Accessing and Changing Relational Data Overview

Accessing and Changing Relational Data contains information about how you retrieve data from SQL Server tables and modify data in SQL Server tables. SQL Server applications typically work with SQL Server data in one of two ways:

- Applications use database Application Programming Interfaces (APIs) such as ADO, OLE DB, or ODBC to execute Transact-SQL statements that work with SQL Server data in the form of tabular result sets.
- Internet applications use Universal Resource Locators (URLs) or the ADO or OLE DB APIs to execute either XPath queries or Transact-SQL statements that work with SQL Server data in the form of XML documents.

Accessing and Changing Relational Data deals with fundamental aspects of building and executing Transact-SQL statements and processing results in the form of relational (or tabular) result sets. For more information about using XPath queries or Transact-SQL statements that work with XML documents, see Overview of XML and Internet Access.

Торіс	Description
Query Tools and	Describes the different classes of tools used
Programming Interfaces	to work with SQL Server. Outlines how
	Transact-SQL interfaces with utilities and
	database APIs.
Transact-SQL Syntax	Describes the primary syntax elements used
<u>Elements</u>	in Transact-SQL statements.
Accessing and Changing	Describes fundamental issues that SQL
<u>Data Fundamentals</u>	Server applications must address, but which
	are not specific to any Transact-SQL
	statement:
	Choosing a database
	Using Transact-SQL batches and scripts
	Using variables and parameters

	Controlling the flow of logic Understanding permissions
Query Fundamentals	Descirbes the fundamental processes of building Transact-SQL statements. Describes the main clauses used in the SELECT, INSERT, DELETE, and UPDATE statements:
	Select list FROM clause WHERE clause ORDER BY clause JOIN clauses
Advanced Query Concepts	Describes advanced concepts such as:
	Using aggregate functions Using GROUP BY and UNION Partitioning views Subqueries Summarizing data Error handling Transact-SQL tips
Modifying Data	Describes the methods for inserting new rows, and updating or deleting existing rows.
Transactions	Describes how several data modification statements can be grouped in a transaction.
<u>Cursors</u>	Describes how SELECT statements always return a set of rows, but applications sometimes need to go through the result set one row at a time. Cursors support processing a result set one row, or a block of rows, at a time.
Locking	Describes how SQL Server prevents multiple users from modifying the same data at the same time.
Distributed Queries	Describes how to code Transact-SQL

	statements that reference data on separate instances of SQL Server, or even in non-SQL Server OLE DB or ODBC data sources.
<u>Full-text Search</u>	Describes how to use more powerful text search functionality than is supported in SQL-92, and include files outside a database in the text searches.

Query Tools and Programming Interfaces

Users who access and change data in instances of Microsoft® SQL ServerTM require different levels of Transact-SQL knowledge, depending on the way users access the database:

- Users of graphical report generators and general business applications need little or no knowledge of Transact-SQL. The applications present either easy-to-use charts and graphs, which require little database knowledge, or dialog boxes based on the user's business functions.
- Users of general purpose SQL applications, such as SQL Query Analyzer and the **osql** utility, must understand how to use Transact-SQL from the utilities or in scripts of Transact-SQL statements.
- Application programmers must have a complete understanding of the Transact-SQL functionality in SQL applications, as well as how to use a database application programming interface (API) to provide data values for Transact-SQL statements and to retrieve data in a relational (tabular) result set.
- Internet application programmers must understand the fundamentals of how Transact-SQL statements work with relational rowsets, but they must also understand how SQL Server works with XML documents. They must understand:
 - How merged schemas present XML-based views of the data in SQL Server tables, and how to use XPath queries to retrieve that data in the form of XML documents.
 - How to use the FOR XML clause to direct SELECT statements to return results as XML documents instead of tabular result sets.

- How to add XML documents to a database and then use the OPENXML clause to present the data from the document as a relational result set.
- How to execute Transact-SQL statements, query templates, and XPath queries using Universal Resource Locators (URLs), ADO, or OLE DB.

Many users may, at different times, work in all categories of applications. Application programmers may use a SQL Server tool, such as SQL Query Analyzer, to test their Transact-SQL statements before coding them into an application. Database administrators work at all levels as they work with users and programmers to design new features and resolve database problems.

The topics in Accessing and Changing Relational Data pertain primarily to users of generic SQL tools and application programmers. Where relevant, information about additional features available to application programmers is provided. Internet programmers must know many of the concepts in Accessing and Changing Relational Data to understand the effects of the many Transact-SQL statements they execute. For information specific to working with data in the form of XML documents, see <u>XML and Internet Support Overview</u>.

Query Tools

Microsoft® offers these tools for accessing and changing data in instances of Microsoft SQL ServerTM:

- SQL Query Analyzer
- SQL Server Enterprise Manager
- osql Utility
- **bcp** Utility

The level of Transact-SQL knowledge required to use these tools varies.

SQL Server Tools

SQL Query Analyzer and the **osql** utility support using Transact-SQL interactively to access and change data, and the **bcp** utility can be used to insert large numbers of new rows into a table quickly. SQL Server Enterprise Manager is used to administer multiple instances of SQL Server from a single console.

SQL Query Analyzer and **osql** are used to:

- Execute one or more Transact-SQL statements.
- Either display the results of a query to the user, or save the results in a text file.

Using SQL Query Analyzer, you can connect simultaneously to multiple instances of SQL Server Enterprise Manager. SQL Server Enterprise Manager also supports working with multiple instances of SQL Server at the same time. The **osql** and **bcp** utilities only support working with one instance at a time.

Using SQL Query Analyzer

SQL Query Analyzer is a Microsoft® Win32® application that is an excellent tool for the ad hoc, interactive execution of Transact-SQL statements and scripts. To use SQL Query Analyzer, users must understand Transact-SQL.

In SQL Query Analyzer, users enter Transact-SQL statements in a full-text window, execute the statements, and view the results in a results window. Users also can open a text file containing Transact-SQL statements, execute the statements, and view the results in the results window.

SQL Query Analyzer also provides tools for determining how Microsoft SQL Server is interpreting and working with a Transact-SQL statement. A user can:

- Display a graphical representation of the execution plan generated for the statement.
- Start the Index Tuning Wizard to determine which indexes can be defined for the underlying tables to optimize the performance of the statement.
- Display statistics about the performance of the statement.

Using SQL Server Enterprise Manager

SQL Query Analyzer can be started from SQL Server Enterprise Manager.

In SQL Server Enterprise Manager, on the **Tools** menu, click **SQL Query Analyzer** to run SQL Query Analyzer. With SQL Query Analyzer, users can interactively design and execute queries.

SQL Server Enterprise Manager also includes the Query Designer, a graphical user interface (GUI) tool for designing queries used in specific objects:

- In the console tree, in the database you are working on, right-click **Views**, and then click **New View**. Query Designer can be used to design the SELECT statement for the view. For more information, see <u>Creating Views</u>.
- In DTS Designer, open a DTS package and add an Execute SQL task, and then click **Build Query** to access the DTS Query Designer. For more information, see <u>DTS Query Designer</u>.
- In the right pane, right-click a table, click **Open Table**, and then click **Return all rows** to see all the rows in the table. Query Designer can be used to change the query to see specific rows. For more information, see <u>Performing Basic Operations with Queries</u>.

The Query Designer window has four separate panes for specifying or displaying different items associated with a query.

Pane	Description
	Has a diagram showing the tables referenced in the query. The diagram shows all columns for each table and checks the columns used in the result set.
-	Shows the SQL statement syntax.
	Shows the details of the result set columns, such as name, data type, and size.

Results Shows the result set returned by the last query execution.

For more information about using Query Designer panes, see <u>Query and View</u> <u>Designer Layout</u>.

Using the osql Utility

The **osql** utility is a Microsoft® Win32® command prompt utility for ad hoc, interactive execution of Transact-SQL statements and scripts. To use **osql**, users must understand Transact-SQL.

The **osql** utility is typically used in these ways:

- Users interactively enter Transact-SQL statements in a manner similar to working on the command prompt. The results are displayed in the command prompt window.
- Users submit an **osql** job either specifying a single Transact-SQL statement to execute or pointing the utility to a text file that contains Transact-SQL statements to execute. The output is usually directed to a text file, but it also can be displayed in the command prompt window.

The **osql** utility uses the ODBC database application programming interface (API). It is a replacement for the **isql** command prompt utility based on the DB-Library API. Both utilities are provided with Microsoft SQL Server[™] 2000. The DB-Library API remains at a SQL Server 6.5 level; therefore, applications that depend on DB-Library, such as **isql**, do not support some SQL Server 2000 features. For example, **isql** cannot access columns defined with the **ntext** data type and truncates any **char**, **varchar**, **nchar**, or **nvarchar** columns longer than 255 bytes. It also cannot retrieve results as XML documents. Except for these limitations in **isql**, both **osql** and **isql** support the same features. For more information about features not supported by **isql**, see <u>Connecting Early Version</u> <u>Clients to SQL Server 2000</u>.

See Also

<u>isql Utility</u> osql Utility

Using the bcp Utility

The **bcp** bulk copy utility can be used to insert large numbers of new rows into Microsoft® SQL Server[™] tables. The utility requires no knowledge of Transact-SQL, but users must understand the structure of the tables into which the new rows are being copied, as well as the types of data that are valid for the rows in the table.

See Also

bcp Utility

Programming Interfaces

Microsoft® SQL Server[™] supports a number of database application programming interfaces (APIs) used to write applications that store their data in SQL Server databases. Although the users of these applications need little, if any, database knowledge, the programmers who develop the applications must know how to use the database APIs to execute the Transact-SQL statements or XPath queries that give the application access to the data stored in SQL Server databases.

Microsoft Programming Environments

Microsoft® Visual Studio® includes the major Microsoft development systems: Microsoft Visual Basic®, Microsoft Visual C++®, Microsoft Visual J++®, Microsoft Visual InterDevTM, and Microsoft Visual FoxPro®.

These systems support the development of Microsoft Windows® and Web-based applications. Microsoft SQL Server[™] supports the database application programming interfaces (APIs) used by these and other languages when building applications that store their data in a database: ADO, OLE DB, ODBC, Embedded SQL, and legacy APIs such as DB-Library. SQL Server 2000 also supports accessing instances of SQL Server through Uniform Resource Locators (URLs), and processing data in the form of XML documents.

In addition to being familiar with the use of Transact-SQL, programmers must understand how the database API interacts with Transact-SQL. They must understand how to supply data values for parameters, and how to move the data values in a result set into variables in their applications.

Internet application programmers must also understand how to work with SQL Server 2000 data as XML documents. For more information, see <u>XML and</u> <u>Internet Support Overview</u>.

Application Programming Interfaces

Microsoft® SQL Server[™] supports several database application programming interfaces (APIs): Active Data Object (ADO), OLE DB, Open Database Connectivity (ODBC), Remote Data Object (RDO), Data Access Object (DAO), the Microsoft Foundation Class (MFC) Database Classes, Embedded SQL, and DB-Library. SQL Server supports these APIs usually in the form of a dynamic-link library (DLL) called a provider or driver. The provider or driver translates the calls made by the application to the database API into commands sent to an instance of SQL Server.

To work with a database, an API:

- 1. Opens a connection to the database.
- 2. Sets options that control certain behaviors, such as whether cursors will be used, what type of cursor will be used, and whether updates are allowed.
- 3. Executes a Transact-SQL statement. Optionally, the application may also use program variables to supply parameter values for the executed statement. An application may execute only one statement at a time on each connection, or it may execute several simultaneously.
- 4. Moves the data values of return codes, output parameters, and result sets into program variables, where they can be used by the application logic. If the statement returns its result set in the form of an XML document, the application can stream that to a component that consumes XML documents.
- 5. Disconnects when finished working in the database.

The application programmer must understand both Transact-SQL and the proper use of the database API.

See Also

Application Development Architecture
Building SQL Server Applications Overview

Connecting to and Disconnecting from an Instance

An application must connect to an instance of Microsoft® SQL Server[™] before it can work with a SQL Server database. Connection occurs through a component such as shared memory or a network. An application can open multiple connections to an instance of SQL Server.

After a connection is made, the application can execute Transact-SQL statements through the connection. After an application completes all the work that must be done in an instance of SQL Server, the application disconnects. This frees all resources held by the connection in the server and terminates the network or shared-memory connection between the application and the instance.

In general, connections:

- Are associated with a database, which the application can change as needed. Object references that do not specify a database are assumed to be in the current database associated with the connection.
- Are associated with a specific login account. The login account is associated with user IDs in the SQL Server databases. A connection cannot perform an action in a database that is not permitted to the user ID associated with the connection's login account.
- Are the units of transaction control. If a connection is broken, all uncommitted modifications made by statements executed through the connection are rolled back without affecting uncommitted modifications made through other connections opened by the same application. Locks held by one connection opened by an application prevent the locked rows from being worked on by other connections opened by the same application.
- Have attributes, such as the transaction isolation level, which can be set by the application to specify connection-level behaviors.

Internet applications also connect to an instance of SQL Server, either through an ADO or OLE DB connection, or by specifying a SQL Server virtual root in a Uniform Resource Locator (URL). While the Internet application uses Transact-SQL statements or XPath queries that use XML documents instead of relational result sets, all of the characteristics listed above still apply to the connection.

Preparing and Executing Statements

With Microsoft® SQL Server[™] tools, such as SQL Query Analyzer or the **osql** utility, a user can key in and execute one or more Transact-SQL statements. Everything needed by the Transact-SQL statements must be specified in the batch of statements executed. Everything must be part of the character text of the Transact-SQL statements.

The database APIs support many options for executing a Transact-SQL statement. The most important options are:

• Using precompiled execution plans of frequently used statements. If a Transact-SQL statement is executed several times, the application can prepare the statement once, and then execute it as many times as needed. Preparing a statement directs the server to compile the statement into an execution plan. Subsequent executions are faster because they use the precompiled plan so the statement has to be compiled only once.

For better performance, the statement preparation may be deferred until the statement is executed or a metaproperty operation (such as **SQLDescribeCol** or **SQLDescribeParam** in ODBC) is performed. This is the default behavior. Any errors in the statement being prepared are not known until the statement is executed or a metaproperty operation is performed. Setting appropriate statement options can turn off this default behavior.

• Binding program variables with the parameters. Instead of having to include the actual data values for input parameters as part of the Transact-SQL statement, an application can associate, or bind, program variables with the parameters. This means that the parameter values do not have to be converted to character strings to be included in the text of the Transact-SQL statement, but instead can be used in their native format.

Prepared statements cannot be used to create temporary objects in SQL Server 2000 or SQL Server 7.0. Prepared statements cannot reference system stored

procedures that create temporary objects, such as temporary tables. These procedures must be executed directly.

Processing Results

With Microsoft® SQL Server[™] tools, such as SQL Query Analyzer or the **osql** utility, the results of a Transact-SQL statement are either displayed as character text or saved in a text file. SQL Server displays the entire result set at once, rather than fetching the rows one at a time.

When an application executes a Transact-SQL statement that returns a relational result set, the database APIs enable an application to associate, or bind, the columns of a result set with variables in the application. When a result set row is retrieved, the data in the row columns is moved into the bound variables where it can then be used by the application. Once again, the data can be retrieved in its native format without being converted to character text.

The database APIs also support cursor processing of relational result sets. This allows the application to retrieve the rows in the result set one at a time, or one block of rows at a time. The application is not forced to retrieve and store the entire result set before processing it.

When an application executes an XPath query or Transact-SQL statement that returns an XML document, the document is returned as a stream object. For more information, see <u>XML and Internet Support Overview</u>.

Data Type Conversions

Programmers building database applications that use relational result sets must handle two levels of data conversion:

- All of the Transact-SQL statements coded in the application must comply with the Transact-SQL data conversion rules when combining objects with operators and functions. The Transact-SQL data conversion and precedence rules also determine the final output data type of these operations.
- The program must comply with the database API data conversion rules when moving data between program variables and database objects

such as result set columns, parameters, and return codes.

Transact-SQL supports conversion of data values from one data type to another. For example, this statement converts an integer value into a character string:

CAST (123 AS VARCHAR(5))

The conversions can be explicit, using the CAST function, or they can be implicit. For example, if an **int** column is compared to a **char** column, the **char** value is implicitly converted to an **int** before the comparison is made. The Transact-SQL Reference defines the implicit and explicit conversions allowed by SQL Server. These rules apply only to conversions between Transact-SQL objects.

Another set of rules applies when converting between Transact-SQL objects such as parameters, return codes, and result set columns and their bound program variables. These rules are defined in the documentation for the provider or driver supporting the API. The rules can vary among the APIs. For example, the SQL Server ODBC driver supports converting the data from a **datetime** result set column into an ODBC **timestamp** data structure, but the DB-Library interface does not allow this conversion because it does not support ODBC **timestamp** data structures.

See Also

Data Type Conversion CAST and CONVERT Data Types Mapping Data Types

Transact-SQL Syntax Elements

Transact-SQL has several syntax elements that are used by, or influence, most statements:

Identifiers

Are the names of objects such as tables, views, columns, databases, and servers.

Data types

Define the types of data contained by data objects, such as columns, variables, and parameters. Most Transact-SQL statements do not reference data types explicitly, but the results of most statements are influenced by the interactions between the data types of the objects referenced in the statement.

Functions

Are syntax elements that take zero, one, or more input values and return a scalar value or a tabular set of values. Examples include the SUM function for adding several values, the DATEDIFF function for determining how many units of time separate two dates, the @@SERVERNAME function for getting the name of an instance of Microsoft® SQL ServerTM, or the OPENQUERY function for executing a Transact-SQL statement against a remote server and retrieving the result set.

Expressions

Are units of syntax that Microsoft SQL Server can resolve to single values. Examples of expressions include constants, functions that return a single value, a reference to a column, or a variable.

Operators

Work with one or more simple expressions to form a more complex expression. For example, combining the minus sign (-) with the constant 12 results in the constant -12. The multiplication sign (*) in the expression PriceColumn *1.1 increases the price by 10 percent.

Comments

Are pieces of text inserted into Transact-SQL statements or scripts to explain the purpose of the statement. The comments are not executed by SQL Server.

Reserved Keywords

Are words reserved for the use of SQL Server and should not be used for the names of objects in a database.

Using Identifiers

The database object name is known as its identifier. Everything in Microsoft® SQL Server[™] can have an identifier. Servers, databases, and database objects such as tables, views, columns, indexes, triggers, procedures, constraints, rules, and so on can have identifiers. Identifiers are required for most objects, but are optional for some objects, such as constraints.

An object identifier is created when the object is defined. The identifier is then used to reference the object. For example, this statement creates a table with the identifier **TableX**, and two columns with the identifiers **KeyCol** and **Description**:

CREATE TABLE TableX (KeyCol INT PRIMARY KEY, Description NVARCHAR(80))

This table also has an unnamed constraint. The PRIMARY KEY constraint has no identifier.

Classes of Identifiers

There are two classes of identifiers:

Regular identifiers

Conform to the rules for the format of identifiers. Regular identifiers are not delimited when used in Transact-SQL statements.

SELECT * FROM TableX WHERE KeyCol = 124

Delimited identifiers

Are enclosed in double quotation marks (") or brackets ([]). Identifiers that comply with the rules for the format of identifiers may or may not be delimited.

SELECT *

FROM [TableX] --Delimiter is optional. WHERE [KeyCol] = 124 --Delimiter is optional.

Identifiers that do not comply with all of the rules for identifiers must be delimited in a Transact-SQL statement.

```
SELECT *
FROM [My Table] --Identifier contains a space and uses a reserved
WHERE [order] = 10 --Identifier is a reserved keyword.
```

Both regular and delimited identifiers must contain from 1 through 128 characters. For local temporary tables, the identifier can have a maximum of 116 characters.

Rules for Regular Identifiers

The rules for the format of regular identifiers are dependent on the database compatibility level, which can be set with **sp_dbcmptlevel**. For more information, see <u>sp_dbcmptlevel</u>. When the compatibility level is **80**, the rules are:

- 1. The first character must be one of the following:
 - A letter as defined by the Unicode Standard 2.0. The Unicode definition of letters includes Latin characters from a through z and from A through Z, in addition to letter characters from other languages.
 - The underscore (_), "at" sign (@), or number sign (#).

Certain symbols at the beginning of an identifier have special meaning in SQL Server. An identifier beginning with the "at" sign denotes a local variable or parameter. An identifier beginning with a number sign denotes a temporary table or procedure. An identifier beginning with double number signs (##) denotes a global temporary object.

Some Transact-SQL functions have names that start with double at signs (@@). To avoid confusion with these

functions, it is recommended that you do not use names that start with @@.

- 2. Subsequent characters can be:
 - Letters as defined in the Unicode Standard 2.0.
 - Decimal numbers from either Basic Latin or other national scripts.
 - The "at" sign, dollar sign (\$), number sign, or underscore.
- 3. The identifier must not be a Transact-SQL reserved word. SQL Server reserves both the uppercase and lowercase versions of reserved words.
- 4. Embedded spaces or special characters are not allowed.

When used in Transact-SQL statements, identifiers that fail to comply with these rules must be delimited by double quotation marks or brackets.

See Also

ALTER TABLE

CREATE DATABASE

CREATE DEFAULT

CREATE PROCEDURE

CREATE RULE

CREATE TABLE

CREATE TRIGGER

CREATE VIEW

DECLARE @local_variable

DELETE

Delimited Identifiers

INSERT

Reserved Keywords

<u>SELECT</u>

<u>UPDATE</u>

Using Identifiers as Object Names

The complete name of an object consists of four identifiers: the server name, database name, owner name, and object name. They appear in the following format:

[[[server.][database].][owner_name].]object_name

The server, database, and owner names are known as the qualifiers of the object name. When referring to an object, it is not necessary to specify the server, database, and owner. The qualifiers can be omitted by marking their positions with a period. The valid forms of object names are:

```
server.database.owner_name.object_name server.database..object_name
server..owner_name.object_name
server...object_name
```

```
database.owner_name.object_name
database.object_name
```

owner_name.object_name

object_name

An object name that specifies all four parts is known as a fully qualified name. Each object created in Microsoft® SQL ServerTM must have a unique, fully qualified name. For example, there can be two tables named **xyz** in the same database if they have different owners.

Column names must be unique within a table or view. Assume that both a table and a view in the **customer** database have the same column named **telephone**. To refer to the **telephone** column in the **employees** table, specify **customer..employees.telephone**. To refer to the **telephone** column in the **mktg_view** view (marketing department view), specify **customer..mktg_view.telephone**.

Most object references use three-part names and default to the local server. Fourpart names are generally used for distributed queries or remote stored procedure calls and use this format.

linkedserver.catalog.schema.object_name

Part name	Description
linkedserver	Name of the linked server that contains the object referenced by the distributed query.
Catalog	Name of the catalog that contains the object referenced by the distributed query.
Schema	Name of the schema that contains the object referenced by the distributed query.
object_name	Object name or table name.

The table shows the part names and their descriptions.

For distributed queries, the server part of a four-part name refers to a linked server. A linked server is a server name defined with **sp_addlinkedserver**. The linked server identifies an OLE DB provider and an OLE DB data source that can return a record set that SQL Server can use as part of a Transact-SQL statement.

See the documentation for the OLE DB provider specified for the linked server to determine what components in the OLE DB data source are used for the catalog and schema parts of the name. If the linked server is running an instance of SQL Server, the catalog name is the database containing the object, and the schema is the owner of the object. For more information about four-part names and distributed queries, see <u>Distributed Queries</u>.

For remote procedure calls, the server part of a four-part name refers to a remote server. A remote server, which is specified with **sp_addserver**, is an instance of SQL Server accessed through the local server. Execute stored procedures on the remote server using this format for the procedure name:

server.database.owner_name.procedure

All four parts of the name are required when using a remote stored procedure. For more information about remote servers, see <u>Configuring Remote Servers</u>.

See Also

FROM

Transact-SQL Syntax Conventions

Object Visibility and Qualification Rules

When you create an object, Microsoft® SQL Server[™] 2000 uses the following defaults for the parts of the name not specified:

- *Server* defaults to the local server.
- *Database* defaults to the current database.
- *Owner_name* defaults to the username in the specified database associated with the login ID of the current connection.

For example, if a user is logged on to **Northwind** as the database owner (**dbo**) user, either of the following two statements creates a table named **Northwind.dbo.TableX**:

CREATE TABLE TableX (cola INT PRIMARY KEY, colb NCHAR(3)

-Or-

CREATE TABLE Northwind.dbo.TableX (cola INT PRIMARY KEY, colb NCHAR(3))

Note It is recommended that the full table or view name be specified to eliminate possible confusion relating to the object in question.

Similarly, when you refer to an object, Microsoft® SQL Server[™] uses the following defaults for the parts of the name not specified:

- Server defaults to the local server.
- *Database* defaults to the current database.
- *owner_name* defaults to the username in the specified database associated with the login ID of the current connection. If that user owns no object with the specified name, SQL Server looks for an object with

the specified name owned by the database owner (**dbo**) user.

For example, assume **LoginX** connects to a server that has two databases: **DBY** and **DBZ**. **LoginX** is associated with **UserA** in database **DBY** and with **UserB** in database **DBZ**.

LoginX executes a SELECT statement in the current database:

USE DBY SELECT * FROM DBY..TableX

Because **LoginX** is associated with **UserA** in **DBY**, SQL Server first looks for **DBY.UserA.TableX**. If there is no table with this name, SQL Server looks for a table **DBY.dbo.TableX**.

In the next example, **LoginX** executes a SELECT statement on a table not in the current database:

USE DBY SELECT * FROM DBZ..TableY

Because **LoginX** is associated with **UserB** in database **DBZ**, SQL Server first looks for **DBZ.UserB.TableY**. If there is no table with this name, SQL Server then looks for a table **DBZ.dbo.TableY**.

Note SQL Server does not try to deduce the owner of remote tables based on the current login. To ensure that distributed queries execute properly, use fully qualified names.

The visibility for stored procedures that begin with **sp_** differs from the visibility for regular stored procedures. For more information, see <u>CREATE</u> <u>PROCEDURE</u>.

Delimited Identifiers

An identifier that complies with all the rules for the format of identifiers can be used with or without delimiters. An identifier that does not comply with the rules for the format of regular identifiers must always be delimited.

Delimited identifiers are used in these situations:

• When reserved words are used for object names or portions of object names.

It is recommended that reserved keywords not be used as object names. Databases upgraded from earlier versions of Microsoft® SQL ServerTM may contain identifiers that include words not reserved in the earlier version, but are reserved words for SQL Server 2000. You can refer to the object using delimited identifiers until the name can be changed.

• When using characters not listed as qualified identifiers.

SQL Server allows any character in the current code page to be used in a delimited identifier; however, indiscriminate use of special characters in an object name may make SQL statements and scripts difficult to read and maintain.

Types of delimiters used in Transact-SQL:

Note Delimiters are for identifiers only. Delimiters cannot be used for keywords, whether or not they are marked as reserved in SQL Server.

- Quoted identifiers are delimited by double quotation marks ("): SELECT * FROM "Blanks in Table Name"
- Bracketed identifiers are delimited by brackets ([]): SELECT * FROM [Blanks In Table Name]

Quoted identifiers are valid only when the QUOTED_IDENTIFIER option is set to ON. By default, the Microsoft OLE DB Provider for SQL Server and SQL Server ODBC driver set QUOTED_IDENTIFIER ON when they connect. DB-Library does not set QUOTED_IDENTIFIER ON by default. Regardless of the interface used, individual applications or users may change the setting at any time. SQL Server provides a number of ways to specify this option. For example, in SQL Server Enterprise Manager and SQL Query Analyzer, the option can be set in a dialog box. In Transact-SQL, the option can be set at various levels using SET QUOTED_IDENTIFIER, the **quoted identifier** option of **sp_dboption**, or the **user options** option of **sp_configure**.

When QUOTED_IDENTIFIER is ON, SQL Server follows the SQL-92 rules for the use of double quotation marks and the single quotation mark (') in SQL statements:

• Double quotation marks can be used only to delimit identifiers. They cannot be used to delimit character strings.

To maintain compatibility with existing applications, SQL Server does not fully enforce this rule. Character strings can be enclosed in double quotation marks if the string does not exceed the length of an identifier; this practice is not recommended.

• Single quotation marks must be used to enclose character strings. They cannot be used to delimit identifiers.

If the character string contains an embedded single quotation mark, insert an additional single quotation mark in front of the embedded mark:

```
SELECT * FROM "My Table"
WHERE "Last Name" = 'O"Brien'
```

When QUOTED_IDENTIFIER is OFF, SQL Server follows these rules for the use of single and double quotation marks:

- Quotation marks cannot be used to delimit identifiers. Instead, use brackets as delimiters.
- Single or double quotation marks can be used to enclose character strings.

If double quotation marks are used, embedded single quotation marks do not have to be denoted by two single quotation marks: SELECT * FROM [My Table] WHERE [Last Name] = "O'Brien"

Delimiters in brackets can always be used, regardless of the setting of QUOTED_IDENTIFIER.

Rules for Delimited Identifiers

The rules for the format of delimited identifiers are:

- 1. Delimited identifiers can contain the same number of characters as regular identifiers (from 1 through 128 characters, not including the delimiter characters). Local temporary table identifiers can be a maximum of 116 characters.
- 2. The body of the identifier can contain any combination of characters in the current code page except the delimiting characters themselves. For example, delimited identifiers can contain spaces, any characters valid for regular identifiers, and any of the following characters:

tilde (~)	hyphen (-)
exclamation point (!)	left brace ({)
percent (%)	right brace (})
caret (^)	apostrophe (')
ampersand (&)	period (.)
left parenthesis (()	backslash (\)
right parenthesis ())	accent grave (`)

These examples use quoted identifiers for table names and column names. Both methods for specifying delimited identifiers are shown:

```
SET QUOTED_IDENTIFIER ON
GO
CREATE TABLE "$Employee Data"
(
```

```
"^First Name" varchar(25) NOT NULL,
"^Last Name"
               varchar(25) NOT NULL,
"^Dept ID"
             int
-- INSERT statements go here.
SET QUOTED_IDENTIFIER OFF
GO
CREATE TABLE [^$Employee Data]
(
[^First Name]
               varchar(25) NOT NULL,
[^Last Name]
               varchar(25) NOT NULL,
[^Dept ID]
            int
)
-- INSERT statements go here.
```

After the **\$Employee Data** and **^\$Employee Data** tables are created and data is entered, rows can be retrieved:

```
SET QUOTED_IDENTIFIER ON
GO
SELECT *
FROM "$Employee Data"
SET QUOTED_IDENTIFIER OFF
GO
-- Or
SELECT *
FROM [^$Employee Data]
```

In this example, a table named **table** contains columns **tablename**, **user**, **select**, **insert**, and so on. Because TABLE, SELECT, INSERT, UPDATE, and DELETE are reserved keywords, the identifiers must be delimited each time the objects are accessed.

```
SET QUOTED_IDENTIFIER ON GO
```

```
CREATE TABLE "table"
(
tablename char(128) NOT NULL,
"USER" char(128) NOT NULL,
"SELECT" char(128) NOT NULL,
"INSERT" char(128) NOT NULL,
"UPDATE" char(128) NOT NULL,
"DELETE" char(128) NOT NULL)
```

If the SET QUOTED_IDENTIFIER option is not ON, the table and columns cannot be accessed unless bracket delimiters are used.

```
SET QUOTED_IDENTIFIER OFF
GO
SELECT *
FROM "table"
```

Here is the result set:

Msg 170, Level 15, State 1 Line 1: Incorrect syntax near 'table'.

Here is the result set (using bracket delimiters):

SET QUOTED_IDENTIFIER OFF GO SELECT * FROM [table]

Delimiting Identifiers with Multiple Parts

When using qualified object names you may have to delimit more than one of the identifiers that make up the object name. Each identifier must be delimited individually, for example:

/* SQL-92 quoted identifier syntax */

```
SELECT *
FROM "My DB"."My#UserID"."My.Table"
```

-Or-

```
/* Transact-SQL bracketed identifier syntax */
/* Not available in SQL Server 6.5 or earlier */
SELECT *
FROM [My DB].[My#UserID].[My.Table]
```

There are some special rules regarding how you delimit multi-part stored procedure names in the ODBC CALL statement. For more information, see <u>Calling a Stored Procedure</u>.

Using Identifiers as Parameters in SQL Server

Many system stored procedures, functions, and DBCC statements take object names as parameters. Some of these parameters accept multipart object names, while others accept only single-part names. Whether a single-part or multipart name is expected determines how a parameter is parsed and used internally by SQL Server.

Single-part Parameter Names

If the parameter is a single-part identifier, the name can be specified:

- Without quotation marks or delimiters.
- Enclosed in single quotation marks.
- Enclosed in double quotation marks.
- Enclosed in brackets.

For single-part names, the string inside the single quotation marks represents the object name. If delimiters are used inside single quotation marks, the delimiter

characters are treated as part of the name.

If the name contains a period or another character that is not part of the character set defined for regular identifiers, you must enclose the object name in single quotation marks, double quotation marks, or brackets.

Multipart Parameter Names

Multipart names are qualified names that include the database or owner name in addition to the object name. SQL Server requires that when a multipart name is used as a parameter, the entire string that constitutes the multipart name must be enclosed in a set of single quotation marks.

EXEC MyProcedure @name = 'dbo.Employees'

If individual name parts require delimiters, each part of the name should be delimited separately as required. For example, if a name part contains a period, double quotation mark, or left or right bracket, use brackets or double quotation marks to delimit the part. Enclose the complete name in single quotation marks.

For example, the table name, **tab.one**, contains a period. To prevent the name from being interpreted as a three-part name, **dbo.tab.one**, delimit the table name part.

EXEC sp_help 'dbo.[tab.one]'

This example shows the same table name delimited with double quotation marks.

```
SET QUOTED_IDENTIFIER ON
GO
EXEC sp_help 'dbo."tab.one"'
GO
```

This table lists some of the Transact-SQL functions, DBCC statements, and system stored procedures that use multipart names.

Function or stored procedure name	Parameter name
COL_LENGTH	table

DBCC CHECKIDENT	table_name		
DBCC CHECKTABLE	table_name		
DBCC DBREINDEX	database.owner.table_name		
DBCC SHOW_STATISTICS	table		
DBCC TEXTALLOC	table_name		
DBCC UPDATEUSAGE	table_name		
IDENT_INCR	table_or_view		
IDENT_SEED	table_or_view		
INDEX_COL	table		
OBJECT_ID	object		
sp_addextendedproc	procedure		
sp_autostats	table_name		
sp_bindefault	default		
	object_name		
sp_bindrule	rule		
	object_name		
sp_changeobjectowner	object		
sp_depends	object		
sp_dropextendedproc	procedure		
sp_fulltext_column	qualified_table_name		
sp_fulltext_table	qualified_table_name		
sp_help	name		
sp_helpconstraint	table		
sp_help_fulltext_columns	table_name		
sp_help_fulltext_columns_cursor	table_name		
sp_help_fulltext_tables	table_name		
sp_help_fulltext_tables_cursor	table_name		
sp_helpindex	name		
sp_helprotect	object_statement		
sp_helptext	name		
sp_helptrigger	table		
sp_procoption	procedure		
sp_recompile	table		

sp_rename	object_name
sp_spaceused	objname
sp_tableoption	table
sp_unbindefault	object_name
sp_unbindrule	object_name

See Also

ALTER DATABASE

ALTER PROCEDURE

ALTER TABLE

ALTER TRIGGER

ALTER VIEW

CREATE DATABASE

CREATE DEFAULT

CREATE PROCEDURE

CREATE RULE

CREATE TABLE

CREATE TRIGGER

Reserved Keywords

SET QUOTED_IDENTIFIER

Using Data Types

Objects that contain data have an associated data type that defines the kind of data (character, integer, binary, and so on) the object can contain. The following objects have data types:

- Columns in tables and views.
- Parameters in stored procedures.
- Variables.
- Transact-SQL functions that return one or more data values of a specific data type.
- Stored procedures that have a return code, which always has an **integer** data type.

Assigning a data type to an object defines four attributes of the object:

- The kind of data contained by the object. For example, character, integer or binary.
- The length of the stored value, or its size.

The length of an **image**, **binary**, and **varbinary** data type is defined in bytes. The length of any of the numeric data types is the number of bytes required to hold the number of digits allowed for that data type. The length of the character string and Unicode data types is defined in characters.

• The precision of the number (numeric data types only).

The precision is the number of digits the number can contain. For example, a **smallint** object can hold a maximum of 5 digits; it has a

precision of 5.

• The scale of the number (numeric data types only).

The scale is the number of digits that can be stored to the right of the decimal point. For example, an **int** object cannot accept a decimal point and has a scale of 0. A **money** object can have a maximum of 4 digits to the right of the decimal point and has a scale of 4.

If an object is defined as **money**, it can contain a maximum of 19 digits, 4 of which can be to the right of the decimal. The object uses 8 bytes to store the data. The **money** data type therefore has a precision of 19, a scale of 4, and a length of 8.

bigint	Binary	bit	char	cursor
datetime	Decimal	float	image	int
money	Nchar	ntext	nvarchar	real
smalldatetime	Smallint	smallmoney	text	timestamp
tinyint	Varbinary	Varchar	uniqueidentifier	

Transact-SQL has these base data types.

All data stored in Microsoft® SQL ServerTM must be compatible with one of these base data types. The **cursor** data type is the only base data type that cannot be assigned to a table column. It can be used only for variables and stored procedure parameters.

Several base data types have synonyms (for example, **rowversion** is a synonym for **timestamp**, and **national character varying** is a synonym for **nvarchar**). For more information about the behavior of synonyms, see <u>Data Type</u> <u>Synonyms</u>.

User-defined data types can also be created, for example:

-- Create a birthday datetype that allows nulls. EXEC sp_addtype birthday, datetime, 'NULL' GO

-- Create a table using the new data type.

CREATE TABLE employee

emp_id	char(5)
emp_first_name	char(30)
emp_last_name	char(40)
emp_birthday	birthday

User-defined data types are always defined in terms of a base data type. They provide a mechanism for applying a name to a data type that is more descriptive of the types of values to be held in the object. This can make it easier for a programmer or database administrator to understand the intended use of any object defined with the data type.

Instances of SQL Server include a user-defined data type named **sysname**. **sysname** is used for table columns, variables, and stored procedure parameters that store object names. The exact definition of **sysname** is related to the rules for identifiers; therefore, it can vary between instances of SQL Server. **sysname** is functionally equivalent to **nvarchar(128)**. SQL Server version 6.5 or earlier only supports only smaller identifiers; thus, in earlier versions, **sysname** is defined as **varchar(30)**.

See Also

CREATE TABLE

<u>Data Types</u>

Designing Tables

sp_addtype

Using Binary Data

The **binary** and **varbinary** data types store strings of bits. Although character data is interpreted based on the Microsoft® SQL ServerTM 2000 code page, **binary** and **varbinary** data is simply a stream of bits. **binary** and **varbinary** data can be a maximum of 8,000 bytes.

Binary constants have a leading 0x (a zero and the lowercase letter x) followed by the hexadecimal representation of the bit pattern. For example, 0x2A specifies the hexadecimal value of 2A, which is equivalent to a decimal value of 42 or a one-byte bit pattern of 00101010.

Use binary data when storing hexadecimal values such as a security identification number (SID), a GUID (using the **uniqueidentifier** data type), or a complex number that can be stored using hexadecimal shorthand.

This Transact-SQL example stores a SID and hexadecimal literal:

```
USE pubs
CREATE TABLE mycustomertable
(
user_login DEFAULT SUSER_SID,
data_value
```

```
INSERT mycustomertable (data_value)
VALUES (0x4F)
```

See Also

Binary Data

Data Types

Using char and varchar Data

The **char** and **varchar** data types store data composed of:

- Uppercase or lowercase characters such as a, b, and C.
- Numerals such as 1, 2, or 3.
- Special characters such as the "at" sign (@), ampersand (&), and exclamation point (!).

char or **varchar** data can be a single character, or a string with a maximum of 8,000 characters.

Each **char** and **varchar** data value has a collation. Collations define attributes such as the bit patterns used to represent each character, comparison rules, and sensitivity to case or accenting. Each database has a default collation. When a column is defined, or a constant specified, they are assigned the default collation of the database unless you assign a specific collation using the COLLATE clause. When two **char** or **varchar** values having different collations are combined or compared, collation precedence rules determine which collation is used for the operation.

Character constants must be enclosed in single quotation marks (') or double quotation marks (''). Enclosing a character constant in single quotation marks is recommended. Enclosing a character constant in double quotation marks is sometimes not allowed when the QUOTED IDENTIFIER option is set to ON.

This Transact-SQL example sets a character variable to a value:

```
DECLARE @MyCharVar CHAR(25)
SET @MyCharVar = 'Ricardo Adocicados'
```

When using single quotation marks to delimit a character constant that contains an embedded single quotation mark, use two single quotation marks to represent the embedded single quotation mark, for example:

SET @MyCharVar = 'O"Leary'

If the data to be stored is longer than the number of characters allowed, the data is truncated. For example, if a column is defined as **char(10)** and the value "This is a really long character string" is stored into the column, Microsoft® SQL ServerTM truncates the character string to "This is a".

The **char** data type is a fixed-length data type when the NOT NULL clause is specified. If a value shorter than the length of the column is inserted into a **char** NOT NULL column, the value is right-padded with blanks to the size of the column. For example, if a column is defined as **char(10)** and the data to be stored is "music", SQL Server stores this data as "music_____", where "__" indicates a blank.

If ANSI_PADDING is ON when a **char** NULL column is created, it behaves the same as a **char** NOT NULL column: values are right-padded to the size of the column. If ANSI_PADDING is OFF when a **char** NULL column is created, it behaves like a **varchar** column with ANSI_PADDING set OFF: trailing blanks are truncated.

The **varchar** data type is a variable-length data type. Values shorter than the size of the column are not right-padded to the size of the column. If the ANSI_PADDING option was set to OFF when the column was created, any trailing blanks are truncated from character values stored in the column. If ANSI_PADDING was set ON when the column was created, trailing blanks are not truncated.

The CHAR function can be used to convert an integer code to an ASCII character. This is useful when trying to specify control characters, such as a carriage return or line feed. Use CHAR(13) and CHAR(10) to put a carriage return and new line in a character string:

PRINT 'First line.' + CHAR(13) + CHAR(10) + 'Second line.'

The way the bit patterns stored in the bytes of a character string are interpreted is based on the Microsoft SQL Server code page specified during Setup. A **char** or **varchar** object can contain any character in the SQL Server code page. For more information, see <u>Collations</u>.

DB-Library applications and applications using the SQL Server ODBC drivers

from SQL Server version 6.5 or earlier support only a maximum of 255 bytes of character data. If these applications attempt to retrieve character parameters of SQL Server version 7.0 or later, or result set columns containing more than 255 bytes of data, the character data is truncated at 255 bytes.

See Also

Character Data Collations Collation Precedence Data Types SET ANSI_PADDING sp_dbcmptlevel

Using Date and Time Data

Microsoft® SQL Server[™] 2000 has the **datetime** and **smalldatetime** data types to store date and time data.

There are no separate time and date data types for storing only times or only dates. If only a time is specified when setting a **datetime** or **smalldatetime** value, the date defaults to January 1, 1900. If only a date is specified, the time defaults to 12:00 A.M. (Midnight).

On **datetime** data you can perform operations such as:

- Entering new or changing existing dates.
- Performing date and time calculations, such as adding or subtracting dates. For more information about date arithmetic, see <u>+ (Add)</u> and <u>- (Subtract)</u>.
- Searching for a particular date and/or time.

You can perform some arithmetic calculations on **datetime** data with the system date functions. For more information, see <u>Functions</u>.

Here are some guidelines when using date and time data:

- To search for an exact match on both date and time, use an equal sign (=). Microsoft SQL Server returns date and time values exactly matching the month, day, and year, and at the precise time of 12:00:000 A.M. (default).
- To search for a partial date or time value, use the LIKE operator. SQL Server first converts the dates to **datetime** format and then to **varchar**. Because the standard display formats do not include seconds or milliseconds, you cannot search for them with LIKE and a matching pattern, unless you use the CONVERT function with the *style* parameter set to 9 or 109. For more information about searching for partial dates or times, see LIKE.

SQL Server evaluates datetime constants at run time. A date string that works for the date formats expected by one language may be unrecognizable if the query is executed by a connection using a different language and date format setting. For example, this view works correctly for connections made with the language set to U.S. English, but not for connections made using other languages: CREATE VIEW USA_Dates AS SELECT *
FROM Northwind.dbo.Orders
WHERE OrderDate < 'May 1, 1997'

When you use **datetime** constants in queries executed by connections using different language settings, ensure that the dates are acceptable for all the language settings. The same care must be taken with **datetime** constants in permanent objects in international databases, such as table constraints and view WHERE clauses. For more information about date formats interpreted the same by all language settings, see <u>Writing</u> <u>International Transact-SQL Statements</u>.

SQL Server recognizes date and time data enclosed in single quotation marks (') in these formats:

- Alphabetic date formats (for example, 'April 15, 1998')
- Numeric date formats (for example, '4/15/1998', 'April 15, 1998')
- Unseparated string formats (for example, '19981207', 'December 12, 1998')

Alphabetic Date Format

Microsoft® SQL Server[™] 2000 allows you to specify date data with a month specified as the full month name (for example, April) or the month abbreviation (for example, Apr) given in the current language; commas are optional and capitalization (case) is ignored.

Here are some guidelines for the use of alphabetic date formats:

- Enclose the date and time data in single quotation marks (').
- These are the valid alphabetic formats for SQL Server date data (characters enclosed in brackets are optional): Apr[il] [15][,] 1996 Apr[il] 15[,] [19]96 Apr[il] 1996 [15]

[15] Apr[il][,] 1996
15 Apr[il][,][19]96
15 [19]96 apr[il]
[15] 1996 apr[il]

1996 APR[IL] [15] 1996 [15] APR[IL]

• If you specify only the last two digits of the year, values less than the last two digits of the value of the **two digit year cutoff** configuration option are in the same century as the cutoff year. Values greater than or equal to the value of this option are in the century that precedes the cutoff year. For example, if **two digit year cutoff** is 2050 (default), 25 is interpreted as 2025 and 50 is interpreted as 1950. To avoid ambiguity, use four-digit years.

- If the day is missing, the first day of the month is supplied.
- The SET DATEFORMAT session setting is not applied when you specify the month in alphabetic form.

See Also

Date and Time Data datetime and smalldatetime

<u>LIKE</u>

SET DATEFORMAT

Numeric Date Format

Microsoft® SQL Server[™] 2000 allows you to specify date data with a numeric month specified. For example, 5/20/97 represents the twentieth day of May, 1997. When using numeric date format, specify the month, day, and year in a string with slash marks (/), hyphens (-), or periods (.) as separators. This string must appear in the following form:

number separator number separator number [time] [time]

These numeric formats are valid:

[0]4/15/[19]96 -- (mdy) [0]4-15-[19]96 -- (mdy) [0]4.15.[19]96 -- (mdy) [04]/[19]96/15 -- (myd)

15/[0]4/[19]96 -- (dmy) 15/[19]96/[0]4 -- (dym) [19]96/15/[0]4 -- (ydm) [19]96/[04]/15 -- (ymd)

When the language is set to **us_english**, the default order for the date is mdy. You can change the date order with the SET DATEFORMAT statement, which can also affect the date order, depending on the language.

The setting for SET DATEFORMAT determines how date values are interpreted. If the order does not match the setting, the values are not interpreted as dates (because they are out of range), or the values are misinterpreted. For example, 12/10/08 can be interpreted as one of six dates, depending on the DATEFORMAT setting.

See Also

Date and Time Data

datetime and smalldatetime
<u>SET DATEFORMAT</u>

Unseparated String Format

Microsoft® SQL Server[™] 2000 allows you to specify date data as an unseparated string. The date data can be specified with four, six, or eight digits, an empty string, or a time value without a date value.

The SET DATEFORMAT session setting does not apply to all-numeric date entries (numeric entries without separators). Six- or eight-digit strings are always interpreted as ymd. The month and day must always be two digits.

This is the valid unseparated string format:

[19]960415

A string of only four digits is interpreted as the year. The month and date are set to January 1. When specifying only four digits, you must include the century.

See Also

Date and Time Data datetime and smalldatetime <u>SET DATEFORMAT</u>

Time Formats

Microsoft[®] SQL Server[™] 2000 recognizes the following formats for time data. Enclose each format with single quotation marks (').

```
14:30
14:30[:20:999]
14:30[:20.9]
4am
4 PM
[0]4[:30:20:500]AM
```

You can specify a suffix of AM or PM to indicate if the time value is before or after 12 noon. The case of AM or PM is ignored.

Hours can be specified using either a 12-hour or 24-hour clock. This is how the hour values are interpreted:

- The hour value of 0 represents the hour after midnight (AM), regardless of whether or not you specify AM. You cannot specify PM when the hour equals 0.
- Hour values from 1 through 11 represent the hours before noon if neither AM nor PM is specified. They also represent the hours before noon when AM is specified. They represent hours after noon if PM is specified.
- The hour value 12 represents the hour that starts at noon if neither AM nor PM is specified. If AM is specified, it represents the hour that starts at midnight. If PM is specified, it represents the hour that starts at noon. For example: 12:01 is 1 minute after noon, as is 12:01 PM, while 12:01 AM is 1 minute after midnight. Specifying 12:01 AM is the same as specifying 00:01 or 00:01 AM.

- Hour values from 13 through 23 represents hours after noon if AM or PM is specified. They also represent the hours after noon when PM is specified. You cannot specify AM when the hour value is from 13 through 23.
- An hour value of 24 is not valid, use 12:00 AM or 00:00 to represent midnight.

Milliseconds can be preceded by either a colon (:) or a period (.). If preceded by a colon, the number means thousandths-of-a-second. If preceded by a period, a single digit means tenths-of-a-second, two digits mean hundredths-of-a-second, and three digits mean thousandths-of-a-second. For example, 12:30:20:1 indicates twenty and one-thousandth seconds past 12:30; 12:30:20.1 indicates twenty and one-tenth seconds past 12:30.

See Also

Date and Time Data datetime and smalldatetime

ODBC Datetime Format

The ODBC API defines escape sequences to represent date and time values, which ODBC calls timestamp data. This ODBC timestamp format is also supported by the OLE DB language definition (DBGUID-SQL) supported by the Microsoft OLE DB Provider for SQL Server. Applications using the ADO, OLE DB, and ODBC-based APIs can use this ODBC timestamp format to represent dates and times.

ODBC timestamp escape sequences are of the format:

```
{ literal_type 'constant_value' }
```

literal_type

Specifies the type of the escape sequence. Timestamps have three *literal_type* specifiers:

d = date only

t = time only

ts = timestamp (time + date)

'constant_value'

Is the value of the escape sequence. *constant_value* must follow these formats for each *literal_type*.

literal_type	constant_value format
D	yyyy-mm-dd
Т	hh:mm:ss[.fff]
Ts	yyyy-mm-dd hh:mm:ss[.fff]

These are examples of ODBC time and date constants:

{ **ts** '1998-05-02 01:23:56.123' }

{ **d** '1990-10-02' }

{ **t** '13:33:41' }

Do not confuse the ODBC and OLE DB timestamp data type name with the Transact-SQL **timestamp** data type name. The ODBC and OLE DB timestamp data type records dates and times. The Transact-SQL **timestamp** data type is a binary data type with no time-related values.

Using Integer Data

Integers are whole numbers and contain no decimals or fractions.

Microsoft[®] SQL Server[™] 2000 has the following sizes of integer data types:

• bigint

Has a length of 8 bytes and stores numbers from -2^63 (-9,223,372,036,854,775,808) through 2^63-1 (9,223,372,036,854,775,807).

• integer or int

Has a length of 4 bytes, and stores numbers from -2,147,483,648 through 2,147,483,647.

• smallint

Has a length of 2 bytes, and stores numbers from -32,768 through 32,767.

• tinyint

Has a length of 1 byte, and stores numbers from 0 through 255.

Integer objects and expressions can be used with any mathematical operations. Any fractions generated by these operations are truncated, not rounded. For example, SELECT 5/3 returns a value of 1, not the value 2, which would return if the fractional result were rounded.

The integer data types are the only ones that can be used with the IDENTITY property, which is an automatically incrementing number. The IDENTITY property is typically used to automatically generate unique identification numbers or primary keys.

Integer data does not need to be enclosed in single quotation marks like character or date and time data.

See Also

<u>CAST and CONVERT</u> int, bigint, smallint, and tinyint <u>Numeric Data</u>

Using bigint Data

The **bigint** data type is intended for use in cases where integer values might exceed the range supported by the **int** data type. For compatibility, the **int** data type remains the primary integer data type in Microsoft® SQL Server[™] 2000.

Unless explicitly stated, functions, statements, and system stored procedures that accept **int** expressions for their parameters have not been changed to support implicit conversion of **bigint** expressions to those parameters. Thus, SQL Server only implicitly converts **bigint** to **int** when the **bigint** value is within the range supported by the **int** data type. A conversion error occurs at run time if the **bigint** expression contains a value outside the range supported by the **int** data type.

bigint in Transact-SQL Functions

Although SQL Server sometimes promotes **tinyint** or **smallint** values to **int** data type, it will not automatically promote **tinyint**, **smallint**, or **int** to **bigint**. For example, if the data type of the parameter expression is **tinyint** or **smallint**, certain aggregate functions promote the data type of the return value to an **int**. These aggregate functions will not return a **bigint** unless the parameter expression is of type **bigint**.

When you specify **bigint** parameters and the return values are of type **bigint**, you may use the following Transact-SQL functions.

ABS	FLOOR	POWER
AVG	IDENTITY	RADIANS
CEILING	MAX	ROUND
COALESCE	MIN	SIGN
DEGREES	NULLIF	SUM

When you reference **bigint** columns or variables, but you are not looking for **bigint** data types for return values, you may use these functions.

@@IDENTITY	ISNULL	VARP
COL_LENGTH	ISNUMERIC	

DATALENGTH	STDEV[P]	
------------	----------	--

SQL Server provides these functions specifically for use with **bigint** values.

COUNT_BIG

Use when counting the number of items in a group if the value exceeds the range supported by the **int** data type, and returns **bigint**. COUNT_BIG is like the COUNT function except for the return type.

ROWCOUNT_BIG

Use when counting the number of rows affected in the last statement executed and the value exceeds the range supported by the **int** data type. This function is similar to the ROWCOUNT function, except that ROWCOUNT_BIG returns a **bigint** data type.

bigint in Other Transact-SQL Elements

The CAST and CONVERT clauses support **bigint**. These clauses apply similar conversion rules for **bigint** as for the other integer data types. The **bigint** data type fits above **int** and below **smallmoney** in the data type precedence chart. For more information about **bigint** conversions, see <u>CAST and CONVERT</u>.

When using the CASE expression, you will get a result of type **bigint** if the *result_expression* or the optional *else_result_expression* evaluate to **bigint**.

You may use the **bigint** data type in all syntax locations where integer data types are specified in these Transact-SQL statements:

- ALTER PROCEDURE
- ALTER TABLE
- CREATE PROCEDURE
- CREATE TABLE

• DECLARE variable

In addition, the SQL Server catalog components report information about **bigint** columns.

Specifying bigint Constants

Whole number constants that are outside the range supported by the **int** data type continue to be interpreted as **numeric**, with a scale of 0 and a precision sufficient to hold the value specified. For example, the constant 3000000000 is interpreted as **numeric**. These **numeric** constants are implicitly convertible to **bigint** and can be assigned to **bigint** columns and variables:

CREATE TABLE BigintTable (ColA bigint)

INSERT INTO BigintTable VALUES (300000000)

```
SELECT *
FROM BigintTable
WHERE ColA = 300000000
```

You can also cast constants to **bigint**:

CAST(300000000 AS bigint)

To get a **bigint** value into an **sql_variant** column, use this method:

CREATE TABLE VariantTable (ColA sql_variant)

-- Inserts a value with a numeric base data type.
INSERT INTO VariantTable VALUES (300000000)
-- Inserts a value with a bigint base data type.
INSERT INTO VariantTable VALUES (CAST(300000000 AS bigint

See Also

<u>CASE</u>

CAST and CONVERT COUNT_BIG int, bigint, smallint, and tinyint Numeric Data sql_variant

Using decimal, float, and real Data

The **decimal** data type can store a maximum of 38 digits, all of which can be to the right of the decimal point. The **decimal** data type stores an exact representation of the number; there is no approximation of the stored value.

The two attributes that define **decimal** columns, variables, and parameters are:

• p

Specifies the precision, or the number of digits the object can hold.

• s

Specifies the scale, or the number of digits that can be placed to the right of the decimal point.

p and *s* must observe the rule: $0 \le s \le p \le 38$.

The default maximum precision of **numeric** and **decimal** data types is 38. In Transact-SQL, **numeric** is functionally equivalent to the **decimal** data type.

Use the **decimal** data type to store numbers with decimals when the data values must be stored exactly as specified.

For more information about how mathematical operations affect the precision and scale of the result, see <u>Precision, Scale, and Length</u>.

Using float and real Data

The **float** and **real** data types are known as approximate data types. The behavior of **float** and **real** follows the IEEE 754 specification on approximate numeric data types.

Approximate numeric data types do not store the exact values specified for many numbers; they store an extremely close approximation of the value. For many applications, the tiny difference between the specified value and the stored approximation is not noticeable. At times, though, the difference becomes noticeable. Because of the approximate nature of the **float** and **real** data types, do not use these data types when exact numeric behavior is required, such as in

financial applications, in operations involving rounding, or in equality checks. Instead, use the integer, **decimal**, **money**, or **smallmoney** data types.

Avoid using **float** or **real** columns in WHERE clause search conditions, especially the = and <> operators. It is best to limit **float** and **real** columns to > or < comparisons.

The IEEE 754 specification provides four rounding modes: round to nearest, round up, round down, and round to zero. Microsoft® SQL Server[™] uses round up. All are accurate to the guaranteed precision but can result in slightly different floating-point values. Because the binary representation of a floating-point number may use one of many legal rounding schemes, it is impossible to reliably quantify a floating-point value.

See Also

Data Types

Numeric Data

Using Monetary Data

Microsoft® SQL ServerTM stores monetary data (currency values) using two data types: **money** and **smallmoney**. These data types can use any one of the following currency symbols.

Symbol	Name	Hexadecimal value
\$	Dollar sign (U. S.)	0x24
£	Pound sign (U. K.)	0xA3
a	Currency sign	0xA4
¥	Yen sign	0xA5
•	Bengali Rupee mark	0x9F2
ե	Bengali Rupee sign	0x9F3
В	Thai Baht symbol	0xE3F
€	Euro-Currency sign	0x20A0
¢	Colon sign	0x20A1
Ē	Cruzeiro sign	0x20A2
Г	French Franc sign	0x20A3
£	Lira sign	0x20.A4
N	Naira sign	0x20A6
₽	Peseta sign	0x20A7
₽s	Rupee sign	0x20A8
₩	Won sign	0x20A9
N	New Sheqel sign	0x20AA
₫	Dong sign	0x20AB

Currency or monetary data does not need to be enclosed in single quotation marks ('). However, the monetary data value must be preceded by the appropriate currency symbol. For example, to specify 100 English pounds, use £100.

money and **smallmoney** are limited to four decimal points. Use the **decimal**

data type if more decimal points are required.

Use a period to separate partial monetary units, like cents, from whole monetary units. For example, 2.15 specifies 2 dollars and 15 cents.

Comma separators are not allowed in **money** or **smallmoney** constants, although the display format of these data types includes comma separators. You can specify the comma separators only in character strings explicitly cast to **money** or **smallmoney**, for example:

```
USE Northwind
GO
CREATE TABLE TestMoney (cola INT PRIMARY KEY, colb MONE
GO
SET NOCOUNT ON
GO
```

-- The following three INSERT statements work. INSERT INTO TestMoney VALUES (1, \$123.45) GO INSERT INTO TestMoney VALUES (2, \$123123.45) GO INSERT INTO TestMoney VALUES (3, CAST('\$444,123.45' AS MOI GO

-- This INSERT statement gets an error because of the comma -- separator in the money string. INSERT INTO TestMoney VALUES (3, \$555,123.45) GO SET NOCOUNT OFF GO SELECT * FROM TestMoney GO

See Also

<u>Data Types</u>

Monetary Data

money and smallmoney

Using text and image Data

Microsoft® SQL Server[™] 2000 stores character strings longer than 8,000 characters and binary data longer than 8,000 bytes in special data types named **text** and **image**. Unicode strings longer than 4,000 characters are stored in the **ntext** data type.

For example, a large text file (.txt) of customer information must be imported into your SQL Server database. This data should be stored as one piece of data rather than integrated into the multiple columns of your data tables. You can create a column with the **text** data type for this purpose. However, if you must store company logos currently stored as Tagged Image File Format (TIFF) images (.tif) that are 10 KB each, create a column with the **image** data type.

If the textual data to be stored is in Unicode format, use the **ntext** data type. For example, a form letter created for international customers is likely to contain international spellings and characters used in various different languages. Store this data in an **ntext** column.

Each **text** and **ntext** data value has a collation. Collations define attributes such as comparison rules and sensitivity to case or accenting. The collations for **text** values also specify a code page, which defines the bit patterns used to represent each character. Each **ntext** value uses the Unicode code page, which is the same for all the collations. Each database has a default collation. When a **text** or **ntext** column is created, it is assigned the default collation of the database unless you assign a specific collation using the COLLATE clause. When two **text** or **ntext** values having different collations are combined or compared, collation precedence rules determine which collation is used for the operation.

Data in an **image** data is stored as a string of bits and is not interpreted by SQL Server. Any interpretation of the data in an **image** column must be made by the application. For example, an application could store data in an **image** column using a BMP, TIFF, GIF, or JPEG format. The application that reads the data from the **image** column must recognize the format of the data and display it correctly. All an **image** column does is provide a location to store the stream of bits that make up the image data value.

Using text in row to Store text, ntext, and image Values

Usually, **text**, **ntext**, or **image** strings are large (a maximum of 2GB) character or binary strings 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.

With Microsoft SQL Server, you can store small to medium **text**, **ntext**, and **image** values in a data row, thereby increasing the speed of queries accessing these values.

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.

To store **text**, **ntext**, or **image** strings in the data row, enable the **text in row** option using the **sp_tableoption** stored procedure.

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

Optionally, you can specify a maximum limit, from 24 through 7000 bytes, for the length of a **text**, **ntext**, and **image** string 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 provides most of the performance benefits: It 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.

Although in general, you should not set the value below 72, you also should not set the value too high, especially for tables where most statements do not reference the **text**, **ntext**, and **image** columns or there are multiple **text**, **ntext**, and **image** columns.

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'

See Also

Character Data

Data Types

<u>sp_tableoption</u>

Using uniqueidentifier Data

The **uniqueidentifier** data type stores 16-byte binary values that operate as globally unique identifiers (GUIDs). A GUID is a unique binary number; no other computer in the world will generate a duplicate of that GUID value. The main use for a GUID is for assigning an identifier that must be unique in a network that has many computers at many sites.

A GUID value for a **uniqueidentifier** column is usually obtained:

- In a Transact-SQL statement, batch, or script by calling the NEWID function.
- In application code by calling an application API function or method that returns a GUID.

The Transact-SQL NEWID function and the application API functions and methods generate new **uniqueidentifier** values from the identification number of their network card plus a unique number from the CPU clock. Each network card has a unique identification number. The **uniqueidentifier** returned by NEWID is generated using the network card on the server. The **uniqueidentifier** returned by application API functions and methods is generated using the network card on the client.

A **uniqueidentifier** is not typically defined as a constant because it is difficult to ensure that the **uniqueidentifier** created is actually unique. There are two ways to specify a **uniqueidentifier** constant:

- Character string format '6F9619FF-8B86-D011-B42D-00C04FC964FF'
- Binary format 0xff19966f868b11d0b42d00c04fc964ff

The **uniqueidentifier** data type does not automatically generate new IDs for inserted rows the way the IDENTITY property does. To get new **uniqueidentifier** values, a table must have a DEFAULT clause specifying the

NEWID function, or INSERT statements must use the NEWID function:

```
CREATE TABLE MyUniqueTable
(UniqueColumn UNIQUEIDENTIFIER DEFAULT NEWID(),
Characters VARCHAR(10))
GO
INSERT INTO MyUniqueTable(Characters) VALUES ('abc')
INSERT INTO MyUniqueTable VALUES (NEWID(), 'def')
GO
```

uniqueidentifier columns may contain multiple occurrences of an individual **uniqueidentifier** value, unless the UNIQUE or PRIMARY KEY constraints are also specified for the column. A foreign key column referencing a **uniqueidentifier** primary key in another table will have multiple occurrences of individual **uniqueidentifier** values when multiple rows reference the same primary key in the source table.

A table can have multiple **uniqueidentifier** columns. One **uniqueidentifier** column for each table may be specified with the ROWGUIDCOL property. The ROWGUIDCOL property indicates that the **uniqueidentifier** values in the column uniquely identify rows in the table. The property does not do anything to enforce this, however. The uniqueness must be enforced through other mechanisms, such as specifying the PRIMARY KEY constraint for the column. The ROWGUIDCOL property is primarily used by SQL Server replication.

The main advantage of the **uniqueidentifier** data type is that the values generated by the Transact-SQL NEWID function or the application GUID functions are guaranteed to be unique throughout the world.

The **uniqueidentifier** data type has several disadvantages:

- The values are long and obscure. This makes them difficult for users to type correctly, and more difficult for users to remember.
- The values are random and cannot accept any patterns that may make them more meaningful to users.

- There is no way to determine the sequence in which **uniqueidentifier** values were generated. They are not suited for existing applications that depend on incrementing key values serially.
- At 16 bytes, the **uniqueidentifier** data type is relatively large compared to other data types such as 4-byte integers. This means indexes built using **uniqueidentifier** keys may be relatively slower than implementing the indexes using an **int** key.

Consider using the IDENTITY property when global uniqueness is not necessary, or when having a serially incrementing key is desirable.

See Also

Data Types

uniqueidentifier

Using Special Data

Special data types are those that do not fit into any of the other data type categories. For example, to store data as 1 or 0 corresponding to yes or no values in a customer survey, use the **bit** data type. Microsoft® SQL Server[™] 2000 has several data types that fit into this category:

• bit

bit data does not need to be enclosed in single quotation marks. It is numeric data similar to SQL Server integer and numeric data, except that only 0s and 1s can be stored in **bit** columns.

• sql_variant

The **sql_variant** data type in SQL Server allows a single column, parameter, or variable to store data values of different data types. Each instance of an **sql_variant** column records the data value and the metadata describing the value: its base data type, maximum size, scale, precision, and collation.

The second table in this example contains an **sql_variant** column:

```
CREATE TABLE ObjectTable
  (ObjectID
                 int
               CONSTRAINT PKObjectTable PRIMARY K
  ObjectName
                   nvarchar(80),
  ObjectWeight
                  decimal(10,3),
  ObjectColor
                nvarchar(20)
  )
CREATE TABLE VariablePropertyTable
  (ObjectID
                 int REFERENCES ObjectTable(ObjectID)
  PropertyName
                   nvarchar(100),
  PropertyValue
                  sql_variant,
  CONSTRAINT PKVariablePropertyTable
          PRIMARY KEY(ObjectID, PropertyName)
```

)

To get the meta data information for any specific **sql_variant** instance, use the SQL_VARIANT_PROPERTY function.

• table

table data type is like temporary tables and can be used to store a result set for later processing. This data type can only be used to define local variables of type **table** and the return value of a user-defined function.

The definition of a table variable or return value includes definitions of the columns, the data type, precision, and scale of each column, and optional PRIMARY KEY, UNIQUE, and CHECK constraints.

The format of the rows stored in a **table** variable or returned by a userdefined function must be defined when the variable is declared or the function is created. The syntax is based on the CREATE TABLE syntax. For example:

DECLARE @TableVar TABLE (Cola int PRIMARY KEY, Colb char(3))

INSERT INTO @TableVar VALUES (1, 'abc') INSERT INTO @TableVar VALUES (2, 'def')

SELECT * FROM @TableVar GO

table variables and user-defined functions that return a **table** can be used only in certain SELECT and INSERT statements, and where tables are supported in the UPDATE, DELETE, and DECLARE CURSOR statements. **table** variables and user-defined functions that return a **table** cannot be used in any other Transact-SQL statements.

Indexes or other constraints applied to the table must be defined as part of the DECLARE variable or CREATE FUNCTION statement. They cannot be applied later, because the CREATE INDEX or ALTER TABLE statements cannot reference table variables and user-defined functions.

For more information about the syntax used to define the **table** variables and user-defined functions, see <u>DECLARE @local_variable</u> and <u>CREATE FUNCTION</u>.

• timestamp

The SQL Server **timestamp** data type has nothing to do with times or dates. SQL Server **timestamps** are binary numbers that indicate the relative sequence in which data modifications took place in a database. The **timestamp** data type was originally implemented to support the SQL Server recovery algorithms. Each time a page was modified, it was stamped with the current @@DBTS value and @@DBTS was incremented by one. This was sufficient for recovery to determine the relative sequence in which pages had been modified, but the **timestamp** values had no relationship to time.

In SQL Server version 7.0 and SQL Server 2000, @@DBTS is only incremented for use in **timestamp** columns. If a table contains a **timestamp** column, every time a row is modified by an INSERT, UPDATE, or DELETE statement, the **timestamp** value in the row is set to the current @@DBTS value, and then @@DBTS is incremented by one.

Never use **timestamp** columns in keys, especially primary keys, because the **timestamp** value changes every time the row is modified.

To record the times data modifications take place in a table, use either a **datetime** or **smalldatetime** data type to record the events and triggers to automatically update the values when any modification takes place.

• User-defined data types

User-defined data types allow you to extend an SQL Server base data type (such as **varchar**) with a descriptive name and format tailored to a specific use. For example, this statement implements a **birthday** user-defined data type that allows NULLs, using the **datetime** base data type:

EXEC sp_addtype birthday, datetime, 'NULL'

Be careful when choosing the base types for implementing user-defined data types. For example, in the United States, Social Security numbers have a format of *nnn-nn-nnn*. While Social Security numbers contain numbers, the numbers form an identifier and are not subjected to mathematical operations. It is therefore common practice to create a user-defined Social Security number data type as **varchar** and create a CHECK constraint to enforce the format of the social security numbers stored in the table:

If the **SSN** columns are typically used as key columns in indexes, especially clustered indexes, the size of the keys can be shrunk from 11 bytes to 4 if the **SSN** user-defined data type is instead implemented using the **int** base data type. This reduction in key size improves data retrieval. The improved efficiency of data retrieval and the elimination of the need for the CHECK constraint will usually outweigh the extra conversion processing from **int** to a character format when displaying or modifying **SSN** values.

See Also

CREATE FUNCTION

Data Types

DECLARE @local_variable

<u>Special Data</u>

SQL_VARIANT_PROPERTY

<u>table</u>

Using sql_variant Data

The **sql_variant** data type operates similarly to the **variant** data type in Microsoft® Visual Basic®. It allows a single column, parameter, or variable to store data values of different data types. For example, a single **sql_variant** column can hold **int**, **decimal**, **char**, **binary**, and **nchar** values. Each instance of an **sql_variant** column records the data value and the meta data information, which includes the base data type, maximum size, scale, precision, and collation.

The **sql_variant** data type follows these rules:

- 1. General value assignment
 - **sql_variant** objects can hold data of any SQL Server data type except **text**, **ntext**, **image**, and **timestamp**. An instance of **sql_variant** data also cannot have **sql_variant** as its underlying base data type.
 - Constants of any type can be specified in predicates or assignments referencing **sql_variant** columns.
 - If an **sql_variant** value is NULL, it is not considered to have an underlying base data type. This rule applies even when the null value comes from a variable or column with a specific data type.

In this example, the value of **VariantCol** is set to NULL with no associated data type, even though the null value came from an **int** variable:

DECLARE @IntVar int SET @IntVar = NULL UPDATE SomeTable SET VariantCol = @IntVar WHI

• In assignments from **sql_variant** objects to an object with any other data type, the **sql_variant** value must be explicitly cast

to the data type of the destination. No implicit conversions are supported when an **sql_variant** value is assigned to an object with another data type.

- For compatibility with other data types, the catalog objects (such as the DATALENGTH function) that report the length of **sql_variant** objects report the length of the data. The length of the meta data contained in an **sql_variant** object is not returned.
- **sql_variant** columns always operate with ANSI_PADDING ON. If **char**, **nchar**, **varchar**, **nvarchar**, or **varbinary** values are assigned from a source that has ANSI_PADDING OFF, the values are not padded.
- 2. **sql_variant** in tables
 - **sql_variant** columns can be used in indexes and unique keys, as long as the length of the data in the key columns does not exceed 900 bytes.
 - **sql_variant** columns do not support the IDENTITY property, but **sql_variant** columns are allowed as part primary or foreign keys.
 - **sql_variant** columns cannot be used in a computed column.
 - Use ALTER TABLE to change a column of any data type except **text**, **ntext**, **image**, **timestamp**, or **sql_variant** to **sql_variant**. All existing values are converted to **sql_variant** values whose base data type is the same as the data type of the column before the ALTER TABLE statement was executed. ALTER TABLE cannot be used to change the data type of an **sql_variant** column to any other data type because there are no supported implicit conversions from **sql_variant** to other

data types.

- 3. Collation
 - The COLLATE clause cannot be used to assign a column collation to an sql_variant column. The character-based values (char, nchar, varchar, and nvarchar) in an sql_variant column can be of any collation, and a single sql_variant column can hold character-based values of mixed collations.
 - When a value is assigned to an sql_variant instance, both the data value and base data type of the source are assigned. If the source value has a collation, the collation is also assigned. If the source value has a user-defined data type, the base data type of the user-defined data type is assigned, not the user-defined data type. The sql_variant instance does not inherit any rules or defaults bound to the user-defined data type. If a value from a column with an identity property is assigned to an sql_variant instance, the sql_variant takes the base data type of the source column but does not inherit the IDENTITY property. It is an error to assign a text, ntext, or image value to an sql_variant instance. Implicit conversions are supported when assigning values from objects with other data types to an sql_variant object.

sql_variant Comparisons

sql_variant columns can contain values of several base data types and collations, so special rules apply when comparing **sql_variant** operands. These rules apply to operations involving comparisons, such as:

- Transact-SQL comparison operators
- ORDER BY, GROUP BY
- Indexes

- The MAX and MIN aggregate functions
- UNION (without ALL)
- CASE expressions

For **sql_variant** comparisons, the SQL Server data type hierarchy order is grouped into data type families (the **sql_variant** family has the highest family precedence).

Data type hierarchy	Data type family
sql_variant	sql_variant
datetime	Datetime
smalldatetime	Datetime
float	approximate number
real	approximate number
decimal	exact number
money	exact number
smallmoney	exact number
bigint	exact number
int	exact number
smallint	exact number
tinyint	exact number
bit	exact number
nvarchar	Unicode
nchar	Unicode
varchar	Unicode
char	Unicode
varbinary	Binary
binary	Binary
uniqueidentifier	Uniqueidentifier

These rules apply to **sql_variant** comparisons:

- When **sql_variant** values of different base data types are compared, and the base data types are in different data type families, the value whose data type family is higher in the hierarchy chart is considered the higher of the two values.
- When **sql_variant** values of different base data types are compared, and the base data types are in the same data type family, the value whose base data type is lower in the hierarchy chart is implicitly converted to the other data type and the comparison is then made.
- When **sql_variant** values of the **char**, **varchar**, **nchar**, or **varchar** data types are compared, they are evaluated based on the following criteria: LCID, LCID version, comparison flags, and sort ID. Each of these criteria are compared as integer values, and in the order listed.

These rules can yield different results for comparisons between **sql_variant** values than comparisons between values of the same base data type.

Operand A			sql_variant comparison result
'123' char	111 int	A > B	B > A
50000 int	5E1 float	A > B	B > A

Because values from different data type families must be explicitly cast before being referenced in comparison predicates, the effects of the rules are observed only when ordering result sets on an **sql_variant** column. The values in this table are examples of the rules regarding data type precedence.

PriKey	VariantCol
1	50.0 (base type float)
2	5000 (base type int)
3	'124000' (base type char(6))

This is the result of the statement SELECT * FROM VariantTest ORDER BY VariantCol ASC.

PriKey	VariantCol
3	'124000' (base type char(6))
2	5000 (base type int)
1	50.0 (base type float)

The values in this table are examples of the rules regarding collation precedence using different collations.

IntKey	VariantCol
1	<pre>qrs (varchar SQL_Latin1_General_Pref_Cp1_CI_AS)</pre>
2	abc (varchar SQL_Latin1_General_Pref_Cp1_CI_AS)
3	qrs (varchar SQL_Latin1_General_CP1_CS_AS)
4	17.5 (decimal)
5	abc (varchar SQL_Latin1_General_CP1_CS_AS)
6	klm (varchar SQL_Latin1_General_CP1_CS_AS)
7	1.2 (decimal)

This is the result of the statement SELECT * FROM CollateTest ORDER BY VariantCol. This table shows values from the exact number data type family grouped together, and **varchar** values grouped within their respective collations.

IntKey	VariantCol
5	abc (varchar SQL_Latin1_General_CP1_CS_AS)
6	klm (varchar SQL_Latin1_General_CP1_CS_AS)
3	qrs (varchar SQL_Latin1_General_CP1_CS_AS)
2	abc (varchar SQL_Latin1_General_Pref_Cp1_CI_AS)
1	qrs (varchar SQL_Latin1_General_Pref_Cp1_CI_AS)
7	1.2 (decimal)
4	17.5 (decimal)

Functions and sql_variant Data

The following Transact-SQL functions support **sql_variant** parameters and return an **sql_variant** value when an **sql_variant** parameter is specified:

- COALESCE
- MAX
- MIN
- NULLIF

These functions support references to **sql_variant** columns or variables and do not use **sql_variant** as the data type of their return values.

COL_LENGTH	DATALENGTH	TYPEPROPERTY
COLUMNPROPERTY	ISNULL	

These Transact-SQL functions do not support **sql_variant** parameters.

AVG	RADIANS	STDEV[P]
IDENTITY	ROUND	SUM
ISNUMERIC	SIGN	VAR[P]
POWER		

The CAST and CONVERT functions support **sql_variant**

The new function SQL_VARIANT_PROPERTY(): is used to obtain property information about **sql_variant** values, such as data type, precision, or scale.

Other Transact-SQL Elements and sql_variant Data

sql_variant columns are not supported in the LIKE predicate.

sql_variant columns are not supported in full-text indexes. They cannot be specified in full-text functions such as CONTAINSTABLE and FREETEXTTABLE.

These Transact-SQL statements support specifying **sql_variant** in the same syntax locations that other integer data types are specified:

- ALTER PROCEDURE
- ALTER TABLE
- CREATE PROCEDURE
- CREATE TABLE
- DECLARE variable

The Microsoft® SQL Server[™] 2000 catalog components report information about **sql_variant** columns.

The result of the CASE expression is **sql_variant** if any of the input or result expressions evaluate to **sql_variant**. The underlying base type of the result is that of the expression evaluated as the result at run time.

Operands of numeric or string concatenation operators cannot be **sql_variant**:

-- Generates an error: SELECT VariantCol + @CharacterVar FROM MyTable

Casting the **sql_variant** operand can perform the operation:

-- Does not generates an error: SELECT CAST(VariantCol AS varchar(25)) + @CharacterVar FROM MyTable

Applications and sql_variant Data

If an application requests a result set in which a given column returns **sql_variant** data of a single underlying base data type, the application can use the CAST or CONVERT functions in the Transact-SQL statements to return the **sql_variant** data by using the underlying base data type. In this case the application treats the data the same way as a result set column of the underlying base data type. This topic describes how Microsoft® SQL ServerTM returns **sql_variant** data that has not been cast or converted to a specific base data type.

The OLE DB Provider for SQL Server introduces a provider-specific OLE DB type DBTYPE_SQLVARIANT for use with **sql_variant** columns and parameters.

The SQL Server ODBC Driver introduces a provider-specific ODBC database data type SQL_SS_VARIANT for use with **sql_variant** columns and parameters.

SQL Server converts **sql_variant** values to **nvarchar(4000)** when working with applications that have connected with the following interfaces:

- The OLE DB Provider for SQL Server version 7.0.
- The SQL Server ODBC Driver from SQL Server 7.0.

If the resulting string exceeds 4000 characters, SQL Server returns the first 4000 characters.

SQL Server converts **sql_variant** values to **varchar(255)** when working with applications that have connected with the following interfaces:

- The SQL Server ODBC Drivers from SQL Server version 6.5 or earlier.
- Any version of the DB-Library dll.

If the resulting string exceeds 255 characters, SQL Server returns the first 255 characters.

See Also

CAST and CONVERT

<u>sql_variant</u>

SQL_VARIANT_PROPERTY

Using Unicode Data

The Unicode specification defines a single encoding scheme for most characters widely used in businesses around the world. All computers consistently translate the bit patterns in Unicode data into characters using the single Unicode specification. This ensures that the same bit pattern is always converted to the same character on all computers. Data can be freely transferred from one database or computer to another without concern that the receiving system will translate the bit patterns into characters incorrectly.

One problem with data types that use 1 byte to encode each character is that the data type can only represent 256 different characters. This forces multiple encoding specifications (or code pages) for different alphabets such as European alphabets, which are relatively small. It is also impossible to handle systems such as the Japanese Kanji or Korean Hangul alphabets that have thousands of characters.

Each Microsoft® SQL Server[™] collation has a code page that defines what patterns of bits represent each character in **char**, **varchar**, and **text** values. Individual columns and character constants can be assigned a different code page. Client computers use the code page associated with the operating system locale to interpret character bit patterns. There are many different code pages. Some characters appear on some code pages, but not on others. Some characters are defined with one bit pattern on some code pages, and with a different bit pattern on other code pages. When you build international systems that must handle different languages, it becomes difficult to pick code pages for all the computers that meet the language requirements of multiple countries. It is also difficult to ensure that every computer performs the correct translations when interfacing with a system using a different code page.

The Unicode specification addresses this problem by using 2 bytes to encode each character. There are enough different patterns (65,536) in 2 bytes for a single specification covering the most common business languages. Because all Unicode systems consistently use the same bit patterns to represent all characters, there is no problem with characters being converted incorrectly when moving from one system to another. You can minimize character conversion issues by using Unicode data types throughout your system. In Microsoft SQL Server, these data types support Unicode data:

- nchar
- nvarchar
- ntext

Note The n prefix for these data types comes from the SQL-92 standard for National (Unicode) data types.

Use of **nchar**, **nvarchar**, and **ntext** is the same as **char**, **varchar**, and **text**, respectively, except that:

- Unicode supports a wider range of characters.
- More space is needed to store Unicode characters.
- The maximum size of **nchar** and **nvarchar** columns is 4,000 characters, not 8,000 characters like **char** and **varchar**.
- Unicode constants are specified with a leading N: N'A Unicode string'.
- All Unicode data uses the same Unicode code page. Collations do not control the code page used for Unicode columns, only attributes such as comparison rules and case sensitivity.

See Also

Collations

Data Types

<u>NCHAR</u>

<u>UNICODE</u>

<u>Unicode Data</u>

Data Type Conversion

In Transact-SQL, two levels of data type conversions are possible:

- When data from one object is moved to, compared with, or combined with data from another object, the data may have to be converted from the data type of one object to the data type of the other.
- When data from a Transact-SQL result column, return code, or output parameter is moved into a program variable, it must be converted from the Microsoft® SQL Server[™] data type to the data type of the variable.

There are two categories of data type conversions:

• Implicit conversions are invisible to the user.

SQL Server automatically converts the data from one data type to another. For example, if a **smallint** is compared to an **int**, the **smallint** is implicitly converted to **int** before the comparison proceeds.

• Explicit conversions use the CAST or CONVERT functions.

The CAST and CONVERT functions convert a value (a local variable, a column, or another expression) from one data type to another. For example, the following CAST function converts the numeric value of \$157.27 into a character string of '\$157.27':

CAST (\$157.27 AS VARCHAR(10))

CAST is based on the SQL-92 standard and is preferred over CONVERT.

When converting from the data type of one SQL Server object to another, some implicit and explicit data type conversions are not supported. For example, an **nchar** value cannot be converted to an **image** value at all. An **nchar** can only be converted to **binary** using explicit conversion; an implicit conversion to **binary** is not supported. An **nchar** can be either explicitly or implicitly converted to **nvarchar**.

When handling **sql_variant** data types, SQL Server supports implicit

conversions of objects with other data types to **sql_variant** type. However, SQL Server does not support implicit conversions from **sql_variant** data to an object with another data type.

For more information about supported conversions between SQL Server objects, see <u>CAST and CONVERT</u>.

When converting between an application variable and an SQL Server result set column, return code, parameter, or parameter marker, the supported data type conversions are defined by the database application programming interface.

Moving Data to Program Variables

Applications accessing Microsoft® SQL Server[™] 2000 databases through a database application programming interface (API) must move data between application variables and:

• Result set columns.

Applications must move data from the columns of a fetched row in a result set into application variables.

• Return codes.

Applications must move data from a stored procedure return code into an application variable.

• Parameters.

Applications must move data between stored procedure parameters and application variables. Parameters can be input or output parameters, so data movement can be either from the variable to the parameter, or from the parameter to the variable.

• Parameter markers.

ODBC and OLE DB parameter markers are used in SQL statements in place of either input expressions (such as in a WHERE clause search condition) or stored procedure parameters and return codes. Applications must move data from application variables and the expression replaced by the parameter marker. For more information, see <u>Parameter Markers</u>.

Many database APIs use the concept of binding to specify how the data is to be moved between an application variable and the SQL Server object. Database APIs provide functions that an application can call to:

• Determine the data type, size, precision, and scale of a result set column, return code, parameter, or parameter marker. After the application has received this information it can allocate a variable or an

array of variables with compatible attributes.

- Bind the result set column, return code, parameter, or parameter marker to a specific variable or array of variables. The binding information typically includes:
 - The address and attributes (data type, size, precision, and scale) of the variable.
 - The name and attributes of the database object.

Data movement typically occurs:

- When a Transact-SQL statement or batch is executed. The OLE DB provider or ODBC driver pulls in the data bound to any input parameters or parameter markers and includes them in the packet sent to SQL Server.
- When a result set row is fetched. The OLE DB provider or ODBC driver moves the data for each column to the bound variables.
- After all the result sets from a stored procedure have been fetched or canceled. The OLE DB provider or ODBC driver moves the data for any output parameters or return codes to their bound variables.

The bound application variables are not required to have the same data type as the SQL Server object to which they are bound. If the data types are different, the OLE DB provider or ODBC driver converts the data when it is moved. The set of conversions supported by each OLE DB provider and ODBC driver are specified in the documentation for the provider or driver.

See Also

Using Variables and Parameters

Converting binary and varbinary Data

When data is converted from a string data type (**char**, **varchar**, **nchar**, **nvarchar**, **binary**, **varbinary**, **text**, **ntext**, or **image**) to a **binary** or **varbinary** data type of unequal length, Microsoft® SQL Server[™] pads or truncates the data on the right. When other data types are converted to **binary** or **varbinary**, the data is padded or truncated on the left. Padding is done with hexadecimal zeros.

Converting data to the **binary** and **varbinary** data types is useful if binary data is the easiest way to move data around. For any value of any type, converting that value to a binary value of large enough size, and then back to the type, will always result in the same value if both conversions on the same version of SQL Server. The binary representation of a value may change from release to release of SQL Server.

The conversion of **money**, **datetime**, **smalldatetime**, and **numeric** data types to or from **binary** or **varbinary** are different from earlier versions of SQL Server because of a change in storage representation.

You can convert **int**, **smallint**, and **tinyint** to **binary** or **varbinary**, but if you convert the **binary** value back to an integer value, it will be different from the original integer value if truncation has occurred. For example, this SELECT statement shows that the integer value 123456 is usually stored as a binary 0x0001e240:

SELECT CAST(123456 AS BINARY(4))

This SELECT statement shows that if the binary target is too small to hold the entire value, the leading digits are silently truncated so that the same number is stored as 0xe240:

SELECT CAST(123456 AS BINARY(2))

This batch shows that the silent truncation can affect arithmetic operations without raising an error:

DECLARE @BinaryVariable2 BINARY(2)

```
SET @BinaryVariable2 = 123456
SET @BinaryVariable2 = @BinaryVariable2 + 1
```

```
SELECT CAST( @BinaryVariable2 AS INT)
GO
```

The final result is 57921, not 123457.

Note Conversions between any data type and the **binary** data types are not guaranteed to be the same between SQL Server versions.

See Also

CAST and CONVERT

Data Types

Converting bit Data

Converting to **bit** promotes any nonzero value to 1.

See Also

CAST and CONVERT

Data Types

Converting Character Data

When character expressions are converted to a character data type of a different size, values too long for the new data type are truncated.

When a character expression is converted to a character expression of a different data type or size (such as from **char(5)** to **varchar(5)**, or **char(20)** to **char(15)**), the collation of the input value is assigned to the converted value. If a non-character expression is converted to a character data type, the default collation of the current database is assigned to the converted value. In either case, you can assign a specific collation using the COLLATE clause.

Note Code page translations are supported for **char** and **varchar** data types, but not for **text** data type. As with previous versions of Microsoft® SQL Server[™], data loss during code page translations are not reported.

Character expressions being converted to an approximate **numeric** data type can include optional exponential notation (a lowercase e or uppercase E followed by an optional plus (+) or minus (-) sign and then a number).

Character expressions being converted to an exact **numeric** data type must consist of digits, a decimal point, and an optional plus (+) or minus (-). Leading blanks are ignored. Comma separators (such as the thousands separator in 123,456.00) are not allowed in the string.

Character expressions being converted to **money** or **smallmoney** data types can also include an optional decimal point and dollar sign (\$). Comma separators (as in \$123,456.00) are allowed.

This example shows how to convert data for display. This example converts sales data to character data prior to performing a string comparison and converts the current date to style 3, dd/mm/yy.

```
USE pubs
GO
SELECT title,
CAST(ytd_sales AS CHAR(12)),
```

CAST(GETDATE() AS CHAR(12)) FROM titles WHERE CAST(ytd_sales AS CHAR(20)) LIKE '1%' GO

This example converts a **uniqueidentifier** value to a **char** data type.

DECLARE @myid uniqueidentifier SET @myid = NEWID() SELECT CONVERT(char(255), @myid) AS 'Char' GO

This example converts the current date to style 3, dd/mm/yy.

```
SELECT CONVERT(char(12), GETDATE(), 3) GO
```

See Also

CAST and CONVERT

Data Types

Converting datetime and smalldatetime Data

When converting to **datetime**, Microsoft® SQL Server[™] 2000 rejects all values it cannot recognize as dates (including dates earlier than January 1, 1753). You can convert **datetime** values to **smalldatetime** when the date is in the proper range (from January 1, 1900 through June 6, 2079). The time value is rounded to the nearest minute.

This example converts **smalldatetime** and **datetime** values to **varchar** and **binary** data types, respectively.

DECLARE @mydate_sm SMALLDATETIME SET @mydate_sm = '4/05/98'

SELECT CAST(@mydate_sm AS VARCHAR) AS SM_DATE_VARGO

DECLARE @mydate DATETIME SET @mydate = '4/05/98'

SELECT CAST(@mydate AS BINARY) AS DATE_BINARY GO

Here is the result set:

(1 row(s) affected)

SM_DATE_VARCHAR

Apr 5 1998 12:00AM

(1 row(s) affected)

DATE_BINARY

(1 row(s) affected)

See Also

CAST and CONVERT

Data Types

Converting float and real Data

Values of **float** are truncated when converted to any integer type.

When converting from **float** or **real** to character data, the string function STR() is usually a better choice than CAST(), because STR() enables more control over formatting. For more information, see <u>Functions</u>.

See Also

CAST and CONVERT

Data Types

Converting money Data

When converting to **money** from integer data types, units are assumed to be in monetary units. For example, the integer value of 4 is converted to the **money** equivalent of 4 monetary units.

This example converts **smallmoney** and **money** values to **varchar** and **decimal** data types, respectively.

```
USE pubs
GO
DECLARE @mymoney_sm SMALLMONEY
SET @mymoney_sm = 3148.29
SELECT CAST(@mymoney_sm AS VARCHAR) AS "SM_MONEY
GO
DECLARE @mymoney MONEY
SET @mymoney = 3148.29
SELECT CAST(@mymoney AS DECIMAL) AS "MONEY DECIN
GO
```

Here is the result set:

```
SM_MONEY VARCHAR
```

```
-----
```

3148.29

(1 row(s) affected)

MONEY DECIMAL

3148

(1 row(s) affected)

See Also

CAST and CONVERT

Data Types

Converting decimal and numeric Data

For the **decimal** and **numeric** data types, Microsoft® SQL Server[™] considers each specific combination of precision and scale as a different data type. For example, **decimal(5,5)** and **decimal(5,0)** are considered different data types.

In Transact-SQL statements, a constant with a decimal point is automatically converted into a **numeric** data value, using the minimum precision and scale necessary. For example, the constant 12.345 is converted into a **numeric** value with a precision of 5 and a scale of 3.

Converting from **decimal** or **numeric** to **float** or **real** can result in some loss of precision. Converting from **int**, **smallint**, **tinyint**, **float**, **real**, **money**, or **smallmoney** to either **decimal** or **numeric** can result in overflow.

By default, SQL Server uses rounding when converting a number to a **decimal** or **numeric** value with a lower precision and scale. However, if the SET ARITHABORT option is ON, SQL Server raises an error when overflow occurs. Loss of only precision and scale is not sufficient to raise an error.

See Also

CAST and CONVERT

Data Types

Data Type Conversions Using OLE Automation Stored Procedures

Because Microsoft® SQL ServerTM uses Transact-SQL data types, and OLE Automation uses Microsoft Visual Basic® data types, the OLE Automation stored procedures must convert the data that passes between them.

The following table describes SQL Server to Visual Basic data type conversions.

SQL Server data type	Visual Basic data type
char, varchar, text, nvarchar, ntext	String
decimal, numeric	String
bit	Boolean
binary, varbinary, image	One-dimensional Byte()
	array
int	Long
smallint	Integer
tinyint	Byte
float	Double
real	Single
money, smallmoney	Currency
datetime, smalldatetime	Date
anything set to NULL	Variant set to Null

All single SQL Server values are converted to a single Visual Basic value with the exception of **binary**, **varbinary**, and **image** values. These values are converted to a one-dimensional Byte() array in Visual Basic. This array has a range of Byte(0 To *length*–1) where *length* is the number of bytes in the SQL Server **binary**, **varbinary**, or **image** values.

These are the conversions from Visual Basic data types to SQL Server data types.

Visual Basic data type	SQL Server data type	

Long, Integer, Byte, Boolean, Object	int
Double, Single	float
Currency	money
Date	datetime
String with 4000 characters or less	varchar/nvarchar
String with more than 4000 characters	text/ntext
One-dimensional Byte() array with 8000 bytes	varbinary
or less	
5 💔 5	image
8000 bytes	

See Also

How to create an OLE Automation object (Transact-SQL)

OLE Automation Sample Script

How to debug a custom OLE Automation server (Transact-SQL)

System Stored Procedures

Using Constants

A constant is a symbol that represents a specific data value. The format of a constant depends on the data type of the value it represents. Constants are also called literals. Some examples of constants are:

- Character strings: 'O"Brien' 'The level for job_id: %d should be between %d and %d.'
- Unicode strings: N'Michél'
- Binary string constants: 0x12Ef 0x69048AEFDD010E
- **bit** constants are represented by the numbers 0 or 1.
- datetime constants: 'April 15, 1998' '04/15/98' '14:30:24' '04:24 PM'
- integer constants:
 1894
 2
- decimal constants: 1894.1204
 2.0
- **float** and **real** constants:

101.5E5 0.5E-2

- money constants: \$12
 \$542023.14
- uniqueidentifier constants: 0xff19966f868b11d0b42d00c04fc964ff
 '6F9619FF-8B86-D011-B42D-00C04FC964FF'

For numeric constants, to specify the sign of the numeric value use the unary + and - operators:

+\$156.45 -73.52E8 -129.42 +442

Character and Unicode constants are assigned the default collation of the current database, unless you assign a specific collation using the COLLATE clause:

'abc' COLLATE French_CI_AI N'lustig' COLLATE German_Phonebook_CS_AS

Using Constants in Transact-SQL

In Transact-SQL, constants can be used in many ways. Here are some examples:

- As a constant value in an arithmetic expression: SELECT Price + \$.10 FROM MyTable
- As the data value a column is compared against in a WHERE clause: SELECT * FROM MyTable WHERE LastName = 'O"Brien'

- As the data value to be placed in a variable: SET @DecimalVar = -1200.02
- As the data value that should be placed in a column of the current row. This is specified with the SET clause of the UPDATE statement or the VALUES clause of an INSERT statement: UPDATE MyTable SET Price = \$99.99 WHERE PartNmbr = 1234

INSERT INTO MyTable VALUES (1235, \$88.88)

- As the character string that specifies the text of the message issued by a PRINT or RAISERROR statement: PRINT 'This is a message.'
- As the value to test for in a conditional statement such as an IF statement or CASE functions: IF (@@SALESTOTAL > \$100000.00) EXECUTE Give_Bonus_Procedure

See Also

<u>Constants</u>

INSERT

Expressions

LIKE

Operators

<u>PRINT</u>

ALTER TABLE

RAISERROR

CREATE TABLE

UPDATE DELETE WHERE

Using Functions

Microsoft® SQL Server[™] 2000 has built-in functions to perform certain operations. The function categories are:

Aggregate functions.

Perform operations that combine multiple values into one. Examples are COUNT, SUM, MIN, and MAX.

Configuration functions.

Are scalar functions that return information about configuration settings.

Cursor functions.

Return information about the status of a cursor.

Date and time functions.

Manipulate **datetime** and **smalldatetime** values.

Mathematical functions.

Perform trigonometric, geometric, and other numeric operations.

Meta data functions.

Return information on the attributes of databases and database objects.

Rowset functions.

Return rowsets that can be used in the place of a table reference in a Transact-SQL statement.

Security functions.

Return information about users and roles.

String functions.

Manipulate **char**, **varchar**, **nchar**, **nvarchar**, **binary**, and **varbinary** values. System functions.

Operate on or report on various system level options and objects.

System statistical functions.

Return information regarding the performance of SQL Server.

Text and image functions.

Manipulate **text** and **image** values.

Uses of Functions

Functions can be used or included in:

- The select list of a query using a SELECT statement to return a value. SELECT DB_NAME()
- A WHERE clause search condition of a SELECT or data-modification (SELECT, INSERT, DELETE, or UPDATE) statement to limit the rows that qualify for the query. SELECT * FROM [Order Details]
 WHERE Quantity = (SELECT MAX(Quantity) FROM [Order Details])
- The search condition (WHERE clause) of a view to make the view dynamically conform to the user or environment at run time. CREATE VIEW ShowMyEmploymentInfo AS SELECT * FROM Employees WHERE EmployeeID = SUSER_SID() GO
- Any expression.
- A CHECK constraint or trigger to check for specified values when data is inserted.
 CREATE TABLE SalesContacts

 (SalesRepID INT PRIMARY KEY CHECK (SalesRepID = ContactName VARCHAR(50) NULL,

```
ContactPhone VARCHAR(13) NULL)
• A DEFAULT constraint or trigger to supply a value in case one is not
 specified on an INSERT.
 CREATE TABLE SalesContacts
    (
               INT PRIMARY KEY CHECK (SalesRepID =
   SalesRepID
    ContactName
                 VARCHAR(50) NULL,
   ContactPhone
                 VARCHAR(13) NULL,
                 DATETIME DEFAULT GETDATE(),
   WhenCreated
              INT DEFAULT SUSER SID()
    Creator
    )
 GO
```

Functions are always used with parentheses, even when there is no parameter. An exception to this are the niladic functions (functions that take no parameters) used with the DEFAULT keyword. For more information about the DEFAULT keyword, see <u>ALTER TABLE</u> and <u>CREATE TABLE</u>, or <u>Defaults</u>.

The parameters to specify a database, computer, login, or database user are sometimes optional. If they are not specified, they default to the current database, host computer, login, or database user.

Functions can be nested (one function used inside another function).

Using Deterministic and Nondeterministic Functions

A function is either deterministic or nondeterministic. When it always returns the same result any time it is called with a specific set of input values, the function is called deterministic. When it returns different results each time it is called with a specific set of input values, it is nondeterministic.

A function's determinism can limit where it can be used. Only deterministic functions can be invoked in views and computed columns indexed.

For more information see <u>Deterministic and Nondeterministic Functions</u>.

Using System Functions

System functions allow you to access information from Microsoft® SQL Server[™] system tables without accessing the system tables directly.

This group of five pairs of system functions for databases, hosts, objects, logins, and users returns a name when given an identifier (ID) and returns an ID when given a name:

- DB_ID and DB_NAME
- HOST_ID and HOST_NAME
- OBJECT_ID and OBJECT_NAME
- SUSER_ID and SUSER_NAME (or SUSER_SID and SUSER_SNAME)
- USER_ID and USER_NAME

For example, use the DB_ID function to get a database ID number rather than executing a SELECT of the **sysobjects** table.

The following example shows how to retrieve the username for the current user who is logged on (using SQL Server Authentication):

SELECT SUSER_NAME()

The following functions are similar, but they do not occur in complementary pairs and they take more than one input parameter:

• COL_NAME

Returns a column name.

• COL_LENGTH

Returns a column length.

• INDEX_COL

Returns an index column name.

COL_LENGTH returns the length of a column, not the length of any individual strings stored in the column. Use the DATALENGTH function to determine the total number of characters in a specific value.

This example returns the column length and data length of the **LastName** column in the **Employees** table:

```
SELECT COL_LENGTH('Employees', 'LastName') AS Col_Length,
DATALENGTH(LastName) AS DataLength
FROM Employees
WHERE EmployeeID > 6
```

Note It is recommended that the system functions, Information Schema Views, or the system stored procedures be used to gain access to system information without querying the system tables directly. System tables can change significantly between versions of SQL Server.

See Also

System Functions ALTER TABLE DEFAULT Definitions CREATE TABLE

Using String Functions

String functions are used for various operations on character and binary strings, and they return values commonly needed for operations on character data. Most string functions can be used only on **char**, **nchar**, **varchar**, and **nvarchar** data types or the data types that implicitly convert to them. A few string functions can also be used on **binary** and **varbinary** data.

You can use string functions to:

- Retrieve only a portion of a string (SUBSTRING).
- Search for similarities in the sounds of a character string (SOUNDEX and DIFFERENCE).
- Find a starting position for a particular string in a column or expression. For example, the position number of the letter A in "What a beautiful day!"
- Concatenate strings into one string. For example, combining a first name, last name, and middle name or initial into a full name.
- Convert a nonstring value to a string value (such as converting the value of 15.7 stored as a **float** to **char**).
- Inserting a specific string into an existing string. For example, inserting the string "Once" into the existing string of "upon a time" to produce the string "Once upon a time".

When string functions operate on strings that have different collations, the collation of the result is determined using the rules of collation precedence. For more information, see <u>Collation Precedence</u>.

Using SUBSTRING

The SUBSTRING function returns a portion of either a character or binary string, or a text string, and takes three parameters:

- A character or binary string, a column name, or a string-valued expression that includes a column name.
- The position at which the substring should begin.
- The length (in number of characters, or in number of bytes for **binary**) of the string to be returned.

This example displays the first initial and last name of each employee, for example, A Fuller:

USE Northwind SELECT SUBSTRING(FirstName, 1, 1), LastName FROM Employees

This example displays the second, third, and fourth characters of the string constant abcdef:

SELECT x = SUBSTRING('abcdef', 2, 3)

Х

bcd

(1 row(s) affected)

See Also

String Functions

Comparing CHARINDEX and PATINDEX

The CHARINDEX and PATINDEX functions return the starting position of a pattern you specify. PATINDEX can use wildcard characters while CHARINDEX cannot.

These functions take two parameters:

- The pattern whose position you want. With PATINDEX, the pattern is a literal string that can contain wildcard characters. With CHARINDEX, the pattern is a literal string (no wildcard characters).
- A string-valued expression, usually a column name, in which Microsoft® SQL Server[™] searches for the specified pattern.

For example, find the position at which the pattern "wonderful" begins in a certain row of the **notes** column in the **titles** table.

USE pubs

SELECT CHARINDEX('wonderful', notes)

FROM titles

WHERE title_id = 'TC3218'

Here is the result set:

46

(1 row(s) affected)

If you do not restrict the rows to be searched, the query returns all rows in the table and it reports nonzero values for those rows in which the pattern was found, and zero for all others.

For example, to use wildcards to find the position at which the pattern "candies"

begins in any row of the Description column in the **Categories** table:

```
USE Northwind
GO
SELECT CategoryID, PATINDEX('%candies%', Description)AS POS
FROM Categories
WHERE PATINDEX('%candies%', Description) <> 0
```

If you do not restrict the rows to be searched, the query returns all rows in the table and reports nonzero values for those rows in which the pattern was found.

PATINDEX is useful with **text** data types; it can be used in a WHERE clause in addition to IS NULL, IS NOT NULL, and LIKE (the only other comparisons that are valid on **text** in a WHERE clause).

See Also

String Functions

Using STR

The STR function converts numbers to characters, with optional parameters for specifying the total length of the result, including the decimal point and the number of places after the decimal point.

Length and decimal parameters to STR (if supplied) should be positive. The default length is 10. The number is rounded to an integer by default or if the decimal parameter is 0. The specified length should be greater than or equal to the part of the number before the decimal point plus the number sign (if any):

This example converts the float expression of 123.45 to a character with a length of 6 characters and 2 decimal places.

SELECT STR(123.45, 6, 2)

Here is the result set:

123.45

(1 row(s) affected)

If the integer part of the expression converted to a character string exceeds the length specified in STR, STR returns ** for the specified length. For example, the number 1234567.89 has 7 digits to the left of the decimal point. If the length parameter on STR is 7 or more, the resulting string contains the integer and as many of the decimals as will fit. If the length parameter in STR is 6 or less, then asterisks are returned. For example, the batch:

SELECT STR(1234567.89, 7, 2) SELECT STR(1234567.89, 6, 2)

Here is the result set:

1234568

(1 row(s) affected)

-----*****

(1 row(s) affected)

STR offers more flexibility than CAST when converting **decimal** data types to characters because it gives explicit control over formatting.

See Also

String Functions

Using STUFF

The STUFF function inserts a string into another string. It deletes a specified length of characters in the first string at the start position and then inserts the second string into the first string at the start position.

If the start position or the length is negative, or if the starting position is larger than length of the first string, a null string is returned. If the length to delete is longer than the first string, it is deleted to the first character in the first string.

This example puts in the character string of "xyz" starting at the second character of the "abc" character expression, and replaces a total of three characters.

SELECT STUFF('abc', 2, 3, 'xyz')

Here is the result set:

axyz

(1 row(s) affected)

See Also

String Functions

Comparing SOUNDEX and DIFFERENCE

The SOUNDEX function converts a character string to a four-digit code for use in a comparison. Vowels are ignored in the comparison. Nonalphabetic characters are used to terminate the comparison. This function always returns some value.

This example displays the results of the SOUNDEX function for the similar character strings of "Smith" and "Smythe". When character strings are similar, both strings have the same SOUNDEX codes.

SELECT SOUNDEX ('smith'), SOUNDEX ('smythe')

Here is the result set:

----- ----S530 S530

(1 row(s) affected)

The DIFFERENCE function compares the SOUNDEX values of two strings and evaluates the similarity between them, returning a value from 0 through 4, where 4 is the best match. This example returns a DIFFERENCE of 4 for the first SELECT because "Smithers" and "Smothers" differ by only one character.

SELECT DIFFERENCE('smithers', 'smothers')

Here is the result set:

------4 (1 row(s) affected)

The following example returns a DIFFERENCE of 3, indicating that the two character strings have a similar sound even though they differ in several characters.

SELECT DIFFERENCE('Jeff', 'Geoffe')

See Also

String Functions

Using text, ntext, and image Functions

There are two **text**, **ntext**, and **image** functions used exclusively for operations on **text**, **ntext**, and **image** data:

- TEXTPTR returns a **binary(16)** object containing a pointer to a **text**, **ntext**, or **image** instance. The pointer remains valid until the row is deleted.
- TEXTVALID function checks whether a specified text pointer is valid or not.

Text pointers are passed to the READTEXT, UPDATETEXT, WRITETEXT, PATINDEX, DATALENGTH, and SET TEXTSIZE Transact-SQL statements used to manipulate **text**, **ntext**, and **image** data.

In Transact-SQL statements, **text**, **ntext**, and **image** data is always referenced using pointers or the address of the data.

This example uses the TEXTPTR function to locate the **text** column (**pr_info**) associated with **pub_id** 0736 in the **pub_info** table of the **pubs** database. It first declares the local variable **@val**. The text pointer (a long binary string) is then put into **@val** and supplied as a parameter to the READTEXT statement, which returns 10 bytes starting at the fifth byte (offset of 4).

```
USE pubs
DECLARE @val varbinary(16)
SELECT @val = textptr(pr_info) FROM pub_info
WHERE pub_id = '0736'
READTEXT pub_info.pr_info @val 4 4
```

Here is the result set:

(1 row(s) affected)

pr_info

yet

Explicit conversion using the CAST function is supported from **text** to **varchar**, from **ntext** to **nvarchar**, and from **image** to **varbinary** or **binary**, but the **text** or **image** data is truncated to 8,000 bytes and **ntext** data is truncated at 4,000 characters (8,000 bytes). Conversion of **text**, **ntext**, or **image** to another data type is not supported, implicitly or explicitly. However, indirect conversion of **text**, **ntext** or **image** data can be done, for example:

CAST(CAST(text_column_name AS VARCHAR(10)) AS INT).

See Also

Text and Image Functions

Using Mathematical Functions

A mathematical function performs a mathematical operation on numeric expressions and returns the result of the operation. Mathematical functions operate on the Microsoft® SQL Server[™] system-supplied numeric data (**decimal**, **integer**, **float**, **real**, **money**, **smallmoney**, **smallint**, and **tinyint**). The precision of built-in operations on **float** data type data is six decimal places by default.

By default, a number passed to a mathematical function will be interpreted as a **decimal** data type. The CAST or CONVERT functions can be used to change the data type to something else, such as a **float**. For example, the value returned by the FLOOR function has the data type of the input value. The input of this SELECT statement is a **decimal**, and FLOOR returns 123, which is a decimal value:

SELECT FLOOR(123.45)

123

(1 row(s) affected)

But, this example uses a **float** value and FLOOR returns a **float** value:

SELECT FLOOR (CONVERT (float, 123.45))

123.000000

(1 row(s) affected)

A floating point underflow error occurs when the **float** or **real** result of a mathematical function is too small to display. A result of 0.0 is returned and no error message is displayed. For example, the mathematical calculation of 2 to the -100.0 power has a result 0.0.

Domain errors occur when the value provided in the mathematical function is not a valid value. For example, values specified for the ASIN function must be from -1.00 through 1.00. If a range of -2 is specified, a domain error occurs.

Range errors occur when the value specified is outside of the allowable values. For example, POWER(10.0, 400) is out of the range of the maximum of \sim 2e+308 of the **float** data type, while POWER(-10.0, 401) is out of the range of the minimum of \sim -2e+308 of the **float** data type.

This table shows mathematical functions that produce either a domain or range error.

Mathematical function	Result
SQRT(-1)	Domain error.
POWER(10.0, 400)	Arithmetic Overflow error.
	Value of 0.0 (floating point underflow).

Error traps are provided to handle domain or range errors of these functions. You can use:

- SET ARITHABORT ON, which terminates the query and quits the user-defined transaction. The SET ARITHABORT setting overrides the setting for SET ANSI_WARNINGS.
- SET ANSI_WARNINGS ON, which stops the command.
- SET ARITHIGNORE ON, which causes no warning message to be displayed. Both the SET ARITHABORT and SET ANSI_WARNINGS settings override the SET ARITHIGNORE setting.

If neither of these options is set, SQL Server returns NULL and returns a warning message after the query is executed. For more information, see <u>SET ARITHABORT</u>, <u>SET ANSI_WARNINGS</u>, and <u>SET ARITHIGNORE</u>.

Internal conversion to **float** can cause loss of precision if either the **money** or **numeric** data types are used.

Using Trigonometric Functions

Microsoft® SQL Server[™] 2000 provides trigonometric functions that return radians.

Functions returning radians	Use radians as input value
ACOS	TAN
COS	SIN
ATAN	ASIN
ATN2	
СОТ	

ACOS and COS

Both ACOS and COS are trigonometric functions. The ACOS function returns the angle, in radians, whose cosine is the given **float** expression. The COS function returns the cosine of the specified angle, in radians, given the **float** expression. For example, this SELECT statement calculates the ACOS of -.997 and the COS for the value 1.134:

```
SELECT ACOS(-.997), COS(1.134)
```

Therefore, the angle that measures 3.06411360866591 radians has a cosine value of -.997, and the angle measuring 1.134 radians has a cosine value of 1.134.

The valid ranges for ACOS are from -1 through 1.

ASIN and SIN

Both ASIN and SIN are trigonometric functions that use a **float** expression. The ASIN function calculates the angle, measured in radians, whose sine is the given **float** expression. The SIN function calculates the trigonometric sine value of the angle, measured in radians, as a **float** expression.

This example calculates the ASIN of -.7582 and the SIN of 5. The angle that measures -0.860548023283932 in radians has a sine value of -.7582 and the

angle that measures 5 radians has a sine value of -0.958924274663138.

SELECT ASIN(-.7582), SIN(5)

The valid ranges for ASIN are from -1 through 1.

ATAN, ATN2, TAN, and COT

The ATAN, ATN2, TAN, and COT functions are mathematical functions. The ATAN function returns the measurement of the angle, in radians, whose tangent is the given **float** expression. An angle having a tangent value of -27.29 measures

-1.53416925536896 in radians.

The ATN2 function returns the angle, in radians, for an angle whose tangent is between the two given **float** expressions. An angle with a tangent from 3.273 through 15 measures 0.214832755968629 in radians.

The TAN function returns the trigonometric tangent of the given **float** expression. An angle that measures 27.92 radians has a tangent of -0.36994766163616.

The COT function returns the trigonometric cotangent of the specified angle, in radians, specified in the given **float** expression. An angle of 97.1928 radians has a cotangent value of -5.02149424849997.

DEGREES

The DEGREES function returns a **numeric** expression, the measurement of an angle, in degrees, from the angle's measurement in radians. An angle measuring -14.578 radians measures -835.257873741714090000 degrees.

SELECT DEGREES(-14.578)

RADIANS

The RADIANS function calculates the angle in radians given the angle's measurement in degrees. To calculate the measurement in radians of an angle that measures 10.75 degrees, use:

SELECT RADIANS(10.75)

See Also

Comparing CEILING and FLOOR

The CEILING function returns the smallest integer greater than or equal to the given numeric expression. The FLOOR function returns the largest integer less than or equal to the given numeric expression. For example, given a numeric expression of 12.9273, CEILING returns 13, and FLOOR returns 12. The return value of both FLOOR and CEILING has the same data type as the input numeric expression.

See Also

Comparing LOG and LOG10

The LOG function returns the natural logarithm for the given **float** expression. Natural logarithms are calculated by using the base-2 system. However, the LOG10 function returns the base-10 logarithm. Use both LOG and LOG10 for trigonometric applications. For example, this SELECT statement calculates both the LOG and LOG10 of the value 1.75:

SELECT LOG(1.75), LOG10(1.75)

See Also

Using the POWER and EXP Exponential Functions

The POWER function returns the value of the given numeric expression to the specified power. POWER(2,3) returns 2 to the third power, or the value 8. Negative powers can be specified, so POWER(2.000, -3) returns 0.125. Notice that the result of POWER(2, -3) is 0. This is because the result is the same data type as the given numeric expression. Therefore, if the result has three decimal places, the number to raise to a given power must have three decimals, too.

The EXP function returns the exponential value in scientific notation of the given **float** expression. So, with a value of 198.1938327, the EXP function returns a value of 1.18710159597953e+086.

SELECT EXP(198.1938327)

See Also

Using RAND

The RAND function calculates a random floating-point number from 0 through 1, and can optionally take a **tinyint**, **int**, or **smallint** value for the starting point of the random number to calculate.

This example calculates two random numbers. The first RAND() function lets Microsoft® SQL Server[™] pick the seed value, and the second RAND() function uses the value of 3 for the starting position.

SELECT RAND(), RAND(3)

The RAND function is a pseudorandom number generator that operates in a manner similar to the C run-time library **rand** function. If no seed is provided, the system generates its own variable seed numbers. If you call RAND with a seed value, you must use variable seed values to generate random numbers. If you call RAND multiple times with the same seed value, it returns the same generated value. This script returns the same value for the calls to RAND because they all use the same seed value:

SELECT RAND(159784) SELECT RAND(159784) SELECT RAND(159784)

A common way to generate random numbers from RAND is to include something relatively variable as the seed value, such as adding several parts of a GETDATE:

SELECT RAND((DATEPART(mm, GETDATE()) * 100000) + (DATEPART(ss, GETDATE()) * 1000) + DATEPART(ms, GETDATE()))

When you use an algorithm based on GETDATE to generate seed values, RAND can still generate duplicate values if the calls to RAND are made within the interval of the smallest datepart used in the algorithm. This is especially likely if the calls to RAND are included in a single batch. Multiple calls to RAND in a

single batch can be executed within the same millisecond, which is the smallest increment of DATEPART. In this case, incorporate a value based on something other than time to generate the seed values.

See Also

Date Functions

Date functions are used to display information about dates and times. These functions manipulate **datetime** and **smalldatetime** values, performing arithmetic operations on them. Date functions can be used anywhere an expression can be used.

SQL Server recognizes a wide variety of **datetime** data entry formats. You can use the SET DATEFORMAT statement to set the order of the dateparts (month/day/year) for entering **datetime** or **smalldatetime** data. Enclose **datetime** or **smalldatetime** values in single quotation marks.

See Also

Date and Time Functions

Using GETDATE

The GETDATE function produces the current date and time in Microsoft® SQL Server[™] internal format for **datetime** values. GETDATE takes the null parameter ().

This example finds the current system date and time:

SELECT GETDATE()

Here is the result set:

July 29 1995 2:50 PM

(1 row(s) affected)

You can use GETDATE in designing a report to have the current date and time printed every time the report is produced. GETDATE is also useful for functions such as logging the time a transaction occurred on an account.

You can use GETDATE anywhere to return the current system date. For example, you can use GETDATE as a default value for data entry, with a local variable, or comparing an order date to today's date.

See Also

Date and Time Functions

Comparing DATEPART and DATENAME

The DATEPART and DATENAME functions produce the specified part of a **datetime** value (the year, quarter, day, hour, and so on) as either an integer or an ASCII string. Because **smalldatetime** is accurate only to the minute, when a **smalldatetime** value is used with either of these functions, the seconds and milliseconds returned are always zero.

These examples assume the date May 29:

SELECT DATEPART(month, GETDATE())

Here is the result set:

5

(1 row(s) affected)

SELECT DATENAME(month, GETDATE())

Here is the result set:

May

(1 row(s) affected)

See Also

Date and Time Functions

Comparing DATEADD and DATEDIFF

The DATEADD function adds an interval to a date you specify. For example, if the publication dates of all books in the **titles** table slipped three days, you could get the new publication dates with this statement:

USE pubs SELECT DATEADD(day, 3, pubdate) FROM titles

If the date parameter is a **smalldatetime** data type, the result is also a **smalldatetime**. You can use DATEADD to add seconds or milliseconds to a **smalldatetime** value, but doing this is meaningful only if the result date returned by DATEADD changes by at least 1 minute.

The DATEDIFF function calculates the amount of time in dateparts between the second and first of two dates you specify. In other words, it finds an interval between two dates. The result is a signed integer value equal to *date2 - date1* in date parts.

This query uses the date November 30, 1995, and finds the number of days that elapsed between **pubdate** and that date:

```
USE pubs
SELECT DATEDIFF(day, pubdate, 'Nov 30 1995')
FROM titles
```

For the rows in titles having a **pubdate** of October 21, 1995, the result produced by the last query is 40. (There are 40 days between October 21 and November 30.) To calculate an interval in months, use this query:

```
USE pubs
SELECT interval = DATEDIFF(month, pubdate, 'Nov 30 1995')
FROM titles
```

The query produces a value of 1 for the rows with a **pubdate** in October and a

value of 5 for the rows with a **pubdate** in June.

When the first date in the DATEDIFF function is later than the second date specified, the resulting value is negative. Because two of the rows in **titles** have values for **pubdate** assigned using the GETDATE function as a default, these values are set to the date that your **pubs** database was created and return negative values in the two preceding queries.

If one or both of the date arguments is a **smalldatetime** value, they are converted to **datetime** values internally for the calculation. Seconds and milliseconds in **smalldatetime** values are automatically set to 0 for the purpose of calculation.

See Also

CAST and CONVERTDATENAMEDate and Time FunctionsDATEPARTDATEADDGETDATEDATEDIFFSET DATEFORMAT

Functions That Return User Names and User IDs

Several system functions return user names and user IDs. Understanding the parameters and output of these functions requires an understanding of the types of names and IDs used in Microsoft® SQL ServerTM 2000.

Each user logging on to SQL Server has two levels of names in SQL Server, and each name is associated with a unique ID:

• Login names.

Each user authorized to log on to SQL Server has one login name that gives them access to an instance of SQL Server. There are two types of login names:

• Microsoft Windows® account names.

Members of the **sysadmin** or **securityadmin** fixed server roles can authorize the Windows accounts of individual users or Windows groups to log on to an instance of SQL Server using **sp_grantlogin**. The user identified by the Windows account, or any person in the Windows group, can then connect to an instance of SQL Server using Windows Authentication. Each Windows account or group name is stored in **master.dbo.syslogins.loginname**. The Windows *security_identifier* for the Windows account or group is stored in **syslogins.sid**.

• SQL Server login names.

These are used when logging-on using SQL Server Authentication. SQL Server login names are defined by members of the **sysadmin** or **securityadmin** fixed server roles using **sp_addlogin**. Each SQL Server login name is stored in **master.dbo.syslogins.loginname**. SQL Server generates a GUID used as a *security_identifier* and stores it in **syslogins.sid**.

SQL Server 2000 uses master.dbo.syslogins.sid as the

security_identifier for the login name.

• Database user name.

Each Windows account or SQL Server login must be associated with a user name in each database the user is authorized to access, or the database must have guest access enabled. Database user names are defined by members of the **db_owner** or **db_accessadmin** fixed database role, and are stored in the **sysusers** table found in each database. Each database user name is associated with a database user ID stored in **sysusers.uid**.

The *security_identifier* for each user is stored in **sysusers.sid**; therefore, users can be mapped back to their associated logins. It is less confusing if the same name for the database user is used for the SQL Server login or Windows account; however, there is no requirement to do this.

For more information about login and database user accounts, see <u>Logins</u>, <u>Users</u>, <u>Roles</u>, <u>and Groups</u>.

Getting Login Accounts or IDs

When connected to SQL Server 2000, use:

- SUSER_SNAME to get the SQL Server login name or Windows account associated with a *security_identifier*.
- SUSER_SID to get the *security_identifier* associated with a SQL Server login name or Windows NT account.
- SUSER_SID() (SUSER_SID specified without a *login_account* parameter) to get the *security_identifier* of the current connection regardless of whether SQL Server Authentication or Windows Authentication is used.
- The SQL-92 function SYSTEM_USER to get the Windows account for a Windows Authentication connection or the SQL Server login name for an SQL Server Authentication connection. In Transact-SQL,

SYSTEM_USER is implemented as a synonym for SUSER_SNAME() (SUSER_SNAME specified without a *security_identifier*).

In SQL Server 2000, the functions that return login names or accounts operate in this manner:

• SUSER_SNAME(security_identifier)

SUSER_SNAME takes either:

- The *security_identifier* for a Windows account or group, in which case it returns the name of the Windows account or group.
- The pseudo *security_identifier* generated for a SQL Server login, in which case it returns the SQL Server login name.

If a *security_identifier* is not specified for a connection made using Windows Authentication, SUSER_SNAME returns the name of the Windows account associated with the connection. If the connection was made using SQL Server Authentication, SUSER_SNAME returns the SQL Server login associated with the connection.

• SYSTEM_USER

This SQL-92 function is implemented as a synonym for SUSER_SNAME() (SUSER_SNAME specified without a *security_identifier* parameter) in Transact-SQL.

Getting Database User Names or User IDs

When connected to SQL Server 2000, use:

- USER_ID to get the database user ID associated with a database user name.
- USER_ID() to get the database user ID associated with the current connection.

- USER_NAME to get the database user name associated with a database user ID.
- Either the SQL-92 CURRENT_USER or SESSION_USER functions to get the database user name associated with the current connection. In Transact-SQL these functions are implemented as synonyms for the USER_NAME() (USER_NAME specified without a *database_user_ID* parameter). The Transact-SQL function USER is also implemented as a synonym for USER_NAME().

SQL-92 allows for SQL statements to be coded in SQL modules that can have authorization identifiers separate from the authorization identifier of the user that has connected to a SQL database. SQL-92 specifies that SESSION_USER always return the authorization identifier of the user that made the connection. CURRENT USER returns the authorization identifier of the SQL module for any statements executed from a SQL module, or of the user that made the connection if the SQL statements were not executed from a SQL module. If the SQL module does not have a separate authorization identifier, SQL-92 specifies that CURRENT_USER return the same value as SESSION_USER. Microsoft SQL Server does not have separate authorization identifiers for SQL modules; therefore, CURRENT USER and SESSION USER are always the same. The USER function is defined by SQL-92 as a backward compatibility function for applications written to earlier versions of the standard. It is specified to return the same value as CURRENT USER.

In SQL Server, the functions that return login names or accounts operate in this manner:

• USER_ID('database_user_name')

USER_ID returns the database user ID associated with the specified database user name. If *database_user_name* is not specified, USER_ID returns the database user ID associated with the current connection.

• USER_NAME(*database_user_ID*)

USER_NAME returns the database user name associated with the

specified database user ID. If *database_user_ID* is not specified, USER_NAME returns the database user name associated with the current connection.

• CURRENT_USER, SESSION_USER, USER

These functions are synonyms for USER_NAME() (USER NAME specified without a *database_user_ID* parameter).

See Also

CURRENT_USER SYSTEM_USER USER_ID SUSER_SID USER_NAME SUSER_SNAME

Conversion Functions

Use the conversion functions, CAST and CONVERT, to convert expressions of one data type to another data type when these conversions are not performed automatically by Microsoft® SQL ServerTM 2000. These conversion functions are also used to obtain a variety of special data formats. Either of the conversion functions can be used in the select list, in the WHERE clause, and anywhere an expression is allowed.

Use CAST rather than CONVERT if you want Transact-SQL program code to comply with SQL-92. Use CONVERT rather than CAST to take advantage of the style functionality in CONVERT.

When using either CAST or CONVERT, two pieces of information are required:

- The expression to convert (for example, the sales report requires the sales data to be converted from monetary data to character data).
- The data type to convert the given expression to, for example, **varchar** or any other SQL Server-supplied data type.

Unless you store the converted value, a conversion is valid only for the length of the CAST or CONVERT function.

This example uses CAST in the first SELECT statement and CONVERT in the second SELECT statement to convert the **title** column to a **char(50)** column, to make the results more readable:

```
USE pubs
SELECT CAST(title AS char(50)), ytd_sales
FROM titles
WHERE type = 'trad_cook'
```

Or

```
USE pubs
SELECT CONVERT(char(50), title), ytd_sales
```

FROM titles WHERE type = 'trad_cook'

Here is the result set: (for either query)

ytd_sales

Onions, Leeks, and Garlic: Cooking Secrets of the375Fifty Years in Buckingham Palace Kitchens15096Sushi, Anyone?4095

(3 row(s) affected)

In this example, the **ytd_sales** column, an **int** column, is converted to a **char(20)** column so that it can be used with the LIKE predicate:

```
USE pubs
SELECT title, ytd_sales
FROM titles
WHERE CAST(ytd_sales AS char(20)) LIKE '15%'
AND type = 'trad_cook'
```

Here is the result set:

Title

ytd_sales

Fifty Years in Buckingham Palace Kitchens 15096

(1 row(s) affected)

SQL Server automatically handles certain data type conversions. For example, if you compare a **char** and a **datetime** expression, or a **smallint** and an **int** expression, or **char** expressions of different lengths, SQL Server converts them automatically. This is called an implicit conversion. You do not have to use the CAST function for these conversions. However, it is acceptable to use the CAST when:

- Two expressions are exactly the same data type.
- Two expressions are implicitly convertible.
- It is necessary to explicitly convert the data types.

If you attempt a conversion that is not possible (for example, converting a **char** expression that includes letters to **int**), SQL Server displays an error message.

If you do not specify a length when converting for the data type, SQL Server automatically supplies a length of 30.

When converting to **datetime** or **smalldatetime**, SQL Server rejects all values it cannot recognize as dates (including dates earlier than January 1, 1753). You can convert **datetime** values to **smalldatetime** when the date is in the proper range (from January 1, 1900 through June 6, 2079). The time value is rounded to the nearest minute.

Converting to **bit** changes any nonzero value to 1.

When converting to **money** or **smallmoney**, integers are assumed to be monetary units. For example, the integer value of 4 is converted to the money equivalent of 4 dollars (for **us_english**, the default language). Numbers to the right of the decimal in floating-point values are rounded to four decimal places for **money** values. Expressions of data types **char** or **varchar** that are being converted to an integer data type must consist only of digits and an optional plus or minus sign (+ or -). Leading blanks are ignored. Expressions of data types **char** or **varchar** converted to **money** can also include an optional decimal point and leading dollar sign (\$).

Expressions of data types **char** or **varchar** that are being converted to **float** or **real** can also include optional exponential notation (e or E, followed by an optional + or - sign, and then a number).

When character expressions are converted to a data type of a different size, values too long for the new data type are truncated, and SQL Server displays an asterisk (*) in both the **osql** utility and SQL Query Analyzer. When numeric expressions are too long for the new data type to display, values are truncated. This is an example of character truncation:

```
USE pubs
SELECT SUBSTRING(title, 1, 25) AS Title, CONVERT(char(2), ytd_
FROM titles
WHERE type = 'trad_cook'
```

Here is the result set:

Title

----- ---

Onions, Leeks, and Garlic * Fifty Years in Buckingham * Sushi, Anyone? *

(3 row(s) affected)

When converting between data types in which the target data type has fewer decimal places than the source data type, the value is truncated. For example, the result of CAST(10.3496 AS money) is \$10.35.

You can explicitly convert **text** data to **char** or **varchar**, and **image** data to **binary** or **varbinary**. Because these data types are limited to 8,000 characters, you are limited to the maximum length of the **character** and **binary** data types, 8,000 characters. You can explicitly convert **ntext** data to **nchar** or **nvarchar**, but the maximum length is 4,000 characters. If you do not specify the length, the converted value has a default length of 30 characters. Implicit conversion is not supported.

The style Parameter

The *style* parameter of CONVERT provides a variety of date display formats when converting **datetime** data to **char** or **varchar**. The number you supply as the *style* parameter determines how the **datetime** data is displayed. The year can be displayed in either two or four digits. By default, SQL Server supplies a two-digit year. To display a four-digit year including the century (yyyy), even if the year data was stored by using a two-digit year format, add 100 to a *style* value to get a four-place year.

This example shows CONVERT with the *style* parameter: SELECT CONVERT(char(12), GETDATE(), 3)

This statement converts the current date to style 3, dd/mm/yy.

See Also

CAST and CONVERT

Functions

Invoking User-Defined Functions

When you reference or invoke a user-defined function, you specify the function name followed by parentheses. Within the parentheses, you can specify expressions called arguments that provide the data to be passed in to the parameters. You cannot specify parameter names in the arguments when invoking a function. When you invoke a function, you must supply argument values for all of the parameters and you must specify the argument values in the same sequence in which the parameters are defined in the CREATE FUNCTION statement. For example, if a function named fn_MyIntFunc that returns an integer is defined with an integer parameter and an **nchar(20)** parameter, it can be invoked using:

SELECT *
FROM SomeTable
WHERE PriKey = dbo.fn_MyIntFunc(1, N'Anderson')

This is an example of invoking a function named fn_MyTableFunc defined to return a table:

```
SELECT *
FROM dbo.fn_MyTableFunc( 123.09, N'O"Neill' )
```

See Also

User-Defined Functions

CREATE FUNCTION

Invoking User-Defined Functions That Return a Scalar Value

You can invoke a user-defined function that returns a scalar value anywhere a scalar expression of the same data type is allowed in Transact-SQL statements:

Queries

User-defined functions that return scalar values are allowed in these locations:

- As an *expression* in the *select_list* of a SELECT statement: SELECT *, dbo.fn_CalculateDaysLate(RequiredDate) AS Day FROM Northwind.dbo.Employees
- As an *expression* or *string_expression* in a WHERE or HAVING clause *predicate*: SELECT * FROM Northwind.dbo.[Order Details] WHERE UnitPrice < dbo.fn_MeanUnitPrice()
- As a *group_by_expression* in a GROUP BY clause.
- As an *order_by_expression* in an ORDER BY clause.
- As an *expression* in the SET clause in an UPDATE statement: UPDATE Orders SET ShipVia = dbo.fn_FindLeastCostShipper(ShipCity) WHERE OrderID = 10274
- As an *expression* in the VALUES clause of an INSERT statement: INSERT INTO Shippers VALUES (4, dbo.fn_GetShipperName(), n'(503)555-9931'

User-defined functions referenced in these locations are logically executed once per row.

CHECK constraints

User-defined functions that return scalar values can be invoked in CHECK constraints if the argument values passed to the function reference columns only in the table or constants. Each time the query processor checks the constraint, query processor calls the function with the argument values associated with the current row being checked. The owner of a table must also be the owner of the user-defined function invoked by a CHECK constraint on the table.

DEFAULT definitions

User-defined functions can be invoked as the *constant_expression* of DEFAULT definitions if the argument values passed to the function contains only constants. The owner of the table must also be the owner of the user-defined function invoked by a DEFAULT definition on the table.

Computed columns

Functions can be invoked by computed columns if the argument values passed to the function reference only columns in the table or constants. The owner of the table must also be the owner of the user-defined function invoked by a computed column in the table.

Assignment operators

Assignment operators (*left_operand* = *right_operand*) can invoke user-defined functions that return a scalar value in the expression specified as the right operand.

Control-of-Flow statements

User-defined functions that return scalar values can be invoked by control-of-flow statements in their Boolean expressions.

CASE expressions

User-defined functions that return a scalar value can be invoked in any of the CASE expressions.

PRINT statements

User-defined functions that return a character string can be invoked as the *string_expr* expression of PRINT statements.

Functions and stored procedures

- Function arguments can also be a reference to a user-defined function that returns a scalar value.
- RETURN *integer_expression* statements in stored procedures can invoke user-defined functions that return an integer as the *integer_expression*.
- RETURN *return_type_spec* statements in user-defined functions can invoke user-defined functions that return a scalar data type such as the *return_type_spec*, provided the value returned by the invoked user-defined function can be implicitly converted to the return data type of the invoking function.

Executing User-Defined Functions That Return a Scalar Value

You can execute user-defined functions that return scalar values in the same manner as stored procedures. When executing a user-defined function that returns a scalar value, the parameters are specified as they are for stored procedures:

- The argument values are not enclosed in parentheses.
- Parameter names can be specified.
- If parameter names are specified, the argument values do not have to be in the same sequence as the parameters.

This is a definition of a user-defined function that returns a decimal:

CREATE FUNCTION fn_CubicVolume

-- Input dimensions in centimeters.

```
(@CubeLength decimal(4,1), @CubeWidth decimal(4,1),
@CubeHeight decimal(4,1) )
RETURNS decimal(12,3) -- Cubic Centimeters.
WITH SCHEMABINDING
AS
BEGIN
RETURN ( @CubeLength * @CubeWidth * @CubeHeight )
END
```

This is an example of executing the fn_CubicVolume function. Using the Transact-SQL EXECUTE statement, the arguments are identified in a different order than the parameters in the function definition:

DECLARE @MyDecimalVar decimal(12,3) EXEC @MyDecimalVar = dbo.fn_CubicVolume @CubeLength = 12.3 @CubeHeight = 4.5, @CubeWidth = 4.5

This is an example of executing the fn_CubicVolume function without specifying the parameter names:

```
DECLARE @MyDecimalVar decimal(12,3)
EXEC @MyDecimalVar = dbo.fn_CubicVolume 12.3, 4.5, 4.5
```

You can also use the ODBC CALL syntax to execute the fn_CubicVolume function from OLE DB or ODBC applications:

-- First use SQLBindParam to bind the return value parameter marker
-- to a program variable of the appropriate type
SQLExecDirect(hstmt,

"{ CALL ? = fn_CubicVolume(12.3, 4.5, 4.5) }", SQL_NTS);

Invoking User-Defined Functions That Return a Table Data Type

You can invoke a user-defined function that returns a **table** where table expressions are allowed in the FROM clause of SELECT, INSERT, UPDATE, or DELETE statements. An invocation of a user-defined function that returns a table can be followed by an optional table alias. This example illustrates calling a function fn_Products and assigning an alias:

SELECT OD.OrderID, OD.ProductID, fnPr.Price FROM OrderDetails as OD, fn_Products('Discontinued') AS fnPr WHERE OD.ProductID = fnPr.ProductID ORDER BY OD.OrderID, OD.ProductID

When a user-defined function that returns a table is invoked in the FROM clause of a subquery, the function arguments cannot reference any columns from the outer query.

Static, read-only cursors are the only type of cursor that can be opened on a SELECT statement whose FROM clause refers to a user-defined function that returns a table.

A SELECT statement that references a user-defined function that returns a **table** invokes the function once.

See Also

User-Defined Functions That Return a Table

Inline User-Defined Functions

<u>table</u>

Invoking Built-in User-Defined Functions

Microsoft® SQL Server[™] 2000 implements several built-in functions as userdefined functions that return a **table**. The invocation of these built-in userdefined functions follows special rules:

- For built-in user-defined functions that return a table, the function name must be specified with a leading double colon (::) to distinguish it from user-defined functions that are not built-in. It also must be specified as a one-part name with no database or owner qualifications. For example: SELECT *
 FROM ::fn_helpcollations()
- For built-in user-defined functions that return a scalar value, the function name must be specified as a one-part name (do not specify database or owner). Do not specify a leading double colon (::).

Expressions

An expression is a combination of identifiers, values, and operators that Microsoft® SQL Server[™] can evaluate to get a result. The data can be used in several different places when accessing or changing data. Expressions can be used, for example, as part of the data to retrieve (in a query) or as a search condition to look for data meeting a set of criteria.

An expression can be a:

- Constant
- Function
- Column name
- Variable
- Subquery
- CASE, NULLIF or COALESCE

An expression can also be built from combinations of these entities joined by operators.

In the following SELECT statement, for each row of the result set, SQL Server can resolve **LastName** to a single value; therefore, it is an expression.

SELECT LastName FROM Northwind..Employees

An expression can also be a calculation, such as (**price** * 1.5) or (**price** + **sales_tax**).

In an expression, enclose character date values in single quotation marks. In the

following SELECT the character literal B% used as the pattern for the LIKE clause must be in single quotation marks:

SELECT LastName, FirstName FROM Northwind..Employees WHERE LastName LIKE 'B%'

In the following SELECT, the date value is enclosed in quotation marks:

SELECT * FROM Northwind..Orders WHERE OrderDate = 'Sep 13 1996'

In this example, more than one expression is used in the query. For example, **col1**, SUBSTRING, **col3**, **price**, and 1.5 are all expressions.

SELECT col1, SUBSTRING('This is a long string', 1, 5), col3, price * FROM mytable

See Also

<u>CASE</u>

INSERT

<u>COALESCE</u>

<u>UPDATE</u>

Functions

<u>DELETE</u>

<u>SELECT</u>

Expressions

Using Operators in Expressions

Operators allow you to perform arithmetic, comparison, concatenation, or assignment of values. For example, you can test data to verify that the **country** column for your customer data is populated (or not NULL).

In queries, anyone who can see the data in the table that needs to be used with some type of operator can perform operations. You need the appropriate permissions before you can successfully change the data.

Operators are used in Microsoft[®] SQL Server[™] to:

- Change data, either permanently or temporarily.
- Search for rows or columns that meet a specified condition.
- Implement a decision between columns of data or between expressions.
- Test for specific conditions before beginning or committing a transaction, or before executing specific lines of code.

SQL Server has seven categories of operators.

To perform this type of operation	Use this operator category
Compare a value against another value or an	Comparison operators
expression.	
Test for the truth of a condition, such as AND,	Logical
OR, NOT, LIKE, ANY, ALL, IN.	
Addition, subtraction, multiplication, division,	Arithmetic operators
modulo.	
Performs an operation on one operand, such as	Unary
positive or negative or one's complement.	
Temporarily turn regular numeric values (like	Bitwise Operators
150) into an integer and perform bitwise (0	
and 1) arithmetic.	

Either permanently or temporarily combine	String Concatenation
two strings (either character or binary data)	Operator
into one string.	
Assigns a value to a variable, or associates a	Assignment
result set column with an alias.	

An expression can be built from several smaller expressions combined by operators. In these complex expressions, the operators are evaluated in order based on the SQL Server definition of operator precedence. Operators with higher precedence are performed before operators with lower precedence. For more information, see <u>Operators</u>.

When simple expressions are combined to form a complex expression, the data type of the result is determined by combining the rules for the operators with the rules for data type precedence. If the result is a character or Unicode value, the collation of the result is determined by combining the rules for the operators with the rules for collation precedence. There are also rules that determine the precision, scale, and length of the result based on the precision, scale, and length of the simple expressions.

See Also

Collation Precedence

Data Type Precedence

Precision, Scale, and Length

Arithmetic Operators

Arithmetic operators can be used for any arithmetic computations, such as:

- Addition.
- Subtraction.
- Multiplication.
- Division.
- Modulo (the remainder from a division operation).

Here is some information about arithmetic operators:

- When there is more than one arithmetic operator in an expression, multiplication, division, and modulo are calculated first, followed by subtraction and addition.
- When all arithmetic operators in an expression have the same level of precedence, the order of execution is left to right.
- Expressions within parentheses take precedence over all other operations.

The following SELECT statement subtracts the part of the year-to-date sales that the author receives (sales * author's royalty percentage / 100) from the total sales. The result is the amount of money the publisher receives. The product of **ytd_sales** and **royalty** is calculated first because the operator is multiplication. Next, the total is divided by 100. The result is subtracted from **ytd_sales**.

USE pubs

SELECT title_id, ytd_sales - ytd_sales * royalty / 100 FROM titles

For clarity, you can use parentheses:

```
USE pubs
SELECT title_id, ytd_sales - ((ytd_sales * royalty) / 100)
FROM titles
```

You can also use parentheses to change the order of execution. Calculations inside parentheses are evaluated first. If parentheses are nested, the most deeply nested calculation has precedence. For example, the result and meaning of the preceding query can be changed if you use parentheses to force the evaluation of subtraction before multiplication:

```
USE pubs
SELECT title_id, (ytd_sales - ytd_sales) * royalty / 100
FROM titles
```

See Also

- (Subtract)
+ (Add)
* (Multiply)
/ (Divide)

Bitwise Operators

Bitwise operators are used on **int**, **smallint**, or **tinyint** data. The ~ (Bitwise NOT) operator can also use **bit** data. All bitwise operators perform an operation on the one or more specified integer values as translated to binary expressions within Transact-SQL statements. For example, the ~ (Bitwise NOT) operator changes binary 1s to 0s and 0s to 1s. To check bitwise operations, you can convert or calculate decimal values.

For example, you want to add 150 and 75 together, but you're interested in not only the decimal value of 225, but want to use binary arithmetic (addition of 0s and 1s). Use the bitwise AND operator (&) for this purpose.

If you are storing integer data (normal decimal values like the 150 and 75 mentioned earlier) and want to perform internal translation to do binary math, use bitwise operators. Bitwise operators are also valuable to get a NOT value which is not necessarily the exact opposite.

See Also

& (Bitwise AND) ~ (Bitwise NOT) | (Bitwise OR) ^ (Bitwise Exclusive OR) Data Types

Comparison Operators

Comparison operators are used with character, numeric, or date data and can be used in the WHERE or HAVING clause of a query. Comparison operators evaluate to a Boolean data type; they return TRUE or FALSE based on the outcome of the tested condition.

For example, to calculate a bonus for those employees who have been hired on or before March 15, 1998, a computation of whether the **hire_date** for an employee is less than or equal to March 15, 1998 provides the list of employees who should receive bonuses.

Valid comparison operators are:

- > (greater than).
- < (less than).
- = (equals).
- <= (less than or equal to).
- >= (greater than or equal to).
- != (not equal to).
- <> (not equal to).
- !< (not less than).
- !> (not greater than).

Comparison operators can also be used in program logic to check for a condition. For example, if the country column is UK rather than Spain, different shipping rates may apply. In this case, a combination of a comparison operator, an expression (the column name), a literal ('UK') and a control-of-flow programming keyword (IF) are used together to achieve this purpose.

Anyone with access to the actual data (for queries) can use comparison operators in additional queries. For those data-modification statements, it is recommended that you use comparison operators only if you know you have the appropriate permissions and that data will be changed by only a limited group of people (to maintain data integrity).

Queries also use string comparisons to compare the value in a local variable, cursor, or column with a constant. For example, all customer rows should be printed if the country is the UK. The table shows string comparison examples between Unicode and non-Unicode data; ST1 is **char** and ST2 is **nchar**.

Comparison	Description
ST1 = ST2	Equivalent to CONVERT(nchar , ST1) =
	ST2 or CAST(ST1 as nchar) = ST2.
ST1 = 'non-Unicode string'	Regular SQL-92 string comparison.
ST2 = 'non-Unicode string'	Equivalent to ST2 = CONVERT(nchar ,
	'non-Unicode string') or ST2 = CAST('non-
	Unicode string' AS nchar).
ST2 = N'Unicode string'	Unicode comparison.
CONVERT(nchar , ST1) =	Unicode comparison.
ST2	
or	
CAST(ST1 AS nchar) = ST2	
ST1 = CONVERT(char ,	Regular SQL-92 string comparison.
ST2)	
or	
ST1 = CAST(ST2 AS char)	
N" (Unicode empty string in	Empty string.
parentheses)	
" (non-Unicode empty string)	
	one blank character (depending on SQL-92
ST1 = CONVERT(char , ST2) or ST1 = CAST(ST2 AS char) N" (Unicode empty string in parentheses)	Empty string.

settings).

See Also

= (Equals)

<> (Not Equal To)

> (Greater Than)

!< (Not Less Than)</pre>

< (Less Than)

<u>!= (Not Equal To)</u>

>= (Greater Than or Equal To)

!> (Not Greater Than)

<= (Less Than or Equal To)

WHERE

String Concatenation Operator

The string concatenation operator is the plus sign (+). You can combine, or concatenate, two or more character strings into a single character string. You can also concatenate binary strings. This is an example of concatenation:

```
SELECT ('abc' + 'def')
```

Here is the result set:

abcdef

(1 row(s) affected)

This query displays names of authors with California addresses under the **Moniker** column in last name, first name order, with a comma and space after the last name.

```
USE Northwind
GO
SELECT LastName + ', ' + FirstName AS Moniker
FROM Employees
WHERE Region = 'WA'
```

Here is the result set:

Moniker

Davolio, Nancy Fuller, Andrew Leverling, Janet Peacock, Margaret Callahan, Laura (15 row(s) affected)

Other data types, such as **datetime** and **smalldatetime**, must be converted to character strings using the CAST conversion function before they can be concatenated with a string.

USE pubs SELECT 'The due date is ' + CAST(pubdate AS varchar(128)) FROM titles WHERE title_id = 'BU1032'

Here is the result set:

The due date is Jun 12 1991 12:00AM

(1 row(s) affected)

The empty string (") is evaluated as a single space:

```
SELECT 'abc' + " + 'def'
```

Here is the result set:

----abcdef

(1 row(s) affected)

Note Whether an empty string (") is interpreted as a single blank character or as an empty character is determined by the compatibility level setting of **sp_dbcmptlevel**. For this example, if **sp_dbcmptlevel** is 65, empty literals are treated as a single blank.

When the input strings both have the same collation, the output string has the same collation as the inputs. When the input strings have different collations, the rules of collation precedence determine the collation of the output string. You

can also assign a specific collation using the COLLATE clause.

See Also

COLLATE

Collation Precedence

Operators

sp_dbcmptlevel

+ (String Concatentation)

Null Values

A value of NULL indicates the value is unknown. A value of NULL is different from an empty or zero value. No two null values are equal. Comparisons between two null values, or between a NULL and any other value, return unknown because the value of each NULL is unknown.

Null values usually indicate data that is unknown, not applicable, or to be added at a later time. For example, a customer's middle initial may not be known at the time the customer places an order.

Here is some information about nulls:

- To test for null values in a query use IS NULL or IS NOT NULL in the WHERE clause.
- When query results are viewed in SQL Query Analyzer, null values are shown as **(null)** in the result set.
- Null values can be inserted into a column by explicitly stating NULL in an INSERT or UPDATE statement, or by leaving a column out of an INSERT statement, or when adding a new column to an existing table using the ALTER TABLE statement.
- Null values cannot be used for information required to distinguish one row in a table from another row in a table (for example, foreign or primary keys).

In program code, you can check for null values so that certain calculations are performed only on rows with valid (or not NULL) data. For example, a report can print the social security column only if there is data that is not NULL in the column. Eliminating null values when performing calculations can be important because certain calculations (such as an average) can be inaccurate if NULL columns are included.

If it is possible that null values may be stored in your data, it is a good idea to

create queries and data-modification statements that either eliminate NULLs or transform NULLs into some other value (if you do not want null values appearing in your data).

IMPORTANT To minimize maintenance and possible effects on existing queries or reports, it is recommended that you minimize the use of null values. Plan your queries and data-modification statements so that null values have minimal effect.

When null values are present in data, logical and comparison operators can potentially return a third result of UNKNOWN instead of just TRUE or FALSE. This need for three-valued logic is a source of many application errors. These tables outline the effect of introducing null comparisons.

This table shows the results of applying an AND operator to two Boolean operands.

AND	TRUE	UNKNOWN	FALSE
TRUE	TRUE	UNKNOWN	FALSE
UNKNOWN	UNKNOWN	UNKNOWN	FALSE
FALSE	FALSE	FALSE	FALSE

This table shows the results of applying an OR operator to two Boolean operands.

OR	TRUE	UNKNOWN	FALSE
TRUE	TRUE	TRUE	TRUE
UNKNOWN	TRUE	UNKNOWN	UNKNOWN
FALSE	TRUE	UNKNOWN	FALSE

This table shows how the NOT operator negates, or reverses, the result of a Boolean operator.

Evaluates to
FALSE
UNKNOWN

FALSE TRUE

The SQL-92 standard introduces the keywords IS NULL and IS NOT NULL to test for the presence of null values.

Boolean expression to which the IS NULL operator		Boolean expression to which the IS NOT NULL	
is applied	Evaluates to	operator is applied	Evaluates to
TRUE	FALSE	TRUE	TRUE
NULL	TRUE	NULL	FALSE
FALSE	FALSE	FALSE	TRUE

Transact-SQL also offers an extension for null processing. If the option ANSI_NULLS is set off, then comparisons between nulls, such as NULL = NULL, evaluate to TRUE. Comparisons between NULL and any data value evaluate to FALSE.

See Also

<u>AND</u>

<u>NOT</u>

CREATE TABLE

<u>OR</u>

Allowing Null Values

Modifying Column Properties

ISNULL

WHERE

Using Comments

Comments are nonexecuting text strings in program code (also known as remarks). Comments can be used to document code or temporarily disable parts of Transact-SQL statements and batches being diagnosed. Using comments to document code makes future program code maintenance easier. Comments are often used to record the program name, the author name, and the dates of major code changes. Comments can be used to describe complicated calculations or explain a programming method.

Microsoft[®] SQL Server[™] supports two types of commenting characters:

- -- (double hyphens). These comment characters can be used on the same line as code to be executed, or on a line by themselves. Everything from the double hyphens to the end of the line is part of the comment. For a multiple-line comment, the double hyphens must appear at the beginning of each comment line. For more information about using the comment characters, see <u>-- (Comment)</u>.
- /* ... */ (forward slash-asterisk character pairs). These comment characters can be used on the same line as code to be executed, on lines by themselves, or even within executable code. Everything from the open comment pair (/*) to the close comment pair (*/) is considered part of the comment. For a multiple-line comment, the open-comment character pair (/*) must begin the comment, and the close-comment character pair (*/) must end the comment. No other comment characters should appear on any lines of the comment. For more information about using the /* ... */ comment characters, see /* ... */ (Comment).

Multiple-line /* */ comments cannot span a batch. The complete comment must be contained within a batch. For example, in SQL Query Analyzer and the **osql** utility, the GO command signals the end of a batch. When the utilities read the characters GO in the first two bytes of a line, they send all the code since the last GO command to the server as one batch. If a GO occurs at the start of a line between the /* and */ delimiters, then an unmatched comment delimiter will be sent with each batch and they will trigger syntax errors. For example, the following script contains syntax errors:

```
USE Northwind
GO
SELECT * FROM Employees
/* The
GO in this comment causes it to be broken in half */
SELECT * FROM Products
GO
```

Here are some valid comments:

```
USE Northwind
GO
-- First line of a multiple-line comment.
-- Second line of a multiple-line comment.
SELECT * FROM Employees
GO
```

```
/* First line of a multiple-line comment.
   Second line of a multipl-line comment. */
SELECT * FROM Products
GO
```

```
-- Using a comment in a Transact-SQL statement
-- during diagnosis.
SELECT EmployeeID, /* FirstName, */ LastName
FROM Employees
```

```
-- Using a comment after the code on a line.
USE Northwind
GO
UPDATE Products
SET UnitPrice = UnitPrice * .9 -- Try to build market share.
```

GO

Here is some basic information regarding comments:

- All alphanumeric characters or symbols can be used within the comment. SQL Server ignores all characters within a comment, although SQL Query Analyzer, **osql**, and **isql** will search for GO as the first two characters in lines within a multiple line comment.
- There is no maximum length for a comment within a batch. A comment can consist of one or more lines.

Using Reserved Keywords

Microsoft® SQL Server[™] 2000 reserves certain keywords for its exclusive use. For example, using either the Transact-SQL DUMP or BACKUP keyword in an **osql** or SQL Query Analyzer session tells SQL Server to make a backup copy of all or part of a database, or a backup copy of the log.

It is not legal to include the reserved keywords in a Transact-SQL statement in any location except that defined by SQL Server. No objects in the database should be given a name that matches a reserved keyword. If such a name exists, the object must always be referred to using delimited identifiers. Although this method does allow for objects whose names are reserved words, it is recommended that you do not name any database objects with a name that is the same as a reserved word.

The system and database administrators roles or the database creator is usually responsible for checking for reserved keywords in Transact-SQL code and database names.

Use a naming convention that avoids using reserved keywords. Consonants or vowels can be removed if an object name must resemble a reserved keyword. For example, a procedure named **bckup** that performs BACKUP statements for all user-defined databases.

See Also

Reserved Keywords SET QUOTED_IDENTIFIER Delimited Identifiers CREATE TABLE ALTER TABLE CREATE RULE CREATE DATABASE ALTER DATABASE

Accessing and Changing Data Fundamentals

The primary purpose of a Microsoft® SQL Server[™] 2000 database is to store data and then make that data available to authorized applications and users. While database administrators create and maintain the database, users work with the contents of the database:

- Accessing, or retrieving, existing data
- Changing, or updating, existing data
- Adding, or inserting, new data
- Deleting existing data

Accessing and changing data in Microsoft SQL Server is accomplished by using an application or utility to send data retrieval and modification requests to SQL Server. For example, you can connect to SQL Server using SQL Server Enterprise Manager, SQL Query Analyzer, or the **osql** utility to begin working with the data in SQL Server.

Applications and utilities use two components to access SQL Server:

- Database application programming interfaces (APIs) send commands to SQL Server and retrieve the results of these commands. The APIs can be general-purpose database APIs such as ADO, OLE DB, ODBC, or DB-Library. They can also be APIs designed specifically to use special features in SQL Server, such as SQL-DMO, SQL-DTS, or the SQL Server replication components.
- Commands sent to SQL Server are Transact-SQL statements.

Transact-SQL statements are built using the SQL language defined in the Transact-SQL Reference. Most of these operations are implemented using one of four Transact-SQL statements:

- The SELECT statement is used to retrieve existing data.
- The UPDATE statement is used to change existing data.
- The INSERT statement is used to add new data rows.
- The DELETE statement is used to remove rows that are no longer needed.

These four statements form the core of the SQL language. Understanding how these four statements work is a large part of understanding how SQL works.

Graphical or forms-based query tools require no knowledge of SQL. They present the user with a graphical representation of the table. The user can graphically select the columns to be retrieved and easily specify how to qualify the rows to be retrieved.

Some applications, such as SQL Query Analyzer and the **osql** utility, are tools for executing Transact-SQL statements. These statements are entered interactively or read from a file. To use these tools, you must be able to build Transact-SQL statements.

Applications written to the general-purpose database APIs, such as ADO, OLE DB, ODBC, or DB-Library, also send Transact-SQL statements to SQL Server. These applications present the user with an interface reflecting the business function they support. When the user has indicated what business function should be performed, the application uses one of the database APIs to pass SQL statements to SQL Server. You must be able to build Transact-SQL statements to code these types of applications.

Other applications, such as SQL Server Enterprise Manager, use an object model that increases efficiency in using SQL Server. SQL Server Enterprise Manager uses an object model that eases the task of administering SQL Servers. APIs such as SQL-DMO, SQL-DTS, and the replication components also use similar object models. The objects themselves, however, communicate with SQL Server using Transact-SQL. Knowing the Transact-SQL language can help you understand these objects.

Building Transact-SQL Statements

Accessing and Changing Data Fundamentals contains information about the basic elements used to build Transact-SQL statements. It also provides information about the functions Transact-SQL can perform, as well as similar functionality offered by the database APIs.

A SELECT statement contains the common elements used in Transact-SQL statements. For example, to select the names, contact names, and telephone numbers of customers who live in the USA from the **Customers** table in the **Northwind** database, these elements are used:

- The name of the database containing the table (**Northwind**)
- The name of the table containing the data (**Customers**)
- A list of the columns for which data is to be returned (**CompanyName**, **ContactName**, **Phone**)
- Selection criteria (only for customers living in the USA)

This is the Transact-SQL syntax to retrieve this information:

SELECT CompanyName, ContactName, Phone FROM Northwind.dbo.Customers WHERE Country = 'USA'

Additional elements used in Transact-SQL statements include:

• Functions.

Functions are used in SQL Server queries, reports, and many Transact-SQL statements to return information, similar to functions in other programming languages. They take input parameters and return a value that can be used in expressions. For example, the DATEDIFF function takes two dates and a datepart (weeks, days, months, and so on) as arguments, and returns the number of datepart units there are between the two dates. • Identifiers.

Identifiers are the names given to objects such as tables, views, databases, and indexes. An identifier can be specified without delimiters (for example, TEST), with quoted delimiters ("TEST"), or in brackets ([TEST]).

• Comments.

Comments are nonexecuting remarks in program code.

• Expressions.

Expressions include constants or literal values (for example, 5 is a numeric literal), functions, column names, arithmetic, bitwise operations, scalar subqueries, CASE functions, COALESCE functions, or NULLIF functions.

• Reserved keywords.

Words that SQL Server reserves for its own functionality. It is recommended that you avoid using these reserved keywords as identifiers.

• Null values.

Null values are values that are unknown. You can use values of NULL to indicate that this information will come later. For example, if the contact at the Leka Trading company changes and the new contact is unknown, you could indicate the unknown contact name with a value of NULL.

• Data types.

Data types define the format in which data is stored. For example, you can use any of the character or Unicode data types (**char**, **varchar**, **nchar**, or **nvarchar**) to store character data such as customer names.

• Batches.

Batches are groups of statements transmitted and executed as a unit. Some Transact-SQL statements cannot be grouped in a batch. For example, to create five new tables in the **pubs** database, each CREATE TABLE statement must be in its own batch or unit. This is an example of a Transact-SQL batch:

USE Northwind SELECT * FROM Customers WHERE Region = 'WA' AND Country = 'USA' ORDER BY PostalCode ASC, CustomerID ASC UPDATE Employees SET City = 'Missoula' WHERE CustomerID = 'THECR' GO

• Control-of-flow language.

Control-of-flow language allows program code to take action, depending on whether a condition is met. For example, IF the amount of products ordered are equal to or less than the amount of products currently on hand, THEN we must order more products.

• Operators.

SQL Server includes operators, which allow certain actions to be performed on data. For example, using arithmetic operators, you can perform mathematical operations such as addition and subtraction on your data.

See Also

Data Types

CREATE TABLE

Expressions

ALTER TABLE

Using Identifiers

Operators

Functions

Choosing a Database

All objects in Microsoft® SQL Server[™] are stored in databases. All references to SQL Server objects have to be resolved to the specific database in which they reside.

- Explicit database references occur when a Transact-SQL statement names the database holding the data. This example explicitly names the Northwind database: SELECT * FROM Northwind..Employees
- Implicit database references occur when a Transact-SQL statement does not specify the database: SELECT * FROM Employees

To resolve implicit database references, SQL Server uses the concept of a current database. Every connection to SQL Server always has a database set as the current database. All object references that do not specify a database name are assumed to refer to the current database. For example, if a connection has **Northwind** set as its current database, any statement referring to an object named **Products** is resolved to the **Products** table in **Northwind**.

Every SQL Server login has a default database. At the time the login is defined by a member of the **sysadmin** fixed server role, the default database for the login can be specified. If a default database is not specified, **master** becomes the default database for the login. The default database for a login can be changed later using the **sp_defaultdb** stored procedure.

When you first connect to SQL Server, the default database for the login is usually made the current database. You can, however, specify a specific database as the current database at connect time. This request overrides the default database designated for the login. Here are the ways you can specify a database on a connect request:

• In the **osql** and **isql** utilities, specify the database name using the /**d**

switch.

- In ADO, specify the database name in the **Initial Catalog** property of an ADO connection object.
- In OLE DB, specify the database name in the DBPROP_INIT_CATALOG property.
- In ODBC, you can set a database name in an ODBC data source using the Database box of the Microsoft SQL Server DSN Configuration Wizard or the DATABASE = parameter on a call to SQLConfigDataSource. You can also specify DATABASE = on a call to SQLDriverConnect or SQLBrowseConnect.

You can switch the current database setting at any time while you are connected to SQL Server. This is called using, or choosing, a database. Here are ways you can switch the current database:

- You can execute the Transact-SQL USE *database_name* statement, regardless of the database API an application is using.
- In SQL Query Analyzer, you can select a database in the **Database** list box at the top of the Query Window.
- In ODBC, you can call **SQLSetConnectAttr** to set the SQL_ATTR_CURRENT_CATALOG connection attribute.
- In DB-Library, you can call the **dbuse** function.

Note In many Transact-SQL reference examples, "USE **pubs**" or "USE **Northwind**" is the first line in the example. This ensures the examples are executed against the explicitly specified database instead of another database.

See Also

osql Utility

<u>USE</u>

Using Multiple Statements

To perform processes that cannot be done using a single Transact-SQL statement, Microsoft® SQL Server[™] allows you to group Transact-SQL statements together in several ways:

• Using batches

A batch is a group of one or more Transact-SQL statements that are sent from an application to the server as one unit. SQL Server executes each batch as a single executable unit.

• Using stored procedures

A stored procedure is a group of Transact-SQL statements that have been predefined and precompiled on the server. The stored procedure can accept parameters, and can return result sets, return codes, and output parameters to the calling application.

• Using triggers

A trigger is a special type of stored procedure. It is not called directly by applications. It is instead executed whenever a user performs a specified modification (INSERT, UPDATE, or DELETE) to a table.

• Using scripts

A script is a series of Transact-SQL statements stored in a file. The file can be used as input to the **osql** utility or SQL Query Analyzer. The utilities then execute the Transact-SQL statements stored in the file.

The following SQL Server features allow you control the use of multiple Transact-SQL statements at a time:

• Control-of-flow statements

Allow you to include conditional logic. For example, if the country is Canada, perform one series of Transact-SQL statements. If the country is U.K., do some other series of Transact-SQL statements.

• Variables

Allow you to store data for use as input in a later Transact-SQL statement. For example, you can code a query that needs different data values specified in the WHERE clause each time the query is executed. You can write the query to use variables in the WHERE clause, and code logic to fill the variables with the proper data. The parameters of stored procedures are a special class of variables.

• Error handling

Lets you customize the way SQL Server responds to problems. You can specify appropriate actions to take when errors occur, or display customized error messages that are more informative to a user than a generic SQL Server error.

Batches

A batch is a group of one or more Transact-SQL statements sent at one time from an application to Microsoft® SQL ServerTM for execution. SQL Server compiles the statements of a batch into a single executable unit, called an execution plan. The statements in the execution plan are then executed one at a time.

A compile error, such as a syntax error, prevents the compilation of the execution plan, so none of the statements in the batch are executed.

A run-time error, such as an arithmetic overflow or a constraint violation, has one of two effects:

- Most run-time errors stop the current statement and the statements that follow it in the batch.
- A few run-time errors, such as constraint violations, stop only the current statement. All the remaining statements in the batch are executed.

The statements executed before the one that encountered the run-time error are not affected. The only exception is if the batch is in a transaction and the error causes the transaction to be rolled back. In this case, any uncommitted data modifications made before the run-time error are rolled back.

Assume there are 10 statements in a batch. If the fifth statement has a syntax error, none of the statements in the batch are executed. If the batch is compiled, and the second statement then fails while executing, the results of the first statement are not affected because it has already executed.

These rules apply to batches:

• CREATE DEFAULT, CREATE PROCEDURE, CREATE RULE, CREATE TRIGGER, and CREATE VIEW statements cannot be combined with other statements in a batch. The CREATE statement must begin the batch. All other statements that follow in that batch will be interpreted as part of the definition of the first CREATE statement.

- A table cannot be altered and then the new columns referenced in the same batch.
- If an EXECUTE statement is the first statement in a batch, the EXECUTE keyword is not required. The EXECUTE keyword is required if the EXECUTE statement is not the first statement in the batch.

Specifying Batches

Batches are implemented as part of the database APIs.

- In ADO, a batch is the string of Transact-SQL statements enclosed in the CommandText property of a Command object: Dim Cmd As New ADODB.Command Set Cmd.ActiveConnection = Cn Cmd.CommandText = "SELECT * FROM Suppliers; SELECT Cmd.CommandType = adCmdText Cmd.Execute
- In OLE DB, a batch is the string of Transact-SQL statements enclosed in the string used to set the command text: WCHAR* wszSQLString = L"SELECT * FROM Employees; SELECT * FROM Products" hr = pICommandText->SetCommandText (DBGUID_DBSQL, wszSQLString)
- In ODBC, a batch is the string of Transact-SQL statements enclosed on a **SQLPrepare** or **SQLExecDirect** call:

SQLExecDirect(hstmt1,

"SELECT * FROM Employees; SELECT * FROM Products SQL_NTS):

 In DB-Library, a batch is comprised of the Transact-SQL statements stored in the command buffer using **dbcmd** or **dbfcmd** before **dbsqlsend** or **dbsqlexec** are called: dbcmd (dbproc,

"SELECT * FROM Suppliers; SELECT * FROM Products") dbsqlexec (dbproc);

Some data access tools, such as Microsoft® Access, do not have an explicit

batch terminator.

The GO Command

SQL Query Analyzer, the **osql** utility, and the **isql** utility use the GO command to signal the end of a batch. GO is not a Transact-SQL statement; it simply signals to the utilities how many SQL statements should be included in a batch. In SQL Query Analyzer and **osql**, all the Transact-SQL statements from one GO command to the next are put in the string sent to **SQLExecDirect**. In **isql**, all the Transact-SQL statements between GO commands are placed into the command buffer before being executed.

For example, if these statements are executed in SQL Query Analyzer:

```
SELECT @@VERSION
SET NOCOUNT ON
GO
```

SQL Query Analyzer does the equivalent of:

```
SQLExecDirect(hstmt,
"SELECT @@VERSION SET NOCOUNT ON",
SQL_NTS);
```

Because a batch is compiled into a single execution plan, a batch must be logically complete. The execution plan created for one batch has no ability to reference any variables declared in another batch. Comments must both start and end in one batch. For more information, see <u>SQL Query Analyzer</u>.

See Also

<u>GO</u>

osql Utility

Batch Examples

These examples are scripts that use SQL Query Analyzer and the **osql** utility GO command to define batch boundaries.

This example creates a view. Because CREATE VIEW must be the only statement in a batch, the GO commands are required to isolate the CREATE VIEW statement from the USE and SELECT statements around it.

USE pubs GO /* Signals the end of the batch */

CREATE VIEW auth_titles AS SELECT * FROM authors GO /* Signals the end of the batch */

SELECT * FROM auth_titles GO /* Signals the end of the batch */

This example shows several batches combined into one transaction. The BEGIN TRANSACTION and COMMIT statements delimit the transaction boundaries. The BEGIN TRANSACTION, USE, CREATE TABLE, SELECT, and COMMIT statements are all in their own single-statement batches. All of the INSERT statements are included in one batch.

```
BEGIN TRANSACTION
GO
USE pubs
GO
CREATE TABLE mycompanies
(
```

id_num int IDENTITY(100, 5), company_name nvarchar(100)) GO INSERT mycompanies (company_name) VALUES ('New Moon Books') INSERT mycompanies (company_name) VALUES ('Binnet & Hardley') INSERT mycompanies (company_name) VALUES ('Algodata Infosystems') INSERT mycompanies (company_name) VALUES ('Five Lakes Publishing') INSERT mycompanies (company_name) VALUES ('Ramona Publishers') INSERT mycompanies (company_name) VALUES ('GGG&G') INSERT mycompanies (company_name) VALUES ('Scootney Books') INSERT mycompanies (company_name) VALUES ('Lucerne Publishing') GO **SELECT** * FROM mycompanies ORDER BY company_name ASC GO COMMIT GO

The following script illustrates two problems. First, the variable **@MyVar** is declared in the second batch and referenced in the third. Also, the second batch has the start of a comment, but no end. The third batch has the end of the comment, but when **osql** reads the GO command it sends the first batch to Microsoft® SQL ServerTM where the /* with no matching */ generates a syntax error.

USE Northwind GO DECLARE @MyVar INT /* Start of the split comment. GO End of the split comment. */ SELECT @MyVar = 29 GO

Stored Procedures and Triggers

A stored procedure is a group of Transact-SQL statements that is compiled one time, and then can be executed many times. This increases performance when the stored procedure is executed because the Transact-SQL statements do not have to be recompiled.

A trigger is a special type of stored procedure that is not called directly by a user. When the trigger is created, it is defined to execute when a specific type of data modification is made against a specific table or column.

A CREATE PROCEDURE or CREATE TRIGGER statement cannot span batches. This means that a stored procedure or trigger is always created in a single batch and compiled into an execution plan.

See Also

<u>Stored Procedures</u> <u>Enforcing Business Rules with Triggers</u>

Transact-SQL Scripts

A script is a series of Transact-SQL statements stored in a file. The file can be used as input to SQL Query Analyzer or the **osql** and **isql** utilities. The utilities then execute the SQL statements stored in the file.

Transact-SQL scripts have one or more batches. The GO command signals the end of a batch. If a Transact-SQL script does not have any GO commands, it is executed as a single batch.

Transact-SQL scripts can be used to:

- Keep a permanent copy of the steps used to create and populate the databases on your server (a backup mechanism).
- Transfer the statements from one computer to another, when necessary.
- Quickly educate new employees by enabling them to find problems in the code, understand the code, or change the code.

Using Variables and Parameters

Transact-SQL has several ways to pass data between Transact-SQL statements. Among these are:

• Transact-SQL local variables.

A Transact-SQL variable is an object in Transact-SQL batches and scripts that can hold a data value. After the variable has been declared, or defined, one Transact-SQL statement in a batch can set the variable to a value and a later statement in the batch can get the value from the variable. For example:

DECLARE @EmpIDVar INT

SET @EmpIDVar = 1234

```
SELECT *
FROM Employees
WHERE EmployeeID = @EmpIDVar
```

• Transact-SQL parameters.

A parameter is an object used to pass data between a stored procedure and the batch or script that executes the stored procedure. Parameters can be either input or output parameters. For example:

```
CREATE PROCEDURE ParmSample @EmpIDParm INT AS
SELECT *
FROM Employees
WHERE EmployeeID = @EmpIDParm
GO
```

```
EXEC ParmSample @EmpIDParm = 1234
GO
```

Applications use application variables and parameter markers to work with the data from Transact-SQL statements.

• Application variables

The application programming languages such as C, C++, Basic, and Java have their own variables for holding data. Applications using the database APIs must move the data returned by Transact-SQL statements into application variables before they can work with the data. This is typically done using a process called binding. The application uses an API function to bind the result set column to a program variable. When a row is fetched the API provider or driver moves the data from the column to the bound program variable.

• Parameter markers

Parameter markers are supported by the ADO, OLE DB, and ODBCbased database APIs. A parameter marker is a question mark (?) placed in the location of an input expression in a Transact-SQL statement. The parameter marker is then bound to an application variable. This allows data from application variables to be used as input in Transact-SQL statements. Parameter markers also let stored procedure output parameters and return codes be bound to application variables. The output data is then returned to the bound variables when the procedure is executed. The DB-Library API also supports binding stored procedure parameter and return codes to program variables.

See Also

DECLARE @local_variable

SELECT

Functions

SET @local variable

Transact-SQL Variables

A Transact-SQL local variable is an object that can hold a single data value of a specific type. Variables in batches and scripts are typically used:

- As a counter either to count the number of times a loop is performed or to control how many times the loop is performed.
- To hold a data value to be tested by a control-of-flow statement.
- To save a data value to be returned by a stored procedure return code.

The following script creates a small test table and populates it with 26 rows. The script uses a variable to do three things:

- Control how many rows are inserted by controlling how many times the loop is executed.
- Supply the value inserted into the integer column.
- Function as part of the expression that generates letters to be inserted into the character column.

-- Create the table. CREATE TABLE TestTable (cola INT, colb CHAR(3)) GO SET NOCOUNT ON GO -- Declare the variable to be used. DECLARE @MyCounter INT

-- Initialize the variable. SET @MyCounter = 0 -- Test the variable to see if the loop is finished.

WHILE (@MyCounter < 26)

BEGIN

-- Insert a row into the table.

INSERT INTO TestTable VALUES

- -- Use the variable to provide the integer value
- -- for cola. Also use it to generate a unique letter
- -- for each row. Use the ASCII function to get the
- -- integer value of 'a'. Add @MyCounter. Use CHAR to

```
-- convert the sum back to the character @MyCounter
```

-- characters after 'a'.

(@MyCounter,

```
CHAR( ( @MyCounter + ASCII('a') ) )
```

```
)
```

-- Increment the variable to count this iteration

```
-- of the loop.
```

```
SET @MyCounter = @MyCounter + 1
```

END

GO

```
SET NOCOUNT OFF
```

```
GO
```

Declaring a Transact-SQL Variable

The DECLARE statement initializes a Transact-SQL variable by:

- Assigning a name. The name must have a single @ as the first character.
- Assigning a system-supplied or user-defined data type and a length. For numeric variables, a precision and scale are also assigned.
- Setting the value to NULL.

Note Use system-supplied data types for local variables to minimize future maintenance issues.

For example, the following DECLARE statement creates a local variable named *@mycounter* with an *int* data type.

DECLARE @MyCounter INT

To declare more than one local variable, use a comma after the first local variable defined, and then specify the next local variable name and data type.

For example, this DECLARE statement creates three local variables named *@last_name*, *@fname* and *@state*, and initializes each to NULL:

DECLARE @LastName NVARCHAR(30), @FirstName NVARCHAI

The scope of a variable is the range of Transact-SQL statements that can reference the variable. The scope of a variable lasts from the point it is declared until the end of the batch or stored procedure in which it is declared. For example, this script generates a syntax error because the variable is declared in one batch and referenced in another:

DECLARE MyVariable INT SET @MyVariable = 1 GO -- This terminates the batch. -- @MyVariable has gone out of scope and no longer exists.

-- This SELECT statement gets a syntax error because it is -- no longer legal to reference @MyVariable. SELECT * FROM Employees WHERE EmployeeID = @MyVariable

Setting a Value in a Transact-SQL Variable

When a variable is first declared, its value is set to NULL. To assign a value to a variable, use the SET statement. This is the preferred method of assigning a value to a variable. A variable can also have a value assigned by being

referenced in the select list of a SELECT statement.

To assign a variable a value by using the SET statement, include the variable name and the value to assign to the variable. This is the preferred method of assigning a value to a variable. This batch, for example, declares two variables, assigns values to them, and then uses them in the WHERE clause of a SELECT statement:

USE Northwind GO -- Declare two variables. DECLARE @FirstNameVariable NVARCHAR(20), @RegionVariable NVARCHAR(30)

-- Set their values. SET @FirstNameVariable = N'Anne' SET @RegionVariable = N'WA'

```
-- Use them in the WHERE clause of a SELECT statement.
SELECT LastName, FirstName, Title
FROM Employees
WHERE FirstName = @FirstNameVariable
OR Region = @RegionVariable
GO
```

A variable can also have a value assigned by being referenced in a select list. If a variable is referenced in a select list, it should be assigned a scalar value or the SELECT statement should only return one row. For example:

USE Northwind GO DECLARE @EmpIDVariable INT

SELECT @EmpIDVariable = MAX(EmployeeID) FROM Employees GO If a SELECT statement returns more than one row and the variable references a nonscalar expression, the variable is set to the value returned for the expression in the last row of the result set. For example, in this batch **@EmpIDVariable** is set to the **EmployeeID** value of the last row returned, which is 1:

USE Northwind GO DECLARE @EmpIDVariable INT

SELECT @EmpIDVariable = EmployeeID FROM Employees ORDER BY EmployeeID DESC

SELECT @EmpIDVariable GO

See Also

DECLARE @local_variable SET @local_variable SELECT

Parameters

Parameters are used to exchange data between stored procedures and the application or tool that called the stored procedure:

- Input parameters allow the caller to pass a data value to the stored procedure.
- Output parameters allow the stored procedure to pass a data value or a cursor variable back to the caller.
- Every stored procedure returns an integer return code to the caller. If the stored procedure does not explicitly set a value for the return code, the return code is 0.

The following stored procedure shows the use of an input parameter, an output parameter, and a return code:

USE Northwind

GO

-- Create a procedure that takes one input parameter

-- and returns one output parameter and a return code.

```
CREATE PROCEDURE SampleProcedure @EmployeeIDParm INT,
@MaxQuantity INT OUTPUT
```

AS

```
-- Declare and initialize a variable to hold @@ERROR.
DECLARE @ErrorSave INT
SET @ErrorSave = 0
```

-- Do a SELECT using the input parameter. SELECT FirstName, LastName, Title FROM Employees WHERE EmployeeID = @EmployeeIDParm -- Save any nonzero @@ERROR value. IF (@@ERROR <> 0) SET @ErrorSave = @@ERROR

```
-- Set a value in the output parameter.
SELECT @MaxQuantity = MAX(Quantity)
FROM [Order Details]
```

```
IF (@@ERROR <> 0)
SET @ErrorSave = @@ERROR
```

```
-- Returns 0 if neither SELECT statement had
-- an error; otherwise, returns the last error.
RETURN @ErrorSave
GO
```

When a stored procedure is executed, input parameters can either have their value set to a constant or use the value of a variable. Output parameters and return codes must return their values into a variable. Parameters and return codes can exchange data values with either Transact-SQL variables or application variables.

If a stored procedure is called from a batch or script, the parameters and return code values can use Transact-SQL variables defined in the same batch. This example is a batch that executes the procedure created earlier. The input parameter is specified as a constant and the output parameter and return code place their values in Transact-SQL variables:

-- Declare the variables for the return code and output parameter. DECLARE @ReturnCode INT DECLARE @MaxQtyVariable INT

-- Execute the stored procedure and specify which variables

```
-- are to receive the output parameter and return code values.
```

EXEC @ReturnCode = SampleProcedure @EmployeeIDParm = 9, @MaxQuantity = @MaxQtyVariable OUTPUT

-- Show the values returned. PRINT ' PRINT 'Return code = ' + CAST(@ReturnCode AS CHAR(10)) PRINT 'Maximum Quantity = ' + CAST(@MaxQtyVariable AS CHAF GO

An application can use parameter markers bound to program variables to exchange data between application variables, parameters, and return codes.

See Also

SQL Stored Procedures Stored Procedures Parameter Markers CREATE PROCEDURE DECLARE @local_variable

Application Variables

The application programming languages such as C, C++, Basic, and Java use variables to hold data. Variables are areas of storage that can hold a data value. Each application variable has a data type and size. Numeric variables also have a precision (the number of digits the variable can hold) and scale (the number of digits that are to the right of the decimal point).

In order for an application to work with the data returned from Transact-SQL statements, it must have a mechanism to move the Transact-SQL data into application variables. The database APIs support the concept of binding a result set column, parameter, return code, or parameter marker in a Transact-SQL statement to an application variable.

To retrieve the data in a result set, an application uses a process similar to this:

- 1. Executes a Transact-SQL statement.
- 2. Calls a database API function to find out how many columns are in the result set.
- 3. For each result set column the application:
 - Calls a database API function that returns the attributes (data type, size, and so on) of the column.
 - Allocates an application variable with attributes compatible with the attributes of the column.
 - Calls a database API function to bind, or map, the result set column with the application variable.
- 4. Uses database API functions to fetch the result set rows one row at a time. On each fetch, the values of each result set column are placed in the application variable bound to the column.

Applications can vary this process. For example, if the application is executing a hard-coded Transact-SQL statement against a known table, the attributes of the result set columns are known in advance and the application does not have to call the database API to get these attributes.

If the application binds a result set column to an application variable whose data type differs from that of the associated database object, then the OLE DB provider or ODBC driver must convert the data. For example, if an application binds a **money** column to a character array, the OLE DB provider or ODBC driver has to convert the **money** data to a character string. The documentation for the Microsoft OLE DB Provider for SQL Server and the SQL Server ODBC driver define the data type conversions they support.

For more information about retrieving parameters and return codes in applications, see <u>Parameter Markers</u>.

See Also

Using the Fields Collection and Field Object

<u>Rowsets</u>

Processing Results

Results Processing

Parameter Markers

Parameter markers are supported by the ADO, OLE DB, and ODBC-based database APIs. A parameter marker is a question mark (?) placed in the location of an input or output expression in a Transact-SQL statement. Parameter markers allow an application to optimize the case where the same Transact-SQL statement is executed several times with different values for the input and output expressions.

For example, a user may have given an application five different stock symbols and the application has to call a stored procedure that gets the current data for each stock. The application could:

- Prepare this Transact-SQL statement: EXEC GetQuoteProcedure @StockSymbolParameter = ?
- Bind an application variable to the parameter marker (?).
- Execute a loop:
 - Move the next stock symbol to the bound variable.
 - Execute the statement to retrieve the quote for that stock. (there are symbols)

Parameter markers are not limited to being mapped to stored procedure parameters. They can be used anywhere an input expression is used:

UPDATE Employees SET Title = ? WHERE EmployeeID = ?

Parameter markers can also be used to map stored procedure output parameters and return codes. When the application executes a stored procedure, the OLE DB provider or ODBC driver moves the data values from any output parameters or return codes into the variables bound to the parameter's markers. For example, an application can execute this procedure, which returns an integer return code and a character output parameter:

- 1. Prepare a statement:
 {? = CALL MyProc (?)}
- 2. Bind the first parameter marker to an integer variable and the second marker to a character array.
- 3. Execute the statement.
- 4. Fetch or cancel all the result sets returned by the stored procedure.

At this point, the Microsoft OLE DB Provider for SQL Server or SQL Server ODBC driver will have placed the return code and output parameter value in the bound variables. Microsoft® SQL Server[™] returns output parameter and return code values in the last packet it returns to the client. Therefore, the application must process or cancel all result sets returned by the stored procedure before it has access to the return code and output parameter values.

The ADO API has a variation on this process for executing stored procedures. An ADO application:

- 1. Sets the **Command** object type to **adCmdStoredProc**.
- 2. Sets the command text to just the name of the procedure.
- 3. Builds a **Parameters** collection binding all the parameters and return codes to application variables.
- 4. Executes the **Command** object.

Parameter markers are associated with a database object that has a specific data type. If the application binds a parameter marker to a variable whose data type differs from that of the associated database object, then the OLE DB provider or

ODBC driver must convert the data. For example, if an application binds an integer return code to a character array, then the OLE DB provider or ODBC driver must convert the return code integer data to a character string. For information about the data type conversions that are supported, see the documentation for OLE DB Provider for SQL Server and SQL Server ODBC driver.

See Also

Executing Prepared Statements Parameters Collection Parameter Object Running Stored Procedures (OLE DB) Command Parameters Using Statement Parameters

Control-of-Flow

Transact-SQL provides special words called control-of-flow language that control the flow of execution of Transact-SQL statements, statement blocks, and stored procedures. These words can be used in ad hoc Transact-SQL statements, in batches, and in stored procedures.

Without control-of-flow language, separate Transact-SQL statements are performed sequentially, as they occur. Control-of-flow language permits statements to be connected, related to each other, and made interdependent using programming-like constructs.

These control-of-flow words are useful when you need to direct Transact-SQL to take some kind of action. For example, use a BEGIN...END pair of statements when including more than one Transact-SQL statement in a logical block. Use an IF...ELSE pair of statements when a certain statement or block of statements needs to be executed IF some condition is met, and another statement or block of statements should be executed if that condition is not met (the ELSE condition).

The control-of-flow statements cannot span multiple batches or stored procedures.

These are the control	-of-flow keywords.
-----------------------	--------------------

BEGINEND	WAITFOR
GOTO	WHILE
IFELSE	BREAK
RETURN	CONTINUE

See Also

Using CASE

<u>CONTINUE</u>

Using GOTO

DECLARE @local_variable

Using RETURN

ELSE (IF...ELSE)

Using WAITFOR

END (BEGIN...END)

Control-of-Flow Language

EXECUTE

-- (Comment)

IF...ELSE

<u>/*...*/ (Comment)</u>

<u>PRINT</u>

BEGIN...END

RAISERROR

BREAK

<u>WHILE</u>

Using BEGIN...END

The BEGIN and END statements are used to group multiple Transact-SQL statements into a logical block. Use the BEGIN and END statements anywhere a control-of-flow statement must execute a block of two or more Transact-SQL statements.

For example, when an IF statement controls the execution of only one Transact-SQL statement, no BEGIN or END statement is needed:

```
IF (@@ERROR <> 0)
SET @ErrorSaveVariable = @@ERROR
```

If @@ERROR is 0, only the single SET statement is jumped.

Use BEGIN and END statements to make the IF statement skip a block of statements when it evaluates to FALSE:

```
IF (@@ERROR <> 0)
BEGIN
SET @ErrorSaveVariable = @@ERROR
PRINT 'Error encountered, ' +
CAST(@ErrorSaveVariable AS VARCHAR(10))
END
```

The BEGIN and END statements must be used as a pair: one cannot be used without the other. The BEGIN statement appears on a line by itself followed by the block of Transact-SQL statements. Finally, the END statement appears on a line by itself to indicate the end of the block.

The BEGIN and END statements are used when:

- A WHILE loop needs to include a block of statements.
- An element of a CASE function needs to include a block of statements.

• An IF or ELSE clause needs to include a block of statements.

See Also

BEGIN...END

END (BEGIN...END)

Control-of-Flow Language

Using GOTO

The GOTO statement causes the execution of a Transact-SQL batch to jump to a label. None of the statements between the GOTO statement and the label are executed. The label name is defined using the syntax:

label_name:

Use the GOTO statement sparingly. Excessive use of the GOTO statement can make it difficult to understand the logic of a Transact-SQL batch. The logic implemented using GOTO can almost always be implemented using the other control-of-flow statements. GOTO is best used for breaking out of deeply nested control-of-flow statements.

The label that is the target of a GOTO identifies only the target of the jump. The label does nothing to isolate the statements following it from the statements immediately before it. Any user executing the statements immediately before the label skips the label and executes the statements after the label. This happens unless the statement immediately preceding the label is itself a control-of-flow statement, such as a RETURN.

This is an example of a GOTO:

IF (SELECT SYSTEM_USER()) = 'payroll'

- GOTO calculate_salary
- -- Other program code would appear here.
- -- When the IF statement evaluates to TRUE, the statements
- -- between the GOTO and the calculate_salary label are
- -- ignored. When the IF statement evaluates to FALSE the
- -- statements following the GOTO are executed.
- calculate_salary:

-- Statements to calculate a salary would appear after the label.

See Also

<u>GOTO</u>

Using IF...ELSE

The IF statement is used to test for a condition. The resulting flow of control depends on whether the optional ELSE statement is specified:

• IF specified without ELSE

When the IF statement evaluates to TRUE, the statement or block of statements following the IF statement are executed. When the IF statement evaluates to FALSE, the statement, or block of statements, following the IF statement is skipped.

• IF specified with ELSE

When the IF statement evaluates to TRUE, the statement, or block of statements, following the IF statement, is executed. Then control jumps to the point after the statement, or block of statements, following the ELSE statement. When the IF statement evaluates to FALSE, the statement, or block of statements, following the IF statement is skipped and the statement, or block of statements, following the optional ELSE statement is executed.

For example, if a stored procedure has been saving any error codes returned by @@ERROR during a transaction, it might have an IF statement similar to the following at the end of the procedure:

```
IF (@ErrorSaveVariable <> 0)
BEGIN
PRINT 'Errors encountered, rolling back.'
PRINT 'Last error encountered: ' +
CAST(@ErrorSaveVariable AS VARCHAR(10))
ROLLBACK
END
ELSE
BEGIN
PRINT 'No Errors encountered, committing.'
```

COMMIT END RETURN @ErrorSaveVariable

See Also

ELSE (IF...ELSE)

IF...ELSE

Using RETURN

The RETURN statement unconditionally terminates a query, stored procedure, or batch. None of the statements in a stored procedure or batch following the RETURN statement are executed.

When used in a stored procedure, the RETURN statement can specify an integer value to return to the calling application, batch, or procedure. If no value is specified on RETURN, a stored procedure returns the value 0.

Most stored procedures follow the convention of using the return code to indicate the success or failure of the stored procedure. The stored procedures return a value of 0 when no errors were encountered. Any nonzero value indicates an error occurred. For example:

USE Northwind

GO

-- Create a procedure that takes one input parameter

-- and returns one output parameter and a return code.

```
CREATE PROCEDURE SampleProcedure @EmployeeIDParm INT,
```

```
@MaxQuantity INT OUTPUT
```

AS

-- Declare and initialize a variable to hold @@ERROR.

```
DECLARE @ErrorSave INT
```

```
SET @ErrorSave = 0
```

-- Do a SELECT using the input parameter. SELECT FirstName, LastName, Title FROM Employees WHERE EmployeeID = @EmployeeIDParm

```
-- Save any nonzero @@ERROR value.
IF (@@ERROR <> 0)
SET @ErrorSave = @@ERROR
```

-- Set a value in the output parameter. SELECT @MaxQuantity = MAX(Quantity) FROM [Order Details]

IF (@@ERROR <> 0) SET @ErrorSave = @@ERROR

-- Returns 0 if neither SELECT statement had -- an error, otherwise returns the last error. RETURN @ErrorSave GO

A Transact-SQL batch or stored procedure that executes a stored procedure can retrieve the return code into an integer variable:

DECLARE @ReturnStatus INT DECLARE @MaxQtyVariable INT EXECUTE @ReturnStatus = SampleProcedure @EmployeeIDParm = @MaxQtyVariable = @MaxQuantity OUTPUT

```
-- Show the values returned.
PRINT '
PRINT 'Return code = ' + CAST(@ReturnStatus AS CHAR(10))
PRINT 'Maximum Quantity = ' + CAST(@MaxQtyVariable AS CHAF
GO
```

Applications that call a stored procedure can bind to an integer variable a parameter marker corresponding to the return code.

See Also

EXECUTE

<u>RETURN</u>

Parameter Markers

Using WAITFOR

The WAITFOR statement suspends the execution of a connection until either:

- A specified time interval has passed.
- A specified time of day is reached.

The WAITFOR statement is specified with one of two clauses:

- The DELAY keyword followed by an *amount_of_time_to_pass* before completing the WAITFOR statement. The time to wait before completing the WAITFOR statement can be up to 24 hours.
- The TIME keyword followed by a *time_to_execute*, which specifies completion of the WAITFOR statement.

This example uses the DELAY keyword to wait for two seconds before performing a SELECT statement:

WAITFOR DELAY '00:00:02' SELECT EmployeeID FROM Northwind.dbo.Employees

This example uses the TIME keyword to wait until 10 P.M. to perform a check of the **pubs** specified database to make sure that all pages are correctly allocated and used:

```
USE pubs
BEGIN
WAITFOR TIME '22:00'
DBCC CHECKALLOC
END
```

The disadvantage of the WAITFOR statement is that the connection from the application remains suspended until the WAITFOR completes. WAITFOR is

best used when an application or stored procedure must suspend processing for some relatively limited amount of time. Using SQL Server Agent or SQL-DMO to schedule a task is a better method of executing an action at a specific time of day.

See Also

WAITFOR

Using WHILE...BREAK or CONTINUE

The WHILE statement repeats a statement or block of statements as long as a specified condition remains true.

Two Transact-SQL statements are commonly used with WHILE: BREAK or CONTINUE. The BREAK statement exits the innermost WHILE loop and the CONTINUE statement restarts a WHILE loop. A program might execute a BREAK statement if, for example, there are no other rows to process. A CONTINUE statement could be executed if, for example, the execution of the code should continue.

Note If a SELECT statement is used as the condition for the WHILE statement, the SELECT statement must be in parentheses.

This example uses a WHILE statement to control how many fetches are done:

USE Northwind GO DECLARE abc CURSOR FOR SELECT * FROM Shippers

OPEN abc

FETCH NEXT FROM abc WHILE (@@FETCH_STATUS = 0) FETCH NEXT FROM abc

CLOSE abc DEALLOCATE abc GO

Other valid WHILE condition tests could be the following:

WHILE (@ACounterVariable < 100)

Or

WHILE EXISTS(SELECT au_lname FROM authors WHERE au_fnar

See Also

BEGIN...END END (BEGIN...END) BREAK WHILE

CONTINUE

Using CASE

The CASE function is a special Transact-SQL expression that allows an alternative value to be displayed depending on the value of a column. This change in data is temporary; therefore, there are no permanent changes to the data. For example, the CASE function can display California in a query result set for rows that have the value CA in the **state** column.

The CASE function consists of:

- The CASE keyword.
- The column name to transform.
- WHEN clauses specifying the expressions to search for and THEN clauses specifying the expressions to replace them with.
- The END keyword.
- An optional AS clause defining an alias for the CASE function.

This example displays, in the query result set, the full name of the state each author lives in:

SELECT au_fname, au_lname, CASE state WHEN 'CA' THEN 'California' WHEN 'KS' THEN 'Kansas' WHEN 'KS' THEN 'Kansas' WHEN 'TN' THEN 'Tennessee' WHEN 'OR' THEN 'Oregon' WHEN 'OR' THEN 'Oregon' WHEN 'MI' THEN 'Michigan' WHEN 'IN' THEN 'Indiana' WHEN 'UT' THEN 'Utah' END AS StateName FROM pubs.dbo.authors ORDER BY au_lname

See Also

<u>CASE</u>

NULLIF

COALESCE

Conditional Data Processing Using CASE

Building Statements at Run Time

Most Microsoft® SQL Server[™] applications that have to dynamically build SQL statements at run time do so before calling a database API function or method to execute the statement. For example, a C-language application using ODBC can dynamically build one or more SQL statements into a character array, then pass that array to the ODBC **SQLPrepare** or **SQLExecDirect** functions.

Transact-SQL supports two methods of building SQL statements at run time in Transact-SQL scripts, stored procedures, and triggers:

- Use the **sp_executesql** system stored procedure to execute a Unicode string. **sp_executesql** supports parameter substitution similar to the RAISERROR statement.
- Use the EXECUTE statement to execute a character string. The EXECUTE statement does not support parameter substitution in the executed string.

This is a simple example of using **sp_executesql** to execute a dynamically built string containing an SQL statement:

USE Northwind DECLARE @SQLString NVARCHAR(500)

/* Set column list. CHAR(13) is a carriage return, line feed.*/
SET @SQLString = N'SELECT FirstName, LastName, Title' + CHAR

/* Set FROM clause with carriage return, line feed. */
SET @SQLString = @SQLString + N'FROM Employees' + CHAR(13)

/* Set WHERE clause. */ SET @SQLString = @SQLString + N'WHERE LastName LIKE "D%'

EXEC sp_executesql @SQLString

GO

Using sp_executesql

Using **sp_executesql** is recommended over using the EXECUTE statement to execute a string. Not only does the support for parameter substitution make **sp_executesql** more versatile than EXECUTE, it also makes **sp_executesql** more efficient because it generates execution plans that are more likely to be reused by SQL Server.

Self-contained Batches

When either **sp_executesql** or the EXECUTE statement executes a string, the string is executed as its own self-contained batch. SQL Server compiles the Transact-SQL statement or statements in the string into an execution plan that is separate from the execution plan of the batch that contained the **sp_executesql** or EXECUTE statement. These rules apply for self-contained batches:

- The Transact-SQL statements in the **sp_executesql** or EXECUTE string are not compiled into an execution plan until **sp_executesql** or the EXECUTE statement are executed. The strings are not parsed or checked for errors until they are executed. The names referenced in the strings are not resolved until they are executed.
- The Transact-SQL statements in the executed string do not have access to any of the variables declared in the batch that contains the **sp_executesql** or EXECUTE statement. The batch containing the **sp_executesql** or EXECUTE statement does not have access to variables or local cursors defined in the executed string.
- If the executed string has a USE statement that changes the database context, the change to the database context only lasts until **sp_executesql** or the EXECUTE statement completes.

Executing these two batches illustrates these points:

/* Show not having access to variables from the calling batch. */

```
DECLARE @CharVariable CHAR(3)
SET @CharVariable = 'abc'
/* sp_executesql fails because @CharVariable has gone out of scope. *
sp_executesql N'PRINT @CharVariable'
GO
```

```
/* Show database context resetting after sp_executesql completes. */
USE pubs
GO
sp_executesql N'USE Northwind'
GO
/* This statement fails because the database context
has now returned to pubs. */
SELECT * FROM Shippers
GO
```

Substituting Parameter Values

sp_executesql supports the substitution of parameter values for any parameters specified in the Transact-SQL string, but the EXECUTE statement does not. Therefore, the Transact-SQL strings generated by **sp_executesql** are more similar than those generated by the EXECUTE statement. The SQL Server query optimizer will probably match the Transact-SQL statements from **sp_executesql** with execution plans from the previously executed statements, saving the overhead of compiling a new execution plan.

With the EXECUTE statement, all parameter values must be converted to character or Unicode and made a part of the Transact-SQL string:

```
DECLARE @IntVariable INT
DECLARE @SQLString NVARCHAR(500)
/* Build and execute a string with one parameter value. */
SET @IntVariable = 35
SET @SQLString = N'SELECT * FROM pubs.dbo.employee WHERE
CAST(@IntVariable AS NVARCHAR(10))
EXEC(@SQLString)
```

```
/* Build and execute a string with a second parameter value. */
SET @IntVariable = 201
SET @SQLString = N'SELECT * FROM pubs.dbo.employee WHERE
CAST(@IntVariable AS NVARCHAR(10))
EXEC(@SQLString)
```

If the statement is executed repeatedly, a completely new Transact-SQL string must be built for each execution, even when the only differences are in the values supplied for the parameters. This generates extra overhead in several ways:

- The ability of the SQL Server query optimizer to match the new Transact-SQL string with an existing execution plan is hampered by the constantly changing parameter values in the text of the string, especially in complex Transact-SQL statements.
- The entire string must be rebuilt for each execution.
- Parameter values (other than character or Unicode values) must be cast to a character or Unicode format for each execution.

sp_executesql supports the setting of parameter values separately from the Transact-SQL string:

DECLARE @IntVariable INT DECLARE @SQLString NVARCHAR(500) DECLARE @ParmDefinition NVARCHAR(500)

/* Build the SQL string once. */

SET @SQLString =

N'SELECT * FROM pubs.dbo.employee WHERE job_lvl = @level /* Specify the parameter format once. */ SET @ParmDefinition = N'@level tinyint'

/* Execute the string with the first parameter value. */

```
SET @IntVariable = 35
EXECUTE sp_executesql @SQLString, @ParmDefinition,
    @level = @IntVariable
/* Execute the same string with the second parameter value. */
SET @IntVariable = 32
EXECUTE sp_executesql @SQLString, @ParmDefinition,
    @level = @IntVariable
```

This **sp_executesql** example accomplishes the same task as the EXECUTE example shown earlier, but with these additional benefits:

- Because the actual text of the Transact-SQL statement does not change between executions, the query optimizer should match the Transact-SQL statement in the second execution with the execution plan generated for the first execution. Therefore, SQL Server does not have to compile the second statement.
- The Transact-SQL string is built only once.
- The integer parameter is specified in its native format. Conversion to Unicode is not required.

Note Object names in the statement string must be fully qualified in order for SQL Server to reuse the execution plan.

Reusing Execution Plans

In earlier versions of SQL Server, the only way to be able to reuse execution plans was to define the Transact-SQL statements as a stored procedure and have the application execute the stored procedure. This generates extra administrative overhead for the applications. Using **sp_executesql** can help reduce this overhead while still allowing SQL Server to reuse execution plans. **sp_executesql** can be used instead of stored procedures when executing a Transact-SQL statement a number of times, when the only variation is in the parameter values supplied to the Transact-SQL statement. Because the Transact-SQL statements themselves remain constant and only the parameter values change, the SQL Server query optimizer is likely to reuse the execution plan it generates for the first execution.

This example builds and executes a DBCC CHECKDB statement for every database on a server, except for the four system databases:

USE master GO SET NOCOUNT ON GO DECLARE AllDatabases CURSOR FOR SELECT name FROM sysdatabases WHERE dbid > 4

OPEN AllDatabases

DECLARE @DBNameVar NVARCHAR(128) DECLARE @Statement NVARCHAR(300)

```
FETCH NEXT FROM AllDatabases INTO @DBNameVar
WHILE (@@FETCH_STATUS = 0)
BEGIN
PRINT N'CHECKING DATABASE ' + @DBNameVar
SET @Statement = N'USE ' + @DBNameVar + CHAR(13)
+ N'DBCC CHECKDB (' + @DBNameVar + N')'
EXEC sp_executesql @Statement
PRINT CHAR(13) + CHAR(13)
FETCH NEXT FROM AllDatabases INTO @DBNameVar
END
```

CLOSE AllDatabases DEALLOCATE AllDatabases GO SET NOCOUNT OFF GO The SQL Server ODBC driver uses **sp_executesql** to implement **SQLExecDirect** when the Transact-SQL statement being executed contains bound parameter markers. The one exception is that **sp_executesql** is not used with data-at-execution parameters. This allows applications that use the standard ODBC functions, or that use the APIs defined over ODBC (such as RDO), to gain the advantages provided by **sp_executesql**. Existing ODBC applications ported to SQL Server 2000 automatically acquire the performance gains without having to be rewritten. For more information, see <u>Using Statement Parameters</u>.

The Microsoft OLE DB Provider for SQL Server also uses **sp_executesql** to implement the direct execution of statements with bound parameters. Applications using OLE DB or ADO gain the advantages provided by **sp_executesql** without having to be rewritten.

See Also

DECLARE @local_variable

<u>SELECT</u>

EXECUTE

Permissions

Every object in a Microsoft® SQL Server[™] 2000 database has an owner, typically the user ID in effect for the connection that created the object. Other users cannot access that object until the owner authorizes their user ID to access the object.

Certain Transact-SQL statements are also limited to specific user IDs. For example, CREATE DATABASE is limited to members of the **sysadmin** and **dbcreator** fixed server roles. Users cannot access an object or execute a statement unless authorized.

All Transact-SQL statements that a user issues are subject to the permissions the user has been granted. Members of the **sysadmin** fixed server role, members of the **db_owner** fixed database role, and owners of database objects can grant, deny, or revoke permissions for a person or role. When using Transact-SQL, use the GRANT, DENY, and REVOKE statements to specify who can use which data-modification statements:

- GRANT gives permissions to either work with data or execute other Transact-SQL statements.
- DENY denies permission and prevents the specified user, group, or role from inheriting the permission through group and role memberships.
- REVOKE removes previously granted or denied permissions.

The permissions that can be granted for objects are:

• SELECT

Permits a user to issue SELECT statements against a table or view.

• INSERT

Permits a user to issue INSERT statements against a table or view.

• UPDATE

Permits a user to issue UPDATE statements against a table or view.

• DELETE

Permits a user to issue DELETE statements against a table or view.

• REFERENCES

Permits a user to make foreign key references to the primary key and unique columns of a table. Also used to allow SCHEMABINDING references from views and functions.

• EXECUTE

Permits a user to issue EXECUTE statements against a stored procedure.

Permissions can also be granted to execute Transact-SQL statements that are usually limited to members of a specific role. For example, a member of the **sysadmin** fixed server role can grant CREATE DATABASE permissions to a user who usually could not create databases.

See Also

Logins DENY Logins, Users, Roles, and Groups GRANT Managing Security REVOKE

Using Options in SQL Server

Microsoft® SQL Server[™] 2000 provides options that affect the result and performance of SQL statements. Transact-SQL allows you to set these options in the following ways:

- Server-wide configuration options (server options) are set by executing the **sp_configure** stored procedure.
- Database-level options (database options) are set by executing the **sp_dboption** stored procedure.
- The database compatibility level is set by executing the **sp_dbcmptlevel** stored procedure.
- Connection-level options (SET options) are specified with SET statements, such as SET ANSI_PADDING and SET ANSI_NULLS.
- Statement-level options (query hints, table hints, and join hints) are specified in individual Transact-SQL statements.

ODBC applications can specify connection options that control some of the ANSI SET options. The Microsoft OLE DB Provider for SQL Server and SQL Server ODBC driver both set several SET options by default. Options can also be set using the SQL Server Enterprise Manager.

Avoid changing SET options and setting them through the SET statements. Instead, it is recommended that SET options be set at the connection level through the connection properties of ODBC or OLE DB. Alternatively, change SET option settings using the **sp_configure** stored procedure.

Hierarchy of Options

When an option is supported at more than one level:

- 1. A database option overrides a server option.
- 2. A SET option overrides a database option.
- 3. A hint overrides a SET option.

Note sp_configure provides the option **user options**, which allows you to change the default values of several SET options. Although **user options** appears to be a server option, it is a SET option.

SET Options

This table contains an alphabetic list of SET options and the corresponding database and server options supported in Microsoft® SQL Server[™] 2000.

	Database	Server	Default
SET option	option	option	setting
ANSI_DEFAULTS	None	None	n/a
ANSI_NULL_DFLT_OFF	ANSI null	user	OFF
<u>ANSI_NULL_DFLT_ON</u>	default	options	
		assigns a	
		default	
ANSI_NULLS	ANSI	user	OFF
	nulls	options	
		assigns a	
		default	
ANSI_PADDING	None	user	ON
		options	
		assigns a	
		default	
ANSI_WARNINGS	ANSI	user	OFF
	warnings	options	
		assigns a	
		default	
ARITHABORT	None	user	OFF
		options	
		assigns a	
		default	
ARITHIGNORE	None	user	OFF
		options	
		assigns a	
		default	
CONCAT_NULL_YIELDS_NULL	concat	None	OFF
	null yields		
	null		

CONTEXT_INFO	None	None	OFF
CURSOR CLOSE ON COMMIT	cursor	user	OFF
	close on	options	
	commit	assigns a	
		default	
DATEFIRST	None	None	7
DATEFORMAT	None	None	mdy
DEADLOCK_PRIORITY	None	None	NORMAL
DISABLE_DEF_CNST_CHK	None	user options	OFF
		assigns a default	
FIPS_FLAGGER	None	None	OFF
FMTONLY	None	None	OFF
SET FORCEPLAN	None	None	OFF
IDENTITY_INSERT	None		OFF
IMPLICIT_TRANSACTIONS	None	user	OFF
		options	
		assigns a	
		default	
LANGUAGE	None	None	us_english
LOCK_TIMEOUT	None	None	No limit
NOCOUNT	None	user	OFF
		options	
		assigns a default	
NOEXEC	None		OEE
NOEXEC	None	None None	OFF
NUMERIC_ROUNDABORT	None		OFF
<u>OFFSETS</u>	None	None	OFF
PARSEONLY	None	None	OFF
QUERY_GOVERNOR_COST_LIMIT	INone	query	OFF
		governor cost limit	
QUOTED IDENTIFIER	quoted	user	OFF
	identifier	options	
	lucifille	vpuolis	

		assigns a default	
REMOTE_PROC_TRANSACTIONS	None	None	OFF
ROWCOUNT	None	None	OFF
SHOWPLAN_ALL	None	None	OFF
SHOWPLAN_TEXT	None	None	OFF
STATISTICS IO	None	None	OFF
STATISTICS PROFILE	None	None	n/a
STATISTICS TIME	None	None	OFF
TEXTSIZE	None	None	OFF
TRANSACTION ISOLATION	None	None	n/a
LEVEL			
XACT_ABORT	None	None	OFF

Parse-Time and Execute-Time SET Options

The point at which a SET option takes effect depends upon whether the option is a parse-time option or an execute-time option. Parse-time options take effect during parsing, as the options are encountered in text, without regard to control of flow statements. Execute-time options take effect during the execution of the code in which they are specified. If execution fails before the SET statement is executed, the option is not set. If execution fails after the SET statement is executed, the option is set.

The QUOTED_IDENTIFIER, PARSEONLY, OFFSETS, and FIPS_FLAGGER options are parse-time options. All other SET options are execute-time options.

SET QUOTED_IDENTIFIER and SET ANSI_NULLS statements that occur within a batch or stored procedure do not affect that batch or stored procedure. Instead, the settings that are used for statements inside the batch or stored procedure are the settings that are in effect when the batch or stored procedure is created.

Duration of SET Options

This section describes the duration of SET options.

- SET options that are set by a user in a script apply until reset or the user's session with the server is terminated.
- SET options that are set within a stored procedure or trigger apply until reset inside that stored procedure or trigger, or until control returns to the code that invoked the stored procedure or trigger.
- Unless explicitly reset, SET option values from all higher level code apply within a stored procedure or trigger.
- Unless explicitly or implicitly reset, SET options set for a connection apply after connecting to a different database.

Note An additional consideration is that when a user connects to a database, some option may be set ON automatically, based on the values specified by the prior use of the **user options**, server option or the values that apply to all ODBC and OLE DB connections.

Shortcut SET Option

Transact-SQL provides the SET ANSI_DEFAULTS statement as a shortcut for setting these SQL-92 standard options:

- SET ANSI_NULLS
- SET CURSOR_CLOSE_ON_COMMIT
- SET ANSI_NULL_DFLT_ON
- SET IMPLICIT_TRANSACTIONS
- SET ANSI_PADDING

- SET QUOTED_IDENTIFIER
- SET ANSI_WARNINGS

The shortcut resets the values for these options. Any individual option set after the shortcut is used overrides the corresponding value set by the shortcut.

Note SET ANSI_DEFAULTS does not set all of the options required to comply with the SQL-92 standard.

Database Options

This table is an alphabetic list of database options and corresponding SET and server options supported in Microsoft® SQL Server[™] 2000.

Database		Server	Default
option	SET option	option	setting
ANSI null default	ANSI_NULL_DFLT_ON ANSI_NULL_DFLT_OFF	user options assigns a default	OFF
ANSI nulls	ANSI_NULLS	user options assigns a default	OFF
ANSI warnings	ANSI_WARNINGS	user options assigns a default	OFF
auto create statistics	None	None	ON
auto update statistics	None	None	ON
autoclose	None	None	FALSE ¹
autoshrink	None	None	FALSE
concat null yields null	CONCAT_NULL_YIELDS_NULL	None	OFF
cursor close on commit	CURSOR_CLOSE_ON_COMMIT	user options assigns a default	OFF
dbo use only	None	None	FALSE
default to local cursor	None	None	FALSE
merge publish	None	None	FALSE
offline	None	None	FALSE
published	None	None	FALSE

quoted identifier	QUOTED_IDENTIFIER	user options assigns a default	OFF
read only	None	None	FALSE
recursive triggers	None	None	FALSE
select into/ bulkcopy	None	None	FALSE
single user	None	None	FALSE
subscribed	None	None	TRUE
torn page detection		None	TRUE
trunc. log on chkpt.	None	None	TRUE

 1 By default, autoclose is set to TRUE in SQL Server 2000 Desktop Engine.

The default database options for a new database are those defined in the **model** database. In new SQL Server installations, the settings in the **model** and **master** databases are the same.

A change to a database option forces a recompile of everything in the cache.

Options and Database Context

The database context of scripts and the batches within scripts is determined by the most recent connection. The connection can be explicitly set with the USE statement in Transact-SQL and by both implicit and explicit means in other environments such as ODBC and OLE DB. For more information, see <u>Choosing a Database</u>.

When a stored procedure is executed from a batch or another stored procedure, it is executed under the option settings of the database in which it is stored. For example, when stored procedure **db1.dbo.sp1** calls stored procedure **db2.dbo.sp2**, **sp1** is executed under the current compatibility level setting of **db1**, and **sp2** is executed under the current compatibility level setting of **db2**.

When a Transact-SQL statement refers to objects in multiple databases, the current database context and the current connection context (the database

defined by the USE statement if it is in a batch, or the database that contains the stored procedure if it is in a stored procedure) apply to that statement.

Server Options

This table is an alphabetic list of server options and corresponding database and SET options supported in Microsoft® SQL Server[™] 2000.

	DalaDase	Default
SET option	option	setting
None	None	0
None	None	5
None	None	-1
None	None	1033
None	None	0
None	None	2147483647
None	None	65536
None	None	255
	None	None

<u>media</u>	None	None	0
<u>retention</u>			
min memory	None	None	1024
<u>per query</u>			
<u>min server</u>	None	None	0
memory			
nested triggers	None	None	1
<u>network packet</u>	None	None	4096
<u>size</u>			
<u>open objects</u>	None	None	0
priority boost	None	None	0
query governor	QUERY_GOVERNOR_COST_	None	0
	LIMIT		
query wait	None	None	-1
<u>recovery</u>	None	None	0
<u>interval</u>			
remote access	None	None	1
<u>remote login</u>	None	None	20
<u>timeout</u>			
<u>remote proc</u>	None	None	0
<u>trans</u>			
<u>remote query</u>	None	None	600
<u>timeout</u>			
<u>scan for</u>	None	None	0
<u>startup procs</u>			
set working set	None	None	0
<u>size</u>			
show advanced	None	None	0
<u>options</u>			
<u>two digit year</u>	None	None	2049
<u>cutoff</u>			
<u>user</u>	None	None	0
<u>connections</u>			
user options	ANSI_NULL_DFLT_ON	ANSI null	OFF

ANSI_NULL_DFLT_OFF	default
ANSI_NULLS	ANSI OFF
	nulls
ANSI_PADDING	None ON
ANSI_WARNINGS	ANSI OFF
	warnings
CURSOR_CLOSE_ON_COM	MMIT cursor OFF
	close on
	commit
IMPLICIT_TRANSACTIONS	IS None OFF
QUOTED_IDENTIFIER	quoted OFF
	identifier
ARITHABORT	None OFF
ARITHIGNORE	None OFF
DISABLE_DEF_CNST_CHK	K None OFF
NOCOUNT	None OFF

Hints

This table lists the options available for join hints, query hints, and table hints in Microsoft® SQL Server[™] 2000.

Hint			Default
type	Option	Description	setting
Join	LOOP HASH MERGE REMOTE	Specifies the strategy to use when joining the rows of two tables.	Chosen by SQL Server.
Query	{ HASH ORDER } GROUP	Specifies whether hashing or ordering is used to compute GROUP BY and COMPUTE aggregations.	Chosen by SQL Server.
Query	{ MERGE HASH CONCAT } UNION	Specifies the strategy to use for all UNION operations within the query.	Chosen by SQL Server.
Query	FAST integer	Optimizes the query for retrieval of the specified number of rows.	No such optimization.
Query	FORCE ORDER	Performs joins in the order in which the tables appear in the query.	Chosen by SQL Server.
Query	ROBUST PLAN	Creates a plan that accommodates maximum potential row size.	Chosen by SQL Server.
Table	FASTFIRSTROW	Has the same effect as specifying the FAST 1 query hint.	No such optimization.
Table	INDEX =	Instructs SQL Server to use the specified indexes for a table.	
Table	HOLDLOCK SERIALIZABLE REPEATABLEREAD	Specifies the isolation level for a table.	Defaults to a transaction isolation

	READCOMMITTED READUNCOMMITTED NOLOCK		level.
Table	ROWLOCK PAGLOCK TABLOCK TABLOCKX NOLOCK	Specifies locking granularity for a table.	Chosen by SQL Server.
Table	READPAST	Skips locked rows altogether.	Wait for locked rows.
Table	UPDLOCK	L T	Take shared locks.
Table		Takes an exclusive lock that will be held until the end of the transaction. Cannot be used with NOLOCK or UPDLOCK.	Chosen by SQL Server.

Database Compatibility Level Option

Transact-SQL provides the **sp_dbcmptlevel** stored procedure that sets certain database behaviors to be compatible with the specified earlier version of Microsoft® SQL ServerTM 2000.

A special rule applies to the relationship among the database compatibility level option, the **concat null yields null** database option, and the CONCAT_NULL_YIELDS_NULL SET option. The settings of **concat null yields null** and CONCAT_NULL_YIELDS_NULL are ignored when the value of compatibility level is for a release earlier than SQL Server 7.0.

The compatibility level affects the behaviors in the specified database, not the entire server. For more information, see <u>sp_dbcmptlevel</u>.

Behavior if Both ARITHABORT and ARITHIGNORE Are Set ON

If both the ARITHABORT and ARITHIGNORE query-processing options are set ON, ARITHABORT takes precedence.

ARITHABORT and ARITHIGNORE are two distinct options; setting one ON does not set the other OFF automatically. For example, if an application contains these statements then both options are set ON:

SET ARITHABORT ON SET ARITHIGNORE ON GO

When a SET statement is executed in a stored procedure, the new setting is active only until the procedure is completed. When the procedure is completed, the connection's setting for that option will go back to what it was before the procedure was executed.

Effect of ANSI_WARNINGS Setting

The ANSI_WARNINGS setting affect query processor behavior despite the current settings of ARITHABORT and ARITHIGNORE.

For example, even if SET ARITHABORT or SET ARITHIGNORE is OFF, if SET ANSI_WARNINGS is ON, SQL Server still returns an error message when encountering divide-by-zero or overflow errors.

This table summarizes the behavior.

ARITHABORT	ANSI_WARNINGS	Behavior
ON	ON	Abort statement only.
ON	OFF	Abort batch.
OFF	ON	Abort statement only.
OFF	OFF	Continue; value is
		NULL.

See Also

<u>SET ANSI_WARNINGS</u>

SET ARITHABORT

SET ARITHIGNORE

Query Fundamentals

A query is a request for data stored in Microsoft® SQL Server[™] 2000. A query can be issued using several forms:

- An MS Query or Microsoft Access user can use a graphical user interface (GUI) to pick the data the user wants to see from one or more SQL Server tables.
- A user of SQL Query Analyzer or the **osql** utility can issue a SELECT statement.
- A Microsoft Visual Basic® application can map the data from a SQL Server table into a bound control, such as a grid.

Although queries have various ways of interacting with a user, they all accomplish the same task: They present the result set of a SELECT statement to the user. Even if the user never specifies a SELECT statement, as is usually the case with graphical tools such as MS Query, the client software transforms each user query into a SELECT statement that is sent to SQL Server.

The SELECT statement retrieves data from SQL Server and presents it back to the user in one or more result sets. A result set is a tabular arrangement of the data from the SELECT. Like an SQL table, the result set comprises columns and rows.

The full syntax of the SELECT statement is complex, but most SELECT statements describe four primary properties of a result set:

- The number and attributes of the columns in the result set. These attributes must be defined for each result set column:
 - The data type of the column.
 - The size of the column, and for numeric columns, the precision and scale.

- The source of the data values returned in the column.
- The tables from which the result set data is retrieved, and any logical relationships between the tables.
- The conditions that the rows in the source tables must meet to qualify for the SELECT. Rows that do not meet the conditions are ignored.
- The sequence in which the rows of the result set are ordered.

This SELECT statement finds the product ID, name, and unit price of any products whose unit price exceeds \$40:

SELECT ProductID, ProductName, UnitPrice FROM Products WHERE UnitPrice > \$40 ORDER BY UnitPrice ASC

The column names listed after the SELECT keyword (**ProductID**, **ProductName**, and **UnitPrice**) form the select list. This specifies that the result set has three columns, and each column has the name, data type, and size of the associated column in the **Products** table. Because the FROM clause specifies only one base table, all column names in the SELECT statement refer to columns in that table.

The FROM clause lists the single table, **Products**, from which the data is to be retrieved.

The WHERE clause specifies that the only rows in **Products** that qualify for this SELECT are those in which the value of the **UnitPrice** column exceeds \$40.

The ORDER BY clause specifies that the result set is to be sorted in ascending sequence based on the value in the **UnitPrice** column.

See Also

FROM

<u>SELECT</u>

Distributed Queries

<u>WHERE</u>

Parts of a SELECT Statement

The full syntax of the SELECT statement is complex, but the main clauses can be summarized as:

```
SELECT select_list
[INTO new_table_name]
FROM table_list
[WHERE search_conditions]
[GROUP BY group_by_list]
[HAVING search_conditions]
[ORDER BY order_list [ASC | DESC] ]
```

select_list

Describes the columns of the result set. It is a comma-separated list of expressions. Each expression defines both the format (data type and size) and the source of the data for the result set column. Each select list expression is usually a reference to a column in the source table or view the data is coming from, but can be any other expression, such as a constant or a Transact-SQL function. Using the * expression in a select list specifies that all columns in the source table are returned.

INTO new_table_name

Specifies that the result set is used to create a new table. *new_table_name* specifies the name of the new table.

FROM table_list

Contains a list of the tables from which the result set data is retrieved. These sources can be:

- Base tables in the local server running Microsoft[®] SQL Server[™].
- Views in the local SQL Server. SQL Server internally resolves a view reference to references against the base tables that make up the view.

• Linked tables, which are tables in OLE DB data sources made accessible to SQL Server. This is called a distributed query. OLE DB data sources can be accessed from SQL Server by linking them as a linked server, or referencing the data source in an OPENROWSET or OPENQUERY function.

The FROM clause can also contain join specifications, which define the specific path SQL Server is to use in navigating from one table to another.

The FROM clause is also used on the DELETE and UPDATE statements to define the tables that are modified.

WHERE search_conditions

The WHERE clause is a filter that defines the conditions each row in the source tables must meet to qualify for the SELECT. Only rows that meet the conditions contribute data to the result set. Data from rows that do not meet the conditions are not used.

The WHERE clause is also used on the DELETE and UPDATE statements to define the rows in the target tables that are modified.

GROUP BY group_by_list

The GROUP BY clause partitions the result set into groups based on the values in the columns of the *group_by_list*. For example, the **Northwind Orders** table has three values in **ShipVia**. A GROUP BY **ShipVia** clause partitions the result set into three groups, one for each value of **ShipVia**.

HAVING search_conditions

The HAVING clause is an additional filter that is applied to the result set. Logically, the HAVING clause filters rows from the intermediate result set built from the application of any FROM, WHERE, or GROUP BY clauses in the SELECT statement. HAVING clauses are most commonly used with a GROUP BY clause, although a GROUP BY clause is not required before a HAVING clause.

ORDER BY order_list [ASC | DESC]

The ORDER BY clause defines the order in which the rows in the result set

are sorted. *order_list* specifies the result set columns that make up the sort list. The ASC and DESC keywords are used to specify if the rows are sorted in an ascending or descending sequence.

ORDER BY is important because relational theory specifies that the rows in a result set cannot be assumed to have any sequence unless ORDER BY is specified. ORDER BY must be used in any SELECT statement for which the order of the result set rows is important.

The clauses in a SELECT statement must be specified in the proper order.

Each reference to a database object must be unambiguous. Ambiguity can come from these sources:

- There may be multiple objects with the same name in a system. For example, both User1 and User2 may have defined a table named TableX. To resolve the ambiguity and specify the TableX owned by User1, qualify the table name with at least the user ID: SELECT * FROM User1.TableX
- The database in which the object resides may not always be the current database when the SELECT statement is executed. To ensure that the proper object is always used, regardless of the current database setting, qualify the object name with the database and owner:
 SELECT *
 FROM Northwind.dbo.Shippers
- The tables and views specified in the FROM clause may have duplicate column names. It is especially likely that foreign keys will have the same column name as their related primary key. To resolve the ambiguity between duplicate names, the column name must be qualified with the table or view name:

SELECT DISTINCT Customers.CustomerID, Customers.Comj FROM Customers JOIN Orders ON

(Customers.CustomerID = Orders.CustomerID) WHERE Orders.ShippedDate > 'May 1 1998' This syntax becomes cumbersome when the table and view names must themselves be fully qualified. This problem is resolved by assigning a correlation name (also known as a range variable or alias) to the table, using the AS keyword in the FROM clause. The fully qualified table or view name has to be specified only in the FROM clause. All other table or view references can then use the correlation name. Applying correlation names and fully qualifying the tables in the earlier sample results in this SELECT statement:

SELECT DISTINCT Cst.CustomerID, Cst.CompanyName FROM Northwind.dbo.Customers AS Cst JOIN Northwind.dbo.Orders AS Ord ON (Cst.CustomerID = Ord.CustomerID) WHERE Ord.ShippedDate > 'May 1 1998'

For more information about object qualification, see <u>Using Identifiers</u>.

Many Transact-SQL examples in the SQL Server Books Online are simplified by not using qualified names. Although these elements are left out of the examples to promote readability, it is recommended that Transact-SQL statements in production systems use qualified names.

See Also

Expressions SELECT IDENTITY (Property)

Using the Select List

The select list defines the columns in the result set of a SELECT statement. The select list is a series of expressions separated by commas. Each expression defines a column in the result set. The columns in the result set are in the same order as the sequence of expressions in the select list.

These attributes of the result set columns are defined by the expressions in the select list:

- The data type, size, precision, and scale of the result set column are the same as those of the expression defining the column.
- The name of the result set column is the name associated with the expression defining the column. The optional AS keyword can be used to change the name, or to assign a name if the expression has no name.
- The data values for the result set column are derived from the evaluation of the expression for each row of the result set.

The select list can also contain keywords controlling the final format of the result set:

• DISTINCT

The DISTINCT keyword eliminates duplicate rows from a result set. For example, there are many rows in the **Northwind Orders** table with the same value for **ShipCity**. To get a list of the **ShipCity** values with duplicates removed:

SELECT DISTINCT ShipCity, ShipRegion FROM Orders ORDER BY ShipCity

• TOP *n*

The TOP keyword specifies that the first *n* rows of the result set are

returned. If ORDER BY is specified, the rows are selected after the result set is ordered. *n* is the number of rows to return, unless the PERCENT keyword is specified. PERCENT specifies that *n* is the percentage of rows in the result set that are returned. For example, this SELECT statement returns the first 10 cities, in alphabetic sequence, from the **Orders** table:

SELECT DISTINCT TOP 10 ShipCity, ShipRegion FROM Orders ORDER BY ShipCity

The items in the select list can include:

- A simple expression: a reference to a function, a local variable, a constant, or a column in a table or view.
- A scalar subquery, which is a SELECT statement that evaluates to a single value for each result set row.
- A complex expression built by using operators on one or more simple expressions.
- The * keyword, which specifies that all columns in a table are returned.
- Variable assignment in the form @local_variable = expression. The SET @local_variable statement can also be used for variable assignment.
- The IDENTITYCOL keyword, which is resolved as a reference to the column in the table having the IDENTITY property. For example, the IDENTITY property has been defined for the **OrderID** column in the **Northwind Orders** table, so the expression Orders. IDENTITYCOL is equal to **Orders.OrderID**.

- The ROWGUILDCOL keyword, which is resolved as a reference to the column in a table having the ROWGUIDCOL property.
- Creating a new column (using SELECT INTO) that uses the IDENTITY property by using the specified syntax. For example, to create a new column named **counter** in the **authors** table that is an **int** column, you should start at a value of 100 and increment by values of 1 for each succeeding number, use **counter** = IDENTITY(**int**, 100, 1).
- Temporarily adding a column to the query results that designates whether the CUBE or ROLLUP operation added the row or not. Use the GROUPING keyword.

This example shows many of the items that can be in a select list:

```
SELECT FirstName + ' ' + LastName AS "Employee Name",
IDENTITYCOL AS "Employee ID",
HomePhone,
Region,
10 AS Constant
FROM Northwind.dbo.Employees
ORDER BY LastName, FirstName ASC
```

See Also

<u>SELECT</u>

Choosing All Columns

The asterisk (*) has a special meaning in SELECT statements:

- When specified without a qualifier, it is resolved as a reference to all columns in all tables or views specified in the FROM clause. This example retrieves all book information stored in the Shippers table: USE Northwind
 GO
 SELECT *
 FROM Shippers
 ORDER BY CompanyName
 GO
- When qualified with a table or view name, it is resolved as a reference to all the columns in the table or view. This example uses the asterisk to reference all the columns in the Shippers table:
 USE Northwind
 GO
 SELECT Orders.OrderID, Shippers.*
 FROM Shippers
 JOIN
 Orders
 ON (Shippers.ShipperID = Orders.ShipVia)
 ORDER BY Orders.OrderID
 GO

When * is used, the order of the columns in the result set is the same as the order in which they were specified in the CREATE TABLE, ALTER TABLE, or CREATE VIEW statements.

Because SELECT * finds all columns currently in a table, changes in the structure of a table (adding, removing, or renaming columns) are automatically reflected each time a SELECT * statement is executed.

If a SELECT is used in an application or script that has logic dependent on the number of columns in the result set, it is better to specify all the columns in the select list rather than specify an asterisk. If columns are later added to the table or views referenced by the SELECT statement, the application is shielded from the change if the columns were listed individually. If * was specified, the new columns become a part of the result set and may affect the logic of the application or script.

This example retrieves all columns in the **publishers** table and displays them in the order in which they were defined when the **publishers** table was created:

```
USE Northwind
GO
SELECT *
FROM [Order Details]
ORDER BY OrderID ASC
GO
```

To get exactly the same results, explicitly list all the column names in the table, in order, after the SELECT statement:

```
USE Northwind
GO
SELECT OrderID, ProductID, UnitPrice, Quantity, Discount
FROM [Order Details]
ORDER BY OrderID ASC
GO
```

Note To find out the column names for a table, use **sp_help**, use SELECT *column_name* FROM INFORMATION_SCHEMA.COLUMNS WHERE TABLE_NAME = *table*, or use SELECT TOP 0 * FROM *table*.

See Also

SELECT

Choosing Specific Columns

To select specific columns in a table, explicitly list each column in the select list. For example, to list only the author first names and their telephone numbers, use:

SELECT FirstName, HomePhone FROM Northwind.dbo.Employees ORDER BY FirstName ASC

Specifying the columns in the select list can also include specifying an alias (for example, **proj_sales** AS "Projected Sales") or other expressions, such as (**price** = **price** * 1.15, or SUM(**SalesAmount**).

See Also

SELECT

Constants in Query Result Sets

Constants are not usually specified as a separate column in a result set. It is usually more efficient for an application itself to build the constant value into the results when they are displayed, rather than requiring the server to incorporate the constant value in every result set row returned across the network.

Exceptions to this general rule include:

- Stored procedures may be called by many different applications or scripts. These procedures do not have access to the constant value that should be incorporated in the results. The SELECT statement in the procedure itself should then specify the constant as part of the select list.
- When a site wants to enforce a formatting or display standard, the format can be built into a view or stored procedure.
- A SELECT statement may be executed from a script or a tool that does not support merging constants with a result set after the result set has been returned from the server.

Character string constants are included for proper formatting or readability when character columns are concatenated. This example combines the **LastName** and **FirstName** columns into a single column. The character string ', ' separates the two parts of the name in the new combined column:

SELECT LastName + ', ' + FirstName AS EmployeeName FROM Northwind.dbo.Employees ORDER BY LastName, FirstName ASC

See Also

```
<u>+ (Add)</u>
```

String Concatenation Operator

<u>SELECT</u>

Computed Values in the Select List

A select list can contain expressions that are built by applying operators to one or more simple expressions. This allows result sets to contain values that do not exist in the base tables, but are calculated from the values stored in the base tables. These result set columns are called derived columns, and include:

- Calculations and computations that use arithmetic operators or functions on numeric columns or constants: SELECT ROUND((UnitPrice * .9), 2) AS DiscountPrice FROM Products WHERE ProductID = 58
- Data type conversions: SELECT (CAST(ProductID AS VARCHAR(10)) + ': ' + ProductName) AS ProductIDName FROM Products

 CASE functions: SELECT ProductID, ProductName, CASE CategoryID WHEN 1 THEN ROUND((UnitPrice * .6), 2) WHEN 2 THEN ROUND((UnitPrice * .7), 2) WHEN 3 THEN ROUND((UnitPrice * .8), 2) ELSE ROUND((UnitPrice * .9), 2) END AS DiscountPrice FROM Products

• Subqueries:

SELECT Prd.ProductID, Prd.ProductName,

- (SELECT SUM(OD.UnitPrice * OD.Quantity)
 FROM Northwind.dbo.[Order Details] AS OD
 WHERE OD.ProductID = Prd.ProductID
-) AS SumOfSales

FROM Northwind.dbo.Products AS Prd ORDER BY Prd.ProductID

Calculations and computations can be performed with data by using numeric columns or numeric constants in a select list with arithmetic operators, functions, conversions, or nested queries. Arithmetic operators let you add, subtract, multiply, and divide numeric data.

Symbol	Operation
+	Addition
-	Subtraction
/	Division
*	Multiplication
%	Modulo

The following arithmetic operators are supported.

The arithmetic operators that perform addition, subtraction, division, and multiplication can be used on any numeric column or expression (**int**, **smallint**, **tinyint**, **decimal**, **numeric**, **float**, **real**, **money**, or **smallmoney**). The modulo operator can only be used on **int**, **smallint**, or **tinyint** columns or expressions.

Arithmetic operations can also be performed on **datetime** and **smalldatetime** columns using the date functions or regular addition or subtraction arithmetic operators.

You can use arithmetic operators to perform computations involving one or more columns. The use of constants in arithmetic expressions is optional, as shown in this example:

SELECT ProductID, ProductName, UnitPrice * UnitsInStock AS InventoryValue FROM Northwind.dbo.Products

```
See Also
FROM
SELECT
```

Operators

Join Fundamentals

Subquery Fundamentals

<u>+ (Add)</u>

<u>- (Subtract)</u>

Functions

Assigning Result Set Column Names

The AS clause can be used either to change the name of a result set column or assign a name to a derived column.

When a result set column is defined by a reference to a column in a table or view, the name of the result set column is the same as the name of the referenced column. The AS clause can be used to assign a different name, or alias, to the result set column. This can be done to increase readability. For example:

SELECT EmpSSN AS "Employee Social Security Number" FROM EmpTable

Derived columns are those columns in the select list that are specified as something other than a simple reference to a column. Derived columns have no name unless the AS clause is used to assign a name. In this example, the derived column specified using the DATEDIFF function would have no name if the AS clause were removed:

SELECT OrderID,

DATEDIFF(dd, ShippedDate, GETDATE()) AS DaysSinceShippe FROM Northwind.dbo.Orders WHERE ShippedDate IS NOT NULL

The AS clause is the syntax defined in the SQL-92 standard for assigning a name to a result set column. This is the preferred syntax to use in Microsoft® SQL ServerTM.

column_name AS column_alias

Or

result_column_expression AS derived_column_name

Transact-SQL also supports the following syntax for compatibility with earlier versions of SQL Server:

```
column_alias = column_name
```

derived_column_name = result_column_expression

For example, the last sample can be coded as:

SELECT OrderID,

DaysSinceShipped = DATEDIFF(dd, ShippedDate, GETDATE()) FROM Northwind.dbo.Orders WHERE ShippedDate IS NOT NULL

See Also

<u>SELECT</u>

Or

Delimiting Result Set Column Names

The name of a result set column is an identifier. If the name is a regular identifier that follows the rules for identifiers, it does not have to be delimited. If the name does not follow the rules for identifiers it must be delimited using either brackets ([]) or double quotation marks (""). Double quotation marks can be used to delimit result set column names, regardless of the setting of the QUOTED_IDENTIFIER option.

Note A name of up to 128 characters can be supplied for a result set column name. However, DB-Library applications, such as the **isql** utility, truncate the name of any result set column to 30 characters in the query output. The SQL Server ODBC drivers from SQL Server version 6.5 or earlier also truncate the result set column names to 30 characters.

This example retrieves the publisher name from the **publishers** table with a column heading of **Book Publisher** rather than the default column heading of **pub_name**:

USE pubs SELECT pub_name AS "Book Publisher" FROM publishers ORDER BY pub_name ASC

In addition, Transact-SQL reserved keywords can be used in quoted column headings. For example, this query uses the reserved word SUM as a column heading:

USE pubs SELECT SUM(ytd_sales) AS "sum" FROM titles

Transact-SQL also supports using single quotation marks (") to delimit a result set column name. This allows compatibility with earlier versions of SQL Server:

USE pubs SELECT SUM(ytd_sales) AS 'sum' FROM titles

See Also

Using Identifiers

<u>SELECT</u>

Eliminating Duplicates with DISTINCT

The DISTINCT keyword eliminates duplicate rows from the results of a SELECT statement. If DISTINCT is not specified, all rows are returned, including duplicates. For example, if you select all the author IDs in **titleauthor** without DISTINCT, the following rows are returned (with some duplicate listings):

USE pubs SELECT au_id FROM titleauthor

Here is the result set:

au_id

_____ 172-32-1176 213-46-8915 213-46-8915 238-95-7766 267-41-2394 267-41-2394 274-80-9391 409-56-7008 427-17-2319 472-27-2349 486-29-1786 486-29-1786 648-92-1872 672-71-3249 712-45-1867 722-51-5454 724-80-9391 724-80-9391

756-30-7391 807-91-6654 846-92-7186 899-46-2035 899-46-2035 998-72-3567

(25 row(s) affected)

With DISTINCT, you can eliminate duplicates and see only the unique author IDs:

USE pubs SELECT DISTINCT au_id FROM titleauthor

Here is the result set:

au_id

172-32-1176 213-46-8915 238-95-7766 267-41-2394 274-80-9391 409-56-7008 427-17-2319 472-27-2349 486-29-1786 648-92-1872 672-71-3249 712-45-1867 722-51-5454 724-80-9391 756-30-7391 807-91-6654 846-92-7186 899-46-2035 998-72-3567

(19 row(s) affected)

IMPORTANT The output for statements involving DISTINCT depends on the collation of the column or expression on which the DISTINCT is applied. For more information about the effects of different collations, see <u>SQL Server</u> <u>Collation Fundamentals</u>.

For the DISTINCT keyword, null values are considered to be duplicates of each other. When DISTINCT is included in a SELECT statement, only one NULL is returned in the results, regardless of how many null values are encountered.

Note For compatibility with the SQL-92 standard and other implementations of Microsoft® SQL Server[™], the ALL keyword can explicitly ask for all rows. However, there is no need to specify ALL because it is the default.

Limiting Result Sets Using TOP and PERCENT

The TOP clause limits the number of rows returned in the result set.

TOP n [PERCENT]

n specifies how many rows are returned. If PERCENT is not specified, *n* is the number of rows to return. If PERCENT is specified, *n* is the percentage of the result set rows to return:

TOP 120 /*Return the top 120 rows of the result set. */ TOP 15 PERCENT /* Return the top 15% of the result set. */.

If a SELECT statement that includes TOP also has an ORDER BY clause, the rows to be returned are selected from the ordered result set. The entire result set is built in the specified order and the top *n* rows in the ordered result set are returned.

The other method of limiting the size of a result set is to execute a SET ROWCOUNT *n* statement before executing a statement. SET ROWCOUNT differs from TOP in these ways:

- The SET ROWCOUNT limit applies to building the rows in the result set after an ORDER BY is evaluated. When ORDER BY is specified, the SELECT statement is terminated when *n* rows have been selected from a set of values that has been sorted according to specified ORDER BY classification.
- The TOP clause applies to the single SELECT statement in which it is specified. SET ROWCOUNT remains in effect until another SET ROWCOUNT statement is executed, such as SET ROWCOUNT 0 to turn the option off.

See Also

<u>SELECT</u>

SET ROWCOUNT

Using the FROM Clause

The FROM clause is required in every SELECT statement in which data is being retrieved from tables or views. Use the FROM clause to:

- List the tables and views containing the columns referenced in the select list and in the WHERE clause. The table or view names can be aliased using the AS clause.
- Join types. These are qualified by join conditions specified in the ON clause.

The FROM clause is a comma-separated list of table names, view names, and JOIN clauses.

Transact-SQL has extensions that support the specification of objects other than tables or views in the FROM clause. These other objects return a result set, or rowset in OLE DB terms, that form a virtual table. The SELECT statement then operates as if the result set were a table.

The FROM clause can specify:

- One or more tables or views. For example: SELECT * FROM Shippers
- Joins between two tables or views: SELECT Cst.CustomerID, Cst.CompanyName, Cst.ContactNai Ord.ShippedDate, Ord.Freight
 FROM Northwind.dbo.Orders AS Ord JOIN Northwind.dbo.Customers AS Cst ON (Cst.CustomerID = Ord.CustomerID)
- One or more derived tables, which are SELECT statements in the FROM clause referred to by an alias or a user-specified name. The

```
result set of the SELECT in the FROM clause forms a table used by the
outer SELECT statement. For example, this SELECT uses a derived
table to find if any store carries all book titles in the pubs database:
SELECT ST.stor_id, ST.stor_name
FROM stores AS ST,
(SELECT stor_id, COUNT(DISTINCT title_id) AS title_co
FROM sales
GROUP BY stor_id
) AS SA
WHERE ST.stor_id = SA.stor_id
AND SA.title_count = (SELECT COUNT(*) FROM titles)
```

- One or more tables or views from a linked server defined using **sp_addlinkedserver**. A linked server can be any OLE DB data source.
- An OLE DB rowset returned by either the OPENROWSET or OPENQUERY functions.

The basis of Microsoft® SQL Server[™] 2000 distributed queries are linked servers, OPENROWSET, and OPENQUERY. They provide the ability to query or modify data in any OLE DB data source as a part of Transact-SQL statements.

SELECT Statements Without FROM Clauses

The SELECT statements that do not require a FROM clause are those that are not selecting data from any tables in the database. These SELECT statements only select data from local variables or Transact-SQL functions that do not operate on a column, for example:

SELECT @MyIntVariable SELECT @@VERSION SELECT DB_ID('Northwind')

See Also

Distributed Queries

OPENQUERY

<u>FROM</u>

<u>Using Joins</u>

OPENROWSET

Using Table Aliases

The readability of a SELECT statement can be improved by giving a table an alias, also known as a correlation name or range variable. A table alias can be assigned either with or without the AS keyword:

- table_name AS table alias
- table_name table_alias

In this example, the alias **p** is assigned to **publishers**.

USE pubs SELECT p.pub_id, p.pub_name FROM publishers AS p

If an alias is assigned to a table, all explicit references to the table in the Transact-SQL statement must use the alias, not the table name. For example, the following SELECT generates a syntax error because it uses the name of the table when an alias has been assigned:

SELECT Customers.CustomerID, /* Illegal reference to Customers. */ Cst.FirstName, Cst.LastName FROM Northwind.dbo.Customers AS Cst

See Also

<u>FROM</u>

Filtering Rows with WHERE and HAVING

The WHERE and HAVING clauses in a SELECT statement control the rows from the source tables that are used to build the result set. WHERE and HAVING are filters. They specify a series of search conditions, and only those rows that meet the terms of the search conditions are used to build the result set. Those rows meeting the search conditions are said to be qualified to participate in the result set. For example, the WHERE clause in this SELECT statement qualifies the rows only where the region is Washington State:

```
SELECT CustomerID, CompanyName
FROM Northwind.dbo.Customers
WHERE Region = 'WA'
```

The HAVING clause is typically used in conjunction with the GROUP BY clause, although it can be specified without GROUP BY. The HAVING clause specifies further filters that are applied after the WHERE clause filters. For example, this WHERE clause only qualifies orders selling a product with a unit price exceeding \$100, and the HAVING clause further restricts the result to only thos orders that include more than 100 units:

```
SELECT OrdD1.OrderID AS OrderID,
SUM(OrdD1.Quantity) AS "Units Sold",
SUM(OrdD1.UnitPrice * OrdD1.Quantity) AS Revenue
FROM [Order Details] AS OrdD1
WHERE OrdD1.OrderID in (SELECT DISTINCT OrdD2.OrderID
FROM [Order Details] AS OrdD2
WHERE OrdD2.UnitPrice > $100)
GROUP BY OrdD1.OrderID
HAVING SUM(OrdD1.Quantity) > 100
```

The search conditions, or qualifications, in the WHERE and HAVING clauses can include:

• Comparison operators (such as =, < >, <, and >). For example, this query retrieves the rows from the **Products** table for the products that

are in product category 2: SELECT ProductID, ProductName FROM Northwind.dbo.Products WHERE CategoryID = 2 ORDER BY ProductID

- Ranges (BETWEEN and NOT BETWEEN). For example, this query retrieves rows from the **Products** table with categories from 2 to 4: SELECT CategoryID, ProductID, ProductName FROM Northwind.dbo.Products
 WHERE CategoryID BETWEEN 2 and 4
 ORDER BY CategoryID, ProductID
- Lists (IN, NOT IN). For example, this query retrieves rows from the **Products** table in which the Category ID matches one in a list of IDs: SELECT CategoryID, ProductID, ProductName FROM Northwind.dbo.Products WHERE CategoryID IN (1,4,5,7) ORDER BY CategoryID, ProductID
- Pattern matches (LIKE and NOT LIKE). For example, this query retrieves rows from the **Products** table in which the product name starts with the letters Ch: SELECT CategoryID, ProductID, ProductName FROM Northwind.dbo.Products
 WHERE ProductName LIKE 'Ch%' ORDER BY CategoryID, ProductID

Note The only WHERE conditions that you can use on **text** columns are functions that return another data type, such as PATINDEX(), or the operators, such as IS NULL, IS NOT NULL, LIKE, and NOT LIKE.

• Null values (IS NULL and IS NOT NULL). For example, this query retrieves rows from the **Customers** table in which the customers' region is not NULL:

SELECT CompanyName, City, Region, Country FROM Northwind.dbo.Customers WHERE Region IS NOT NULL ORDER BY CompanyName

Note Use caution when comparing null values. For example, specifying = NULL is not the same as specifying IS NULL. For more information, see <u>Null Values</u>.

 All records (=ALL, >ALL, <= ALL, ANY). For example, this query retrieves order and product IDs from the Order Details table in which the quantity of the product shipped is larger than the quantity shipped for any product in category 1: USE Northwind GO SELECT OrdD1.OrderID, OrdD1.ProductID FROM "Order Details" OrdD1 WHERE OrdD1.Quantity > ALL (SELECT OrdD2.Quantity

```
FROM "Order Details" OrdD2 JOIN Products Prd
ON OrdD2.ProductID = Prd.ProductID
```

```
WHERE Prd.CategoryID = 1)
```

```
GO
```

Combinations of these conditions (AND, OR, NOT). For example, this query retrieves all products for which either the stock level is lower than the reorder point or the product comes from supplier 15 and is in category 4:
 SELECT ProductID, ProductName FROM Northwind.dbo.Products
 WHERE UnitsInStock < ReorderLevel

```
OR (SupplierID = 15 AND CategoryID = 4)
```

Note When you search for a Unicode string in a WHERE clause, place the N character before the search string, for example:

SELECT CompanyName, ContactName, Phone, Fax FROM Northwind.dbo.Customers WHERE CompanyName = N'Berglunds snabbköp'

See Also

IS [NOT] NULL

<u>Operators</u>

ISNULL

WHERE

Comparison Search Conditions

Microsoft[®] SQL Server[™] 2000 uses these comparison operators.

Operator	Meaning
=	Equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
<>	Not equal to (SQL-92 compatible)
!>	Not greater than
!<	Not less than
!=	Not equal to

Comparison operators are specified between two <u>expressions</u>. For example, to retrieve the names of only those products for which the unit price is greater than \$50, use:

SELECT ProductName FROM Northwind.dbo.Products WHERE UnitPrice > \$50.00

When you compare character string data, the logical sequence of the characters is defined by the collation of the character data. The result of comparison operators such as < and > are controlled by the character sequence defined by the collation. The same SQL Collation might have different sorting behavior for Unicode and non-Unicode data. (For more information, see <u>SQL Server</u> <u>Collation Fundamentals</u>.)

Trailing blanks are ignored in comparisons in non-Unicode data; for example, these are equivalent:

WHERE au_lname = 'White' WHERE au_lname = 'White ' WHERE au_lname = 'White' + SPACE(1) The use of NOT negates an expression. For example, this query finds all products that have a unit price of \$50 or more, which is logically the same as asking for all products that do not have a unit price of less than \$50:

SELECT ProductID, ProductName, UnitPrice FROM Northwind.dbo.Products WHERE NOT UnitPrice < \$50 ORDER BY ProductID

See Also

String Concatenation Operator

+ (String Concatentation)

Data Types

Operators

SQL Server Collation Fundamentals

Range Search Conditions

A range search is one that returns all values between two specified values. Inclusive ranges return any values that match the two specified values. Exclusive ranges do not return any values that match the two specified values.

The BETWEEN keyword specifies an inclusive range to search. For example, this SELECT returns all products whose units in stock is between 15 and 25:

SELECT UnitsInStock, ProductID, ProductName FROM Northwind.dbo.Products WHERE UnitsInStock BETWEEN 15 AND 25 ORDER BY UnitsInStock

The results of this SELECT statement contains any products that have either 15 or 25 units in stock.

SELECT UnitsInStock, ProductID, ProductName FROM Northwind.dbo.Products WHERE UnitsInStock = 15 OR UnitsInStock = 25 ORDER BY UnitsInStock

To specify an exclusive range, use the greater-than and less-than operators (> and <). The following query using the greater-than and less-than operators returns different results than the last example because these operators do not include rows matching the values that limit the range.

SELECT UnitsInStock, ProductID, ProductName FROM Northwind.dbo.Products WHERE UnitsInStock > 15 AND UnitsInStock < 25 ORDER BY UnitsInStock

NOT BETWEEN finds all rows outside the range you specify. Use this query to find all products for which the number of units in stock are outside the 15 to 25 range:

SELECT UnitsInStock, ProductID, ProductName FROM Northwind.dbo.Products WHERE UnitsInStock NOT BETWEEN 15 AND 25 ORDER BY UnitsInStock

See Also

WHERE

Data Types

List Search Conditions

The IN keyword allows you to select rows that match any one of a list of values. For example, without IN, if you want a list of the names and states of all authors who live in California, Indiana, or Maryland, you would need this query:

SELECT ProductID, ProductName FROM Northwind.dbo.Products WHERE CategoryID = 1 OR CategoryID = 4 OR CategoryID = 5

However, you can get the same results with less typing if you use IN:

SELECT ProductID, ProductName FROM Northwind.dbo.Products WHERE CategoryID IN (1, 4, 5)

The items following the IN keyword must be separated by commas and be enclosed in parentheses.

Perhaps the most important use for the IN keyword is in nested queries, also referred to as subqueries. For more information about subqueries, see <u>Subquery</u> <u>Fundamentals</u>.

This query finds all **au_id**s in the **titleauthor** table for authors who make less than 50 percent of the royalty on any one book, and then selects from the **authors** table all author names with **au_id**s that match the results from the **titleauthor** query:

USE pubs SELECT au_lname, au_fname FROM authors WHERE au_id IN (SELECT au_id FROM titleauthor WHERE royaltyper < 50) The results show that several authors fall into the less than 50 percent category.

This query finds the names of authors who do not make less than 50 percent of the royalties on at least one book:

```
USE pubs
SELECT au_lname, au_fname
FROM authors
WHERE au_id NOT IN
(SELECT au_id
FROM titleauthor
WHERE royaltyper < 50)
```

NOT IN finds the authors who do not match the items in the values list.

See Also

WHERE

Data Types

Pattern Matching in Search Conditions

The LIKE keyword searches for character string, date, or time values that match a specified pattern. For more information, see <u>Data Types</u>. The LIKE keyword uses a regular expression to contain the pattern that the values are matched against. The pattern contains the character string to search for, which can contain any combination of four wildcards.

Wildcard	Meaning
%	Any string of zero or more characters.
_	Any single character.
[]	Any single character within the specified range (for example, [a-f]) or set (for example, [abcdef]).
[^]	Any single character not within the specified range (for example, [^a - f]) or set (for example, [^abcdef]).

Enclose the wildcard(s) and the character string in single quotation marks, for example:

- LIKE 'Mc%' searches for all strings that begin with the letters Mc (McBadden).
- LIKE '%inger' searches for all strings that end with the letters inger (Ringer, Stringer).
- LIKE '%en%' searches for all strings that contain the letters en anywhere in the string (Bennet, Green, McBadden).
- LIKE '_heryl' searches for all six-letter names ending with the letters heryl (Cheryl, Sheryl).
- LIKE '[CK]ars[eo]n' searches for Carsen, Karsen, Carson, and Karson (Carson).

- LIKE '[M-Z]inger' searches for all names ending with the letters inger that begin with any single letter from M through Z (Ringer).
- LIKE 'M[^c]%' searches for all names beginning with the letter M that do not have the letter c as the second letter (MacFeather).

This query finds all phone numbers in the **authors** table that have area code 415:

SELECT phone FROM pubs.dbo.authors WHERE phone LIKE '415%'

You can use NOT LIKE with the same wildcards. To find all phone numbers in the **authors** table that have area codes other than 415, use either of these equivalent queries:

SELECT phone FROM pubs.dbo.authors WHERE phone NOT LIKE '415%'

-- Or

SELECT phone FROM pubs.dbo.authors WHERE NOT phone LIKE '415%'

The IS NOT NULL clause can be used with wildcards and the LIKE clause. For example, this query retrieves telephone numbers from the **authors** table in which the telephone number begins with 415 and IS NOT NULL:

USE pubs SELECT phone FROM authors WHERE phone LIKE '415%' and phone IS NOT NULL

IMPORTANT The output for statements involving the LIKE keyword depends on

the sort order chosen during installation. For more information about the effects of different sort orders, see <u>Collations</u>.

The only WHERE conditions that you can use on **text** columns are LIKE, IS NULL, or PATINDEX.

Wildcards used without LIKE are interpreted as constants rather than as a pattern, that is, they represent only their own values. The following query attempts to find any phone numbers that consist of the four characters 415% only. It will not find phone numbers that start with 415. For more information about constants, see <u>Using Constants</u>.

SELECT phone FROM pubs.dbo.authors WHERE phone = '415%'

Another important consideration in using wildcards is their effect on performance. If a wildcard begins the expression, an index cannot be used. (Just as you wouldn't know where to start in a phone book if given the name '%mith', not 'Smith'.) A wildcard in or at the end of an expression does not preclude use of an index (just as in a phone book, you would know where to search if the name was 'Samuel%', regardless of whether the names Samuels and Samuelson are both there).

Searching for Wildcard Characters

You can search for wildcard characters. There are two methods for specifying a character that would ordinarily be a wildcard:

Use the ESCAPE keyword to define an escape character. When the escape character is placed in front of the wildcard in the pattern, the wildcard is interpreted as a character. For example, to search for the string 5% anywhere in a string, use:
 WHERE ColumnA LIKE '%5/%%' ESCAPE '/'

In this LIKE clause, the leading and ending percent signs (%) are interpreted as wildcards, and the percent sign preceded by a slash (/) is interpreted as the % character.

• Use square brackets ([]) to enclose the wildcard by itself. To search for a dash (-), rather than using it to specify a search range, use the dash as the first character inside a set of brackets: WHERE ColumnA LIKE '9[-]5'

Symbol	Meaning
LIKE '5[%]'	5%
LIKE '5%'	5 followed by any string of 0 or more characters
LIKE '[_]n'	_n
LIKE '_n'	an, in, on (and so on)
LIKE '[a-cdf]'	a, b, c, d, or f
LIKE '[-acdf]'	-, a, c, d, or f
LIKE '[[]'	[
LIKE ']']

The table shows the use of wildcards enclosed in square brackets.

When string comparisons are performed with LIKE, all characters in the pattern string are significant, including every leading and/or trailing blank (space). If a comparison to return all rows with a string LIKE 'abc ' (abc followed by a single space) is requested, a row in which the value of that column is abc (abc without a space) is not returned. The reverse, however, is not true. Trailing blanks in the expression to which the pattern is matched are ignored. If a comparison to return all rows with a string LIKE 'abc' (abc without a space) is requested, all rows that start with abc and have zero or more trailing blanks are returned.

See Also

<u>LIKE</u>

WHERE

NULL Comparison Search Conditions

The value NULL means the data value for the column is unknown or not available. NULL is not synonymous with zero (numeric or binary value), a zerolength string, or blank (character value). Rather, null values allow you to distinguish between an entry of zero (numeric columns) or blank (character columns) and a nonentry (NULL for both numeric and character columns).

NULL can be entered in a column for which null values are permitted (as specified in the CREATE TABLE statement) in two ways:

- Microsoft[®] SQL Server[™] 2000 automatically enters the value NULL if no data is entered and there is no default or DEFAULT constraint on the column or data type.
- The user can explicitly enter the value NULL by typing NULL without quotation marks. If the word NULL is typed into a character column with quotation marks, it is treated as the letters N, U, L, and L, not as a null value.

When null values are retrieved, an application typically displays a string such as NULL, or (NULL), or (null) in the appropriate position. For example, the **advance** column of the **titles** table allows null values:

SELECT title_id, type, advance FROM pubs.dbo.titles WHERE advance IS NULL

Here is the result set:

title_id type advance

MC3026 UNDECIDED (null) PC9999 popular_comp (null)

(2 row(s) affected)

Comparing Null Values

Care must be taken when comparing null values. The behavior of the comparison depends on the setting of the SET ANSI_NULLS option.

When SET ANSI_NULLS is ON, a comparison in which one or more of the expressions is NULL does not yield either TRUE or FALSE; it yields UNKNOWN. This is because a value that is unknown cannot be compared logically against any other value. This occurs if either an expression is compared to the literal NULL, or if two expressions are compared and one of them evaluates to NULL. For example, this comparison always yields UNKNOWN when ANSI_NULLS is ON:

ytd_sales > NULL

This comparison also yields UNKNOWN any time the variable contains the value NULL:

ytd_sales > @MyVariable

Use the IS NULL or IS NOT NULL clauses to test for a NULL value. This can add complexity to the WHERE clause. For example, the **Region** column in the **Northwind Customers** table allows null values. If a SELECT statement is to test for null values in addition to others, it must include an IS NULL clause:

SELECT CustomerID, CompanyName, Region FROM Northwind.dbo.Customers WHERE Region IN ('WA', 'SP', 'BC') OR Region IS NULL

Transact-SQL supports an extension that allows for the comparison operators to return TRUE or FALSE when comparing against null values. This option is activated by setting ANSI_NULLS OFF. When ANSI_NULLS is OFF, comparisons such as **ColumnA** = NULL return TRUE when **ColumnA** contains a null value and FALSE when **ColumnA** contains some value besides NULL. Also, a comparison of two expressions that have both evaluated to null values yields TRUE. With ANSI_NULLS set OFF, this SELECT statement returns all the rows in the **Customer** table for which **Region** is a null value:

SELECT CustomerID, CompanyName, Region FROM Northwind.dbo.Customers WHERE Region = NULL

Regardless of the ANSI_NULLS setting, Null values are always considered equal for the purposes of the ORDER BY, GROUP BY, and DISTINCT keywords. Also, a unique index or UNIQUE constraint that allows NULL can contain only one row with a NULL key value. A subsequent row with NULL is rejected. A primary key cannot have NULL in any column that is part of the key.

Computations involving NULL evaluate to NULL because the result must be UNKNOWN if any of the factors is unknown. For example, *column1* + 1 evaluates to NULL if **column1** is NULL.

When the columns being searched include those defined as allowing null values, you can find null or nonnull values in the database with this pattern:

WHERE column_name IS [NOT] NULL

See Also

Null Values

IS [NOT] NULL

Logical Operators

The logical operators are AND, OR, and NOT. AND and OR are used to connect search conditions in WHERE clauses. NOT reverses the result of a search condition.

AND joins two conditions and returns TRUE only when both conditions are true. For example, this query returns only the one row in which the customer ID starts with the letter F and the contact name is Lino Rodriguez:

SELECT CustomerID, CompanyName, ContactName FROM Northwind.dbo.Customers WHERE CustomerID LIKE N'F%' AND ContactName = N'Lino Rodriguez'

OR also connects two conditions, but it returns TRUE when either of the conditions is true. The following query returns two rows, one with a customer ID of CACTU and the other with a contact name of Lino Rodriguez:

SELECT CustomerID, CompanyName, ContactName FROM Customers WHERE CustomerID = N'CACTU' OR ContactName = N'Lino Rodriguez'

See Also

WHERE

Operators

Logical Operator Precedence

When more than one logical operator is used in a statement, NOT is evaluated first, then AND, and finally OR. Arithmetic (and bitwise) operators are handled before logical operators.

In this example, the advance condition pertains to psychology books and not to business books because AND has precedence over OR:

```
SELECT title_id, type, advance
FROM pubs.dbo.titles
WHERE type = 'business' OR type = 'psychology'
AND advance > $5500
```

You can change the meaning of the query by adding parentheses to force evaluation of the OR first. This query finds all business and psychology books that have advances over \$5,500:

```
SELECT title_id, type, advance
FROM titles
WHERE (type = 'business' OR type = 'psychology')
AND advance > $5500
```

The use of parentheses, even when not required, can improve the readability of queries and reduce the chance of making a subtle mistake because of operator precedence. There is no significant performance penalty in using parentheses. This example is more readable than the original example, although they are syntactically the same:

```
SELECT title_id, type, advance
FROM pubs.dbo.titles
WHERE type = 'business'
OR (type = 'psychology' AND advance > $5500)
```

See Also

<u>Operators</u>

<u>WHERE</u>

Transact-SQL Joins

In earlier versions of Microsoft® SQL Server[™] 2000, left and right outer join conditions were specified in the WHERE clause using the *= and =* operators. In some cases, this syntax results in an ambiguous query that can be interpreted in more than one way. SQL-92 compliant outer joins are specified in the FROM clause and do not result in this ambiguity. Because the SQL-92 syntax is more precise, detailed information about using the old Transact-SQL outer join syntax in the WHERE clause is not included with this release. The syntax may not be supported in a future version of SQL Server. Any statements using the Transact-SQL outer joins should be changed to use the SQL-92 syntax.

The SQL-92 standard does support the specification of inner joins in either the FROM or WHERE clause. Inner joins specified in the WHERE clause do not have the same problems with ambiguity as the Transact-SQL outer join syntax.

See Also

<u>FROM</u> Join Fundamentals SELECT

Sorting Rows with ORDER BY

The ORDER BY clause sorts query results by one or more columns up to 8,060 bytes. For more information about the maximum ORDER BY clause size, see <u>SELECT</u>.

A sort can be ascending (ASC) or descending (DESC). If neither is specified, ASC is assumed.

IMPORTANT The exact results of an ORDER BY clause depend on the collation chosen during installation. For information about the effects of different collations, see <u>SQL Server Collation Fundamentals</u>.

This query returns results ordered by ascending **pub_id**:

USE pubs SELECT pub_id, type, title_id FROM titles ORDER BY pub_id

Here is the result set:

pub_id type title_id __ ____ 0736 business BU2075 0736 psychology PS2091 0736 psychology PS2106 0736 psychology PS3333 0736 psychology PS7777 0877 mod_cook MC2222 0877 mod_cook MC3021 0877 UNDECIDED MC3026 0877 psychology PS1372 0877 trad_cook **TC3218** 0877 trad cook TC4203 0877 trad cook TC7777

```
1389 business BU1032
1389 business BU1111
1389 business BU7832
1389 popular_comp PC1035
1389 popular_comp PC8888
1389 popular_comp PC9999
```

(18 row(s) affected)

If more than one column is named in the ORDER BY clause, sorts are nested. The following statement sorts the rows in the **titles** table, first by publisher in descending order, and then by type in ascending order within each publisher, and finally by price (also ascending, because DESC is not specified).

USE pubs SELECT pub_id, type, title_id, price FROM titles ORDER BY pub_id DESC, type, price

Note You cannot use ORDER BY on columns that have the **text** or **image** data types. Also, subqueries, aggregates, and constant expressions are not allowed in the ORDER BY list; however, a user-specified name can be used in the select list for aggregates or expressions, for example:

SELECT type, sum (ytd_sales) AS sales_total FROM titles GROUP BY type ORDER BY sales_total

Join Fundamentals

By using joins, you can retrieve data from two or more tables based on logical relationships between the tables. Joins indicate how Microsoft® SQL Server[™] 2000 should use data from one table to select the rows in another table.

A join condition defines the way two tables are related in a query by:

- Specifying the column from each table to be used for the join. A typical join condition specifies a foreign key from one table and its associated key in the other table.
- Specifying a logical operator (=, <>, and so on) to be used in comparing values from the columns.

Joins can be specified in either the FROM or WHERE clauses. The join conditions combine with the WHERE and HAVING search conditions to control the rows that are selected from the base tables referenced in the FROM clause.

Specifying the join conditions in the FROM clause helps separate them from any other search conditions that may be specified in a WHERE clause, and is the recommended method for specifying joins. A simplified SQL-92 FROM clause join syntax is:

FROM first_table join_type second_table [ON (join_condition)]

join_type specifies what kind of join is performed: an inner, outer, or cross join. *join_condition* defines the predicate to be evaluated for each pair of joined rows. This is an example of a FROM clause join specification:

FROM Suppliers JOIN Products ON (Suppliers.SupplierID = Products.SupplierID)

This is a simple SELECT statement using this join:

SELECT ProductID, Suppliers.SupplierID, CompanyName

```
FROM Suppliers JOIN Products
ON (Suppliers.SupplierID = Products.SupplierID)
WHERE UnitPrice > $10
AND CompanyName LIKE N'F%'
GO
```

The select returns the product and supplier information for any combination of parts supplied by a company for which the company name starts with the letter F and the price of the product is more than \$10.

When multiple tables are referenced in a single query, all column references must be unambiguous. In the previous example, both the **Products** and **Suppliers** table have a column named **SupplierID**. Any column name that is duplicated between two or more tables referenced in the query must be qualified with the table name. All references to the **SupplierID** columns in the example are qualified.

When a column name is not duplicated in two or more tables used in the query, references to it do not have to be qualified with the table name. This is shown in the previous example. Such a SELECT statement is sometimes difficult to understand because there is nothing to indicate the table that provided each column. The readability of the query is improved if all columns are qualified with their table names. The readability is further improved if table aliases are used, especially when the table names themselves must be qualified with the database and owner names. This is the same example, except that table aliases have been assigned and the columns qualified with table aliases to improve readability:

SELECT P.ProductID,

S.SupplierID, S.CompanyName FROM Suppliers AS S JOIN Products AS P ON (S.SupplierID = P.SupplierID) WHERE P.UnitPrice > \$10 AND S.CompanyName LIKE N'F%'

The previous examples specified the join conditions in the FROM clause, which

is the preferred method. This query contains the same join condition specified in the WHERE clause:

SELECT P.ProductID, S.SupplierID, S.CompanyName FROM Suppliers AS S, Products AS P WHERE S.SupplierID = P.SupplierID AND P.UnitPrice > \$10 AND S.CompanyName LIKE N'F%'

The select list for a join can reference all the columns in the joined tables, or any subset of the columns. The select list is not required to contain columns from every table in the join. For example, in a three-table join, only one table can be used to bridge from one of the other tables to the third table, and none of the columns from the middle table have to be referenced in the select list.

Although join conditions usually have equality comparisons (=), other comparison or relational operators can be specified, as can other predicates. For more information, see <u>Using Operators in Expressions</u> and <u>WHERE</u>.

When SQL Server processes joins, the query engine chooses the most efficient method (out of several possibilities) of processing the join. Although the physical execution of various joins uses many different optimizations, the logical sequence is:

- The join conditions in the FROM clause are applied.
- The join conditions and search conditions from the WHERE clause are applied.
- The search conditions from the HAVING clause are applied.

This sequence can sometimes influence the results of the query if conditions are moved between the FROM and WHERE clauses.

Columns used in a join condition are not required to have the same name or be the same data type. However, if the data types are not identical, they must be compatible, or be types that SQL Server can implicitly convert. If the data types cannot be implicitly converted, the join condition must explicitly convert the data type using the CAST function. For more information about implicit and explicit conversions, see <u>Data Type Conversion</u>.

Most queries using a join can be rewritten using a subquery (a query nested within another query), and most subqueries can be rewritten as joins. For more information about subqueries, see <u>Subquery Fundamentals</u>.

Note Tables cannot be joined directly on **ntext**, **text**, or **image** columns. However, tables can be joined indirectly on **ntext**, **text**, or **image** columns by using SUBSTRING. For example, SELECT * FROM t1 JOIN t2 ON SUBSTRING(t1.*textcolumn*, 1, 20) = SUBSTRING(t2.*textcolumn*, 1, 20) performs a two-table inner join on the first 20 characters of each text column in tables **t1** and **t2**. In addition, another possibility for comparing **ntext** or **text** columns from two tables is to compare the lengths of the columns with a WHERE clause, for example (where a self-join is performed on the **pub_info** table):

WHERE DATALENGTH(p1.pr_info) = DATALENGTH(p2.pr_info)

See Also

Logical Operator Precedence

SELECT Examples

Using Joins

Join conditions can be specified in either the FROM or WHERE clauses; specifying them in the FROM clause is recommended. WHERE and HAVING clauses can also contain search conditions to further filter the rows selected by the join conditions.

Joins can be categorized as:

• Inner joins (the typical join operation, which uses some comparison operator like = or <>). These include equi-joins and natural joins.

Inner joins use a comparison operator to match rows from two tables based on the values in common columns from each table. For example, retrieving all rows where the student identification number is the same in both the **students** and **courses** tables.

• Outer joins. Outer joins can be a left, a right, or full outer join.

Outer joins are specified with one of the following sets of keywords when they are specified in the FROM clause:

• LEFT JOIN or LEFT OUTER JOIN

The result set of a left outer join includes all the rows from the left table specified in the LEFT OUTER clause, not just the ones in which the joined columns match. When a row in the left table has no matching rows in the right table, the associated result set row contains null values for all select list columns coming from the right table.

• RIGHT JOIN or RIGHT OUTER JOIN.

A right outer join is the reverse of a left outer join. All rows from the right table are returned. Null values are returned for the left table any time a right table row has no matching row in the left table.

• FULL JOIN or FULL OUTER JOIN.

A full outer join returns all rows in both the left and right

tables. Any time a row has no match in the other table, the select list columns from the other table contain null values. When there is a match between the tables, the entire result set row contains data values from the base tables.

• Cross joins.

Cross joins return all rows from the left table, each row from the left table is combined with all rows from the right table. Cross joins are also called Cartesian products.

For example, here is an inner join retrieving the authors who live in the same city and state as a publisher:

```
USE pubs
SELECT a.au_fname, a.au_lname, p.pub_name
FROM authors AS a INNER JOIN publishers AS p
ON a.city = p.city
AND a.state = p.state
ORDER BY a.au_lname ASC, a.au_fname ASC
```

The tables or views in the FROM clause can be specified in any order with an inner join or full outer join; however, the order of tables or views specified when using either a left or right outer join is important. For more information about table ordering with left or right outer joins, see <u>Using Outer Joins</u>.

See Also

<u>Operators</u>

CAST and CONVERT

Using Operators in Expressions

<u>SELECT</u>

Using Inner Joins

An inner join is a join in which the values in the columns being joined are compared using a comparison operator.

In the SQL-92 standard, inner joins can be specified in either the FROM or WHERE clause. This is the only type of join that SQL-92 supports in the WHERE clause. Inner joins specified in the WHERE clause are known as old-style inner joins.

This Transact-SQL query is an example of an inner join:

```
USE pubs
SELECT *
FROM authors AS a INNER JOIN publishers AS p
ON a.city = p.city
ORDER BY a.au_lname DESC
```

This inner join is known as an equi-join. It returns all the columns in both tables, and returns only the rows for which there is an equal value in the join column.

Here is the result set:

au_lname au_fname phone address au id city -----238-95-7766 Carson Cheryl 415 548-7723 589 Darwin Ln. Berk 409-56-7008 Bennet Abraham 415 658-9932 6223 Bateman St. Be state zip contract pub_id pub_name city state country CA 94705 1 1389 Algodata Infosystems Berkeley CA USA 1389 Algodata Infosystems Berkeley CA CA 94705 1 USA

(2 row(s) affected)

In the result set, the **city** column appears twice. Because there is no point in repeating the same information, one of these two identical columns can be eliminated by changing the select list. The result is called a natural join. You can restate the preceding Transact-SQL query to form a natural join. For example:

```
USE pubs
SELECT p.pub_id, p.pub_name, p.state, a.*
FROM publishers p INNER JOIN authors a
ON p.city = a.city
ORDER BY a.au_lname ASC, a.au_fname ASC
```

Here is the result set:

pub_id pub_name	state au_	_id au_lr	name au_fna	ame
1389 Algodata Infosyste			1389 6 Carson	0
phone address	city state	e zip contra	ct	
415 658-9932 6223 Bater 415 548-7723 589 Darwi		0		

(2 row(s) affected)

In this example, **publishers.city** does not appear in the results.

Joins Using Operators Other Than Equal

You can also join values in two columns that are not equal. The same operators and predicates used for inner joins can be used for not-equal joins. For more information about the available operators and predicates that can be used in joins, see <u>Using Operators in Expressions</u> and <u>WHERE</u>.

This Transact-SQL example is of a greater-than (>) join which finds New Moon authors who live in states that come alphabetically after Massachusetts, where New Moon Books is located.

USE pubs SELECT p.pub_name, p.state, a.au_lname, a.au_fname, a.state FROM publishers p INNER JOIN authors a ON a.state > p.state WHERE p.pub_name = 'New Moon Books' ORDER BY au_lname ASC, au_fname ASC

Here is the result set:

pub_name stat	te au_	_lname a	au_fname	state
New Moon Books	MA	Blotchet-Halls	Reginald	OR
New Moon Books	MA	del Castillo	Innes	MI
New Moon Books	MA	Greene	Morningstar	TN
New Moon Books	MA	Panteley	Sylvia	MD
New Moon Books	MA	Ringer	Albert	UT
New Moon Books	MA	Ringer	Anne	UT

(6 row(s) affected)

Joins Using the Not-equal Operator

The not-equal join (< >) is rarely used. As a general rule, not-equal joins make sense only when used with a self-join. For example, this not-equal Transact-SQL join and self-join are used to find the categories with two or more inexpensive (less than \$15) books of different prices:

```
USE pubs
SELECT DISTINCT t1.type, t1.price
FROM titles t1 INNER JOIN titles t2
ON t1.type = t2.type
AND t1.price <> t2.price
WHERE t1.price < $15 AND t2.price < $15
```

Note The expression NOT *column_name* = *column_name* is equivalent to *column_name* < > *column_name*.

This Transact-SQL example uses a not-equal join combined with a self-join to find all rows in the **titleauthor** table in which two or more rows have the same **title_id** but different **au_id** numbers (that is, books with more than one author):

```
USE pubs
SELECT DISTINCT t1.au_id, t1.title_id
FROM titleauthor t1 INNER JOIN titleauthor t2
ON t1.title_id = t2.title_id
WHERE t1.au_id <> t2.au_id
ORDER BY t1.au_id
```

Here is the result set:

au_id	title_id	
213-46-8915	BU1032	
267-41-2394	BU1111	
267-41-2394	· TC7777	
409-56-7008	BU1032	
427-17-2319	PC8888	
472-27-2349	TC7777	
672-71-3249	TC7777	
722-51-5454	MC3021	
724-80-9391	BU1111	
724-80-9391	PS1372	
756-30-7391	PS1372	
846-92-7186	PC8888	
899-46-2035	MC3021	
899-46-2035	PS2091	
998-72-3567	PS2091	

(15 row(s) affected)

See Also

Conversion Functions

<u>WHERE</u>

SELECT Examples

<u>SELECT</u>

Using Outer Joins

Inner joins return rows only when there is at least one row from both tables that matches the join condition. Inner joins eliminate the rows that do not match with a row from the other table. Outer joins, however, return all rows from at least one of the tables or views mentioned in the FROM clause, as long as those rows meet any WHERE or HAVING search conditions. All rows are retrieved from the left table referenced with a left outer join, and all rows from the right table referenced in a right outer join. All rows from both tables are returned in a full outer join

Microsoft® SQL Server[™] 2000 uses these SQL-92 keywords for outer joins specified in a FROM clause:

- LEFT OUTER JOIN or LEFT JOIN
- RIGHT OUTER JOIN or RIGHT JOIN
- FULL OUTER JOIN or FULL JOIN

SQL Server supports both the SQL-92 outer join syntax and a legacy syntax for specifying outer joins based on using the *= and =* operators in the WHERE clause. The SQL-92 syntax is recommended because it is not subject to the ambiguity that sometimes results from the legacy Transact-SQL outer joins.

Using Left Outer Joins

Consider a join of the **authors** table and the **publishers** table on their **city** columns. The results show only the authors who live in cities in which a publisher is located (in this case, Abraham Bennet and Cheryl Carson).

To include all authors in the results, regardless of whether a publisher is located in the same city, use an SQL-92 left outer join. The following is the query and results of the Transact-SQL left outer join:

USE pubs

SELECT a.au_fname, a.au_lname, p.pub_name FROM authors a LEFT OUTER JOIN publishers p ON a.city = p.city ORDER BY p.pub_name ASC, a.au_lname ASC, a.au_fname ASC

Here is the result set:

au_fname	au_lname	pub_name
Reginald	Blotchet-Halls	NULL
Michel	DeFrance	NULL
Innes	del Castillo	NULL
Ann	Dull	NULL
Marjorie	Green	NULL
Morningstar	Greene	NULL
Burt	Gringlesby	NULL
Sheryl	Hunter	NULL
Livia	Karsen	NULL
Charlene	Locksley	NULL
Stearns	MacFeather	NULL
Heather	McBadden	NULL
Michael	O'Leary	NULL
Sylvia	Panteley	NULL
Albert	Ringer	NULL
Anne	Ringer	NULL
Meander	Smith	NULL
Dean	Straight	NULL
Dirk	Stringer	NULL
Johnson	White	NULL
Akiko	Yokomoto	NULL
Abraham	Bennet	Algodata Infosyster
Cheryl	Carson	Algodata Infosystems

(23 row(s) affected)

The LEFT OUTER JOIN includes all rows in the **authors** table in the results, whether or not there is a match on the **city** column in the **publishers** table. Notice that in the results there is no matching data for most of the authors listed; therefore, these rows contain null values in the **pub_name** column.

Using Right Outer Joins

Consider a join of the **authors** table and the **publishers** table on their **city** columns. The results show only the authors who live in cities where a publisher is located (in this case, Abraham Bennet and Cheryl Carson). The SQL-92 right outer join operator, RIGHT OUTER JOIN, indicates all rows in the second table are to be included in the results, regardless of whether there is matching data in the first table.

To include all publishers in the results, regardless of whether a city has a publisher located in the same city, use an SQL-92 right outer join. Here is the Transact-SQL query and results of the right outer join:

USE pubs

SELECT a.au_fname, a.au_lname, p.pub_name

FROM authors AS a RIGHT OUTER JOIN publishers AS p

ON a.city = p.city

ORDER BY p.pub_name ASC, a.au_lname ASC, a.au_fname ASC

Here is the result set:

au_fname	au_lname	pub_name
Abraham	Bennet	Algodata Infosystems
Cheryl	Carson	Algodata Infosystems
NULL	NULL	Binnet & Hardley
NULL	NULL	Five Lakes Publishing
NULL	NULL	GGG&G
NULL	NULL	Lucerne Publishing
NULL	NULL	New Moon Books
NULL	NULL	Ramona Publishers
NULL	NULL	Scootney Books

```
(9 row(s) affected)
```

An outer join can be further restricted by using a predicate (such as comparing the join to a constant). This example contains the same right outer join, but eliminates all titles that have sold fewer than 50 copies:

```
USE pubs
SELECT s.stor_id, s.qty, t.title
FROM sales s RIGHT OUTER JOIN titles t
ON s.title_id = t.title_id
AND s.qty > 50
ORDER BY s.stor_id ASC
```

Here is the result set:

stor_id qty title

- (null) (null) But Is It User Friendly?
- (null) (null) Computer Phobic AND Non-Phobic Individuals: Behavior Variations
- (null) (null) Cooking with Computers: Surreptitious Balance Sheets
- (null) (null) Emotional Security: A New Algorithm
- (null) (null) Fifty Years in Buckingham Palace Kitchens
- 7066 75 Is Anger the Enemy?
- (null) (null) Life Without Fear
- (null) (null) Net Etiquette
- (null) (null) Onions, Leeks, and Garlic: Cooking Secrets of the Mediterranean
- (null) (null) Prolonged Data Deprivation: Four Case Studies
- (null) (null) Secrets of Silicon Valley
- (null) (null) Silicon Valley Gastronomic Treats
- (null) (null) Straight Talk About Computers
- (null) (null) Sushi, Anyone?
- (null) (null) The Busy Executive's Database Guide

```
(null) (null) The Gourmet Microwave(null) (null) The Psychology of Computer Cooking(null) (null) You Can Combat Computer Stress!
```

(18 row(s) affected)

For more information about predicates, see <u>WHERE</u>.

Using Full Outer Joins

To retain the nonmatching information by including nonmatching rows in the results of a join, use a full outer join. Microsoft® SQL Server[™] 2000 provides the full outer join operator, FULL OUTER JOIN, which includes all rows from both tables, regardless of whether or not the other table has a matching value.

Consider a join of the **authors** table and the **publishers** table on their **city** columns. The results show only the authors who live in cities in which a publisher is located (in this case, Abraham Bennet and Cheryl Carson). The SQL-92 FULL OUTER JOIN operator indicates that all rows from both tables are to be included in the results, regardless of whether there is matching data in the tables.

To include all publishers and all authors in the results, regardless of whether a city has a publisher located in the same city, or whether a publisher is located in the same city, use a full outer join. The following is the query and results of the Transact-SQL full outer join:

```
USE pubs
SELECT a.au_fname, a.au_lname, p.pub_name
FROM authors a FULL OUTER JOIN publishers p
ON a.city = p.city
ORDER BY p.pub_name ASC, a.au_lname ASC, a.au_fname ASC
```

Here is the result set:

au_fname	au_lname	pub_name	
Reginald	Blotchet-Halls	NULL	

Michel	DeFrance	NULL
Innes	del Castillo	NULL
Ann	Dull	NULL
Marjorie	Green	NULL
Morningstar	Greene	NULL
Burt	Gringlesby	NULL
Sheryl	Hunter	NULL
Livia	Karsen	NULL
Charlene	Locksley	NULL
Stearns	MacFeather	NULL
Heather	McBadden	NULL
Michael	O'Leary	NULL
Sylvia	Panteley	NULL
Albert	Ringer	NULL
Anne	Ringer	NULL
Meander	Smith	NULL
Dean	Straight	NULL
Dirk	Stringer	NULL
Johnson	White	NULL
Akiko	Yokomoto	NULL
Abraham	Bennet	Algodata Infosystems
Cheryl	Carson	Algodata Infosystems
NULL	NULL	Binnet & Hardley
NULL	NULL	Five Lakes Publishing
NULL	NULL	GGG&G
NULL	NULL	Lucerne Publishing
NULL	NULL	New Moon Books
NULL	NULL	Ramona Publishers
NULL	NULL	Scootney Books

(30 row(s) affected)

See Also

Operators SELECT Using Operators in Expressions SELECT Examples Writing Readable Code

Using Cross Joins

A cross join that does not have a WHERE clause produces the Cartesian product of the tables involved in the join. The size of a Cartesian product result set is the number of rows in the first table multiplied by the number of rows in the second table. This is an example of a Transact-SQL cross join:

USE pubs SELECT au_fname, au_lname, pub_name FROM authors CROSS JOIN publishers ORDER BY au_lname DESC

The result set contains 184 rows (**authors** has 23 rows and **publishers** has 8; 23 multiplied by 8 equals 184).

However, if a WHERE clause is added, the cross join behaves as an inner join. For example, these Transact-SQL queries produce the same result set:

USE pubs SELECT au_fname, au_lname, pub_name FROM authors CROSS JOIN publishers WHERE authors.city = publishers.city ORDER BY au_lname DESC

-- Or USE pubs SELECT au_fname, au_lname, pub_name FROM authors INNER JOIN publishers ON authors.city = publishers.city ORDER BY au_lname DESC

See Also

WHERE

SELECT Operators SELECT Examples Using Operators in Expressions

Using Self-Joins

A table can be joined to itself in a self-join. For example, you can use a self-join to find out the authors in Oakland, California who live in the same ZIP Code area.

Because this query involves a join of the **authors** table with itself, the **authors** table appears in two roles. To distinguish these roles, you must give the **authors** table two different aliases (**au1** and **au2**) in the FROM clause. These aliases are used to qualify the column names in the rest of the query. This is an example of the self-join Transact-SQL statement:

```
USE pubs
SELECT au1.au_fname, au1.au_lname, au2.au_fname, au2.au_lname
FROM authors au1 INNER JOIN authors au2
ON au1.zip = au2.zip
```

```
WHERE au1.city = 'Oakland'
```

ORDER BY au1.au_fname ASC, au1.au_lname ASC

au_fname	au_lname	au_fname	au_lname
Dean	Straight	Dean	Straight
Dean	Straight	Dirk	Stringer
Dean	Straight	Livia	Karsen
Dirk	Stringer	Dean	Straight
Dirk	Stringer	Dirk	Stringer
Dirk	Stringer	Livia	Karsen
Livia	Karsen	Dean	Straight
Livia	Karsen	Dirk	Stringer
Livia	Karsen	Livia	Karsen
Marjorie	Green	Marjorie	Green
Stearns	MacFeather	Stearns	MacFeather

Here is the result set:

(11 row(s) affected)

To eliminate the rows in the results in which the authors match themselves and to eliminate rows that are identical, except the order of the authors is reversed, make this change to the Transact-SQL self-join query:

```
USE pubs
SELECT au1.au_fname, au1.au_lname, au2.au_fname, au2.au_lname
FROM authors au1 INNER JOIN authors au2
ON au1.zip = au2.zip
WHERE au1.city = 'Oakland'
AND au1.state = 'CA'
AND au1.au_id < au2.au_id
ORDER BY au1.au lname ASC, au1.au fname ASC
```

Here is the result set:

au_fname	au_lname	au_fname	au_lname
Dean	Straight	Dirk	Stringer
Dean	Straight	Livia	Karsen
Dirk	Stringer	Livia	Karsen

(3 row(s) affected)

It is now clear that Dean Straight, Dirk Stringer, and Livia Karsen all have the same ZIP Code and live in Oakland, California.

See Also WHERE SELECT Operators SELECT Examples

Using Operators in Expressions

Joining Three or More Tables

Although each join specification joins only two tables, FROM clauses can contain multiple join specifications. This allows many tables to be joined for a single query.

The **titleauthor** table of the **pubs** database offers a good example of a situation in which joining more than two tables is helpful. This Transact-SQL query finds the titles of all books of a particular type and the names of their authors:

```
USE pubs
SELECT a.au_lname, a.au_fname, t.title
FROM authors a INNER JOIN titleauthor ta
ON a.au_id = ta.au_id JOIN titles t
ON ta.title_id = t.title_id
WHERE t.type = 'trad_cook'
ORDER BY t.title ASC
```

Here is the result set:

au_lname	au_fname	title	
Blotchet-Halls	Reginald	Fifty Years in Buckingham Palace	
Kitchens			
Panteley	Sylvia	Onions, Leeks, and Garlic:	
Cooking Secrets of the Mediterranean			
O'Leary	Michael	Sushi, Anyone?	
Gringlesby	Burt	Sushi, Anyone?	
Yokomoto	Akiko	Sushi, Anyone?	

(5 row(s) affected)

Notice that one of the tables in the FROM clause, **titleauthor**, does not contribute any columns to the results. Also, none of the joined columns, **au_id**

and **title_id**, appear in the results. Nonetheless, this join is possible only by using **titleauthor** as an intermediate table.

The middle table of the join (the **titleauthor** table) can be called the translation table or intermediate table, because **titleauthor** is an intermediate point of connection between the other tables involved in the join.

When there is more than one join operator in the same statement, either to join more than two tables or to join more than two pairs of columns, the join expressions can be connected with AND or with OR.

See Also WHERE SELECT Operators SELECT Examples

Null Values and Joins

When there are null values in the columns of the tables being joined, the null values do not match each other. The presence of null values in a column from one of the tables being joined can be returned only by using an outer join (unless the WHERE clause excludes null values).

Here are two tables that each have NULL in the column that will participate in the join:

table1		table2	
a	b	c d	
1	one	NULL	two
NUL	L three	4	four
4	join4		

A join that compares the values in column **a** against column **c** does not get a match on the columns that have values of NULL:

```
SELECT *
FROM table1 t1 JOIN table2 t2
ON t1.a = t2.c
ORDER BY t1.a
```

Only one row with 4 in column **a** and **c** is returned:

a b c d ------4 join4 4 four

(1 row(s) affected)

Null values returned from a base table are also difficult to distinguish from the null values returned from an outer join. For example, this SELECT statement does a left outer join on these two tables:

SELECT * FROM table1 t1 LEFT OUTER JOIN table2 t2 ON t1.a = t2.c ORDER BY t1.a

Here is the result set:

a b c d ------NULL three NULL NULL 1 one NULL NULL 4 join4 4 four

(3 row(s) affected)

The results do not make it easy to distinguish a NULL in the data from a NULL that represents a failure to join. When null values are present in data being joined, it is usually preferable to omit them from the results by using a regular join.

See Also

sp_dbcmptlevel

<u>WHERE</u>

Specifying Joins in FROM or WHERE Clauses

The rows selected by a query are filtered first by the FROM clause join conditions, then the WHERE clause search conditions, and then the HAVING clause search conditions. Inner joins can be specified in either the FROM or WHERE clause without affecting the final result.

Outer join conditions, however, may interact differently with the WHERE clause search conditions, depending on whether the join conditions are in the FROM or WHERE clause. Therefore, the ability to specify Transact-SQL outer joins in the WHERE clause is not recommended, is no longer documented, and will be dropped in a future release.

For example, these queries both specify a left outer join to SELECT 23 rows that display the title identification number, title name, and the number of books sold:

-- Join in WHERE clause. USE pubs SELECT t.title_id, t.title, s.qty FROM titles AS t, sales AS s WHERE t.title_id *= s.title_id

```
-- Join in FROM clause.
USE pubs
SELECT t.title_id, t.title, s.qty
FROM titles AS t LEFT OUTER JOIN sales AS s
ON t.title_id = s.title_id
```

In this query, a search condition is also specified in the WHERE clause:

-- Join and search condition in WHERE clause. USE pubs SELECT t.title_id, t.title, s.qty FROM titles AS t, sales AS s WHERE t.title_id *= s.title_id AND s.stor_id = '7066'

The condition **stor_id** = '7066' is evaluated along with the join. The join only selects the rows for **stor_id** 7066 from the sales table, but because it is an outer join null values are supplied as the store information in all the other rows. This query returns 18 rows.

The join condition can be moved to the FROM clause, and the **stor_id** condition left in the WHERE clause:

```
USE pubs
SELECT t.title_id, t.title, s.qty
FROM titles AS t LEFT OUTER JOIN sales AS s
ON t.title_id = s.title_id
WHERE s.stor_id = '7066'
```

This query returns only two rows because the restriction of **stor_id** = '7066' is applied after the left outer join has been performed. This eliminates all the rows from the outer join that have NULL for their **stor_id**. To return the same information with the join condition in the FROM clause, specify the **stor_id** = '7066' condition as part of the ON *join_criteria* section in the FROM clause and remove the WHERE clause:

USE pubs SELECT t.title_id, t.title, s.qty FROM titles AS t LEFT OUTER JOIN sales AS s ON t.title_id = s.title_id AND s.stor_id = '7066'

See Also

WHERE

Using Operators in Expressions

Advanced Query Concepts

After you have mastered query fundamentals, you can explore these advanced query concepts for query solutions:

- Using aggregate functions in the select list
- Grouping rows with GROUP BY
- Combining results with UNION
- Subquery fundamentals
- Conditional data processing using CASE
- Parallel queries
- Summarizing data

See Also

Query Fundamentals

Using Aggregate Functions in the Select List

Aggregate functions (such as SUM, AVG, COUNT, COUNT(*), MAX, and MIN) generate summary values in query result sets. An aggregate function (with the exception of COUNT(*)) processes all the selected values in a single column to produce a single result value. Aggregate functions can be applied to all rows in a table, to a subset of the table specified by a WHERE clause, or to one or more groups of rows in the table. When an aggregate function is applied, a single value is generated from each set of rows.

This example calculates the sum of year-to-date sales for all books in the **titles** table:

USE pubs SELECT SUM(ytd_sales) FROM titles

Here is the result set:

97446

(1 row(s) affected)

With this query, you can find the average price of all books if prices were doubled:

USE pubs SELECT avg(price * 2) FROM titles

Here is the result set:

29.53

(1 row(s) affected)

The table shows the syntax of the aggregate functions and their results (*expression* is almost always a column name).

Aggregate function	Result
SUM([ALL DISTINCT] expression)	Total of the values in the numeric
	expression
AVG([ALL DISTINCT] expression)	Average of the values in the
	numeric expression
COUNT([ALL DISTINCT]	Number of values in the
expression)	expression
COUNT(*)	Number of selected rows
MAX(expression)	Highest value in the expression
MIN(expression)	Lowest value in the expression

SUM, AVG, COUNT, MAX, and MIN ignore null values; COUNT(*) does not.

The optional keyword DISTINCT can be used with SUM, AVG, and COUNT to eliminate duplicate values before an aggregate function is applied (the default is ALL).

SUM and AVG can be used only with numeric columns, for example **int**, **smallint**, **tinyint**, **decimal**, **numeric**, **float**, **real**, **money**, and **smallmoney** data types. MIN and MAX cannot be used with **bit** data types. Aggregate functions other than COUNT(*) cannot be used with **text** and **image** data types.

With these exceptions, aggregate functions can be used with any type of column. For example, in a **character** data type column, use MIN (minimum) to find the lowest value (the one closest to the beginning of the alphabet):

USE pubs SELECT MIN(au_lname) FROM authors

Here is the result set:

Bennet

(1 row(s) affected)

The result type returned by an aggregate function may have a larger precision than the inputs so that the result type is large enough to hold the aggregated result value. For example, the SUM or AVG functions return an **int** value when the data type of the inputs is **smallint** or **tinyint**. For more information about the data type returned by an aggregate function, see the topic for the function in Microsoft® SQL Server[™] 2000 Transact-SQL Reference.

Note The output for statements, involving MIN or MAX on character columns, depends on the collation chosen during installation. For more information about the effects of different collations, see <u>SQL Server Collation Fundamentals</u>.

When aggregate functions are used in a select list, the select list can contain only:

- Aggregate functions.
- Grouping columns from a GROUP BY clause.
- An expression that returns the same value for every row in the result set, such as a constant.

For more information about generating aggregate values for result sets containing multiple rows, see <u>Grouping Rows with GROUP BY</u>.

Aggregate functions cannot be used in a WHERE clause. However, a SELECT statement with aggregate functions in its select list often includes a WHERE clause that restricts the rows to which the aggregate function is applied. If a SELECT statement includes a WHERE clause (but not a GROUP BY clause), an aggregate function produces a single value for the subset of rows specified by the WHERE clause. This is true whether it is operating on all rows in a table or on a subset of rows defined by a WHERE clause. Such a function is called a scalar aggregate.

This query returns the average advance and the sum of year-to-date sales for business books only:

USE pubs SELECT AVG(advance), SUM(ytd_sales) FROM titles WHERE type = 'business'

Here is the result set:

-----6,281.25 30788

(1 row(s) affected)

You can use more than one aggregate function in the same select list and produce more than one scalar aggregate in a single SELECT statement.

See Also

Aggregate Functions
SELECT

Using COUNT(*)

COUNT(*) does not require an *expression* parameter because it does not use information about any particular column. It counts the total number of rows that meet the qualifications of the query. COUNT(*) returns the number of rows that match the search conditions specified in the query without eliminating duplicates. It counts each row separately, including rows that contain null values. This query finds the total number of books in **titles**:

USE pubs SELECT COUNT(*) FROM titles

Here is the result set:

18

(1 row(s) affected)

COUNT(*) can be combined with other aggregate functions. This query shows COUNT(*) combined with an AVG function in which both aggregate functions aggregate data only from the rows that satisfy the WHERE clause search condition:

USE pubs SELECT COUNT(*), AVG(price) FROM titles WHERE advance > \$1000

Here is the result set:

15 14.42

(1 row(s) affected)

See Also

<u>COUNT</u>

Using DISTINCT

The DISTINCT keyword is optional with SUM, AVG, and COUNT. When DISTINCT is used, duplicate values are eliminated before the sum, average, or count is calculated.

If you use DISTINCT, the expression must consist of a column name only. It cannot include an arithmetic expression.

This query returns the average prices of business books (without duplicate values):

USE pubs SELECT AVG(DISTINCT price) FROM titles WHERE type = 'business'

Here is the result set:

14.64

(1 row(s) affected)

Without DISTINCT, the AVG function finds the average price of all business titles:

USE pubs SELECT AVG(price) FROM titles WHERE type = 'business

Here is the result set:

13.73

(1 row(s) affected)

See Also

Aggregate Functions

<u>SELECT</u>

Null Values

Null values in a column are ignored while an aggregate function is operating. For example, the count of advances in the **titles** table is not the same as the count of title names because null values in the **advance** column are not counted.

USE pubs SELECT COUNT(advance) FROM titles

Here is the result set:

16

(1 row(s) affected)

USE pubs SELECT COUNT(title) FROM titles

Here is the result set:

18

(1 row(s) affected)

If no rows meet the condition(s) specified in the WHERE clause, COUNT returns a value of zero. The other functions all return NULL. COUNT(*), counts each row, even if all column values are NULL. Here are examples:

USE pubs SELECT COUNT(DISTINCT title) FROM titles WHERE type = 'poetry'

Here is the result set:

0

(1 row(s) affected)

USE pubs SELECT AVG(advance) FROM titles WHERE type = 'poetry'

Here is the result set:

(null)

(1 row(s) affected)

Grouping Rows with GROUP BY

The GROUP BY clause is used to produce aggregate values for each row in the result set. When used without a GROUP BY clause, aggregate functions report only one aggregate value for a SELECT statement.

This example returns the number of units sold for each product in category 2:

```
USE Northwind
SELECT OrdD.ProductID AS ProdID,
SUM(OrdD.Quantity) AS AmountSold
FROM [Order Details] AS OrdD JOIN Products as Prd
ON OrdD.ProductID = Prd.ProductID
AND Prd.CategoryID = 2
GROUP BY OrdD.ProductID
```

Here is the result set:

ProdID	AmountSold
3	328
4	453
5	298
6	301
8	372
15	122
44	601
61	603
63	445
65	745
66	239
77	791

(12 row(s) affected)

The GROUP BY keywords are followed by a list of columns, known as the grouping columns. The GROUP BY clause restricts the rows of the result set; there is only one row for each distinct value in the grouping column or columns. Each result set row contains summary data related to the specific value in its grouping columns.

There are restrictions on the items that can be specified in the select list when a SELECT statement contains a GROUP BY. Items allowed in the select list are:

- The grouping columns.
- Expressions that return only one value for each value in the grouping columns, such as aggregate functions that have a column name as one of their parameters. These are known as vector aggregates.

ColumnA	ColumnB	ColumnC
1	abc	5
1	def	4
1	ghi	9
2	jkl	8
2	mno	3

For example, **TableX** contains:

If **ColumnA** is the grouping column, there will be two rows in the result set, one summarizing the information for the value 1, and the other summarizing the information for value 2.

When **ColumnA** is the grouping column, the only way **ColumnB** or **ColumnC** can be referenced is if they are parameters in an aggregate function that can return a single value for each value in **ColumnA**. It is legal for the select list to include expressions such as MAX(**ColumnB**), SUM(**ColumnC**), or AVG(**ColumnC**):

SELECT ColumnA, MAX(ColumnB) AS MaxB, SUM(ColumnC) AS SumC FROM TableX GROUP BY ColumnA

This select returns two rows, one for each unique value in **ColumnA**:

ColumnA MaxB SumC -----1 ghi 18 2 mno 11

(2 row(s) affected)

It is not legal, however, to have just the expression **ColumnB** in the select list:

```
SELECT ColumnA,
ColumnB,
SUM(ColumnC) AS SumC
FROM TableX
GROUP BY ColumnA
```

Because the GROUP BY can return only one row with a value of 1 in **ColumnA**, there is no way to return the three values of **ColumnB** (abc, def, and ghi) associated with the value 1 in **ColumnA**.

You cannot use GROUP BY or HAVING on **ntext,text,image**, or **bit** columns unless they are in a function that returns a value having another data type. Examples of such functions are SUBSTRING and CAST.

See Also

CAST and CONVERT
SUBSTRING

<u>SELECT</u>

GROUP BY Components

The GROUP BY clause contains the following components:

- One or more aggregate-free expressions. These are usually references to the grouping columns.
- Optionally, the ALL keyword, which specifies that all groups produced by the GROUP BY clause are returned, even if some of the groups do not have any rows that meet the search conditions.
- CUBE or ROLLUP.
- Typically, the HAVING clause is used with the GROUP BY clause, although HAVING can be specified separately.

You can group by an expression as long as it does not include aggregate functions, for example:

SELECT DATEPART(yy, HireDate) AS Year, COUNT(*) AS NumberOfHires FROM Northwind.dbo.Employees GROUP BY DATEPART(yy, HireDate)

This is the result set.

- Year NumberOfHires
- 1992 3
- 1993 3
- 1994 3

(3 row(s) affected)

In a GROUP BY, you must specify the name of a table or view column, not the

name of a result set column assigned with an AS clause. For example, replacing the GROUP BY DATEPART(yy, HireDate) clause with GROUP BY Year is not legal.

You can list more than one column in the GROUP BY clause to nest groups; that is, you can group a table by any combination of columns. For example, this query finds the average price and the sum of year-to-date sales, grouped by type and publisher ID:

USE pubs SELECT type, pub_id, 'avg' = AVG(price), 'sum' = sum(ytd_sales) FROM titles GROUP BY type, pub_id

Here is the result set:

type pub_id avg	sum	
business 0736 2.99	18722	
psychology 0736 11.48	9564	
mod_cook 0877 11.49	24278	
psychology 0877 21.59	375	
trad_cook 0877 15.96	19566	
UNDECIDED 0877 NULL	NULL	
business 1389 17.31	12066	
popular_comp 1389 21.48	12875	

(8 row(s) affected)

See Also

<u>SELECT</u>

GROUP BY and the WHERE Clause

You can use a WHERE clause in a query containing a GROUP BY clause. Rows not meeting the conditions in the WHERE clause are eliminated before any grouping is done. For example:

USE pubs SELECT type, AVG(price) FROM titles WHERE advance > \$5000 GROUP BY type

Here is the result set:

type

business 2.99 mod_cook 2.99 popular_comp 21.48 psychology 14.30 trad_cook 17.97

(5 row(s) affected)

Only rows with advances greater than \$5,000 are included in the groups shown in the query results.

See Also

<u>SELECT</u>

WHERE

Choosing Rows with the HAVING Clause

The HAVING clause sets conditions on the GROUP BY clause similar to the way WHERE interacts with SELECT. The WHERE search condition is applied before the grouping operation occurs; the HAVING search condition is applied after the grouping operation occurs. The HAVING syntax is similar to the WHERE syntax, except HAVING can contain aggregate functions. HAVING clauses can reference any of the items that appear in the select list.

This query finds publishers whose year-to-date sales are greater than \$40,000:

USE pubs SELECT pub_id, total = SUM(ytd_sales) FROM titles GROUP BY pub_id HAVING SUM(ytd_sales) > 40000

Here is the result set:

pub_id total

0877 44219

(1 row(s) affected)

To make sure there are at least six books involved in the calculations for each publisher, this example uses HAVING COUNT(*) > 5 to eliminate the publishers that return totals for fewer than six books:

USE pubs SELECT pub_id, total = SUM(ytd_sales) FROM titles GROUP BY pub_id HAVING COUNT(*) > 5 Here is the result set:

pub_id total

0877 44219 1389 24941

(2 row(s) affected)

Understanding the correct sequence in which the WHERE, GROUP BY, and HAVING clauses are applied helps in coding efficient queries:

- The WHERE clause is used to filter the rows that result from the operations specified in the FROM clause.
- The GROUP BY clause is used to group the output of the WHERE clause.
- The HAVING clause is used to filter rows from the grouped result.

For any search conditions that could be applied either before or after the grouping operation, it is more efficient to specify them in the WHERE clause. This reduces the number of rows that have to be grouped. The only search conditions that should be specified in the HAVING clause are those search conditions that must be applied after the grouping operation has been performed.

The Microsoft® SQL Server[™] 2000 query optimizer can deal with most of these conditions. If the query optimizer determines that a HAVING search condition can be applied before the grouping operation, it will do so. The query optimizer might not be able to recognize all of the HAVING search conditions that can be applied before the grouping operation. It is recommended that you place all such search conditions in the WHERE clause instead of the HAVING clause.

The following query shows HAVING with an aggregate function. It groups the rows in the **titles** table by type and eliminates the groups that include only one book:

USE pubs

SELECT type FROM titles GROUP BY type HAVING COUNT(*) > 1

Here is the result set:

type

business mod_cook popular_comp psychology trad_cook

(5 row(s) affected)

This is an example of a HAVING clause without aggregate functions. It groups the rows in **titles** by type and eliminates those types that do not start with the letter p.

USE pubs SELECT type FROM titles GROUP BY type HAVING type LIKE 'p%'

Here is the result set:

type

popular_comp psychology

(2 row(s) affected)

When multiple conditions are included in HAVING, they are combined with AND, OR, or NOT. The following example shows how to group **titles** by publisher, including only those publishers with identification numbers greater than 0800, who have paid more than \$15,000 in total advances, and who sell books for an average of less than \$20.

```
SELECT pub_id, SUM(advance) AS AmountAdvanced,
        AVG(price) AS AveragePrice
FROM pubs.dbo.titles
WHERE pub_id > '0800'
GROUP BY pub_id
HAVING SUM(advance) > $15000
        AND AVG(price) < $20</pre>
```

ORDER BY can be used to order the output of a GROUP BY clause. This example shows using the ORDER BY clause to define the order in which the rows from a GROUP BY clause are returned:

```
SELECT pub_id, SUM(advance) AS AmountAdvanced,
        AVG(price) AS AveragePrice
FROM pubs.dbo.titles
WHERE pub_id > '0800'
        AND price >= $5
GROUP BY pub_id
HAVING SUM(advance) > $15000
        AND AVG(price) < $20
ORDER BY pub_id DESC
```

See Also

SELECT

GROUP BY and ALL

Transact-SQL provides the ALL keyword in the GROUP BY clause. ALL is meaningful only when the SELECT statement also includes a WHERE clause.

If you use ALL, the query results include all groups produced by the GROUP BY clause, even if some of the groups have no rows that meet the search conditions. Without ALL, a SELECT statement that includes GROUP BY does not show groups for which no rows qualify.

Here are examples:

```
USE pubs
SELECT type, AVG(price)
FROM titles
WHERE royalty = 10
GROUP BY type
```

Here is the result set:

type

business 17.31 popular_comp 20.00 psychology 14.14 trad_cook 17.97

(4 row(s) affected)

USE pubs SELECT type, AVG(price) FROM titles WHERE royalty = 10 GROUP BY ALL type Here is the result set:

type

business 17.31 mod_cook (null) popular_comp 20.00 psychology 14.14 trad_cook 17.97 UNDECIDED (null)

(6 row(s) affected)

The first query produces groups only for those books that commanded royalties of 10 percent. Because no modern cookbooks have a royalty of 10 percent, there is no group in the results for the **mod_cook** type.

The second query produces groups for all types, including modern cookbooks and UNDECIDED, even though the modern cookbook group does not include any rows that meet the qualification specified in the WHERE clause.

The column that holds the aggregate value (the average price) is NULL for groups that lack qualifying rows.

See Also

<u>ALL</u>

<u>SELECT</u>

GROUP BY and Null Values

If the grouping column contains a null value, that row becomes a group in the results. If the grouping column contains more than one null value, the null values are put into a single group. This behavior is defined in the SQL-92 standard.

The **royalty** column in the **titles** table contains some null values, for example:

```
SELECT royalty, AVG(price * 2) AS AveragePrice
FROM pubs.dbo.titles
GROUP BY royalty
```

Here is the result set:

royalty AveragePrice

(null)	(null)
10	32.89
12	30.94
14	23.90
16	45.90
24	5.98

(6 row(s) affected)

This SELECT statement can be changed to remove the null values by adding a WHERE clause:

SELECT royalty, AVG(price * 2) AS AveragePrice FROM pubs.dbo.titles WHERE royalty IS NOT NULL GROUP BY royalty

Combining Results with UNION

The UNION operator allows you to combine the results of two or more SELECT statements into a single result set. The result sets combined using UNION must all have the same structure. They must have the same number of columns, and the corresponding result set columns must have compatible data types. For more information, see <u>Guidelines for Using UNION</u>.

UNION is specified as:

select_statement UNION [ALL] select_statement

Table1 Table2 ColumnB ColumnD ColumnA ColumnC char(4) char(4) int int _____ ___ ___ ghi 3 abc 1 def 2 ikl 4 3 ghi 5 mno

For example, **Table1** and **Table2** have the same two-column structure.

This query creates a UNION between the tables:

SELECT * FROM Table1 UNION SELECT * FROM Table2

Here is the result set:

ColumnA ColumnB

abc 1 def 2

ghi 3

jkl 4 mno 5

The result set column names of a UNION are the same as the column names in the result set of the first SELECT statement in the UNION. The result set column names of the other SELECT statements are ignored.

By default, the UNION operator removes duplicate rows from the result set. If you use ALL, all rows are included in the results and duplicates are not removed.

The exact results of a UNION operation depend on the collation chosen during installation and the ORDER BY clause. For more information about the effects of different collations, see <u>SQL Server Collation Fundamentals</u>.

Any number of UNION operators can appear in a Transact-SQL statement, for example:

```
SELECT * FROM TableA
UNION
SELECT * FROM TableB
UNION
SELECT * FROM TableC
UNION
SELECT * FROM TableD
```

By default, Microsoft® SQL Server[™] 2000 evaluates a statement containing UNION operators from left to right. Use parentheses to specify the order of evaluation. For example, the following statements are not equivalent:

```
/* First statement. */
SELECT * FROM TableA
UNION ALL
( SELECT * FROM TableB
UNION
SELECT * FROM TableC
)
GO
```

```
/* Second statement. */
(SELECT * FROM TableA
UNION ALL
SELECT * FROM TableB
)
UNION
SELECT * FROM TableC)
GO
```

In the first statement, duplicates are eliminated in the union between **TableB** and **TableC**. In the union between that set and **TableA**, duplicates are not eliminated. In the second statement, duplicates are included in the union between **TableA** and **TableB** but are eliminated in the subsequent union with **TableC**. ALL has no effect on the final result of this expression.

When UNION is used, the individual SELECT statements cannot have their own ORDER BY or COMPUTE clauses. There can be only one ORDER BY or COMPUTE clause after the last SELECT statement; it is applied to the final, combined result set. GROUP BY and HAVING can be specified only in the individual SELECT statements.

See Also

<u>UNION</u>

Guidelines when Using UNION

Follow these guidelines when using UNION operators:

- All select lists in the statements being combined with UNION must have the same number of expressions (column names, arithmetic expressions, aggregate functions, and so on).
- Corresponding columns in the result sets being combined with UNION, or any subset of columns used in individual queries, must be of the same data type, have an implicit data conversion possible between the two data types, or have an explicit conversion supplied. For example, UNION is not possible between a column of **datetime** data type and one of **binary** data type unless an explicit conversion is supplied, while UNION is possible between a column of **money** data type and one of **int** data type because they can be implicitly converted.
- Corresponding result set columns in the individual statements being combined with UNION must occur in the same order because UNION compares the columns one-to-one in the order given in the individual queries.

table3			table4	
a	b	С	а	b
int	char(4)	char(4)	char(4)	float
1	abc	jkl	jkl	1.000
2	def	mno	mno	5.000
3	ghi	pqr		

Here is an example.

Execute this query:

SELECT a, b FROM table3 UNION SELECT b, a FROM table4

Here is the result set:

a b 1.000000 abc 2.000000 def 3.000000 ghi 1.000000 jkl 5.000000 mno

When different data types are combined in a UNION operation, they are converted using the rules of data type precedence. In the preceding example, the **int** values are converted to **float** because **float** has a higher precedence than **int**. For more information, see <u>Data Type Precedence</u>.

This query produces an error message because the data types of corresponding columns are not compatible:

```
SELECT b, c FROM table3
UNION
SELECT a, b FROM table4
```

 The column names in the table resulting from UNION are taken from the first individual query in the UNION statement. To refer to a column in the result set by a new name (for example, in an ORDER BY clause), the column must be referred to that way in the first SELECT: SELECT city AS Cities FROM stores_west UNION SELECT city FROM stores_east ORDER BY city See Also

<u>UNION</u>

Using UNION with Other Transact-SQL Statements

Follow these guidelines when using UNION with other Transact-SQL statements:

- The first query in the UNION statement can contain an INTO clause that creates a table to hold the final result set. Only the first query can use an INTO clause. If it appears anywhere else, Microsoft® SQL Server[™] 2000 displays an error message. Also, remember that if the select into/bulkcopy option is not set, SELECT INTO can create only temporary tables.
- ORDER BY and COMPUTE clauses to define the order of the final results or compute summary values are allowed only at the end of the UNION statement. They cannot be used within the individual queries that make up the UNION statement.
- GROUP BY and HAVING clauses can be used within individual queries only; they cannot be used to affect the final result set.
- The UNION operator can be used within an INSERT statement.
- The FOR BROWSE clause cannot be used in statements involving the UNION operator.

See Also

<u>SELECT</u>

<u>UNION</u>

Using Partitioned Views

Partitioned views allow the data in a large table to be split into smaller member tables. The data is partitioned between the member tables based on ranges of data values in one of the columns. The data ranges for each member table are defined in a CHECK constraint specified on the partitioning column. A view that uses UNION ALL to combine selects of all the member tables into a single result set is then defined. When SELECT statements referencing the view specify a search condition on the partition column, the query optimizer uses the CHECK constraint definitions to determine which member table contains the rows.

For example, a sales table that records sales for 1998 has been partitioned into 12 member tables, one for each month. Each member table has a constraint defined on the **OrderMonth** column:

```
CREATE TABLE May1998Sales
 (OrderID INT
                PRIMARY KEY,
 CustomerID
              INT
                      NOT NULL,
 OrderDate
            DATETIME
                          NULL
  CHECK (DATEPART(vy, OrderDate) = 1998),
 OrderMonth
              INT
  CHECK (OrderMonth = 5),
 DeliveryDate
               DATETIME
                            NULL.
   CHECK (DATEPART(mm, OrderDate) = OrderMonth)
 )
```

The application populating **May1998Sales** must ensure all rows have 5 in the **OrderMonth** column and the order date specifies a date in May, 1998. This is enforced by the constraints defined on the table.

A view is then defined that uses UNION ALL to select the data from all 12 member tables as a single result set:

CREATE VIEW Year1998Sales AS SELECT * FROM Jan1998Sales

UNION ALL SELECT * FROM Feb1998Sales UNION ALL SELECT * FROM Mar1998Sales UNION ALL SELECT * FROM Apr1998Sales **UNION ALL** SELECT * FROM May1998Sales **UNION ALL** SELECT * FROM Jun1998Sales UNION ALL SELECT * FROM Jul1998Sales UNION ALL SELECT * FROM Aug1998Sales UNION ALL SELECT * FROM Sep1998Sales **UNION ALL** SELECT * FROM Oct1998Sales UNION ALL SELECT * FROM Nov1998Sales **UNION ALL** SELECT * FROM Dec1998Sales

For example, this SELECT statement

SELECT * FROM Year1998Sales WHERE OrderMonth IN (5,6) AND CustomerID = 64892

The SQL Server query optimizer recognizes that the search condition in this SELECT statement references only rows in the **May1998Sales** and **Jun1998Sales** tables, and limits its search to those tables.

CHECK constraints are not needed for the partitioned view to return the correct results. However, if the CHECK constraints have not been defined, the query

optimizer must search all the tables instead of only those that cover the search condition on the partitioning column. Without the CHECK constraints, the view operates like any other view with UNION ALL. The query optimizer cannot make any assumptions about the values stored in different tables and it cannot skip searching the tables that participate in the view definition.

If all the member tables referenced by a partitioned view are on the same server, the view is a local partitioned view. If the member tables are on multiple servers, the view is a distributed partitioned view. Distributed partitioned views can be used to spread the database processing load of a system across a group of servers.

Partitioned views make it easier to maintain the member tables independently. For example, at the end of a period:

- The definition of the partitioned view for current results can be changed to add the newest period and drop the oldest period.
- The definition of the partitioned view for past results can be changed to add the period just dropped from the current results view. The past results view can also be updated to remove and archive the oldest period it covers.

When you insert data into the partitioned views, the **sp_executesql** system stored procedure can be used to create INSERT statements with execution plans that have a high chance of being reused in systems with many concurrent users.

See Also

Creating a Partitioned View Resolving Distributed Partitioned Views Federated SQL Server 2000 Servers ALTER VIEW SELECT CREATE VIEW sp_executesql

Designing Applications to Use Federated Database Servers

Updatable distributed partitioned views support having groups of Microsoft® SQL Server[™] 2000 servers cooperate in processing the database workload of the largest, multi-tier Web sites. While each server is administered independently, the instances of SQL Server 2000 on each server use distributed partitioned views to share the work. A group of autonomous servers that cooperate to share work is called a federation.

You build federated database servers by creating databases on each server, and then partitioning tables across the databases. Each original table is split into a member table on each member server. Each member table has a subset of the rows from the original table; the original table is partitioned horizontally across the member tables. When designing a federated database system, partition all the tables so that all related data is located on the same member server.

One result of partitioning tables across a set of federated database servers is a set of data routing rules. An application can match some piece of data it can infer from user requests against the data routing rules to determine which member server has most of the data required by the SQL statements the application must generate to satisfy the user request. For more information, see <u>Designing</u> <u>Federated Database Servers</u>.

In a multi-tier Windows DNA architecture, a system is implemented in these tiers:

- Users services tier. A set of thin clients that focus on managing the application user interface. The user services tier calls the next tier to perform the business functions needed to support user requests.
- Business services tier. A set of COM+ components that encapsulate the business logic of the organization. The business services tier uses the next tier for any permanent data storage that needs to be done.
- Data services tier. A set of components, such as SQL Server databases,

that can store data in a permanent medium. This is also called persisting the data.

In Windows DNA, the business services tier is designed as a set of COM+ components running on application servers. This allows Microsoft® Windows® 2000 Network Load Balancing to distribute the user requests evenly across the business tier. Because any user request can be processes on any application server, the business components must have some mechanism for routing the SQL statements they generate to the appropriate member server. The business components must be able to match some piece of information in the data received from the client against the data routing rules to determine what member server should process the request.

A flexible mechanism for implementing data routing in the business services tier is store the routing rules in a persistent store, such as SQL Server 2000 or Windows 2000 Active Directory, and having the business components retrieve them at run time. You can code a COM+ component that will match keys against the routing rules to determine which member server would most efficiently process the query. This COM+ routing component can then be called by any other COM+ component in the business services tier that needs to access the partitioned data. For example, in a system accessing customer data partitioned on customer ID, you could:

- Create a routing rules table recording which keys are maintained on each member server.
- Create a data routing business component that takes either one key value or the starting and ending keys of a range of key values as input. The COM+ component would read the routing rules table, compare the input key or key ranges against the key ranges recorded for each member server, and then return the name of the member server having the best match to the calling component or application.
- Code the general business services tier components or applications to always call the data routing component when executing an SQL statement referencing the partitioned view. The business component will use the server name returned by the data routing component to select the

database connection on which to execute the SQL statement.

This method requires no changes to application code if the partitioning of the data is changed. The data routing rules can be changed while the applications are running.

Subquery Fundamentals

A subquery is a SELECT query that returns a single value and is nested inside a SELECT, INSERT, UPDATE, or DELETE statement, or inside another subquery. A subquery can be used anywhere an expression is allowed. In this example a subquery is used as a column expression named **MaxUnitPrice** in a SELECT statement.

SELECT Ord.OrderID, Ord.OrderDate, (SELECT MAX(OrdDet.UnitPrice) FROM Northwind.dbo.[Order Details] AS OrdDet WHERE Ord.OrderID = OrdDet.OrderID) AS MaxUnitPrice FROM Northwind.dbo.Orders AS Ord

A subquery is also called an inner query or inner select, while the statement containing a subquery is also called an outer query or outer select.

Many Transact-SQL statements that include subqueries can be alternatively formulated as joins. Other questions can be posed only with subqueries. In Transact-SQL, there is usually no performance difference between a statement that includes a subquery and a semantically equivalent version that does not. However, in some cases where existence must be checked, a join yields better performance. Otherwise, the nested query must be processed for each result of the outer query to ensure elimination of duplicates. In such cases, a join approach would yield better results. This is an example showing both a subquery SELECT and a join SELECT that return the same result set:

```
/* SELECT statement built using a subquery. */
SELECT ProductName
FROM Northwind.dbo.Products
WHERE UnitPrice =
  (SELECT UnitPrice
  FROM Northwind.dbo.Products
  WHERE ProductName = 'Sir Rodney''s Scones')
```

/* SELECT statement built using a join that returns
 the same result set. */
SELECT Prd1.ProductName
FROM Northwind.dbo.Products AS Prd1
 JOIN Northwind.dbo.Products AS Prd2
 ON (Prd1.UnitPrice = Prd2.UnitPrice)
WHERE Prd2.ProductName = 'Sir Rodney''s Scones'

A subquery nested in the outer SELECT statement has the following components:

- A regular SELECT query including the regular select list components.
- A regular FROM clause including one or more table or view names.
- An optional WHERE clause.
- An optional GROUP BY clause.
- An optional HAVING clause.

The SELECT query of a subquery is always enclosed in parentheses. It cannot include a COMPUTE or FOR BROWSE clause, and may only include an ORDER BY clause when a TOP clause is also specified.

A subquery can be nested inside the WHERE or HAVING clause of an outer SELECT, INSERT, UPDATE, or DELETE statement, or inside another subquery. Up to 32 levels of nesting is possible, although the limit varies based on available memory and the complexity of other expressions in the query. Individual queries may not support nesting up to 32 levels. A subquery can appear anywhere an expression can be used, if it returns a single value.

If a table appears only in a subquery and not in the outer query, then columns from that table cannot be included in the output (the select list of the outer query).

Statements that include a subquery usually take one of these formats:

- WHERE *expression* [NOT] IN *(subquery)*
- WHERE *expression comparison_operator* [ANY | ALL] (*subquery*)
- WHERE [NOT] EXISTS (subquery)

In some Transact-SQL statements, the subquery can be evaluated as if it were an independent query. Conceptually, the subquery results are substituted into the outer query (although this is not necessarily how Microsoft® SQL Server[™] actually processes Transact-SQL statements with subqueries).

There are three basic types of subqueries. Those that:

- Operate on lists introduced with IN, or those that a comparison operator modified by ANY or ALL.
- Are introduced with an unmodified comparison operator and must return a single value.
- Are existence tests introduced with EXISTS.

See Also

<u>SELECT</u>

Subquery Rules

A subquery is subject to a number of restrictions:

- The select list of a subquery introduced with a comparison operator can include only one expression or column name (except that EXISTS and IN operate on SELECT * or a list, respectively).
- If the WHERE clause of an outer query includes a column name, it must be join-compatible with the column in the subquery select list.
- The **ntext**, **text** and **image** data types are not allowed in the select list of subqueries.
- Because they must return a single value, subqueries introduced by an unmodified comparison operator (one not followed by the keyword ANY or ALL) cannot include GROUP BY and HAVING clauses.
- The DISTINCT keyword cannot be used with subqueries that include GROUP BY.
- The COMPUTE and INTO clauses cannot be specified.
- ORDER BY can only be specified if TOP is also specified.
- A view created with a subquery cannot be updated.
- The select list of a subquery introduced with EXISTS by convention consists of an asterisk (*) instead of a single column name. The rules for a subquery introduced with EXISTS are identical to those for a standard select list because a subquery introduced with EXISTS constitutes an

existence test and returns TRUE or FALSE, rather than data.

Qualifying Column Names in Subqueries

In the following example, the **pub_id** column in the WHERE clause of the outer query is implicitly qualified by the table name in the outer query's FROM clause, **publishers**. The reference to **pub_id** in the select list of the subquery is qualified by the subquery's FROM clause, that is, by the **titles** table.

```
USE pubs
SELECT pub_name
FROM publishers
WHERE pub_id NOT IN
(SELECT pub_id
FROM titles
WHERE type = 'business'
```

The general rule is that column names in a statement are implicitly qualified by the table referenced in the FROM clause at the same level.

Here's what the query looks like with these implicit assumptions specified:

```
USE pubs
SELECT pub_name
FROM publishers
WHERE publishers.pub_id NOT IN
(SELECT titles.pub_id
FROM titles
WHERE type = 'business')
```

It is never wrong to state the table name explicitly, and it is always possible to override implicit assumptions about table names with explicit qualifications.

```
See Also
```

<u>FROM</u>

<u>WHERE</u>

Subquery Types

Subqueries can be specified in many places:

- With aliases
- With IN or NOT IN
- In UPDATE, DELETE, and INSERT statements
- With comparison operators
- With ANY, SOME, or ALL
- With EXISTS or NOT EXISTS
- In place of an expression

Subqueries with Aliases

Many statements in which the subquery and the outer query refer to the same table can be stated as self-joins (joining a table to itself). For example, you can find authors who live in the same city as Livia Karsen by using a subquery:

USE pubs SELECT au_lname, au_fname, city FROM authors WHERE city IN (SELECT city FROM authors WHERE au_fname = 'Livia' AND au_lname = 'Karsen')

Here is the result set:

au_lname	au_fname	city
Green	Marjorie	Oakland
Straight	Dean	Oakland
Stringer	Dirk	Oakland
MacFeather	Stearns	Oakland
Karsen	Livia	Oakland

(5 row(s) affected)

Or you can use a self-join:

USE pubs SELECT au1.au_lname, au1.au_fname, au1.city FROM authors AS au1 INNER JOIN authors AS au2 ON au1.city = au AND au2.au_lname = 'Karsen' AND au2.au_fname = 'Livia'

Table aliases are required because the table being joined to itself appears in two different roles. Aliases can also be used in nested queries that refer to the same table in an inner and outer query.

USE pubs SELECT au1.au_lname, au1.au_fname, au1.city FROM authors AS au1 WHERE au1.city in (SELECT au2.city FROM authors AS au2 WHERE au2.au_fname = 'Livia' AND au2.au_lname = 'Karsen')

Explicit aliases make it clear that reference to **authors** in the subquery does not mean the same thing as the reference in the outer query.

Subqueries with IN

The result of a subquery introduced with IN (or with NOT IN) is a list of zero or more values. After the subquery returns results, the outer query makes use of them.

This query finds the names of the publishers who have published business books.

```
USE pubs
SELECT pub_name
FROM publishers
WHERE pub_id IN
(SELECT pub_id
FROM titles
WHERE type = 'business')
```

Here is the result set:

pub_name

Algodata Infosystems New Moon Books

(2 row(s) affected)

This statement is evaluated in two steps. First, the inner query returns the identification numbers of the publishers that have published business books (1389 and 0736). Second, these values are substituted into the outer query, which finds the names that go with the identification numbers in **publishers**.

USE pubs SELECT pub_name FROM publishers

WHERE pub_id in ('1389', '0736')

One difference in using a join rather than a subquery for this and similar problems is that the join lets you show columns from more than one table in the result. For example, if you want to include the titles of the business books in the result, you must use a join version.

```
USE pubs
SELECT pub_name, title
FROM publishers INNER JOIN titles ON publishers.pub_id = titles.pu
AND type = 'business'
```

Here is the result set:

pub_name title	e
8	The Busy Executive's Database Guide Cooking with Computers: Surreptitious Balanc
New Moon Books Algodata Infosystems	You Can Combat Computer Stress! Straight Talk About Computers

```
(4 row(s) affected)
```

This query shows the join produces four rows, not two as in the preceding subquery.

Here is another example of a query that can be formulated with either a subquery or a join. This query finds the names of all second authors who live in California and who receive less than 30 percent of the royalties for a book.

USE pubs SELECT au_lname, au_fname FROM authors WHERE state = 'CA' AND au_id IN (SELECT au_id FROM titleauthor WHERE royaltyper < 30 AND au_ord = 2)

Here is the result set:

au_lnameau_fname------------MacFeatherStearns

(1 row(s) affected)

The inner query is evaluated, producing the ID numbers of the three authors who meet the subquery qualifications. The outer query is then evaluated. Notice that you can include more than one condition in the WHERE clause of both the inner and the outer query.

Using a join, the same query is expressed like this:

```
USE pubs
SELECT au_lname, au_fname
FROM authors INNER JOIN titleauthor ON authors.au_id = titleautho
WHERE state = 'CA'
AND royaltyper < 30
AND au_ord = 2
```

A join can always be expressed as a subquery. A subquery can often, but not always, be expressed as a join. This is because joins are symmetric: you can join table A to B in either order and get the same answer. The same is not true if a subquery is involved.

Subqueries with NOT IN

Subqueries introduced with the keyword NOT IN also return a list of zero or more values.

This query finds the names of the publishers who have not published business books.

```
USE pubs
SELECT pub_name
FROM publishers
WHERE pub_id NOT IN
(SELECT pub_id
FROM titles
WHERE type = 'business')
```

The query is exactly the same as the one in Subqueries with IN, except that NOT IN is substituted for IN. However, this statement cannot be converted to a join. The analogous not-equal join has a different meaning: It finds the names of publishers who have published some book that is not a business book. For information about interpreting the meaning of joins not based on equality, see Joining Three or More Tables.

Subqueries in UPDATE, DELETE, and INSERT Statements

Subqueries can be nested in UPDATE, DELETE, and INSERT statements, as well as in SELECT statements.

The following query doubles the price of all books published by New Moon Books. The query updates the **titles** table; its subquery references the **publishers** table.

```
UPDATE titles
SET price = price * 2
WHERE pub_id IN
(SELECT pub_id
FROM publishers
WHERE pub_name = 'New Moon Books')
```

Here's an equivalent UPDATE statement using a join:

```
UPDATE titles
SET price = price * 2
FROM titles INNER JOIN publishers ON titles.pub_id = publishers.pu
AND pub_name = 'New Moon Books'
```

You can remove all sales records of business books with this nested query:

```
DELETE sales
WHERE title_id IN
(SELECT title_id
FROM titles
WHERE type = 'business')
```

Here's an equivalent DELETE statement using a join:

DELETE sales

FROM sales INNER JOIN titles ON sales.title_id = titles.title_id AND type = 'business'

Subqueries with Comparison Operators

Subqueries can be introduced with one of the comparison operators (=, < >, >, > =, <, ! >, ! <, or < =).

A subquery introduced with an unmodified comparison operator (a comparison operator not followed by ANY or ALL) must return a single value rather than a list of values, like subqueries introduced with IN. If such a subquery returns more than one value, Microsoft® SQL ServerTM displays an error message.

To use a subquery introduced with an unmodified comparison operator, you must be familiar enough with your data and with the nature of the problem to know that the subquery will return exactly one value.

For example, if you assume each publisher is located in only one city, and you want to find the names of authors who live in the city in which Algodata Infosystems is located, you can write a statement with a subquery introduced with the simple = comparison operator.

```
USE pubs
SELECT au_lname, au_fname
FROM authors
WHERE city =
(SELECT city
FROM publishers
WHERE pub_name = 'Algodata Infosystems')
```

Here is the result set:

au_lname au_fname ------Carson Cheryl Bennet Abraham

(2 row(s) affected)

If, however, Algodata Infosystems was located in multiple cities, then an error message would result. Instead of the = comparison operator, an IN formulation could be used (= ANY also works).

Subqueries introduced with unmodified comparison operators often include aggregate functions, because these return a single value. For example, this statement finds the names of all books priced higher than the current minimum price.

USE pubs SELECT DISTINCT title FROM titles WHERE price > (SELECT MIN(price) FROM titles)

Here is the result set:

title

But Is It User Friendly? Computer Phobic and Non-Phobic Individuals: Behavior Variations Cooking with Computers: Surreptitious Balance Sheets Emotional Security: A New Algorithm Fifty Years in Buckingham Palace Kitchens Is Anger the Enemy? Life Without Fear Onions, Leeks, and Garlic: Cooking Secrets of the Mediterranean Prolonged Data Deprivation: Four Case Studies Secrets of Silicon Valley Silicon Valley Gastronomic Treats Straight Talk About Computers Sushi, Anyone? The Busy Executive's Database Guide

(14 row(s) affected)

Because subqueries introduced with unmodified comparison operators must return a single value, they cannot include GROUP BY or HAVING clauses unless you know the GROUP BY or HAVING clause itself returns a single value. For example, this query finds the books priced higher than the lowest priced book that has a type '**trad_cook**'.

USE pubs SELECT DISTINCT title FROM titles WHERE price > (SELECT MIN(price) FROM titles GROUP BY type HAVING type = 'trad_cook')

Here is the result set:

title

But Is It User Friendly?

Computer Phobic AND Non-Phobic Individuals: Behavior Variations Onions, Leeks, and Garlic: Cooking Secrets of the Mediterranean Prolonged Data Deprivation: Four Case Studies Secrets of Silicon Valley Silicon Valley Gastronomic Treats Straight Talk About Computers Sushi, Anyone? The Busy Executive's Database Guide

(9 row(s) affected)

Comparison Operators Modified by ANY, SOME, or ALL

Comparison operators that introduce a subquery can be modified by the keywords ALL or ANY. SOME is an SQL-92 standard equivalent for ANY.

Subqueries introduced with a modified comparison operator return a list of zero or more values and can include a GROUP BY or HAVING clause. These subqueries can be restated with EXISTS.

Using the > comparison operator as an example, >ALL means greater than every value--in other words, greater than the maximum value. For example, >ALL (1, 2, 3) means greater than 3. >ANY means greater than at least one value, that is, greater than the minimum. So >ANY (1, 2, 3) means greater than 1.

For a row in a subquery with >ALL to satisfy the condition specified in the outer query, the value in the column introducing the subquery must be greater than each value in the list of values returned by the subquery.

Similarly, >ANY means that for a row to satisfy the condition specified in the outer query, the value in the column that introduces the subquery must be greater than at least one of the values in the list of values returned by the subquery.

Note This example can be run many different ways, as long as the inner query returns only one value.

```
USE pubs

-- Option 1 using MAX in the inner query

SELECT title

FROM titles

HAVING MAX(advance) > ALL

WHERE advance > ALL

(

SELECT MAX(advance)

FROM publishers INNER JOIN titles ON

titles.pub_id = publishers.pub_id
```

```
WHERE pub_name = 'Algodata Infosystems'
)
```

```
-- Option 2 using GROUP BY and HAVING and no ALL
USE pubs
SELECT title
FROM titles
GROUP BY title
HAVING MAX(advance) >
(
SELECT MAX(advance)
FROM publishers INNER JOIN titles ON
titles.pub_id = publishers.pub_id
WHERE pub_name = 'Algodata Infosystems'
)
```

The following query provides an example of a subquery introduced with a comparison operator modified by ANY. It finds the titles that received an advance larger than the minimum advance amount paid by Algodata Infosystems.

```
USE pubs
SELECT title
FROM titles
WHERE advance > ANY
(SELECT advance
FROM publishers INNER JOIN titles
ON titles.pub_id = publishers.pub_id
AND pub_name = 'Algodata Infosystems')
```

Here is the result set:

title

You Can Combat Computer Stress!

The Gourmet Microwave But Is It User Friendly? Secrets of Silicon Valley Computer Phobic and Non-Phobic Individuals: Behavior Variations Life Without Fear Onions, Leeks, and Garlic: Cooking Secrets of the Mediterranean Sushi, Anyone?

(8 row(s) affected)

For each title, the inner query finds a list of advance amounts paid by Algodata. The outer query looks at all values in the list and determines whether the title currently being considered has commanded an advance larger than any of those amounts. In other words, it finds titles with advances as large or larger than the lowest value paid by Algodata.

If the subquery does not return any values, the entire query fails to return any values.

The =ANY operator is equivalent to IN. For example, to find authors who live in the same city as a publisher, you can use either IN or =ANY.

USE pubs SELECT au_lname, au_fname FROM authors WHERE city IN (SELECT city FROM publishers)

USE pubs SELECT au_lname, au_fname FROM authors WHERE city = ANY (SELECT city FROM publishers) Here is the result set for either query:

au_lnameau_fname----------CarsonCherylBennetAbraham

(2 row(s) affected)

The < >ANY operator, however, differs from NOT IN: < >ANY means not = a, or not = b, or not = c. NOT IN means not = a, and not = b, and not = c. <>ALL means the same as NOT IN.

For example, this query finds the authors who live in a city in which no publisher is located.

USE pubs SELECT au_lname, au_fname FROM authors WHERE city <> ANY (SELECT city FROM publishers)

Here is the result set:

au_lname	au_fname
White	Johnson
Green	Marjorie
Carson	Cheryl
O'Leary	Michael
Straight	Dean
Smith	Meander
Bennet	Abraham
Della Buena	Ann
Gringlesby	Burt

Locksley	Charlene
Greene	Morningstar
Blotchet-Halls	Reginald
Yokomoto	Akiko
del Covello	Innes
DeFrance	Michel
Stringer	Dirk
MacFeather	Stearns
Karsen	Livia
Panteley	Sylvia
Hunter	Sheryl
McBadden	Heather
Ringer	Anne
Ringer	Albert

(23 row(s) affected)

The results include all 23 authors because every author lives in a city in which one or more of the publishers is not located. The inner query finds all the cities in which publishers are located, and then, for each city, the outer query finds the authors who don't live there.

However, when you use NOT IN in this query, the results include all the authors except Cheryl Carson and Abraham Bennet, who live in Berkeley, where Algodata Infosystems is located.

USE pubs SELECT au_lname, au_fname FROM authors WHERE city NOT IN (SELECT city FROM publishers)

Here is the result set:

au_lname

White	Johnson
Green	Marjorie
O'Leary	Michael
Straight	Dean
Smith	Meander
Della Buena	Ann
Gringlesby	Burt
Locksley	Charlene
Greene	Morningstar
Blotchet-Halls	Reginald
Yokomoto	Akiko
del Covello	Innes
DeFrance	Michel
Stringer	Dirk
MacFeather	Stearns
Karsen	Livia
Panteley	Sylvia
Hunter	Sheryl
McBadden	Heather
Ringer	Anne
Ringer	Albert

(21 row(s) affected)

You can get the same results with the < >ALL operator, which is equivalent to NOT IN.

USE pubs SELECT au_lname, au_fname FROM authors WHERE city <> ALL (SELECT city FROM publishers)

Subqueries with EXISTS

When a subquery is introduced with the keyword EXISTS, it functions as an existence test. The WHERE clause of the outer query tests for the existence of rows returned by the subquery. The subquery does not actually produce any data; it returns a value of TRUE or FALSE.

A subquery introduced with EXISTS has the following syntax:

WHERE [NOT] EXISTS (subquery)

This query finds the names of all publishers who publish business books:

USE pubs SELECT pub_name FROM publishers WHERE EXISTS (SELECT * FROM titles WHERE pub_id = publishers.pub_id AND type = 'business')

Here is the result set:

pub_name

-

New Moon Books Algodata Infosystems

(2 row(s) affected)

To determine the results of this query, consider each publisher's name in turn. Does this value cause the subquery to return at least one row? In other words, does it cause the existence test to evaluate to TRUE?

In this case, the first publisher name is Algodata Infosystems, with identification

number 1389. Are there any rows in the **titles** table in which **pub_id** is 1389 and **type** is business? If so, Algodata Infosystems should be one of the values selected. The same process is repeated for each of the other publisher names.

Notice that subqueries introduced with EXISTS are a bit different from other subqueries in these ways:

- The keyword EXISTS is not preceded by a column name, constant, or other expression.
- The select list of a subquery introduced by EXISTS almost always consists of an asterisk (*). There is no reason to list column names because you are simply testing for the existence of rows that meet the conditions specified in the subquery.

The EXISTS keyword is important because often there is no alternative, nonsubquery formulation. Although some queries formulated with EXISTS cannot be expressed any other way, all queries that use IN or a comparison operator modified by ANY or ALL can be expressed with EXISTS.

Examples of queries using EXISTS and equivalent alternatives follow.

Here are two ways to find authors who live in the same city as a publisher:

```
USE pubs
SELECT au_lname, au_fname
FROM authors
WHERE city =ANY
(SELECT city
FROM publishers)
-- Or
USE pubs
SELECT au_lname, au_fname
FROM authors
WHERE exists
(SELECT *
FROM publishers
```

WHERE authors.city = publishers.city)

Here is the result set for either query:

au_lname au_fname ------Carson Cheryl Bennet Abraham

(2 row(s) affected)

These two queries find titles of books published by any publisher located in a city that begins with the letter B:

USE pubs SELECT title **FROM** titles WHERE pub_id IN (SELECT pub_id FROM publishers WHERE city LIKE 'B%') -- Or USE pubs SELECT title FROM titles WHERE EXISTS (SELECT * FROM publishers WHERE pub_id = titles.pub_id AND city LIKE 'B%')

Here is the result set for either query:

title

The Busy Executive's Database Guide Cooking with Computers: Surreptitious Balance Sheets You Can Combat Computer Stress! Straight Talk About Computers But Is It User Friendly? Secrets of Silicon Valley Net Etiquette Is Anger the Enemy? Life Without Fear Prolonged Data Deprivation: Four Case Studies Emotional Security: A New Algorithm

(11 row(s) affected)

See Also

EXISTS

Subqueries with NOT EXISTS

NOT EXISTS works like EXISTS, except the WHERE clause in which it is used is satisfied if no rows are returned by the subquery.

For example, to find the names of publishers who do not publish business books:

USE pubs SELECT pub_name FROM publishers WHERE NOT EXISTS (SELECT * FROM titles WHERE pub_id = publishers.pub_id AND type = 'business')

Here is the result set:

pub_name

Binnet & Hardley Five Lakes Publishing Ramona Publishers GGG&G Scootney Books Lucerne Publishing

(6 row(s) affected)

This query finds the titles for which there have been no sales.

USE pubs SELECT title FROM titles WHERE NOT EXISTS (SELECT title_id FROM sales WHERE title_id = titles.title_id)

Here is the result set:

title

The Psychology of Computer Cooking Net Etiquette

(2 row(s) affected)

See Also

EXISTS

<u>NOT</u>

Using EXISTS and NOT EXISTS to Find Intersection and Difference

Subqueries introduced with EXISTS and NOT EXISTS can be used for two settheory operations: intersection and difference. The intersection of two sets contains all elements that belong to both of the original sets. The difference contains elements that belong only to the first of the two sets.

The intersection of **authors** and **publishers** over the **city** column is the set of cities in which both an author and a publisher are located.

USE pubs SELECT DISTINCT city FROM authors WHERE EXISTS (SELECT * FROM publishers WHERE authors.city = publishers.city)

Here is the result set:

city

Berkeley

```
(1 row(s) affected)
```

Of course, this query could be written as a simple join.

USE pubs SELECT DISTINCT authors.city FROM authors INNER JOIN publishers ON authors.city = publishers.city The difference between **authors** and **publishers** over the **city** column is the set of cities where an author lives but no publisher is located, that is, all the cities except Berkeley.

USE pubs SELECT DISTINCT city FROM authors WHERE NOT EXISTS (SELECT * FROM publishers WHERE authors.city = publishers.city)

This query could also be written as:

USE pubs SELECT DISTINCT city FROM authors WHERE city NOT IN (SELECT city FROM publishers)

Subqueries Used in Place of an Expression

In Transact-SQL, a subquery can be substituted anywhere an expression can be used in SELECT, UPDATE, INSERT, and DELETE statements, except in an ORDER BY list.

The following example illustrates how you might use this enhancement. This query finds the price of a popular computer book, the average price of all books, and the difference between the price of the book and the average price of all books.

```
USE pubs
SELECT title, price,
(SELECT AVG(price) FROM titles) AS average,
price-(SELECT AVG(price) FROM titles) AS difference
FROM titles
WHERE type='popular_comp'
```

Here is the result set:

title	price	average	e dif	fference	
But Is It User Fr	tiendly? 22	2.95	14.77	8.18	
Secrets of Silico	n Valley 2	0.00	14.77	5.23	
Net Etiquette	(null)	14.	77	(null)	

(3 row(s) affected)

Multiple Levels of Nesting

A subquery can itself include one or more subqueries. Any number of subqueries can be nested in a statement.

This query finds the names of authors who have participated in writing at least one popular computer book.

USE pubs
SELECT au_lname, au_fname
FROM authors
WHERE au_id IN
(SELECT au_id
FROM titleauthor
WHERE title_id IN
(SELECT title_id
FROM titles
WHERE type = 'popular_comp'))

Here is the result set:

Carson Cheryl	au_lname
DullAnnLocksleyCharleneHunterSheryl	Dull Locksley

(4 row(s) affected)

The innermost query returns the title ID numbers PC1035, PC8888, and PC9999. The query at the next higher level is evaluated with these title IDs and returns the author ID numbers. Finally, the outer query uses the author IDs to find the names of the authors.

You can also express this query as a join:

USE pubs SELECT au_lname, au_fname FROM authors INNER JOIN titleauthor ON authors.au_id = titleautho JOIN titles ON titleauthor.title_id = titles.title_id WHERE type = 'popular_comp'

Correlated Subqueries

Many queries can be evaluated by executing the subquery once and substituting the resulting value or values into the WHERE clause of the outer query. In queries that include a <u>correlated subquery</u> (also known as a repeating subquery), the subquery depends on the outer query for its values. This means that the subquery is executed repeatedly, once for each row that might be selected by the outer query.

This query finds the names of all authors who earn 100 percent of the shared royalty (**royaltyper**) on a book.

USE pubs SELECT au_lname, au_fname FROM authors WHERE 100 IN (SELECT royaltyper FROM titleauthor WHERE titleauthor.au_ID = authors.au_id)

au fname au lname White Johnson Green Marjorie Carson Cheryl Straight Dean Locksley Charlene **Blotchet-Halls** Reginald del Castillo Innes **Sylvia** Panteley Albert Ringer

Here is the result set:

(9 row(s) affected)

Unlike most of the subqueries shown earlier, the subquery in this statement cannot be resolved independently of the main query. It needs a value for **authors.au_id**, but this value is a variable. It changes as Microsoft® SQL ServerTM examines different rows of the **authors** table.

That is exactly how this query is evaluated: SQL Server considers each row of the **authors** table for inclusion in the results by substituting the value in each row into the inner query. For example, if SQL Server first examines the row for Cheryl Carson, the variable **authors.au_id** takes the value 238-95-7766, which SQL Server substitutes into the inner query.

USE pubs SELECT royaltyper FROM titleauthor WHERE au_id = '238-95-7766'

The result is 100, so the outer query evaluates to:

USE pubs SELECT au_lname, au_fname FROM authors WHERE 100 IN (100)

Because this is true, the row for Cheryl Carson is included in the results. Go through the same procedure with the row for Abraham Bennet; you'll see that this row is not included in the results.

Correlated Subqueries with Aliases

Correlated subqueries can be used in operations such as selecting data from a table referenced in the outer query. In this case a table alias (also called a correlation name) must be used to specify unambiguously which table reference to use. For example, you can use a correlated subquery to find the types of books published by more than one publisher. Aliases are required to distinguish the two different roles in which the **titles** table appears.

USE pubs SELECT DISTINCT t1.type FROM titles t1 WHERE t1.type IN (SELECT t2.type FROM titles t2 WHERE t1.pub_id <> t2.pub_id)

Here is the result set:

type

business psychology

(2 row(s) affected)

The preceding nested query is equivalent to this self-join:

```
USE pubs
SELECT DISTINCT t1.type
FROM titles t1 INNER JOIN titles t2 ON t1.type = t2.type
AND t1.pub_id <> t2.pub_id
```

Correlated Subqueries with Comparison Operators

Use a correlated subquery with a comparison operator to find sales where the quantity is less than the average quantity for sales of that title.

```
USE pubs
SELECT s1.ord_num, s1.title_id, s1.qty
FROM sales s1
WHERE s1.qty <
(SELECT AVG(s2.qty)
FROM sales s2
WHERE s2.title_id = s1.title_id)
```

Here is the result set:

ord_num	title_id qty
6871	BU1032 5
722a	PS2091 3
D4482	PS2091 10
N914008	PS2091 20
423LL922	MC3021 15

(5 row(s) affected)

The outer query selects the rows of **sales** (that is, of **s1**) one by one. The subquery calculates the average quantity for each sale being considered for selection in the outer query. For each possible value of **s1**, Microsoft® SQL ServerTM evaluates the subquery and puts the record being considered in the results if the quantity is less than the calculated average.

Sometimes a correlated subquery mimics a GROUP BY clause. This example finds all titles that have a price greater than the average for books of its type.

USE pubs SELECT t1.type, t1.title FROM titles t1 WHERE t1.price > (SELECT AVG(t2.price) FROM titles t2 WHERE t1.type = t2.type)

Here is the result set:

type title

business The Busy Executive's Database Guide
 business Straight Talk About Computers
 mod_cook Silicon Valley Gastronomic Treats
 popular_comp But Is It User Friendly?
 psychology Computer Phobic AND Non-Phobic Individuals: Behavic Variations
 psychology Prolonged Data Deprivation: Four Case Studies
 trad_cook Onions, Leeks, and Garlic: Cooking Secrets of the Mediterranean

(7 row(s) affected)

For each possible value of **t1**, SQL Server evaluates the subquery and includes the row in the results if the **price** value of that row is greater than the calculated average. It is not necessary to group by type explicitly, because the rows for which average price is calculated are restricted by the WHERE clause in the subquery.

Correlated Subqueries in a HAVING Clause

A correlated subquery can also be used in the HAVING clause of an outer query. This construction can be used to find the types of books for which the maximum advance is more than twice the average within a given group.

In this case, the subquery is evaluated once for each group defined in the outer query (once for each type of book).

```
USE pubs
SELECT t1.type
FROM titles t1
GROUP BY t1.type
HAVING MAX(t1.advance) >=ALL
(SELECT 2 * AVG(t2.advance)
FROM titles t2
WHERE t1.type = t2.type)
```

Here is the result set:

type

mod_cook

(1 row(s) affected)

Conditional Data Processing Using CASE

The CASE function is used to evaluate several conditions and return a single value for each condition. A common use of the CASE function is to replace codes or abbreviations with more readable values. The following query uses the CASE function to rename book categories so that they are more understandable.

USE pubs SELECT CASE type WHEN 'popular_comp' THEN 'Popular Computing' WHEN 'mod_cook' THEN 'Modern Cooking' WHEN 'business' THEN 'Business' WHEN 'psychology' THEN 'Psychology' WHEN 'psychology' THEN 'Psychology' WHEN 'trad_cook' THEN 'Traditional Cooking' ELSE 'Not yet categorized' END AS Category, CONVERT(varchar(30), title) AS "Shortened Title", price AS Price FROM titles WHERE price IS NOT NULL ORDER BY 1

Here is the result set:

category	shortened title	Price	
Business	Cooking with Co	mputers: Surre	р 11.95
Business	Straight Talk Ab	out Computers	19.99
Business	The Busy Executive's Database 19.99		19.99
Business	You Can Combat	t Computer Stre	ess 2.99
Modern Cookir	ng Silicon Valle	ey Gastronomic	Tre 19.99
Modern Cookir	ng The Gourme	et Microwave	2.99
Popular Compu	iting But Is It Us	er Friendly?	22.95

Popular Computing Secrets of Silicon Valley 20.00Psychology Computer Phobic AND Non-Phobic 21.59 Psychology Emotional Security: A New Algo 7.99 Psychology Is Anger the Enemy? 10.95 Psychology Life Without Fear 7.00 Psychology Prolonged Data Deprivation: Fo 19.99 Traditional Cooking Fifty Years in Buckingham Pala 11.95 Traditional Cooking Onions, Leeks, and Garlic: Coo 20.95 Traditional Cooking Sushi, Anyone? 14.99

```
(16 row(s) affected)
```

Another use of CASE is to categorize data. The following query uses the CASE function to categorize prices.

SELECT CASE WHEN price IS NULL THEN 'Not yet priced' WHEN price < 10 THEN 'Very Reasonable Title' WHEN price >= 10 and price < 20 THEN 'Coffee Table Title' ELSE 'Expensive book!' END AS "Price Category", CONVERT(varchar(20), title) AS "Shortened Title" FROM pubs.dbo.titles ORDER BY price

Here is the result set:

Price Category Shortened Title

Not yet pricedThe Psychology of CoNot yet pricedNet EtiquetteVery ReasonableTitle You Can Combat CompuVery ReasonableTitle The Gourmet MicrowavVery ReasonableTitle Life Without Fear

Very Reasonable Title Emotional Security: Coffee Table Title Is Anger the Enemy? Coffee Table Title Cooking with Compute Coffee Table Title Fifty Years in Bucki Coffee Table Title Sushi, Anyone? Coffee Table Title The Busy Executive's Coffee Table Title Straight Talk About Coffee Table Title Silicon Valley Gastr Coffee Table Title Prolonged Data Depri Secrets of Silicon V **Expensive book!** Expensive book! Onions, Leeks, and G **Expensive book! Computer Phobic AND** Expensive book! But Is It User Frien

(18 row(s) affected)

See Also

<u>CASE</u>

Summarizing Data

Producing summary reports of aggregated transaction data for decision support systems can be a complex and resource-intensive operation. Microsoft® SQL Server[™] 2000 provides two flexible and powerful components for building SQL Server 2000 Analysis Services. These components are the main tools programmers should use in performing multidimensional analysis of SQL Server data:

• Data Transformation Services (DTS)

DTS supports extracting transaction data and transforming it into summary aggregates in a data warehouse or data mart. For more information, see <u>DTS Overview</u>.

• Microsoft SQL Server Analysis Services

Analysis Services organizes data from a data warehouse into multidimensional cubes with precalculated summary information to provide rapid answers to complex analytical queries. PivotTable® Service provides client access to multidimensional data. Analysis Services also provides a set of wizards for defining the multidimensional structures used in the Analysis processing, and a Microsoft Management Console snap-in for administering the Analysis structures. Applications can then use either the OLE DB for Analysis API or the Microsoft ActiveX® Data Objects (Multidimensional) (ADO MD) API to analyze the Analysis data. For more information, see <u>Analysis Services Overview</u>.

Using Transact-SQL for Simple Summary Reports

Applications generating simple summary reports can use these Transact-SQL elements:

• The CUBE or ROLLUP operators, which are both part of the GROUP BY clause of the SELECT statement.

• The COMPUTE or COMPUTE BY operators, which are also associated with GROUP BY.

These operators generate result sets that contain both detail rows for each item in the result set and summary rows for each group showing the aggregate totals for that group. The GROUP BY clause can be used to generate results that contain aggregates for each group, but no detail rows.

It is recommended that applications use Analysis Services instead of CUBE, ROLLUP, COMPUTE, or COMPUTE BY. CUBE and ROLLUP should be reserved for environments that do not have access to OLE DB or ADO, such as scripts or stored procedures.

COMPUTE and COMPUTE BY are supported for backward compatibility. The ROLLUP operator is preferred over either COMPUTE or COMPUTE BY. The summary values generated by COMPUTE or COMPUTE BY are returned as separate result sets interleaved with the result sets returning the detail rows for each group, or a result set containing the totals appended after the main result set. Handling these multiple result sets increases the complexity of the code in an application. Neither COMPUTE nor COMPUTE BY are supported with server cursors, and ROLLUP is. CUBE and ROLLUP generate a single result set containing embedded subtotal and total rows. The query optimizer can also sometimes generate more efficient execution plans for ROLLUP than it can for COMPUTE and COMPUTE BY.

When GROUP BY is used without these operators, it returns a single result set with one row per group containing the aggregate subtotals for the group. There are no detail rows in the result set.

Summarizing Data Using CUBE

The CUBE operator generates a result set that is a multidimensional cube. A multidimensional cube is an expansion of fact data, or data that records individual events. The expansion is based on columns that the user wants to analyze. These columns are called dimensions. The cube is a result set containing a cross tabulation of all the possible combinations of the dimensions.

The CUBE operator is specified in the GROUP BY clause of a SELECT statement. The select list contains the dimension columns and aggregate function expressions. The GROUP BY specifies the dimension columns and the keywords WITH CUBE. The result set contains all possible combinations of the values in the dimension columns, along with the aggregate values from the underlying rows that match that combination of dimension values.

Item	Color	Quantity
Table	Blue	124
Table	Red	223
Chair	Blue	101
Chair	Red	210

For example, a simple table **Inventory** contains:

This query returns a result set that contains the **Quantity** subtotal for all possible combinations of **Item** and **Color**:

SELECT Item, Color, SUM(Quantity) AS QtySum FROM Inventory GROUP BY Item, Color WITH CUBE

Here is the result set:

Item	Color	QtySum
Chair	Blue	101.00

Chair	Red	210.00
Chair	(null)	311.00
Table	Blue	124.00
Table	Red	223.00
Table	(null)	347.00
(null)	(null)	658.00
(null)	Blue	225.00
(null)	Red	433.00

The following rows from the result set are of special interest:

311.00

This row reports a subtotal for all rows having the value Chair in the **Item** dimension. The value NULL is returned for the **Color** dimension to show that aggregate reported by the row includes rows with any value of the **Color** dimension.

Table	(null)	347.00
-------	--------	--------

This row is similar, but reports the subtotal for all rows having Table in the **Item** dimension.

(null) (null)	658.00
---------------	--------

This row reports the grand total for the cube. Both the **Item** and **Color** dimensions have the value NULL showing that all values of both dimensions are summarized in the row.

(null)	Blue	225.00
(null)	Red	433.00

These two rows report the subtotals for the **Color** dimension. Both have NULL in the **Item** dimension to show that the aggregate data came from rows having any value for the **Item** dimension.

Using GROUPING to Distinguish Null Values

The null values generated by the CUBE operation present a problem: How can a NULL generated by the CUBE operation be distinguished from a NULL returned in the actual data? This is achieved using the GROUPING function. The GROUPING function returns 0, if the column value came from the fact data, and 1 if the column value is a NULL generated by the CUBE operation. In a CUBE operation, a generated NULL represents all values. The SELECT statement can be written to use the GROUPING function to substitute the string ALL in place of any generated NULL. Because a NULL from the fact data indicates the data value is unknown, the SELECT can also be coded to return the string UNKNOWN in place of any NULL from the fact data. For example:

SELECT CASE WHEN (GROUPING(Item) = 1) THEN 'ALL' ELSE ISNULL(Item, 'UNKNOWN') END AS Item, CASE WHEN (GROUPING(Color) = 1) THEN 'ALL' ELSE ISNULL(Color, 'UNKNOWN') END AS Color, SUM(Quantity) AS QtySum FROM Inventory GROUP BY Item, Color WITH CUBE

Multidimensional Cubes

The CUBE operator can be used to generate *n*-dimensional cubes, or cubes with any number of dimensions. A single dimension cube can be used to generate a total, for example:

SELECT CASE WHEN (GROUPING(Item) = 1) THEN 'ALL' ELSE ISNULL(Item, 'UNKNOWN') END AS Item, SUM(Quantity) AS QtySum FROM Inventory GROUP BY Item WITH CUBE GO

This SELECT statement returns a result set showing both the subtotals for each

value of **Item** and the grand total for all values of **Item**:

Item	QtySum	
Chair	311.00	
Table	347.00	
ALL	658.00	

SELECT statements that contain a CUBE with many dimensions can generate large result sets, because these statements generate rows for all combinations of the values in all the dimensions. These large result sets may contain too much data to be easily read and understood. One solution to this problem is to put the SELECT statement into a view:

```
CREATE VIEW InvCube AS
SELECT CASE WHEN (GROUPING(Item) = 1) THEN 'ALL'
ELSE ISNULL(Item, 'UNKNOWN')
END AS Item,
CASE WHEN (GROUPING(Color) = 1) THEN 'ALL'
ELSE ISNULL(Color, 'UNKNOWN')
END AS Color,
SUM(Quantity) AS QtySum
FROM Inventory
GROUP BY Item, Color WITH CUBE
```

The view can then be used to query only the dimension values of interest:

```
SELECT *
FROM InvCube
WHERE Item = 'Chair'
AND Color = 'ALL'
```

Item	Color	QtySum
Chair	ALL	311.00

(1 row(s) affected)

See Also

<u>SELECT</u>

Summarizing Data Using ROLLUP

Summarizing Data Using ROLLUP

The ROLLUP operator is useful in generating reports that contain subtotals and totals. The ROLLUP operator generates a result set that is similar to the result sets generated by the CUBE operator. For more information, see <u>Summarizing Data Using CUBE</u>.

The differences between CUBE and ROLLUP are:

- CUBE generates a result set showing aggregates for all combinations of values in the selected columns.
- ROLLUP generates a result set showing aggregates for a hierarchy of values in the selected columns.

For example, a simple table **Inventory** contains:

Item	Color	Quantity
Table	Blue	124
Table	Red	223
Chair	Blue	101
Chair	Red	210

This query generates a subtotal report:

SELECT CASE WHEN (GROUPING(Item) = 1) THEN 'ALL' ELSE ISNULL(Item, 'UNKNOWN') END AS Item, CASE WHEN (GROUPING(Color) = 1) THEN 'ALL' ELSE ISNULL(Color, 'UNKNOWN') END AS Color, SUM(Quantity) AS QtySum FROM Inventory GROUP BY Item, Color WITH ROLLUP

Item	Color	QtySum
Chair	Blue	101.00
Chair	Red	210.00
Chair	ALL	311.00
Table	Blue	124.00
Table	Red	223.00
Table	ALL	347.00
ALL	ALL	658.00

(7 row(s) affected)

If the ROLLUP keyword in the query is changed to CUBE, the CUBE result set is the same, except these two additional rows are returned at the end:

ALL	Blue	225.00
ALL	Red	433.00

The CUBE operation generated rows for possible combinations of values from both **Item** and **Color**. For example, not only does CUBE report all possible combinations of **Color** values combined with the **Item** value Chair (Red, Blue, and Red + Blue), it also reports all possible combinations of **Item** values combined with the **Color** value Red (Chair, Table, and Chair + Table).

For each value in the columns on the right in the GROUP BY clause, the ROLLUP operation does not report all possible combinations of values from the column (or columns) on the left. For example, ROLLUP does not report all the possible combinations of **Item** values for each **Color** value.

The result set of a ROLLUP operation has functionality similar to that returned by a COMPUTE BY; however, ROLLUP has these advantages:

- ROLLUP returns a single result set; COMPUTE BY returns multiple result sets that increase the complexity of application code.
- ROLLUP can be used in a server cursor; COMPUTE BY cannot.

• The query optimizer can sometimes generate more efficient execution plans for ROLLUP than it can for COMPUTE BY.

See Also

SELECT

Summarizing Data Using COMPUTE and COMPUTE BY

Summarizing Data Using COMPUTE and COMPUTE BY

The COMPUTE and COMPUTE BY clauses are provided for backward compatibility. Instead, use these components:

- Microsoft[®] SQL Server[™] 2000 Analysis Services in conjunction with OLE DB for Analysis Services or Microsoft ActiveX[®] Data Objects Multidimensional (ADO MD). For more information, see Microsoft SQL Server[™] 2000 Analysis Services.
- The ROLLUP operator. For more information, see <u>Summarizing Data</u> <u>Using ROLLUP</u>.

A COMPUTE BY clause allows you to see both detail and summary rows with one SELECT statement. You can calculate summary values for subgroups, or a summary value for the entire result set.

The COMPUTE clause takes the following information:

- The optional BY keyword, which calculates the specified row aggregate on a per column basis.
- A row aggregate function name; for example, SUM, AVG, MIN, MAX, or COUNT.
- A column to perform the row aggregate function upon.

Results Sets Generated by COMPUTE

The summary values generated by COMPUTE appear as separate result sets in the query results. The results of a query that includes a COMPUTE clause are like a control-break report, which is a report whose summary values are controlled by the groupings, or breaks, that you specify. You can produce summary values for groups, and you can calculate more than one aggregate function for the same group.

When COMPUTE is specified with the optional BY clause, there are two result sets for each group that qualifies for the SELECT:

- The first result set for each group has the set of details rows containing the select list information for that group.
- The second result set for each group has one row containing the subtotals of the aggregate functions specified in the COMPUTE clause for that group.

When COMPUTE is specified without the optional BY clause, there are two result sets for the SELECT:

- The first result set for each group has all of the detail rows containing the select list information.
- The second result set has one row containing the totals of the aggregate functions specified in the COMPUTE clause.

Examples Using COMPUTE

This SELECT statement uses a simple COMPUTE clause to produce a grand total of the sum of the price and advances from the titles table:

USE pubs SELECT type, price, advance FROM titles ORDER BY type COMPUTE SUM(price), SUM(advance)

This query adds the optional BY keyword to the COMPUTE clause to produce subtotals for each group:

USE pubs SELECT type, price, advance

FROM titles ORDER BY type COMPUTE SUM(price), SUM(advance) BY type

The results of this SELECT statement are returned in 12 result sets, 2 result sets for each of the 6 groups. The first result set for each group has a set of rows containing the information called for in the select list. The second result set for each group contains the subtotals of the two SUM functions in the COMPUTE clause.

Note Some utilities, such as **osql**, display multiple subtotal or total aggregate summaries in a way that may lead users to assume that each subtotal is a separate row in a result set. This is due to how the utility formats the output; the subtotal or total aggregates are returned in one row. Other applications, such as SQL Query Analyzer, format multiple aggregates on the same line.

Comparing COMPUTE to GROUP BY

To summarize the differences between COMPUTE and GROUP BY:

- GROUP BY produces a single result set. There is one row for each group containing only the grouping columns and aggregate functions showing the subaggregate for that group. The select list can contain only the grouping columns and aggregate functions.
- COMPUTE produces multiple result sets. One type of result set contains the detail rows for each group containing the expressions from the select list. The other type of result set contains the subaggregate for a group, or the total aggregate for the SELECT statement. The select list can contain expressions other than the grouping columns or aggregate functions. The aggregate functions are specified in the COMPUTE clause, not in the select list.

This query uses GROUP BY and aggregate functions; it returns one result set having one row per group containing the aggregate subtotals for that group:

USE pubs SELECT type, SUM(price), SUM(advance) FROM titles GROUP BY type

Note You cannot include **ntext**, **text**, or **image** data types in a COMPUTE or COMPUTE BY clause.

See Also

Query Fundamentals

<u>SELECT</u>

Error Handling

Errors raised in Microsoft® SQL Server™ 2000 have several attributes:

• Error number.

Each error condition has a unique error number.

• Error message string.

The error message gives diagnostic information about the cause of the error. Many error messages have substitution variables in which information, such as the name of the object generating the error, is placed. Every error number has a unique error message.

• Severity.

The severity indicates how serious the error is. Errors with a low severity, such as 1 or 2, are information messages or low-level warnings. Errors with a high severity indicate problems that should be addressed as soon as possible.

• State code.

Some error codes can be raised at multiple points in the source code for SQL Server. For example, an "1105" error can be raised for several different conditions. Each place the error code is raised assigns a unique state code. A Microsoft support engineer can use the state code from an error to find the location in the source code where that error code is being raised, which may provide additional ideas on how to diagnose the problem.

• Procedure name.

If the error occurred in a stored procedure, the name of the stored procedure may be available.

• Line number.

The line number indicates which statement in a stored procedure generated the error.

All of the SQL Server errors are stored in the system table **master.dbo.sysmessages**. User-defined messages can also be stored in **sysmessages**. The RAISERROR statement can then be used to return these user-defined errors to an application if necessary.

All the database APIs, such as ADO, OLE DB, ODBC, DB-Library, and Embedded SQL, report the basic error attributes: the error number and message string. However, there are variations in how many of the other error attributes each database can report.

Other SQL Server components can also raise errors:

- The OLE DB provider, ODBC driver, and DB-Library dynamic-link library raise errors of their own. The format of these errors is consistent with the formats defined in the API specifications.
- The Net-Libraries raise errors of their own.
- Open Data Services raises errors in its own format.
- The SQL Server wizards, applications, and utilities such as the Index Tuning Wizard, SQL Server Enterprise Manager, and the **osql** utility can raise their own errors.
- Embedded SQL can raise SQL-92 errors.

See Also

<u>sysmessages</u>

Handling Errors and Messages in Applications

Errors raised either by Microsoft® SQL Server[™] 2000 or the RAISERROR statement are not part of a result set. Errors are returned to applications through an error handling mechanism separate from the processing of result sets.

Each database API has some set of functions, interfaces, methods, objects or structures through which they return errors and messages. Each API function or method typically returns a status code indicating the success of that operation. If the status is anything other than success, the application can call the error functions, methods, or objects to retrieve the error information.

SQL Server actually has two mechanisms for returning error information:

- Errors
 - The errors from **sysmessages** with a severity of 11 or higher.
 - Any RAISERROR statement with a severity of 11 or higher.
- Messages
 - The output of the PRINT statement.
 - The output of several DBCC statements.
 - The errors from **sysmessages** with a severity of 10 or lower.
 - Any RAISERROR statement with a severity of 10 or lower.

Applications using APIs such as ADO and OLE DB cannot generally distinguish between errors and messages. In ODBC applications, messages generate a SQL_SUCCESS_WITH_INFO function return code, and errors usually generate a SQL_ERROR return code. The difference is most pronounced in DB-Library, in which errors are returned to the application error handler function, and messages are returned to the application message handler function.

ODBC Error Handling

The ODBC specification introduced an error model that has served as the foundation of the error models of the generic database APIs such as ADO, OLE DB, and the APIs built over ODBC (RDO, DAO, and the MFC Database Classes). In the ODBC model, errors have the following attributes:

• SQLSTATE

The SQLSTATE is a five-character error code defined originally in the ODBC specification. SQLSTATEs are common across all ODBC drivers and provide a way for applications to code basic error handling without testing for all the different error codes returned by various databases. The ODBC SQLSTATE has nothing to do with the state attribute of SQL Server error messages.

ODBC 2.*x* returns one set of SQLSTATE codes, and ODBC 3.*x* returns a set of SQLSTATE codes aligned with the X/Open Data Management: Structured Query Language (SQL), version 2 standard. Because all ODBC drivers return the same sets of SQLSTATE codes, applications basing their error handling on SQLSTATE codes are more portable.

• Native error number

The native error number is the error number from the underlying database. ODBC applications receive the SQL Server error numbers as native error numbers.

• Error message string

The error message is returned in the error message string parameter.

When an ODBC function returns a status other than SQL_SUCCESS, the application can call **SQLGetDiagRec** to get the error information. For example, if an ODBC application gets a syntax error (SQL Server error number 170), **SQLGetDiagRec** returns:

```
szSqlState = 42000, pfNative = 170
szErrorMsg =
'[Microsoft][ODBC SQL Server Driver][SQL Server]
Line 1: Incorrect syntax near *'
```

The ODBC **SQLGetDiagField** function allows ODBC drivers to specify driverspecific diagnostic fields in the diagnostic records returned by the driver. The SQL Server ODBC driver specifies driver-specific fields to hold SQL Server error information such as the SQL Server severity and state codes.

For more information about retrieving error messages in ODBC applications, see <u>Handling Errors and Messages</u>.

ADO Error Handling

ADO uses an **Errors** object and **Errors** collection to return standard error information such as SQLSTATE, native error number, and the error message string. These are the same as their ODBC counterparts. ADO does not support any provider-specific error interfaces, so SQL Server-specific error information such as the severity or state are available to ADO applications.

For more information about retrieving error messages in ADO applications, see <u>Handling Errors and Messages in ADO</u>.

OLE DB Error Handling

OLE DB uses the **IErrorInfo** interface to return standard error information such as the SQLSTATE, native error number, and error string. These are the same as their ODBC counterparts. The Microsoft OLE DB Provider for SQL Server defines an **ISQLServerErrorInfo** interface to return SQL Server-specific information such as the severity, state, procedure name, and line number.

For more information about retrieving error messages in OLE DB applications, see <u>Errors</u>.

DB-Library Error Handling

DB-Library uses a different mechanism for returning error information to an application. An application defines two call-back functions, an error handler and a message handler. When the DB-Library dynamic-link library has errors or messages to return, it calls the application's error handler or message handler function. Because DB-Library is specific to SQL Server, all SQL Server error information is available in the error and message handlers. DB-Library returns PRINT messages and low-severity error or RAISERROR messages to the

message handler. High severity errors and RAISERROR messages are returned to the error handler function.

For more information about DB-Library error handling see <u>Error and Message</u> <u>Handling</u>.

Using @@ERROR

The @@ERROR system function returns 0 if the last Transact-SQL statement executed successfully; if the statement generated an error, @@ERROR returns the error number. The value of @@ERROR changes on the completion of each Transact-SQL statement.

Because @@ERROR gets a new value when every Transact-SQL statement completes, process @@ERROR in one of two ways:

- Test or use @@ERROR immediately after the Transact-SQL statement.
- Save @@ERROR in an integer variable immediately after the Transact-SQL statement completes. The value of the variable can be used later.

@@ERROR is the only part of a Microsoft® SQL Server[™] 2000 error available within the batch, stored procedure, or trigger that generated the error. All other parts of the error, such as its severity, state, and message text containing replacement strings such as object names, are returned only to the application in which they can be processed using the API error handling mechanisms. Also, @@ERROR is raised only for errors, not for warnings; therefore, batches, stored procedures, and triggers do not have visibility to any warnings that may have occurred.

A common use of @@ERROR is to indicate the success or failure of a stored procedure. An integer variable is initialized to 0. After each Transact-SQL statement completes, @@ERROR is tested for being 0, and if it is not 0, it is stored in the variable. The procedure then returns the variable on the RETURN statement. If none of the Transact-SQL statements in the procedure had an error, the variable remains at 0. If one or more statements generated an error, the variable holds the last error number. This is a simple stored procedure with this logic:

USE Northwind GO

DROP PROCEDURE SampleProcedure

GO

-- Create a procedure that takes one input parameter

-- and returns one output parameter and a return code.

CREATE PROCEDURE SampleProcedure @EmployeeIDParm INT,

@MaxQuantity INT OUTPUT

AS

-- Declare and initialize a variable to hold @@ERROR. DECLARE @ErrorSave INT SET @ErrorSave = 0

-- Do a SELECT using the input parameter. SELECT FirstName, LastName, Title FROM Employees WHERE EmployeeID = @EmployeeIDParm

```
-- Save any non-zero @@ERROR value.
IF (@@ERROR <> 0)
SET @ErrorSave = @@ERROR
```

```
-- Set a value in the output parameter.
SELECT @MaxQuantity = MAX(Quantity)
FROM [Order Details]
```

```
-- Save any non-zero @@ERROR value.
IF (@@ERROR <> 0)
SET @ErrorSave = @@ERROR
```

```
-- Returns 0 if neither SELECT statement had
-- an error, otherwise returns the last error.
RETURN @ErrorSave
GO
```

There are situations when @@ERROR can be used with @@ROWCOUNT. In

the following example, @@ERROR is used to determine if a constraint violation error occurred, and @@ROWCOUNT is used to determine the number of rows modified by the UPDATE statement, if any rows were successfully changed.

BEGIN TRAN UPDATE Northwind.dbo.Products SET UnitPrice = UnitPrice * 1.1 WHERE CategoryID IN (1, 2, 5, 6)

```
IF @@ERROR = 547
PRINT 'A CHECK CONSTRAINT violation occurred'
IF @@ROWCOUNT = 0
PRINT 'No rows updated.'
ELSE
PRINT STR(@@ROWCOUNT) + ' rows updated.'
COMMIT -- Commits rows successfully updated.
```

See Also

@@ERROR

@@ROWCOUNT

Using PRINT

The PRINT statement takes either one character or a Unicode string expression as a parameter. It returns the string as a message to the application. The message is returned as an informational error in ADO, OLE DB, and ODBC applications. SQLSTATE is set to 01000, the native error is set to 0, and the error message string is set to the character string specified in the PRINT statement. The string is returned to the message handler call-back function in DB-Library applications.

The PRINT statement accepts any character string expression, including character or Unicode constants, a character or Unicode local variable name, or a function that returns a character or Unicode string. With Microsoft® SQL ServerTM 2000, PRINT also accepts complex strings built by concatenating two or more constants, local variables, or functions.

Use PRINT to help in troubleshooting Transact-SQL code, to check the values of data, or to produce reports.

This example uses PRINT inside an IF statement to return a message to the application:

IF (SELECT COUNT(au_lname) FROM authors WHERE state = 'UT' PRINT 'More than one author resides in the state of Utah.'

This example prints a combination of a local variable, system functions, and a text string using concatenation:

USE Northwind GO DECLARE @MyObject NVARCHAR(128)

SET @MyObject = 'Products'

```
PRINT 'Object Name: ' + @MyObject
PRINT ' Object ID: ' + STR(Object_ID(@MyObject))
PRINT 'The computer ' + RTRIM(@@SERVERNAME) + ' is running
+ RTRIM(@@VERSION)
```

GO

-- This shows building a character variable into a print

-- message. This is required for earlier versions of SQL

-- Server, in which the PRINT statement did not support

-- concatenation.

DECLARE @Msg VARCHAR(255)

PRINT @Msg

Using RAISERROR

RAISERROR is a more powerful statement than PRINT for returning messages back to applications. RAISERROR can return messages in either of these forms:

- A user-defined error message that has been added to **master.dbo.sysmessages** using the **sp_addmessage** system stored procedure.
- A message string specified in the RAISERROR statement.

RAISERROR also has these extensions to the capabilities of PRINT:

- RAISERROR can assign a specific error number, severity, and state.
- RAISERROR can request that the error be logged in the Microsoft® SQL Server[™] 2000 error log and the Microsoft Windows NT® application log.
- The message string can contain substitution variables and arguments, much like the C language **printf** function.

When RAISERROR is used with the *msg_id* of a user-defined message in **sysmessages**, *msg_id* is returned as the SQL Server error number, or native error code. When RAISERROR is used with a *msg_str* instead of a *msg_id*, the SQL Server error number and native error number returned is 50000.

When you use RAISERROR to return a user-defined error message, use a different state number in each RAISERROR that references that error. This can aid in diagnosing the errors when they are raised.

Use RAISERROR to help in troubleshooting Transact-SQL code, to check the values of data, or to return messages that contain variable text.

This example substitutes the values from the DB_ID and DB_NAME functions in a message sent back to the application:

DECLARE @DBID INT SET @DBID = DB_ID()

DECLARE @DBNAME NVARCHAR(128) SET @DBNAME = DB_NAME()

RAISERROR

('The current database ID is:%d, the database name is: %s.', 16, 1, @DBID, @DBNAME)

This example accomplishes the same process using a user-defined message:

DECLARE @DBNAME NVARCHAR(128) SET @DBNAME = DB_NAME()

RAISERROR (50005, 16, 1, @DBID, @DBNAME) GO

This second RAISERROR example shows that substitution parameters can be specified in a user-defined error and filled with substitution arguments at the time the RAISERROR statement executes.

Querying SQL Server System Catalogs

Dynamic applications that are not hard-coded to work with a specific set of tables and views must have a mechanism for determining the structure and attributes of the objects in any database to which it connects. The applications may need information such as:

- The number and names of the tables and views in a database.
- The number of columns in a table or view, along with each column's name, data type, scale, and precision.
- The constraints defined on a table.
- The indexes and keys defined for a table.

The Microsoft® SQL Server[™] 2000 system catalog provides this information for SQL Server databases. The core of the SQL Server system catalogs is a set of system tables containing meta data describing the objects in a SQL Server database. Meta data is data that describes the attributes of objects in a system.

SQL Server applications can access the information in the system catalogs in several ways by using:

• Information Schema Views

Information Schema Views are based on catalog view definitions in the SQL-92 standard. They present the catalog information in a format independent of any catalog table implementation, thus are not affected by changes in the underlying catalog tables. Applications that use these views are portable between heterogeneous SQL-92 compliant database systems. For more information, see <u>Information Schema Views</u>.

• OLE DB schema rowsets

The OLE DB specification defines an IDBSchemaRowset interface that exposes a set of schema rowsets that contain the catalog information.

The OLE DB schema rowsets are a standard method of presenting catalog information supported by different OLE DB providers. The rowsets are independent of the structure of the underlying catalog tables. For more information, see <u>Schema Rowset Support in</u> <u>SQLOLEDB</u>.

The OLE DB Provider for SQL Server supports an extension to IDBSchemaRowset that reports catalog information for the linked servers used in distributed queries. For more information, see <u>LINKEDSERVERS Rowset</u>.

• ODBC catalog functions

The ODBC specification defines a set of catalog functions that return result sets that contain the catalog information. These functions are a standard method of presenting catalog information supported by different ODBC drivers. The result sets are independent of the structure of the underlying catalog tables.

The SQL Server ODBC driver supports two driver-specific functions that report catalog information for the linked servers used in distributed queries. For more information, see <u>Using Catalog Functions</u>.

• System stored procedures and functions

Transact-SQL defines server system stored procedures and system functions that return catalog information. Although these stored procedures and functions are specific to SQL Server, they insulate users from the structure of the underlying system catalog tables. For more information, see <u>Metadata Functions</u> and <u>System Stored Procedures</u>.

It is not recommended that users query the system catalog tables directly. This should only be done if none of the methods above supply the needed information. The structure of the system catalog tables is dependent on the underlying architecture of SQL Server, and changes from one version to another. Even an application that only issues SELECT statements may have to be at least partially rewritten when migrating to a new version of SQL Server if it directly queries system tables that change or are not present in the new version.

Updating, deleting, or inserting data in a system table can cause unpredictable effects in a SQL Server system. Such updates are not supported by Microsoft.

Managing ntext, text, and image Data

The Microsoft® SQL Server[™] **ntext**, **text**, and **image** data types are capable of holding extremely large amounts of data (up to 2 GB) in a single value. A single data value is typically larger than can be retrieved by an application in one step; some values may be larger than the virtual memory available on the client. Therefore, special steps are usually needed to retrieve these values.

If an **ntext**, **text**, and **image** data value is no longer than a Unicode, character, or binary string (4,000 characters, 8,000 characters, 8,000 bytes respectively), the value can be referenced in SELECT, UPDATE, and INSERT statements much the same way as the smaller data types. For example, an **ntext** column with a short value can be referenced in a SELECT statement select list the same way an **nvarchar** column is referenced. Some restrictions that must be observed, such as not being able to directly reference an **ntext**, **text**, or **image** column in a WHERE clause. These columns can be included in a WHERE clause as parameters of a function that returns another data type (such as ISNULL, SUBSTRING or PATINDEX) or in an IS NULL, IS NOT NULL, or LIKE expression.

Handling Larger Data Values

When the **ntext**, **text**, and **image** data values get larger, however, they must be handled on a block-by-block basis. Both Transact-SQL and the database APIs contain functions that allow applications to work with **ntext**, **text**, and **image** data block by block.

The database APIs follow a common pattern in the ways they handle long **ntext**, **text**, and **image** columns:

- To read a long column, the application simply includes the **ntext**, **text**, or **image** column in a select list, and then binds the column to a program variable large enough to hold a reasonable block of the data. The application then executes the statement and uses an API function or method to retrieve the data into the bound variable one block at a time.
- To write a long column, the application executes an INSERT or

UPDATE statement with a parameter marker (?) in the place of the value to be placed in the **ntext**, **text**, or **image** column. The parameter marker (or parameter in the case of ADO) is bound to a program variable large enough to hold the blocks of data. The application goes into a loop where it first moves the next set of data into the bound variable, and then calls an API function or method to write that block of data. This is repeated until the entire data value has been sent.

Using text in row

In Microsoft SQL Server 2000, users can enable a **text in row** option on a table so it could store **text**, **ntext**, or **image** data in its data row.

To enable the option, execute the **sp_tableoption** stored procedure, specifying **text in row** as the option name and **on** as the option value. The default maximum size that can be stored in a row for a BLOB (binary large object: **text**, **ntext**, or **image** data) is 256 bytes, but values may range from 24 through 7000. To specify a maximum size that is not the default, specify an integer within the range as the option value.

text, **ntext**, or **image** strings are stored in the data row if the following conditions apply:

- **text in row** is enabled.
- The length of the string is shorter than the limit specified in @OptionValue
- There is enough space available in the data row.

When BLOB strings are stored in the data row, reading and writing the **text**, **ntext**, or **image** strings can be as fast as reading or writing character and binary strings. SQL Server does not have to access separate pages to read or write the BLOB string.

If a **text**, **ntext**, or **image** string is larger than the specified limit or the available space in the row, pointers are stored in the row instead. The conditions for storing the BLOB strings in the row still apply though: There must be enough

space in the data row to hold the pointers.

For more information, see <u>sp_tableoption</u>.

Using text pointers

Unless the **text in row** option is specified, **text**, **ntext**, or **image** strings are stored outside a data row; only the text pointers to these strings reside in the data rows. Text pointers point to the root node of a tree built of internal pointers that map to the pages in which string fragments (of **text**, **ntext**, and **image** data) are actually stored.

In row text pointers in SQL Server 2000 are different from the text pointers in earlier versions of SQL Server. In row text pointers behave like file handles for BLOB data; earlier text pointers function like addresses to the BLOB data. Thus, when using in row text pointers, keep in mind the following characteristics:

IMPORTANT Although an in row text is allowed in a cursor, an in row text pointer is not. SQL Server will return the error message (8654, 16, 1, 'A cursor plan could not be generated for the given statement because it contains textptr(inrow lob).', 1033) if you attempt to declare a cursor that contains an in row text pointer.

1. Number

A maximum of 1024 active in row text pointers are allowed per transaction per database.

2. Locking

When a user obtains an active text pointer, SQL Server 2000 locks the data row and ensures no other user modifies or deletes the row while the first user has the text pointer. The lock is released when the text pointer becomes invalid. To invalidate a text pointer, use <u>sp_invalidate_textptr</u>.

A text pointer cannot be used to update BLOB values when the isolation level of the transaction is read uncommitted, or the database is in read-only mode.

SQL Server 2000 does not lock the data row if the database is in single-user mode.

To illustrate, given the following table:

CREATE TABLE t1 (c1 int, c2 text) EXEC sp_tableoption 't1', 'text in row', 'on' INSERT t1 VALUES ('1', 'a')

The following transaction will succeed:

INSERT t1 VALUES ('1','This is text.') SET TRANSACTION ISOLATION LEVEL READ UNCOM GO BEGIN TRAN DECLARE @ptr varbinary(16) SELECT @ptr = textptr(c2) FROM t1 WHERE c1 = 1 READTEXT t1.c2 @ptr 0 5 COMMIT TRAN GO

The following transaction will fail:

```
SET TRANSACTION ISOLATION LEVEL READ UNCOM
GO
BEGIN TRAN
DECLARE @ptr varbinary(16)
SELECT @ptr = textptr(c2)
FROM t1
WHERE c1 = 1
WRITETEXT t1.c2 @ptr 'xx'
COMMIT TRAN
GO
```

3. Duration

In row text pointers are valid only within a transaction. When a

transaction is committed, the text pointer becomes invalid.

Within a transaction, in row text pointers can be invalidated when any of the following actions take place:

- The session ends.
- The data row is deleted in the same transaction. (Other transactions cannot delete a data row because of the lock obtained on it.)
- The schema of a table in which the text pointer resides is changed. Schema-changing actions that invalidate text pointers include: creating or dropping clustered index, altering or dropping the table, truncating the table, changing the **text in row** option through **sp_tableoption**, and executing **sp_indexoption**.

Using the earlier example, the following script would work in earlier versions of SQL Server, but will generate an error in SQL Server 2000.

```
DECLARE @ptrval varbinary(16)
PRINT 'get error here'
SELECT @ptrval = TEXTPTR(c2)
FROM t1
WHERE c1 = 1
READTEXT t1.c2 @ptrval 0 1
```

In SQL Server 2000, the in row text pointer must be used inside a transaction:

```
BEGIN TRAN
DECLARE @ptrval varbinary(16)
SELECT @ptrval = TEXTPTR(c2)
FROM t1
WHERE c1 = 1
READTEXT t1.c2 @ptrval 0 1
```

COMMIT

4. NULL text

You can get an in row text pointer on NULL text that is generated by INSERT. Previously, you can get text pointers only after updating a BLOB to NULL.

For example, the following code does not work in SQL Server 7.0, but works in SQL Server 2000.

SET TRANSACTION ISOLATION LEVEL READ COMMI' GO INSERT INTO t1 VALUES (4, NULL) BEGIN TRAN DECLARE @ptrval VARBINARY(16) SELECT @ptrval = TEXTPTR(c2) FROM t1 WHERE c1 = 4 WRITETEXT t1.c2 @ptrval 'x4' COMMIT

In SQL Server 7.0, you must do the following:

```
INSERT INTO t1 VALUES (4, NULL)

UPDATE t1

SET c2 = NULL

WHERE c1 = 4

DECLARE @ptrval VARBINARY(16)

SELECT @ptrval = TEXTPTR(c2)

FROM t1

WHERE c1 = 4

WRITETEXT t1.c2 @ptrval 'x4'
```

This table summarizes the differences.

Difference	in row text pointer	pointer
Number	Maximum of 1024 active per transaction per database.	Unlimited.
Locking	Data row is S locked until the pointer becomes invalid. Locks are not obtained when the transaction is 'read uncommitted', or the database is 'single-user' or 'read- only' mode.	Data row is not locked.
Duration	Becomes invalid at the end of transaction or session, when a row is deleted or the schema of the table is changed.	Becomes invalid when the row is deleted.
NULL text	Obtainable right after the insert of NULL text.	Obtainable only after update.

Using ntext, text, and image Data with Database APIs

This is a summary of the ways the database APIs handle **ntext**, **text**, and **image** data:

• ADO

ADO can map **ntext**, **text**, or **image** columns or parameters to a **Field** or **Parameter** object. Use the **GetChunk** method to retrieve the data one block at a time and the **AppendChunk** method to write data one block at a time. For more information, see <u>Managing Long Data Types</u>.

• OLE DB

OLE DB uses the **ISequentialStream** interface to support **ntext**, **text**, and **image** data types. The **ISequentialStream::Read** method reads the long data one block at a time, and **ISequentialStream::Write** writes the long data to the database one block at a time. For more information, see <u>BLOBs and OLE Objects</u>.

• ODBC

ODBC has a feature called "data-at-execution" to deal with the ODBC data types for long data: SQL_WLONGVARCHAR (**ntext**), SQL_LONGVARCHAR (**text**), and SQL_LONGVARBINARY (**image**). These data types are bound to a program variable. **SQLGetData** is then called to retrieve the long data one block at a time, and **SQLPutData** is called to send long data one block at a time. For more information, see <u>Managing text and image Columns</u>.

• DB-Library

DB-Library applications also bind **ntext**, **text**, and **image** columns to program variables. The DB-Library function **dbtxtptr** is used to get a pointer to the location of the long column occurrence in the database. **dbreadtext** is used to read the long data one block at a time. Functions such as **dbwritetext**, **dbupdatetext**, and **dbmoretext** are used to write the long data one block at a time.

Note Accessing in row text with DB-Library is not supported.

For more information, see <u>Text and Image Functions</u>.

Retrieving ntext, text, or image Values

You can retrieve **ntext**, **text** or **image** values by:

• Simply referencing the column in a SELECT statement.

For example, this query returns all information in the **pr_info** column for each publisher:

USE pubs SELECT pr_info FROM pub_info

This is the method used in a database application using an API such as ADO, OLE DB, ODBC, or DB-Library. The column is bound to a program variable, and then a special API function or method is used to retrieve the data one block at a time.

When this method is used in Transact-SQL scripts, stored procedures, and triggers, it works only for relatively short values. If the length of the data is longer than the length specified in SET TEXTSIZE, you must use increase TEXTSIZE or use another method. The current TEXTSIZE setting is reported by the @@TEXTSIZE function and is changed with the SET TEXTSIZE statement:

SET TEXTSIZE 64512

The default setting for TEXTSIZE is 4096 (4 KB). This statement resets TEXTSIZE to its default value:

SET TEXTSIZE 0

The full amount of data is returned if the length is less than TEXTSIZE.

The DB-Library API also supports a **dbtextsize** parameter that controls the length of **ntext**, **text**, and **image** data that can be selected. The Microsoft OLE DB Provider for SQL Server and the SQL Server ODBC driver automatically set @@TEXTSIZE to its maximum of 2 GB.

• Using the TEXTPTR function to get a text pointer that is passed to the READTEXT statement.

The READTEXT statement is used to read blocks of **ntext**, **text**, or **image** data. For example, this query returns the first 25 characters (or first row) of the sample text data for each publisher:

USE pubs DECLARE @textpointer varbinary(16) SELECT @textpointer = TEXTPTR(pr_info) FROM pub_info READTEXT pub_info.pr_info @textpointer 1 25

• Using the SUBSTRING function to retrieve a block of data starting at a specific offset from the start of the column.

For example, this query returns the first 25 characters (or first row) of the sample text data for each publisher:

```
USE pubs
SELECT SUBSTRING(pr_info, 1, 25) AS pr_info
FROM pub_info
```

• Using the PATINDEX function to retrieve offset of some particular pattern of bytes.

This value can then be used in a SUBSTRING function or READTEXT statement to retrieve the data. For example, this query searches for the string Germany in the **pr_info** column of the **pub_info** table and returns the starting position of 103 (the G of the string Germany begins at character 103 of the **pr_info** column):

USE pubs SELECT PATINDEX('%Germany%', pr_info) AS pr_info FROM pub_info

PATINDEX operates on text and character data types only; it does not accept image values.

Retrieving Parts of ntext, text, or image Values

These methods are not limited to retrieving the entire **ntext**, **text**, or **image** value starting with the first byte. The methods can be combined to provide flexible processing that retrieves different parts of the **ntext**, **text**, or **image** values. For example, this SELECT statement retrieves whatever part of a **text** value is between a start tag and an end tag:

```
USE Northwind
GO
CREATE TABLE TextParts (ColA INT PRIMARY KEY, ColB TEXT)
GO
INSERT INTO TextParts
 VALUES(1,
      'Sample string START TAG What I want END TAG Trailing tex
GO
SELECT SUBSTRING( ColB,
           /* Calculate start as start of tag + tag length. */
           (PATINDEX('%START TAG%', ColB) + 10),
          /* Calculate SUBSTRING length as end - start. */
           (
            PATINDEX('%END TAG%', ColB) -
            (PATINDEX('%START TAG%', ColB) + 10)
           )
         )
FROM TextParts
GO
Here is the result set:
What I want
```

(1 row(s) affected)

Note When you are selecting **image** data, the returned value includes the characters 0x, which indicate that the data is hexadecimal. These two characters are counted as part of TEXTSIZE.

See Also

READTEXT SET TEXTSIZE SELECT Text and Image Functions BLOBs and OLE Objects Managing Long Data Types Managing text and image Columns

Modifying ntext, text, or image Values

You can modify **ntext**, **text**, or **image** values by:

- Using a database API such as ADO, OLE DB, or ODBC to execute an UPDATE or INSERT statement with a program variable bound to a parameter marker for the **ntext**, **text**, or **image** column. Then call the appropriate database API functions to send long data to the database one block at a time. DB-Library supports the same functionality with its **text** and **image** functions.
- Using the WRITETEXT statement to rewrite the entire data value for the column.

For example, this query changes the contents of the **pr_info** column for New Moon Books:

```
USE pubs

sp_dboption 'pubs', 'select into/bulkcopy', 'true'

DECLARE @ptrval varbinary(16)

SELECT @ptrval = TEXTPTR(pr_info)

FROM pub_info pr INNER JOIN publishers p

ON p.pub_id = pr.pub_id

AND p.pub_name = 'New Moon Books'

WRITETEXT pub_info.pr_info @ptrval 'New Moon Books (N

sp_dboption 'pubs', 'select into/bulkcopy', 'true'
```

• Use the UPDATETEXT statement to update specific blocks of an **ntext**, **text**, or **image** column.

For example, this query replaces the eighty-eighth character in the **text** column for New Moon Books (the second letter o in Moon) with the letter z:

USE pubs sp_dboption 'pubs', 'select into/bulkcopy', 'true' DECLARE @ptrval varbinary(16) SELECT @ptrval = TEXTPTR(pr_info) FROM pub_info pr INNER JOIN publishers p ON p.pub_id = pr.pub_id AND p.pub_name = 'New Moon Books' UPDATETEXT pub_info.pr_info @ptrval 88 1 'z' sp_dboption 'pubs', 'select into/bulkcopy', 'false'

See Also

WRITETEXT UPDATETEXT Managing Long Data Types BLOBs and OLE Objects

Managing text and image Columns

Text and Image Functions

OLE Automation Objects in Transact-SQL

Transact-SQL includes several system stored procedures that allow OLE Automation objects to be referenced in Transact-SQL batches, stored procedures, and triggers. These system stored procedures run as extended stored procedures, and the OLE Automation objects that are executed through the stored procedures run in the Microsoft® SQL Server[™] 2000 address space in the same way that an extended stored procedure runs.

The OLE Automation stored procedures allow Transact-SQL batches to reference SQL DMO objects and custom OLE Automation objects, such as objects that expose the IDispatch interface. A custom in-process OLE server created using Microsoft Visual Basic® must have an error handler (specified with the **On Error GoTo** statement) for the **Class_Initialize** and **Class_Terminate** subroutines. The error handlers prevent unhandled errors from occurring in these subroutines. Unhandled errors in the **Class_Initialize** and **Class_Terminate** subroutines can cause unpredictable errors, such as an SQL Server access violation. Error handlers for other subroutines are also recommended.

The first step when using an OLE Automation object in Transact-SQL is to call the **sp_OACreate** system stored procedure to create an instance of the object in the SQL Server address space.

After an instance of the object has been created, call these stored procedures to work with the properties, methods, and error information related to the object:

- **sp_OAGetProperty** obtains the value of a property.
- **sp_OASetProperty** sets the value of a property.
- **sp_OAMethod** calls a method.
- **sp_OAGetErrorInfo** obtains the most recent error information.

When there is no more need for the object, call **sp_OADestroy** to deallocate the

instance of the object created with **sp_OACreate**.

OLE Automation objects return data through property values and methods. **sp_OAGetProperty** and **sp_OAMethod** return these data values in the form of a result set.

The scope of an OLE Automation object is a batch. All references to the object must be contained in a single batch, stored procedure, or trigger.

When referencing objects, the SQL Server OLE Automation objects support traversing the object to other objects it contains. For example, when using the SQL-DMO **SQLServer** object, references can be made to databases and tables contained on that server. For more information, see <u>Object Hierarchy Syntax</u>.

See Also

sp_OACreate sp_OAGetProperty sp_OASetProperty sp_OAMethod sp_OAGetErrorInfo sp_OADestroy

OLE Automation Return Codes and Error Information

The OLE Automation system stored procedures return an **int** return code that is the HRESULT returned by the underlying OLE Automation operation. An HRESULT of 0 indicates success. A nonzero HRESULT is an OLE error code of the hexadecimal form 0x800*nnnn*, but when returned as an **int** value in a stored procedure return code, it has the form –214*nnnnn*.

For example, passing an invalid object name (SQLDMO.Xyzzy) to **sp_OACreate** causes the procedure to return an **int** HRESULT of –2147221005, which is 0x800401f3 in hexadecimal.

You can use CONVERT(**binary(4**), **@hresult**) to convert an **int** HRESULT to a **binary** value. However, using CONVERT(**char(10**), CONVERT(**binary(4**), **@hresult**)) results in an unreadable string because each byte of the HRESULT is converted to a single ASCII character. You can use the following sample **sp_hexadecimal** stored procedure to convert an **int** HRESULT to a **char** value that contains a readable hexadecimal string.

```
CREATE PROCEDURE sp_hexadecimal
@binvalue varbinary(255),
@hexvalue varchar(255) OUTPUT
AS
DECLARE @charvalue varchar(255)
DECLARE @i int
DECLARE @length int
DECLARE @hexstring char(16)
SELECT @charvalue = '0x'
SELECT @i = 1
SELECT @length = DATALENGTH(@binvalue)
SELECT @hexstring = '0123456789abcdef'
WHILE (@i <= @length)
BEGIN
DECLARE @tempint int
```

```
DECLARE @firstint int
DECLARE @secondint int
SELECT @tempint = CONVERT(int, SUBSTRING(@binvalue,@i,1)
SELECT @firstint = FLOOR(@tempint/16)
SELECT @secondint = @tempint - (@firstint*16)
SELECT @charvalue = @charvalue +
SUBSTRING(@hexstring, @firstint+1, 1) +
SUBSTRING(@hexstring, @secondint+1, 1)
SELECT @i = @i + 1
END
SELECT @hexvalue = @charvalue
```

You can use the following sample stored procedure, **sp_displayoaerrorinfo**, to display OLE Automation error information when one of the OLE Automation procedures returns a nonzero HRESULT return code. This sample stored procedure uses **sp_hexadecimal**.

```
CREATE PROCEDURE sp_displayoaerrorinfo
  @object int,
  @hresult int
AS
DECLARE @output varchar(255)
DECLARE @hrhex char(10)
DECLARE @hr int
DECLARE @source varchar(255)
DECLARE @description varchar(255)
PRINT 'OLE Automation Error Information'
EXEC sp_hexadecimal @hresult, @hrhex OUT
SELECT @output = ' HRESULT: ' + @hrhex
PRINT @output
EXEC @hr = sp_OAGetErrorInfo @object, @source OUT, @descript
IF @hr = 0
BEGIN
SELECT @output = ' Source: ' + @source
```

PRINT @output SELECT @output = ' Description: ' + @description PRINT @output END ELSE BEGIN PRINT ' sp_OAGetErrorInfo failed.' RETURN END

See Also

sp_OAGetErrorInfo

OLE Automation Result Sets

If an OLE Automation property or method returns data in an array with one or two dimensions, the array is returned to the client as a result set:

- A one-dimensional array is returned to the client as a single-row result set with as many columns as there are elements in the array. For example, an array(10) is returned as a single row of 10 columns.
- A two-dimensional array is returned to the client as a result set with as many columns as there are elements in the first dimension of the array and with as many rows as there are elements in the second dimension of the array. For example, an array(2,3) is returned as 2 columns in 3 rows.

When a property return value or method return value is an array, **sp_OAGetProperty** or **sp_OAMethod** returns a result set to the client. (Method output parameters cannot be arrays.) These procedures scan all the data values in the array to determine the appropriate Microsoft® SQL Server[™] data types and data lengths to use for each column in the result set. For a particular column, these procedures use the data type and length required to represent all data values in that column.

When all data values in a column share the same data type, that data type is used for the whole column. When data values in a column use different data types, the data type of the whole column is chosen based on the following table.

	int	float	money	datetime	varchar	nvarchar
int	int	float	money	varchar	varchar	nvarchar
float	float	float	money	varchar	varchar	nvarchar
money	money	money	money	varchar	varchar	nvarchar
datetime	varchar	varchar	varchar	datetime	varchar	nvarchar
varchar	varchar	varchar	varchar	varchar	varchar	nvarchar
nvarchar						

For more information about how the OLE Automation stored procedures convert

data between Microsoft Visual Basic® and SQL Server data types, see <u>Data</u> <u>Type Conversions Using OLE Automation Stored Procedures</u>.

Diagnosing OLE Automation Objects in Transact-SQL

When developing Transact-SQL batches that call custom OLE Automation objects, errors can occur in processing the data returned from the object. The problems can be diagnosed through Microsoft® Visual Basic® if the source code for the object is available. To do this, Visual Basic must be installed on the computer running the instance of Microsoft SQL ServerTM 2000 and both SQL Server and Visual Basic must be running under the same Microsoft Windows® user account. SQL Server must be started from the command prompt and started independently of Windows Service Control Manager (by using the **sqlservr /c** command), or the SQL Server service must be started under the same Windows user account used to log on to the system.

When running Visual Basic and SQL Server under the same Windows user accounts:

- 1. Load the custom OLE Automation server project into Visual Basic.
- 2. Set breakpoint(s) on the desired lines of source code.
- 3. On the Visual Basic **Run** menu, click **Start With Full Compile**.

This registers and runs the custom OLE Automation server.

4. Use the OLE Automation stored procedures to call the OLE objects exposed by the custom OLE Automation server.

When a breakpoint is hit, the Visual Basic debugger is activated.

For more information, see the Microsoft Visual Basic documentation.

OLE Automation Sample Script

This is an example of a Transact-SQL statement batch that uses the OLE Automation stored procedures to create and use an SQL-DMO **SQLServer** object. Portions of the code are used as examples in the stored procedure references.

DECLARE @object int DECLARE @hr int DECLARE @property varchar(255) DECLARE @return varchar(255) DECLARE @src varchar(255), @desc varchar(255)

-- Create an object.

EXEC @hr = sp_OACreate 'SQLDMO.SQLServer', @object OUT IF @hr <> 0

BEGIN

EXEC sp_OAGetErrorInfo @object, @src OUT, @desc OUT SELECT hr=convert(varbinary(4),@hr), Source=@src, Description= RETURN

END

```
-- Set a property.
```

EXEC @hr = sp_OASetProperty @object, 'HostName', 'Gizmo' IF @hr <> 0

BEGIN

EXEC sp_OAGetErrorInfo @object, @src OUT, @desc OUT SELECT hr=convert(varbinary(4),@hr), Source=@src, Description= RETURN

END

-- Get a property using an output parameter.

EXEC @hr = sp_OAGetProperty @object, 'HostName', @property OI

```
IF @hr <> 0
BEGIN
    EXEC sp_OAGetErrorInfo @object, @src OUT, @desc OUT
    SELECT hr=convert(varbinary(4),@hr), Source=@src, Description=
    RETURN
END
PRINT @property
```

```
-- Get a property using a result set.
EXEC @hr = sp_OAGetProperty @object, 'HostName'
IF @hr <> 0
BEGIN
EXEC sp_OAGetErrorInfo @object, @src OUT, @desc OUT
SELECT hr=convert(varbinary(4),@hr), Source=@src, Description=
RETURN
END
```

-- Get a property by calling the method.

```
EXEC @hr = sp_OAMethod @object, 'HostName', @property OUT IF @hr <> 0
```

BEGIN

```
EXEC sp_OAGetErrorInfo @object, @src OUT, @desc OUT
SELECT hr=convert(varbinary(4),@hr), Source=@src, Description=
RETURN
```

END

PRINT @property

-- Call a method.

```
EXEC @hr = sp_OAMethod @object, 'Connect', NULL, 'my_server', '
IF @hr <> 0
```

BEGIN

```
EXEC sp_OAGetErrorInfo @object, @src OUT, @desc OUT
SELECT hr=convert(varbinary(4),@hr), Source=@src, Description=
```

RETURN END

```
-- Call a method that returns a value.
EXEC @hr = sp_OAMethod @object, 'VerifyConnection', @return OI
IF @hr <> 0
BEGIN
 EXEC sp_OAGetErrorInfo @object, @src OUT, @desc OUT
 SELECT hr=convert(varbinary(4),@hr), Source=@src, Description=
  RETURN
END
PRINT @return
-- Destroy the object.
EXEC @hr = sp_OADestroy @object
IF @hr <> 0
BEGIN
 EXEC sp_OAGetErrorInfo @object, @src OUT, @desc OUT
 SELECT hr=convert(varbinary(4),@hr), Source=@src, Description=
  RETURN
END
```

See Also

sp_OACreate

```
sp_OAGetProperty
```

sp_OASetProperty

sp_OAMethod

sp_OADestroy

Transact-SQL Tips

Database programmers sometimes encounter puzzling query issues that other programmers may already have resolved. This section lists some of the common challenges and guidelines that improve query performance.

- <u>Cross-Tab Reports.</u> Creates summary information. Most cross-tab reports can be generated by using either the CASE function or the CUBE or ROLLUP options of SELECT.
- Expanding Hierarchies. Shows the hierarchical steps of getting to a particular result (for example, tracing a genealogical family tree from a great-great-grandparent to yourself).
- <u>Expanding Networks</u>. Shows the multiple hierarchical steps (for example, all flights originating from Seattle and landing in New York).
- <u>Writing International Transact-SQL Statements</u>. Lists guidelines for writing applications that can be adapted for use around the world.
- <u>Writing Readable Code</u> Lists good programming practices that make code usable, flexible, and understandable.

Transact-SQL Programming for Improved Performance

When writing Transact-SQL statements, batches, stored procedures, and triggers, use the programming features in Microsoft® SQL ServerTM 2000 to create efficient code.

Reusing Execution Plans

SQL Server 2000 has a better chance of reusing execution plans of Transact-SQL statements if they are written following these guidelines.

- Use fully qualified names of objects such as tables and views.
- Use parameterized queries, and supply the parameter values instead of specifying stored procedure parameter values or the values in search condition predicates directly. Use either the parameter substitution in **sp_executesql** or the parameter binding of the ADO, OLE DB, ODBC, and DB-Library APIs.

For example, do not code this SELECT:

SELECT * FROM Northwind.dbo.Shippers WHERE ShipperII

Instead, using ODBC as an example, use the **SQLBindParameter** ODBC function to bind the parameter marker (?) to a program variable and code the SELECT statement as:

SELECT * FROM Northwind.dbo.Shippers WHERE ShipperIl

In a Transact-SQL script, stored procedure, or trigger, use **sp_executesql** to execute the SELECT statement:

```
DECLARE @IntVariable INT
DECLARE @SQLString NVARCHAR(500)
DECLARE @ParmDefinition NVARCHAR(500)
```

```
/* Build the SQL string. */
SET @SQLString =
    N'SELECT * FROM Northwind.dbo.Shippers WHERE Shij
/* Specify the parameter format once. */
SET @ParmDefinition = N'@ShipID int'
```

```
/* Execute the string. */
SET @IntVariable = 3
EXECUTE sp_executesql @SQLString, @ParmDefinition,
@ShipID = @IntVariable
```

• Use **sp_executesql** when you do not need the overhead of defining stored procedures. Always use **sp_executesql** instead of a temporary stored procedure.

For more information, see <u>Execution Plan Caching and Reuse</u> and <u>Building</u> <u>Statements at Run Time</u>.

Reusing Execution Plans for Batches

When multiple concurrent applications will be executing the same batch with a known set of parameters, implement the batch as a stored procedure that will be called by the applications.

When an ADO, OLE DB, or ODBC application will be executing the same batch multiple times, use the PREPARE/EXECUTE model of executing the batch. Use parameter markers bound to program variables to supply all needed input values, such as the expressions used in an UPDATE VALUES clause or in the predicates in a search condition.

Using the ODBC Escape Sequence

When calling a stored procedure from an ADO, OLE DB, or ODBC application, use the ODBC { CALL *procedure_name* } escape sequence instead of the Transact-SQL EXECUTE statement. For more information, see <u>Calling a Stored Procedure</u>.

Outstanding Transactions and Result Sets

Do not keep a transaction outstanding for long periods of time. A long-standing transaction can reduce throughput by holding locks on rows for long times, preventing other connections from accessing the rows in a timely manner.

Do not keep a result set outstanding for a long period of time. After executing a Transact-SQL batch, fully process or cancel all result sets from the batch as quickly as possible.

Minimizing Rows and Operations

Minimize the number of rows returned from a SELECT statement by using the

WHERE and HAVING clauses to select only the rows needed.

Minimize the use of not equal operations, <>, or !=. SQL Server has to scan a table or index to find all values to see if they are not equal to the value given in the expression. Try rephrasing the expression using ranges:

WHERE KeyColumn < 'TestValue' AND KeyColumn > 'TestValue'

Reduce roundtrips between the application and the server by:

- Including multiple statements in a single batch sent from the application to the server. For more information, see <u>Batches</u>.
- Placing several Transact-SQL statements in a single stored procedure. This reduces the amount of information that has to be sent from the application.
- Reserving the use of server cursors to when the cursor functionality is needed by the application; use a default result set instead. For more information, see <u>Cursors</u>.

For ODBC applications, consider using a fast forward-only cursor with the autofetch option. For more information, see <u>Fast Forward-Only</u> <u>Cursors (ODBC</u>).

Using Advanced Features

Use advanced features available in Transact-SQL to perform work in one batch on the server instead of pulling the results to the application and then using them to send another Transact-SQL statement to SQL Server:

- Use variables and control-of-flow statements to build logic into batches, stored procedures, and triggers instead of pulling large result sets to the client and performing the logic there. For more information, see <u>Using Multiple Statements</u>.
- Use constructs, such as CASE, to include logic in individual Transact-SQL statements. For more information, see <u>Using CASE</u>.

- Use the UPDATE statement with the FROM clause to update values in one table using values from other tables in one operation instead of selecting the source result set to the client and then updating the target table one row at a time.
- Use the join capabilities of SQL Server 2000. For more information, see <u>Join Fundamentals</u>.

Keeping Data Definition Language Statements Together

Within a batch, keep all data definition language (DDL) statements for a temporary table together. For example:

```
/* Example 1. */
CREATE TABLE #temp1 (ColA INT NOT NULL)
CREATE UNIQUE INDEX MyIndex ON #temp1(ColA)
INSERT INTO #temp1 SELECT IntCol FROM SomeTable
SELECT * FROM #temp1
GO
```

```
/* Example 2. */
CREATE TABLE #temp1 (ColA INT UNIQUE NOT NULL)
INSERT INTO #temp1 SELECT IntCol FROM SomeTable
SELECT * FROM #temp1
GO
```

Do not code: /* Example 3. */ CREATE TABLE #temp1 (ColA INT NOT NULL) INSERT INTO #temp1 SELECT IntCol FROM SomeTable CREATE UNIQUE INDEX MyIndex ON #temp1(ColA) SELECT * FROM #temp1 GO Each time a DDL operation is performed on a temporary table, all batches that refer to it must be recompiled. The query optimizer ensures that the CREATE statements in Examples 1 and 2 are done in one operation and the batches are recompiled only once. In Example 3, the INSERT statement between the CREATE statements forces a separate recompile for each CREATE statement.

Minimizing the Use of Temporary Tables

Minimize the use of temporary tables as places to store intermediate results in a series of Transact-SQL statements. Some logic is too complex to perform in a single Transact-SQL statement. In these cases, you must code multiple Transact-SQL statements and use temporary tables to pass the results of one statement to the next. Creating and maintaining the temporary tables requires overhead; if possible, consider coding the operation as a single, more complex Transact-SQL statement.

In SQL Server 2000, use of temporary tables in stored procedures and triggers may cause the stored procedure or trigger to be recompiled every time it is used. To avoid such recompilation, stored procedures or triggers that use temporary tables must meet the following requirements:

- In the stored procedure or trigger, all statements that contain the name of a temporary table must refer to a temporary table created in the same stored procedure. The temporary table cannot have been created in a calling or called stored procedure, or in a string executed using EXECUTE or **sp_executesql**.
- All statements that contain the name of a temporary table must appear syntactically after its creation in the stored procedure or trigger.
- The stored procedure or trigger cannot contain any DECLARE CURSOR statement whose SELECT statement references a temporary table.
- All statements that contain the name of any temporary table must precede any DROP TABLE statement that references a temporary table. DROP TABLE statements are not needed for temporary tables created

in a stored procedure; the tables are dropped automatically when the procedure terminates.

• Statements creating a temporary table (such as CREATE TABLE or SELECT INTO) may not appear in a control-of-flow statement such as IF...ELSE or WHILE.

Preventing Issues with Dates

To prevent issues with the interpretation of centuries in dates, do not specify years using two digits. For example:

```
/* Do this. */
SELECT *
FROM Northwind.dbo.Orders
WHERE OrderDate > '12/31/1997'
/* Do not do this. */
SELECT *
FROM Northwind.dbo.Orders
WHERE OrderDate > '12/31/97'
```

Cross-Tab Reports

Sometimes it is necessary to rotate results so that columns are presented horizontally and rows are presented vertically. This is known as creating a PivotTable®, creating a cross-tab report, or rotating data.

Assume there is a table **Pivot** that has one row per quarter. A SELECT of **Pivot** reports the quarters vertically:

Year	Quarte	er Amount
1990	1	1.1
1990	2	1.2
1990	3	1.3
1990	4	1.4
1991	1	2.1
1991	2	2.2
1991	3	2.3
1991	4	2.4

A report must be produced with a table that contains one row for each year, with the values for each quarter appearing in a separate column, such as:

Year	Q1	Q2	Q3	Q4
1990	1.1	1.2	1.3	1.4
1991	2.1	2.2	2.3	2.4

These are the statements used to create the **Pivot** table and populate it with the data from the first table:

USE Northwind GO

```
CREATE TABLE Pivot
(Year
        SMALLINT,
 Quarter TINYINT,
           DECIMAL(2,1))
 Amount
GO
INSERT INTO Pivot VALUES (1990, 1, 1.1)
INSERT INTO Pivot VALUES (1990, 2, 1.2)
INSERT INTO Pivot VALUES (1990, 3, 1.3)
INSERT INTO Pivot VALUES (1990, 4, 1.4)
INSERT INTO Pivot VALUES (1991, 1, 2.1)
INSERT INTO Pivot VALUES (1991, 2, 2.2)
INSERT INTO Pivot VALUES (1991, 3, 2.3)
INSERT INTO Pivot VALUES (1991, 4, 2.4)
GO
```

This is the SELECT statement used to create the rotated results:

```
SELECT Year,
SUM(CASE Quarter WHEN 1 THEN Amount ELSE 0 END) AS Q
SUM(CASE Quarter WHEN 2 THEN Amount ELSE 0 END) AS Q
SUM(CASE Quarter WHEN 3 THEN Amount ELSE 0 END) AS Q
SUM(CASE Quarter WHEN 4 THEN Amount ELSE 0 END) AS Q
FROM Northwind.dbo.Pivot
GROUP BY Year
GO
```

This SELECT statement also handles a table in which there are multiple rows for each quarter. The GROUP BY combines all rows in **Pivot** for a given year into a single row in the output. When the grouping operation is being performed, the CASE functions in the SUM aggregates are applied in such a way that the **Amount** values for each quarter are added into the proper column in the result set and 0 is added to the result set columns for the other quarters.

If the results of this SELECT statement are used as input to a spreadsheet, it is

easy for the spreadsheet to calculate a total for each year. When the SELECT is used from an application it may be easier to enhance the SELECT statement to calculate the yearly total. For example:

```
SELECT P1.*, (P1.Q1 + P1.Q2 + P1.Q3 + P1.Q4) AS YearTotal
FROM (SELECT Year,
SUM(CASE P.Quarter WHEN 1 THEN P.Amount ELSE 0 EN
SUM(CASE P.Quarter WHEN 2 THEN P.Amount ELSE 0 EN
SUM(CASE P.Quarter WHEN 3 THEN P.Amount ELSE 0 EN
SUM(CASE P.Quarter WHEN 4 THEN P.Amount ELSE 0 EN
FROM Pivot AS P
GROUP BY P.Year) AS P1
```

Both GROUP BY with CUBE and GROUP BY with ROLLUP compute the same sort of information as shown in the example, but in a slightly different format.

See Also

<u>SELECT</u>

Expanding Hierarchies

Databases often store hierarchical information. For example, the following data is a hierarchical representation of regions of the world. This representation does not clearly show the structure implied by the data.

Parent	Child
World	Europe
World	North America
Europe	France
France	Paris
North America	United States
North America	Canada
United States	New York
United States	Washington
New York	New York City
Washington	Redmond
This example is easier to interpret:	
World	
North America	
Canada	
United States	
Washington	
Redmond	
New York	
New York City	
Europe	
France	
Paris	

The following Transact-SQL procedure expands an encoded hierarchy to any

arbitrary depth. Although Transact-SQL supports recursion, it is more efficient to use a temporary table as a stack to keep track of all of the items for which processing has begun but is not complete. When processing is complete for a particular item, it is removed from the stack. New items are added to the stack as they are identified.

```
CREATE PROCEDURE expand (@current char(20)) as
SET NOCOUNT ON
DECLARE @level int, @line char(20)
CREATE TABLE #stack (item char(20), level int)
INSERT INTO #stack VALUES (@current, 1)
SELECT @level = 1
WHILE @level > 0
BEGIN
 IF EXISTS (SELECT * FROM #stack WHERE level = @level)
   BEGIN
    SELECT @current = item
    FROM #stack
    WHERE level = @level
    SELECT @line = space(@level - 1) + @current
    PRINT @line
    DELETE FROM #stack
    WHERE level = @level
      AND item = @current
    INSERT #stack
      SELECT child, @level + 1
      FROM hierarchy
      WHERE parent = @current
    IF @@ROWCOUNT > 0
      SELECT @level = @level + 1
   END
 ELSE
   SELECT @level = @level - 1
```

END -- WHILE

The input parameter (**@current**) indicates the place in the hierarchy to start. It also keeps track of the current item in the main loop.

The local variables used are **@level**, which keeps track of the current level in the hierarchy, and **@line**, which is a work area used to construct the indented line.

The SET NOCOUNT ON statement avoids cluttering the output with ROWCOUNT messages from each SELECT.

The temporary table, **#stack**, is created and primed with the item identifier of the starting point in the hierarchy, and **@level** is set to match. The **level** column in **#stack** allows the same item to appear at multiple levels in the database. Although this situation does not apply to the geographic data in the example, it can apply in other examples.

In this example, when **@level** is greater than 0, the procedure follows these steps:

- 1. If there are any items in the stack at the current level (*@level*), the procedure chooses one and calls it *@current*.
- 2. Indents the item *@level* spaces, and then prints the item.
- 3. Deletes the item from the stack so it will not be processed again, and then adds all its child items to the stack at the next level (**@level** + 1). This is the only place where the hierarchy table (**#stack**) is used.

With a conventional programming language, you would have to find each child item and add it to the stack individually. With Transact-SQL, you can find all child items and add them with a single statement, avoiding another nested loop.

- If there are child items (IF @@ROWCOUNT > 0), descends one level to process them (@level = @level + 1); otherwise, continues processing at the current level.
- 5. If there are no items on the stack awaiting processing at the current

level, goes back one level to see if there are any awaiting processing at the previous level (**@level** = **@level** - 1). When there is no previous level, the expansion is complete.

Expanding Networks

In a network, an item can have more than one superior. For example, the following data is a representation of airline flights among a number of cities:

Departure	Destination
Chicago	New York
Chicago	Milwaukee
Denver	Chicago
Seattle	Chicago
Seattle	Denver
Seattle	San Francisco

With this data, finding all routes between a given pair of cities is a common problem:

Itineraries

Seattle, Chicago, New York Seattle, Denver, Chicago, New York

To solve this problem, you can make these changes to the example in Expanding **Hierarchies:**

- Two additional input parameters are required: the goal city and the depth-of-search limit.
- The current itinerary is saved in another temporary table and displayed only when a goal is reached.
- To avoid expanding around a cycle in the network, do not expand cities that appear in the current itinerary.

These changes are shown in this example (not from the **pubs** database):

```
CREATE PROCEDURE route
(@current char(20), @dest char(20), @maxlevel int = 5) AS
SET NOCOUNT ON
DECLARE @level int
CREATE TABLE #stack (city char(20), level int)
CREATE TABLE #list (city char(20), level int)
INSERT #stack VALUES (@current, 1)
SELECT @level = 1
WHILE @level > 0
BEGIN
 IF EXISTS (SELECT * FROM #stack WHERE level = @level)
   BEGIN
    SELECT @current = city
    FROM #stack
    WHERE level = @level
    DELETE FROM #stack
    WHERE level = @level
      AND city = @current
    DELETE FROM #list
    WHERE level >= @level
    IF EXISTS (SELECT * FROM #list WHERE city = @current)
      CONTINUE
    INSERT #list VALUES(@current, @level)
    IF(@current = @dest)
    BEGIN
      SELECT city AS itinerary
      FROM #list
      CONTINUE
    END
    INSERT #stack
```

```
SELECT destination, @level + 1
```

```
FROM flights
WHERE departure = @current
AND @level < @maxlevel
IF @@rowcount > 0
SELECT @level = @level + 1
END
ELSE
SELECT @level = @level - 1
END -- WHILE
```

In this example, when **@level** is greater than 0, the procedure follows these steps:

- The current city is added to #list by clearing anything at the current level or below (DELETE FROM #list WHERE level > = @level), and then by adding the current city (INSERT #list VALUES(@current, @level)).
- When the goal city is reached (@current = @dest), the procedure displays the path (SELECT itinerary = city FROM #list) and does not expand the path any further (CONTINUE).
- 3. The depth of search is limited by adding a condition (*@level < @maxlevel*) to the INSERT statement that adds cities to the stack.

The IF EXISTS statement at the beginning of the WHILE loop skips the current city if it is already in the current itinerary.

If the **flights** table also contains cost information, the lowest cost route can be found by saving the current itinerary if its total cost is less than the best cost so far:

```
SELECT @cost = sum(cost)
FROM #list
IF @cost < @lowest_cost
BEGIN
```

```
@lowest_cost = @cost
TRUNCATE TABLE #best_route
INSERT #best_route
SELECT *
FROM #list
END
```

For greater efficiency, stop expanding the current route if the current cost exceeds the cost of the best route:

```
IF (SELECT SUM(cost) FROM #list) > @lowest_cost
CONTINUE
```

If the **flights** table includes a departure and arrival time, you can add an IF statement to expand only the routes that have a departure time at least one hour after the arrival time of the current route:

```
IF ((SELECT SUM(cost) FROM #list) > @lowest_cost)
AND datediff(hh, departuretime, @arrivaltime) > 1)
CONTINUE
```

Writing International Transact-SQL Statements

Databases and database applications that use Transact-SQL statements will become more portable from one language to another, or will support multiple languages, if these guidelines are followed:

- Replace all uses of the **char**, **varchar**, and **text** data types with **nchar**, **nvarchar**, and **ntext**. This eliminates the need to consider code page conversion issues.
- When performing month and day-of-week comparisons and operations, use the numeric dateparts rather than the name strings. Different language settings return different names for the months and week days. For example, DATENAME(MONTH,GETDATE()) returns May when the language is set to U.S. English, returns Mai when the language is set to German, and returns mai when the language is set to French. Instead, use a function such as DATEPART that uses the number of the month instead of the name. Use the DATEPART names when building result sets to be displayed to a user because the date names are often more meaningful than a numeric representation; however, do not code any logic that depends on the displayed names being from a specific language.
- When specifying dates in comparisons or for input to INSERT or UPDATE statements, use constants that are interpreted the same for all language settings:
 - ADO, OLE DB, and ODBC applications should use the ODBC timestamp, date, and time escape clauses of:
 { ts 'yyyy-mm-dd hh:mm:ss[.fff] '} such as: { ts '1998-09-24
 10:02:20' }
 { d 'yyyy-mm-dd'} such as: { d '1998-09-24' }
 { t 'hh:mm:ss'} such as: { t '10:02:20'}
 - Applications using other APIs, or Transact-SQL scripts, stored

procedures, and triggers, should use the unseparated numeric strings (for example, *yyyymmdd* as 19980924).

Applications using other APIs, or Transact-SQL scripts stored procedures, and triggers should use the CONVERT statement with an explicit style parameter for all conversions between the date and smalldate data types and character string data types. For example, this statement is interpreted the same for all language or date format connection settings:
 SELECT *
 FROM Northwind.dbo.Orders
 WHERE OrderDate = CONVERT(DATETIME, '7/19/1)

For more information, see <u>CAST and CONVERT</u>.

Writing Readable Code

Here are guidelines for writing readable code:

- Use comments to describe the program or script, including the author, the date, and a description of the modifications.
- Put each major Transact-SQL clause on a separate line so the statements are easier to read: USE pubs SELECT au_fname, au_lname FROM authors WHERE state = 'CA'
- Put Transact-SQL keywords such as SELECT and FROM, function names such as SUM, AVG, DATEPART, CASE, and CONVERT, and data types such as INT, CHAR, NTEXT in uppercase:
 USE pubs
 CREATE TABLE myauthors
 (
 first VARCHAR(30) NOT NULL,
 last VARCHAR(40) NOT NULL,
 address VARCHAR(40) NOT NULL,
 city VARCHAR(30) NOT NULL,
 state VARCHAR(30) NOT NULL,
 state VARCHAR(2) NOT NULL,
 phone VARCHAR(20) NULL
)
- Define and use a style convention for object names consistently. Two typical conventions are:
 - Capitalize the first letter in each name part; do not separate name parts with underscores: **TableName**.

• Make all characters lowercase and separate name parts with underscore characters (_): **table_name**.

Even if the current instance of Microsoft® SQL Server[™] is not case sensitive, readability is improved if a consistent style is used. It is good practice to always code object names in Transact-SQL statements using the exact same case as was used to define the object.

- For objects that are common in your organization, define a set of standard abbreviations to be used consistently in object names.
- Use single quotation marks for all character, string, binary, and Unicode constants, so that quoted identifiers are the only items that use double quotation marks (").
- Use easy-to-type and easy-to-remember alias names when using multitable joins. For example, an alias of **t** for the **titles** table and an alias of **a** for the **authors** table.
- If the information following a Transact-SQL keyword wraps to another line, consider tabbing the second and successive lines in one tab (usually five spaces) to make it easier to find the major keywords.
- Use parentheses to indicate the execution order of complex mathematical computations. This allows for easier readability. For example, use "(price * 1.15) + sales" instead of "price * 1.15 + sales".

See Also

Batches

Functions

Modifying Data

The topics in this section describe the techniques for manipulating data in the tables of a relational database. You can add new rows of data, change the data in existing rows, and delete rows.

See Also

<u>Adding Data</u> <u>Changing Data</u> <u>Deleting Data</u>

Adding Data

Microsoft[®] SQL Server[™] supports these ways to add data to a table:

- The INSERT statement with one of two options:
 - A VALUES clause to insert one row with a specific set of values.
 - A SELECT subquery to insert data selected from a table or view.
- Database application programming interfaces (APIs)

Support options for inserting data while processing a result set.

- The WRITETEXT statement and several database API options Can be used to add **ntext**, **text**, or **image** data to a row.
- Functions in the database APIs (ADO, OLE DB, ODBC, and DB-Library)

Support adding new **ntext**, **text**, and **image** data to a row.

• The SELECT INTO statement

Can be used to create a new table containing all the rows of the SELECT INTO result set.

• The bulk copy component for inserting large numbers of rows

There are three main ways to specify bulk copy operations:

- The bulk copy program (the **bcp** utility), a command prompt utility.
- The BULK INSERT statement used in Transact-SQL batches, stored procedures, and triggers.

• The bulk copy APIs for OLE DB, ODBC, and DB-Library applications.

INSERT statements work on views as well as on tables, with some restrictions. For more information, see <u>Creating a View</u>.

Adding Rows with INSERT

The INSERT statement adds one or more new rows to a table. In a simplified treatment, INSERT has this form:

INSERT [INTO] table_or_view [(column_list)] data_values

The statement causes the *data_values* to be inserted as one or more rows into the named table or view. *column_list* is a list of column names, separated by commas, that can be used to specify the columns for which data is supplied. If *column_list* is not specified, all the columns in the table or view receive data.

When a *column_list* does not name all the columns in a table or view, a value of NULL (or the default value if a default is defined for the column) is inserted into any column not named in the list. All columns not specified in the column list must either allow null values or have a default assigned.

INSERT statements do not specify values for the following types of columns because Microsoft® SQL ServerTM generates the values for columns of these types:

- Columns with an IDENTITY property that generates the values for the column.
- Columns that have a default that uses the NEWID function to generate a unique GUID value.
- Computed columns.

These are virtual columns that were defined as an expression calculated from one or more other columns in the CREATE TABLE statement, such as:

CREATE TABLE TestTable (ColA INT PRIMARY KEY, ColB INT NOT NULL, ColC AS (ColA + ColB) * 2) The data values supplied must match the column list. The number of data values must be the same as the number of columns, and the data type, precision, and scale of each data value must match those of the corresponding column. There are two ways to specify the data values:

- Use a VALUES clause to specify the data values for one row: INSERT INTO MyTable (PriKey, Description) VALUES (123, 'A description of part 123.')
- Use a SELECT subquery to specify the data values for one or more rows.
 INSERT INTO MyTable (PriKey, Description) SELECT ForeignKey, Description FROM SomeView

Inserting a Row Using INSERT...Values

The VALUES keyword specifies the values for one row of a table. The values are specified as a comma-separated list of scalar expressions whose data type, precision, and scale must be the same as or implicitly convertible to the corresponding column in the column list. If a column list is not specified, the values must be specified in the same sequence as the columns in the table or view.

For example, this statement inserts a new shipper into the **Shippers** table using the VALUES clause:

INSERT INTO Northwind.dbo.Shippers (CompanyName, Phone) VALUES (N'Snowflake Shipping', N'(503)555-7233')

A column list is required for this insert because the **ShipperID** column has the IDENTITY property; therefore, values cannot be inserted into it.

To insert data using INSERT

⊞ <u>Transact-SQL</u>

Inserting Rows Using INSERT...SELECT

The SELECT subquery in the INSERT statement can be used to add values into a table from one or more other tables or views. Using a SELECT subquery also lets more than one row be inserted at one time.

This INSERT statement inserts into a separate table some of the data from all the rows in **titles** whose **type** is modern cooking:

```
USE pubs
INSERT INTO MyBooks
SELECT title_id, title, type
FROM titles
WHERE type = 'mod_cook'
```

The select list of the subquery must match the column list of the INSERT statement. If no column list is specified, the select list must match the columns in the table or view being inserted into.

Another use of the INSERT...SELECT statement is to insert data from a source outside of Microsoft® SQL Server[™]. The SELECT in the INSERT statement can:

- Reference a remote table on a linked server by using a four-part name. For more information, <u>Identifying a Data Source Using a Linked Server</u> <u>Name</u>.
- Reference a remote table using OPENROWSET. For more information, see <u>Identifying a Data Source Using the Ad Hoc Name</u>.
- Use the result set of a query executed on a remote server. For more information, see <u>Using Pass-through Queries as Tables</u>.

Inserting Rows Using SELECT INTO

The SELECT INTO statement creates a new table and populates it with the result set of the SELECT. The structure of the new table is defined by the attributes of the expressions in the select list, for example:

SELECT Shippers.*, Link.Address, Link.City, Link.Region, Link.PostalCode INTO NewShippers FROM Shippers JOIN LinkServer.DB.dbo.Shippers AS Link ON (Shippers.ShipperID = Link.ShipperID)

SELECT INTO can be used to combine data from several tables or views into one table. It can also be used to create a new table containing data selected from a linked server.

See Also

<u>SELECT</u>

Adding a Row Using a Result Set Position

The ADO, OLE DB and ODBC application programming interfaces (APIs) support adding rows while processing the result set of a query. The fundamental process is to:

- 1. Bind the result set columns to program variables.
- 2. Execute the query.
- 3. Execute API functions or methods to position the application on a row within the result set.
- 4. Fill the bound program variables with the data values for the new row to be inserted.
- 5. Execute one of these functions or methods to insert the row:
 - In ADO, call the **AddNew** method of the **Recordset** object.
 - In OLE DB, call the **InsertRow** method of the **IRowsetChange** interface.
 - In ODBC 3.*x*, call the **SQLBulkOperations** function with the SQL_ADD option.

The new row is not necessarily inserted at a position based on the application's position within the result set. The new row is inserted at a position in the base tables related to the values of any clustered key values specified.

Adding ntext, text, or image Data to Inserted Rows

These are ways to add **ntext**, **text**, or **image** values to a row:

- Specify relatively short amounts of data in an INSERT statement in the same way **char**, **nchar**, or **binary** data is.
- Use the WRITETEXT statement. For more information, see <u>WRITETEXT</u>.
- ADO applications can use the **AppendChunk** method to specify long amounts of **ntext**, **text**, or **image** data. For more information, see <u>Managing Long Data Types</u>.
- OLE DB applications can use the **ISequentialStream** interface to write new **ntext**, **text**, or **image** values. For more information, see <u>BLOBs</u> and <u>OLE Objects</u>.
- ODBC applications can use the data-at-execution form of **SQLPutData** to write new **ntext**, **text**, or **image** values. For more information, see <u>Managing text and image Columns</u>.
- DB-Library applications can use the **dbwritetext** function. For more information, see <u>Text and Image Functions</u>.

See Also

BACKUP

<u>UPDATETEXT</u>

INSERT

Adding Rows Using Bulk Copy Operations

The Microsoft[®] SQL Server[™] bulk copy components support:

- Inserting numbers of rows into a table or view.
- Retrieving large numbers of rows from a table, view, or query.

Bulk copy is the fastest way to add large numbers of rows in SQL Server. There are three ways to run bulk copy operations:

• Use the bulk copy program (the **bcp** utility).

bcp is a command prompt utility. **bcp** provides for running bulk copies in .bat and .cmd scripts. **bcp** is used to bulk copy large files into tables or views in SQL Server databases.

• Use the BULK INSERT statement in Transact-SQL batches, stored procedures, and triggers to bulk copy data from a file into a table or view in a SQL Server database.

The BULK INSERT statement is executed on the server in the context of the MSSQLServer service, not on the client. If the file being bulk copied is also on the server the data is not moved across the network at all. This makes a BULK INSERT from a file on the server the fastest bulk copy option. For more information, see <u>BULK INSERT</u>.

• Use the bulk copy APIs for OLE DB, ODBC, and DB-Library applications.

The **bcp** utility is an ODBC command prompt utility that uses the SQL Server ODBC driver bulk copy functions. Any application can use these published bulk copy functions in ODBC or DB-Library applications to run bulk copy operations. Applications can bulk copy from files into a SQL Server table or view. Applications can also bulk copy from program variables into a SQL Server table or view. For more information about OLE DB bulk copies, see <u>Bulk-Copy Rowsets</u>. For more information about ODBC bulk copies, see <u>Performing Bulk Copy</u> <u>Operations</u>. For more information about DB-Library bulk copies, see Bulk-Copy Functions.

To add data using the bcp utility

∃ <u>Client Utility</u>

Changing Data

After the tables have been created and the data added, changing or updating data in the tables becomes one of the day-to-day processes in maintaining a database. Microsoft® SQL Server[™] provides these ways to change data in an existing table:

• The UPDATE statement

Can be used to update data in specific rows in a table or view.

• Database application programming interfaces (APIs)

Support options for updating data at the current position of a result set. Transact-SQL server cursors also support updating data at the current row of a cursor.

• The UPDATETEXT statement

Can be used to update specific **ntext**, **text**, and **image** values.

Updates work on views as well as on tables, with some restrictions. For more information, see <u>Creating a View</u>.

Changing Data with UPDATE

The UPDATE statement can change data values in single rows, groups of rows, or all the rows in a table or view. It can also be used to update rows in a remote server using either a linked server name or the OPENROWSET, OPENDATASOURCE, and OPENQUERY functions, as long as the OLE DB provider used to access the remote server supports updates. An UPDATE statement referencing a table or view can change the data in only one base table at a time.

The UPDATE statement has these major clauses:

• SET

Contains a comma-separated list of the columns to be updated and the new value for each column, in the form *column_name = expression*. The value supplied by the expressions includes items such as constants, values selected from a column in another table or view, or values calculated by a complex expression.

• FROM

Identifies the tables or views that supply the values for the expressions in the SET clause, and optional join conditions between the source tables or views.

• WHERE

Specifies the search condition that defines the rows from the source tables and views that qualify to provide values to the expressions in the SET clause.

This update statement increases the prices of all the **Northwind** products in category 2 by 10 percent:

UPDATE Northwind.dbo.Products SET UnitPrice = UnitPrice * 1.1 WHERE CategoryID = 2

To change data using UPDATE

⊞ <u>Transact-SQL</u>

Note UPDATE is logged; if you are changing large blocks of **text** or **image** data, consider using the UPDATETEXT or WRITETEXT statement, which by default is not logged. For more information, see <u>Adding ntext, text, or image</u> <u>Data to Inserted Rows</u>.

Changing Data Using the SET Clause

SET specifies the columns to be changed and the new values for the columns. The values in the specified columns are updated with the values given in the SET in all rows that match the WHERE clause search condition. If no WHERE clause is specified, all rows are updated. For example, if all the publishing houses in the **publishers** table move their head offices to Atlanta, Georgia, this UPDATE statement would be used:

```
UPDATE publishers SET city = 'Atlanta', state = 'Georgia'
```

Computed column values can be calculated and used in an update. For example, to double all the prices in the **titles** table, the **price** column in the **titles** table can be set to equal **price** * 2.

The expressions used in the SET clause can also be subqueries that return only one value; for example, if the **Northwind** database had an **OrderSummary** table:

```
UPDATE OrderSummary
SET Last30Days =
(SELECT SUM(OrdDet.UnitPrice * OrdDet.Quantity)
FROM [Order Details] AS OrdDet
JOIN Orders AS Ord
ON (OrdDet.OrderID = Ord.OrderID
AND Ord.OrderDate > DATEADD(dd,-30,GETDATE()) )
)
```

Changing Data Using the WHERE Clause

The WHERE clause performs two functions:

- Specifies the rows to be updated.
- Indicates the rows from the source tables that qualify to supply values for the update if a FROM clause is also specified.

If no WHERE clause is specified all rows in the table are updated.

This UPDATE statement implements a name change for one of the shippers:

UPDATE Northwind.dbo.Shippers SET CompanyName = 'United Shippers' WHERE CompanyName = 'United Packages'

Changing Data Using the FROM Clause

Use the FROM clause to pull data from one or more tables or views into the table you want to update. For example, when author Dirk Stringer gets a contract, a title identification number is assigned to his book, *The Psychology of Computer Cooking*, in the **titles** table. Dirk's row in the **titleauthor** table can be updated by adding a title identification number for this latest book.

This example updates Dirk Stringer's row in the **titleauthor** table to add a title identification number for his latest book:

UPDATE titleauthor SET title_id = titles.title_id FROM titles INNER JOIN titleauthor ON titles.title_id = titleauthor.title_id INNER JOIN authors ON titleauthor.au_id = authors.au_id WHERE titles.title = 'Net Etiquette' AND au_lname = 'Locksley'

To update data using UPDATE

⊞ <u>Transact-SQL</u>

See Also

FROM

WHERE

<u>UPDATE</u>

Changing Data with a Cursor

The ADO, OLE DB, and ODBC application programming interfaces (APIs) support updating the current row on which the application is positioned in a result set. The fundamental process is to:

- 1. Bind the result set columns to program variables.
- 2. Execute the query.
- 3. Execute API functions or methods to position the application on a row within the result set.
- 4. Fill the bound program variables with the new data values for any columns to be updated.
- 5. Execute one of these functions or methods to insert the row:
 - In ADO, call the **Update** method of the **Recordset** object.
 - In OLE DB, call the **SetData** method of the **IRowsetChange** interface.
 - In ODBC, call the **SQLSetPos** function with the SQL_UPDATE option.

When using a Transact-SQL server cursor, you can update the current row by using an UPDATE statement that includes a WHERE CURRENT OF clause. Changes made with this clause affect only the row on which the cursor is positioned. When a cursor is based on a join, only the *table_name* specified in the UPDATE statement is modified. Other tables participating in the cursor are not affected.

USE Northwind

GO DECLARE abc CURSOR FOR SELECT CompanyName FROM Shippers

OPEN abc GO

FETCH NEXT FROM abc GO

UPDATE Shippers SET CompanyName = N'Speedy Express, Inc.' WHERE CURRENT OF abc GO

CLOSE abc DEALLOCATE abc GO

For information about joins, see <u>Join Fundamentals</u>.

See Also

<u>UPDATE</u>

Changing ntext, text, or image Data

These are ways to update **ntext**, **text**, or **image** values in a row when replacing the entire value:

- Specify relatively short amounts of data in an UPDATE statement in the same way **char**, **nchar**, or **binary** data is.
- Use the Transact-SQL WRITETEXT statement. For more information, see <u>WRITETEXT</u>.
- ADO applications can use the **AppendChunk** method to specify long amounts of **ntext**, **text**, or **image** data. For more information, see <u>Managing Long Data Types</u>.
- OLE DB applications can use the **ISequentialStream** interface to write new **ntext**, **text**, or **image** values. For more information, see <u>BLOBs</u> <u>and OLE Objects</u>.
- ODBC applications can use the data-at-execution form of **SQLPutData** to write new **ntext**, **text**, or **image** values. For more information, see <u>Managing text and image Columns</u>.
- DB-Library applications can use the **dbwritetext** function. For more information, see <u>Text and Image Functions</u>.

Microsoft® SQL Server[™] also supports updating only a portion of an **ntext**, **text**, or **image** value. In DB-Library this can be done using the **dbupdatetext** function. For more information, see <u>dbupdatetext</u>. All other applications and Transact-SQL scripts, batches, stored procedures, and triggers can use the UPDATETEXT statement to update only a portion of an **ntext**, **text**, or **image** column.

This script shows using UPDATETEXT in conjunction with PATINDEX to find

and replace a specific string in a **text** value:

```
USE Northwind
GO
CREATE TABLE TextParts (ColA INT PRIMARY KEY, ColB TEXT)
GO
INSERT INTO TextParts
 VALUES(1,
     'Sample string START TAG Text to go END TAG Trailing text.'
GO
DECLARE @PtrVar BINARY(16)
DECLARE @InsertPos INT
DECLARE @DeleteLen INT
SELECT @PtrVar = TEXTPTR(ColB),
   @InsertPos = (PATINDEX('%START TAG%', ColB) + 9),
   @DeleteLen = (
           PATINDEX('%END TAG%', ColB) -
           (PATINDEX('%START TAG%', ColB) + 9
               + 2 /* allow for blanks */)
FROM TextParts
WHERE ColA = 1
UPDATETEXT TextParts.ColB
     @PtrVar
     @InsertPos
     @DeleteLen
     WITH LOG
     'The new text'
GO
SELECT * FROM TextParts
GO
```

The result set from the final SELECT statement is:

ColA ColB

1 Sample string START TAG The new text END TAG Trailing t

To update data using UPDATETEXT

Deleting Data

Microsoft[®] SQL Server[™] supports these ways to delete data in an existing table:

- The DELETE statement
- The deletion of the current row in a result set or cursor
- The TRUNCATE TABLE statement

The data modification statements work on views as well as on tables, with some restrictions. For more information, see <u>Creating a View</u>.

Deleting Rows with DELETE

The DELETE statement removes one or more rows in a table or view. A simplified form of the DELETE syntax is:

DELETE table_or_view FROM table_sources WHERE search_condition

table_or_view names a table or view from which the rows are to be deleted. All rows in *table_or_view* that meet the qualifications of the WHERE search condition are deleted. If a WHERE clause is not specified, all the rows in *table_or_view* are deleted. The FROM clause specifies additional tables or views and join conditions that can be used by the predicates in the WHERE clause search condition to qualify the rows to be deleted from *table_or_view*. Rows are not deleted from the tables named in the FROM clause, only from the table named in *table_or_view*.

Any table that has all rows removed remains in the database. The DELETE statement deletes only rows from the table; the table must be removed from the database by using the DROP TABLE statement.

To delete rows using DELETE

⊞ <u>Transact-SQL</u>

This script shows the three DELETE statements needed to delete the rows associated with products supplied by the company named Lyngbysild in the **Northwind** database. This would not be a typical business operation because it involves deleting lines from existing orders, but it does show a series of deletes of differing complexity.

```
USE Northwind
GO
DELETE [Order Details]
FROM Suppliers, Products
WHERE Products.SupplierID = Suppliers.SupplierID
AND Suppliers.CompanyName = 'Lyngbysild'
AND [Order Details].ProductID = Products.ProductID
GO
```

DELETE Products FROM Suppliers WHERE Products.SupplierID = Suppliers.SupplierID AND Suppliers.CompanyName = 'Lyngbysild' GO DELETE Suppliers WHERE CompanyName = 'Lyngbysild' GO

See Also

DROP TABLE

Deleting Rows in Result Sets

The ADO, OLE DB, and ODBC application programming interfaces (APIs) support deleting the current row on which an application is positioned in a result set. The application executes a statement, and then fetches rows from the result set. After an application has fetched the row, it can use the following functions or methods to delete the row:

- ADO applications use the **Delete** method of the **Recordset** object.
- OLE DB applications use the **DeleteRows** method of the **IRowsetChange** interface.
- ODBC applications use the **SQLSetPos** function with the SQL_DELETE option.
- DB-library applications use **dbcursor** to perform a CRS_DELETE operation.

Transact-SQL scripts, stored procedures, and triggers can use the WHERE CURRENT OF clause on a DELETE statement to delete the cursor row on which they are currently positioned, for example:

DECLARE abc CURSOR FOR SELECT * FROM MyTable

OPEN abc

FETCH NEXT FROM abc

DELETE MyTable WHERE CURRENT OF abc

CLOSE abc

DEALLOCATE abc

Deleting All Rows Using TRUNCATE TABLE

The TRUNCATE TABLE statement is a fast, nonlogged method of deleting all rows in a table. It is almost always faster than a DELETE statement with no conditions because DELETE logs each row deletion, and TRUNCATE TABLE logs only the deallocation of whole data pages. TRUNCATE TABLE immediately frees all the space occupied by that table's data and indexes. The distribution pages for all indexes are also freed.

As with DELETE, the definition of a table emptied using TRUNCATE TABLE remains in the database, along with its indexes and other associated objects. The DROP TABLE statement must be used to drop the definition of the table.

To delete all rows in a table using TRUNCATE TABLE

⊞ <u>Transact-SQL</u>

See Also

DROP TABLE

Transactions

A transaction is a sequence of operations performed as a single logical unit of work. A logical unit of work must exhibit four properties, called the ACID (Atomicity, Consistency, Isolation, and Durability) properties, to qualify as a transaction:

Atomicity

A transaction must be an atomic unit of work; either all of its data modifications are performed, or none of them is performed.

Consistency

When completed, a transaction must leave all data in a consistent state. In a relational database, all rules must be applied to the transaction's modifications to maintain all data integrity. All internal data structures, such as B-tree indexes or doubly-linked lists, must be correct at the end of the transaction.

Isolation

Modifications made by concurrent transactions must be isolated from the modifications made by any other concurrent transactions. A transaction either sees data in the state it was in before another concurrent transaction modified it, or it sees the data after the second transaction has completed, but it does not see an intermediate state. This is referred to as serializability because it results in the ability to reload the starting data and replay a series of transactions to end up with the data in the same state it was in after the original transactions were performed.

Durability

After a transaction has completed, its effects are permanently in place in the system. The modifications persist even in the event of a system failure.

Specifying and Enforcing Transactions

SQL programmers are responsible for starting and ending transactions at points that enforce the logical consistency of the data. The programmer must define the

sequence of data modifications that leave the data in a consistent state relative to the organization's business rules. The programmer then includes these modification statements in a single transaction so that Microsoft® SQL ServerTM can enforce the physical integrity of the transaction.

It is the responsibility of an enterprise database system, such as SQL Server, to provide mechanisms ensuring the physical integrity of each transaction. SQL Server provides:

- Locking facilities that preserve transaction isolation.
- Logging facilities that ensure transaction durability. Even if the server hardware, operating system, or SQL Server itself fails, SQL Server uses the transaction logs, upon restart, to automatically roll back any uncompleted transactions to the point of the system failure.
- Transaction management features that enforce transaction atomicity and consistency. After a transaction has started, it must be successfully completed, or SQL Server undoes all of the data modifications made since the transaction started.

Controlling Transactions

Applications control transactions mainly by specifying when a transaction starts and ends. This can be specified using either Transact-SQL statements or database API functions. The system must also be able to correctly handle errors that terminate a transaction before it completes.

Transactions are managed at the connection level. When a transaction is started on a connection, all Transact-SQL statements executed on that connection are part of the transaction until the transaction ends.

Starting Transactions

You can start transactions in Microsoft[®] SQL Server[™] as explicit, autocommit, or implicit transactions.

Explicit transactions

Explicitly start a transaction by issuing a BEGIN TRANSACTION statement.

Autocommit transactions

This is the default mode for SQL Server. Each individual Transact-SQL statement is committed when it completes. You do not have to specify any statements to control transactions.

Implicit transactions

Set implicit transaction mode on through either an API function or the Transact-SQL SET IMPLICIT_TRANSACTIONS ON statement. The next statement automatically starts a new transaction. When that transaction is completed, the next Transact-SQL statement starts a new transaction.

Connection modes are managed at the connection level. If one connection changes from one transaction mode to another it has no effect on the transaction modes of any other connection.

Ending Transactions

You can end transactions with either a COMMIT or ROLLBACK statement.

COMMIT

If a transaction is successful, commit it. A COMMIT statement guarantees all of the transaction's modifications are made a permanent part of the database. A COMMIT also frees resources, such as locks, used by the transaction.

ROLLBACK

If an error occurs in a transaction, or if the user decides to cancel the transaction, then roll the transaction back. A ROLLBACK statement backs out all modifications made in the transaction by returning the data to the state it was in at the start of the transaction. A ROLLBACK also frees resources held by the transaction.

Specifying Transaction Boundaries

You can identify when SQL Server transactions start and end with Transact-SQL statements or API functions and methods.

Transact-SQL statements

Use the BEGIN TRANSACTION, COMMIT TRANSACTION, COMMIT WORK, ROLLBACK TRANSACTION, ROLLBACK WORK, and SET IMPLICIT_TRANSACTIONS statements to delineate transactions. These are primarily used in DB-Library applications and in Transact-SQL scripts, such as the scripts that are run using the **osql** command prompt utility.

API functions and methods

Database APIs such as ODBC, OLE DB, and ADO contain functions or methods used to delineate transactions. These are the primary mechanisms used to control transactions in a SQL Server application.

Each transaction must be managed by only one of these methods. Using both methods on the same transaction can lead to undefined results. For example, you should not start a transaction using the ODBC API functions, and then use the Transact-SQL COMMIT statement to complete the transaction. This would not notify the SQL Server ODBC driver that the transaction was committed. In this case, use the ODBC **SQLEndTran** function to end the transaction.

Errors During Transaction Processing

If a severe error prevents the successful completion of a transaction, SQL Server automatically rolls back the transaction and frees all resources held by the transaction. If the client's network connection to SQL Server is broken, any outstanding transactions for the connection are rolled back when the network notifies SQL Server of the break. If the client application fails or if the client computer goes down or is restarted, this also breaks the connection, and SQL Server rolls back any outstanding connections when the network notifies it of the break. If the client logs off the application, any outstanding transactions are rolled back.

If a run-time statement error (such as a constraint violation) occurs in a batch, the default behavior in SQL Server is to roll back only the statement that generated the error. You can change this behavior using the SET XACT_ABORT statement. After SET XACT_ABORT ON is executed, any run-time statement error causes an automatic rollback of the current transaction. Compile errors, such as syntax errors, are not affected by SET XACT_ABORT.

It is the responsibility of the programmer to code the application to specify the correct action (COMMIT or ROLLBACK) if a run-time or compile error occurs.

See Also

BEGIN TRANSACTIONROLLBACK TRANSACTIONCOMMIT TRANSACTIONROLLBACK WORKCOMMIT WORKSET IMPLICIT_TRANSACTIONSPerforming Transactions in ADOTransactionsPerforming Transactions (ODBC)SET XACT_ABORT

Explicit Transactions

An explicit transaction is one in which you explicitly define both the start and end of the transaction. Explicit transactions were also called user-defined or user-specified transactions in earlier versions of Microsoft® SQL ServerTM.

DB-Library applications and Transact-SQL scripts use the BEGIN TRANSACTION, COMMIT TRANSACTION, COMMIT WORK, ROLLBACK TRANSACTION, or ROLLBACK WORK Transact-SQL statements to define explicit transactions.

BEGIN TRANSACTION

Marks the starting point of an explicit transaction for a connection.

COMMIT TRANSACTION or COMMIT WORK

Used to end a transaction successfully if no errors were encountered. All data modifications made in the transaction become a permanent part of the database. Resources held by the transaction are freed.

ROLLBACK TRANSACTION or ROLLBACK WORK

Used to erase a transaction in which errors are encountered. All data modified by the transaction is returned to the state it was in at the start of the transaction. Resources held by the transaction are freed.

You can also use explicit transactions in OLE DB. Call the

ITransactionLocal::StartTransaction method to start a transaction. Call either the **ITransaction::Commit** or **ITransaction::Abort** method with *fRetaining* set to FALSE to end the transaction without automatically starting another transaction.

In ADO, use the **BeginTrans** method on a **Connection** object to start an explicit transaction. To end the transaction, call the **Connection** object's **CommitTrans** or **RollbackTrans** methods.

The ODBC API does not support explicit transactions, only autocommit and implicit transactions.

Explicit transaction mode lasts only for the duration of the transaction. When the

transaction ends, the connection returns to the transaction mode it was in before the explicit transaction was started, either implicit or autocommit mode.

See Also

BEGIN TRANSACTION ROLLBACK TRANSACTION COMMIT TRANSACTION ROLLBACK WORK COMMIT WORK Performing Transactions in ADO Supporting Local Transactions

Autocommit Transactions

Autocommit mode is the default transaction management mode of Microsoft® SQL ServerTM. Every Transact-SQL statement is committed or rolled back when it completes. If a statement completes successfully, it is committed; if it encounters any error, it is rolled back. A SQL Server connection operates in autocommit mode whenever this default mode has not been overridden by either explicit or implicit transactions. Autocommit mode is also the default mode for ADO, OLE DB, ODBC, and DB-Library.

A SQL Server connection operates in autocommit mode until a BEGIN TRANSACTION statement starts an explicit transaction, or implicit transaction mode is set on. When the explicit transaction is committed or rolled back, or when implicit transaction mode is turned off, SQL Server returns to autocommit mode.

Compile and Run-time Errors

In autocommit mode, it sometimes appears as if SQL Server has rolled back an entire batch instead of just one SQL statement. This happens only if the error encountered is a compile error, not a run-time error. A compile error prevents SQL Server from building an execution plan, so nothing in the batch is executed. Although it appears that all the statements before the one generating the error were rolled back, the error prevented anything in the batch from being executed. In this example, none of the INSERT statements in the third batch are executed because of a compile error. It appears that the first two INSERT statements are rolled back when they are never executed.

USE pubs GO CREATE TABLE TestBatch (Cola INT PRIMARY KEY, Colb CHAR(GO INSERT INTO TestBatch VALUES (1, 'aaa') INSERT INTO TestBatch VALUES (2, 'bbb') INSERT INTO TestBatch VALUSE (3, 'ccc') /* Syntax error */ GO

```
SELECT * FROM TestBatch /* Returns no rows */
GO
```

In this example, the third INSERT statement generates a run-time duplicate primary key error. The first two INSERT statements are successful and committed, so they remain after the run-time error.

```
USE pubs
GO
CREATE TABLE TestBatch (Cola INT PRIMARY KEY, Colb CHAR(
GO
INSERT INTO TestBatch VALUES (1, 'aaa')
INSERT INTO TestBatch VALUES (2, 'bbb')
INSERT INTO TestBatch VALUES (1, 'ccc') /* Duplicate key error */
GO
SELECT * FROM TestBatch /* Returns rows 1 and 2 */
GO
```

SQL Server uses delayed name resolution, in which object names are not resolved until execution time. In this example, the first two INSERT statements are executed and committed, and those two rows remain in the **TestBatch** table after the third INSERT statement generates a run-time error by referring to a table that does not exist.

```
USE pubs
GO
CREATE TABLE TestBatch (Cola INT PRIMARY KEY, Colb CHAR(
GO
INSERT INTO TestBatch VALUES (1, 'aaa')
INSERT INTO TestBatch VALUES (2, 'bbb')
INSERT INTO TestBch VALUES (3, 'ccc') /* Table name error */
GO
SELECT * FROM TestBatch /* Returns rows 1 and 2 */
GO
```

See Also

Transactions

Transactions in ODBC

Implicit Transactions

When a connection is operating in implicit transaction mode, Microsoft® SQL ServerTM automatically starts a new transaction after the current transaction is committed or rolled back. You do nothing to delineate the start of a transaction; you only commit or roll back each transaction. Implicit transaction mode generates a continuous chain of transactions.

After implicit transaction mode has been set on for a connection, SQL Server automatically starts a transaction when it first executes any of these statements:

ALTER TABLE	INSERT
CREATE	OPEN
DELETE	REVOKE
DROP	SELECT
FETCH	TRUNCATE TABLE
GRANT	UPDATE

The transaction remains in effect until you issue a COMMIT or ROLLBACK statement. After the first transaction is committed or rolled back, SQL Server automatically starts a new transaction the next time any of these statements are executed by the connection. SQL Server keeps generating a chain of implicit transactions until implicit transaction mode is turned off.

Implicit transaction mode is set either using the Transact-SQL SET statement, or through database API functions and methods.

Transact-SQL Implicit Transactions

DB-Library applications and Transact-SQL scripts use the Transact-SQL SET IMPLICIT_TRANSACTIONS ON statement to start implicit transaction mode. Use the SET IMPLICIT_TRANSACTIONS OFF statement to turn implicit transaction mode off. Use the COMMIT TRANSACTION, COMMIT WORK, ROLLBACK TRANSACTION, or ROLLBACK WORK statements to end each transaction.

SET QUOTED IDENTIFIER OFF GO SET NOCOUNT OFF GO USE pubs GO CREATE TABLE ImplicitTran (Cola int PRIMARY KEY, Colb char(3) NOT NULL) GO SET IMPLICIT_TRANSACTIONS ON GO /* First implicit transaction started by an INSERT statement */ INSERT INTO ImplicitTran VALUES (1, 'aaa') GO INSERT INTO ImplicitTran VALUES (2, 'bbb') GO /* Commit first transaction */ COMMIT TRANSACTION GO /* Second implicit transaction started by a SELECT statement */ SELECT COUNT(*) FROM ImplicitTran GO INSERT INTO ImplicitTran VALUES (3, 'ccc') GO

SELECT * FROM ImplicitTran GO /* Commit second transaction */ COMMIT TRANSACTION GO SET IMPLICIT_TRANSACTIONS OFF GO

See Also

COMMIT TRANSACTION

ROLLBACK WORK

COMMIT WORK

SET IMPLICIT_TRANSACTIONS

ROLLBACK TRANSACTION

BEGIN TRANSACTION

API Implicit Transactions

The API mechanisms used to set implicit transactions are ODBC and OLE DB. ODBC

- Call the **SQLSetConnectAttr** function with *Attribute* set to SQL_ATTR_AUTOCOMMIT and *ValuePtr* set to SQL_AUTOCOMMIT_OFF to start implicit transaction mode.
- The connection remains in implicit transaction mode until you call **SQLSetConnectAttr** with *Attribute* set to SQL_ATTR_AUTOCOMMIT and *ValuePtr* set to SQL_AUTOCOMMIT_ON.
- Call the **SQLEndTran** function with *CompletionType* set to either SQL_COMMIT or SQL_ROLLBACK to commit or roll back each transaction.
- When SQL_AUTOCOMMIT_OFF is set by an ODBC application, the Microsoft® SQL Server[™] ODBC driver issues a SET IMPLICIT_TRANSACTION ON statement.

OLE DB

OLE DB does not have a method to set implicit transaction mode specifically.

- Call the **ITransactionLocal::StartTransaction** method to start an explicit transaction.
- When you then call either the **ITransaction::Commit** or **ITransaction::Abort** method with *fRetaining* set to TRUE, OLE DB completes the current transaction and goes into implicit transaction mode. The connection remains in implicit transaction mode as long as

you set *fRetaining* on **ITransaction::Commit** or **ITransaction::Abort** to TRUE.

• Call **ITransaction::Commit** or **ITransaction::Abort** with *fRetaining* set to FALSE to stop implicit transaction mode.

ADO

ADO does not support implicit transactions. ADO applications use either autocommit mode or explicit transactions.

See Also

Transactions

Performing Transactions

Distributed Transactions

Distributed transactions span two or more servers known as resource managers. The management of the transaction must be coordinated between the resource managers by a server component called a transaction manager. Microsoft® SQL ServerTM can operate as a resource manager in distributed transactions coordinated by transaction managers such as the Microsoft Distributed Transaction Coordinator (MS DTC), or other transaction managers that support the X/Open XA specification for Distributed Transaction Processing. For more information, see the Microsoft Distributed Transaction Coordinator documentation.

A transaction within a single SQL Server that spans two or more databases is actually a distributed transaction. SQL Server, however, manages the distributed transaction internally; to the user it operates as a local transaction.

At the application, a distributed transaction is managed much the same as a local transaction. At the end of the transaction, the application requests the transaction to be either committed or rolled back. A distributed commit must be managed differently by the transaction manager to minimize the risk that a network failure may result in some resource managers successfully committing while others roll back the transaction. This is achieved by managing the commit process in two phases (the prepare phase and the commit phase), which is known as a two-phase commit (2PC).

Prepare phase

When the transaction manager receives a commit request, it sends a prepare command to all the resource managers involved in the transaction. Each resource manager then does everything required to make the transaction durable and all buffers holding log images for the transaction are flushed to disk. As each resource manager completes the prepare phase, it returns success or failure of the prepare to the transaction manager.

Commit phase

If the transaction manager receives successful prepares from all the resource managers, it sends commit commands to each resource manager. The resource managers can then complete the commit. If all the resource managers report a successful commit, the transaction manager then sends a success notification to the application. If any resource manager reported a failure to prepare, the transaction manager sends a rollback command to each resource manager and indicates the failure of the commit to the application.

SQL Server applications can manage distributed transactions either through Transact-SQL or the database API.

Transact-SQL Distributed Transactions

The distributed transactions started in Transact-SQL have a relatively simple structure:

- 1. A Transact-SQL script or application connection executes a Transact-SQL statement that starts a distributed transaction.
- 2. The Microsoft® SQL Server[™] executing the statement becomes the controlling server in the transaction.
- 3. The script or application then executes either distributed queries against linked servers or remote stored procedures against remote servers.
- 4. As distributed queries and remote procedure calls are made, the controlling server automatically calls MS DTC to enlist the linked and remote servers in the distributed transaction.
- 5. When the script or application issues either a COMMIT or ROLLBACK statement, the controlling SQL Server calls MS DTC to manage the two phase commit process, or to notify the linked and remote servers to roll back their transactions.

Required Transact-SQL Statements

The Transact-SQL statements controlling the distributed transactions are few because most of the work is done internally by Microsoft® SQL Server[™] and MS DTC. The only Transact-SQL statements required in the Transact-SQL script or application are those required to:

- Start a distributed transaction.
- Perform distributed queries against linked servers or execute remote

procedure calls against remote servers.

• Call the standard Transact-SQL COMMIT TRANSACTION, COMMIT WORK, ROLLBACK TRANSACTION, or ROLLBACK WORK statements to complete the transaction.

For any Transact-SQL distributed transaction, the SQL Server processing the Transact-SQL script or connection automatically calls MS DTC to coordinate the commitment or rollback of the transaction.

Starting Distributed Transactions

You can start distributed transactions in Transact-SQL in these ways:

• Start an explicit distributed transaction using the BEGIN DISTRIBUTED TRANSACTION statement.

You can also execute a distributed query against a linked server. The SQL Server you have connected to calls MS DTC to manage the distributed transaction with the linked server. You can also call remote stored procedures on a remote SQL Server as part of the distributed transaction.

• While in a local transaction, execute a distributed query.

If the OLE DB data source supports the **ITransactionJoin** interface, the transaction is promoted to a distributed transaction, even if the query is a read-only query. If the data source does not support **ITransactionJoin**, only read-only statements are allowed.

• If SET REMOTE_PROC_TRANSACTIONS ON has been executed and a local transaction calls a remote stored procedure on another SQL Server, the local transaction is promoted to a distributed transaction.

SQL Server uses MS DTC to coordinate the transaction with the remote server.

• Calls to remote stored procedures execute outside the scope of a local transaction if REMOTE_PROC_TRANSACTIONS is set to OFF. The work done by the remote procedure is not

rolled back if the local transaction is rolled back. The work done by the remote stored procedure is committed at the time the procedure completes, not when the local transaction is committed.

The REMOTE_PROC_TRANSACTIONS option is a compatibility option that affects only remote stored procedure calls made to remote servers defined using **sp_addserver**. For more information about remote stored procedures, see Remote Stored Procedure Architecture. The option does not apply to distributed queries that execute a stored procedure on a linked server defined using **sp_addlinkedserver**. For more information about distributed queries, see Distributed Queries.

See Also

BEGIN DISTRIBUTED TRANSACTION ROLLBACK TRANSACTION COMMIT TRANSACTION ROLLBACK WORK COMMIT WORK SET REMOTE PROC TRANSACTIONS

MS DTC Distributed Transactions

Applications written using OLE DB, ODBC, ADO, or DB-Library can use Transact-SQL distributed transactions by issuing Transact-SQL statements to start and stop Transact-SQL distributed transactions. OLE DB and ODBC, however, also contain support at the API level for managing distributed transactions. OLE DB and ODBC applications can use these API functions to manage distributed transactions that include other COM resource managers that support MS DTC transactions other than Microsoft® SQL Server[™]. They can also use the API functions to gain more control over the boundaries of a distributed transaction that includes several SQL Servers.

ODBC Distributed Transactions

You can control local transactions at the ODBC API level by setting the connection attribute SQL_ATTR_AUTOCOMMIT to SQL_AUTOCOMMIT_OFF, and then by calling the ODBC **SQLEndTran** function to commit or roll back each transaction. Do not use these functions to manage a distributed transaction in an ODBC application. Use the MS DTC COM methods instead:

- Call **DtcGetTransactionManager** to connect to MS DTC.
- Call **ITransactionDispenser::BeginTransaction** to start the distributed transaction and get a transaction object.
- For each ODBC connection participating in the distributed transaction, call the ODBC function SQLSetConnectAttr with *fOption* set to SQL_COPT_SS_ENLIST_IN_DTC and *vParam* holding the address of the transaction object from ITransactionDispenser::BeginTransaction.
- When the transaction is completed, instead of calling the ODBC **SQLEndTran** function, call the **ITransaction::Commit** or **ITransaction::Rollback** methods on the transaction object obtained

from ITransactionDispenser::BeginTransaction.

OLE DB Distributed Transactions

The model for controlling a distributed transaction in OLE DB is similar to controlling a local transaction. To control a local transaction, an OLE DB consumer:

- Uses the **ITransactionLocal::StartTransaction** method to start the local transaction and get a transaction object.
- The consumer then calls the **ITransaction::Commit** or **ITransaction::Rollback** methods on the transaction object obtained by **ITransactionLocal::StartTransaction**.

To control a distributed transaction, the consumer instead:

- Calls **DtcGetTransactionManager** to connect to MS DTC.
- Calls **ITransactionDispenser::BeginTransaction** to start the distributed transaction and get a transaction object.
- Calls the **ITransactionJoin** interface of the distributed transaction object for each connection participating in the distributed transaction.
- Calls the **ITransaction::Commit** or **ITransaction::Rollback** methods of the distributed transaction object to complete the transaction.

See Also

Supporting Distributed Transactions Performing Distributed Transactions

Distributed Queries and Distributed Transactions

Microsoft® SQL Server[™] allows you to create links to OLE DB data sources called linked servers. After linking to an OLE DB data source, you can:

- Reference rowsets from the OLE DB data sources as tables in Transact-SQL statements.
- Pass commands to the OLE DB data sources and include the resulting rowsets as tables in Transact-SQL statements.

Each distributed query can reference multiple linked servers and can perform either update or read operations against each individual linked server. A single distributed query can perform read operations against some linked servers and update operations against other linked servers. In general, Microsoft SQL Server requires distributed transactions support from the corresponding OLE DB provider whenever data from more than one linked server are likely to be updated in a transaction. Hence, the types of queries that are supported against linked servers depend on the level of support for transactions present in the OLE DB providers. OLE DB defines two optional interfaces for transaction management:

- **ITransactionLocal** supports local transactions in the OLE DB data source.
- **ITransactionJoin** lets the provider join a distributed transaction that includes other resource managers.

Any provider that supports **ITransactionJoin** also supports **ITransactionLocal**.

If a distributed query is executed when the connection is in autocommit mode, these rules apply:

• Only read operations are allowed against providers that do not support **ITransactionLocal**.

• All update operations are allowed against any providers that support **ITransactionLocal**.

The controlling SQL Server automatically calls **ITransactionLocal** in each linked server participating in an update operation to start a local transaction, and commits them when the statement succeeds or rolls them back if the statement fails.

If a distributed query is against a distributed partitioned view or if it is executed when the connection is in either an explicit or implicit transaction, these rules apply:

- Only read operations are allowed against providers that do not support **ITransactionJoin**. Providers that do not support any transactions or only support **ITransactionLocal** cannot participate in update operations.
- If SET XACT_ABORT is ON, all update operations are allowed against any providers that support **ITransactionJoin**. The controlling SQL Server automatically calls **ITransactionJoin** in each linked server participating in an update operation to enroll it in the distributed transaction. MS DTC then either commits them or rolls them back when the controlling server indicates the transaction is either committed or rolled back.
- If SET XACT_ABORT is OFF, the linked server must also support nested transactions before update operations are allowed. Nested transactions are supported if the provider supports calling **ITransactionLocal::StartTransaction** when there is already an existing transaction for the session. This allows SQL Server to roll back individual statements in distributed queries without rolling back the entire transaction.

The above rules imply the following restriction for providers that do not support nested transaction: update operations are allowed in a distributed transaction only if the XACT_ABORT option is ON.

See Also

Distributed Queries

Advanced Topics

Mismanagement of transactions often leads to contention and performance problems in systems that have many users. As the number of users in a system increases, it becomes important to have applications that use transactions efficiently. Microsoft® SQL Server[™] also supports nesting transactions, transaction savepoints, and bound transactions, which offer programmers additional options for writing efficient transactions.

Nesting Transactions

Explicit transactions can be nested. This is primarily intended to support transactions in stored procedures that can be called either from a process already in a transaction or from processes that have no active transaction.

The following example shows the intended use of nested transactions. The procedure **TransProc** enforces its transaction regardless of the transaction mode of any process that executes it. If **TransProc** is called when a transaction is active, the nested transaction in **TransProc** is largely ignored, and its INSERT statements are committed or rolled back based on the final action taken for the outer transaction. If **TransProc** is executed by a process that does not have an outstanding transaction, the COMMIT TRANSACTION at the end of the procedure effectively commits the INSERT statements.

```
SET QUOTED IDENTIFIER OFF
GO
SET NOCOUNT OFF
GO
USE pubs
GO
CREATE TABLE TestTrans(Cola INT PRIMARY KEY,
       Colb CHAR(3) NOT NULL)
GO
CREATE PROCEDURE TransProc @PriKey INT, @CharCol CHAR(
BEGIN TRANSACTION InProc
INSERT INTO TestTrans VALUES (@PriKey, @CharCol)
INSERT INTO TestTrans VALUES (@PriKey + 1, @CharCol)
COMMIT TRANSACTION InProc
GO
/* Start a transaction and execute TransProc */
BEGIN TRANSACTION OutOfProc
GO
EXEC TransProc 1, 'aaa'
```

GO

/* Roll back the outer transaction, this will roll back TransProc's nested transaction */
ROLLBACK TRANSACTION OutOfProc GO
EXECUTE TransProc 3,'bbb'
GO
/* The following SELECT statement shows only rows 3 and 4 are still in the table. This indicates that the commit of the inner transaction from the first EXECUTE statement of TransProc was overridden by the subsequent rollback. */
SELECT * FROM TestTrans
GO

Committing inner transactions is ignored by Microsoft® SQL Server[™]. The transaction is either committed or rolled back based on the action taken at the end of the outermost transaction. If the outer transaction is committed, the inner nested transactions are also committed. If the outer transaction is rolled back, then all inner transactions are also rolled back, regardless of whether or not the inner transactions were individually committed.

Each call to COMMIT TRANSACTION or COMMIT WORK applies to the last executed BEGIN TRANSACTION. If the BEGIN TRANSACTION statements are nested, then a COMMIT statement applies only to the last nested transaction, which is the innermost transaction. Even if a COMMIT TRANSACTION *transaction_name* statement within a nested transaction refers to the transaction name of the outer transaction, the commit applies only to the innermost transaction.

It is not legal for the *transaction_name* parameter of a ROLLBACK TRANSACTION statement to refer to the inner transactions of a set of named nested transactions. *transaction_name* can refer only to the transaction name of the outermost transaction. If a ROLLBACK TRANSACTION *transaction_name* statement using the name of the outer transaction is executed at any level of a set of nested transactions, all the nested transactions are rolled back. If a ROLLBACK WORK or ROLLBACK TRANSACTION statement without a *transaction_name* parameter is executed at any level of a set of nested transaction, it rolls back all the nested transactions, including the outermost transaction.

The @@TRANCOUNT function records the current transaction nesting level. Each BEGIN TRANSACTION statement increments @@TRANCOUNT by one. Each COMMIT TRANSACTION or COMMIT WORK statement decrements @@TRANCOUNT by one. A ROLLBACK WORK or a ROLLBACK TRANSACTION statement that does not have a transaction name rolls back all nested transactions and decrements @@TRANCOUNT to 0. A ROLLBACK TRANSACTION that uses the transaction name of the outermost transaction in a set of nested transactions rolls back all the nested transactions and decrements @@TRANCOUNT to 0. When you are unsure if you are already in a transaction, SELECT @@TRANCOUNT to determine if it is 1 or more. If @@TRANCOUNT is 0 you are not in a transaction.

See Also

<u>@@TRANCOUNT</u> <u>COMMIT WORK</u> <u>BEGIN TRANSACTION</u> <u>ROLLBACK TRANSACTION</u> <u>COMMIT TRANSACTION</u> <u>ROLLBACK WORK</u>

Transaction Savepoints

Savepoints offer a mechanism to roll back portions of transactions. You create a savepoint using the SAVE TRANSACTION *savepoint_name* statement, and then later execute a ROLLBACK TRANSACTION *savepoint_name* statement to roll back to the savepoint instead of rolling back to the start of a transaction.

Savepoints are useful in situations where errors are unlikely to occur. The use of a savepoint to roll back part of a transaction in the case of an infrequent error can be more efficient than having each transaction test to see if an update is valid before making the update. Updates and rollbacks are expensive operations, so savepoints are effective only if the probability of encountering the error is low and the cost of checking the validity of an update beforehand is relatively high.

This example shows the use of a savepoint in an order system in which there is a low probability of running out of stock because the company has effective suppliers and reorder points. Usually an application would verify that there is enough stock on hand before attempting to make the updates that would record the order. This example assumes that, for some reason (such as connecting over a slow modem or WAN), first verifying the quantity of stock available is relatively expensive. The application could be coded to just make the update, and if it gets an error indicating that there is not enough stock, it rolls back the update. In this case, a quick check of @@ERROR after the insert is much faster than verifying the amount before the update.

The **InvCtrl** table has a CHECK constraint that triggers a 547 error if the **QtyInStk** column goes below 0. The **OrderStock** procedure creates a savepoint. If a 547 error occurs, it rolls back to the savepoint and returns the number of items on hand to the calling process. The calling process can then decide if it wants to replace the order for the quantity on hand. If **OrderStock** returns a 0, the calling process knows there was enough stock on hand to satisfy the order.

SET NOCOUNT OFF GO USE pubs GO CREATE TABLE InvCtrl

```
(WhrhousID
                int,
   PartNmbr
              int,
   QtyInStk
             int,
   ReordrPt
             int,
   CONSTRAINT InvPK PRIMARY KEY
   (WhrhousID, PartNmbr),
   CONSTRAINT QtyStkCheck CHECK (QtyInStk > 0) )
GO
CREATE PROCEDURE OrderStock @WhrhousID int, @PartNmbr in
      @OrderQty int
AS
DECLARE @ErrorVar int
SAVE TRANSACTION StkOrdTrn
UPDATE InvCtrl SET QtyInStk = QtyInStk - @OrderQty
WHERE Whrhous ID = 1
 AND PartNmbr = 1
SELECT @ErrorVar = @@error
IF (@ErrorVar = 547)
BEGIN
 ROLLBACK TRANSACTION StkOrdTrn
 RETURN (SELECT QtyInStk
     FROM InvCtrl
     WHERE WhrhousID = @WhrhousID
     AND PartNmbr = @PartNmbr)
END
ELSE
 RETURN 0
GO
```

See Also

ROLLBACK TRANSACTION

SAVE TRANSACTION

Using Bound Connections

Bound connections allow two or more connections to share the same transaction and locks. Bound connections can work on the same data without lock conflicts. Bound connections can be created from multiple connections within the same application, or from multiple applications with separate connections. Bound connections make coordinating actions across multiple connections easier.

To participate in a bound connection, a connection calls **sp_getbindtoken** or **srv_getbindtoken** (Open Data Services) to get a bind token. A bind token is a character string that uniquely identifies each bound transaction. The bind token is then sent to the other connections participating in the bound connection. The other connections bind to the transaction by calling **sp_bindsession**, using the bind token received from the first connection.

Bind tokens must be transmitted from the application code that makes the first connection to the application code making any of the subsequent bound connections. There is no Transact-SQL statement or API function that an application can use to get the bind token for a transaction started by another process. Some methods that can be used to transmit a bind token are:

- If the connections are all made from the same application process, bind tokens can be stored in global memory or passed into functions as a parameter.
- If the connections are made from separate application processes, bind tokens can be transmitted using interprocess communication (IPC), such as a remote procedure call (RPC) or dynamic data exchange (DDE).
- Bind tokens can be stored in a table in Microsoft® SQL Server[™] that can be read by processes wanting to bind to the first connection.

Only one connection in a set of bound connections can be active at any time. If one connection is executing a statement on the server or has results pending from the server, no other connections that share the same transaction can access the server until the current connection finishes processing or cancels the current statement. If the server is busy, an error occurs indicating the transaction space is in use and the connection should retry later.

Types of Bound Connections

The two types of bound connections are local and distributed.

• Local bound connection

Allows bound connections to share the transaction space of a single transaction on a single server.

• Distributed bound connection

Allows bound connections to share the same transaction across two or more servers until the entire transaction is either committed or rolled back by using Microsoft Distributed Transaction Coordinator (MS DTC).

Distributed bound connections are not identified by a character string bind token; they are identified by distributed transaction identification numbers. If a bound connection is involved in a local transaction and executes an RPC on a remote server with SET REMOTE_PROC_TRANSACTIONS ON, the local bound transaction is automatically promoted to a distributed bound transaction by MS DTC and an MS DTC session is started.

When to Use Bound Connections

Bound connections are useful in developing extended stored procedures that must execute Transact-SQL statements on behalf of the process that calls them. Having the calling process pass in a bind token as one parameter of the extended stored procedure allows the procedure to join the transaction space of the calling process, thereby integrating the extended stored procedure with the calling process.

Bound connections can be used to develop three-tier applications in which business logic is represented in separate programs that work cooperatively on a single business transaction.

The following example of bound connections illustrates how two connections can access the same transaction: A customer decides to purchase a product at a

local department store. The salesperson accesses a sales transaction system that inserts a row into the sales transaction table, including a credit card authorization number. Two connections are made to the same server, connection C1 and connection C2. C1 begins a transaction that adds a product sale row to the sales table. A credit card authorization number must be added to the new sales transaction row. During the credit card authorization process, the extended stored procedure creates connection C2 to dial out across a telephone line to the credit card company and modifies the sales transaction row with the credit card authorization number. Only by using bound connections can both connections access the same row without locking conflicts.

See Also

<u>sp_bindsession</u> <u>srv_getbindtoken</u> <u>sp_getbindtoken</u> <u>SET REMOTE_PROC_TRANSACTIONS</u>

Adjusting Transaction Isolation Levels

The isolation property is one of the four ACID properties a logical unit of work must display to qualify as a transaction. It is the ability to shield transactions from the effects of updates performed by other concurrent transactions. The level of isolation is actually customizable for each transaction.

Microsoft® SQL Server[™] supports the transaction isolation levels defined in SQL-92. Setting transaction isolation levels allows programmers to trade off increased risk of certain integrity problems with support for greater concurrent access to data. Each isolation level offers more isolation than the previous level, but does so by holding more restrictive locks for longer periods. The transaction isolation levels are:

- READ UNCOMMITTED
- READ COMMITTED
- REPEATABLE READ
- SERIALIZABLE

Transaction isolation levels can be set using Transact-SQL or through a database API:

Transact-SQL

Transact-SQL scripts and DB-Library applications use the SET TRANSACTION ISOLATION LEVEL statement.

ADO

ADO applications set the **IsolationLevel** property of the **Connection** object to adXactReadUncommitted, adXactReadCommitted, adXactRepeatableRead, or adXactReadSerializable.

OLE DB

OLE DB applications call **ITransactionLocal::StartTransaction** with *isoLevel* set to ISOLATIONLEVEL_READUNCOMMITTED, ISOLATIONLEVEL_READCOMMITTED, ISOLATIONLEVEL_REPEATABLEREAD, or ISOLATIONLEVEL_SERIALIZABLE

ODBC

ODBC applications call **SQLSetConnectAttr** with *Attribute* set to SQL_ATTR_TXN_ISOLATION and *ValuePtr* set to SQL_TXN_READ_UNCOMMITTED, SQL_TXN_READ_COMMITTED, SQL_TXN_REPEATABLE_READ, or SQL_TXN_SERIALIZABLE.

See Also

Isolation Levels in SQLOLEDB

SET TRANSACTION ISOLATION LEVEL

Rollbacks in Stored Procedures and Triggers

If @@TRANCOUNT has a different value when a stored procedure finishes than it had when the procedure was executed, an informational error 266 is generated. This error is not generated by the same condition in triggers.

A 266 error is generated when a stored procedure is called with an @@TRANCOUNT of 1 or greater and the procedure executes a ROLLBACK TRANSACTION or ROLLBACK WORK statement. This is because ROLLBACK rolls back all outstanding transactions and decrements @@TRANCOUNT to 0, which is a lower value than it had when the procedure was called.

If a ROLLBACK TRANSACTION is issued in a trigger:

- All data modifications made to that point in the current transaction are rolled back, including any that were made by the trigger.
- The trigger continues executing any remaining statements after the ROLLBACK statement. If any of these statements modify data, the modifications are not rolled back. No nested triggers are fired by the execution of these remaining statements.
- None of the statements in the batch after the statement that fired the trigger are executed.
- A ROLLBACK in a trigger closes and deallocates all cursors that were declared and opened in the batch containing the statement that fired the trigger. This includes cursors declared and opened in stored procedures called by the batch that fired the trigger. Cursors declared in a batch prior to the batch that fired the trigger are only closed, except that STATIC or INSENSITIVE cursors are left open if:
 - CURSOR_CLOSE_ON_COMMIT is set OFF.

• The static cursor is either synchronous, or a fully populated asynchronous cursor.

A trigger always operates as if there were an outstanding transaction in effect when the trigger is executed. This is definitely true if the statement firing the trigger is in an implicit or explicit transaction. It is also true in autocommit mode. When a statement begins executing in autocommit mode, there is an implied BEGIN TRANSACTION to allow the recovery of all modifications generated by the statement if it encounters an error. This implied transaction has no effect on the other statements in the batch because it is either committed or rolled back when the statement completes. This implied transaction is still in effect, however, when a trigger is called.

This means that any time a BEGIN TRANSACTION statement is issued in the trigger, it is actually beginning a nested transaction. Because a nested BEGIN TRANSACTION statement is ignored when rolling back nested transactions, ROLLBACK TRANSACTION issued in the trigger always rolls back past any BEGIN TRANSACTION statements issued by the trigger itself. ROLLBACK rolls back to the outermost BEGIN TRANSACTION.

You must use the SAVE TRANSACTION statement to do a partial rollback in a trigger, even if it is always called in autocommit mode. This is illustrated by the following trigger:

```
CREATE TRIGGER TestTrig ON TestTab FOR UPDATE AS
SAVE TRANSACTION MyName
INSERT INTO TestAudit
SELECT * FROM inserted
IF (@@error <> 0)
BEGIN
ROLLBACK TRANSACTION MyName
END
```

This also affects COMMIT TRANSACTION statements that follow a BEGIN TRANSACTION statement in a trigger. Because BEGIN TRANSACTION starts a nested transaction, a subsequent COMMIT statement applies only to the nested transaction. If a ROLLBACK TRANSACTION statement is executed after COMMIT, ROLLBACK rolls back everything to the outermost BEGIN TRANSACTION. This is illustrated by the following trigger:

CREATE TRIGGER TestTrig ON TestTab FOR UPDATE AS BEGIN TRANSACTION INSERT INTO TrigTarget SELECT * FROM inserted COMMIT TRANSACTION ROLLBACK TRANSACTION

This trigger will never insert into the **TrigTarget** table. BEGIN TRANSACTION always starts a nested transaction. COMMIT TRANSACTION commits only the nested transaction, while the following ROLLBACK TRANSACTION rolls everything back to the outermost BEGIN TRANSACTION.

See Also

@@TRANCOUNT ROLLBACK WORK ROLLBACK TRANSACTION Nesting Transactions

Transact-SQL Statements Allowed in Transactions

You can use all Transact-SQL statements in a transaction, except for the following statements:

ALTER DATABASE	LOAD DATABASE
BACKUP LOG	LOAD TRANSACTION
CREATE DATABASE	RECONFIGURE
DISK INIT	RESTORE DATABASE
DROP DATABASE	RESTORE LOG
DUMP TRANSACTION	UPDATE STATISTICS

Also, you cannot use **sp_dboption** to set database options or use any system procedures that modify the **master** database inside user-defined transactions.

Coding Efficient Transactions

It is important to keep transactions as short as possible. When a transaction is started, a DBMS must hold many resources to the end of the transaction to protect the ACID properties of the transaction. If data is modified, the modified rows must be protected with exclusive locks that prevent any other transaction from reading the rows, and exclusive locks must be held until the transaction is committed or rolled back. Depending on transaction isolation level settings, SELECT statements may acquire locks that must be held until the transaction is committed or rolled back. Especially in systems with many users, transactions must be kept as short as possible to reduce locking contention for resources between concurrent connections. Long-running, inefficient transactions may not be a problem with small numbers of users, but they are intolerable in a system with thousands of users.

Coding Guidelines

These are guidelines for coding efficient transactions:

• Do not require input from users during a transaction.

Get all required input from users before a transaction is started. If additional user input is required during a transaction, roll back the current transaction and restart the transaction after the user input is supplied. Even if users respond immediately, human reaction times are vastly slower than computer speeds. All resources held by the transaction are held for an extremely long time, which has the potential for causing blocking problems. If users do not respond, the transaction remains active and locking critical resources until they respond, which may not happen for several minutes, or even hours.

• Do not open a transaction while browsing through data, if at all possible.

Transactions should not be started until all preliminary data analysis has been completed.

• Keep the transaction as short as possible.

After you know the modifications that have to be made, start a transaction, execute the modification statements, then immediately commit or roll back. Do not open the transaction before it is required.

• Make intelligent use of lower transaction isolation levels.

Many applications can be readily coded to use a read-committed transaction isolation level. Not all transactions require the serializable transaction isolation level.

• Make intelligent use of lower cursor concurrency options, such as optimistic concurrency options.

In a system with a low probability of concurrent updates, the overhead of dealing with an occasional "somebody else changed your data after you read it" error can be much lower than the overhead of always locking rows as they are read.

• Access the least amount of data possible while in a transaction.

This lessens the number of locked rows, thereby reducing contention between transactions.

Avoiding Concurrency Problems

To prevent concurrency problems, manage implicit transactions carefully. When using implicit transactions, the next Transact-SQL statement after COMMIT or ROLLBACK automatically starts a new transaction. This can cause a new transaction to be opened while the application browses through data, or even when it requires input from the user. After completing the last transaction required to protect data modifications, turn off implicit transactions until a transaction is once again required to protect data modifications. This process lets Microsoft® SQL Server[™] use autocommit mode while the application is browsing data and getting input from the user.

Locking

Microsoft® SQL Server[™] 2000 uses locking to ensure transactional integrity and database consistency. Locking prevents users from reading data being changed by other users, and prevents multiple users from changing the same data at the same time. If locking is not used, data within the database may become logically incorrect, and queries executed against that data may produce unexpected results.

Although SQL Server enforces locking automatically, you can design applications that are more efficient by understanding and customizing locking in your applications.

Concurrency Problems

If locking is not available and several users access a database concurrently, problems may occur if their transactions use the same data at the same time. Concurrency problems include:

- Lost or buried updates.
- Uncommitted dependency (<u>dirty read</u>).
- Inconsistent analysis (<u>nonrepeatable read</u>).
- Phantom reads.

Lost Updates

Lost updates occur when two or more transactions select the same row and then update the row based on the value originally selected. Each transaction is unaware of other transactions. The last update overwrites updates made by the other transactions, which results in lost data.

For example, two editors make an electronic copy of the same document. Each editor changes the copy independently and then saves the changed copy, thereby overwriting the original document. The editor who saves the changed copy last overwrites changes made by the first editor. This problem could be avoided if the second editor could not make changes until the first editor had finished.

Uncommitted Dependency (Dirty Read)

Uncommitted dependency occurs when a second transaction selects a row that is being updated by another transaction. The second transaction is reading data that has not been committed yet and may be changed by the transaction updating the row.

For example, an editor is making changes to an electronic document. During the changes, a second editor takes a copy of the document that includes all the

changes made so far, and distributes the document to the intended audience. The first editor then decides the changes made so far are wrong and removes the edits and saves the document. The distributed document contains edits that no longer exist, and should be treated as if they never existed. This problem could be avoided if no one could read the changed document until the first editor determined that the changes were final.

Inconsistent Analysis (Nonrepeatable Read)

Inconsistent analysis occurs when a second transaction accesses the same row several times and reads different data each time. Inconsistent analysis is similar to uncommitted dependency in that another transaction is changing the data that a second transaction is reading. However, in inconsistent analysis, the data read by the second transaction was committed by the transaction that made the change. Also, inconsistent analysis involves multiple reads (two or more) of the same row and each time the information is changed by another transaction; thus, the term nonrepeatable read.

For example, an editor reads the same document twice, but between each reading, the writer rewrites the document. When the editor reads the document for the second time, it has changed. The original read was not repeatable. This problem could be avoided if the editor could read the document only after the writer has finished writing it.

Phantom Reads

Phantom reads occur when an insert or delete action is performed against a row that belongs to a range of rows being read by a transaction. The transaction's first read of the range of rows shows a row that no longer exists in the second or succeeding read, as a result of a deletion by a different transaction. Similarly, as the result of an insert by a different transaction, the transaction's second or succeeding read shows a row that did not exist in the original read.

For example, an editor makes changes to a document submitted by a writer, but when the changes are incorporated into the master copy of the document by the production department, they find that new unedited material has been added to the document by the author. This problem could be avoided if no one could add new material to the document until the editor and production department finish working with the original document.

Optimistic and Pessimistic Concurrency

Microsoft® SQL Server[™] 2000 offers both optimistic and pessimistic concurrency control. Optimistic concurrency control uses cursors. Pessimistic concurrency control is the default for SQL Server.

Optimistic Concurrency

Optimistic concurrency control works on the assumption that resource conflicts between multiple users are unlikely (but not impossible), and allows transactions to execute without locking any resources. Only when attempting to change data are resources checked to determine if any conflicts have occurred. If a conflict occurs, the application must read the data and attempt the change again.

Pessimistic Concurrency

Pessimistic concurrency control locks resources as they are required, for the duration of a transaction. Unless deadlocks occur, a transaction is assured of successful completion.

See Also

Cursor Concurrency

Deadlocking

Isolation Levels

When locking is used as the concurrency control mechanism, it solves concurrency problems. This allows all transactions to run in complete isolation of one another, although there can be more than one transaction running at any time.

Serializability is the database state achieved by running a set of concurrent transactions equivalent to the database state that would be achieved if the set of transactions were executed serially in order.

SQL-92 Isolation Levels

Although serialization is important to transactions to ensure that the data in the database is correct at all times, many transactions do not always require full isolation. For example, several writers are working on different chapters of the same book. New chapters can be submitted to the project at any time. However, after a chapter has been edited, a writer cannot make any changes to the chapter without the editor's approval. This way, the editor can be assured of the accuracy of the book project at any point in time, despite the arrival of new unedited chapters. The editor can see both previously edited chapters and recently submitted chapters.

The level at which a transaction is prepared to accept inconsistent data is termed the isolation level. The isolation level is the degree to which one transaction must be isolated from other transactions. A lower isolation level increases concurrency, but at the expense of data correctness. Conversely, a higher isolation level ensures that data is correct, but can affect concurrency negatively. The isolation level required by an application determines the locking behavior SQL Server uses.

SQL-92 defines the following isolation levels, all of which are supported by SQL Server:

• Read uncommitted (the lowest level where transactions are isolated only enough to ensure that physically corrupt data is not read).

- Read committed (SQL Server default level).
- Repeatable read.
- Serializable (the highest level, where transactions are completely isolated from one another).

If transactions are run at an isolation level of serializable, any concurrent overlapping transactions are guaranteed to be serializable.

Isolation level	Dirty read	Nonrepeatable read	Phantom
Read uncommitted	Yes	Yes	Yes
Read committed	No	Yes	Yes
Repeatable read	No	No	Yes
Serializable	No	No	No

These isolation levels allow different types of behavior.

Transactions must be run at an isolation level of repeatable read or higher to prevent lost updates that can occur when two transactions each retrieve the same row, and then later update the row based on the originally retrieved values. If the two transactions update rows using a single UPDATE statement and do not base the update on the previously retrieved values, lost updates cannot occur at the default isolation level of read committed.

Understanding Locking in SQL Server

Microsoft® SQL Server[™] 2000 has multigranular locking that allows different types of resources to be locked by a transaction. To minimize the cost of locking, SQL Server locks resources automatically at a level appropriate to the task. Locking at a smaller granularity, such as rows, increases concurrency, but has a higher overhead because more locks must be held if many rows are locked. Locking at a larger granularity, such as tables, are expensive in terms of concurrency because locking an entire table restricts access to any part of the table by other transactions, but has a lower overhead because fewer locks are being maintained.

Resource	Description
RID	Row identifier. Used to lock a single row within a table.
Key	Row lock within an index. Used to protect key ranges in serializable transactions.
Page	8 kilobyte –(KB) data page or index page.
Extent	Contiguous group of eight data pages or index pages.
Table	Entire table, including all data and indexes.
DB	Database.

SQL Server can lock these resources (listed in order of increasing granularity).

SQL Server locks resources using different lock modes that determine how the resources can be accessed by concurrent transactions.

SQL Server uses these resource lock modes.

Lock mode	Description
Shared (S)	Used for operations that do not change or update data (read-only operations), such as a SELECT statement.
Update (U)	Used on resources that can be updated. Prevents a common form of deadlock that occurs when multiple sessions are reading, locking, and potentially updating resources later.
Exclusive (X)	Used for data-modification operations, such as INSERT,

(BU)	TABLOCK hint is specified.
Bulk Update	Used when bulk-copying data into a table and the
	table is executing. The types of schema locks are: schema modification (Sch-M) and schema stability (Sch-S).
Schema	Used when an operation dependent on the schema of a
	shared with intent exclusive (SIX).
intent	locks are: intent shared (IS), intent exclusive (IX), and
Intent	Used to establish a lock hierarchy. The types of intent
	UPDATE, or DELETE. Ensures that multiple updates cannot be made to the same resource at the same time.
	LIDDATE or DELETE Ensures that multiple undates

Shared Locks

Shared (S) locks allow concurrent transactions to read (SELECT) a resource. No other transactions can modify the data while shared (S) locks exist on the resource. Shared (S) locks on a resource are released as soon as the data has been read, unless the transaction isolation level is set to repeatable read or higher, or a locking hint is used to retain the shared (S) locks for the duration of the transaction.

Update Locks

Update (U) locks prevent a common form of deadlock. A typical update pattern consists of a transaction reading a record, acquiring a shared (S) lock on the resource (page or row), and then modifying the row, which requires lock conversion to an exclusive (X) lock. If two transactions acquire shared-mode locks on a resource and then attempt to update data concurrently, one transaction attempts the lock conversion to an exclusive (X) lock. The shared-mode-to-exclusive lock conversion must wait because the exclusive lock for one transaction is not compatible with the shared-mode lock of the other transaction; a lock wait occurs. The second transaction attempts to acquire an exclusive (X) locks, and they are each waiting for the other transaction to release its shared-mode lock, a deadlock occurs.

To avoid this potential deadlock problem, update (U) locks are used. Only one

transaction can obtain an update (U) lock to a resource at a time. If a transaction modifies a resource, the update (U) lock is converted to an exclusive (X) lock. Otherwise, the lock is converted to a shared-mode lock.

Exclusive Locks

Exclusive (X) locks prevent access to a resource by concurrent transactions. No other transactions can read or modify data locked with an exclusive (X) lock.

Intent Locks

An intent lock indicates that SQL Server wants to acquire a shared (S) lock or exclusive (X) lock on some of the resources lower down in the hierarchy. For example, a shared intent lock placed at the table level means that a transaction intends on placing shared (S) locks on pages or rows within that table. Setting an intent lock at the table level prevents another transaction from subsequently acquiring an exclusive (X) lock on the table containing that page. Intent locks improve performance because SQL Server examines intent locks only at the table level to determine if a transaction can safely acquire a lock on the table. This removes the requirement to examine every row or page lock on the table to determine if a transaction can lock the entire table.

Intent locks include intent shared (IS), intent exclusive (IX), and shared with intent exclusive (SIX).

Lock mode	Description
Intent shared (IS)	Indicates the intention of a transaction to read some
	(but not all) resources lower in the hierarchy by placing
	S locks on those individual resources.
Intent exclusive	Indicates the intention of a transaction to modify some
(IX)	(but not all) resources lower in the hierarchy by placing
	X locks on those individual resources. IX is a superset
	of IS.
Shared with intent	Indicates the intention of the transaction to read all of
exclusive (SIX)	the resources lower in the hierarchy and modify some
	(but not all) resources lower in the hierarchy by placing
	IX locks on those individual resources. Concurrent IS
	locks at the top-level resource are allowed. For

example, an SIX lock on a table places an SIX lock on the table (allowing concurrent IS locks), and IX locks on the pages being modified (and X locks on the modified rows). There can be only one SIX lock per resource at one time, preventing updates to the resource made by other transactions, although other transactions can read resources lower in the hierarchy by obtaining IS locks at the table level.

Schema Locks

Schema modification (Sch-M) locks are used when a table data definition language (DDL) operation (such as adding a column or dropping a table) is being performed.

Schema stability (Sch-S) locks are used when compiling queries. Schema stability (Sch-S) locks do not block any transactional locks, including exclusive (X) locks. Therefore, other transactions can continue to run while a query is being compiled, including transactions with exclusive (X) locks on a table. However, DDL operations cannot be performed on the table.

Bulk Update Locks

Bulk update (BU) locks are used when bulk copying data into a table and either the **TABLOCK** hint is specified or the **table lock on bulk load** table option is set using **sp_tableoption**. Bulk update (BU) locks allow processes to bulk copy data concurrently into the same table while preventing other processes that are not bulk copying data from accessing the table.

See Also

Deadlocking Cursor Locking Locking Hints

Lock Compatibility

Only compatible lock types can be placed on a resource that is already locked. For example, while an exclusive (X) lock is held, no other transaction can acquire a lock of any kind (shared, update, or exclusive) on that resource until the exclusive (X) lock is released at the end of the first transaction. Alternatively, if a shared (S) lock has been applied to a resource, other transactions can also acquire a shared lock or an update (U) lock on that item, even if the first transaction has not completed. However, other transactions cannot acquire an exclusive lock until the shared lock has been released.

Resource lock modes have a compatibility matrix that shows which locks are compatible with other locks obtained on the same resource (listed in increasing lock strength).

	Existing granted mode					
Requested mode	IS	S	U	IX	SIX	X
Intent shared (IS)	Yes	Yes	Yes	Yes	Yes	No
Shared (S)	Yes	Yes	Yes	No	No	No
Update (U)	Yes	Yes	No	No	No	No
Intent exclusive	Yes	No	No	Yes	No	No
(IX)						
Shared with intent	Yes	No	No	No	No	No
exclusive (SIX)						
Exclusive (X)	No	No	No	No	No	No

Note An intent exclusive (IX) lock is compatible with an IX lock mode because IX means the intention to update only some of the rows rather than all of them. Other transactions that want to read or update some of the rows are also permitted providing they are not the same rows being updated by other transactions.

The schema stability (Sch-S) lock is compatible with all lock modes except the schema modification (Sch-M) lock mode.

The schema modification (Sch-M) lock is incompatible with all lock modes.

The bulk update (BU) lock is compatible only with schema stability (Sch-S) and other bulk update (BU) locks.

Key-Range Locking

Key-range locking solves the phantom read concurrency problem and supports serializable transactions. Key-range locks cover individual records and the ranges between records, preventing phantom insertions or deletions into a set of records accessed by a transaction. Key-range locks are used only on behalf of transactions operating at the serializable isolation level.

Serializability requires that any query executed during a transaction must obtain the same set of rows if it is executed again at some later point within the same transaction. If this query attempts to fetch a row that does not exist, the row must not be inserted by other transactions until the transaction that attempts to access the row completes. If a second transaction were allowed to insert the row, it would appear as a phantom.

If a second transaction attempts to insert a row that resides on a locked data page, page-level locking prevents the phantom row from being added, and serializability is maintained. However, if the row is added to a data page not already locked by the first transaction, a locking mechanism should be in place to prevent the row from being added.

A key-range lock works by covering the index rows and the ranges between those index rows rather than locking the entire base table rows. Because any attempt to insert, update, or delete any row within the range by a second transaction requires a modification to the index, the second transaction is blocked until the first transaction completes because key-range locks cover the index entries.

Key-Range Lock Modes

Key-range locks include both a range and a row component, specified in rangerow format:

- Range represents the lock mode protecting the range between two consecutive index entries.
- Row represents the lock mode protecting the index entry.

• Mode represents the combined lock mode used. Key-range lock modes consist of two parts. The first represents the type of lock used to lock the index range (Range*T*) and the second represents the lock type used to lock a specific key (*K*). The two parts are connected with an underscore (_), such as Range*T_K*.

Range	Row	Mode	Description
RangeS	S	-	Shared range, shared resource lock; serializable range scan.
RangeS	U	0	Shared range, update resource lock; serializable update scan.
RangeI	Null		Insert range, null resource lock; used to test ranges before inserting a new key into an index.
RangeX	X	RangeX_X	Exclusive range, exclusive resource lock; used when updating a key in a range.

Note The internal Null lock mode is compatible with all other lock modes.

Key-range lock modes have a compatibility matrix that shows which locks are compatible with other locks obtained on overlapping keys and ranges.

	Existing granted mode						
Requested							
mode	S	U	X	RangeS_S	RangeS_U	RangeI_N	RangeX_X
Shared (S)	Yes	Yes	No	Yes	Yes	Yes	No
Update (U)	Yes	No	No	Yes	No	Yes	No
Exclusive	No	No	No	No	No	Yes	No
(X)							
RangeS_S	Yes	Yes	No	Yes	Yes	No	No
RangeS_U	Yes	No	No	Yes	No	No	No
RangeI_N	Yes	Yes	Yes	No	No	Yes	No
RangeX_X	No	No	No	No	No	No	No

Conversion Locks

Conversion locks are created when a key-range lock overlaps another lock.

Lock 1	Lock 2	Conversion Lock
S	RangeI_N	RangeI_S
U	RangeI_N	RangeI_U
Х	RangeI_N	RangeI_X
RangeI_N	RangeS_S	RangeX_S
RangeI_N	RangeS_U	RangeX_U

Conversion locks can be observed for a short period of time under different complex circumstances, sometimes while running concurrent processes.

Serializable Range Scan, Singleton Fetch, Delete, and Insert

Key-range locking ensures that these scenarios are serializable:

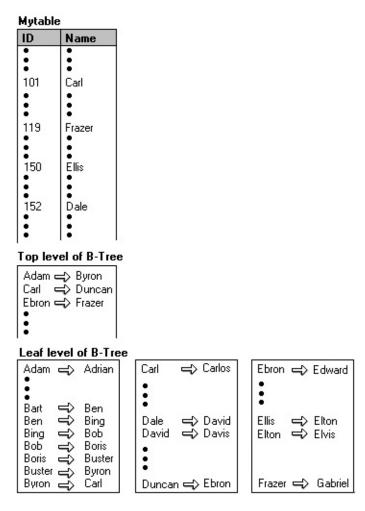
- Range scan query
- Singleton fetch of nonexistent row
- Delete operation
- Insert operation

However, the following conditions must be satisfied before key-range locking can occur:

- The transaction-isolation level must be set to SERIALIZABLE.
- The operation performed on the data must use an index range access. Range locking is activated only when query processing (such as the

optimizer) chooses an index path to access the data.

The following examples for each of the scenarios are based upon this table and index.



Range Scan Query

To ensure a range scan query is serializable, the same query should return the same results each time it is executed within the same transaction. New rows must not be inserted within the range scan query by other transactions; otherwise, these become phantom inserts. For example, the following query uses the table and index in the previous illustration:

SELECT name FROM mytable WHERE name BETWEEN 'A' AND '(

Key-range locks are placed on the index entries corresponding to the range of

data rows where the name is between the values Adam and Dale, preventing new rows qualifying in the previous query from being added or deleted. Although the first name in this range is Adam, the RangeS_S mode key-range lock on this index entry ensures that no new names beginning with the letter A can be added before Adam, such as Abigail. Similarly, the RangeS_S key-range lock on the index entry for Dale ensures that no new names beginning with the letter C can be added after Carlos, such as Clive.

Note The number of RangeS_S locks held is *n*+1, where *n* is the number of rows that satisfy the query.

Singleton Fetch of Nonexistent Data

If a query within a transaction attempts to select a row that does not exist, issuing the query at a later point within the same transaction has to return the same result. No other transaction can be allowed to insert that nonexistent row. For example, given this query:

SELECT name FROM mytable WHERE name = 'Bill'

A key-range lock is placed on the index entry corresponding to the name range from Ben to Bing because the name Bill would be inserted between these two adjacent index entries. The RangeS_S mode key-range lock is placed on the index entry Bing. This prevents any other transaction from inserting values, such as Bill, between the index entries Ben and Bing.

Delete Operation

When deleting a value within a transaction, the range the value falls into does not have to be locked for the duration of the transaction performing the delete operation. Locking the deleted key value until the end of the transaction is sufficient to maintain serializability. For example, given this DELETE statement:

```
DELETE mytable WHERE name = 'Bob'
```

An exclusive (X) lock is placed on the index entry corresponding to the name Bob. Other transactions can insert or delete values before or after the deleted value Bob. However, any transaction attempting to read, insert, or delete the value Bob will be blocked until the deleting transaction either commits or rolls back.

Range delete can be executed using three basic lock modes: row, page, or table lock. The page, table, or row locking strategy is decided by query optimizer, or can be specified by the user through optimizer hints such as ROWLOCK, PAGLOCK, or TABLOCK. In case page or table lock is used, SQL Server immediately releases the index page containing the deleted rows assuming that all rows are deleted from this page. In contrast, when row lock is used, all deleted rows are marked only as deleted; they are removed from the index page later using a background task.

Insert Operation

When inserting a value within a transaction, the range the value falls into does not have to be locked for the duration of the transaction performing the insert operation. Locking the inserted key value until the end of the transaction is sufficient to maintain serializability. For example, given this INSERT statement:

INSERT mytable VALUES ('Dan')

The RangeI_N mode key-range lock is placed on the index entry corresponding to the name David to test the range. If the lock is granted, Dan is inserted and an exclusive (X) lock is placed on the value Dan. The RangeI_N mode key-range lock is necessary only to test the range and is not held for the duration of the transaction performing the insert operation. Other transactions can insert or delete values before or after the inserted value Dan. However, any transaction attempting to read, insert, or delete the value Dan will be locked until the inserting transaction either commits or rolls back.

Lock Escalation

Lock escalation is the process of converting many fine-grain locks into fewer coarse-grain locks, reducing system overhead. Microsoft® SQL ServerTM 2000 automatically escalates row locks and page locks into table locks when a transaction exceeds its escalation threshold.

For example, when a transaction requests rows from a table, SQL Server automatically acquires locks on those rows affected and places higher-level intent locks on the pages and table, or index, which contain those rows. When the number of locks held by the transaction exceeds its threshold, SQL Server attempts to change the intent lock on the table to a stronger lock (for example, an intent exclusive (IX) would change to an exclusive (X) lock). After acquiring the stronger lock, all page and row level locks held by the transaction on the table are released, reducing lock overhead.

SQL Server may choose to do both row and page locking for the same query, for example, placing page locks on the index (if enough contiguous keys in a nonclustered index node are selected to satisfy the query) and row locks on the data. This reduces the likelihood that lock escalation will be necessary.

Lock escalation thresholds are determined dynamically by SQL Server and do not require configuration.

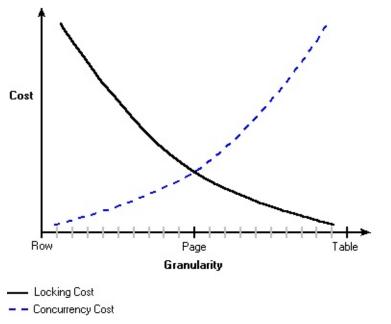
See Also

locks Option

Dynamic Locking

Although row level locks increase concurrency, it is at the cost of system overhead. Table or page locks lower overhead, but at the expense of lowering concurrency.

Dynamic Locking



Microsoft[®] SQL Server[™] 2000 uses a dynamic locking strategy to determine the most cost-effective locks. SQL Server automatically determines what locks are most appropriate when the query is executed, based on the characteristics of the schema and query. For example, to reduce the overhead of locking, the optimizer may choose page-level locks in an index when performing an index scan.

Dynamic locking has the following advantages:

- Simplified database administration. Database administrators no longer have to be concerned with adjusting lock escalation thresholds.
- Increased performance. SQL Server minimizes system overhead by using locks appropriate to the task.

• Application developers can concentrate on development. SQL Server adjusts locking automatically.

Displaying Locking Information

Microsoft® SQL Server[™] 2000 provides a report of the active locks when the **sp_lock** system stored procedure is executed.

Here is the result set.

spid	dbid	ObjId	IndId	Туре	Resource	Mode	Status
1	1	0	0	DB		S	GRANT
6	1	0	0	DB		S	GRANT
7	1	0	0	DB		S	GRANT
8	1	0	0	DB		S	GRANT
8	1	1396200024	0	RID	1:1225:2	Х	GRANT
8	1	1396200024	0	PAG	1:1225	IX	GRANT
8	1	1396200024	2	PAG	1:1240	IX	GRANT
8	1	21575115	0	TAB		IS	GRANT
8	1	1396200024	2	KEY	(03000100cb04)	X	GRANT
8	1	1396200024	0	TAB		IX	GRANT

Type Column

The **Type** column shows the type of the resource currently locked.

Resource type	Description
RID	Row identifier used to lock a single row within a table.
KEY	Row lock within an index. Used to protect key ranges in
	serializable transactions.
PAG	Data or index page.
EXT	Contiguous group of eight data pages or index pages.
TAB	Entire table, including all data and indexes.
DB	Database.

Resource Column

The **Resource** column provides information about the resource being locked.

Resource type	Description
RID	Row identifier of the locked row within the table. The row
	is identified by a fileid:page:rid combination, where rid is
	the row identifier on the page.
KEY	Hexadecimal number used internally by SQL Server.
PAG	Page number. The page is identified by a fileid:page
	combination, where fileid is the fileid in the sysfiles table,
	and page is the logical page number within that file.
EXT	First page number in the extent being locked. The page is
	identified by a fileid:page combination.
TAB	No information is provided because the ObjId column
	already contains the object ID of the table.
DB	No information is provided because the dbid column
	already contains the database ID of the database.

In the result set from **sp_lock**, the RID resource type being locked has a resource description of 1:1225:2. This indicates that row identifier 2, on page number 1225, on fileid 1 has a lock applied to it. For more information, see <u>Troubleshooting Deadlocks</u>.

Mode Column

The **Mode** column describes the type of lock being applied to the resource. The types of locks include any multigranular lock.

Status Column

The **Status** column shows if the lock has been obtained (GRANT), is blocking on another process (WAIT), or is being converted to another lock (CNVT). A lock being converted to another lock is held in one mode, but is waiting to acquire a stronger lock mode (for example, update to exclusive). When diagnosing blocking issues, a CNVT can be considered similar to WAIT.

Other Tools for Monitoring Locking Activity

Using **sp_lock** to display locking information may not always be feasible when many locks are held and released faster than **sp_lock** can display them. In this case, SQL Profiler can be used to monitor and record locking information. Additionally, Windows Performance Monitor can be used to monitor lock activity using the SQL Server Locks Object counter.

To view the current locks

⊞ <u>Transact-SQL</u>

Customizing Locking with SQL Server

Although Microsoft® SQL Server[™] 2000 implements locking automatically, it is possible to customize this in applications by:

- Handling deadlocks and setting the deadlock priority.
- Handling time-outs and setting the lock time-out duration.
- Setting the transaction isolation level.
- Using table-level locking hints with the SELECT, INSERT, UPDATE, and DELETE statements.
- Configuring the locking granularity for an index.

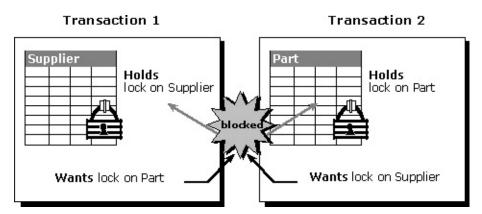
Deadlocking

A deadlock occurs when there is a cyclic dependency between two or more threads for some set of resources.

Deadlock is a condition that can occur on any system with multiple threads, not just on a relational database management system. A thread in a multi-threaded system may acquire one or more resources (for example, locks). If the resource being acquired is currently owned by another thread, the first thread may have to wait for the owning thread to release the target resource. The waiting thread is said to have a dependency on the owning thread for that particular resource.

If the owning thread wants to acquire another resource that is currently owned by the waiting thread, the situation becomes a deadlock: both threads cannot release the resources they own until their transactions are committed or rolled back, and their transactions cannot be committed or rolled back because they are waiting on resources the other owns. For example, thread T1 running transaction 1 has an exclusive lock on the **Supplier** table. Thread T2 running transaction 2 obtains an exclusive lock on the **Part** table, and then wants a lock on the **Supplier** table. Transaction 2 cannot obtain the lock because transaction 1 has it. Transaction 2 is blocked, waiting on transaction 1. Transaction 1 then wants a lock on the **Part** table, but cannot obtain it because transaction 2 has it locked. The transactions cannot release the locks held until the transaction is committed or rolled back. The transactions cannot commit or roll back because they require a lock held by the other transaction to continue.

Note Deadlocking is often confused with normal blocking. When one transaction has a lock on a resource that another transaction wants, the second transaction waits for the lock to be released. By default, SQL Server transactions do not time out (unless LOCK_TIMEOUT is set). The second transaction is blocked, not deadlocked. For more information, see <u>Customizing the Lock Timeout</u>.



In this illustration, thread T1 has a dependency on thread T2 for the **Part** table lock resource. Similarly, thread T2 has a dependency on thread T1 for the **Supplier** table lock resource. Because these dependencies form a cycle, there is a deadlock between threads T1 and T2.

Detecting and Ending Deadlocks

In Microsoft® SQL ServerTM 2000, a single user session may have one or more threads running on its behalf. Each thread may acquire or wait to acquire a variety of resources, such as:

- Locks.
- Parallel query execution-related resources (coordinator, producer, consumer threads associated with an exchange port).
- Threads.
- Memory.

All these resources, except memory, participate in the SQL Server deadlock detection scheme. For memory, SQL Server uses a time-out based mechanism, which is controlled by the **query wait** option in **sp_configure**.

In SQL Server 2000, deadlock detection is performed by a separate thread called the lock monitor thread. The lock monitor thread initiates a deadlock search for a particular thread in one of the following conditions:

- The thread has been waiting for the same resource for a specified period of time. The lock monitor thread periodically wakes up and identifies all the threads waiting on some resource. If these threads continue to wait on the same resource when the lock monitor wakes up again, it initiates a deadlock search for the waiting thread.
- The thread waits on a resource and initiates an eager deadlock search.

SQL Server typically performs periodic deadlock detection only; it does not use the eager mode. Because the number of deadlocks encountered in the system is usually small, periodic deadlock detection helps to reduce the overhead of deadlock detection in the system.

When the lock monitor initiates deadlock search for a particular thread, it identifies the resource on which the thread is waiting. The lock monitor then finds the owner(s) for that particular resource and recursively continues the deadlock search for those threads until it finds a cycle. A cycle identified in this manner forms a deadlock.

After a deadlock is identified, SQL Server ends a deadlock by choosing the thread automatically (the deadlock victim) that can break the deadlock. SQL Server rolls back the deadlock victim's transaction, notifies the thread's application (by returning error message number 1205), cancels the thread's current request, and then allows the transactions of the nonbreaking threads to continue.

Typically, SQL Server chooses the thread running the transaction that is least expensive to undo as the deadlock victim. Alternatively, a user can set the DEADLOCK_PRIORITY of a session to LOW, using the SET statement. The DEADLOCK_PRIORITY option controls how sessions are weighed in deadlock situations. If a session's setting is set to LOW, that session becomes the preferred victim when involved in a deadlock situation.

Identifying Deadlocks

After a deadlock is identified, SQL Server chooses a particular thread as the deadlock victim and returns an error message with a list of resources involved in the deadlock. The deadlock message takes the following form:

Your transaction (process ID #52) was deadlocked on {lock | communi

The threads and resources involved in a deadlock are located in the error log. For more information about how to identify the deadlocked threads and the resources involved in a deadlock, see <u>Troubleshooting Deadlocks</u>.

See Also

SET DEADLOCK_PRIORITY

Handling Deadlocks

When a transaction submitted by an application is chosen as the deadlock victim, the transaction is terminated automatically and rolled back, and error message 1205 is returned to the application. Because any application submitting SQL queries can be chosen as the deadlock victim, applications should have an error handler that can trap error message 1205. If an application does not trap the error, it can proceed unaware that its transaction has been rolled back, and errors can occur.

Implementing an error handler that traps error message 1205 allows an application to handle the deadlock situation and take remedial action (for example, automatically resubmitting the query that was involved in the deadlock). Resubmitting the query automatically can mean that the user does not need to know that a deadlock occurred.

Before resubmitting a query automatically, client programs should pause to give the transaction holding the required locks a chance to complete and release those locks. This minimizes the likelihood of the transaction being deadlocked again as it attempts to obtain those locks.

Note A deadlock does not always cancel the batch in which the error was returned. It is important for the client program to check for errors because a deadlock does not always return a failed return code. In most cases, if a deadlock has occurred and the batch has not been canceled automatically, the application should cancel the current query. If this is not done, SQL Server may still have results pending on the connection that it expects the client to process. If any pending results are not processed, an error occurs when the application next tries to send a command to SQL Server.

See Also

Handling Errors and Messages

Minimizing Deadlocks

Although deadlocks cannot be avoided completely, the number of deadlocks can be minimized. Minimizing deadlocks can increase transaction throughput and reduce system overhead because fewer transactions are:

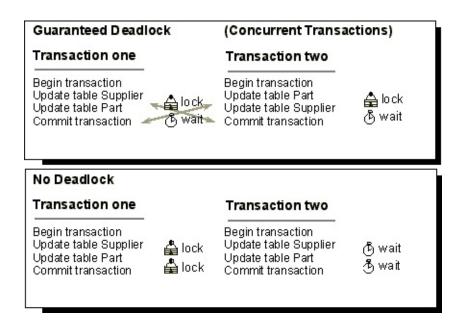
- Rolled back, undoing all the work performed by the transaction.
- Resubmitted by applications because they were rolled back when deadlocked.

To help minimize deadlocks:

- Access objects in the same order.
- Avoid user interaction in transactions.
- Keep transactions short and in one batch.
- Use a low isolation level.
- Use bound connections.

Access Objects in the Same Order

If all concurrent transactions access objects in the same order, deadlocks are less likely to occur. For example, if two concurrent transactions obtain a lock on the **Supplier** table, and then on the **Part** table, one transaction is blocked on the **Supplier** table until the other transaction is completed. After the first transaction commits or rolls back, the second continues. A deadlock does not occur. Using stored procedures for all data modifications can standardize the order of accessing objects.



Avoid User Interaction in Transactions

Avoid writing transactions that include user interaction because the speed of batches running without user intervention is much faster than the speed at which a user can manually respond to queries, such as replying to a prompt for a parameter requested by an application. For example, if a transaction is waiting for user input, and the user goes to lunch, or even home for the weekend, the user holds up the transaction from completing. This degrades system throughput because any locks held by the transaction are released only when the transaction is committed or rolled back. Even if a deadlock situation does not arise, other transactions accessing the same resources are blocked, waiting for the transaction to complete.

Keep Transactions Short and in One Batch

A deadlock typically occurs when several long-running transactions execute concurrently in the same database. The longer the transaction, the longer the exclusive or update locks are held, blocking other activity and leading to possible deadlock situations.

Keeping transactions in one batch minimizes network roundtrips during a transaction, reducing possible delays in completing the transaction and releasing locks.

Use a Low Isolation Level

Determine whether a transaction can run at a lower isolation level. Implementing read committed allows a transaction to read data previously read (not modified) by another transaction without waiting for the first transaction to complete. Using a lower isolation level, such as read committed, holds shared locks for a shorter duration than a higher isolation level such as serializable, thereby reducing locking contention.

Use Bound Connections

Using bound connections, two or more connections opened by the same application can cooperate. Any locks acquired by the secondary connections are held as if they were acquired by the primary connection, and vice versa, and therefore do not block each other.

See Also

Using Bound Connections

Customizing the Lock Time-out

When Microsoft® SQL Server[™] 2000 cannot grant a lock to a transaction on a resource because another transaction already owns a conflicting lock on that resource, the first transaction becomes blocked waiting on that resource. If this causes a deadlock, SQL Server terminates one of the participating transactions (with no time-out involved). If there is no deadlock, the transaction requesting the lock is blocked until the other transaction releases the lock. By default, there is no mandatory time-out period, and no way to test if a resource is locked before locking it, except to attempt to access the data (and potentially get blocked indefinitely).

Note The **sp_who** system stored procedure can be used to determine if a process is being blocked, and who is blocking it.

The LOCK_TIMEOUT setting allows an application to set a maximum time that a statement waits on a blocked resource. When a statement has waited longer than the LOCK_TIMEOUT setting, the blocked statement is canceled automatically, and error message 1222 "Lock request time-out period exceeded" is returned to the application.

However, any transaction containing the statement is not rolled back or canceled by SQL Server. Therefore, the application must have an error handler that can trap error message 1222. If an application does not trap the error, it can proceed unaware that an individual statement within a transaction has been canceled, and errors can occur because statements later in the transaction may depend on the statement that was never executed.

Implementing an error handler that traps error message 1222 allows an application to handle the time-out situation and take remedial action for example, automatically resubmitting the statement that was blocked, or rolling back the entire transaction.

To determine the current LOCK_TIMEOUT setting, execute the @@LOCK_TIMEOUT function, for example:

DECLARE @Timeout int SELECT @Timeout = @@lock_timeout SELECT @Timeout GO

See Also

@@LOCK_TIMEOUT

SET LOCK_TIMEOUT

<u>sp_who</u>

Handling Errors and Messages

Customizing Transaction Isolation Level

By default, Microsoft® SQL Server[™] 2000 operates at an isolation level of READ COMMITTED. However, an application may have to operate at a different isolation level. To make use of either more or less strict isolation levels in applications, locking can be customized for an entire session by setting the isolation level of the session with the SET TRANSACTION ISOLATION LEVEL statement.

When the isolation level is specified, the locking behavior for all SELECT statements in the SQL Server session operates at that isolation level and remains in effect until the session terminates, or until the isolation level is set to another level. For example, to set the transaction isolation level to SERIALIZABLE, ensuring that no phantom rows can be inserted by concurrent transactions into the **authors** table, execute:

USE pubs GO SET TRANSACTION ISOLATION LEVEL SERIALIZABLE GO BEGIN TRANSACTION SELECT au_lname FROM authors

Note The isolation level can be overridden, if necessary, for individual SELECT statements by specifying a table-level locking hint. Specifying a table-level locking hint does not affect other statements in the session. It is recommended that table-level locking hints be used to change the default locking behavior only when absolutely necessary.

To determine the transaction isolation level currently set, use the DBCC USEROPTIONS statement, for example:

USE pubs GO SET TRANSACTION ISOLATION LEVEL REPEATABLE READ GO

DBCC USEROPTIONS GO

Set Option	Value
Textsize	4096
Language	us_english
Dateformat	mdy
Datefirst	7
isolation level	repeatable read

(5 rows affected)

DBCC execution completed. If DBCC printed error messages, see you

See Also

DBCC USEROPTIONS

SET TRANSACTION ISOLATION LEVEL

<u>SELECT</u>

Locking Hints

A range of table-level locking hints can be specified using the SELECT, INSERT, UPDATE, and DELETE statements to direct Microsoft® SQL ServerTM 2000 to the type of locks to be used. Table-level locking hints can be used when a finer control of the types of locks acquired on an object is required. These locking hints override the current transaction isolation level for the session.

Note The SQL Server query optimizer automatically makes the correct determination. It is recommended that table-level locking hints be used to change the default locking behavior only when necessary. Disallowing a locking level can affect concurrency adversely.

Locking hint	Description
HOLDLOCK	Hold a shared lock until completion of the transaction instead of releasing the lock as soon as the required table, row, or data page is no longer required. HOLDLOCK is equivalent to SERIALIZABLE.
NOLOCK	Do not issue shared locks and do not honor exclusive locks. When this option is in effect, it is possible to read an uncommitted transaction or a set of pages that are rolled back in the middle of a read. Dirty reads are possible. Only applies to the SELECT statement.
PAGLOCK	Use page locks where a single table lock would usually be taken.
READCOMMITTED	Perform a scan with the same locking semantics as a transaction running at the READ COMMITTED isolation level. By default, SQL Server 2000 operates at this isolation level.
READPAST	Skip locked rows. This option causes a transaction to skip rows locked by other transactions that would ordinarily appear in the result set, rather than block the transaction

	waiting for the other transactions to release their locks on these rows. The READPAST lock hint applies only to transactions operating at READ COMMITTED isolation and will read only past row-level locks. Applies only to the SELECT statement.
READUNCOMMITTED	Equivalent to NOLOCK.
REPEATABLEREAD	Perform a scan with the same locking semantics as a transaction running at the REPEATABLE READ isolation level.
ROWLOCK	Use row-level locks instead of the coarser- grained page- and table-level locks.
SERIALIZABLE	Perform a scan with the same locking semantics as a transaction running at the SERIALIZABLE isolation level. Equivalent to HOLDLOCK.
TABLOCK	Use a table lock instead of the finer-grained row- or page-level locks. SQL Server holds this lock until the end of the statement. However, if you also specify HOLDLOCK, the lock is held until the end of the transaction.
TABLOCKX	Use an exclusive lock on a table. This lock prevents others from reading or updating the table and is held until the end of the statement or transaction.
UPDLOCK	Use update locks instead of shared locks while reading a table, and hold locks until the end of the statement or transaction. UPDLOCK has the advantage of allowing you to read data (without blocking other readers) and update it later with the assurance that the data has not changed since you last read it.
XLOCK	Use an exclusive lock that will be held until the end of the transaction on all data processed by the statement. This lock can be specified with either PAGLOCK or TABLOCK, in which case the exclusive lock applies to the appropriate level

of granularity.

For example, if the transaction isolation level is set to SERIALIZABLE, and the table-level locking hint NOLOCK is used with the SELECT statement, key-range locks typically used to maintain serializable transactions are not taken.

USE pubs GO SET TRANSACTION ISOLATION LEVEL SERIALIZABLE GO BEGIN TRANSACTION SELECT au_lname FROM authors WITH (NOLOCK) GO

The locks generated are:

EXEC sp_lock GO

spid	dbid	ObjId	IndId	Туре	Resource	Mode	Status
1	1	0	0	DB		S	GRANT
6	1	0	0	DB		S	GRANT
7	1	0	0	DB		S	GRANT
8	4	0	0	DB		S	GRANT
8	4	0	0	DB		S	GRANT
8	4	117575457	0	TAB		Sch-S	GRANT
9	4	0	0	DB		S	GRANT
9	1	21575115	0	TAB		IS	GRANT

SELECT object_name(117575457) GO

authors

The only lock taken that references **authors** is a schema stability (Sch-S) lock. In this case, serializability is no longer guaranteed.

See Also DELETE INSERT SELECT SET TRANSACTION ISOLATION LEVEL UPDATE

Customizing Locking for an Index

The Microsoft® SQL Server[™] 2000 dynamic locking strategy automatically chooses the best locking granularity for queries in most cases. In cases where access patterns are well understood and consistent, limiting the locking levels available for an index can be beneficial.

For example, a database application uses a lookup table that is refreshed weekly in a batch process. The most efficient locking strategy is to turn off page and row locking and allow all concurrent readers to get a shared (S) lock on the table, reducing overhead. During the weekly batch update, the update process can take an exclusive (X) lock, and then update the entire table.

The granularity of locking used on an index can be set using the **sp_indexoption** system stored procedure. To display the current locking option for a given index, use the INDEXPROPERTY function. Page-level locks, row-level locks, or a combination of page-level and row-level locks can be disallowed for a given index.

Disallowed locks	Index accessed by
Page level	Row-level and table-level locks
Row level	Page-level and table-level locks
Page level and row level	Table-level locks

For example, when a table is known to be a point of contention, it can be beneficial to disallow page-level locks, thereby allowing only row-level locks. Or, if table scans are always used to access an index or table, disallowing pagelevel and row-level locks can help by allowing only table-level locks.

IMPORTANT The SQL Server query optimizer automatically makes the correct determination. It is recommended that you do not override the choices the optimizer makes. Disallowing a locking level can affect the concurrency for a table or index adversely. For example, specifying only table-level locks on a large table accessed heavily by many users can affect performance significantly. Users must wait for the table-level lock to be released before accessing the table.

INDEXPROPERTY

sp_indexoption

Cursors

Operations in a relational database act on a complete set of rows. The set of rows returned by a SELECT statement consists of all the rows that satisfy the conditions in the WHERE clause of the statement. This complete set of rows returned by the statement is known as the result set. Applications, especially interactive online applications, cannot always work effectively with the entire result set as a unit. These applications need a mechanism to work with one row or a small block of rows at a time. Cursors are an extension to result sets that provide that mechanism.

Cursors extend result processing by:

- Allowing positioning at specific rows of the result set.
- Retrieving one row or block of rows from the current position in the result set.
- Supporting data modifications to the rows at the current position in the result set.
- Supporting different levels of visibility to changes made by other users to the database data that is presented in the result set.
- Providing Transact-SQL statements in scripts, stored procedures, and triggers access to the data in a result set.

Requesting a Cursor

Microsoft[®] SQL Server[™] 2000 supports two methods for requesting a cursor:

• Transact-SQL

The Transact-SQL language supports a syntax for using cursors modeled after the SQL-92 cursor syntax.

- Database application programming interface (API) cursor functions SQL Server supports the cursor functionality of these database APIs:
 - ADO (Microsoft ActiveX[®] Data Object)
 - OLE DB
 - ODBC (Open Database Connectivity)
 - DB-Library

An application should never mix these two methods of requesting a cursor. An application that has used the API to specify cursor behaviors should not then execute a Transact-SQL DECLARE CURSOR statement to also request a Transact-SQL cursor. An application should only execute DECLARE CURSOR if it has set all the API cursor attributes back to their defaults.

If neither a Transact-SQL nor API cursor has been requested, SQL Server defaults to returning a complete result set, known as a <u>default result set</u>, to the application.

Cursor Process

Transact-SQL cursors and API cursors have different syntax, but the following general process is used with all SQL Server cursors:

- 1. Associate a cursor with the result set of a Transact-SQL statement, and define characteristics of the cursor, such as whether the rows in the cursor can be updated.
- 2. Execute the Transact-SQL statement to populate the cursor.
- 3. Retrieve the rows in the cursor you want to see. The operation to retrieve one row or one block of rows from a cursor is called a fetch. Performing a series of fetches to retrieve rows in either a forward or backward direction is called scrolling.

- 4. Optionally, perform modification operations (update or delete) on the row at the current position in the cursor.
- 5. Close the cursor.

Default Result Sets

Microsoft® SQL Server[™] 2000 sends result sets back to clients in the following way:

- 1. SQL Server receives a network packet from the client containing the Transact-SQL statement or batch of Transact-SQL statements to be executed.
- 2. SQL Server compiles and executes the statement or batch.
- 3. SQL Server begins putting the rows of the result set (or multiple result sets from a batch or stored procedure) in network packets and sending them to the client. SQL Server puts as many result set rows as possible in each packet.
- 4. The packets containing the result set rows are cached in the network buffers of the client. As the client application fetches the rows, the ODBC driver, OLE DB provider, or DB-Library dynamic-link library (DLL) pulls the rows from the network buffers and transfers the data to the client application. The client retrieves the results one row at a time in a forward direction.

The client cannot send any other Transact-SQL statements on that connection until the application has either processed all the rows returned by SQL Server or sent SQL Server a request to cancel the rest of the results. No updates can be done on the connection until all the results have been processed; the result sets are read-only.

Note Firehose cursor is an obsolete term for default result sets.

This is the type of result set processing SQL Server uses when no cursors have been requested. This happens when the following conditions are met:

• The application does not use the DECLARE CURSOR statement to request a Transact-SQL server cursor. The application instead executes

the Transact-SQL statements, such as SELECT, directly.

- If the application uses ADO, OLE DB, and ODBC, it leaves all API cursor attributes at their default settings so that no API cursors are requested. This default set of attributes is to request a forward-only, read-only cursor with a rowset size of 1.
- If the application uses DB-Library, it uses the DB-Library core functions to execute the statement and process the result set.

Because this type of processing is used when all cursor attributes are set to their defaults, and when no cursor processing is actually involved from SQL Server or the database API, this is called a <u>default result set</u>.

A default result set is not given to an application in one large block. The result set is cached in the network buffers on the client. The application fetches through the result set one row at a time. On each fetch, the OLE DB provider, ODBC driver, or DB-Library DLL moves the data from the next row in the network buffer into variables in the application. OLE DB, ODBC, and ADO applications use the same API functions to retrieve the rows that they would use to fetch the rows from a cursor. DB-Library applications use the core function **dbnextrow** to fetch each row.

Default result sets are the most efficient way to transmit results to the client. The only packet sent from the client computer to the server is the original packet with the statement to execute. When the results are sent back to the client, SQL Server puts as many result set rows as it can into each packet, minimizing the number of packets sent to the client.

All Transact-SQL statements are supported when using default result sets. You can also execute batches or stored procedures containing multiple statements that return result sets when using default result sets.

Default result sets can only be used to send result sets back to a client application. The data in a default result set is not available to any other Transact-SQL statement or variable in a batch, stored procedure, or trigger. For example, consider this statement in a stored procedure or trigger:

SELECT ProductID FROM Northwind.dbo.Products

The statement generates a default result set containing the IDs of all the products in the **Northwind** database. None of the other Transact-SQL statements or variables in the stored procedure or trigger can reference this list of product IDs. The only thing that is done with this result set is that SQL Server sends it to the client. For the data in this result set to be used by other Transact-SQL statements, it would have to be in a Transact-SQL server cursor:

DECLARE abc CURSOR FOR SELECT ProductID FROM Northwind.dbo.Products

See Also

Using Default Result Sets

Rowsets and SQL Server Cursors

Using Default Result Sets

Cursor Implementations

Microsoft[®] SQL Server[™] 2000 supports three cursor implementations:

Transact-SQL cursors.

Are based on the DECLARE CURSOR syntax and are used mainly in Transact-SQL scripts, stored procedures, and triggers. Transact-SQL cursors are implemented on the server and are managed by Transact-SQL statements sent from the client to the server. They are also contained in batches, stored procedures, or triggers.

Application programming interface (API) server cursors

Support the API cursor functions in OLE DB, ODBC and DB-Library. API server cursors are implemented on the server. Each time a client application calls an API cursor function, the SQL Server OLE DB provider, ODBC driver, or DB-Library dynamic-link library (DLL) transmits the request to the server for action against the API server cursor.

Client cursors

Are implemented internally by the SQL Server ODBC driver, the DB-Library DLL, and by the DLL that implements the ADO API. Client cursors are implemented by caching all the result set rows on the client. Each time a client application calls an API cursor function, the SQL Server ODBC driver, the DB-Library DLL, or the ADO DLL performs the cursor operation on the result set rows cached on the client.

Because Transact-SQL cursors and API server cursors are implemented on the server, they are referred to collectively as server cursors.

Do not mix the use of these various types of cursors. If you execute a DECLARE CURSOR and OPEN statement from an application, first set the API cursor attributes to their defaults. If you set API cursor attributes to something other than their defaults and then execute a DECLARE CURSOR and OPEN statement, you are asking SQL Server to map an API cursor over a Transact-SQL cursor. For example, do not set the ODBC attributes that call for mapping a keyset-driven cursor over a result set, and then use that statement handle to execute a DECLARE CURSOR and OPEN calling for an INSENSITIVE cursor.

A potential drawback of server cursors is that they currently do not support all Transact-SQL statements. Server cursors do not support Transact-SQL statements that generate multiple result sets; therefore, they cannot be used when the application executes a stored procedure or a batch that contain more than one SELECT statement. Server cursors also do not support SQL statements containing the keywords COMPUTE, COMPUTE BY, FOR BROWSE, or INTO.

Server Cursors vs. Default Result Sets

Using a cursor is less efficient than using a default result set. In a default result set the only packet sent from the client to the server is the packet containing the statement to execute. When using a server cursor, each FETCH statement must be sent from the client to the server, where it must be parsed and compiled into an execution plan.

If a Transact-SQL statement will return a relatively small result set that can be cached in the memory available to the client application, and you know before executing the statement that you must retrieve the entire result set, use a default result set. Use server cursors only when cursor operations are required to support the functionality of the application, or when only part of the result set is likely to be retrieved.

Server Cursors vs. Client Cursors

There are several advantages to using server cursors instead of client cursors:

• Performance

If you are going to access a fraction of the data in the cursor (typical of many browsing applications), using server cursors provides optimal performance because only fetched data is sent over the network. Client cursors cache the entire result set on the client.

• Additional cursor types

If the SQL Server ODBC driver used only client cursors, it could support only forward-only and static cursors. By using API server cursors the driver can also support keyset-driven and dynamic cursors. SQL Server also supports the full range of cursor concurrency attributes only through server cursors. Client cursors are limited in the functionality they support.

• More accurate positioned updates

Server cursors directly support positioned operations, such as the ODBC **SQLSetPos** function or UPDATE and DELETE statements with the WHERE CURRENT OF clause. Client cursors, on the other hand, simulate positioned cursor updates by generating a Transact-SQL searched UPDATE statement, which leads to unintended updates if more than one row matches the WHERE clause conditions of the UPDATE statement.

• Memory usage

When using server cursors, the client does not need to cache large amounts of data or maintain information about the cursor position because the server does that.

• Multiple active statements

When using server cursors, no results are left outstanding on the connection between cursor operations. This allows you to have multiple cursor-based statements active at the same time.

The operation of all server cursors, except static or insensitive cursors, depends on the schema of the underlying tables. Any schema changes to those tables after a cursor has been declared results in an error on any subsequent operation on that cursor.

Specifying Cursors

In the OLE DB, ODBC, and ADO specifications, a cursor is implicitly opened over any result set returned by a Transact-SQL statement. Before executing a Transact-SQL statement, set attributes or properties to define the characteristics of the cursor. You can then call API functions to fetch one row or batch of rows at a time. The default settings for the API cursor attributes or properties have the same characteristics as a SQL Server default result set. The SQL Server ODBC driver and Microsoft OLE DB Provider for SQL Server both implement default result sets when the cursor attributes or properties are set to their defaults. No cursor is used in this case. The driver and provider only implement a cursor if any of the cursor attributes or properties are changed from their defaults.

OLE DB, ODBC, and ADO each use different terms in referring to this combination of a result set and cursor:

- OLE DB uses the term rowset to refer to the combination of a result set and its associated cursor behaviors.
- ODBC uses the terms result set and cursor somewhat interchangeably because it considers each result set to have a cursor automatically mapped over it. A rowset in ODBC is specifically the number of rows returned on a fetch.
- ADO uses the term, recordset, in the same way that OLE DB uses the term, rowset.

The DB-Library API supports two ways to process result sets. The DB-Library core functions support processing the results of a Transact-SQL statement only as a default result set. The core functions allow you only to retrieve the rows of a result set one at a time in a forward-only direction, and do not support making updates through the result set. If you want to use cursor processing in a DB-Library application, you must use the special functions from the DB-Library Cursor Library instead of the core functions.

The default of the Microsoft OLE DB Provider for SQL Server, SQL Server

ODBC driver, and DB-Library is to use an API server cursor to implement any requested cursor functions. ODBC client applications can use client cursors instead of server cursors by loading the ODBC Cursor Library. DB-Library applications can use client cursors by setting the DBCLIENTCURSOR option. OLE DB does not have its own implementation of client cursors, but the ADO API does.

The following illustration summarizes what type of cursor or result set is implemented based on the current API cursor settings and the Transact-SQL statement being executed. The cells with "Do not combine cursor types" indicate that both a Transact-SQL server cursor and an API cursor are being requested at the same time. This can result in undefined behavior and should not be done.

API Cursor Settings	Transact-SQL Statement Executed	
All Cursor Scillings	DECLARE CURSOR	SELECT, Batch Procedure
OLE DB/ODBC/ADO Default cursor settings	Transact-SQL cursor	Default result set
OLE DB/ODBC/ADO Nondefault cursor settings no client cursor library	Do not combine cursor types	API server cursor Batches and procedures not supported
OLE DB/ODBC/ADO Nondefault cursor settings client cursor library loaded	Do not combine cursor types	Default result set cached into client cursor
DB-Library Core Functions	Transact-SQL cursor	Default result set
DB-Library Cursor Functions DBCLIENTCURSOR not set	Do not combine cursor types	API server cursor Batches and procedures not supported
DB-Library Cursor Functions DBCLIENTCURSOR set	Do not combine cursor types	Default result set cached into client cursor

Transact-SQL Cursors

Transact-SQL cursors are used mainly in stored procedures, triggers, and Transact-SQL scripts in which they make the contents of a result set available to other Transact-SQL statements.

The typical process for using a Transact-SQL cursor in a stored procedure or trigger is:

- 1. Declare Transact-SQL variables to contain the data returned by the cursor. Declare one variable for each result set column. Declare the variables to be large enough to hold the values returned by the column and with a data type that can be implicitly converted from the data type of the column.
- 2. Associate a Transact-SQL cursor with a SELECT statement using the DECLARE CURSOR statement. The DECLARE CURSOR statement also defines the characteristics of the cursor, such as the cursor name and whether the cursor is read-only or forward-only.
- 3. Use the OPEN statement to execute the SELECT statement and populate the cursor.
- 4. Use the FETCH INTO statement to fetch individual rows and have the data for each column moved into a specified variable. Other Transact-SQL statements can then reference those variables to access the fetched data values. Transact-SQL cursors do not support fetching blocks of rows.
- 5. When you are finished with the cursor, use the CLOSE statement. Closing a cursor frees some resources, such as the cursor's result set and its locks on the current row, but the cursor structure is still available for processing if you reissue an OPEN statement. Because the cursor is still present, you cannot reuse the cursor name at this

point. The DEALLOCATE statement completely frees all resources allocated to the cursor, including the cursor name. After a cursor is deallocated, you must issue a DECLARE statement to rebuild the cursor.

Monitoring Transact-SQL Cursor Activity

You can use the **sp_cursor_list** system stored procedure to get a list of cursors visible to the current connection, and **sp_describe_cursor**, **sp_describe_cursor_columns**, and **sp_describe_cursor_tables** to determine the characteristics of a cursor.

After the cursor is opened, the @@CURSOR_ROWS function or the **cursor_rows** column returned by **sp_cursor_list** or **sp_describe_cursor** indicates the number of rows in the cursor.

After each FETCH statement, @@FETCH_STATUS is updated to reflect the status of the last fetch. You can also get this status information from the **fetch_status** column returned by **sp_describe_cursor**. @@FETCH_STATUS reports conditions such as fetching beyond the first or last row in the cursor. @@FETCH_STATUS is global to your connection and is reset by each fetch on any cursor open for the connection. If you must know the status later, save @@FETCH_STATUS into a user variable before executing another statement on the connection. Even though the next statement may not be a FETCH, it could be an INSERT, UPDATE or DELETE that fires a trigger containing FETCH statements that reset @@FETCH_STATUS. The **fetch_status** column returned by **sp_describe_cursor** is specific to the cursor specified and is not affected by FETCH statements that reference other cursors. **sp_describe_cursor** is, however, affected by FETCH statements that reference the same cursor, so care is still needed in its use.

After a FETCH is completed, the cursor is positioned on the fetched row. The fetched row is known as the current row. If the cursor was not declared as a read-only cursor, you can execute an UPDATE or DELETE statement with a WHERE CURRENT OF *cursor_name* clause to modify the current row.

The name given to a Transact-SQL cursor by the DECLARE CURSOR statement can be either global or local. Global cursor names are referenced by any batch, stored procedure, or trigger executing on the same connection. Local

cursor names cannot be referenced outside the batch, stored procedure, or trigger in which the cursor is declared. Local cursors in triggers and stored procedures are therefore protected from unintended references outside the stored procedure or trigger.

Using the cursor Variable

Microsoft[®] SQL Server[™] 2000 also supports variables with a **cursor** data type. A cursor can be associated with a **cursor** variable by either of two methods:

/* Use DECLARE @local_variable, DECLARE CURSOR and SET. * DECLARE @MyVariable CURSOR

DECLARE MyCursor CURSOR FOR SELECT LastName FROM Northwind.dbo.Employees

SET @MyVariable = MyCursor

/* Use DECLARE @local_variable and SET */ DECLARE @MyVariable CURSOR

SET @MyVariable = CURSOR SCROLL KEYSET FOR SELECT LastName FROM Northwind.dbo.Employees

After a cursor has been associated with a **cursor** variable, the **cursor** variable can be used instead of the cursor name in Transact-SQL cursor statements. Stored procedure output parameters can also be assigned a **cursor** data type and associated with a cursor. This allows stored procedures to expose local cursors in a controlled manner.

Referencing Transact-SQL Cursors

Transact-SQL cursor names and variables are referenced only by Transact-SQL statements; they cannot be referenced by the API functions of OLE DB, ODBC, ADO, and DB-Library. For example, if you use DECLARE CURSOR and OPEN a Transact-SQL cursor, there is no way to use the ODBC **SQLFetch** or

SQLFetchScroll functions to fetch a row from the Transact-SQL cursor. Applications that need cursor processing and are using these APIs should use the cursor support built into the database API instead of Transact-SQL cursors.

You can use Transact-SQL cursors in applications by using FETCH and binding each column returned by the FETCH to a program variable. The Transact-SQL FETCH does not support batches, however, so this is the least efficient way to return data to an application. Fetching each row requires a roundtrip to the server. It is more efficient to use the cursor functionality built into the database APIs that support fetching batches of rows.

Transact-SQL cursors are extremely efficient when contained in stored procedures and triggers. This is because everything is compiled into one execution plan on the server and there is no network traffic associated with fetching rows.

See Also

@@FETCH_STATUS FETCH CLOSE Cursor Functions OPEN DEALLOCATE WHERE DECLARE CURSOR Scope of Transact-SQL Cursor Names

API Server Cursors

The OLE DB, ODBC, ADO, and DB-Library APIs support mapping cursors over the result sets of executed SQL statements. The Microsoft® SQL Server[™] OLE DB provider, SQL Server ODBC driver, and DB-Library dynamic-link library (DLL) implement these operations through the use of API server cursors. API server cursors are cursors implemented on the server and managed by API cursor functions. As the application calls the API cursor functions, the cursor operation is transmitted to the server by the OLE DB provider, ODBC driver, or DB-Library DLL.

When using an API server cursor in OLE DB, ODBC, and ADO, use the functions or methods of the API to:

- 1. Open a connection.
- 2. Set attributes or properties defining the characteristics of the cursor the API automatically maps over each result set.
- 3. Execute one or more Transact-SQL statements.
- 4. Use API functions or methods to fetch the rows in the result sets.

In DB-Library, use the special DB-Library Cursor Library functions to work with an API server cursor.

When the API cursor attributes or properties are set to their default settings, the SQL Server OLE DB provider and SQL Server ODBC driver use default result sets. Although the API is technically asking for a cursor, the default cursor characteristics match the behavior of a default result set. The OLE DB provider and ODBC driver, therefore, implement the default cursor options using a default result set because it is the most efficient way to retrieve rows from the server. When using default result sets, an application can execute any Transact-SQL statement or batch, but it can only have one outstanding statement on a connection. This means the application must process or cancel all the result sets returned by one statement before it can execute another statement on the

connection.

When the API cursor attributes or properties are set to anything other than their defaults, the OLE DB provider for SQL Server and the SQL Server ODBC driver use API server cursors instead of default result sets. Each call to an API function that fetches rows generates a roundtrip to the server to fetch the rows from the API server cursor.

DB-Library applications use the DB-Library Cursor Library functions to request cursors. If DBCLIENTCURSOR is not set, the DB-Library Cursor Library functions use API server cursors in the same way as the SQL Server OLE DB provider and SQL Server ODBC driver.

API Server Cursor Restrictions

An application cannot execute the following statements when using API server cursors:

- Transact-SQL statements that SQL Server does not support in server cursors.
- Batches or stored procedures that return multiple result sets.
- SELECT statements that contain COMPUTE, COMPUTE BY, FOR BROWSE, or INTO clauses.
- An EXECUTE statement referencing a remote stored procedure.

API Server Cursor Implementation

The OLE DB provider for SQL Server, the SQL Server ODBC driver, and the DB-Library DLL use these special system stored procedures to signal cursor operations to the server:

• **sp_cursoropen** defines the SQL statement to be associated with the cursor and the cursor options, then populates the cursor.

- **sp_cursorfetch** fetches a row or block of rows from the cursor.
- **sp_cursorclose** closes and deallocates the cursor.
- **sp_cursoroption** is used to set various cursor options.
- **sp_cursor** is used to request positioned updates.
- **sp_cursorprepare** compiles the Transact-SQL statement or batch associated with a cursor into an execution plan but does not create the cursor.
- **sp_cursorexecute** creates and populates a cursor from the execution plan created by **sp_cursorprepare**.
- **sp_cursorunprepare** discards the execution plan from **sp_cursorprepare**.

These system stored procedures will show up in SQL Profiler traces of ADO, OLE DB,, ODBC, and DB-Library applications that are using API server cursors. They are intended only for the internal use of the SQL Server Provider for OLE DB, the SQL Server ODBC driver, and the DB-Library DLL. The full functionality of these procedures is available to the applications through the use of the cursor functionality of the database APIs. Specifying the procedures directly in an application is not supported.

When SQL Server executes a statement for a connection, no other statements can be executed on the connection until all the results from the first statement have been processed or canceled. This rule still holds when using API server cursors, but to the application it looks like SQL Server has started supporting multiple active statements on a connection. This is because the full result set is stored in the server cursor and the only statements being transmitted to SQL Server are the executions of the **sp_cursor** system stored procedures. SQL Server executes the stored procedure, and as soon as the client retrieves the result set it can execute any other statement. The OLE DB provider and ODBC driver always retrieve all the results from an **sp_cursor** stored procedure before they return control to the application. This lets applications interleave fetches against multiple active server cursors.

This table shows how an application can process two cursors at the same time on a connection using two statement handles.

Statement handle 1	Statement handle 2
Set cursor attributes such that an	
API server cursor will be used.	
SQLExecDirect an SQL statement.	
The ODBC driver calls	
sp_cursoropen and retrieves the	
result set returned by the procedure.	
	Set cursor attributes such that an API server cursor will be used.
	SQLExecDirect an SQL statement. The ODBC driver calls
	sp_cursoropen and retrieves the result
	set returned by the procedure.
SQLFetchScroll to retrieve the first block of rows. The driver calls sp_cursorfetch and then retrieves the result set returned by the procedure.	
	SQLFetchScroll to retrieve the first
	block of rows. The driver calls
	sp_cursorfetch and then retrieves the
	result set returned by the procedure.
SQLFetchScroll to retrieve another	
block of rows. The driver calls	
sp_cursorfetch and then retrieves	
the result set returned by the	
procedure.	
	SQLFetchScroll to retrieve another block of rows. The driver calls

	sp_cursorfetch and then retrieves the result set returned by the procedure.
Call SQLFreeStmt or	
SQLCloseCursor . The driver calls	
sp_cursorclose.	
	Call SQLFreeStmt or
	SQLCloseCursor. The driver calls
	sp_cursorclose.

Because no results are left outstanding on the connection after any call to an **sp_cursor** stored procedure, you can execute multiple Transact-SQL statements concurrently on a single connection, provided they are all executed with API server cursors.

Specifying API Server Cursors

Here is a summary of how API server cursors are used in the APIs:

- OLE DB
 - a. Open a session object, open a command object, and specify the command text.
 - b. Set rowset properties such as DBPROP_OTHERINSERT, DBPROP_OTHERUPDATEDELETE, DBPROP_OWNINSERT, DBPROP_OWNUDPATEDELETE to control cursor behaviors.
 - c. Execute the command object.
 - d. Fetch the rows in the result set using such methods as IRowset::GetNextRows, IRowsetLocate::GetRowsAt, IRowsetLocate::GetRowsAtBookmark, and IRowsetScroll::GetRowsAtRatio.
- ODBC

- a. Open a connection and call **SQLAllocHandle** to allocate statement handles.
- b. Call SQLSetStmtAttr to set the SQL_ATTR_CURSOR_TYPE, SQL_ATTR_CONCURRENCY, and SQL_ATTR_ROW_ARRAY_SIZE attributes. Alternatively, you can specify cursor behaviors by setting the attributes SQL_ATTR_CURSOR_SCROLLABLE and SQL_ATTR_CURSOR_SENSITIVITY.
- c. Execute a Transact-SQL statement using either **SQLExecDirect** or **SQLPrepare** and **SQLExecute**.
- d. Fetch rows or blocks of rows using **SQLFetch** or **SQLFetchScroll**.
- ADO
- a. Define a Connection object and a Recordset object, and then execute the Open method on the Connection object.
- b. Execute the Open method on the Recordset object specifying a *CursorType* and/or a *LockType* parameter.
- c. Fetch rows using the Move, MoveFirst, MoveLast, MoveNext, and MovePrevious recordset methods.
- DB-Library
 - a. The DB-Library core functions always use a default result set.
 - b. Use the DB-Library Cursor Library functions without setting DBCLIENTCURSOR to use API server cursors.

See Also

Using Cursors with ADO Rowsets and SQL Server Cursors Using Cursors Cursor Functions

Client Cursors

Both ODBC and DB-Library support client cursors, cursors implemented on the client. In a client cursor, a default result set is used to cache the entire result set on the client and all cursor operations are performed against this client cache. None of the server cursor functionality of Microsoft® SQL Server[™] 2000 is used. Client cursors support only forward-only and static cursors, not keyset-driven or dynamic cursors.

The DB-Library client cursors were originally implemented before SQL Server supported server cursors. ODBC implements client cursors that use the ODBC Cursor Library. This is intended for use with ODBC drivers that support only the default settings for cursor characteristics. Because both DB-Library and the SQL Server ODBC driver offer full support for cursor operations through server cursors, limit the use of client cursors.

Client cursors should be used only to alleviate the restriction that server cursors do not support all Transact-SQL statements or batches. If a static scrolling cursor is needed on a Transact-SQL statement or batch that cannot be executed with a server cursor, consider using a client cursor.

See Also

ODBC Cursor Library

Cursor Functions

Fetching and Scrolling

The operation to retrieve a row from a cursor is called a fetch. These are the fetch options:

• FETCH FIRST

Fetches the first row in the cursor.

• FETCH NEXT

Fetches the row after the last row fetched.

• FETCH PRIOR

Fetches the row before the last row fetched.

• FETCH LAST

Fetches the last row in the cursor.

• FETCH ABSOLUTE *n*

Fetches the *n*th row from the first row in the cursor if *n* is a positive integer. If *n* is a negative integer, the row *n* rows before the end of the cursor is fetched. If *n* is 0, no rows are fetched.

• FETCH RELATIVE *n*

Fetches the row n rows from the last row fetched. If n is positive, the row n rows after the last row fetched is fetched. If n is negative, the row n rows before the last row fetched is fetched. If n is 0, the same row is fetched again.

When a cursor is opened, the current row position in the cursor is logically before the first row. This causes the different fetch options to have the following behaviors if they are the first fetch performed after the cursor is opened:

• FETCH FIRST

Fetches the first row in the cursor.

• FETCH NEXT

Fetches the first row in the cursor.

• FETCH PRIOR

Does not fetch a row.

• FETCH LAST

Fetches the last row in the cursor.

• FETCH ABSOLUTE n

Fetches the *n*th row from the first row in the cursor if *n* is a positive integer. If *n* is a negative integer, then the row *n* rows before the end of the cursor is fetched (for example, n = -1 returns the last row in the cursor). If *n* is 0, no rows are fetched.

• FETCH RELATIVE *n*

Fetches the *n*th row in the cursor if *n* is positive. No rows are fetched if *n* is negative or 0.

Transact-SQL cursors are limited to fetching one row at a time. API server cursors support fetching blocks of rows with each fetch. A cursor that supports fetching multiple rows at a time is called a block cursor.

Cursor Classifications

A cursor can be classified by the fetch options it supports:

• Forward-only

Rows must be fetched serially from the first row to the last row. FETCH NEXT is the only fetch operation allowed.

• Scrollable

Rows can be randomly fetched from anywhere in the cursor. All the fetch operations are allowed (except that dynamic cursors do not support fetch absolute).

Scrollable cursors are especially useful for supporting online applications. A cursor can be mapped to a grid or list box in the application. As the user scrolls

up and down and all around the grid, the application uses scroll fetches to retrieve the rows from the cursor the user wants to see.

APIs for Fetching Rows

The APIs for the actual statements, functions, or methods used have different names to fetch rows:

- Transact-SQL cursors use the FETCH FIRST, FETCH LAST, FETCH NEXT, FETCH PRIOR, FETCH ABSOLUTE(*n*), and FETCH RELATIVE(*n*) statements.
- OLE DB uses methods such as IRowset::GetNextRows, IRowsetLocate::GetRowsAt, IRowsetLocate::GetRowsAtBookmark, and IRowsetScroll::GetRowsAtRatio.
- ODBC uses the **SQLFetch** function, which is the same as a FETCH NEXT for one row, or the **SQLFetchScroll** function. **SQLFetchScroll** supports block cursors and all the fetch options (first, last, next, prior, absolute, relative).
- ADO uses the **Move**, **MoveFirst**, **MoveLast**, **MoveNext**, and **MovePrevious Recordset** methods to acquire a position in a cursor. The **GetRows** recordset method is then used to retrieve one or more rows at that position. **GetRows** can also be called directly with the *Start* parameter set to the number of the row to fetch.
- DB-Library uses the **dbcursorfetch** and **dbcursorfetchex** functions.

See Also

Scrolling and Retrieving Rows Fetching Rows Cursor Functions

FETCH

Using Cursors

Cursor Functions

Controlling Cursor Behavior

There are two models for specifying the behavior of a cursor:

Cursor types

The database APIs usually specify the behavior of cursors by dividing them into four cursor types: forward-only, static (sometimes called snapshot or insensitive), keyset-driven, and dynamic.

Cursor behaviors

The SQL-92 standard defines the DECLARE CURSOR keywords SCROLL and INSENSITIVE to specify the behavior of cursors. Some database APIs also support defining cursor behavior in terms of scrollability and sensitivity.

ADO and DB-Library support specifying only cursor types, not cursor behaviors.

ODBC supports specifying cursor behavior using either the cursor types or the cursor behaviors of scrollability and insensitivity.

Prior to Microsoft® SQL Server[™] version 7.0, the DECLARE CURSOR statement used to define Transact-SQL cursors supported only cursor behaviors of SCROLL and INSENSITIVE. In SQL Server 7.0, DECLARE CURSOR has been extended to support cursor-type keywords.

OLE DB's cursor behavior model differs from both cursor behaviors and cursor types.

Do not specify both cursor types and cursor behaviors for a cursor. Use one or the other. Because ODBC and Transact-SQL cursors support both cursor behaviors and cursor types, use either ODBC or Transact-SQL when defining the cursor. The ODBC specification states that specifying both cursor behaviors and cursor types can lead to unpredictable results.

Cursor Types

ODBC, ADO, and DB-Library define four cursor types supported by Microsoft® SQL Server[™]2000. The DECLARE CURSOR statement has been extended; thus you can specify the four cursor types for Transact-SQL cursors. These cursors vary in their ability to detect changes to the result set and in the resources, such as memory and space in **tempdb**, they consume. A cursor can detect changes to rows only when it attempts to fetch those rows a second time. There is no way for the data source to notify the cursor of changes to the currently fetched rows. The ability of a cursor to detect changes is also influenced by the transaction isolation level.

The four API server cursor types supported by SQL Server are:

- Static cursors
- Dynamic cursors
- Forward-only cursors
- Keyset-driven cursors

Static cursors detect few or no changes but consume relatively few resources while scrolling, although they store the entire cursor in **tempdb**. Dynamic cursors detect all changes but consume more resources while scrolling, although they make the lightest use of **tempdb**. Keyset-driven cursors lie in between, detecting most changes but at less expense than dynamic cursors.

Although the database API cursor models consider a forward-only cursor to be a distinct type of cursor, SQL Server does not. SQL Server considers both forward-only and scroll as options that can be applied to static, keyset-driven, and dynamic cursors.

See Also

DECLARE CURSOR Using Server Cursors with ADO Rowsets and SQL Server Cursors How Cursors Are Implemented dbcursoropen

Forward-only Cursors

A forward-only cursor does not support scrolling; it supports only fetching the rows serially from the start to the end of the cursor. The rows are not retrieved from the database until they are fetched. The effects of all INSERT, UPDATE, and DELETE statements made by the current user or committed by other users that affect rows in the result set are visible as the rows are fetched from the cursor. Because the cursor cannot be scrolled backward, however, changes made to rows in the database after the row was fetched are not visible through the cursor, except for the case where a value used to determine the location of the row within the result set is modified, such as updating a column covered by a clustered index.

Although the database API cursor models consider a forward-only cursor to be a distinct type of cursor, Microsoft® SQL ServerTM 2000 does not. SQL Server considers both forward-only and scroll as options that can be applied to static, keyset-driven, and dynamic cursors. Transact-SQL cursors support forward-only static, keyset-driven, and dynamic cursors. The database API cursor models assume that static, keyset-driven, and dynamic cursors are always scrollable. When a database API cursor attribute or property is set to forward-only, SQL Server implements this as a forward-only dynamic cursor.

Fast Forward-only Cursors

Microsoft® SQL Server[™] 2000 implements a performance optimization called a fast forward-only cursor. Fast forward-only cursors are supported in two environments:

- Transact-SQL cursors can specify the FAST_FORWARD clause on DECLARE CURSOR. This opens an optimized forward-only, read-only cursor.
- Applications using the Microsoft OLE DB Provider for SQL Server can set the rowset properties DBPROP_SERVERCURSOR, DBPROP_OTHERINSERT, DBPROP_OTHERUPDATEDELETE, DBPROP_OWNINSERT, and DBPROP_OWNUPDATEDELETE to VARIANT_TRUE.
- Applications using the Microsoft SQL Server ODBC driver can set the driver-specific statement attribute SQL_SOPT_SS_CURSOR_OPTIONS to SQL_CO_FFO or SQL_CO_FFO_AF. Setting SQL_CO_FFO requests that the cursor be opened with the same optimizations as the FAST_FORWARD clause on DECLARE CURSOR. SQL_CO_FFO_AF request that an autofetch option also be enabled.

Using the Autofetch Option

Although some performance improvements are realized by specifying FAST_FORWARD on DECLARE CURSOR, or by specifying SQL_CO_FFO in ODBC applications, the most important performance gain comes from specifying SQL_CO_FFO_AF in ODBC applications to enable the autofetch option. Autofetch enables two optimizations that can significantly reduce network traffic:

• When the cursor is opened, the first row or batch of rows is

automatically fetched from the cursor. This saves having to send a fetch request across the network.

• When a fetch hits the end of the cursor, the cursor is automatically closed. This saves having to send a separate close request across the network.

The most dramatic improvement is seen when processing cursors with relatively small result sets that can be cached in the memory of an application. The fast forward-only cursor with autofetch enabled represents the most efficient method of getting a result set into an ODBC application. When the autofetch option is on for a cursor containing *n* rows, an ODBC application can:

- Specify a rowset size of *n*+1.
- Allocate arrays of *n*+1 variables to hold the data from the result set columns.
- Bind the result set columns to the arrays.
- Execute the SQL statement that generates the cursor.

When the SQL Server ODBC driver executes the statement, it requests that the cursor be opened. Because autofetch is enabled, the server fetches and sends back *n* rows. The server fits as many rows as possible into each network packet returned to the client. When the server attempts to fetch the row at *n*+1 it detects the end of the cursor and automatically closes the cursor. When the application then executes **SQLCloseCursor** or **SQLFreeStmt** the ODBC driver does not have to send any close request to the server. The entire operation is done with only one packet being sent from the client to the server, and a minimal number of packets being returned from the server to the client.

Implicit Conversion of Fast Forward-only Cursors

Fast forward-only cursors are implicitly converted to other cursor types when:

- If the SELECT statement joins one or more tables with triggers to tables without triggers, the cursor is converted to a static cursor.
- If the SELECT statement references **text**, **ntext**, or **image** columns the cursor is converted to a dynamic cursor if the OLE DB Provider for SQL Server or the SQL Server ODBC driver are used.
- If a fast forward-only cursor is not read-only, it is converted to a dynamic cursor.
- If the SELECT statement is a distributed query that references one or more remote tables on linked servers, the cursor is converted to a keyset-driven cursor.
- If the SELECT statement references **text**, **ntext**, or **image** columns and a TOP clause, the cursor is converted to a keyset-driven cursor.

See Also

Rowsets and SQL Server Cursors

Fast Forward-Only Cursors (ODBC)

Static Cursors

The complete result set of a static cursor is built in **tempdb** when the cursor is opened. A static cursor always displays the result set as it was when the cursor was opened.

The cursor does not reflect any changes made in the database that affect either the membership of the result set or changes to the values in the columns of the rows that make up the result set. A static cursor does not display new rows inserted in the database after the cursor was opened, even if they match the search conditions of the cursor SELECT statement. If rows making up the result set are updated by other users, the new data values are not displayed in the static cursor. The static cursor displays rows deleted from the database after the cursor was opened. No UPDATE, INSERT, or DELETE operations are reflected in a static cursor (unless the cursor is closed and reopened), not even modifications made using the same connection that opened the cursor.

Microsoft[®] SQL Server[™] static cursors are always read-only.

Because the result set of a static cursor is stored in a work table in **tempdb**, the size of the rows in the result set cannot exceed the maximum row size for a SQL Server table.

Transact-SQL and DB-Library use the term insensitive for static cursors. Some database APIs identify them as snapshot cursors.

Keyset-driven Cursors

The membership and order of rows in a keyset-driven cursor are fixed when the cursor is opened. Keyset-driven cursors are controlled by a set of unique identifiers (keys) known as the keyset. The keys are built from a set of columns that uniquely identify the rows in the result set. The keyset is the set of the key values from all the rows that qualified for the SELECT statement at the time the cursor was opened. The keyset for a keyset-driven cursor is built in **tempdb** when the cursor is opened.

Changes to data values in nonkeyset columns (made by the cursor owner or committed by other users) are visible as the user scrolls through the cursor. Inserts to the database made outside the cursor are not visible in the cursor unless the cursor is closed and reopened. Inserts made through the cursor using an API function such as the ODBC **SQLSetPos** function are visible at the end of the cursor. @@FETCH_STATUS returns a "row missing" status when an attempt is made to fetch a row deleted after the cursor was opened. An update to a key column operates like a delete of the old key value followed by an insert of the new key value. The new key value is not visible if the update was made through the cursor; it is visible at the end of the cursor if the update was made through the cursor using either an API function such as **SQLSetPos** or the Transact-SQL WHERE CURRENT OF clause and the SELECT statement did not contain a JOIN condition in the FROM clause. The new key value is not visible if the insert contained a remote table in the FROM clause. Attempts to retrieve the old key value get the same missing row fetch status as a deleted row.

Note An index on computed columns cannot be used when declaring a keysetdriven cursor on a remote table. You can create another index to provide the unique keys for that remote table.

Dynamic Cursors

Dynamic cursors are the opposite of static cursors. Dynamic cursors reflect all changes made to the rows in their result set when scrolling through the cursor. The data values, order, and membership of the rows in the result set can change on each fetch. All UPDATE, INSERT, and DELETE statements made by all users are visible through the cursor. Updates are visible immediately if they are made through the cursor using either an API function such as **SQLSetPos** or the Transact-SQL WHERE CURRENT OF clause. Updates made outside the cursor are not visible until they are committed, unless the cursor transaction isolation level is set to read uncommitted.

Cursor Behaviors

Both ODBC and Transact-SQL cursors support specifying cursor characteristics using the SQL-92 cursor behaviors of scrollability and sensitivity.

Cursor behaviors are specified in the following ways:

- Transact-SQL cursors use the SQL-92 syntax of specifying SCROLL and INSENSITIVE before the CURSOR keyword on the DECLARE statement.
- The ODBC API uses the SQL_ATTR_CURSOR_SCROLLABLE and SQL_ATTR_CURSOR_SENSITIVITY statement attributes.

See Also

DECLARE CURSOR

How Cursors Are Implemented

Scrollable

The scrollable behavior of a cursor defines the fetch options the cursor supports.

If the SCROLL keyword is specified on a DECLARE statement, or if SQL_ATTR_CURSOR_SCROLLABLE is set to SQL_SCROLLABLE, the cursor supports all of the fetch options.

If the SCROLL keyword is omitted from a SQL-92 style DECLARE statement or if SQL_ATTR_CURSOR_SCROLLABLE is set to SQL_NONSCROLLABLE, the cursor supports only fetch next operations.

See Also

DECLARE CURSOR

<u>SQLSetStmtAttr</u>

Sensitivity

The sensitivity behavior of a cursor defines whether updates made against the base rows (used to build the cursor) are visible through the cursor. Sensitivity also defines whether updates can be made through the cursor.

If the INSENSITIVE keyword is specified on a Transact-SQL DECLARE statement, or if SQL_ATTR_CURSOR_SENSITIVITY is set to either SQL_UNSPECIFIED or SQL_INSENSITIVE in ODBC, the cursor does not reflect data modifications. The cursor is read-only and does not support updates.

If the INSENSITIVE keyword is omitted from a Transact-SQL DECLARE statement, or if SQL_ATTR_CURSOR_SENSITIVITY is set to SQL_SENSITIVE in ODBC, the cursor reflects data modifications made by the current user or committed by other users. Positioned updates can be made using the cursor, except when using a read-only cursor.

See Also

DECLARE CURSOR SQLSetStmtAttr

Cursor Locking

In Microsoft® SQL Server[™], the SELECT statement in a cursor definition is subject to the same transaction locking rules that apply to any other SELECT statement. In cursors, however, an additional set of scroll locks can be acquired based on the specification of a cursor concurrency level.

The transaction locks acquired by any SELECT statement, including the SELECT statement in a cursor definition, are controlled by:

- The transaction isolation level setting for the connection.
- Any locking hints specified in the FROM clause.

These locks are held until the end of the current transaction for both cursors and independent SELECT statements. When SQL Server is running in autocommit mode, each individual SQL statement is a transaction and the locks are freed when the statement finishes. If SQL Server is running in explicit or implicit transaction mode, then the locks are held until the transaction is either committed or rolled back.

For example, the locking done for these two Transact-SQL examples is essentially the same:

```
/* Example 1 */
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ
GO
BEGIN TRANSACTION
GO
SELECT * FROM authors
GO
```

/* Example 2 */ SET TRANSACTION ISOLATION LEVEL REPEATABLE READ GO BEGIN TRANSACTION

```
GO
DECLARE abc CURSOR STATIC FOR
SELECT * FROM authors
GO
OPEN abc
GO
```

Setting the transaction isolation level to repeatable read means that both the independent SELECT statement in Example 1 and the SELECT statement contained in the DECLARE CURSOR of Example 2 generate share locks on each row they read, and the share locks are held until the transaction is committed or rolled back.

Acquiring Locks

Although cursors obey the same rules as independent SELECT statements, regarding the type of transaction locks acquired, the locks are acquired at different times. The locks generated by an independent SELECT or a cursor are always acquired when a row is retrieved. For an independent SELECT, all the rows are retrieved when the statement is executed. Cursors, however, retrieve the rows at different times depending on the type of cursor:

- Static cursors retrieve the entire result set at the time the cursor is opened. This locks each row of the result set at open time.
- Keyset-driven cursors retrieve the keys of each row of the result set at the time the cursor is opened. This locks each row of the result set at open time.
- Dynamic cursors (including regular forward-only cursors) do not retrieve rows until they are fetched. Locks are not acquired on the rows until they have been fetched.
- Fast forward-only cursors vary in when they acquire their locks depending on the execution plan chosen by the query optimizer. If a

dynamic plan is chosen, no locks are taken until the rows are fetched. If worktables are generated, then the rows are read into the worktable and locked at open time.

Cursors also support their own concurrency specifications, some of which generate additional locks on the rows in each fetch. These scroll locks are held until the next fetch operation or until the cursor is closed, whichever comes first. If the connection option to keep cursors open on a commit is set on, these locks will be kept across a commit or rollback operation.

Cursors and Transactions

Microsoft® SQL Server[™] 2000 supports setting either connection or database options to control whether cursors are closed or left open on commits and rollbacks.

If the option is set that cursors are closed on commits or rollbacks, then all scroll locks are automatically freed when the cursor closes. If the option is set that cursors remain open on a commit, then any active scroll locks are kept until the next fetch or until the cursor closes. All transaction locks, even those on rows in a cursor, are freed when the transaction is committed or rolled back, regardless of whether the cursors stay open.

See Also

SET CURSOR_CLOSE_ON_COMMIT

sp_dboption

Cursor Concurrency

Microsoft® SQL Server[™] 2000 supports four concurrency options for server cursors:

- READ_ONLY
- OPTIMISTIC WITH VALUES
- OPTIMISTIC WITH ROW VERSIONING
- SCROLL LOCKS

READ_ONLY:

Positioned updates through the cursor are not allowed, and no locks are held on the rows that make up the result set.

OPTIMISTIC WITH VALUES

Optimistic concurrency control is a standard part of transaction control theory. Optimistic concurrency control is used in situations when there is only a slight chance that a second user may update a row in the interval between when a cursor is opened and when the row is updated. When a cursor is opened with this option, no locks are held on the underlying rows, which helps maximize throughput. If the user attempts to modify a row, the current values in the row are compared with the values retrieved when the row was last fetched. If any of the values have changed, the server knows that someone else has already updated the row, and it returns an error. If the values are the same, the server performs the modification.

Selecting this concurrency option forces the user or programmer to accept the responsibility of dealing with the occasional error indicating another user has modified the row. A typical action taken by an application that receives this error is to refresh the cursor, get the new values, and then let the user decide whether to perform the modification on the new values. **text**, **ntext**, and **image** columns are not used for concurrency comparisons in SQL Server version 6.5 or earlier.

OPTIMISTIC WITH ROW VERSIONING

This optimistic concurrency control option is based on row versioning. With row versioning, the underlying table must have a version identifier of some type that the server can use to determine whether the row has been changed after it was read into the cursor. In SQL Server that capability is provided by the **timestamp** data type, which is a binary number that indicates the relative sequence of modifications in a database. Each database has a global current timestamp value, @@DBTS. Each time a row with a **timestamp** column is modified in any way, SQL Server stores the current @@DBTS value in the timestamp column and then increments @@DBTS. If a table has a **timestamp** column, then the timestamps are taken down to the row level. The server can then compare the current timestamp value of a row with the timestamp value that was stored when the row was last fetched to determine whether the row has been updated. The server does not have to compare the values in all columns, only the **timestamp** column. If an application requests optimistic concurrency with row versioning on a table that does not have a timestamp column, the cursor defaults to values-based optimistic concurrency control.

SCROLL LOCKS

This option implements pessimistic concurrency control, in which the application attempts to lock the underlying database rows at the time they are read into the cursor result set. When using server cursors, an update lock is placed on the row when it is read into the cursor. If the cursor is opened within a transaction, the transaction update lock is held until the transaction is either committed or rolled back; the cursor lock is dropped when the next row is fetched. If the cursor has been opened outside a transaction, the lock is dropped when the next row is fetched. Therefore, a cursor should be opened in a transaction whenever the user wants full pessimistic concurrency control. An update lock prevents any other task from acquiring an update or exclusive lock, which prevents any other task from updating the row. An update lock, however, does not block a shared lock, so it does not prevent other tasks from reading the row unless the second task is also requesting a read with an update lock.

Scroll Locks

These cursor concurrency options may generate scroll locks, depending on the locking hints specified in the SELECT statement in the cursor definition. Scroll locks are acquired on each row in a fetch and held until the next fetch or the close of the cursor, whichever occurs first. On the next fetch, the server acquires scroll locks for the rows in the new fetch and then releases the scroll locks for the rows in the previous fetch. Scroll locks are independent of transaction locks and may persist past a commit or rollback operation. If the option to close cursors on commit is off, a COMMIT does not close any open cursors and scroll locks are preserved past the commit to maintain the isolation of the fetched data.

The type of scroll locks acquired depends on the cursor concurrency option and the locking hints in the cursor SELECT statement.

		Optimistic	Optimistic with	
Locking hints	Read only	with values	row versioning	Locking
No Hints	No locking	No locking	No locking	Update
NOLOCK*	No locking	No locking	No locking	No locking
HOLDLOCK	Share	Share	Share	Update
UPDLOCK	Error	Update	Update	Update
TABLOCKX	Error	No locking	No locking	Update
All Others	No locking	No locking	No locking	Update

*Specifying the NOLOCK hint makes the table on which it is specified read-only through the cursor.

Specifying Cursor Concurrency Options

The concurrency options are specified differently in each cursor environment:

• Transact-SQL cursors

Specify the READ_ONLY, SCROLL_LOCK, and OPTIMISTIC keywords on the DECLARE CURSOR statement. The OPTIMISTIC keyword specifies optimistic with row versioning, Transact-SQL cursors do not support the optimistic with values concurrency option.

• ADO applications

Specify adLockReadOnly, adLockPessimistic, adLockOptimistic, or

adLockBatchOptimistic in the **LockType** property of a **Recordset** object.

• ODBC applications

Set the statement attribute SQL_ATTR_CONCURRENCY to SQL_CONCUR_READ_ONLY, SQL_CONCUR_ROWVER, SQL_CONCUR_VALUES, or SQL_CONCUR_LOCK.

• DB-Library applications

Set the **dbcursoropen** parameter *concuropt* to CUR_READONLY, CUR_OPTCC (for optimistic using row versioning), CUR_OPTCCVAL, or CUR_LOCKCC.

See Also

DECLARE CURSOR

Using Server Cursors with ADO

<u>dbcursoropen</u>

Cursor Transaction Isolation Levels

The transaction locking behavior of a specific cursor is determined by combining the locking behaviors of the cursor concurrency setting, any locking hints specified in the cursor SELECT, and transaction isolation level options.

Microsoft® SQL Server[™] 2000 supports these cursor transaction isolation levels:

Read Committed

SQL Server acquires a share lock while reading a row into a cursor but frees the lock immediately after reading the row. Because shared lock requests are blocked by an exclusive lock, a cursor is prevented from reading a row that another task has updated but not yet committed. Read committed is the default isolation level setting for both SQL Server and ODBC.

Read Uncommitted

SQL Server requests no locks while reading a row into a cursor and honors no exclusive locks. Cursors can be populated with values that have already been updated but not yet committed. The user is bypassing all of the locking transaction control mechanisms in SQL Server.

Repeatable Read or Serializable

SQL Server requests a shared lock on each row as it is read into the cursor as in READ COMMITTED, but if the cursor is opened within a transaction, the shared locks are held until the end of the transaction instead of being freed after the row is read. This has the same effect as specifying HOLDLOCK on a SELECT statement.

See Also

Customizing Transaction Isolation Level

Adjusting Transaction Isolation Levels

Cursor Concurrency

Changing Rows with Positioned Operations

Updatable cursors support data modification statements that update rows through the cursor. When positioned on a row in an updatable cursor, you can perform update or delete operations that target the base table rows used to build the current row in the cursor. These are called position updates.

The positioned updates are performed on the same connection that opened the cursor. This allows the data modifications to share the same transaction space as the cursor, and prevents the updates from being blocked by locks held by the cursor.

There are two methods for performing positioned updates in a cursor:

- The Transact-SQL WHERE CURRENT OF clause on an UPDATE or DELETE statement.
- A database API positioned update function or method, such as the ODBC **SQLSetPos** function.

Performing Positioned Updates with Transact-SQL

The Transact-SQL WHERE CURRENT OF clause is typically used in Transact-SQL stored procedures, triggers, and scripts when modifications need to be made based on specific rows in a cursor. The stored procedure, trigger, or script will:

- DECLARE and OPEN a cursor.
- Use FETCH statements to get positioned on a row in the cursor.
- Execute an UPDATE or DELETE statement using a WHERE CURRENT OF clause. Use the *cursor_name* from the DECLARE statement as the *cursor_name* in the WHERE CURRENT OF clause.

Performing Positioned Updates with APIs

Cursors created through the OLE DB, ADO, and DB-Library API functions and methods are not used in WHERE CURRENT OF clauses because they do not have names. ODBC, however, supports getting a name for an API server cursor with the **SQLGetCursorName** function. After setting the cursor attributes and opening a cursor by executing a Transact-SQL statement, use the **SQLGetCursorName** function to get a name for the cursor. After positioning in the cursor, execute an UPDATE or DELETE statement with a WHERE CURRENT OF clause referencing the name returned by **SQLGetCursorName**. But this method is not recommended. Instead, it is better to use the positioned update functions in the ODBC API.

The database APIs support two different methods for performing positioned operations on API server cursors. ODBC and DB-Library share one model, OLE DB and ADO the other.

In ODBC and DB-Library, bind the columns in the cursor to program variables, then position on a specific row in a cursor. If performing a positioned update, change the data values in the program variables to the new values. Call these functions to perform the positioned operation:

- ODBC: The **SQLSetPos** function
- DB-Library: The **dbcursor** function

These functions have the following options:

SQLSetPos(SQL_POSITION)

ODBC only, positions the ODBC cursor on a specific row in the current rowset.

SQLSetPos(SQL_REFRESH), dbcursor(CRS_REFRESH)

Refreshes program variables bound to the result set columns with the values from the row the cursor is currently positioned on.

SQLSetPos(SQL_UPDATE), dbcursor(CRS_UPDATE)

Updates the current row in the cursor with the values stored in the program variables bound to the result set columns.

SQLSetPos(SQL_DELETE), dbcursor(CRS_DELETE)

Deletes the current row in the cursor.

dbcursor(CRS_LOCKCC)

DB-Library only, locks the current row.

OLE DB and ADO use a different model to support positioned updates.

In OLE DB, when positioned on a row within the rowset, call the **IRowsetChange::SetData** or **IRowsetChange::DeleteRows** methods to perform positioned updates. If the OLE DB provider supports **IRowsetUpdate::Update**, the changes made with the **IRowsetChange** methods are cached until you call **IRowsetUpdate::Update**. If the OLE DB provider does not support **IRowsetUpdate::Update**, the changes made with the **IRowsetChange** methods are made immediately.

In ADO, when positioned on a row within the recordset, call the **Recordset** object's **Update** or **Delete** methods to perform positioned updates. If the OLE DB provider supports **IRowsetUpdate::Update**, the changes made with the **Recordset** object's **Update** or **Delete** methods are cached until you call the **Recordset** object's **UpdateBatch** method. If the OLE DB provider does not support **IRowsetUpdate::Update**, the changes made with the **Recordset** object's **Update** or **Delete** methods are made with the **Recordset** object's **Update**.

See Also

WHERE

Positioned Updates (ODBC)

<u>dbcursor</u>

Cursor Programming Details

Choosing the correct cursor options is an important part of developing a Microsoft® SQL Server[™] 2000 application.

Using block cursors can reduce the number of network roundtrips between the client and SQL Server, thereby improving performance. SQL Server may implicitly convert a cursor type if you execute a Transact-SQL statement not supported by the cursor type you requested. SQL Server populates the keyset of a large keyset-driven cursor asynchronously, which shortens the time between when the cursor is opened and when you can fetch the first rows.

Choosing a Cursor Type

Choosing a cursor type depends on several variables, including:

- Size of the result set.
- Percentage of the data likely to be needed.
- Performance of the cursor open.
- Need for cursor operations, such as scrolling or positioned updates.
- Level of visibility to data modifications made by other users.

The default settings are fine for a small result set if no updating is done, but a dynamic cursor is preferred for a large result set in which the user is likely to find an answer before retrieving many of the rows.

Rules for Choosing a Cursor Type

Some simple rules to follow in choosing a cursor type are:

- Use default settings for singleton selects (returns one row), or other small result sets. It is more efficient to cache a small result set on the client and scroll through the cache instead of asking the server to implement a cursor.
- Use the default settings when fetching an entire result set to the client, such as when producing a report. Default result sets are the fastest way to transmit data to the client.
- Default result sets cannot be used if the application is using positioned updates.

- Default result sets cannot be used if the application is using multiple active statements. If cursors are being used only to support multiple active statements, choose fast forward-only cursors.
- Default result sets must be used for any Transact-SQL statement or batch of Transact-SQL statements that will generate multiple result sets.
- Dynamic cursors open faster than static or keyset-driven cursors. Internal temporary work tables must be built when static and keysetdriven cursors are opened, but they are not required for dynamic cursors.
- In joins, keyset-driven and static cursors can be faster than dynamic cursors.
- Keyset-driven or static cursors must be used if you want to do absolute fetches.
- Static and keyset-driven cursors increase the usage of **tempdb**. Static server cursors build the entire cursor in **tempdb**; keyset-driven cursors build the keyset in **tempdb**.
- If a cursor must remain open through a rollback operation, use a synchronous static cursor and set CURSOR_CLOSE_ON_COMMIT to OFF.

Each call to an API fetch function or method causes a roundtrip to the server when using server cursors. Applications should minimize these roundtrips by using block cursors with a reasonably large number of rows returned on each fetch.

See Also

Cursor Types

Block Cursors

API server cursors are not limited to fetching one row at a time; they can retrieve multiple rows in each fetch. When working with a client/server database, such as Microsoft® SQL ServerTM, it is more efficient to fetch several rows at a time. The number of rows returned on a fetch is called the rowset size. Cursors that have a rowset size greater than one are called block cursors. In the supported APIs, you can use the block cursors in these ways:

- ODBC
 - a. Set the statement attribute SQL_ATTR_ROWSET_SIZE to the size of the rowset.
 - b. Use column-wise or row-wise binding to bind the columns to arrays of variables to hold the data from the rows returned. The number of elements in each array is equal to the rowset size.
 - c. Each call to **SQLFetchScroll** fetches the number of rows set with SQL_ATTR_ROWSET_SIZE.
- OLE DB
 - a. Allocate an array of row handles and bind the columns represented by each handle to an array of variables to hold the column data.
 - b. Call **IRowset::GetNextRows** with the *cRows* parameter set to the number of handles in the row handle array and the address of the array of row handles in the *prghRows* parameter.
- ADO
- a. Dim a variant to hold the data for the number of rows you want to retrieve.

- b. Optionally, position at the proper point in the cursor using the recordset methods **Move**, **MoveFirst**, **MoveLast**, **MoveNext**, and **MovePrevious**.
- c. Call the recordset method **GetRows** with the *array* parameter holding the address of the variant to hold the rows and the *Crows* parameter holding the number of rows to return. Optionally, have the *Start* parameter indicate the fetch position in the recordset.
- DB-Library
 - a. Allocate arrays to hold the data for each column, and bind each array to its target column using **dbcursorbind**.
 - b. Call **dbcursorfetchex** with the *rownum* parameter set to the number of rows to fetch.

See Also

Rowsets and SQL Server Cursors

Cursor Rowset Size

<u>dbcursorfetchex</u>

dbcursorbind

Implicit Cursor Conversions

Applications can request a cursor type and then execute a Transact-SQL statement that is not supported by server cursors of the type requested. Microsoft® SQL Server[™] returns an error that indicates the cursor type has changed.

These are the factors that trigger SQL Server to implicitly convert a cursor from one type to another.

	Conversion	Forward-	Fast-	Keyset-		Go to
Step	triggered by	only	forward	driven	Dynamic	step
1	Cursor references a view with a TOP clause.	Becomes static.	Becomes static.	Becomes static.	Becomes static.	Done
2	Query FROM clause references no tables.	Becomes static.		Becomes static.	Becomes static.	Done
3	Query contains: select list aggregates GROUP BY UNION DISTINCT HAVING	Becomes static.		Becomes static.	Becomes static.	Done
4	Query references an inserted or deleted table within a trigger.			Becomes static.	Becomes static.	Done
5	Query joins a trigger table to another table.		Becomes static.			Done
6	READ_ONLY is not specified.		Becomes static.			8
7	ODBC API server		Becomes			8

	cursor references		dynamic.			
	text, ntext , or					
	image columns.					
8	Query generates an internal work table, for example the columns of an ORDER BY are not covered by an index.	keyset.			Becomes keyset.	10
9	Query references remote tables in linked servers.	Becomes keyset.	Becomes keyset.		Becomes keyset.	10
10	Query references at least one table without a unique index.			Becomes static.		Done
11	Cursor references		Becomes			Done
	text , ntext , or		keyset.			
	image columns;					
	and the query					
	contains a TOP					
	clause.					

SQL Server version 6.5 has this restriction in addition to the restrictions in SQL Server 7.0:

• If a dynamic cursor is requested and the Transact-SQL statement contains an ORDER BY that does not match an index or subquery, the cursor is converted to a keyset-driven or static cursor. If all the tables have a unique index, but no index that covers the ORDER BY, the cursor is converted to a keyset-driven cursor. If at least one table has no index that covers the ORDER BY and at least one has no unique index (not necessarily the same table), the cursor is converted to static.

An index column cannot be used to cover the ORDER BY if there are

index columns to its left that are not referenced by the ORDER BY. For example, if an index is defined as using **MyTable** (**LastName**, **FirstName**), the index cannot be used to cover a statement using ORDER BY **FirstName**.

The SQL Server version 6.0 restriction regarding dynamic cursors is more simply defined:

• If a dynamic cursor is requested and there is at least one table that does not have a unique index, the cursor is converted to a static cursor.

If you are using API server cursors and get a message indicating the cursor type has been changed, you can call the following functions to see the type of cursor SQL Server opened:

- ODBC: Call **SQLGetInfo** for the SQL_CURSOR_TYPE attribute.
- DB-Library: Call **dbcursorinfoex** and refer to the Type field in the DBCURSORINFO structure returned by **dbcursorinfoex**.
- Transact-SQL: Use **sp_describe_cursor** and refer to the **model** and **scrollable** columns in the cursor returned by the procedure.

See Also

Implicit Cursor Conversions (ODBC)

<u>dbcursorinfoex</u>

Asynchronous Population

Microsoft® SQL Server[™] 2000 offers a performance optimization of populating large keyset-driven or static cursors asynchronously. Keyset-driven and static cursors use work tables built in **tempdb**. Keyset-driven cursors use the work table to store their keyset, the set of keys that identify the rows in the cursor. Static cursors use work table to store the rows comprising the cursor. If the SQL Server query optimizer estimates that the number of rows returned in a keyset-driven or static cursor will exceed the value of the **sp_configure cursor threshold** parameter, the server starts a separate thread to populate the work table. Control is immediately returned to the application, which can start fetching the first rows in the cursor instead of having to wait until the entire work table has been populated before performing the first fetch.

There is some extra overhead associated with populating a cursor asynchronously. It is more efficient not to populate small cursors asynchronously, so the **sp_configure cursor threshold** value should not be set too low. Reserve the use of asynchronous population for large cursors.

The @@CURSOR_ROWS function reports the number of rows in a cursor. If you select @@CURSOR_ROWS on a cursor with a work table that is still being populated, @@CURSOR_ROWS returns a negative number. The absolute value of the number returned is the number of the rows that have been populated in the work table up to that time. For example, if @@CURSOR_ROWS is selected while the keyset of a keyset-driven cursor is still being populated, but 1,243 keys are already in the keyset, @@CURSOR_ROWS returns a value of -1243.

See Also

@@CURSOR_ROWS

<u>sp_configure</u>

Scope of Transact-SQL Cursor Names

Microsoft® SQL Server[™] 2000 supports the GLOBAL and LOCAL keywords on the DECLARE CURSOR statement to define the scope of the cursor name. GLOBAL specifies that the cursor name is global to the connection. LOCAL specifies that the cursor name is LOCAL to the stored procedure, trigger, or batch containing the DECLARE CURSOR statement.

Prior to Microsoft SQL Server version 7.0, the names of Transact-SQL cursors were global to the connection. You could execute one stored procedure that creates a cursor, and then call another stored procedure that fetches the rows from that cursor:

USE pubs GO CREATE PROCEDURE OpenCrsr AS

DECLARE SampleCrsr CURSOR FOR SELECT au_lname FROM authors WHERE au_lname LIKE 'S%'

OPEN SampleCrsr GO

CREATE PROCEDURE ReadCrsr AS FETCH NEXT FROM SampleCrsr WHILE (@@FETCH_STATUS <> -1) BEGIN FETCH NEXT FROM SampleCrsr END GO

EXEC OpenCrsr /* DECLARES and OPENS SampleCrsr. */

```
GO
EXEC ReadCrsr /* Fetches the rows from SampleCrsr. */
GO
CLOSE SampleCrsr
GO
DEALLOCATE SampleCrsr
GO
```

Local cursors offer important protection for cursors implemented in stored procedures and triggers. Global cursors can be referenced outside the stored procedure or trigger in which they are declared. Consequently, they can be inadvertently changed by statements outside the stored procedure or trigger. Local cursors are more secure than global cursors because they cannot be referenced outside a stored procedure, unless deliberately passed back to the caller as a cursor output parameter.

Because global cursors can be referenced outside a stored procedure or trigger, they can have unintended side effects that influence other statements. An example is a stored procedure that creates a global cursor with a name of **xyz** and leaves the cursor open when it completes. An attempt to declare another global cursor with the name **xyz** after the stored procedure completed fails with a duplicate name error.

Global and local cursors have separate name spaces, so it is possible to have both a global cursor and a local cursor with the same name at the same time. The Transact-SQL statements that accept a cursor name parameter also support the GLOBAL keyword to identify the scope of the name. If GLOBAL is not specified, and there are both a local and global cursor with the name specified in the cursor name parameter, the local cursor is referenced.

The database option **default to local cursor** controls the default taken by the DECLARE CURSOR statement if neither LOCAL nor GLOBAL is specified. If **default to local cursor** is **true**, Transact-SQL cursors default to local. If the option is **false**, Transact-SQL cursors default to global. In SQL Server 2000, the **default to local cursors** option itself defaults to FALSE to match the behavior of earlier versions of SQL Server.

Stored procedures that DECLARE and OPEN local cursors can pass the cursors

out for use by the calling stored procedure, trigger, or batch. This is done using an OUTPUT parameter defined with the new CURSOR VARYING data type. Cursor variables can only be used as OUTPUT parameters. They cannot be used for input parameters. The cursor must be open when the stored procedure completes to be passed back in an OUTPUT parameter. Local variables can also be declared with the new CURSOR data type to hold a reference to a local cursor.

USE pubs GO /* Create a procedure with a cursor output parameter. */ CREATE PROCEDURE OpenCrsr @OutCrsr CURSOR VARYING O

SET @OutCrsr = CURSOR FOR SELECT au_lname FROM authors WHERE au_lname LIKE 'S%'

OPEN @OutCrsr GO

/* Allocate a cursor variable. */ DECLARE @CrsrVar CURSOR

/* Execute the procedure created earlier to fill
 the variable. */
EXEC OpenCrsr @OutCrsr = @CrsrVar OUTPUT

/* Use the variable to fetch the rows from the cursor. */ FETCH NEXT FROM @CrsrVar WHILE (@@FETCH_STATUS <> -1) BEGIN FETCH NEXT FROM @CrsrVar END CLOSE @CrsrVar

DEALLOCATE @CrsrVar GO

The database APIs do not support cursor output parameters on stored procedures. A stored procedure that contains a cursor output parameter cannot be executed directly from a database API function. These stored procedures can only be executed from another stored procedure, a trigger, or a Transact-SQL batch or script.

A GLOBAL cursor is available until it is explicitly deallocated or the connection is closed. LOCAL cursors are implicitly deallocated when the stored procedure, trigger, or batch in which they were created terminates, unless the cursor has been passed back as a parameter. The LOCAL cursor will then be implicitly deallocated when the parameter or variable referencing the cursor in the code that called the procedure goes out scope.

See Also

DECLARE CURSOR

sp_dboption

Getting Server Cursor Meta Data

There are two ways to get meta data describing a server cursor:

- Applications using API server cursors with a database API such as ADO, OLE DB, ODBC, or DB-Library typically use the cursor functionality of the API to get information about the state of the cursor.
- Transact-SQL scripts, stored procedures, and triggers can use the Transact-SQL functions and system stored procedures discussed in this topic to get information about a Transact-SQL cursor.

There are several system functions that report status information for a server cursor, or a server cursor assigned to a cursor variable:

• CURSOR_STATUS

Indicates whether a cursor is open or closed, or if a cursor variable is currently associated with a cursor.

• @@FETCH_STATUS

Indicates the success or failure of the last fetch operation performed for the connection.

• @@CURSOR_ROWS

Reports the number of rows populated in the last cursor opened for the connection.

There are several system stored procedures that report the characteristics of a server cursor, or a server cursor assigned to a cursor variable:

• sp_describe_cursor

Returns a cursor describing the attributes of a cursor, such as its scope, name, type, status, and the number of rows.

• sp_describe_cursor_columns

Returns a cursor describing the attributes of each column in the cursor,

such as the column's name, position, size, and data type.

• sp_describe_cursor_tables

Returns a cursor describing the base tables referenced by the cursor.

• sp_cursor_list

Returns a cursor listing all the currently visible cursors for the connection. The format of the cursor returned by **sp_cursor_list** is the same as the cursor from **sp_describe_cursor**.

These system stored procedures return their result sets as output cursor variables. The database APIs do not support cursor variables, so these procedures cannot be called from applications, only within Transact-SQL scripts, stored procedures, and batches. Applications should use the cursor functionality of the database APIs to get the metadata for API server cursors.

Be careful with the status information returned by these functions and stored procedures, especially @@FETCH_STATUS. The information returned by @@FETCH_STATUS changes every time a FETCH statement is issued against any cursor open for the connection. A stored procedure or trigger that may need to refer to the status information after executing several additional statements should save @@FETCH_STATUS in an integer variable immediately after the FETCH statement. @@FETCH_STATUS may be reset even if there are no FETCH statements in the batch between the FETCH and the statement that tests the status. If an intervening INSERT, UPDATE or DELETE statement fires a trigger, the trigger can open and fetch from a cursor. @@FETCH_STATUS would then contain the status of the last FETCH statement in the trigger.

The stored procedures report their status information for a specific cursor, so their status information is not affected by operations on other cursors. Their status information is still affected by operations on the same cursor, so care must still be taken in using the status information returned by the stored procedures.

See Also

@@CURSOR_ROWS @@FETCH_STATUS

CURSOR_STATUS

<u>sp_cursor_list</u>

sp_describe_cursor

sp_describe_cursor_columns

sp_describe_cursor_tables

Using Cursors with Distributed Queries

When using cursors with distributed queries, both insensitive (static) and keysetdriven cursor types are supported when the provider supports the necessary OLE DB functionality. Dynamic or forward-only cursors requested with a distributed query are implicitly converted to keyset-driven cursors.

A keyset-driven cursor is supported on a distributed query if all local and remote tables in the query have a unique key. For the conditions under which a keyset cursor is supported on a distributed query, <u>Keyset-driven Cursors Requirements</u> <u>for OLE DB Providers</u>.

If a keyset-driven cursor is allowed on a distributed query, then it can also be used to update any of the underlying remote tables if the provider requirements for updatability are met. For the requirements for updates on a remote table, see <u>UPDATE and DELETE Requirements for OLE DB Providers</u>.

See Also

DECLARE CURSOR Using Cursors Using Server Cursors with ADO

Distributed Queries

Distributed queries access data from multiple heterogeneous data sources, which can be stored on either the same or different computers. Microsoft® SQL Server[™] 2000 supports distributed queries by using OLE DB, the Microsoft specification of an application programming interface (API) for universal data access.

This Distributed Queries section discusses general distributed query concepts and describes how to use Transact-SQL statements in distributed queries to access data on separate instances of SQL Server, or non-SQL Server data sources. Other sections of SQL Server Books Online contain additional related information.

For this information	See			
Configuring OLE DB providers and	Configuring Linked Servers			
linked servers	Configuring OLE DB Providers for			
	Distributed Queries			
Distributed queries in distributed	Distributed Queries and Distributed			
transactions	<u>Transactions</u>			
Linked servers	Accessing External Data			
ODBC and distributed queries	Schema Functions Supporting			
	Distributed Queries			
Information about specific OLE DB	OLE DB Providers Tested with SQL			
Providers	<u>Server</u>			
Performance issues	Optimizing Distributed Queries			
Reference for OLE DB Provider	OLE DB Provider Reference for			
developers	Distributed Queries			
Transact-SQL language in distributed	External Data and Transact-SQL			
queries	System Stored Procedures			
Troubleshooting distributed queries	Distributed Queries Error Messages			

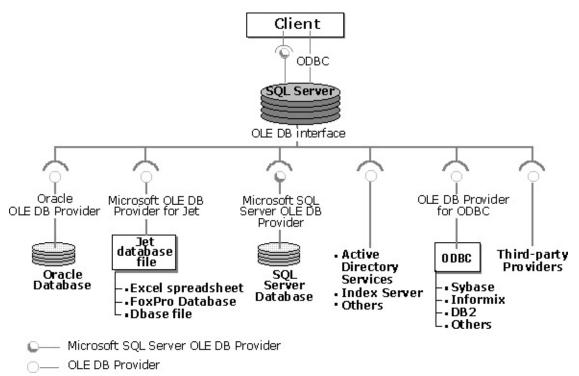
Distributed queries provide SQL Server users with access to:

- Distributed data stored in multiple instances of SQL Server.
- Heterogeneous data stored in various relational and non-relational data sources accessed using an OLE DB provider.

OLE DB providers expose data in tabular objects called rowsets. SQL Server 2000 allows rowsets from OLE DB providers to be referenced in Transact-SQL statements as if they were a SQL Server table.

Tables and views in external data sources can be referenced directly in SELECT, INSERT, UPDATE, and DELETE Transact-SQL statements. Because distributed queries use OLE DB as the underlying interface, distributed queries can access traditional relational DBMS systems with SQL query processors, as well as data managed by data sources of varying capabilities and sophistication. As long as the software owning the data exposes it in a tabular rowset through an OLE DB provider, the data can be used in distributed queries.

Note Using distributed queries in SQL Server is similar to the linked table functionality through ODBC, which was supported previously by Microsoft Access. This functionality is now built into SQL Server with OLE DB as the interface to external data.



Example

See Also

Configuring Linked Servers

Distributed Queries and Distributed Transactions

Security Account Delegation

Accessing External Data

To access data from an OLE DB data source, provide Microsoft® SQL Server[™] 2000 with the following information:

- The name of the OLE DB provider that exposes the data source.
- Any information the OLE DB provider needs to locate the source of the data.
- Either the name of an object that the OLE DB data source can expose as a rowset, or a query that can be sent to the OLE DB provider that will cause it to expose a rowset. The objects that can be exposed as rowsets are known as remote tables. The queries that generate rowsets are known as pass-through queries.
- Optionally, you can supply SQL Server with valid login IDs for the OLE DB data source.

SQL Server 2000 supports these methods for referencing heterogeneous OLE DB data sources in Transact-SQL statements: the linked server name and the ad hoc computer name.

Linked Server Names

A linked server is a virtual server that has been defined to Microsoft® SQL Server[™] 2000 with all the information needed to access an OLE DB data source. A linked server name is defined using the **sp_addlinkedserver** system stored procedure. The linked server definition contains all the information needed to locate the OLE DB data source. Local SQL Server logins are then mapped to logins in the linked server using **sp_addlinkedsrvlogin**. Remote tables can then be referenced by using the linked server name:

• As the server name in a four-part name used as a table or view reference in a Transact-SQL statement. The other three parts of the name

reference an object in the linked server that is exposed as a rowset.

• As an input parameter to an OPENQUERY function. OPENQUERY sends the OLE DB provider a command to execute. The returned rowset can then be used as a table or view reference in a Transact-SQL statement.

Ad Hoc Names

An ad hoc name is used for infrequent queries against OLE DB data sources that are not defined as a linked server name. In SQL Server 2000, the OPENROWSET and OPENDATASOURCE functions provide connection information for accessing data from OLE DB data sources.

OPENROWSET and OPENDATASOURCE should be used only to reference OLE DB data sources that are accessed infrequently. For any data sources that will be accessed more than a few times, define a linked server. Neither OPENDATASOURCE nor OPENROWSET provide all of the functionality of linked server definitions, including security management and the ability to query catalog information. Each time these functions are called, all connection information, including passwords, must be provided.

OPENROWSET and OPENDATASOURCE appear to be functions; however, they are macros and do not support supplying Transact-SQL variables as arguments.

The OPENROWSET function can be used with any OLE DB provider that returns a rowset, and can be used anywhere a table or view reference is used in a Transact-SQL statement. OPENROWSET is specified with:

- All the information needed to connect to the OLE DB data source.
- Either the name of an object that will generate a rowset, or a query that will generate a rowset.

The OPENDATASOURCE function provides connection information as part of a four-part object name. This function supports only OLE DB providers that expose multiple rowsets using the catalog.schema.object notation.

OPENDATASOURCE can be used in the same Transact-SQL syntax locations a linked server name can be used. OPENDATASOURCE is specified with:

- The name registered as the PROGID of the OLE DB provider used to access the data source.
- A connection string that specifies the various connection properties to be passed to the OLE DB provider. The connection string syntax is a sequence of keyword-value pairs. The basic syntax is defined in the Microsoft® Data Access Software Development Kit, and each provider documents the specific keyword-value pairs it supports. For more information about connection strings, see <u>OPENDATASOURCE</u>.

Accessing Linked Servers

After a linked server is created using **sp_addlinkedserver**, it can be accessed using:

- Distributed queries. Accessing tables in the linked server through SELECT, INSERT, UPDATE, and DELETE statements using a linked server-based name.
- Remote stored procedures. Stored procedures can be executed against the linked server using a four-part name.

Servers running an instance of SQL Server can be defined as a remote server using **sp_addserver**. The remote server then can be referenced in remote stored procedure calls. The remote server component is maintained as a compatibility feature for existing applications.

As applications are ported to SQL Server 2000, they may need to run for a period of time with some new code using distributed queries against a linked server definition and some legacy code using a remote server definition. Both linked servers and remote servers use the same name space, so either the linked server or the remote server definition has to use a name that is different than the network name of the server being accessed remotely. Define one of the entries with a different server name, and use **sp_setnetname** to associate that definition

with the network name of the remote server.

Note The examples in this section use system stored procedures to configure linked servers because these system stored procedures succinctly show the parameters used. However, SQL Server Enterprise Manager also supports configuring linked servers. For more information, see <u>Configuring Linked Servers</u>.

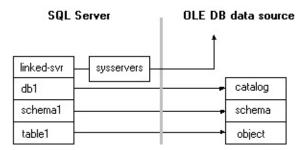
Identifying a Data Source Using a Linked Server Name

After a linked server is defined, a four-part name in the form *linked_server_name.catalog.schema.object_name* can be used in Transact-SQL statements to reference data objects in that linked server. The table describes the parts of a four-part name.

Part name	Description		
linked_server_name	Linked server referencing the OLE DB data source		
J	Catalog in the OLE DB data source that contains the object		
schema	Schema in the catalog that contains the object		
object_name	Data object in the schema		

Microsoft® SQL Server[™] uses the linked server name to identify the OLE DB provider and the data source. The *catalog*, *schema*, and *object_name* parameters are passed to the OLE DB provider to identify a specific data object. When the linked server refers to an instance of SQL Server, *catalog* refers to a database and *schema* refers to an owner ID.

This illustration shows how a four-part SQL Server name resolves to an object in the OLE DB provider.



Always use fully qualified names when working with objects on linked servers. There is no support for implicit resolution to the **dbo** owner name for tables in linked servers. Because of this, a query without a schema name generates a 7314 error even when the linked server is another instance of SQL Server. SQL Server does not support full-text search over linked servers.

See Also

OPENQUERY

Identifying a Data Source Using an Ad Hoc Name

An ad hoc name can be used as a table reference when the OLE DB data source will not be referenced often enough to warrant configuring a linked server. In Microsoft® SQL ServerTM 2000, you can use the OPENROWSET and OPENDATASOURCE functions to provide an ad hoc name.

Both the OPENROWSET and OPENDATASOURCE functions provide ad hoc connection information. You can use these functions to specify all the information needed to access the OLE DB data source. However, you cannot use OPENROWSET and OPENDATASOURCE interchangeably.

You can use the OPENROWSET function wherever the OLE DB provider returns rowsets either by specifying a table (or view) name or by specifying a query that returns a rowset. The OPENROWSET function can be used in the place of a table or view name in a Transact-SQL statement.

--This example uses an ad hoc name to retrieve data from the Custome SELECT *

FROM OPENROWSET('Microsoft.Jet.OLEDB.4.0',

'c:\MSOffice\Access\Samples\northwind.mdb';'admin';", Customers)

Use OPENDATASOURCE only when the provider exposes rowsets *and* uses the catalog.schema.object notation. This function can be used in the same Transact-SQL syntax locations a linked server name can be used. Thus, in the catalog.schema.object notation, OPENDATASOURCE can be used as the first part of a four-part name that refers to a table or a view name.

-- SELECT from a table on another instance of SQL Server. SELECT * FROM OPENDATASOURCE('SQLOLEDB', 'Data Source=ServerName;User ID=MyUID;Password=MyPass').Northwind.dbo.Categories Both OPENROWSET and OPENDATASOURCE should be used only for accessing external data for ad hoc situations, when it is not possible to configure a permanent linked server. These functions do not provide all of the functionality available from a linked server, such as management of login mappings, ability to query the linked server's meta data, and the ability to configure various connection settings such as time-out values.

The arguments of OPENROWSET and OPENDATASOURCE do not support variables. They have to be specified as string-literal. If variables need to be passed in as arguments to these functions, a query string containing these variables can be constructed dynamically and executed using the EXEC statement.

See Also

OPENDATASOURCE

OPENROWSET

Using Pass-Through Queries as Tables

Microsoft® SQL Server[™] 2000 sends pass-through queries as uninterpreted query strings to an OLE DB data source. The query must be in a syntax the OLE DB data source will accept. A Transact-SQL statement uses the results from a pass-through query as though it is a regular table reference.

This example uses a pass-through query to retrieve a result set from a Microsoft Access version of the **Northwind** sample database.

```
SELECT *
FROM OpenRowset('Microsoft.Jet.OLEDB.4.0',
'c:\northwind.mdb';'admin'; ",
'SELECT CustomerID, CompanyName
FROM Customers
WHERE Region = "WA" ')
```

The ways to generate a rowset from an OLE DB provider are:

- Reference an object in the data source that the provider can expose as a tabular rowset. All providers support this capability.
- Send the provider a command that the provider can process and expose the results of the command as a rowset. This capability requires that the provider support the OLE DB **Command** object and all of its mandatory interfaces.

When a provider supports the **Command** object, these Transact-SQL functions can be used to send it commands (called pass-through queries):

- OPENQUERY sends a command string to an OLE DB data source using a linked server name.
- OPENROWSET and OPENDATASOURCE support sending a command string to an OLE DB data source. The resulting rowset can be referenced using an ad hoc name.

The OLE DB specification does not define a single command language to be used by all OLE DB providers. OLE DB providers are allowed to support any command language that is related to the data they expose. OLE DB providers that expose the data in relational databases generally support the SQL language. Other types of providers, such as those exposing the data in an e-mail file or network directory, generally support a different language.

See Also

FROM OPENDATASOURCE OPENQUERY OPENROWSET

External Data and Transact-SQL

As long as the provider supports the required OLE DB interfaces, each class of Transact-SQL statements mentioned later is allowed. Here is the subset of the Transact-SQL language allowed on remote tables accessed through linked server-based names or ad hoc names:

- All queries with the standard form of SELECT *select_list* FROM clause WHERE clause are allowed. The INTO *new_table_name* clause of SELECT is not allowed when the *new_table_name* refers to a remote table.
- In SELECT, INSERT, UPDATE, and DELETE statements, columns in remote tables cannot be qualified with a single-part or four-part table name. The remote tables should be aliased in the FROM clause and the alias name should be used to qualify the column name.
- When specifying a large object (LOB) column from a remote table as an item in the *select_list* of a SELECT statement, the SELECT statement cannot contain an ORDER BY clause.
- The IS NULL and IS NOT NULL predicates cannot reference LOB columns in a remote table.
- GROUP BY ALL is not allowed in a distributed query when the query also has a WHERE clause. GROUP BY without specifying ALL is supported.
- INSERT statements are allowed against remote tables as long as the provider meets the OLE DB requirements for INSERT statements. For more information, see <u>INSERT Requirements for OLE DB Providers</u>.
- UPDATE and DELETE statements are allowed against remote tables if

the provider meets the OLE DB interface requirements on the specified table. For more information, see <u>UPDATE and DELETE Requirements</u> for OLE DB Providers.

- A remote table can be updated or deleted through a cursor defined on a distributed query when the remote table is specified in the UPDATE or DELETE statement (UPDATE or DELETE *remote_table* WHERE CURRENT OF *cursor_name*) if the provider meets the conditions for updatability on the remote table. For more information, see <u>Using Cursors with Distributed Queries</u>.
- READTEXT, WRITETEXT, and UPDATETEXT statements are not supported against remote tables.
- Columns with large object data types (such as text, ntext, or image) cannot be referenced in update or insert operations if the provider is instantiated outside the Microsoft® SQL Server[™] 2000 process (provider option AllowInProcess is 0). For more information, see Configuring OLE DB Providers for Distributed Queries.
- <u>Data Definition Language</u> statements (such as CREATE, ALTER, or DROP statements) are not allowed against linked servers.
- No other database-level operations or statements are allowed on linked servers.
- STATIC or INSENSITIVE cursors can reference remote tables. KEYSET cursors can reference remote tables if the OLE DB provider meets certain requirements. For more information about these requirements, see <u>Keyset-driven Cursors Requirements for OLE DB</u> <u>Providers</u>. No other type of cursor can reference a remote table.
- Stored procedures are supported only against SQL Server data sources.

Setting SQL-92 Options for Distributed Queries

A connection must have the ANSI_NULLS and ANSI_WARNINGS options turned on before it can execute distributed queries.

See Also

SET ANSI_DEFAULTS

Using Transactions with Distributed Queries

Microsoft® SQL Server[™] 2000 supports transaction-based access to external data using the **ITransactionLocal** (local transaction) and **ITransactionJoin** (distributed transactions) OLE DB interfaces. Using distributed transactions, SQL Server ensures that a transaction involving multiple nodes is either committed or rolled back in all the nodes. If the provider does not support participating in a distributed transaction (does not support **ITransactionJoin**), only read-only operations are allowed against that provider when inside a transaction.

When a disallowed statement is encountered, the statement returns an error message and the user transaction, if any, terminates execution.

See Also

Transactions

Data Type Mapping

An OLE DB provider exposes the data types of its data in terms of OLE DB type identifiers called DBTYPEs. Data types are converted between OLE DB data types and native Microsoft® SQL ServerTM data types by mapping data:

- From OLE DB data types to SQL Server native data types. This conversion occurs when SQL Server reads data from the OLE DB data source, either in SELECT statements or in the reading side of UPDATE, INSERT, or in DELETE statements.
- From SQL Server native data types to OLE DB data types. This conversion occurs when SQL Server writes data, mostly in INSERT or UPDATE statements, into the OLE DB data source in which the modified table is a remote table.

Data Type Mapping from the OLE DB Provider to SQL Server

Data type mapping from the OLE DB provider to SQL Server defines the allowed comparisons and expressions, and the valid explicit conversions involving remote data. The data type of *remote_column* corresponds to the mapped local data type as shown in the Data Type Mapping table.

The type rules for remote table columns in expressions can be described by this rule: a given remote column value is legal in a Transact-SQL expression if the corresponding mapped SQL Server data type in the following table is legal in the same context. For example, consider the expression:

local_column operator remote_column

local_column is a local table column and *remote_column* is a remote table column. This is a valid expression if *operator* is a valid operator for the local column's data type and for the data type to which the DBTYPE of *remote_column* maps.

Similarly, CAST(*remote_column* AS *data_type_1*) is allowed if the DBTYPE of *remote_column* maps to the SQL Server native *data_type_2* and explicit

conversion from *data_type_2* to *data_type_1* is allowed. For example, a column of data type DBTYPE_DATE on the provider side can be converted to a **datetime** column in SQL Server. However, the DBTYPE_DATE data cannot be converted directly to **varchar**.

The mapping to a SQL Server type is determined by the DBTYPE and the DBCOLUMNFLAGS values describing the column. This information comes from the provider through either the COLUMNS schema rowset or through the **IColumnsInfo** interface. In the case of the COLUMNS schema rowset, the DATA_TYPE and COLUMN_FLAGS columns represent these values. In the case of the **IColumnsInfo::GetColumnInfo** interface, the *wType* and *dwFlags* members of the DBCOLUMNINFO structure represent this information.

The Data Type Mapping table shows data type mappings from the OLE DB provider to SQL Server. For a given column, given its DBTYPE and its DBCOLUMNFLAG value, the corresponding SQL Server data type can be found.

		SQL Sei
DBTYPE	DBCOLUMNFLAGS	data typ
DBTYPE_I1		numeric
DBTYPE_I2		smallint
DBTYPE_I4		Int
DBTYPE_I8		bigint
DBTYPE_UI1		tinyint
DBTYPE_UI2		numeric
DBTYPE_UI4		numeric
DBTYPE_UI8		numeric
DBTYPE_R4		Float
DBTYPE_R8		Real
DBTYPE_NUMERIC		numeric
DBTYPE_DECIMAL		decimal
DBTYPE_CY		money
DBTYPE_BSTR	DBCOLUMNFLAGS_ISLONG = true	ntext
DBTYPE_BSTR	DBCOLUMNFLAGS_ISFIXEDLENGTH	nchar
	= true	

DBTYPE_BSTR	DBCOLUMNFLAGS_ISFIXEDLENGTH = false	nvarcha
DBTYPE_IDISPATCH		Error
 DBTYPE_ERROR		Error
 DBTYPE_BOOL		Bit
 DBTYPE_VARIANT		nvarcha
DBTYPE_IUNKNOWN		Error
DBTYPE_GUID		uniquei
DBTYPE_BYTES	DBCOLUMNFLAGS_ISLONG = true or Maximum column size > 8,000 bytes	image
DBTYPE_BYTES	DBCOLUMNFLAGS_ISROWVER = true, DBCOLUMNFLAGS_ISFIXEDLENGTH = true, Column size = 8	timestar
DBTYPE_BYTES	DBCOLUMNFLAGS_ISFIXEDLENGTH = true	binary
DBTYPE_BYTES	DBCOLUMNFLAGS_ISFIXEDLENGTH = false	varbina
DBTYPE_STR	DBCOLUMNFLAGS_ISFIXEDLENGTH = true	char
DBTYPE_ STR	DBCOLUMNFLAGS_ISFIXEDLENGTH = false	varchar
DBTYPE_STR	DBCOLUMNFLAGS_ISLONG = true or Maximum column size > 4,000 characters	text
DBTYPE_WSTR	DBCOLUMNFLAGS_ISFIXED	nchar
DBTYPE_WSTR	DBCOLUMNFLAGS_ISFIXEDLENGTH = false	nvarcha
DBTYPE_WSTR	DBCOLUMNFLAGS_ISLONG = true or Maximum column size > 4,000 characters	ntext
DBTYPE_UDT		Error
DBTYPE_DATE		datetim
DBTYPE_DBDATE		Error
DBTYPE_DBTIME		Error

DBTYPE_DBTIMESTAMP	datetim
DBTYPE_ARRAY	Error
DBTYPE_BYREF	Ignored
DBTYPE_VECTOR	Error
DBTYPE_RESERVED	Error

¹ **numeric(p,s)** indicates the SQL Server data type **numeric** with precision *p* and scale *s*.

Note If the data must be converted to a native data type different from the shown default, an explicit conversion (using either the CAST or CONVERT function) is required. For more information, see <u>CAST and CONVERT</u>.

Data Type Mapping from SQL Server to the OLE DB Provider

Native SQL Server data types map to OLE DB types using the same table name. A mapping from a SQL Server type *S1* to a given OLE DB type *T* is allowed if either of these conditions exist:

- The corresponding mapping can be found in Table 1.
- There is an allowed implicit conversion of the data type *S1* to another SQL Server data type *S2* and a mapping from *S2* to T is defined in Table 1.

Collations in Distributed Queries

In Microsoft® SQL Server[™] version 7.0, the local SQL Server always had only one server-wide collation (code page and sort order). All character data from remote data sources were interpreted using the local server-wide collation. However, Microsoft SQL Server 2000 supports multiple collations, which can be different for each column; each character value has an associated collation property. SQL Server 2000 interprets the collation property of character data from a remote data source and treats it accordingly.

SQL Server 2000 uses the collation of remote data for comparison and ordering operations on character data (both Unicode and non-Unicode). The collation information for remote character data is determined differently depending on whether or not the data source corresponds to a SQL Server:

- The SQL Server data OLE DB provider automatically reports the collation for each column it returns.
- For remote tables that are not in SQL Server, but for which the collation is known to be the same as one of the collations supported by SQL Server, the administrator can specify the default collation of the OLE DB data source as part of the linked server definition. SQL Server can then use the default collation as the collation for all columns returned from that linked server.

After SQL Server determines the collation of a remote character column, it follows the same rules for converting, comparing, and operating on remote table columns as it does for local columns. For more information about the rules SQL Server applies to collations and the collation names supported by SQL Server, see <u>SQL Server Collation Fundamentals</u>.

The linked server options defined by using **sp_serveroption** control if and how SQL Server uses collations from linked servers:

• **UseRemoteCollation** specifies whether the collation of a remote column or of a local server will be used. When TRUE, the collation of remote columns is used for SQL Server data sources, and the collation

specified in **CollationName** is used for data sources other than SQL Server. When FALSE, distributed queries always use the default collation of the local server instance, and **CollationName** and the collation of remote columns are ignored.

• **CollationName** specifies the name of the collation used by the remote data source if **UseRemoteCollation** is TRUE and the data source is not a SQL Server data source. The name must be one of the collations supported by SQL Server. Use this option when accessing an OLE DB data source other than SQL Server, but whose collation matches one of the SQL Server collations. SQL Server data sources report their column collations, and **CollationName** is ignored for linked servers that reference SQL Server data sources.

Note The only way to enable using remote collations is through the linked server options, therefore, queries constructed using ad hoc names such as OPENROWSET and OPENDATASOURCE cannot use collation information of remote character data. In addition, all linked servers in SQL Server 7.0 that are upgraded to SQL Server 2000 are set to **UseRemoteCollation** = False.

The following table summarizes how SQL Server determines the collation used for each column.

	Use Remote Collation = ON		Use Remote Collation = OFF		
	Collation	Collation	Collation	Collation	
Linked server	name	name	name	name	
type	Not set	Set (to <i>CollX</i>)	Not set	Set (to <i>CollX</i>)	
SQL Server	Remote	Remote	Default	Collation of	
	column's	column's	collation of	local SQL	
	actual	actual	local SQL	Server	
	collation	collation	Server	instance	
			instance		
Others	Default	CollX	Default	Default	
	collation of		collation of	collation of	

local SQL	local SQL	local SQL
Server	Server	Server
instance	instance	instance

Obtaining Meta Data from Linked Servers

OLE DB providers expose meta data about their data through the **IDBSchemaRowset** interface, which can be used to retrieve information in the form of OLE DB rowset objects. Microsoft® SQL Server[™] 2000 uses this interface to get meta data about remote tables. This meta data is exposed to the user through these interfaces:

- Driver-specific functions from the SQL Server ODBC driver. For more information, see <u>SQLLinkedServers</u> and <u>SQLLinkedCatalogs</u>.
- Provider-specific rowsets from the SQL Server OLE DB provider. For more information, see <u>Schema Rowsets</u>.
- Several ODBC catalog functions accept two-part names referencing objects on linked servers.
- System stored procedures similar to the existing catalog stored procedures that return meta data on linked servers.

These system stored procedures can be used to retrieve meta data from linked servers.

sp_linkedservers	sp_primarykeys
sp_catalogs	sp_indexes
sp_column_privileges	sp_table_privileges
sp_columns_ex	sp_tables_ex
sp_foreignkeys	

Note SQL Server is the only data source against which stored procedures are supported in distributed queries.

SQL Server uses the **IDBSchemaRowset** interface of the OLE DB provider to implement these meta data extensions. Because this interface is an optional

interface, a provider may not implement this interface. Meta data is not available on linked servers defined against such providers.

See Also

System Stored Procedures

Upgrading the Catalog Stored Procedures

OLE DB Providers Tested with SQL Server

Microsoft® SQL Server[™] 2000 distributed queries have been tested with several OLE DB providers. Some of the tested OLE DB providers are installed with SQL Server 2000:

- Microsoft <u>OLE DB Provider for SQL Server</u>
- Microsoft <u>OLE DB Provider for ODBC</u>
- Microsoft OLE DB Provider for Jet
- Microsoft <u>OLE DB Provider for DTS Packages</u>
- Microsoft <u>OLE DB Provider for Oracle</u>

The other tested providers are available through Microsoft Windows® 2000 (Microsoft <u>OLE DB Provider for Microsoft Directory Services</u> and the Microsoft <u>OLE DB Provider for Microsoft Indexing Service</u>), and the Microsoft Host Integration Server (<u>Microsoft OLE DB Provider for DB2</u>).

This table shows the OLE DB providers that have been tested with SQL Server distributed queries. All of these providers support being referenced in a SELECT statement by specifying a pass-through query in the OPENQUERY and OPENROWSET functions.

Data source	Provider name	Use in four- part names	Use in pass- through		Use in distributed transactions
SQL Server 6.5 or later	Microsoft OLE DB Provider	Yes	Yes	Yes	Yes

	for SQL Server				
ODBC Data Sources	Microsoft OLE DB Provider for ODBC	Yes*	Yes	Yes*	Yes*
Microsoft Access (Jet) databases	Microsoft OLE DB Provider for Jet version 4.00	Yes	Yes	Yes (No if the database was created with Microsoft Jet version 4.0 or earlier)	No
Microsoft Excel spreadsheets	Microsoft OLE DB Provider for Jet version 4.00	Yes	Yes	Yes	No
U	Microsoft OLE DB Provider for DTS Packages	Yes	Yes	No	No
Oracle databases	Microsoft OLE DB Provider for Oracle version 2.6	Yes	Yes	Yes	Yes
Microsoft Windows® 2000 Directory	Microsoft OLE DB Provider for	No	Yes	No	No

	Microsoft Directory Services				
Local file system (through Indexing Services)	Microsoft OLE DB Provider for Microsoft Indexing Service (Requires Microsoft Windows NT® 4.0 Service Pack 4 or later)	No	Yes	No	No
IBM DB2 databases	Microsoft OLE DB Provider for DB2	Yes	Yes	Yes	No

* The capabilities of the Microsoft OLE DB Provider for ODBC depend on the ODBC driver being used. The provider may not support all these capabilities with some ODBC drivers.

Although Microsoft supports only distributed queries that reference the providers tested by Microsoft, distributed queries should work with any OLE DB provider that meets the requirements documented in the OLE DB Provider Reference for Distributed Queries.

If a provider does not support being used in a four-part name, it can be referenced in an OPENQUERY or OPENROWSET function using a passthrough query.

OLE DB Provider for SQL Server

The Microsoft OLE DB Provider for SQL Server provides an OLE DB interface to Microsoft® SQL Server[™] 2000 databases. Using the OLE DB Provider for SQL Server, SQL Server distributed queries can query data in remote instances of SQL Server.

To create a linked server to access a SQL Server database

1. Execute **sp_addlinkedserver** to create the linked server, specifying **SQLOLEDB** as *provider_name*, and the network name of the server running the remote instance of SQL Server as *data_source*.

For example, to create a linked server named **LinkSQLSrvr** that operates against the instance of SQL Server running on the server whose network name is **NetSQLSrvr**, execute:

sp_addlinkedserver N'LinkSQLSrvr', ' ', N'SQLOLEDB', N'N(

2. Map each local SQL Server login that needs access to the linked server to a SQL Server Authentication login on the linked server.

This example maps access for the local login **Joe** to the SQL Server Authentication login **Visitor** on the linked server named **LinkedSQLSrvr**.

sp_addlinkedsrvlogin N'LinkSQLSrvr', false, N'Joe', N'Visitor

When distributed queries are executed against a server running SQL Server version 7.0 or earlier, the catalog stored procedures on the earlier version must be upgraded to ensure the proper operation of the distributed queries. For example, if a server is running an instance of SQL Server 7.0, the catalog stored procedures on the server must be upgraded to SQL Server 2000 before it can be referenced in a distributed query from a server running an instance of SQL Server 2000. For more information, see <u>Upgrading the Catalog Stored</u> Procedures (OLE DB).

When a remote SQL Server table is updated, the local server or client will not receive any result sets or messages resulting from triggers fired for that update.

When using four-part names, always specify the schema name. Not specifying a schema name in a distributed query prevents OLE DB from finding tables. When referencing local tables, SQL Server uses defaults if an owner name is not specified. The following SELECT statement would generate a 7314 error, even if the linked server login mapped to a **dbo** user in the **Northwind** database on the linked server:

```
sp_addlinkedserver @server = N'LinkServer',
@srvproduct = N' ',
@provider = N'SQLOLEDB',
@datasrc = N'ServerNetName',
@catalog = N'Northwind'
GO
SELECT *
FROM LinkServer.Northwind..Shippers
```

This example defines both a linked server and a remote server that both access the same computer whose network name is **othersite**. The linked server definition uses the same name as the network name of the remote server; the remote server definition uses another name.

```
/* Create a linked server definition to othersite. */
EXEC sp_addlinkedserver 'othersite', N'SQL Server'
```

/* Create a remote server definition using a
fictitious name. */

EXEC sp_addserver 'RPCothersite'

/* Set the fictitious nameto the network name faraway. */ EXEC sp_setnetname 'RPCothersite', 'othersite'

These names can be referenced in distributed queries or remote procedure calls.

/* A distributed query referencing othersite. */ SELECT * FROM othersite.Northwind.dbo.Employees

```
/* A remote procedure call to the same server. */
EXEC RPCothersite.master.dbo.sp_who
/* Distributed queries can be used to execute
  stored procedures on the other server. */
EXEC othersite.master.dbo.sp_who
```

There are differences in the login mapping mechanism between stored procedures executed through linked server and stored procedures executed through remote servers.

Transaction Considerations With Linked SQL Servers

The Microsoft OLE DB Provider for SQL Server does not support nested transactions. Therefore, XACT_ABORT should be set to ON for data modification operations inside implicit or explicit transactions and for data modification operations against distributed partitioned views.

Loopback connections to the same instance of SQL Server are not supported when inside an implicit or explicit transaction or distributed partitioned view.

See Also

Distributed Queries and Distributed Transactions

Linked Server Considerations in a Clustered SQL Server

When linked servers are configured in a clustered SQL Server against OLE DB providers not shipped with Microsoft® SQL Server[™] 2000, make sure that the OLE DB providers are installed in all nodes of the cluster. In addition, any properties that define the linked server should be location transparent; they should not contain information that assumes SQL Server is always running on a given node of the cluster.

This example defines a linked server against a server running SQL Server and references one of the remote tables using a four-part name in a SELECT statement.

```
sp_addlinkedserver @server = N'LinkServer',
 @srvproduct = N' ',
 @provider = N'SQLOLEDB',
 @datasrc = N'ServerNetName',
 @catalog = N'Northwind'
GO
SELECT *
FROM LinkServer.Northwind.dbo.Shippers
GO
```

Loopback Linked Servers

Linked servers can be defined to point back (loop back) to the server on which they are defined. Loopback servers are most useful when testing an application that uses distributed queries on a single server network.

For example, executing the **sp_addlinkedserver** stored procedure on a server named **MyServer** defines a loopback linked server:

```
sp_addlinkedserver @server = N'MyLink',
@srvproduct = N' ',
```

```
@provider = N'SQLOLEDB',
@datasrc = N'MyServer',
@catalog = N'Northwind'
GO
```

Transact-SQL statements that use **MyLink** as the server name loop through the SQLOLEDB provider and back to the local server.

Loopback linked servers cannot be used in a distributed transaction. Attempting a distributed query against a loopback linked server from within a distributed transaction causes an error:

Msg: 3910 Level: 16 State: 1 [Microsoft][ODBC SQL Server Driver][SQL Server]Transaction conte

Distributed Queries on Multiple Instances of SQL Server

Specifying an instance of Microsoft® SQL Server[™] 2000 on a server running multiple instances of SQL Server requires no syntax changes to the Transact-SQL elements used in distributed queries. Instances can be specified in distributed queries using one of these methods:

- Specify a server name using the syntax '*server_name/instance_name*' in the **@datasrc** parameter of **sp_addlinkedserver**.
- Specify Server=*server_name*; INSTANCENAME=*instance_name* in a connection string.

If an instance is not specified, the distributed query connects to the default instance of SQL Server 2000 on the specified server.

Examples of specifying a specific instance named Payroll on a server named London are:

-- Define a linked server on an instance of SQL Server using @datasrc sp_addlinkedserver

```
@server = 'LondonPayroll1',
@provider = 'SQLOLEDB',
@datasource = 'London/Payroll'
```

-- Define a linked server on an instance of SQL Server using

-- INSTANCENAME in a provider string.

sp_addlinkedserver

```
@server = 'LondonPayroll2',
```

```
@provider = 'SQLOLEDB',
```

@provstr = 'Server=London;INSTANCENAME=Payroll'

-- Specify an instance of SQL Server in OPENDATASOURCE

```
-- using Data Source.
SELECT *
FROM OPENDATASOURCE(
'SQLOLEDB',
'Data Source=London/Payroll;User ID=MyUID;Password=MyP
).Northwind.dbo.Categories
```

```
-- Specify an instance of SQL Server in OPENROWSET

-- using a provider string.

SELECT a.*

FROM OPENROWSET(

'SQLOLEDB',

'Data Source=London;INSTANCENAME=Payroll;

User ID=MyUID;Password=MyPass',

Northwind.dbo.Categories

) AS a
```

```
-- Specify an instance of SQL Server in OPENROWSET

-- using a the datasource parameter.

SELECT a.*

FROM OPENROWSET(

'SQLOLEDB','London/Payroll','MyUID','MyPass',

'SELECT * FROM Northwind.dbo.Categories'

) AS a
```

OLE DB Provider for ODBC

The OLE DB Provider for ODBC provides an OLE DB interface to ODBC data sources. Using the OLE DB Provider for ODBC, Microsoft® SQL Server[™] distributed queries can access all ODBC data.

Note For SQL Server 2000 data sources, use the OLE DB Provider for SQL Server. Do not use the OLE DB Provider for ODBC.

To create a linked server to access an ODBC database when using an ODBC data source:

- 1. Create a System data source on the computer on which SQL Server is installed.
- 2. Execute **sp_addlinkedserver** to create the linked server, specifying **MSDASQL** or NULL as *provider_name*, and the name of an ODBC system data source as *data_source*.

ODBC user data sources cannot be used for distributed queries because SQL Server runs as a service on Microsoft Windows NT®, and services do not always have access to user data sources. For example, a system data source with a name of SystemDSN references a server that is running SQL Server and that has **pubs** as the default database:

sp_addlinkedserver 'SQLPubs', ' ', 'MSDASQL', 'SystemDSN'

Linked servers can use the OLE DB Provider for ODBC without using an ODBC data source. The linked server is defined in one step. All the information the OLE DB Provider for ODBC needs to locate an ODBC driver and connect to a source of ODBC data must be defined in *provider_string*.

This example creates a linked server named **SQLPubs** on the **pubs** database of the SQL Server named **SalesSvr** through ODBC by specifying *provider_string*.

```
sp_addlinkedserver 'SQLPubs', ' ', 'MSDASQL', NULL, NULL,
'Driver={SQL Server};Database=pubs;Server=SalesSvr;UID=sa;PWD
```

These restrictions exist if a linked server is defined using the OLE DB Provider for ODBC and accesses a SQL Server database:

- Tables cannot be referenced if they have one or more **timestamp** columns.
- Tables cannot be referenced if they have nullable **char**, **varchar**, **nchar**, **nvarchar**, **binary**, or **varbinary** columns and the ANSI_PADDING option was set OFF when the table was created.

OLE DB Provider for Jet

The Microsoft® OLE DB Provider for Jet provides an OLE DB interface to Microsoft Access databases, and allows Microsoft SQL Server[™] 2000 distributed queries to query Access databases.

To create a linked server to access an Access database

1. Execute sp_addlinkedserver to create the linked server, specifying Microsoft.Jet.OLEDB.4.0 as provider_name, and the full path name of the Access .mdb database file as data_source. The .mdb database file must reside on the server. data_source is evaluated on the server, not the client, and the path must be valid on the server.

For example, to create a linked server named **Nwind** that operates against the Access database named **Nwind.mdb** in the C:\Mydata directory, execute:

sp_addlinkedserver 'Nwind', 'Access 97', 'Microsoft.Jet.OLEC 'c:\mydata\Nwind.mdb'

2. To access an unsecured Access database, SQL Server logins attempting to access an Access database should have a login mapping defined to the username Admin with no password.

This example enables access for the local user **Joe** to the linked server named **Nwind**.

sp_addlinkedsrvlogin 'Nwind', false, 'Joe', 'Admin', NULL

To access a secured Access database, configure the registry (using the Registry Editor) to use the correct Workgroup Information file used by Access. Use the Registry Editor to add the full path name of the Workgroup Information file used by Access to this registry entry:

HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Jet\4.0\

After the registry entry is configured, use **sp_addlinkedsrvlogin** to create login mappings from local logins to Access logins:

sp_addlinkedsrvlogin 'Nwind', false, 'Joe', 'AccessUser', 'AccessPwd'

Access databases do not have catalog and schema names. Therefore, tables in an Access-based linked server can be referenced in distributed queries using a four-part name of the form *linked_server...table_name*.

This example retrieves all rows from the **Employees** table in the linked server named **Nwind**.

SELECT * FROM Nwind...Employees

To create a linked server against an Excel spreadsheet:

The Microsoft OLE DB Provider for Jet 4.0 can be used to access Microsoft Excel spreadsheets.

• To create a linked server that accesses an Excel spreadsheet, use the format of this example.

sp_addlinkedserver N'Excel', N'Jet 4.0',

```
N'Microsoft.Jet.OLEDB.4.0',
```

N'c:\data\MySheet.xls', NULL, N'Excel 5.0'

GO

sp_addlinkedsrvlogin N'Excel', false, sa, N'ADMIN', NULL GO

To access data from an Excel spreadsheet, associate a range of cells with a name. A named range can be accessed by using the name of the range as the table name. The following query can be used to access a named range called SalesData using the linked server set up in the previous example.
 SELECT *
 FROM EXCEL...SalesData

When you insert a row into a named range of cells, the row will be

added after the last row that is part of the named range of cells. Thus, if you want to insert row **rA** after the column heading, associate the column heading cells with a name and use that name as the table name. The range of cells will grow automatically as rows are inserted.

To set up a linked server against a formatted text file:

Microsoft OLE DB Provider for Jet can be used to access and query text files.

• To create a linked server for accessing text files directly without linking the files as tables in an Access .mdb file, execute **sp_addlinkedserver**, as in this example.

The provider is Microsoft.Jet.OLEDB.4.0 and the provider string is 'Text'. The data source is the full path name of the directory that contains the text files. A schema.ini file, which describes the structure of the text files, must exist in the same directory as the text files. For more information about creating a schema.ini file, see the Jet Database Engine documentation.

```
--Create a linked server.
EXEC sp_addlinkedserver txtsrv, 'Jet 4.0',
'Microsoft.Jet.OLEDB.4.0',
'c:\data\distqry',
NULL,
'Text'
GO
```

--Set up login mappings. EXEC sp_addlinkedsrvlogin txtsrv, FALSE, NULL, Admin, NI GO

--List the tables in the linked server. EXEC sp_tables_ex txtsrv GO

--Query one of the tables: file1#txt

--using a 4-part name. SELECT * FROM txtsrv...[file1#txt]

OLE DB Provider for DTS Packages

The Microsoft OLE DB Provider for DTS Packages is a read-only provider that exposes Data Transformation Services Package Data Source Objects. The provider can be used to expose the rowset from a package using either OPENROWSET or referencing the package using a four-part name in a Transact-SQL statement. OPENQUERY can also be used to send a command to the provider using its command language. For more information about using the Microsoft OLE DB Provider for DTS Packages, see <u>DTS Driver Support for Heterogeneous Data Types</u>.

To define a linked server to access a DTS package data source object after the data source object has been defined in DTS

• Execute **sp_addlinkedserver** to create the linked server, specifying **DTSPackageDSO** as *provider_name*, the package name as *product_name*, and switches for the **dtsrun** command prompt utility as *data_source*.

For example, to create a linked server named **MyDTSPackage** that accesses a DTS package saved to the file C:\Dts\DTSFilePackage.dts, execute:

sp_addlinkedserver MyDTSPackage, 'PackageName', 'DTSPac

Then, the rowset exposed by this DTS package can be referenced using either the name of the DTS package or the name of a step in the DTS package:

SELECT * FROM MyDTSPackage...DTSStep1

The package name or step name are defined in DTS.

In addition, DTS packages can be referenced using the OPENROWSET function. For more information, see <u>Querying a DTS Package from External</u> <u>Sources</u>.

OLE DB Provider for Oracle

The Microsoft OLE DB Provider for Oracle allows distributed queries to query data in Oracle databases.

Note Oracle client software does not support distributed queries using Microsoft OLE DB Provider for Oracle when the distributed transactions are executed from an instance of SQL Server installed on Microsoft Windows® 98.

To create a linked server to access an Oracle database instance

- Ensure the Oracle client software on the server running SQL Server is at the level required by the provider. The Microsoft OLE DB Provider for Oracle requires Oracle Client Software Support File version 7.3.3.4.0 or later, and SQL*Net version 2.3.3.0.4.
- 2. Create an SQL*Net alias name on the server running SQL Server that points to an Oracle database instance. For more information, see the Oracle documentation.
- 3. Execute **sp_addlinkedserver** to create the linked server, specifying **MSDAORA** as *provider_name*, and the SQL*Net alias name for the Oracle database instance as *data_ source*.

This example assumes that an SQL*Net alias name has been defined as **OracleDB**.

sp_addlinkedserver 'OrclDB', 'Oracle', 'MSDAORA', 'OracleE

4. Use **sp_addlinkedsrvlogin** to create login mappings from SQL Server logins to Oracle logins.

This example maps the SQL Server login **Joe** to the linked server defined in Step 3 using the Oracle login and password OrclUsr and OrclPwd:

sp_addlinkedsrvlogin 'OrclDB', false, 'Joe', 'OrclUsr', 'OrclPw

Each Oracle database instance has only one catalog with an empty name. Tables in an Oracle linked server must be referenced using a four-part name of the form *OracleLinkedServerName..OwnerUserName.TableName*. For example, this SELECT statement references the table SALES owned by the Oracle user **MARY** in the server mapped by the OrclDB linked server:

SELECT * FROM OrclDB..MARY.SALES

Use these rules when referencing tables in an Oracle linked server:

- If the table and column names were created in Oracle without quoted identifiers, use all uppercase names.
- If the table and column names were created in Oracle with quoted identifiers, use the same case for all letters of the names as was used when the names were created in Oracle.
- INSERT statements should supply values for all columns in a table even if certain columns in the table can be NULL or have default values.

Registry Entries

To enable the OLE DB Provider for Oracle to work with your Oracle client software, the client's registry must be modified by running a registry file from a command line. Multiple instances of the client software should not run concurrently. These files are listed in the following table and are located within the same directory structure that contains your Microsoft Data Access Component (MDAC) installation, which typically is in C:\Program Files\Common Files\System Files\OLE DB.

Oracle client	Windows NT or 9x	Windows 2000
7.x	mtxoci7x_winnt.reg	mtxoci7x_win2k.reg
8.0	mtxoci80x_winnt.reg	mtxoci80x_win2k.reg
8.1	mtxoci81x_winnt.reg	mtxoci81x_win2k.reg

OLE DB Provider for Microsoft Directory Services

The Microsoft OLE DB Provider for Microsoft Directory Services provides access to information in the Microsoft® Windows® 2000 Directory Service. This OLE DB Provider supports two command dialects, LDAP and SQL, to access the directory service and return results in a tabular form that can be queried using SQL Server distributed queries.

To create a linked server against Windows 2000 Directory Service

Create a linked server using ADSDSOObject as the *provider_name* and adsdatasource as the *data_source* argument of the sp_addlinkedserver system stored procedure.
 EXEC sp_addlinkedserver 'ADSI', 'Active Directory Services 2 'ADSDSOObject', 'adsdatasource'
 GO

For Windows authenticated logins, the self-mapping is sufficient to access the directory using SQL Server Security Delegation. Because the self-mapping is created by default for linked servers created through **sp_addlinkedserver**, no other login mapping is necessary.

For SQL Server authenticated logins, suitable login/passwords can be configured for connecting to the directory service using the **sp_addlinkedsrvlogin** system stored procedure.

Querying the Directory Service

The Microsoft OLE DB Provider for Microsoft Directory Services supports two command dialects, LDAP and SQL, to query the Directory Service. The OPENQUERY function can be used to send a command to the Directory Service and consume its results in a SELECT statement.

The following example shows creating a view that uses OPENQUERY to return information from the directory at the server *ADSISrv* whose domain address is *sales.northwind.com*. The command inside the OPENQUERY function is an SQL query against the directory to return the *Name*, *SN*, and *ST* attributes of

```
objects belonging to Class Contact at a specified hierarchical location (OU=Sales) in the directory. The view then can be used in any SQL Server queries.
```

```
CREATE VIEW viewADContacts
AS
SELECT [Name], SN [Last Name], ST State
FROM OPENQUERY( ADSI,
'SELECT Name, SN, ST
FROM "LDAP://ADSISrv/ OU=Sales,DC=sales,DC=northwind,DC=
WHERE objectCategory = "Person" AND
objectClass = "contact"")
GO
SELECT * FROM viewADContacts
```

For more information about the LDAP and SQL dialects, see Microsoft Active Directory Services documentation.

OLE DB Provider for Microsoft Indexing Service

Microsoft® Windows 2000 includes Microsoft Internet Information Services (IIS) and Microsoft Indexing Service version 3.0. These services enable filtering files based on their properties and performing full-text indexing and retrieval of file data. (For Microsoft Windows NT® 4.0 systems, the Windows NT 4.0 Option Pack includes IIS 4.0 and Microsoft Indexing Service 2.0.)

Indexing Service also includes the Microsoft OLE DB Provider for Microsoft Indexing Service. This provider can be used to perform full-text or property value searches on nondatabase files. A linked server definition can be made using **sp_addlinkedserver**, and security can be set up using SQL Server Enterprise Manager. Distributed queries can then reference the provider to retrieve indexing information.

To create a linked server to access an Indexing Service full-text index

- 1. Create the full-text index using Indexing Service. By default, Indexing Service installs a catalog named **default**. For more information, see the Indexing Service documentation.
- 2. Execute **sp_addlinkedserver** to create the linked server, specifying **MSIDXS** as *provider_name*, and the name of the full-text index as *data_source*.

For example, to create a linked server named **FTIndexWeb** that accesses a full-text index named **Web**, execute:

sp_addlinkedserver FTIndexWeb, 'Index Server', 'MSIDXS', '\

3. The security authorization of Indexing Service clients is based on the Windows account of the process consuming the OLE DB Provider for Microsoft Index Service. Distributed queries are run in the context of the Microsoft SQL Server[™] 2000 process. Because SQL Server typically runs under an account that has a high level of authorization, some SQL Server users use an Indexing Service linked server to access information that they are not authorized to access. Members of the

sysadmin fixed server role address this problem by strictly controlling the SQL Server logins that are authorized to perform distributed queries using an Indexing Service linked server.

The administrator first uses **sp_droplinkedsrvlogin** to remove all login mappings to the Indexing Service linked server, for example:

sp_droplinkedsrvlogin FTIndexWeb, NULL

The administrator then uses **sp_addlinkedsrvlogin** to authorize individual logins to access the linked server, for example:

sp_addlinkedsrvlogin FTIndexWeb, true, 'SomeLogin'

To perform these actions, right-click the linked server in SQL Server Enterprise Manager, click **Properties**, and then select the **Security** tab.

In addition, ad hoc access to the OLE DB Provider for Microsoft Indexing Service using OPENQUERY can be disabled by setting the provider option **DisallowAdhocAccess**. For more information, see <u>Configuring OLE DB Providers for Distributed Queries</u>.

Transact-SQL statements can use the OPENQUERY function to send commands to Indexing Service using an SQL syntax that is consistent with the full-text query syntax supported in SQL Server for full-text searches of data stored in the database. The SQL full-text syntax supported by Indexing Service is defined in the Index Server Programmer's Guide found either on the Microsoft Development Network or in the Indexing Service documentation in the Windows NT 4.0 Option Pack. For more information about using the OLE DB Provider for Microsoft Indexing Service in Transact-SQL statements, see <u>Full-text Querying</u> <u>of File Data</u>.

OLE DB Provider for DB2

The Microsoft OLE DB Provider for DB2, distributed with Microsoft® Host Integration Server 2000, allows Microsoft SQL Server[™] 2000 distributed queries to query data in DB2 databases.

To create a linked server to access a DB2 database

- 1. Install the Windows NT Client for Host Integration Server 2000 or the Windows 9x Client for Host Integration Server 2000 on a computer running an instance of SQL Server. Select the options to install the OLE DB Provider for DB2 and the network components needed to communicate with an IBM computer running in an SNA network.
- 2. Determine the connection string the OLE DB Provider for DB2 needs to access the DB2 data source you want to query. The best way to determine a connection string is to build a Data Link file using the Host Integration Server New OLE DB Data Source application. For more information, see the Microsoft Host Integration Server 2000 documentation.
- 3. Execute **sp_addlinkedserver** to create a linked server, specifying DB2OLEDB as the *provider_name*, the name of the DB2 catalog containing the data you want to access as *catalog*, and the connection string from Step 2 as *provider_string*.

This example shows how to use **sp_addlinkedserver** to create a linked server definition accessing a DB2 database:

EXEC sp_addlinkedserver @server = 'DB2SRV', @srvproduct = 'Microsoft OLE DB Provider for DB2', @catalog = 'SEATTLE', @provider = 'DB2OLEDB', @provstr = 'NetLib=SNA;NetAddr=;NetPort=;RemoteLU=SEATTLE; ModeName=QPCSUPP;InitCat=SEATTLE; Default Schema=WNW3XX;PkgCol=WNW3XX;TPNam IsoLvl=NC;AccMode=;CCSID=37;PCCodePage=1252;Bi Data Source=Seattle_WNW3XX'

4. Execute **sp_addlinkedsrvlogin** to create login mappings from SQL Server 2000 logins to DB2 logins.

This example maps the SQL Server 2000 login **SQLJoe** to DB2 login **DB2Joe**:

EXEC sp_addlinkedsrvlogin 'DB2SRV', false, 'SQLJoe', 'DB2

After completing these steps, you can use the linked server name DB2SRV as the server name in four part names and as *linked_server* in the OPENQUERY function. For example:

SELECT * FROM DB2SRV.SEATTLE.WNW3XX.DEPARTMENT

Or

SELECT * FROM OPENQUERY(DB2SRV, 'SELECT * FROM SEATTLE.WNW

When the distributed queries against DB2 data sources involve NULL comparisons, use IS NULL or IS NOT NULL rather than comparison operators, such as =, <, or >. In addition, INSERT statements should supply values for all columns in a table even if certain columns in the table can be NULL or have default values.

OLE DB Provider for Exchange

The Microsoft® OLE DB Provider for Exchange exposes data stored in a Microsoft Exchange 2000 Web Store in tabular form. This data can be queried using an SQL-like language that is very similar to the SQL subset supported by the OLE DB Provider for Microsoft Indexing Service.

Microsoft SQL Server[™] 2000 distributed queries can be used to query data from the Exchange Web Store through this OLE DB Provider and can be joined with tables in SQL Server. The Exchange Web Store should be located in the same computer as SQL Server. Web Stores located in other computers cannot be accessed using the OLE DB Provider for Exchange.

The OLE DB Provider for Exchange is available as part of Microsoft Exchange 2000.

To create a linked server against an Exchange Web Store

• Use 'exoledb.datasource.1' as the *provider_name* argument, and the URL corresponding to the root folder of the Web Store as the *data_source* argument of the **sp_addlinkedserver** system stored procedure.

EXEC sp_addlinkedserver 'exchange', 'Exchange OLE DB provider', 'exoledb.DataSource.1', 'file:\\.\backofficestorage\localhost\public folders'

• For Windows NT Authenticated logins, there are no login mappings necessary. They are impersonated by SQL Server when it connects to the OLE DB Provider for Exchange. For SQL Authenticated logins, set up login mappings by supplying the user name and password, as necessary.

The following restrictions are applicable when querying data from the OLE DB Provider for Exchange:

• Only pass-through queries are supported. Four-part names cannot be used against the Exchange OLE DB provider.

- All character columns from the OLE DB Provider for Exchange are exposed to SQL Server as **ntext** columns. In order to perform comparisons against these columns, they have to be converted explicitly to **nvarchar** using the CONVERT function.
- Multi-valued columns from the Exchange provider with OLE DB DBTYPE DBTYPE_VECTOR are not supported from SQL Server Distributed Queries.

To access data in the Exchange Web Store from SQL Server, through a linked server established as above

• Create views that retrieve the required properties as columns from the Web Store folder of interest. The view definition converts string columns to **nvarchar** so that they can be filtered through conditions in the WHERE clause.

For example, let the Web Store contain a folder called Contacts that contains a list of contacts. The following script creates a view against the Contacts folder while retrieving the Contact's first name, last name, company name, and date of birth.

CREATE VIEW Contacts

AS

SELECT convert(nvarchar(30),"urn:schemas:contacts:sn") Las Convert(nvarchar(30),"urn:schemas:contacts:givenname") FirstName,

Convert(nvarchar(30), "urn:schemas:contacts:o") Company, Convert(nvarchar(50), "urn:schemas:contacts:email1") Emai

"urn:schemas:contacts:bday" BirthDay

FROM OpenQuery(Exchange,

'SELECT "urn:schemas:contact:sn",

"urn:schema:contacts:givenname",

"urn:schemas:contacts:o",

"urn:schemas:contacts:email1"

```
"urn:schemas:contacts:bday"
FROM SCOPE(".\contacts")'
```

Now the views can be queried and joined with local SQL Server tables like regular tables. For example, the Contacts view can be joined with a local Suppliers table to determine Contact information for the list of Supplier companies.

SELECT FirstName, LastName, Email, Company FROM Suppliers S, Contacts C WHERE S.Company = C.CompanyName

For information on the SQL language supported by the Exchange OLE DB provider see Microsoft Exchange 2000 documentation.

OLE DB Provider Reference for Distributed Queries

The behavior of distributed queries against a remote table depends on the functionality of the OLE DB provider used to access the table. The OLE DB specification defines a set of objects for OLE DB providers. Each object has a set of interfaces. Many of these objects and interfaces are optional and may not be supported by a provider. If an OLE DB provider does not support some of these objects and interfaces, distributed query functionality that depends on these components will not work with remote tables accessed through that provider.

OLE DB Objects Consumed by Distributed Queries

The following table shows the OLE DB objects and interfaces consumed by Transact-SQL distributed queries. An OLE DB provider must support at least a minimal set of objects and interfaces before it can be used in any Transact-SQL distributed queries. These objects and interfaces are marked with a Yes in the Required column of the table. The objects and interfaces with a No in the Required column are needed only to support advanced distributed query functionality. If the interface is not supported by the provider, related Transact-SQL functionality is not supported. For example, if the **IRowsetLocate** and **IRowsetChange** interfaces are not supported, UPDATE or DELETE statements do not function on remote tables.

Object	Interface	Required	Description
Data	IDBInitialize	Yes	Initialize and set up data
Source			and security context.
	IDBCreateSession	Yes	Create a DB Session object.
	IDBProperties	Yes	Get information about the
			capabilities of provider and
			set initialization properties.
	IDBInfo	No	Get information about the
			SQL syntax supported by
			the provider.
DB Session	IDBSchemaRowset	No	Get table and/or column
			meta data. Rowsets needed
			are TABLES and
			COLUMNS. Other rowsets
			used, if available, are
			TABLES_INFO,
			CATALOGS, INDEXES,
			STATISTICS,
			TABLE_STATISTICS,
			VIEWS,
			PRIMARY_KEYS,
			TABLE_PRIVILEGES, and

			COLUMN_PRIVILEGES.
	IopenRowset	Yes	Open a rowset on a table, index, or histogram.
	IGetDataSource	Yes	Return to the data source object from a DB Session object.
	IDBCreateCommand	No	Create a command object (query).
	ITransactionLocal	No	Use to start a transaction on the provider.
	ITransactionJoin	No	Use for distributed transaction support. If this interface is not supported, updates against a remote provider are not allowed in a user transaction.
Rowset (on a table)	Irowset	Yes	Scan rows.
	Iaccessor	Yes	Bind to columns in a rowset.
	IcolumnsInfo	Yes	Get information about columns in a rowset.
	IrowsetInfo	Yes	Get information about rowset properties.
	IRowsetLocate	No	Required for UPDATE or DELETE operations and index-based lookups.
	IRowsetChange	No	Required for INSERT, UPDATE, or DELETE operations on a table. Rowsets against base tables should support this interface for supporting INSERT, UPDATE, or DELETE statements.

	IconvertType	Yes	Verify if a rowset supports specific data type conversions on its columns.
Rowset (on an index)	Irowset	Yes	Scan rows.
	Iaccessor	Yes	Bind to columns in a rowset.
	IcolumnsInfo	Yes	Get information about columns in a rowset.
	IrowsetInfo	Yes	Get information about rowset properties.
	IrowsetIndex	Yes	Required for rowsets on an index; used for indexing functionality (set range, seek).
	IconvertType	Yes	Verify if the rowset supports specific data type conversions on its columns.
Command Ico (optional)	Icommand	Yes	Use for executing queries.
	ICommandText	Yes	Use for defining the query text.
	ICommandProperties	Yes	Specify required properties on rowsets returned by the command.
	ICommandWith Parameters	No	Use for parameterized query execution.
	ICommandPrepare	No	Use for preparing a command to get meta data.
Error (optional)	IerrorRecords	Yes	Get a pointer to an IErrorInfo interface for an error record.
	IerrorInfo	Yes	Get a text description of an error record.
Any	ISupportErrorInfo	No	Determine if a given

Object	interface supports error
(optional)	objects.

Four-Part Name Requirements for OLE DB Providers

Distributed queries can use four-part names only if the OLE DB provider supports:

- The **IDBSchemaRowset** interface.
- Restrictions on all the name parts that it supports in the **IDBSchemaRowset** interface. Restrictions are a mechanism defined in OLE DB for specifying the search criteria for meta data queries using the OLE DB schema rowsets.
- The literals DBLITERAL_CATALOG_SEPARATOR, DBLITERAL_SCHEMA_SEPARATOR, and DBLITERAL_QUOTE in the **IDBInfo** interface. Microsoft® SQL Server[™] 2000 uses defaults, which may not work with providers that do not support these literals.

If an OLE DB provider does not meet these requirements, it can be referenced only using pass-through queries in the OPENDATASOURCE or OPENROWSET function.

UPDATE and DELETE Requirements for OLE DB Providers

Transact-SQL UPDATE and DELETE statements can reference remote tables only if the following conditions are met by the OLE DB provider that is used to access the remote table:

- The provider must support bookmarks on the rowset opened through **IOpenRowset** on the table being updated or deleted.
- The provider must support the **IRowsetLocate** and **IRowsetChange** interfaces on the rowset opened through **IOpenRowset** on the table being updated or deleted.
- The **IRowsetChange** interface must support update (**SetData**) and delete (**DeleteRows**) methods.

The Microsoft OLE DB Provider for SQL Server supports these interfaces only on a table that has a unique index. Consequently, UPDATE or DELETE statements are permitted against a remote table in another instance of Microsoft® SQL ServerTM only if the table has a unique index.

INSERT Requirements for OLE DB Providers

Transact-SQL INSERT statements can reference remote tables only if the following conditions are met by the OLE DB provider that is used to access the remote table:

- The provider must support the **IRowsetChange** interface on the rowset opened through **IOpenRowset** on the table having data inserted into it.
- The **IRowsetChange** interface on the base table being inserted into must support the insert (**InsertRow**) method.

Keyset-Driven Cursors Requirements for OLE DB Providers

Transact-SQL keyset-driven cursors can reference remote tables only if the following conditions are met:

- The distributed query must meet the requirements for SELECT statements used in a DECLARE CURSOR statement that declares the keyset-driven cursor. For more information about the Transact-SQL conditions for keyset-driven cursor support, see <u>DECLARE CURSOR</u>.
- All local tables in the query must have a unique index. The index of the remote table should be exposed through the INDEXES rowset of the **IDBSchemaRowset** interface.

Index Requirements on OLE DB Providers

SQL Server can use indexes on tables from an OLE DB provider to evaluate certain queries. For this, the provider should expose OLE DB interfaces that allow scanning an index rowset and seek into the base table rowset using bookmarks obtained from the index rowset.

Using the OLE DB provider's indexes has performance benefits only when the index and table rowsets are on the same computer as the instance of Microsoft® SQL ServerTM. Thus, the **Index AS Access Path** option should be set only if the data source is on the same computer as SQL Server.

SQL Server can use an OLE DB provider's indexes only if the following conditions are met:

- The provider must support the **IDBSchemaRowset** interface with the TABLES, COLUMNS, and INDEXES schema rowsets.
- The provider must support opening a rowset on an index through **IOpenRowset** by specifying the index name and the corresponding base table name.

- The Index object must support all its mandatory interfaces: **IRowset**, **IRowsetIndex**, **IAccessor**, **IColumnsInfo**, **IRowsetInfo**, and **IConvertTypes**.
- Rowsets opened against the indexed base table (through **IOpenRowset**) must support the **IRowsetLocate** interface for positioning on a row based off a bookmark.

If the OLE DB provider meets these requirements, the SQL Server administrator can set the **Index As Access Path** provider option to enable SQL Server to use the provider's indexes to evaluate the queries. By default, SQL Server does not attempt to use the provider's indexes unless this option is set.

Updatable Keyset Cursor Requirements

A remote table can be updated or deleted through a keyset cursor defined on a distributed query. For example:

UPDATE | DELETE remote_table WHERE CURRENT OF cursor_na

Here are the conditions under which updatable cursors against distributed queries are allowed:

- The provider should meet the conditions for updates and deletes on the remote table. For more information, see <u>UPDATE and DELETE</u> <u>Requirements for OLE DB Providers</u>.
- All the cursor operations must be in an explicit user transaction (or multi-statement transaction) with read-repeatable isolation level or serializable isolation level.

The provider must support distributed transactions with the **ITransactionJoin** interface.

Distribution Statistics Requirements for OLE DB Providers

Microsoft® SQL Server[™] 2000 defines extensions to the OLE DB specification that allow OLE DB providers to report statistics on numbers of rows and ranges of key values in the data they provide. SQL Server can use this information to increase the performance of distributed queries.

SQL is a nonprocedural language. SQL statements do not specify the steps needed to accomplish the result you want. The statements define the format of the result set and the conditions rows in base tables must meet to be used in building the result set. The database engine must analyze each SQL statement and determine the most efficient way to access the base tables. The part of the database engine that performs this task is called the optimizer.

The results of the optimization process is improved if the optimizer has access to statistics describing the distribution of the values in base table columns referenced in WHERE clause predicates. The distribution statistics used by the optimizer include:

- The number of rows in a table, also called the cardinality of the table.
- The number of distinct values stored in a column, also called the cardinality of the column.
- Information about how the distinct values in a column are distributed across the rows of the table.

To improve the optimization of distributed queries, SQL Server defines extensions to the OLE DB specification that OLE DB providers can use to report distribution statistics on the rowsets, or tables, they expose. While these extensions are defined in the SQL Server documentation, individual OLE DB provider developers must code support for the extensions in their providers if they want to make the information available to SQL Server. If a provider has code that supports the extensions, SQL Server can use the extensions to optimize the performance of distributed queries. If a provider does not support the extensions, SQL Server uses simple estimates of the distribution statistics.

Note The Microsoft OLE DB Provider for SQL Server and the Microsoft OLE DB Provider for Oracle support distribution statistics.

The distribution statistics extensions are built around a unit called a statistic. Each table can have zero or more statistics, and each statistic reports data for one or more columns. A statistic records:

- The cardinality of the values, or the number of unique values, in each individual column covered by the statistic.
- The cardinality of the concatenated values of all the columns covered by the statistic.
- Optionally, a histogram reporting information about different ranges of key values in the first column covered by the statistic. The values reported can include the number of rows in each key range, the number of unique values in each key range, or the number of rows in the table whose key values are less than or equal to the highest key value in the range.

ColumnA	ColumnB
'abc'	'xyz'
'abc'	'xyz'
'def'	'xyz'
'mno'	'xyz'
'mno'	'mmm'
'tuv'	'xyz'

Here is an example table.

If a statistic covers **ColumnA** and **ColumnB**, the cardinality of the combined values of the two columns is 5 because the first two rows have the same value ('abc' + 'xyz') for the combination of **ColumnA** and **ColumnB**. The cardinality of **ColumnA** is 4 and the cardinality of **ColumnB** is 2. A simple, 4-step

histogram on **ColumnA** could report.

Value range	Percentage of table rows in the range
'aaa' to 'hzz'	50%
'iaa' to 'nzz'	33%
'oaa' to 'rzz'	00%
'taa' to 'zzz'	17%

Different OLE DB data sources record distribution statistics on different combinations of columns, and the set of statistics reported by an OLE DB provider is implementation defined. For example, SQL Server versions 6.5 or earlier build distribution statistics only for columns covered by indexes and have one statistic for each index defined on a table. SQL Server version 7.0 and later builds these statistics:

- One statistic for each index defined on a table.
- One statistic for each CREATE STATISTIC statement.
- One statistic for each statistic generated automatically. For more information, see <u>Statistical Information</u>.

A column has a high degree of selectivity if it is likely to return a small number of rows for a given value specified in a predicate argument. The distribution statistics can be used to estimate the degree of selectivity:

- Columns with high cardinality have more data values, and each data value is likely to match a smaller number of rows than a column with low cardinality.
- If an OLE DB provider provides a histogram reporting how the values are distributed in a column, the SQL Server optimizer can also estimate if the specific value in a predicate argument is in a range that has good or poor selectivity.

Having good distribution statistics for a linked server can also help the optimizer build an efficient execution plan for the local part of a distributed query.

The SQL Server optimizer uses the distribution statistics in an attempt to reduce the amount of data that must be communicated between the OLE DB provider and SQL Server. For example, when performing a distributed join between **TableA** on the local server and **TableB** on a linked server, SQL Server can use the distribution statistics to determine which of these processes is most efficient:

- Send the rows from **TableA** that match non-join predicates to the linked server and have the linked server perform the join.
- Retrieve the rows from **TableB** that match non-join predicates to the local server and perform the join on the local server.

If an OLE DB provider does not report cardinality information about a column, the SQL Server optimizer estimates a low cardinality. If a provider does not report a distribution histogram for a statistic, the optimizer operates as if the values are evenly distributed in the rows of the table.

SQL Server uses the following extensions from OLE DB providers to report distribution statistics:

- A new data source property, DBPROP_TABLESTATISTICS, indicates if the provider reports distribution statistics.
- A new **IDBSchemaRowset**, TABLE_STATISTICS, lists the statistics available for a given base table, including column and row cardinality.
- **IOpenRowset::OpenRowset** is enhanced to accept new arguments identifying a statistic. When a statistic is specified, **OpenRowset** returns a histogram rowset showing the distribution of values in the first column covered by the statistic specified in **StatisticID**.

These extensions to OLE DB are included in OLE DB version 2.6. For information about these extensions regarding distribution statistics, see the OLE DB 2.6 specification.

An OLE DB provider can choose to implement a performance enhancement of sampling only a part of the rows in a base table to determine the distribution statistics and histograms. These providers should scale their cardinality and histogram data to reflect the total values for the table before reporting them in the TABLE_STATISTICS and histogram rowsets.

Whether or not an OLE DB provider keeps the data in the TABLE_STATISTICS and the histogram rowset up-to-date with the current contents of the base table is implementation defined.

SQL Dialect Requirements for OLE DB Providers

The level of SQL supported by an OLE DB provider determines how effectively Microsoft® SQL Server[™] 2000 delegates distributed query operations to the OLE DB provider. If a provider does not support SQL, but opens only rowsets, SQL Server must retrieve the entire rowset and perform all logical operations, even if the distributed query only needs a subset of the rows in the source rowset. If an OLE DB provider supports many SQL syntax elements, SQL Server generates more sophisticated queries that let the source provider filter unnecessary rows before returning the rowset to SQL Server.

The OLE DB specification defines a DBPROP_SQLSUPPORT property through which providers can report the level of SQL syntax they support. The minimum level of SQL support that SQL Server versions require in distributed queries are:

- SQL Server 2000: DBPROPVAL_SQL_SUBMINIMUM
- SQL Server 7.0: DBPROPVAL_SQL_ANSI92_ENTRY or DBPROPVAL_SQL_ODBC_CORE

In addition to supporting a lower level of SQL syntax from underlying OLE DB providers, SQL Server 2000 defines a new SQLPROPSET_OPTHINTS property set that providers can use to specify that they support individual SQL syntax elements that are beyond those defined for

DBPROPVAL_SQL_SUBMINIMUM. If a provider supports one or two features that can be used to optimize distributed queries, but does not support the full DBPROPVAL_SQL_ANSI92_ENTRY or DBPROPVAL_SQL_ODBC_CORE syntax, the provider can use the SQLPROPSET_OPTHINTS properties to notify SQL Server of the optimization features it does support.

DBPROPVAL_SQL_SUBMINIMUM Syntax

The requirements for DBPROPVAL_SQL_SUBMINIMUM are that the provider supports the features of DBPROPVAL_SQL_ODBC_MINIMUM, with these differences:

• Features in DBPROPVAL_SQL_ODBC_MINIMUM, but not in DBPROPVAL_SQL_SUBMINIMUM:

DDL statements. INSERT, UPDATE, and DELETE statements. Dynamic parameter markers. Multiple tables in the FROM clause.

• Features in DBPROPVAL_SQL_SUBMINIMUM, but not in DBPROPVAL_SQL_ODBC_MINIMUM:

Column aliases in the select list. Integer and exact numeric constants IS [NOT] NULL predicate.

Although the DBPROPVAL_SQL_SUBMINIMUM grammar is defined in relation to DBPROPVAL_SQL_ODBC_MINIMUM, the SQL Server distributed query optimizer never tests for DBPROPVAL_SQL_ODBC_MINIMUM.

The DBPROPVAL_SQL_SUBMINIMUM grammar uses these conventions.

Convention	Used for
UPPERCASE	SQL keywords.
italic	User-supplied parameters in SQL syntax.
(vertical bar)	Separating syntax items within brackets or braces. You can choose only one of the items.
[] (brackets)	Optional syntax items. Do not type the brackets.
{} (braces)	Required syntax items. Do not type the braces.
[,n]	Indicating that the preceding item can be repeated <i>n</i> number of times. Commas separate the occurrences.

[n]	Indicating that the preceding item can be repeated <i>n</i> number of times. The occurrences are separated by blanks.
<label> ::=</label>	The name for a block of syntax. This convention is used to group and label portions of lengthy syntax or a unit of syntax that can be used in more than one place within a statement. Each location in which the block of syntax can be used is indicated with the label enclosed in chevrons: <label>.</label>

This is the syntax grammar for DBPROPVAL_SQL_SUBMINIMUM, as expected by SQL Server:

<select_statement> ::=

SELECT [ALL | DISTINCT] <select_list>

FROM <table_reference_list>

[WHERE <search_condition>]

[<order_by_clause>]

SELECT clause

<select_list> ::= * | <select_sublist>[,...n]

<select_sublist> ::=

expression [<alias>]

<alias> ::= <user_defined_name>

FROM clause

<table_reference_list> ::=

<table_reference>

<table_reference> ::=

<table_name>

<table_name> ::=

<table_identifier>

<table_identifier> ::=

<user_defined_name>

WHERE clause

<search_condition> ::=

<boolean_term> [OR <search_condition>]

<boolean_term> ::=

<boolean_factor> [AND <boolean_term>]

<boolean_factor> ::=

[NOT] <boolean_primary>

<boolean_primary> ::=

<comparison_predicate> | <search_condition>

<comparison_predicate> ::=

<expression> <comparison_operator> <expression>

| <expression> IS [NOT] NULL

<comparison_operator> ::=

< | > | <= | >= | = | <>

ORDER BY clause

```
<order_by_clause> ::=
```

ORDER BY <sort_specification>[,...n]

<sort_specification> ::=

{ | <column_name> } [ASC | DESC]

Common Syntactic Elements

```
<expression> ::=
```

```
<term> | <expression> {+|_} <term>
```

<term> ::= <factor> | <term> {*|/} <factor>

<factor>::= [+|-] <primary>

<primary> ::= <column_name> | <literal> | (<expression>)

<column_name> ::=

[<table_name>].<column_identifier>

```
literal> ::= <character_string_literal>
```

| <integer_literal>

| <exact_numeric_literal>

```
<character_string_literal> ::=
```

'{character}[...n]'

Character is any character in the character set of the provider or data source. Use two single quotation marks (") to represent a single quotation mark (apostrophe) in the literal string.

<integer_literal> ::=

[+|-] <unsigned_integer>

<exact_numeric_literal>::=

[+|-] <unsigned_integer> [<period><unsigned_integer]

| <period><unsigned_integer>

<column_identifier> ::=

<user_defined_name>

<user_defined_name> ::=

```
<letter>[<digit>|<letter>|_][...n]
```

<unsigned_integer> ::=

{<digit>}[...n]

<digit> ::= 0|1|2|3|4|5|6|7|8|9

<letter> ::= <lower_case_letter>|<upper_case_letter>

<lower_case_letter> ::=

a|b|c|d|e|f|g|h|I|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|z

<upre><upre>case_letter> ::=

A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|X|Y|Z

<period> ::= .

Programming the SQLPROPSET_OPTHINTS Property Set

Individual OLE DB providers can support some SQL functionality beyond that defined in DBPROPVAL_SQL_SUBMINIMUM, but not all of the functionality in DBPROPVAL_SQL_ODBC_CORE or

DBPROPVAL_SQL_ANSI92_ENTRY. The Microsoft® SQL Server[™] 2000 query optimizer can use some of the functionality supported by these drivers to increase the performance of distributed queries. These providers can use the SQLPROPSET_OPTHINTS property set to inform SQL Server of the features they support that can speed distributed queries.

Although the SQLPROPSET_OPTHINTS property set is defined in the SQL Server documentation, individual OLE DB provider developers must code support for the property set in their providers. After support for this property set is coded into the provider, SQL Server uses it to optimize the performance of distributed queries.

OLE DB providers that support DBPROPVAL_SQL_ANSI92_ENTRY or DBPROPVAL_SQL_ODBC_CORE do not need any of the SQLPROPSET_OPTHINTS properties, except for SQLPROP_DATELITERALS. These providers must support all of the functionality covered by the SQLPROPSET_OPTHINTS property set (except for SQLPROP_DATELITERALS) to qualify for DBPROPVAL_SQL_ANSI92_ENTRY or DBPROPVAL_SQL_ODBC_CORE support.

Property	Description
SQLPROP_ANSILIKE	Specifies the LIKE clause is supported as
	defined in the SQL-92 Entry Level, with
	the % and _ wildcards.
SQLPROP_DATELITERALS	Specifies the provider supports datetime
	literals, or constants as per Transact-SQL
	syntax.

These are the properties reported through SQLPROPSET_OPTHINTS.

SQLPROP_DYNAMICSQL	Specifies the provider supports the ODBC parameter marker syntax using question marks: ?.
SQLPROP_INNERJOIN	Specifies the provider supports references to multiple tables in the WHERE clause, as long as they are not outer join references.
SQLPROP_GROUPBY	Specifies the provider supports the GROUP BY and HAVING clauses in a SELECT statement. The property also specifies the provider supports the AVG, COUNT, MIN, MAX, and SUM aggregate functions, as long as DISTINCT is not specified as an aggregate argument.
SQLPROP_NESTEDQUERIES	Specifies the provider supports nested SELECT statements in the FROM clause.
SQLPROP_SUBQUERIES	Specifies the provider supports subqueries as defined in the SQL-92 Entry Level.

These are the constants used to define the SQLPROPSET_OPTHINTS property set in the code of OLE DB providers:

```
Extern const GUID SQLPROPSET_OPTHINTS =
```

```
{ 0x2344480c, 0x33a7, 0x11d1,
```

```
{ 0x9b, 0x1a, 0x0, 0x60, 0x8, 0x26, 0x8b, 0x9e }
```

};

```
enum SQLPROPERTIES
```

```
{
```

```
SQLPROP_NESTEDQUERIES = 0x4,
```

```
SQLPROP_DYNAMICSQL = 0x5,
```

```
SQLPROP_GROUPBY = 0x6,
```

```
SQLPROP_DATELITERALS = 0x7,
```

```
SQLPROP_ANSILIKE = 0x8,
SQLPROP_INNERJOIN = 0x9,
SQLPROP_SUBQUERIES = 0x10,
```

}

Full-text Search

Traditionally, retrieving specific text data from database columns or file systems has been a cumbersome and expensive process, often requiring third-party tools.

Microsoft® SQL Server[™] provides a rich text data retrieval system. SQL Server 2000 offers an enhanced full-text search service that allows you to:

• Update indexes in the background.

Populating or updating an index does not have to interfere with other tasks. Full-text index updates can be scheduled in the background using the Full-text Indexing wizard, SQL Server Enterprise Manager, or the SQL Server Agent job scheduler.

• Choose among three methods of maintaining a full-text index.

Depending on your data and resources, you can choose among the full rebuild, the timestamp-based incremental rebuild, and the change tracking methods to maintain your full-text indexes. The full rebuild method involves rescanning all rows. The timestamp-based incremental rebuild method only rescans those rows that have changed since the last rebuild (full or incremental) of the index. With the change tracking method, SQL Server maintains a list of all changes to the indexed data and you can use this list to update the full-text index. For more information, see <u>Maintaining Full-text Indexes</u>.

• Index and search certain types of data stored in **image** columns.

Using full-text search, you can index and query certain types of data stored in **image** columns. Full-text search uses one of several supported filters to interpret the data and extract the text data for indexing and querying. SQL Server provides filters for the .doc, .xls, .ppt, .txt, and .htm file extensions. For more information, see <u>Filtering Supported File</u> <u>Types</u>.

Once the **image** column is indexed, you can search the column using the search predicates CONTAINS and FREETEXT. For more information, see <u>Using Full-text Predicates to Query Image Columns</u>.

• Limit the number of matches returned.

When you use the optional *top_n_by_rank* argument of the CONTAINSTABLE or FREETEXTTABLE rowset function in your query, SQL Server will only return the top ranked matches, up to the *n* number specified. For more information, see Limiting Result Sets in the Using the CONTAINSTABLE and FREETEXTTABLE Rowset-valued Functions.

See Also

CONTAINS CONTAINSTABLE FREETEXT FREETEXTTABLE

sp_fulltext_columns

Full-text Querying SQL Server Data

Digital information is stored in both database columns and in the file system as unstructured data, primarily text. Some text data is stored in database character-type columns as well. For example, in Microsoft® SQL Server[™], such data can be stored in database columns with the **char**, **varchar**, **text**, **ntext**, **nchar**, or **nvarchar** data types.

Consequently, a method for retrieving this text data from the database is needed. Relational database management systems traditionally have had limited capabilities for finding patterns in textual data. For example, a system may be able to retrieve text based on pattern matching, but cannot handle searches that look up words and phrases in close proximity to one another.

Previously, corporate users of relational database management systems had to buy expensive third-party offerings to retrieve or query data stored in these character-based database columns. These solutions typically involved a two-step process:

- 1. Pull data out of the database through a bridge or gateway.
- 2. Store the data as character-based operating-system files so that full-text indexing can be applied.

Using this two-step process meant that there was no seamless way for combining a full-text query with a regular, structured relational query. SQL Server solves this problem by allowing full-text queries to be issued against plain characterbased data in SQL Server tables, including words and phrases, or multiple forms of a word or phrase.

To enable a database for full-text indexing

Full-text Index and Querying Concepts

The principal design requirement for full-text indexing, querying, and synchronization is the presence of a full-text unique key column (or singlecolumn primary key) on all tables that are registered for full-text search. A fulltext index keeps track of which significant words are used and where they are located.

For example, consider a full-text index for a **DevTools** table. A full-text index may indicate that the word Microsoft is found at word number 423 and word number 982 in the **Abstract** column for the row associated with a ProductID of 6. This index structure supports an efficient search for all items containing indexed words and advanced search operations, such as phrase searches and proximity searches.

To prevent the full-text index from becoming bloated with words that do not help the search, extra words such as a, and, is, or the are ignored. For example, specifying the phrase "the products ordered during these summer months" is the same as specifying the phrase "products ordered during summer months." Rows with either string are returned.

Noise-word lists for many languages are provided in the directory \Mssql\Ftdata\Sqlserver\Config. This directory is created, and the noise-word files are installed when you set up Microsoft® SQL Server[™] with the full-text search support. The noise-word files can be edited. For example, system administrators at high-tech companies might add the word computer to their noise-word list. (If you edit a noise-word file, you must repopulate the full-text catalogs before the changes will take effect.) The table shows the noise-word files and their respective languages.

Noise-word file	Language
Noise.chs	Simplified Chinese
Noise.cht	Traditional Chinese
Noise.dat	Language Neutral
Noise.deu	German
Noise.eng	English UK

Noise.enu	English US
Noise.esn	Spanish
Noise.fra	French
Noise.ita	Italian
Noise.jpn	Japanese
Noise.kor	Korean
Noise.nld	Dutch
Noise.sve	Swedish

When processing a full-text query, the search engine returns the key values of the rows that match the search criteria to Microsoft SQL Server. Consider a **SciFi** table in which the **Book_No** column is the primary key column.

Book_No	Writer	Title
A025	Asimov	Foundation's Edge
A027	Asimov	Foundation and Empire
C011	Clarke	Childhood's End
V109	Verne	Mysterious Island

If you want to use a full-text retrieval query to find the book titles that include the word Foundation. In this case, the values of A025 and A027 are obtained from the full-text index. SQL Server then uses these keys and other field information to respond to the query.

This table shows the language in which the full-text index data is stored. The language is based on the Unicode collation locale identifier selected during SQL Server Setup.

Language for full-text data storage
Traditional Chinese
Simplified Chinese
Simplified Chinese
Traditional Chinese
Dutch

English UK	English UK
French	French
General Unicode	English US
German	German
German Phonebook	German
Italian	Italian
Japanese	Japanese
Japanese Unicode	Japanese
Korean	Korean
Korean Unicode	Korean
Spanish Modern	Spanish
Swedish/Finnish	Swedish

All other Unicode collation locale identifier values that are not in this list get mapped to the neutral language word-breaker and -stemmer, which uses white spaces to delimit words.

Note The Unicode collation locale identifier setting is used against all data types eligible for full-text indexing (such as **char**, **nchar**, and so on). If you have the sort order of a **char**, **varchar**, or **text** type column set to a language setting different from the Unicode collation locale identifier language, the Unicode collation locale identifier is still used during full-text indexing and querying of the **char**, **varchar**, and **text** type columns.

Implementation of Full-text Search

With a full-text query, you can perform a linguistic search of character data in tables enabled for full-text search. A linguistic search operates on words and phrases, unlike the LIKE predicate, which is used to search character patterns. Also, the Full-Text Search feature can weigh query terms and report how well a match scored or ranked against the original search term.

Implementing a full-text search in a given database involves these tasks:

- 1. Identify the tables and columns that are to be registered for full-text search.
- 2. Index the data in the registered columns and populate full-text indexes with the nonextraneous words.
- 3. Issue queries against the registered columns for populated full-text indexes.
- 4. Ensure that subsequent changes to the data in registered columns get propagated to the index, thus keeping the full-text index synchronized with the data.

Tasks 1, 2, and 4 are accomplished using graphical tools and wizards, available through SQL Server Enterprise Manager or built-in procedures. For more information about administering full-text indexes, see <u>Maintaining Full-Text</u> <u>Indexes</u>.

Note SQL Server does not support full-text search over linked servers.

Information about issuing queries against registered columns for populated full-text indexes (task 3) is the primary subject of the full-text topics in Accessing and Changing Data.

Unlike standard relational database indexes, full-text indexes are not instantly modified when values in full-text registered columns are updated, when rows are added to full-text registered tables, or when rows are deleted from full-text

registered tables. Rather, full-text indexes are repopulated asynchronously because:

- It typically takes significantly more time to update a full-text index than a standard index.
- Full-text searches are usually less precise than standard searches in that the search result is a set of rows that contain the word or phrase being searched no matter where they appear in the character stream. For example, using a standard index, a search returns a precise character pattern or number that exactly matched the original query. For full-text search, you can retrieve close approximations of the data, such as the plural forms of a noun, the various forms that a verb may take, or the uppercase or lowercase forms of the original search condition.

Full-text indexes are used for a different purpose than regular indexes, which must be updated immediately when data in its associated table changes. Full-text indexes can be synchronized with its table data. Although full-text index population can take time, these updates need not be disruptive. They can be scheduled in the background using the SQL Server Agent job scheduler, the **sp_add_job** stored procedures, or the Full-text Indexing Wizard.

When you repopulate an index, after changes to data have been made, the unique key column values are passed to the index engine to identify those items that need to be reindexed.

Accessing and Changing Relational Data

Full-text Query Transact-SQL Components

Microsoft® SQL Server[™] provides these Transact-SQL components for full-text querying.

Transact-SQL predicates

These predicates can be used in any search condition (including a WHERE clause) of a SELECT statement.

- CONTAINS
- FREETEXT

Transact-SQL rowset-valued functions

These functions can be used in the FROM clause of a SELECT statement.

- CONTAINSTABLE
- FREETEXTTABLE

Transact-SQL full-text properties

These Transact-SQL functions return information about the full-text properties of database objects.

SERVERPROPERTY

• **IsFullTextInstalled** indicates that the full-text component is installed with the current instance of SQL Server. This property is the counterpart of the FULLTEXTSERVICEPROPERTY function property with the same name.

DATABASEPROPERTYEX and DATABASEPROPERTY

• **IsFullTextEnabled** indicates whether a database has been enabled for

full-text indexing.

COLUMNPROPERTY

• **IsFullTextIndexed** indicates whether a column has been enabled for full-text indexing.

OBJECTPROPERTY

- **TableFullTextBackgroundUpdateIndexOn** indicates whether a table has full-text background update indexing.
- **TableFullTextCatalogId** provides the full-text catalog ID in which the full-text index data for the table resides.
- **TableFullTextChangeTrackingOn** indicates whether full-text change-tracking is enabled on the table.
- **TableFullTextKeyColumn** provides the column ID of the full-text unique key column.
- **TableFullTextPopulateStatus** indicate the population status of a full-text table.
- **TableHasActiveFulltextIndex** indicates whether a table has an active full-text index.

INDEXPROPERTY

• **IsFulltextKey** indicates whether the index is the full-text key for a table.

Transact-SQL has functions that specifically return full-text properties.

FULLTEXTCATALOGPROPERTY returns information about full-text catalog properties: **PopulateStatus**, **ItemCount**, **IndexSize**, **UniqueKeyCount**, **LogSize**, and **PopulateCompletionAge**. For more information, see

FULLTEXTCATALOGPROPERTY.

FULLTEXTSERVICEPROPERTY returns information about the full-text service-level properties: **ResourceUsage**, **ConnectTimeout**, and **IsFulltextInstalled**. **IsFulltextInstalled** returns the same information as the SERVERPROPERTY property of the same name. For more information, see <u>FULLTEXTSERVICEPROPERTY</u>.

Transact-SQL full-text system stored procedures

These stored procedures can be used in conjunction with writing a query. For example, you can use them to find the names of the full-text indexed columns for a table and the column ID of a full-text unique key column before specifying a query.

- **sp_fulltext_database** is a stored procedure that enables or removes full-text indexing from the current database.
- **sp_fulltext_catalog**, **sp_fulltext_table**, and **sp_fulltext_column** are stored procedures used in defining full-text indexes and initiating full-text index population.
- **sp_help_fulltext_catalogs**, **sp_help_fulltext_tables**, **sp_help_fulltext_columns**, and a variation of these stored procedures are used to query the full-text index meta data defined by the system stored procedures identified earlier.

Note The full-text system stored procedures cannot be used in a transaction.

Accessing and Changing Relational Data

Using the CONTAINS Predicate

You can use the CONTAINS predicate to search a database for a specific phrase. Of course, such a query can be written using the LIKE predicate. However, many forms of CONTAINS provide far more text query capabilities than can be obtained with LIKE. Additionally, unlike using the LIKE predicate, a CONTAINS search is always case insensitive.

Note The full-text search queries behave in a case-insensitive manner for those languages (mostly Latin-based) for which case sensitivity is meaningful. However, in Japanese, there are multiple phonetic orthographies in which the concept of orthographic normalization is akin to case insensitivity (for example, kana = insensitivity). This type of orthographic normalization is not supported.

Assume that you want to search within the **Northwind** database to find the phrase "bean curd". If you use the CONTAINS predicate, this is a fairly easy query.

USE Northwind GO SELECT Description FROM Categories WHERE Description LIKE '%bean curd%' GO

Or, using CONTAINS:

USE Northwind GO SELECT Description FROM Categories WHERE CONTAINS(Description, ' "bean curd" ') GO

The CONTAINS predicate uses functional notation in which the first parameter is the name of the column being searched and the second parameter is a full-text search condition. The search condition, in this case "bean curd", can be quite complex and is made up of one or more terms, which are described later.

The CONTAINS predicate supports complex syntax to search character-based columns for:

• One or more specific words and/or phrases (simple term).

A word is one or more characters without spaces or punctuation. A valid phrase can consist of multiple words with spaces with or without punctuation between them. For example, croissant is a word, and café au lait is a phrase. Words and phrases such as these are called simple terms.

• Inflectional form of a specific word (generation term).

For example, search for the inflectional form of the word drive. If various rows in the table include the words drive, drives, drove, driving, and driven, all would be in the result set because each of these can be inflectionally generated from the word drive.

• A word or a phrase where the words begin with specified text (prefix term).

In case of a phrase, each word within the phrase is considered to be a prefix. For example, the term auto tran* matches automatic transmission and automobile transducer.

• Words or phrases using weighted values (weighted term).

For example, you want to find a word that has a higher designated weighting than another word. It returns ranked query results.

• A word or phrase close to another word or phrase (proximity term).

For example, you want to find the rows in which the word ice is near the word hockey or in which the phrase ice skating is near the phrase ice hockey.

A CONTAINS predicate can combine several of these terms by using AND and OR, for example, to find all rows with latte and New York-style bagel in the same full-text enabled database column. Furthermore, terms can be negated by

the use of AND NOT, for example bagel and not cream cheese.

When you use CONTAINS, remember SQL Server discards noise words from the search criteria. Noise words are those words such as a, and, is, or the, which can occur frequently but do not really help when searching for specific text.

Searching for Specific Words or Phrases (Simple Term)

In Latin-based and other single-byte languages, a group of characters is typically interpreted as a word if it is framed by spaces or punctuation, and a phrase if it consists of multiple words with spaces, and with (or without) punctuation between them. For example, in the English language, a word such as clock or calendar consists of one or more characters without spaces or punctuation. In most languages, a phrase consists of multiple words with spaces, and with (or without) punctuation between them, such as cheese, crackers, and apple juice.

Asian languages are different in that an Asian language character can also be a word, and a phrase is a group of words that do not necessarily have to have spaces or punctuation between them.

The following query searches for the word business in the **notes** column of the **titles** table.

USE pubs GO SELECT title_id, title, notes FROM titles WHERE CONTAINS(notes, 'business') GO

This query searches for the phrase "common business applications" in the **notes** column of the **titles** table.

```
USE pubs
GO
SELECT title_id, title, notes
FROM titles
WHERE CONTAINS(notes, ' "common business applications" ')
GO
```

A CONTAINS predicate, can only be used with tables that have columns enabled for full-text querying. For more information about enabling one or more columns for full-text querying, see <u>sp_fulltext_table</u> and <u>sp_fulltext_column</u> and <u>Full-text Indexes</u>.

One or more table columns can be enabled for full-text querying. A given full-text predicate against that table can either access a single, enabled column or all of the enabled columns in a table. Assuming that both the **title** and **notes** columns in the **titles** table in the **pubs** database are full-text enabled, then the following query returns the title ID, title, and price for all rows in which the phrase "French gourmet" is present in either of the full-text enabled columns:

```
USE pubs
GO
SELECT title_id, title, price
FROM titles
WHERE CONTAINS( *, ' "French gourmet" ' )
GO
```

The asterisk (*), used in place of a column name, indicates all full-text enabled columns for the table.

Combining Full-text Search Operators Using AND, OR, and AND NOT

You can use parentheses and the Boolean operators (AND, AND NOT, and OR) between search conditions in a CONTAINS predicate. Assume one or more rows in the **titles** table contains information about favorite recipes and gourmet recipes. To retrieve rows that contain text for either type of recipe, use an OR between the "favorite recipes" and "gourmet recipes" phrases.

```
USE pubs
GO
SELECT title, notes
FROM titles
WHERE CONTAINS( notes, ' "favorite recipes" OR "gourmet recipes'
GO
```

This example searches for all rows in the **titles** table in which the **title** contains cooking, but neither computers nor computer.

```
USE pubs
GO
SELECT title_id, title, ytd_sales
FROM titles
WHERE CONTAINS( title, ' cooking AND NOT ("computer*" )' )
GO
```

This example obtains a list of product category descriptions in which the description mentions both words beers and ales.

```
USE Northwind
GO
SELECT CategoryName, Description
FROM Categories
WHERE CONTAINS( Description, ' beers AND ales ' )
```

GO

Phrases and predicates can be combined to search for combinations of words and phrases. For example, you can search for all rows that contain either ice skating or hockey but not references to the Olympics. The WHERE clause for using the CONTAINS predicate looks like this:

WHERE CONTAINS (*, '("ice skating" or hockey) AND NOT olympi

Searching for Multiple Forms of Words or Phrases (Prefix Term)

You can search columns for text that begin with a specified word or phrase. The specified text used for the search is called a prefix term.

When the prefix term is a word, all entries in the column that begin with the word will be returned. For example, to search for all rows that contain the word **ice**, as in **ice**, **ice** cream, or **ice**-shaved drinks, the query looks like this:

```
USE Northwind
GO
SELECT Description, CategoryName
FROM Categories
WHERE CONTAINS (Description, ' "ice*" ' )
GO
```

All text that matches the text specified before the asterisk (*) is returned. If the text and asterisk are not delimited by double quotation marks, as in CONTAINS (DESCRIPTION, 'ice*'), full-text search considers the asterisk as a character and will search for exact matches to ice*.

When the prefix term is a phrase, each word making up the phrase is considered a separate prefix term. All rows that have words beginning with the prefix terms will be returned. For example, the prefix term "light bread*" will find rows with text of either "light breaded", "lightly breaded", or "light bread."

```
USE Northwind
GO
SELECT Description, CategoryName
FROM Categories
WHERE CONTAINS (Description, ' "light bread*" ' )
GO
```

Searching for Any Form of a Specific Word (Generation Term)

You can search for all the different tenses of a verb or both the singular and plural forms of a noun. For example, this query searches for any form of dry (dry, dried, drying, and so on) in the **Description** column of the **Categories** table.

USE Northwind GO SELECT Description, CategoryName FROM Categories WHERE CONTAINS (Description, 'FORMSOF(INFLECTIONAL, ''d GO

Note that a single term cannot be used to match both nouns and verbs in the same query.

Searching for Words or Phrases Using Weighted Values (Weighted Term)

You can search for words or phrases and specify a weighting value. Weight, a number from 0.0 through 1.0, indicates the degree of importance for each word and phrase within a set of words and phrases. A weight value of 0.0 is the lowest value available, and a weight value of 1.0 is the highest available value. For example, this query searches for all customer addresses, using weight values, in which any text beginning with the string "des" is near either Rue or Bouchers. Microsoft® SQL Server™ gives a higher rank to those rows with more of the words specified. Therefore, SQL Server gives a higher rank to a row with des Rue Bouchers than to a row with des Rue.

```
USE Northwind
GO
SELECT CompanyName, ContactName, Address
FROM Customers
WHERE CONTAINS(Address, 'ISABOUT ("*des*",
Rue WEIGHT(0.5),
Bouchers WEIGHT(0.9)
)')
GO
```

A weighted term can be used in conjunction with any of the other four types of terms.

Searching for Words or Phrases Close to Another Word or Phrase (Proximity Term)

You can search for words or phrases in close proximity to another word or phrase. In addition, you can specify two words or phrases in any order and get the same result. This example searches for the word **user** close to the word **computers**.

```
USE pubs
GO
SELECT title, notes
FROM titles
WHERE CONTAINS (notes, 'user NEAR computers')
GO
```

However, you can also reverse the words in the WHERE clause to get the same result:

WHERE CONTAINS (notes, 'computers NEAR user')

You can specify the tilde (~) in place of the NEAR keyword in the earlier queries, and get the same results:

WHERE CONTAINS (notes, 'computers ~ user')

More than two words or phase can be specified in the search conditions. For example, it is possible to say:

WHERE CONTAINS (notes, ' hardware ~ softward ~ computer ')

This means that **hardware** should be in close proximity to **software**, and **software** should be in close proximity to **computer**.

In addition, matching the prefix of a word can be combined with searching for a word or phrase in close proximity to another word or phrase. This example searches for all descriptions in which the description has **sauces** in close

proximity to any form of **mix**, such as mixing, or mixed.

WHERE CONTAINS(Description, ' sauces ~ "mix*" ')

To find **wheat bread mix** and also **wheatberry bread mix**, you could use this type of search:

WHERE CONTAINS(Description, ' "wheat*" ~ "bread mix" ')

See Also

CONTAINS

WHERE

Accessing and Changing Relational Data

Using the FREETEXT Predicate

With a FREETEXT predicate, you can enter any set of words or phrases, or even a complete sentence. The full-text query engine examines this text, identifies all the significant words and noun phrases, and internally constructs a query with those terms. This example uses a FREETEXT predicate against a column named **description**.

FREETEXT (description, ' "The Fulton County Grand Jury said Friday

The search engine identifies words and noun phrases such as the following:

Words

Fulton, county, grand, jury, Friday, investigation, Atlanta, recent, primary, election, produce, evidence, irregularities

Phrases

Fulton county grand jury, primary election, grand jury, Atlanta's recent primary election

The words and phrases in the FREETEXT string (and their inflectionally generated variations) are internally combined into a query, weighted for proper ranking, and then the actual search is performed.

See Also

FREETEXT

Accessing and Changing Relational Data

Using Full-text Predicates to Query image Columns

CONTAINS and FREETEXT predicates may be used to search indexed **image** columns.

Many document types can be stored in a single **image** column. Microsoft® SQL Server[™] supports certain document types and provides a filter for these types. This release provides filters for Office documents, text files, and HTML files.

When an **image** column participates in a full-text index, the full-text service looks at the extensions of the documents in the **image** column and applies a corresponding filter to interpret the binary data and extract the textual information needed for indexing and querying.

Thus, when you set up full-text indexing on an **image** column in a table, you must create a separate column to hold information about the document. 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 full-text service will assume the document is a text file.

- In the **Full-Text Indexing Wizard**, if you select an image column for indexing, you must also specify a **Binding column** to hold the document type.
- The **sp_fulltext_column** stored procedure also accepts an argument for the column to contain the document types.
- The **sp_help_fulltext_columns** stored procedure also returns column name and column id of the document type column.

For more information about setting up full-text indexes and searches on **image** columns, see <u>Filtering Supported File Types</u>.

Once indexed, the **image** column can be queried like any other column in a table, using the predicates CONTAINS and FREETEXT.

Accessing and Changing Relational Data

Combining Full-text Predicates with Other Transact-SQL Predicates

The CONTAINS and FREETEXT predicates can be combined with any of the other Transact-SQL predicates, such as LIKE and BETWEEN; they can also be used in a subquery. This example searches for descriptions in which the category is not Seafood, and in which the description contains the word sauces and the word seasonings.

```
USE Northwind
GO
SELECT Description
FROM Categories
WHERE CategoryName <> 'Seafood' AND
CONTAINS(Description, ' sauces AND seasonings ')
GO
```

The following query uses CONTAINS within a subquery. Using the **pubs** database, the query obtains the title value of all the books in the **titles** table for the publisher that is located close to the flying saucer in Moonbeam, Ontario. (This information about the publisher is in the **pr_info column** in the **pub_info** table, and there is only one such publisher.)

```
USE pubs
GO
-- Add some interesting rows to some tables.
INSERT INTO publishers
VALUES ('9970',
'Penumbra Press',
'Moonbeam',
'ON',
'Canada')
INSERT INTO pub_info (pub_id, pr_info)
VALUES ('9970',
```

'Penumbra press is located in the small village of Moonbeam. M INSERT INTO titles

VALUES ('FP0001',

'Games of the World',

'crafts', '9970', 9.85, 0.00, 20, 213, 'A crafts book! A sports book! A history book! The fun and exc '1977/09/15')

GO

-- Given the full-text catalog for these tables is pubs_ft_ctlg,

-- repopulate it so new rows are included in the full-text indexes.

```
sp_fulltext_catalog 'pubs_ft_ctlg', 'start_full'
```

WAITFOR DELAY '00:00:30' -- Wait 30 seconds for population. GO

-- Issue the query.

SELECT T.title, P.pub_name

FROM publishers P,

titles T

```
WHERE P.pub_id = T.pub_id
```

```
AND P.pub_id = (SELECT pub_id
```

FROM pub_info

```
WHERE CONTAINS (pr_info,
```

' moonbeam AND

ontario AND

"flying saucer" '))

GO

See Also

CONTAINS FREETEXT WHERE Accessing and Changing Relational Data

Using the CONTAINSTABLE and FREETEXTTABLE Rowset-valued Functions

The CONTAINSTABLE and FREETEXTTABLE functions are used to specify full-text queries that return relevance rankings for each row. These functions are very similar but used differently from the full-text predicates, CONTAINS and FREETEXT.

Differentiating the full-text predicates from the functions

Although both the full-text predicates and the full-text rowset-valued functions are used for full-text queries, and the Transact-SQL statement used to specify the full-text search condition is the same in both the predicates and the functions, there are major differences in the way that these are used:

• CONTAINS and FREETEXT both return a TRUE or FALSE value, so they are typically specified in the WHERE clause of a SELECT statement.

CONTAINSTABLE and FREETEXTTABLE both return a table of zero, one, or more rows, so they must always be specified in the FROM clause.

• CONTAINS and FREETEXT can only be used to specify selection criteria, which Microsoft® SQL Server[™] uses to determine the membership of the result set.

CONTAINSTABLE and FREETEXTTABLE are also used to specify selection criteria. The table returned has a column named KEY that contains full-text key values. Each full-text registered table has a column whose values are guaranteed to be unique. The values returned in the KEY column of CONTAINSTABLE or FREETEXTTABLE are the unique values, from the full-text registered table, of the rows that match the selection criteria specified in the full-text search condition.

Furthermore, the table produced by CONTAINSTABLE and FREETEXTTABLE has a column named RANK, which contains values from 0 through 1000. These values are used to rank the rows returned

according to how well they met the selection criteria.

Queries that use the CONTAINSTABLE and FREETEXTTABLE functions are more complex than those that use the CONTAINS and FREETEXT predicates because qualifying rows returned by the functions must be explicitly joined with the rows in the original SQL Server table.

This example returns the description and category name of all food categories for which the **Description** column contains the words "sweet and savory" near either the word "sauces" or the word "candies." All rows with a category name "Seafood" are disregarded. Only rows with a rank value of 2 or higher are returned.

USE Northwind GO SELECT FT_TBL.Description, FT_TBL.CategoryName, KEY_TBL.RANK FROM Categories AS FT_TBL INNER JOIN CONTAINSTABLE (Categories, Description, '("sweet and savory" NEAR sauces) OR ("sweet and savory" NEAR candies)') AS KEY_TBL ON FT_TBL.CategoryID = KEY_TBL.[KEY] WHERE KEY_TBL.RANK > 2 AND FT_TBL.CategoryName <> 'Seafood' ORDER BY KEY_TBL.RANK DESC

This example returns the description and category name of the top 10 food categories where the **Description** column contains the words "sweet and savory" near either the word "sauces" or the word "candies."

SELECT FT_TBL.Description, FT_TBL.CategoryName, KEY_TBL.RANK FROM Categories AS FT_TBL INNER JOIN

```
CONTAINSTABLE (Categories, Description,

'("sweet and savory" NEAR sauces) OR

("sweet and savory" NEAR candies)'

, 10

) AS KEY_TBL

ON FT_TBL.CategoryID = KEY_TBL.[KEY]
```

Comparison between CONTAINSTABLE and CONTAINS

The CONTAINSTABLE function and the CONTAINS predicate use similar search conditions.

However, in CONTAINSTABLE you specify the table that will be full-text searched, the column (or all the columns) in the table to be searched, and the search condition. A fourth parameter, an optional one, makes it possible for the user to indicate that only the highest specified number of matches be returned. For more information, see the Limiting Result Sets section.

CONTAINSTABLE returns a table that includes a column named **RANK**. This RANK column contains a value for each row that indicates how well a row matched the selection criteria.

This query specifies using CONTAINSTABLE to return a rank value for each row.

```
USE Northwind
GO
SELECT K.RANK, CompanyName, ContactName, Address
FROM Customers AS C
INNER JOIN
CONTAINSTABLE(Customers,Address, 'ISABOUT ("des*",
Rue WEIGHT(0.5),
Bouchers WEIGHT(0.9)
)
```

) AS K ON C.CustomerID = K.[KEY] Here is the result set:

RANK CompanyName ContactName address

---- -----

- 123 Bon app' Laurence Lebihan 12, rue des Bouchers
- 65 Du monde entier Janine Labrune 67, rue des Cinquante Otage
- 15 France restauration Carine Schmitt 54, rue Royale
- 15 La maison d'Asie Annette Roulet 1 rue Alsace-Lorraine
- 15 Maison Dewey Catherine Dewey Rue Joseph-Bens 532
- 15 Mère Paillarde Jean Fresnière 43 rue St. Laurent
- 15 Spécialités du monde Dominique Perrier 25, rue Lauriston
- 15 Vins et alcools Paul Henriot 59 rue de l'Abbaye Chevalier
- 15 Victuailles en stock Mary Saveley 2, rue du Commerce

Comparison between FREETEXTTABLE and FREETEXT

The following query extends a FREETEXTTABLE query to return the highest ranked rows first and to add the ranking of each row to the select list. To specify the query, you must know that **CategoryID** is the unique key column for the **Categories** table.

```
USE Northwind
GO
SELECT KEY_TBL.RANK, FT_TBL.Description
FROM Categories AS FT_TBL
INNER JOIN
FREETEXTTABLE(Categories, Description,
'How can I make my own beers and ales?') AS KEY_TBL
ON FT_TBL.CategoryID = KEY_TBL.[KEY]
ORDER BY KEY_TBL.RANK DESC
GO
```

The only difference in the syntax of FREETEXTTABLE and FREETEXT is the insertion of the table name as the first parameter.

Here is an extension of the same query that only returns rows with a rank value of 10 or greater:

```
USE Northwind
GO
SELECT KEY_TBL.RANK, FT_TBL.Description
FROM Categories FT_TBL
INNER JOIN
FREETEXTTABLE (Categories, Description,
'How can I make my own beers and ales?') AS KEY_TBL
ON FT_TBL.CategoryID = KEY_TBL.[KEY]
WHERE KEY_TBL.RANK >= 10
ORDER BY KEY_TBL.RANK DESC
GO
```

Identifying the Unique Key Column Name

Queries that use rowset-valued functions are complicated because it is necessary to know the name of the unique key column. Each full-text enabled table has the **TableFulltextKeyColumn** property that contains the column ID number of the column that has been selected for enforcing unique rows for the table. This example shows how the name of the key column can be obtained and used programmatically.

```
USE Northwind
GO
DECLARE @key_column sysname
SET @key_column = Col_Name(Object_Id('Categories'),
ObjectProperty(Object_id('Categories'),
'TableFulltextKeyColumn')
)
print @key_column
EXECUTE ('SELECT Description, KEY_TBL.RANK
FROM Categories FT_TBL
INNER JOIN
```

FreetextTable (Categories, Description,

"How can I make my own beers and ales?") AS KEY_TBI ON FT_TBL.'

```
@key_column
+
' = KEY_TBL.[KEY]
WHERE KEY_TBL.RANK >= 10
ORDER BY KEY_TBL.RANK DESC
')
GO
```

+

You can avoid the complexity of using CONTAINSTABLE and FREETEXTTABLE by writing stored procedures that accept a few facts about the query and then create and execute the appropriate query. A simplified procedure that submits a FREETEXTTABLE query follows. The table shows the procedure parameters (all input).

Parameter	Required	Description
@additional_predicates	Optional	If there are any, these get added with
		AND after the FREETEXT
		predicate. KEY_TBL.RANK can
		be used within expressions.
@freetext_column	Yes	
@freetext_search	Yes	Search condition.
@from_table	Yes	
@order_by_list	Optional	KEY_TBL.RANK can be one of
		the columns specified.
@select_list	Yes	KEY_TBL.RANK can be one of
		the columns specified.

The code for the procedure is:

CREATE PROCEDURE freetext_rank_proc @select_list nvarchar(1000), @from_table nvarchar(517),

```
@freetext column
                     sysname,
   @freetext_search
                       nvarchar(1000),
   @additional_predicates nvarchar(500)
                                       =",
                       nvarchar(500)
   @order by list
AS
BEGIN
 DECLARE @table id
                            integer,
      @unique_key_col_name sysname,
                        nvarchar(510),
     @add pred var
      @order by var
                        nvarchar(510)
 -- Get the name of the unique key column for this table.
 SET @table_id = Object_Id(@from_table)
 SET @unique_key_col_name =
```

```
Col_Name( @table_id,
```

```
ObjectProperty(@table_id, 'TableFullTextKeyColumn') )
```

```
-- If there is an additional_predicate, put AND() around it. IF @additional_predicates <> "
```

```
SET @add_pred_var = 'AND (' + @additional_predicates + ')'
ELSE
```

SET @add_pred_var = "

```
-- Insert ORDER BY, if needed.
```

```
IF @order_by_list <> "
```

```
SET @order_by_var = 'ORDER BY ' + @order_by_var
ELSE
```

SET @order_by_var = "

```
-- Execute the SELECT statement.
EXECUTE ( 'SELECT'
+ @select_list
+ ' FROM '
```

```
+ @from_table
       + ' AS FT_TBL, FreetextTable('
       + @from table
       + '.'
       + @freetext column
       + '.'''
       + @freetext_search
       + "") AS KEY TBL '
       + 'WHERE FT TBL.'
       + @unique_key_col_name
       + ' = KEY TBL.[KEY] '
       + @add pred var
       + ' '
       + @order_by_var
      )
END
This procedure can be used to submit the query:
USE Northwind
GO
EXECUTE freetext_rank_proc
  'Description, KEY_TBL.RANK', -- Select list
  'Categories',
                                -- From
  'Description',
                                -- Column
  'How can I make my own beers and ales?', -- Freetext search
  'KEY TBL.RANK \geq 10',
                                         -- Additional predicate
```

```
-- Order by
```

GO

Limiting Result Sets

'KEY TBL.RANK DESC'

In many full-text queries, the number of items matching the search condition is very large. To prevent queries from returning too many matches, use the optional argument, *top_n_by_rank*, in CONTAINSTABLE and FREETEXTTABLE to

specify the number of matches according to rank you want returned.

With this information, Microsoft® SQL Server[™] orders the matches by rank and returns only up to the specified number. This choice can result in a dramatic increase in performance. For example, a query that would normally return 100,000 rows from a table of one million rows will be processed more quickly if only the top 100 rows are requested.

If you want only the top 3 matches returned on an earlier example using CONTAINSTABLE, here's how the query looks:

```
USE Northwind
GO
SELECT K.RANK, CompanyName, ContactName, Address
FROM Customers AS C
INNER JOIN
CONTAINSTABLE(Customers,Address, 'ISABOUT ("des*",
Rue WEIGHT(0.5),
Bouchers WEIGHT(0.9))', 3) AS K
ON C.CustomerID = K.[KEY]
```

Here is the result set:

RANK CompanyName ContactName address

123 Bon app' Laurence Lebihan 12, rue des Bouchers

65 Du monde entier Janine Labrune 67, rue des Cinquante Otage

15 France restauration Carine Schmitt 54, rue Royale

See Also

CONTAINSTABLE

FREETEXTTABLE

Using Transact-SQL Functions to Obtain Full-text Property Values

Several Transact-SQL functions can be used to obtain the values of full-text properties. For example, the **TableFulltextKeyColumn** property can be used to programmatically obtain the identity of a unique key column for a table. Also, the **IsFullTextEnabled** property can be used to check whether full-text querying is enabled for a database. This example checks to see whether full-text querying is enabled for the **Northwind** database.

USE Northwind GO SELECT DATABASEPROPERTY('Northwind', 'IsFullTextEnabled')

If a value of 1 is returned, the **Northwind** database has been enabled for full-text querying. A value of 0 indicates that the **Northwind** database has not been enabled for full-text querying.

The table contains a complete list of properties. It should be noted that many of these properties are useful only for full-text administration.

Function	Property
COLUMNPROPERTY	IsFulltextIndexed
DATABASEPROPERTY	IsFulltextEnabled
DATABASEPROPERTYEX	IsFulltextEnabled
INDEXPROPERTY	IsFulltextKey
OBJECTPROPERTY	TableFulltextBackgroundUpdateIndexOn
	TableFulltextCatalogId
	TableFulltextChangeTrackingOn
	TableFulltextKeyColumn
	TableFulltextPopulateStatus
	TableHasActiveFulltextIndex
FULLTEXTCATALOGPROPERTY	PopulateStatus
	ItemCount

	IndexSize
	UniqueKeyCount
	LogSize
	PopulateCompletionAge
FULLTEXTSERVICEPROPERTY	ResourceUsage
	ConnectTimeout
	IsFullTextInstalled

See Also

COLUMNPROPERTY DATABASEPROPERTY DATABASEPROPERTYEX FULLTEXTCATALOGPROPERTY FULLTEXTSERVICEPROPERTY INDEXPROPERTY OBJECTPROPERTY

Example of Combining Full-text Administration and Full-text Query

Full-text indexes can be administered using either SQL Server Enterprise Manager or stored procedures. Sometimes it is convenient to combine full-text administrative stored procedures in the same script as the queries. The following example script combines these tasks:

- Create and populate a table.
- Enable the **pubs** database for full-text searching.
- Create a full-text catalog.
- Register the new table and certain columns in it for full-text search.
- Populate the new full-text catalog with full-text index information from the new table.
- Execute a full-text query against the new table.

```
USE pubs

--- Create and populate a table.

IF EXISTS (SELECT TABLE_NAME FROM INFORMATION_SCH

WHERE TABLE_NAME = 'FulltextTest')

DROP TABLE FulltextTest

GO

CREATE TABLE FulltextTest

( article_id int IDENTITY(100,1)

CONSTRAINT PK_title_id PRIMARY KEY,

article_title nvarchar(200)

)
```

GO

INSERT FulltextTest (article_title) VALUES (N'Steven Buchanan has INSERT FulltextTest (article_title) VALUES (N'Elvis Stoiko: The best INSERT FulltextTest (article_title) VALUES (N'Steven Buchanan On] INSERT FulltextTest (article_title) VALUES (N'Last night, Steven Buc INSERT FulltextTest (article_title) VALUES (N'Ice-skating brings out GO

-- Enable full-text searching in the database. EXEC sp_fulltext_database 'enable' GO

-- Create a new full-text catalog. EXEC sp_fulltext_catalog 'StevenBCatalog', 'create'

GO

-- Register the new table and column within it for full-text querying, -- then activate the table.

GO

-- Start full population of the full-text catalog. Note that it is

-- asynchronous, so delay must be built in if populating a

-- large index.

GO

-- Execute a full-text query against the new table. SELECT article_title FROM FulltextTest WHERE CONTAINS(article_title, ' "Steven Buchanan" AND "ice skat

Here is the result set:

article_title

Steven Buchanan has always enjoyed ice skating.

Last night, Steven Buchanan skated on the ice!! Skating fans cheer! Steven Buchanan On Ice: Skating Reaches Tops in Public Opinion Poll Ice-skating brings out the best in Steven. Buchanan exults in first victo (4 row(s) affected)

Full-text Querying of File Data

Microsoft® SQL Server[™] supports textual queries against data residing in the file system, as well as SQL Server data. Products and features that support this capability include SQL Server distributed queries, Microsoft Internet Information Services 4.0, and Microsoft Indexing Service version 2.0.

A large portion of digitally stored information is still in the form of unstructured data, primarily text, stored in the file system. This information is often related to data within the database, which requires that queries be run against both sources. However, it is often inappropriate to import this data from the file system. Distributed queries coupled with extensions to the SQL language make it possible to write such queries without the data. This is known as file content search.

There are two major types of textual queries:

Property search

Applies filters to documents to extract properties, such as author, subject, type, word count, printed page count, and time last written, and then issues queries against those properties.

Full-text search

Creates indexes of all nonnoise words in the documents, and then uses these indexes to support linguistic searches and proximity searches.

For example, the following query selects the names, sizes, and authors of all Microsoft Word files on the D drive that contains the phrase "SQL Server" in close proximity to text. It then joins this with the **writers** table to obtain the author's citizenship.

```
SELECT Q.FileName, Q.Size, Q.DocAuthor, W.Citizenship
FROM OpenQuery(MyLinkedServer,
'SELECT FileName, Size, DocAuthor
FROM SCOPE(" "D:\" ")
WHERE CONTAINS(""SQL Server"
NEAR() text")
```

```
AND FileName LIKE "%.doc%" '
) > 0 AS Q,
writers AS W
WHERE Q.DocAuthor = W.writer_name
```

File content search relies upon the Microsoft OLE DB Provider for Microsoft Indexing Service 2.0. It also relies upon Indexing Service for the support of the underlying filters and full-text indexes.

The OLE DB Provider for Indexing Service 2.0 supports the ability to support SQL queries against data in the file system independent of SQL Server. The core extensions to the SQL language to support such queries are the same in both products. However, there are certain extensions that are not relevant to Indexing Service 2.0. For more information about the syntax of full-text queries against data in SQL Server, see <u>Full-text Querying SQL Server Data</u>.

See Also

CONTAINS WHERE FREETEXT

Using Microsoft Internet Information Services and Indexing Service for File Content Searches

Microsoft® Internet Information Services (IIS) 4.0 and Indexing Service version 2.0 (both part of Microsoft Windows NT® 4.0 Option Pack) combine to provide property filtering and searching as well as full-text indexing and searching of file data. Text query support against file data has an advantage over text query support against database data because, in Microsoft SQL Server[™] the latter is limited to queries against character-based columns. These file content search capabilities are independent of SQL Server, and support SQL-based queries within ADO (ActiveX® Data Objects). The SQL used in ADO queries is consistent with the SQL extensions explained here.

Format Filters

Indexing Service provides filters for several popular file formats including Microsoft Word, Microsoft PowerPoint®, Microsoft Excel, and HTML. Filters are also available for plain-text. Filters can be written by customers and third-party vendors for other formats as well. One purpose of a filter is to provide support for nonplain-text documents. The other purpose is to capture property values both from the file content and about the files. Assuming that each file is a document, examples of properties include each document's title, the number of pages with notes in each PowerPoint document, the number of paragraphs in each document, the last date and time each file was accessed, and the physical path to each file. For a list of properties, see <u>Using File Properties for File</u> <u>Content Searches</u>. For a complete list, the Indexing Service documentation.

Phrase and Proximity Searches

Full-text indexes for file system searches are created by scanning the content of files. The process consists of keeping track of the significant words that are used and where they are located. For example, a full-text index may indicate that the word Canada is found at word number 227, word 473, and word number 1017 in a given file. This index structure supports an efficient search for all items containing indexed words, and advanced search operations such as phrase searches and proximity searches. An example of a phrase search is looking for

"white elephant", where white is followed by elephant. An example of a proximity search is looking for phrases in which big occurs near house. To prevent the full-text index from becoming bloated with words that do not help the search, noise-words, such as a, and, and the, are ignored.

Noise Words

Noise-word lists for many languages are provided and the set of supported languages is growing. The choice of a particular noise-word list is based on the language of the material which is file-format-dependent during the filtering process: Some files have the language setting by section (for example, by paragraph), whereas some specify the language setting as a property of the document. These noise-word lists should be sufficient for most normal operations, but can be modified for specific environments with a text editor. For more information, see the Indexing Service 2.0 documentation in the Windows NT 4.0 Option Pack.

Text-search Catalog

Indexing Service stores indexes and property values in a text-search catalog. By default, a text-search catalog named Web is created when Indexing Service is installed. A given text-search catalog references one or more IIS virtual directories (also known as virtual roots). A virtual directory references one or more physical directories and, optionally, other virtual directories. After a real file is linked to the text catalog through a virtual directory, Indexing Service is notified of the new files that must be indexed and begins the filtering and indexing of the properties and content associated with these files. Indexing Service is also notified of any subsequent changes to these files and will refilter and reindex the updated files.

Using Virtual Tables for File Content Queries

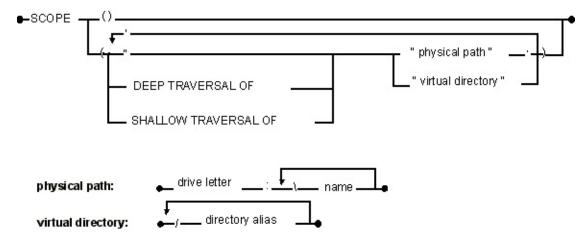
Every SQL query must have at least one defined table specified, which means the number and types of columns is either known in advance or specified as part of the query. A relational database usually contains a number of predefined tables and the meta data about the columns of these tables is stored in a schema.

The collection of files in a file system, however, does not generally have such a predefined structure. The properties of a file are perhaps similar to columns, but there is no deterministic set of properties for files. A file itself may be similar to a row, but files are usually not grouped into a homogeneous collection akin to the rows in a table. Thus, the table concept is unclear, SELECT * is meaningless, and both the rows and the columns are unbounded. Another way of looking at this is that a file system effectively has a universal schema consisting of every possible file property, both already known and as yet unknown.

Microsoft[®] Indexing Service solves this problem by providing the SCOPE function which defines the set of rows that makes up a virtual table and provides file properties that substitute for columns.

Using SCOPE Function for File System Queries

SCOPE is specified in the FROM clause of the Indexing Service query and is used to specify the set of files that makes up a virtual table. Here is the syntax.



The table shows the syntax elements and their descriptions.

Syntax element	Description
0	The virtual table consists of all the files registered in the text catalog and data source for the linked server specified in the OPENQUERY() function.
DEEP TRAVERSAL OF	The virtual table consists of all the files in the directory at the specified path or virtual directory as well as all the files in all subdirectories (to any level) are considered to be part of the virtual table. If neither DEEP nor SHALLOW is specified, DEEP is the default.
SHALLOW TRAVERSAL OF	The virtual table consists of only of the files in the top-level directory at the specified path or virtual directory are considered to be part of the virtual table.
physical_path	This is a path to a real directory. If a real directory is specified, the filtering and indexing is done as part of the processing of the query. This

	can be time-consuming.	
virtual_directory	This is the alias (or chains of aliases) assigned to	
	a virtual directory that has been registered in the	
	text catalog and data source for the linked server	
	specified in the OPENQUERY() function. In this	
	case, the filtering and indexing probably has been	
	done and, thus, the query is much faster than	
	when a <i>physical_path</i> is specified.	

Using File Properties for File Content Searches

Microsoft® Indexing Service filters and maintains in excess of 50 file properties. All of these can be specified in text file search queries. From the perspective of writing a SELECT statement, there are 3 types of properties:

- Those that can only be specified in a WHERE clause.
- Those that can be specified in a WHERE clause and, in addition, can be specified in the ORDER BY clause.
- Those that can be specified in a WHERE clause and, in addition, can be specified in a select list.

The table shows a list of the properties.

Property name	Data type	Description	Use in ORDER BY	Use in select list
Access	datetime	Most recent date and time that the file was accessed.	Yes	Yes
Characterization	nvarchar or ntext	Abstract of the contents of the file. Note that, in this version of Indexing Service this is usually the first paragraph or first section of a document. In a future version, it will be an actual summary.		Yes

Contents	nvarchar or ntext	Main contents of the file.		
Create	datetime	Date and time that the file was created.	Yes	Yes
Directory	nvarchar	Physical path to the file, not including the file name.	Yes	Yes
DocAuthor	nvarchar	Document author.	Yes	Yes
DocComments	nvarchar	Comments about the document.	Yes	Yes
DocLastAuthor	nvarchar	Most recent user that edited the document.	Yes	Yes
DocLastPrinted	datetime	Date and time that the document was last printed.	Yes	
DocPageCount	integer	Number of pages in the document.	Yes	
DocPartTitles	array of varchar	Names of document parts: Slide titles in Microsoft PowerPoint® Spreadsheets in Microsoft Excel Documents in Microsoft Word.		
DocSubject	nvarchar	Subject of the document.	Yes	Yes
DocTitle	nvarchar	Title of the document.	Yes	Yes
DocWordCount	integer	Number of words in the document.	Yes	-

FileIndex	decimal(19,0)	Unique identifier of the file.	Yes	Yes
FileName	nvarchar	Name of the file.	Yes	Yes
HitCount	integer	Number of words matching query.	Yes	Yes
Path	nvarchar	Full physical path to the file, including file name.	Yes	Yes
Rank	integer	Value from 0 to 1000 indicating how closely this row matches the selection criteria.	Yes	Yes
Size	decimal(19,0)	Size of file, in bytes.	Yes	Yes

Customers and third-party vendors can write filters to add to this set of properties. They can also add properties, for example, by adding their own tags to an HTML document. In addition, to permit the query and retrieval of such user-defined properties, the Transact-SQL extensions to Indexing Service include support for a SET statement that allows the specification of new property names and their associated types.

Sample Full-text Query Using File Content and Database Data

The first query returns the title and publication year of qualifying books that are represented by files in the virtual directory with the alias /pubs. To satisfy the query, a book must cost less than \$20.00 and text in the **Characterization** property must indicate that the book is about ice hockey. It is known that the year portion of the **Create** property is always the publication year of the book. The customer has defined the **BookCost** property (of type money), which filters out the cost of each book.

```
SELECT Q.DocTitle, DATEPART(YEAR, Q.Create)
FROM OPENQUERY(FileSystem,
'SELECT DocTitle, Create
FROM SCOPE(" "/pubs" ")
WHERE BookCost <= 20.00
AND CONTAINS( Characterization, " "ice hockey" " )
') AS Q
```

The table alias value Q has been assigned to the table returned by the OPENQUERY function. This alias is then used to qualify the items in the outer select list. Here, the SQL Server DATEPART() function is used to pass on only the year portion of the **create datetime** value.

This second query returns the same information as the previous one. The difference is that the price of a book is obtained from the **document_cost** column in the **BookCost** table in the database, rather than from a property in the file system. The primary key of the **BookCost** table is the combination of **document_author** and **document_title**.

```
SELECT Q.DocTitle, DATEPART(YEAR, Q.Create)
FROM OPENQUERY(FileSystem,
'SELECT DocTitle, Create, DocAuthor, DocTitle
FROM SCOPE(" "/pubs" ")
AND CONTAINS( Characterization, " "ice hockey" " )
```

```
') AS Q,
BookCost as B
WHERE Q.DocAuthor = B.document_author
AND Q.DocTitle = B.document_title
AND B.document_cost <= 20.00
```

The table returned by the OPENQUERY function is joined to the real BookCost table in the database, then rows with a suitable cost are filtered out for inclusion in the outer SELECT.

This last query also joins data from the file system and the database and, this time, data from both appears in the outer select list. Furthermore, the **Rank** property, which indicates how well the selected rows met the selection criteria, appears in the select list and is used to ensure that higher-ranking rows appear before lower-ranking rows in the outer SELECT. In this example, the wording on the plaques in the Hockey Hall of Fame is recorded on files. There is a file for each plaque, and the plaque number can be obtained with the **DocSubject** property. The **HockeyHall** table contains **PlaqueNo**, **PlayerName**, **StartYear**, and **LastYear** columns with the primary key being **PlaqueNo**. The query returns the **PlayerName** and **PlaqueNo** from the table and the Rank and DocComments properties from the file. Only players who might have played for the Canadian or U.S. teams in the early 1900s are returned.

SELECT HH.PlayerName, HH.PlaqueNo, Q.Rank, Q.DocComments FROM OPENQUERY(FileSystem,

'SELECT DocSubject, DocComments, Rank FROM SCOPE(" "/hall_of_fame" ") WHERE CONTAINS(Contents, " Canada OR "United State ') AS Q, HockeyHall as HH WHERE Q.DocSubject = HH.PlaqueNo AND HH.StartYear < 1915 AND HH.EndYear < 1899 ORDER BY Q.Rank DESC