text indexes.

6. For each table, specify the names of the columns that are to support full-text queries by executing this stored procedure once for each column:

sp_fulltext_column 'writers', 'organization', 'add' sp_fulltext_column 'writers', 'bio', 'add' sp_fulltext_column 'books', 'writer_name', 'add' sp_fulltext_column 'books', 'the words', 'add' sp_fulltext_column 'books', 'abstract', 'add'

For more information, see <u>sp_fulltext_column</u>.

Note A mistake was made for the sake of illustration: for the **books** table, the **writer_name** column, rather than the **titles** column, has been registered.

These stored procedure calls augment meta data about both full-text indexes.

7. Create a full-text index for these tables by executing this stored

procedure once for each table: sp_fulltext_table 'writers', 'activate' sp_fulltext_table 'books', 'activate'

This does not actually create the full-text indexes. Rather, it registers the tables in the full-text catalog so that data from these tables will be included in the next population.

 Start a full population of the **PubsCatalog** full-text catalog by executing this stored procedure: sp_fulltext_catalog 'PubsCatalog', 'start_full'

Because the population of a full-text catalog is an asynchronous operation, it is unlikely that the full-text indexes are created immediately.

 Verify the progress of the population of the **PubsCatalog** full-text catalog by executing this statement: SELECT FulltextCatalogProperty ('PubsCatalog', 'PopulateS')

For more information, see <u>FULLTEXTCATALOGPROPERTY</u>.

This returns 0 if the service is idle for the full-text catalog and therefore finished, and 1 or more to indicate the stage of population.

10. Issue Transact-SQL queries to confirm that the administration was executed correctly. For example: SELECT B.writer_name, B.pub_date, B.the_words, A.royalti¢ FROM writers A, books B WHERE A.writer_name = B.writer_name AND A.citizenship = 'Canadian' AND CONTAINS (B.the_words, '"Indexing Service" NEAR '

SELECT writer_name, pub_date, abstract FROM books WHERE CONTAINS (title, "Classic" NEAR "French" NEAF This last query results in an error because the **title** column was not enabled for full-text queries.

 Check for errors by executing this statement: SELECT ColumnProperty (ObjectId('books'), 'title', 'IsFullTe

For more information, see <u>COLUMNPROPERTY</u>.

This returns 1 if the **title** column is part of the full-text index for the **books** table, and 0 if it is not.

12. List the columns participating in full-text processing for the **books** table by executing this stored procedure:sp_help_fulltext_columns 'books'

For more information, see <u>sp_help_fulltext_columns</u>.

Note The results of this query show there was a mistake and that the **writer_name** column, rather than the **title** column, was included in the full-text index definition.

13. Deactivate the **books** table so that the **title** column can be added to the full-text index and the **writer_name** column can be removed by executing this stored procedure: sp_fulltext_table 'books', 'deactivate'

In addition to allowing columns to be added and deleted, deactivating the **books** table means the table no longer participates in the population of the **PubsCatalog** full-text catalog. However, the meta data remains and the table can be reactivated. The existing full-text index for the **books** table remains in place until the next full population of the **PubsCatalog** full-text catalog, but it is unused because Microsoft® SQL Server[™] 2000 blocks queries on deactivated tables.

14. Add the **title** column and remove the **writer_name** column from the meta data for the full-text index of the **books** table. Execute this stored procedure once for each column: sp_fulltext_column 'books', 'writer_name', 'drop' sp_fulltext_column 'books', 'title', 'add'

For more information, see <u>sp_fulltext_column</u>.

15. Reactivate the **books** table using this stored procedure: sp_fulltext_table 'books', 'activate'

If the table is reactivated and the index is not repopulated, the old index is still available for queries against the remaining full-text enabled columns, but not for queries against any new full-text enabled columns. Before repopulation, data from deleted columns can be matched on queries that specify a search of all full-text columns by typing an asterisk (*) for the column name.

16. Start an incremental population of the **PubsCatalog** full-text catalog by executing this stored procedure: sp_fulltext_catalog 'PubsCatalog', 'start_incremental'

An incremental population refreshes the full-text catalog by indexing data in full-text enabled columns with these characteristics:

- Rows that have been updated or inserted since the last population.
- Tables that have a **timestamp** column.
- All rows that have been enabled for full-text processing since the last population, or that have a schema that has been modified in any way since the last population.
- 17. After repopulation of the **PubsCatalog** full-text catalog completes, reissue the Transact-SQL query from Step 10. This time, no error occurs.

Performing Investigation and Clean-up Tasks for Full-Text Catalogs

In this example, you perform typical investigation and clean-up tasks. Assume that you have already connected to the **pubs** database, the full-text service has been started, and that you are working with the fictitious **writers** and **books** tables. For a description of the tables, see Administering Full-Text Features Using Stored Procedures and Scalar Functions_administering_full-text_features_using_stored_procedures_and_scalar_functions

 Obtain a list of all the full-text catalogs linked to the **pubs** database by executing this stored procedure: sp_help_fulltext_catalogs

For more information, see <u>sp_help_fulltext_catalogs</u>

Because the **pubs** database is the current database, this stored procedure returns the following meta data for all the full-text catalogs linked to the **pubs** database:

- Name and integer identifier of the full-text catalog
- Full-text catalog root directory
- Full-text catalog population status
- Number of tables linked to this full-text catalog

A variation of this stored procedure in which a full-text catalog name parameter is specified returns this information for a single full-text catalog.

2. Obtain a list of all the tables in the database that have been enabled for full-text processing by executing this stored procedure:

sp_help_fulltext_tables

For more information, see <u>sp_help_fulltext_tables</u>

This stored procedure returns the following meta data for each table:

- The two-part name of the table
- The integer identifier of the column used as the table's full-text key
- The name of the index that is used to impose a unique constraint on the full-text key column
- The full-text status of the table
- The name of the full-text catalog of the table

Two other variations of this stored procedure are supported. If the **fulltext_catalog_name** parameter is specified, this information is returned for all the tables linked to that full-text catalog. If both the **catalog_name** and **table_name** parameters are specified, or if just the **table_name** parameter is specified, then this information is returned for that table.

3. Obtain a list of all the columns in the database that have been enabled for full-text processing by executing this stored procedure: sp_help_fulltext_columns

For more information, see <u>sp_help_fulltext_columns</u>

This stored procedure returns the following meta data about each column:

- The two-part name of the table in the column
- The name and integer identifier of the column

A variation of this stored procedure, in which a table name parameter is specified, returns this information for a single table. The compiled lists indicate some issues. The **mycatalog** full-text catalog is no longer used except by the **mytable** table, which no longer has any full-text columns that can be queried.

4. Unregister the **mytable** table for full-text processing by executing this stored procedure:

sp_fulltext_table 'MyTable', 'drop'

For more information, see <u>sp_fulltext_table</u>.

This drops the meta data about full-text indexing for the table. The existing full-text index remains in place until the next full population or until the full-text catalog is dropped. However, it remains unused. For more information, see <u>sp_fulltext_table</u>.

5. Drop the **mycatalog** full-text catalog from the file system and its meta data from the system tables by executing this stored procedure: sp_fulltext_catalog 'MyCatalogue', 'drop'

For more information, see <u>sp_fulltext_catalog</u>.

You must complete Step 4 before a full-text catalog can be dropped because its text-catalog meta data must be updated to remove all fulltext indexes.

There is at least one full-text catalog in the file system that no longer has corresponding SQL Server meta data. The usual cause of this is the removal of a database.

6. Remove from the file system all full-text catalogs that no longer have meta data for them in SQL Server by executing this stored procedure: sp_fulltext_service 'Clean_Up'

For more information, see <u>sp_fulltext_service</u>.

The structure of the **MixedUpCtlg** full-text catalog does not match the meta data currently recorded for it in SQL Server. This can occur when the full-text catalog is being dropped, or the database is being dropped and the Microsoft Search service is not running. The drop action changes the meta data related to the full-text catalogs, but is unable to

complete the operation because the Microsoft Search service is not running. This leads to inconsistency between the full-text meta data in SQL Server and the associated physical full-text catalog in the file system. This inconsistency can be corrected by invoking the clean-up action on **sp_fulltext_service**. (Microsoft Search service must be running.)

 Rebuild, but do not repopulate, the MixedUpCtlg full-text catalog by executing this stored procedure: sp_fulltext_catalog 'MixedUpCtlg', 'Rebuild'

For more information, see <u>sp_fulltext_catalog</u>.

The **sp_fulltext_database** stored procedure with the ENABLE option may be used to rebuild all known full-text catalogs.

 Start a full population of the MixedUpCtlg full-text catalog by executing this stored procedure: sp_fulltext_catalog 'MixedUpCtlg', 'start_full'

Note Full-text catalogs can be created, dropped, and modified as needed; however, avoid making schema changes on multiple catalogs at the same time.

Performing Infrequent Tasks

The examples in this scenario demonstrate the full-text indexing tasks typically required less frequently than other administrative tasks.

Changing the Amount of Resources Used for Full-Text Indexing and Searching

Determine if the level of resource usage has been assigned to the full-text service and how long the full-text service has to wait after a request for population.

 After starting the server, execute this statement: SELECT FulltextServiceProperty ('IsFullTextInstalled')

For more information, see <u>FULLTEXTSERVICEPROPERTY</u>.

This returns 1 if the service has been installed, and 0 if it has not.

 Determine the level of resource usage assigned to the full-text service by executing this statement: SELECT FulltextServiceProperty ('ResourceUsage')

This returns a value from 1 (background) through 5 (dedicated).

3. To determine how long the full-text service has to wait to initialize after a request to populate a full-text index, execute this statement: SELECT FulltextServiceProperty ('ConnectTimeout')

This returns the current time-out value.

 Because this installation of Microsoft® SQL Server[™] 2000 is heavily loaded, you may decide to increase the initialization time-out from the default of 20 seconds to 90 seconds by executing this stored procedure: sp_fulltext_service 'ConnectTimeout', 90

For more information, see <u>sp_fulltext_service</u>.

Inquiring About the Full-Text Key Column

Determine whether the regular SQL Server **mag_id_index** index on the **magazines** table in the **pubs** database is used to enforce the uniqueness of the full-text key column.

 Execute this statement: SELECT IndexProperty (Object_Id('magazines', 'mag_id_ind)

For more information, see **<u>INDEXPROPERTY</u>**.

This returns a value of 1 if the index is used to enforce uniqueness of the full-text key column, and 0 if it does not.

 Find the name of the full-text key column in the **books** table by executing this statement: SELECT ObjectProperty(Object_Id('books', 'TableFulltextKey

For more information, see <u>OBJECTPROPERTY</u>.

Maintaining Full-Text Indexes

There are three ways to maintain a full-text index:

• Full rebuild

Rescans all rows. Completely rebuilds the full-text index. You can perform a full rebuild immediately or on a schedule, using SQL Server Agent.

• Timestamp-based incremental rebuild

Rescans those rows that have changed since the last full or incremental rebuild. This requires a **timestamp** column on the table. Changes that do not update the timestamp, such as WRITETEXT and UPDATETEXT, are not detected. You can perform an incremental rebuild immediately or on a schedule.

• Change tracking

Maintains a list of all changes to the indexed data. Changes made with WRITETEXT and UPDATETEXT are not detected. You can update the full-text index with these changes immediately, on a schedule, or as they occur, using the background update index option.

The method you use depends on factors such as the CPU and available memory, the amount and rate of change of data, the amount of available disk space, and the importance of the full-text index being current. Use these recommendations as a guide for selecting a maintenance method.

- Use change tracking with the background update index option when CPU and memory are available, the value of an up-to-date index is high, and immediate propagation can keep up with the rate of changes.
- Use change tracking with scheduled propagation when CPU and memory can be used at scheduled times, disk space for storing changes is available, and changes between the scheduled times are not so significant that the propagation takes longer than a full rebuild.

- Use a full rebuild when a large percentage of records change or are added at once. If a large percentage of records change over an extended period of time, consider using change tracking with scheduled or background update index.
- Use an incremental rebuild when a large number, but not a large percentage, of documents change at one time. If a large number of records change over an extended period of time, consider using change tracking with scheduled or background update index.

See Also

OBJECTPROPERTY

sp_fulltext_table

Filtering Supported File Types

When a cell in an **image** column contains one of certain types of documents, full-text search uses a filter to interpret the binary data. The filter extracts the textual information from the document and submits it for indexing and subsequent querying.

Microsoft® SQL Server[™] 2000 includes filters for these file extensions: .doc, .xls, .ppt, .txt, and .htm.

Many document types can be stored in a single **image** column. For each document, SQL Server chooses the correct filter based on the file extension. Because the file extension is not visible when the file is stored in an **image** column, the file extension must be stored in a separate column on the table. This type column can be of any character-based data type and contains the document file extension, such as .doc for a Microsoft Word document. If the type column is NULL, the document is assumed to be a text file (.txt).

Note For full-text indexing, a document must be less than 16 megabytes (MB) in size and must not contain more than 256 kilobytes (KB) of filtered text.

The document-type column is created in these ways:

- In the Full-Text Indexing Wizard, select the **image** column for indexing, and then specify a **Binding column** to hold the document type.
- The **sp_fulltext_column** stored procedure accepts an argument for the column to contain the document types.

To view the document type, use the **sp_help_fulltext_columns** stored procedure to return the column name and column ID.

After the **image** column is indexed, it can be queried using the search predicates CONTAINS and FREETEXT.

Note A filter may be able to handle objects embedded in the parent object, depending on its implementation. Filters do not follow links to other objects.

You can create custom filters for full-text indexing of additional file types. For

more information about creating custom filters, search on "custom filters" in the Platform SDK section of the MSDN® Library at <u>Microsoft Web site</u>.

See Also

- sp_fulltext_column
- sp_help_fulltext_columns

Column-Level Linguistic Analysis

Linguistic analysis is performed on all full-text indexed data. Linguistic analysis involves finding word boundaries (word-breaking) and conjugating verbs (stemming). The rules for this analysis differ for different languages, and you can specify a different language for each full-text indexed column. Microsoft® SQL ServerTM 2000 includes linguistic analysis packages for these locales.

Neutral	German
Chinese_Simplified	Italian
Chinese_Traditional	Japanese
Dutch	Korean
English_UK	Spanish_Modern
English_US	Swedish_Default
French	

Use neutral when a column contains data in multiple languages or in an unsupported language.

The linguistic analysis performed on arguments to the full-text query functions, CONTAINS, FREETEXT, CONTAINSTABLE, and FREETEXTTABLE, is determined by the language of the full-text indexed column being queried. If no language is specified for a column, the default is the value of the configuration option, 'default full-text language'.

For a localized version of SQL Server, SQL Server Setup sets the **default fulltext language** option to the language of the server if an appropriate match exists. For a nonlocalized version of SQL Server, the **default full-text language** option defaults to neutral.

Note All columns listed in a single full-text query function clause must use the same language.

The **default full-text language** configuration option replaces the **language neutral full-text** option in SQL Server version 7.0. During an upgrade from SQL Server 7.0 to SQL Server 2000, the **default full-text language** value is set, based on the values of the **Unicode locale id** and **language neutral full-text**

options.

See Also

<u>sp_fulltext_column</u>