CHAPTER 2

SQL Building Blocks and Server Settings

This chapter covers the building blocks available to the database designer and database user to create and modify database objects and data. The SQL Data Types will be covered along with data representation as literals (constants) and expressions. Also covered are methods to display and change server and database settings including using Enterprise Manager and the **sp_configure** and **sp_dboption** system stored procedures.

Throughout the book, the following terms will be used.

- *SQL-92* will refer to the 1992 ANSI/ISO SQL Standard, also called SQL2. *SQL-99* will refer to the 1999 ANSI/ISO SQL Standard,[1] also called SQL3 and ISO/IEC 9075.
- *SQL Server* will refer to Microsoft® SQL Server 2000 (version 8.0).
- *Books Online* will refer to Microsoft SQL Server 2000 Books Online.[2]

It is recommended that Books Online be used in conjunction with this book for coverage of some details not covered here and for different examples. Books Online is an excellent reference. This book, however, gathers related information

1. ANSI+ISO+IEC+9075-1-1999 through ANSI+ISO+IEC+9075-5-1999, American National Standards Institute.

2. Microsoft SQL Server 2000 Books Online, © 1988-2001 Microsoft Corporation. All Rights Reserved.

together and generally provides more examples per topic than does Books Online. Used together they provide quite strong reference material.

2.1 SQL SERVER INSTANCE

Each instance (or installation) of Microsoft SQL Server 2000 is composed of several databases, four system databases (**master**, **model**, **msdb** and **tempdb)** and as many user databases as specified by the database administrator. SQL Server 2000 supports installing more than one instance on the same host.

2.1.1 The SQL Server Database

Each database consists of a collection of system and user tables and related database objects such as views, indexes, stored procedures, triggers, etc. A database in a Microsoft SQL Server database maintains a list of its database objects in its `sysobjects` table in that database. Each object type and its `sysobjects` table symbol are listed in Table 2-1.

Table 2-1 Object Types

Object Type	Sysobjects Symbol
CHECK constraint	C
Default or DEFAULT constraint	D
FOREIGN KEY constraint	F
Scalar function	FN
Inlined table function	IF
PRIMARY KEY or UNIQUE constraint	K
Log	L
Stored procedure	P
Rule	R
Replication filter stored procedure	RF
System table	S
Table function	TF
Trigger	TR
User table	U

Table 2-1 Object Types (cont.)

Object Type	Sysobjects Symbol
View	V
Extended stored procedure	X

By default, two sample user databases, **Northwind** and **Pubs**, are installed with SQL Server. These databases will often be used in examples in this book.

Example: List all USER TABLE objects in the pubs database.

```
SQL SELECT * FROM pubs..sysobjects WHERE type = 'U';
```

Other database objects are recorded in a separate table such as **sysindexes** for indexes and **syscolumns** for details of table and view columns as well as stored procedure parameters.

2.1.1.1 System Stored Procedures

SQL Server provides some built-in system stored procedures to assist in determining information about databases and database objects and to assist in database administration. System stored procedures are covered as needed, and a complete listing starts on page 168. Two of the more useful are shown here: **sp_helpdb** and **sp_help**.

sp_helpdb This procedure reports information about a specified database or all databases.

Syntax

```
sp_helpdb [ [ @dbname= ] 'name' ]
```

Arguments

[@dbname=] 'name'

This is the name of the database for which to provide information. If a name is not specified, sp_helpdb reports on all databases in master.dbo.sysdatabases. (*dbo* stands for database owner, the predefined user name in each database that is able to perform all database operations. Any sysadmin server role member becomes **dbo** inside each database.)

Return Code Values

```
0 (success) or 1 (failure)
```

Example: Show information on database **pubs**.

```
SQL EXEC sp_helpdb pubs
```

Reports **db** size, owner, file locations and other information for the **pubs** database.

sp_help This procedure reports information about a database object, a user-defined data type or a system-supplied data type.

Syntax

```
sp_help [ [ @objname = ] name ]
```

Arguments

[@objname =] name

This is the name of any object in `sysobjects` or any user-defined data type in the `systypes` table. `name` has a default of `NULL`. (Database names are not acceptable.)

Return Code Values

```
0 (success) or 1 (failure)
```

Example: Show the structure of the titles table of the pubs database.

```
SQL
USE pubs -- must first move the context of SQL client to pubs database
go
```

```
SQL EXEC sp_help titles
```

lists all column names and data types, constraints, table owner, etc., of titles table.

2.2 DATA REPRESENTATION

2.2.1 General Data Type Information

Every data value belongs to some data type such as INTEGER, CHARACTER, etc. Microsoft SQL Server Transact-SQL has a number of native data types, which are described in the next section.

In addition to the built-in, native data types, Transact-SQL provides for the definition of user-defined types, covered later in this chapter.

2.2.2 Domain and NULL Value

2.2.2.1 Domain

The **domain** of a column is the set of legal data type values for that column as determined by the data type assigned to the column and applicable constraints.

A **domain value** is any value within the domain. For example, a column called Age may have a data type of INT and a CHECK constraint that values must be between 0 and 200. Then the domain is the set of all integers between 0 and 200 inclusive. In this case, 21 would be a valid domain value for the Age column.

2.2.2.2 NULL

NULL means ”no domain value assigned.” It indicates that the value is unknown. **NULL can also be read as “not available,” “not applicable” or “unknown.”** If you want to add an employee to the employees table who does not have a telephone number, NULL would be assigned to the field value in the database. Note that NULL is very different from a value of zero for INT or an empty string for VARCHAR, both of which are domain values.

A constraint of NOT NULL may be placed on any column that is not allowed to have a NULL value. This is true for a primary key, for example. Every row must have a valid domain value for each column specified as NOT NULL. The NULL value is represented by the keyword `NULL`.

The ANSI SQL-99 Standard says the following about `NULL`.[3]

> **null value:** A special value that is used to indicate the absence of any data value. Every data type includes a special value, called the *null valu*e, sometimes denoted by the keyword NULL. This value differs from other values in the following respects: — Since the null value is in every data type, the data type of the null value implied by the keyword NULL cannot be inferred; hence NULL can be used to denote the null value only in certain contexts, rather than everywhere that a literal is permitted. —Although the null value is neither equal to any other value nor not equal to any other value—it is *unknown* whether or not it is equal to any given value—in some contexts, multiple null values are treated together; for example, the <group by clause> treats all null values together.

3. ANSI+ISO+IEC+9075-1-1999, American National Standards Institute, Inc. (1999), pp. 5, 13.

Every column of every row in a table must be assigned either a column domain value for that column or NULL. SQL Server Query Analyzer displays the word NULL when a query returns NULL for a column.

2.2.3 Constant (Literal) Defined

A constant or literal is a non-NULL specific data value of an indicated data type.

- **String Literal** is one or more characters enclosed in single quotes, e.g., 'Mary'.
- **Unicode String Literal** has capital N preceding the string, e.g., N'Mary'.
- **Date Literal** is a date contained in single quotes, '03-May-2000'.
- **Numeric Literal** is an integer (**int**) or floating point number (no quotation marks), e.g., 12 , 2.3.

The format for constants for each data type are given below.

2.2.4 Identifiers—Naming SQL Objects

An *identifier* is the name a database user assigns to a database object such as a table, column or view. This section describes the rules for creating identifier names. The next section describes how to use identifiers in one-, two-, three-, and four-part object names. The complete rules for forming legal Microsoft SQL Server identifiers are given below, but the safest and most portable subset of these rules is given here.

2.2.4.1 Identifier Format Rules, Short Form—Suggested Form

- The first character may be any alphabetic character in upper or lower case (a-z, A-Z).
- Subsequent characters may be any upper or lower case alphabetic or numeric or underscore character (a-z, A-Z, 0-9, _).
- It must contain between 1 and 128 characters (max of 116 for local temporary table).
- The identifier must not be a Transact-SQL reserved word (see Appendix A for list).

Examples: Table1, employees, hire_date

Every database object in SQL Server can have an identifier. Some objects, such as tables, are required to have an identifier; for other objects, such as constraints, identifiers are optional. Remember that identifiers are case-insensitive in SQL statements (unless installed case sensitive).

The maximum of 116 for a local temporary table name allows the system to add a unique suffix. This permits different users to concurrently call the same global stored procedure and each have a uniquely identified table.

2.2.4.2 Identifier Format Rules, Complete Form— Regular and Delimited

Microsoft SQL Server has two classes of identifiers: regular and delimited.

Regular Identifier Rules A regular identifier starts with a letter followed by an alphanumeric character or underscore, and it does not contain embedded spaces. A regular identifier is the most common and is the suggested form of name to give a database object.

Regular identifier characters must conform to the following rules.

- The first character must be
  a. an upper or lower case alphabetic character (a-z, A-Z), or
  b. a Unicode Latin character of another language, or
  c. underscore, “at” sign or number sign (_, @, #). See First Character below.
- Subsequent characters must be
  a. an upper or lower case alphabetic or numeric character (a-z, A-Z, 0-9), or
  b. a Unicode Latin character of another language, or
  c. underscore, “at” sign, number sign or dollar sign (_, @, #, $).

  (Note that embedded spaces are not allowed by these rules.)
- Identifiers must contain between 1 and 128 characters (max 116 for local temp table).
- The identifier must not be a Transact-SQL reserved word (see Appendix A for list).

First Character identifiers starting with some characters have special meaning as shown in Table 2-2.

Table 2-2 First Characters

First Character(s)	Meaning	Examples
@	local variable or parameter name	**@variablename**
@@	system function (do not start your own object names with @@)	**@@version**
#	temporary table (max of 116 characters) or a local temporary procedure (max of 128 characters)	**#localtable** **#localproc**
##	global temporary object	**##globaltable, ##globalproc**

Example: Typical use of regular identifiers

```
SQL
-- Regular Identifiers:  table1, column_av
CREATE TABLE  table1   (   column_a  VARCHAR(40)   )

SQL
INSERT INTO    table1  ( column_a )
VALUES ( 'String Data in single quotes' )

SQL
SELECT  *  FROM  table1

Result
column_a
-------------------------------------
String Data in single quotes
```

I recommend restricting your identifiers to regular identifiers such as **table1** and **column_a**.

2.2.4.3 Delimited Identifiers

A delimited identifier is an identifier enclosed in brackets ([ ]) or double quotes (" ") and may include special characters such as embedded spaces or tabs in the identifier name.

Remember the following comments about delimited identifiers.

- Many people, like me, don't recommend special characters such as spaces because they can cause problems.
- If you must use spaces or other special characters in an identifier, it is suggested that you use brackets instead of double quotes since the latter require the session setting QUOTED_IDENTIFIER be set to ON, but brackets are always valid.
- A regular identifier enclosed in brackets refers to the same object. E.g., **[table1]** and **table1** are the same object.

Note: When SQL Server generates scripts, it puts all identifiers in brackets, so don't be concerned that pubs.dbo.jobs looks a little funny when it appears as **[pubs.[dbo].[jobs]**.

2.2.4.4 Delimited Identifier Rules

First and subsequent characters may be any alphanumeric, punctuation or special character (including space or tab) except for the delimiter characters themselves. Delimited identifiers may include basically any character on the keyboard (except the delimiters) including spaces, tabs and punctuation marks.

Identifier must contain between 1 and 128 characters not counting the delimiters (max of 116 for local temporary table, see page 299). Also, Transact-SQL keywords may be used as identifiers if delimited.

`CREATE TABLE [table] ( column_a INT )` `--` Keyword as table name is a bad idea!

Example: Using a delimited identifier using [ ] to delimit the identifier

SQL

```
CREATE TABLE  [three word table]
( column_a  VARCHAR(40) )
```

```
SQL
INSERT [three word table]  ( column_a )
VALUES ( 'String Data in single quotes' )
SQL
SELECT  *  FROM  [three word table]

Result
column_a
---------------------------------------
String Data in single quotes
```

Underscore or capital letters can be used to avoid embedded spaces: three_word_table or ThreeWordTable.

I recommend adhering to ANSI SQL and good form as follows.

- Use the ON setting for QUOTED_IDENTIFIER.
- Use regular identifiers (no embedded spaces, see Identifier Format Rules, Short Form—Suggested Form above).
- If you must delimit an identifier use brackets as they are always valid.
- Use single quotes to delimit all string literals.

If you follow these suggestions then you may skip Section 2.2.4.5.

2.2.4.5 QUOTED_IDENTIFIER

This section is applicable if you have identifiers, which are delimited by double quotes.

A **QUOTED_IDENTIFIER** is a database option that, when ON, causes adherence to ANSI SQL rules regarding quotation mark delimiting identifiers and literal strings. When the option **QUOTED_IDENTIFIER** is set to ON (usual and recommended) follow these recommendations.

- Either brackets ([ ]) or double quotes (" ") may be used to delimit identifiers.
- All strings delimited by double quotes are interpreted as object identifiers.
- String literals must be delimited by single quotes and NOT by double quotes.

When database option QUOTED_IDENTIFIER is set to OFF follow these guidelines.

- Only brackets ([ ]) may be used to delimit identifiers.
- String literals may be delimited by either single or double quotes, though double quotes do not conform to the ANSI SQL and so single quotes are always recommended to enclose string literals.

The default setting for the QUOTED_IDENTIFIER is technically OFF, effectively ON. Although the default database setting for QUOTED_IDENTIFIER is OFF , both the ODBC driver and the OLE DB Provider for SQL Server automatically set QUOTED_IDENTIFIER to ON when connecting which overrides the default database setting.

This ON setting for QUOTED_IDENTIFIER is in effect unless the user explicitly executes

SET QUOTED_IDENTIFIER OFF

as is done in the following example. So clients using ODBC or OLE DB (almost all SQL Server clients today including Query Analyzer) see an ON setting for QUOTED_IDENTIFIER. (ON is good.)

See a complete discussion in Server, Database and Session Settings on p.174 and also see Which Session Setting Is in Effect? on p. 219.

Example: QUOTED_IDENTIFIER is ON, so either [ ] or " " may be used to delimit identifier.

```
SQL
CREATE TABLE  [three word table] (   a  VARCHAR(40)   )
SQL
INSERT INTO    "three word table"  ( a )
VALUES ( 'String Data in single quotes' )
SQL
SELECT  *  FROM  [three word table]

Result
a
----------------------------------------
String Data in single quotes
```

Example: Setting QUOTED_IDENTIFIER is OFF allows only [ ] to delimit identifier.

```
SQL SET  QUOTED_IDENTIFIER  OFF
SQL
CREATE TABLE  [multi word table name in brackets]  (  a  VARCHAR(40)  )
SQL
INSERT INTO "multi word table name in dbl quotes"  ( a )
VALUES ( 'String Data in single quotes' )

Result
Server: Msg 170, Level 15, State 1, Line 1
Line 1: Incorrect syntax near 'three word table'.
```

2.2.5 Using Identifiers as Qualified Object Names

Names of objects used in examples in this book usually use a one-part name, the object name itself. This section describes SQL Server one-, two-, three- and four-part 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*

server defaults to the server of the current connection.

database defaults to the current database.

owner_name defaults to current login.

Qualifiers may be omitted as follows.

- Leading default qualifiers may be omitted resulting in three-, two- or one-part names.
- Intermediate default qualifier(s) may be replaced by a period. owner_name marked by a period defaults first to the current login if the object owned by that owner can be found and, if not, then to dbo.

Valid forms of object names are as follows.

- *server.database.owner_name.object_name*: Fully Qualified Object Name
- *database.owner_name.object_name*: Current server
- *database..object_name*: Current server and current login or dbo
- *owner_name.object_name*: Current server and current database
- *object_name*: Current server and database and current login or dbo

Example: The local server is named amy.

Create a link to SQL Server instance CAT2 on host CAT and do a query to it.

```
SQL
EXEC  sp_addlinkedserver @server = 'CAT2_Link'        --
Specify Server Link Name
,    @srvproduct = ' '
,    @provider = 'SQLOLEDB'
,   @datasrc = 'CAT\CAT2'
-- hostname\SQLServerInstanceName

USE pubs
        go
SELECT *   FROM  authors             -- OR:  .authors
SELECT *   FROM  dbo.authors         -- OR:  ..authors
SELECT *   FROM  pubs.dbo.authors    -- OR:  pubs..authors
SELECT *   FROM  .pubs.dbo.authors
-- OR:  ...authors OR:  .pubs..authors
SELECT *   FROM  amy.pubs.dbo.authors
SELECT *   FROM  northwind.dbo.orders
-- Etc. for any database on the current server
SELECT *   FROM  CAT2_Link.pubs.dbo.authors
-- OR:  CAT2_Link.northwind.dbo.orders
```

The following forms do not work.

```
SQL
SELECT *   FROM  amy.pubs..authors
-- OR: amy..dbo.authors    OR:  amy...authors
SELECT *   FROM  CAT2_Link.pubs..authors
-- OR: CAT2_Link..dbo.authors OR:  CAT2_Link...authors
```

2.2.5.1 Qualified Column Names

Column names of a table or view may be qualified using the form:

table_name.column_name, view_name.column_name, or **table_alias.column_name**

where **table_name** or **view_name** may be a one-, two-, three- or four-part name as described above.

Example: Create a link to SQL Server instance CAT2 on host CAT and do a query to it.

Run a distributed query from the local host to the linked server using tables from both.

```
SQL
```

```
USE pubs
go
```

```
SQL
```

```
SELECT  TOP 2   p.pub_name , e.lname + ', ' + e.fname EmpName
   FROM      publishers p , CAT2_Link.pubs.dbo.employee e
   WHERE      p.pub_id  =  e.pub_id
```

```
Result
```

```
pub_name                 EmpName
-------------------------  ---------------------------
Algodata Infosystems     Afonso, Pedro
Binnet & Hardley         Accorti, Paolo
```

Notice that pub_id column names p.pub_id and e.pub_id must be qualified by table alias (or table name if no table alias had been declared) to avoid ambiguity.

The SELECT list uses the form *table_alias.column_name.* The FROM clause uses the fully qualified four-part name for the employee table on CAT2.

2.3 EXPRESSIONS IN AN SQL STATEMENT

An expression is a combination of operands and operators, which evaluates to a scalar:

- **Scalar:** a single data value such as number 13, date '1 Jan 2003' or string 'Jim Doe'

- **Operand:** a constant (literal), column name, variable, scalar function subquery whose result set is a scalar value
- **Operator:** any legal operator symbol allowed by the data type of the operand(s)

An expression generally assumes the data type of its component operands except that operands combined using comparison or logical operators result in a value of Boolean data type.

String, numeric and date expressions may usually be used anywhere a scalar value of their data type is required in INSERT, UPDATE, DELETE and SELECT statements. (See the SELECT example below.)

Boolean (logical) expressions may appear where a <search_condition> is specified as in a WHERE clause of UPDATE, DELETE or SELECT statements. They may evaluate to TRUE, FALSE or NULL (unknown). A Boolean expression may also be used in an IF or WHILE construct.

Syntax:

expression ::= operand | (expression) | unary_operator expression
| expression binary_operator expression

operand ::= constant | [table_alias.]column_name | variable | scalar_function
| (scalar_subquery)

unary_operator ::= operator that operates on one expression: operator expression
E.g., unary minus, -, as with (- 6).

binary_operator ::= operator that operates on two expressions: expr operator expr.
E.g., binary minus, -, as with (12 - 6).

For complete syntax see Books Online, Index: expressions, overview.

Example: Examples of string expressions:

```
ename
'Mary Smith'
first_name || ' ' || last_name
```

Examples of numeric expressions:

```
salary
123.45
22 + 33
(salary + 22) * 1.2
AVG(salary)
1.5 * AVG(salary)
```

Example of Boolean (logical) expressions:

```
qty > 45    -- evaluates to TRUE, FALSE or NULL (unknown, as if qty is NULL)
```

Example of subquery expression:

A subquery is a SELECT statement enclosed in parenthesis used inside another DML statement. See page 554 for more detail on subqueries.

The bold text in the following statement is a subquery expression

```
SELECT *  FROM titles
      WHERE  price >=
      ( SELECT AVG(price) FROM titles WHERE type = 'business')
```

Example of numeric, string and date expressions in SELECT list

SQL

```
SELECT   'Hello', 1 ,  (SELECT 1) + 2 ,   GETDATE()
```

Result

-------	----------	----------	----------
Hello	1	3	2001-05-11 16:28:27.140

Example of Boolean expression in WHERE clause <search condition>

SQL

```
SELECT ord_num, qty  FROM sales WHERE qty > 45
```

Result

ord_num	qty
---------	------
A2976	50
QA7442.3	75

2.3.1 Operators

An operator is a symbol specifying an action that is performed on one or more expressions. Microsoft SQL Server 2000 uses the following operator categories.

- Arithmetic operators: all numeric data types
- Assignment operators: all data types
- Bitwise operators: all integer data types plus binary and varbinary data types
- Comparison operators: all data types except text, ntext or image
- Logical operators: operand data type depends on operator

- String concatenation operators: character or binary string data types
- Unary operators: all numeric data types

When two expressions of different compatible data types are combined by a binary operator the expression of lower precedence is implicitly converted to the data type of higher precedence, the operation is performed and the resulting value is of the higher precedence data type. See Data Type Precedence List on page 74.

2.3.1.1 Arithmetic Operators

The arithmetic operators, as shown in Table 2-3, may be used with any of the numeric types. Addition and subtraction also work with datetime and smalldatetime for adding or subtracting a number of days.

Table 2-3 Arithmetic Operators

+	Addition
–	Subtraction and unary negation
*	Multiplication
/	Division
%	Module (Remainder)
( )	Grouping

Notes:

Both symbols **+** and **–** are unary and binary operators.

The symbol **/** returns the data type with the higher precedence as usual. So if both dividend (top) and divisor are integer types, the result is truncated (not rounded) to an integer.

The symbol **%** is *modulo* or *remainder* and returns the integer remainder of the division. Both operands must be integer data types. So 1%3 is 1 and 4%3 is also 1 as is 7%3.

Examples:

```
1 + 2
4 / 2
( 2.5 + 3 ) * 2
```

2.3.1.2 Relational Operators

These relational operators, shown in Table 2-4, may be used with string, numeric or date values.

Table 2-4 Relational Operators

=	equal to
<>	not equal to
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to

2.3.1.3 Assignment Operator: =

There is one assignment operator, equal sign (=), used primarily with local variables.

Example:

```
SQL
DECLARE   @MyInt   INT
SET  @MyInt = 123  -- Assignment Operation
PRINT @MyInt
```

```
Result
123
```

2.3.1.4 Bitwise Operators

Bitwise operators work on all integer data types and bit data types. (See Transact-SQL Data Type Hierarchy, page 74.)

Binary and varbinary data types may be one operand if an integer type is the other operand. Table 2-5 shows the necessary symbols for this use.

Table 2-5 Bitwise Operators

Bitwise AND	Bitwise OR	Bitwise EOR
&	|	^

Example:

```
SQL
PRINT 1 | 2
```

SQL (cont.)

Result

```
3
```

2.3.1.5 Comparison Operators

Comparison operators, shown in Table 2-6, also called relational operators, work on all data types except text, ntext or image.

Table 2-6 Comparison Operators

Equal To	Greater Than	Less Than	Greater or Equal To	Less Than or Equal To	Not Equal To	Not Less Than	Not Greater Than
=	>	<	>=	<=	<>	!<	!>

The result of a comparison is of Boolean type with three possible values: TRUE, FALSE or NULL (unknown). Boolean type may not be used as column or variable data types.

Comparisons are most often used in a WHERE clause or an IF statement.

Example:

SQL

```
SELECT price FROM titles WHERE price > 21
```

Result

```
price
---------------------
22.95
21.59
```

2.3.1.6 Logical Operators and Boolean Predicates

Logical operations test the truth of some condition and return a Boolean value: TRUE, FALSE or NULL (unknown). Acceptable operand data types of logical operations depend on the operator.

Books Online lists all of the entries in Table 2-7 as logical operators. They can all appear in a WHERE clause as a Boolean <search condition>.

Table 2-7 Logical Operators

AND	OR	NOT	ANY/ SOME	ALL	BETWEEN	IN	EXISTS	LIKE

2.3.1.7 AND, OR, NOT with NULL

The first three are the well-known logical operators from classical Boolean algebra. A significant difference of SQL Boolean logic from classical Boolean logic is the effect that NULL has in creating three-way logic (TRUE, FALSE, NULL).

Using NULL in SQL Boolean Logic The ANSI SQL-99 Standard[4] says the following about NULL.

> Although the null value is neither equal to any other value nor not equal to any other value — it is *unknown* whether or not it is equal to any given value

So NULL can be read as UNKNOWN in SQL Boolean logic expressions.

In evaluating a WHERE clause as in a SELECT, UPDATE or DELETE statement,

WHERE <search_condition>

the <search_condition> is a Boolean predicate which evaluates to TRUE, FALSE or NULL. Only those rows that are *known to be true* are selected to be in the result set. So any row for which the WHERE predicate evaluates to FALSE (known to be FALSE) or `NULL` (unknown) will not be retained. See WHERE clause syntax page 478. The same is true when applied to the HAVING clause of a SELECT statement.

Example: Consider the rows in Table 2-8 of the `titles` table in the `pubs` database.

Table 2-8 Rows from Titles Table

title_id	price
PC1035	22.95
MC3021	2.99
PC9999	NULL
...	...

4. ANSI+ISO+IEC+9075-1-1999, American National Standards Institute, Inc. (1999), pp. 5, 13.

Which of these rows will be returned by the following SELECT statement on that table?

```
SQL
SELECT  title_id, price
FROM   pubs.dbo.titles
WHERE price > 22
```

Of these rows only the first row is returned since, for the other two rows, the predicate evaluates to FALSE and NULL (unknown) respectively.

```
Result
title_id        price
--------        ---------------------
PC1035          22.9500
```

IS NULL The correct way to find all rows for which a column value is NULL is to use the IS NULL predicate.

```
SQL
SELECT  title_id, price
FROM    pubs.dbo.titles
WHERE  price IS NULL
```

This returns only PC9999 from the previous table.

```
Result
title_id        price
--------        ---------------------
PC9999          NULL
```

IS NOT NULL or NOT IS NULL The correct way to find all rows for which a column value is not NULL is to use the IS NOT NULL predicate or to negate the IS NULL predicate by preceding it with NOT.

```
SQL
SELECT  title_id, price
FROM     pubs.dbo.titles
WHERE  price IS NOT NULL   -- Or: WHERE  NOT price  IS NULL
```

Either form returns both of the first two rows from the previous table.

Caution: Never use "`columnname = NULL`" or "`columnname <> NULL`" to find rows with `NULL` or non-`NULL` values. These forms are syntactically correct but logically of no value since they always return zero rows. Always use **IS NULL**.

Despite this caution, the = as an assignment operator is used in UPDATE to change a column value to `NULL`.

Updating a Value to `NULL` A value may be set to `NULL` using the form

UPDATE *tablename* SET *columnname* = `NULL`

Example: Change the price to `NULL` for title_id MC3021.

```
SQL
UPDATE   pubs.dbo.titles
SET price = NULL
WHERE title_id = 'MC3021'
```

Compound Predicates using AND, OR and NOT with NULL The WHERE or HAVING predicate of a SELECT statement may include compound conditions using AND and OR operators.

Example: Consider the SELECT statement in Figure 2-1.

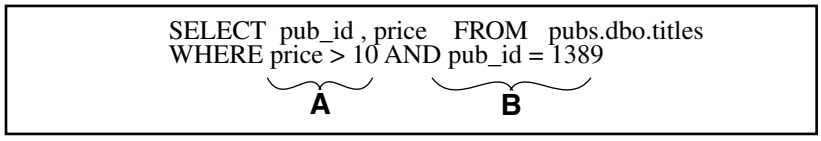

Figure 2-1 Selected Statement.

When executed, the WHERE predicate is evaluated on each row in the table and only if both A and B are known to be TRUE then the row is included in the result set. So any row where price or pub_id is NULL will not appear in the final result set.

Tables 2-9, 2-10 and 2-11 are truth tables for AND, OR and NOT.

Table 2-9 Truth Table for A AND B

		B		
		TRUE	**FALSE**	**NULL**
A	**TRUE**	TRUE	FALSE	NULL
	FALSE	FALSE	FALSE	FALSE
	NULL	NULL	FALSE	NULL

Hint: Read NULL as UNKNOWN

Table 2-10 Truth Table for A OR B

		B		
		TRUE	**FALSE**	**NULL**
A	**TRUE**	TRUE	TRUE	TRUE
	FALSE	TRUE	FALSE	NULL
	NULL	TRUE	NULL	NULL

Table 2-11 Truth Table for NOT A

A	**TRUE**	**FALSE**	**NULL**
NOT A	FALSE	TRUE	NULL

2.3.1.8 ANY/SOME

ANY and SOME are equivalent and may be used interchangeably.

Syntax:

```
scalar_expression { = | < > | ! = | > | > = | ! > | < | < = | ! < }
              { ANY | SOME } ( subquery )
```

Arguments

`scalar_expression`

This is any valid Microsoft SQL Server expression.

`{ = | <> | != | > | >= | !> | < | <= | !< }`

This is any valid comparison operator.

`subquery`

This is a SELECT statement that has a result set of one column. The data type of the column returned must be the same data type as scalar_expression.

Return

Type: Boolean

ANY/SOME returns TRUE if the comparison specified is TRUE for ANY pair (scalar_expression, x) where x is a value in the single-column set. Otherwise, ANY/SOME returns FALSE.

Example: List all publishers' names that have a `title` in the `titles` table.

SQL

```
USE  pubs      --  Move to the pubs database
go

SELECT pub_id, pub_name FROM publishers
WHERE pub_id = ANY ( SELECT pub_id FROM titles )
```

Result

pub_id	pub_name
0736	New Moon Books
0877	Binnet & Hardley
1389	Algodata Infosystems

This same result can be obtained by using the IN and EXISTS operators (explained on the following pages) or by doing an INNER JOIN of **`publishers`** and **`titles`**.

SQL

```
SELECT DISTINCT p.pub_id, p.pub_name -- Same result as previous query
FROM publishers p , titles t
WHERE p.pub_id = t.pub_id
```

2.3.1.9 ALL

ALL compares a scalar value with a single-column set of values.

Syntax:

```
scalar_expression { = | <> | != | > | >= | !> | < |
<= | !< } ALL ( subquery )
```

Arguments

scalar_expression

This is any valid SQL Server (scalar) expression.

```
{ = | <> | != | > | >= | !> | < | <= | !< }
```

This is a comparison operator.

subquery

This is a subquery that returns a result set of one column. The data type of the returned column must be the same data type as the data type of *scalar_expression*. It may not contain an ORDER BY clause, COMPUTE clause or INTO keyword.

Result

Type: Boolean

ALL returns TRUE if the specified comparison is TRUE for *scalar_expression* and every value in the single-column set returned from the subquery, otherwise ALL returns FALSE.

`ALL` can often be used to find the complement of an `ANY/SOME` query.

Example: List all publishers' names that have NO `title` in the `titles` table.

SQL

```
SELECT pub_id, pub_name FROM publishers
WHERE pub_id <>  ALL ( SELECT pub_id FROM titles )
```

Result

```
pub_id      pub_name
------      ------------------------------
1622        Five Lakes Publishing
1756        Ramona Publishers
9901        GGG&G
9952        Scootney Books
9999        Lucerne Publishing
```

2.3.1.10 BETWEEN

BETWEEN specifies inclusion in a specified range.

Syntax

```
test_expr [ NOT ] BETWEEN begin_expr AND end_expr
```

Arguments

test_expr

This is the expression to test for in the range defined by *begin_expr* and *end_expr.*

NOT

Not specifies that the result of the predicate be negated.

begin_expr

This is any valid expression.

end_expr

This too is any valid expression.

Result

Type: Boolean

BETWEEN returns TRUE if the value of test_expr is greater than or equal to the value of begin_expr and less than or equal to the value of end_expr.

NOT BETWEEN returns TRUE if the value of test_expr is less than the value of begin_expr or greater than the value of end_expr.

Remarks

test_expr, *begin_expr* and *end_expr* must be the same data type.

If any input to the BETWEEN or NOT BETWEEN predicate is NULL, the result is UNKNOWN.

The BETWEEN operator is a shorthand equivalent to

```
( test_expr >= begin_expr AND test_expr <= end_expr )
```

Example: Find all publishers with pub_id values between 500 and 900 — use both forms.

SQL

```
SELECT pub_id, pub_name FROM publishers
WHERE pub_id BETWEEN 500 AND 900

SELECT pub_id, pub_name FROM publishers  -- equivalent result
WHERE pub_id >= 500 AND pub_id <= 900
```

Result

```
pub_id      pub_name
------      ------------------------------------
0736        New Moon Books
0877        Binnet & Hardley
```

Example: Find all publishers with pub_id values outside of 500 and 900—use both forms.

SQL

```
SELECT pub_id, pub_name FROM publishers
WHERE pub_id NOT BETWEEN 500 AND 900
SELECT pub_id, pub_name FROM publishers -- equivalent result
WHERE NOT (pub_id >= 500 AND pub_id <= 900)
```

Result

pub_id	pub_name
------	----------
1389	Algodata Infosystems
1622	Five Lakes Publishing
...	...

2.3.1.11 IN

IN determines if a given value matches any value in a subquery or a list.

Syntax

```
test_expression [ NOT ] IN
    (      subquery   | expression [ ,...n ]     )
```

Arguments

test_expression

This is any valid Microsoft SQL Server expression.

subquery

A subquery has a result set of one column. This column must have the same data type as *test_expression*.

expression [,...n]

This is a list of expressions to test for a match. All expressions must be of the same type as *test_expression*.

Result

Type: Boolean

If the value of *test_expression* is equal to any value returned by *subquery* or is equal to any expression from the comma-separated list, the result value is TRUE. Otherwise, the result value is FALSE.

Using NOT IN negates the returned value.

Example with *subquery*: List all publishers' names with a **`title`** in the **`titles`** table.

SQL

```
SELECT pub_id, pub_name FROM publishers
WHERE pub_id IN ( SELECT pub_id FROM titles )
```

Result

Same as shown above using ANY.

Example: With *expression* list, list all publishers' names with a **`pub_id`** of 0736 or 0877.

SQL

```
SELECT pub_id, pub_name FROM publishers
WHERE pub_id IN ( 0736 , 0877 )
```

Result

```
pub_id  pub_name
------  ----------------------
0736    New Moon Books
0877    Binnet & Hardley
```

2.3.1.12 EXISTS

EXISTS is used only with a subquery (a SELECT statement enclosed in parentheses used inside another DML statement). It is TRUE if the subquery result set is nonempty.

See page 702 for more detail on subqueries.

Syntax

```
EXISTS subquery
```

Result

Type: Boolean

EXISTS returns TRUE if the subquery returns any rows at all (nonempty result set).

Example: List all publishers' names that have a `title` in the `titles` table.

SQL

```
SELECT pub_id, pub_name FROM publishers p
WHERE EXISTS
( SELECT pub_id FROM titles WHERE pub_id = p.pub_id)
```

Result

The result of this is the same as that shown above using ANY.

2.3.1.13 LIKE

LIKE provides pattern matching searching of character strings. LIKE determines whether or not a given character string matches a specified pattern. A pattern can include regular characters and wildcard characters. During pattern matching, regular characters must exactly match the characters specified in the character string. Wildcard characters, however, can be matched with arbitrary fragments of the character string. If any of the arguments are not of character string data type, SQL Server converts them to character string data type, if possible.

Syntax

```
match_expression [ NOT ] LIKE pattern [ ESCAPE
escape_character ]
```

Arguments

match_expression

This is any valid SQL Server expression of character string data type.

pattern

This syntax is the pattern to search for in match_expression, and can include the valid SQL Server wildcard characters shown in Table 2-12.

Table 2-12 Valid Wildcard Characters

LIKE Wildcard	Matches
% (percent)	Any string of zero or more characters
_ (underscore)	Exactly one of any character
[ ]	Any single among those listed inside the [ ]. E.g., a, b, c or d will match [abcd] or its shortcut form [a-d].
[^]	Any single character NOT listed in the [ ]. E.g., any character other than a, b, c or d will match [^abcd] or [^a-d].
Note: The [ ] and [^] notations are not included in the ANSI SQL standard and are thus not guaranteed to be portable. The rest of the LIKE notation does comply with SQL-92 and 99.	

escape_character

This is any valid SQL Server expression of any of the character string data types. The escape_character has no default and may consist of only one character.

For example, if you want to search for a literal percent sign (%) embedded in text, you could declare an escape character that does not occur in the text, e.g., \, and use it to escape the %. A sample search might be as follows.

```
SELECT * FROM titles
       WHERE notes LIKE '% 30\% discount %'
       ESCAPE '\'
```

The first and last % are wildcards, the \% causes the search for a literal %.

Some people prefer to use % as the escape character so that %% becomes a literal percent sign and a single % is a wildcard.

Result

Type: Boolean

LIKE returns TRUE if the match_expression (usually a column name) matches the specified pattern.

Example: List all authors whose last names begin with Gr.

```
SQL
SELECT au_lname FROM authors WHERE au_lname LIKE 'Gr%'
```

SQL (cont.)

Result

```
au_lname
------------------
Green
Greene
Gringlesby
```

Remarks

When you perform string comparisons with LIKE, all characters in the pattern string are significant, including leading or trailing spaces.

For example, using LIKE 'abc ' (abc followed by one space), a row with the value abc (without a space) is not returned. However, trailing blanks, in the expression to which the pattern is matched, are ignored. For example, using LIKE'abc'(abc without a space), all rows that start with abc followed by zero or more trailing blanks are returned.

2.3.1.14 Performance Note: Anywhere Search

A LIKE query which begins with a wildcard, either % or _, is called an Anywhere Search and is usually not recommended if it can be avoided.

```
WHERE  au_lname  LIKE '%st%'
```

, or

```
WHERE  au_lname  LIKE '%st'
```

, or

```
WHERE  au_lname  LIKE '_st'
```

The reason to avoid anywhere searches is the impact on performance, especially if the table being searched is large. This is because an index, if one exists on the au_lname column, can not be used in an anywhere search. Queries on large tables will take a very, very, very long time if no index is used.

To see why an anywhere search disables the use of an index, think of an index being similar to a telephone book and imagine the usefulness of the telephone book if looking for each of the queries above.

On the other hand, the following query does NOT disable the use of an index on au_lname if one exists since the match string does not BEGIN with a wildcard: WHERE au_lname LIKE 'st%'

2.3.1.15 String Concatenation Operator: +

There is one concatenation operator, plus sign (+), and it may be used with character or binary string data types.

Example:

```
SQL
SELECT lname, fname, lname + ', ' + fname FROM employee

Result
lname       fname
-------     -------     ----------------
Cruz        Aria        Cruz, Aria
```

Note: The ANSI SQL concatenation operator is "||" used in Oracle, DB2 and others. The + string concatenation operator in SQL Server is not portable to other RDBMSs.

2.3.1.16 Unary Operators

Unary operators work on a single numeric operand expression.

Bitwise NOT only works with integer data types, the other two take any numeric data type. See Table 2-13.

Table 2-13 Unary Operators

Positive	Negative	Bitwise NOT
+	—	~

Notes:
Positive (+) means the numeric value is positive.
Negative (–) means the numeric value is negative.
Bitwise NOT returns the ones complement of the number.

Example:

```
SQL
print  1  +  -3    -- + is binary operator, - is unary

Result
-2
```

2.3.1.17 Operator Precedence

When a complex expression has multiple operators, operator precedence determines the sequence in which the operations are performed. Operators have the precedence levels shown in Table 2-14. An operator on higher levels is evaluated before an operator on a lower level. In case of two operators of equal precedence evaluation proceeds from left to right

Table 2-14 Operator Precedence

Operator
+ (Positive), – (Negative), ~ (Bitwise NOT)
* (Multiply), / (Division), % (Modulo)
+ (Add), (+ Concatenate), – (Subtract)
=, >, <, >=, <=, <>, !=, !>, !< (Comparison operators)
^ (Bitwise Exclusive OR), & (Bitwise AND), | (Bitwise OR)
NOT
AND
ALL, ANY, BETWEEN, IN, LIKE, OR, SOME
= (Assignment)

2.4 SQL SERVER 2K DATA TYPES

SQL Server supports a wide variety of native data types which may be used for column data type in a CREATE TABLE statement and for local variables and function return types. Virtually all SQL-92 data types are supported as well as a few additional ones as shown in Table 2-15.

Table 2-15 Data Types

T-SQL Data Type Name	SQL-92	General Comments Each type is described individually in the following sections.
BIGINT		
BINARY		SQL-92 has **BLOB** (binary large object)
BIT	Yes	SQL-92 has a different **BIT** data type.
CHAR	Yes	

Table 2-15 Data Types (cont.)

T-SQL Data Type Name	SQL-92	General Comments Each type is described individually in the following sections.
DATETIME		SQL-92 has **DATE** and **TIME**
DECIMAL	Yes	
FLOAT	Yes	
IMAGE		SQL-92 has **BLOB** (binary large object)
INT	Yes	
MONEY		
NCHAR	Yes	
NTEXT		
NVARCHAR	Yes	
NUMERIC	Yes	
REAL	Yes	
SMALLDATETIME		
SMALLINT	Yes	
SMALLMONEY		
SQL_VARIANT		
SYSNAME		NVARCHAR(128) — predefined by SQL Server
TEXT		
TIMESTAMP		SQL-92 has a different **TIMESTAMP** type.
TINYINT		
VARBINARY		SQL-92 has **BLOB** (binary large object)
VARCHAR	Yes	
UNIQUEIDENTIFIER		

SQL Server data types are divided into three major categories: Numeric, String and Other as shown in the following tables.

2.4.1 Numeric Data Type Overview

Table 2-16 broadly describes the numeric data types. The last column gives the page where each is described in detail with examples given.

Table 2-16 Numeric Data Types

SQL Server 2000 Numeric Data Types	Description	Details and Examples
BIGINT	Integer values from -2^{63} through $+(2^{63}-1)$ or -9.22×10^{18} through 9.22×10^{18}	page 76
INT INTEGER	Integer values from -2^{31} through $+(2^{31}-1)$ or –2 billion through + 2 billion INTEGER is a synonym for data type INT.	page 76
SMALLINT	Integer values from -2^{15} through $+(2^{15}-1)$ or (–32,768) through +32,767)	page 76
TINYINT	Integer values from 0 through +255	page 76
BIT	Integer data with value either 0 or 1 (or NULL)	page 82
DECIMAL(p,s) **DEC(p,s)** **NUMERIC (p,s)**	Numeric data , fixed precision (*p*) and scale (*s*) from -10^{38} through $+(10^{38}-1)$. $p <= 38$, $s <= p$ DEC is a synonym for DECIMAL. DECIMAL and NUMERIC are functionally equivalent. See DECIMAL and NUMERIC details below.	page 83
MONEY	Monetary data values from -2^{63} through $+(2^{63}-1)$ with accuracy to a ten-thousandth of a monetary unit. (-9.22×10^{18} through 9.22×10^{18})	page 86
SMALLMONEY	Monetary data values from –214,748.3648 through +214,748.3647, with accuracy to a ten-thousandth of a monetary unit.	page 86
FLOAT(n) DOUBLE PRECISION	Floating point number data from –1.79E + 308 through 1.79E + 308. FLOAT(*n*) causes *n* bits to be used to store the mantissa, *n* = 1–53 If *n* = 1–24, storage size is 4 Bytes and precision is 7 digits If *n* = 25–53, storage size is 8 Bytes and precision is 15 digits If (*n*) is missing as in FLOAT, then *n* defaults to 53. DOUBLE PRECISION is a synonym for FLOAT(53)	page 87

Table 2-16 Numeric Data Types (cont.)

SQL Server 2000 Numeric Data Types	Description	Details and Examples
REAL FLOAT(24)	Floating point number data from –3.40E + 38 through 3.40E + 38. FLOAT(24) is a synonym for REAL	page 87
DATETIME	Date and time data from January 1, 1753, through December 31, 9999, with an accuracy of three-hundredths of a second, or 3.33 milliseconds.	page 90
SMALLDATETIME	Date and time data from January 1, 1900, through June 6, 2079, with an accuracy of one minute.	page 90
The words in bold are the SQL Server 2K base data type name. The other word appearing in the same cell is a synonym, which may be used interchangeably with the base data type name in Transact-SQL statements. It is the base data type and not the synonym that is stored and will be seen from operations such as **sp_help**.		

2.4.2 String Data Type Overview

Table 2-17 broadly describes the parameters of string, or character, data types. In Sections 2.4.12 through 2.4.16, the string data types listed here are described in detail.

Table 2-17 String Data Types

SQL Server 2000 String Data Types	Description	Details and Examples
CHAR(*n*) CHARACTER(*n*)	Fixed-length non-Unicode character data with a length of *n* bytes where n = 1 to 8,000. Default length with DDL is 1, with CAST function is 30. Always stores *n* bytes, with blanks appended for strings less than *n* char. Example non-Unicode literal: 'Sue'	page 97
VARCHAR(*n*) CHAR VARYING(*n*) CHARACTER VARYING(*n*)	Variable-length non-Unicode character data with maximum length of *n* bytes. *n* = 1 to 8,000. Default length with DDL is 1, with CAST function is 30. Stores the actual number of bytes in the string up to the maximum of *n*.	page 97

Table 2-17 String Data Types (cont.)

SQL Server 2000 String Data Types	Description	Details and Examples
NCHAR (*n*) NATIONAL CHAR NATIONAL CHARACTER	Fixed-length Unicode data with a maximum length of 4,000 characters. Always stores 2x*n* bytes, with blanks appended as needed as for CHAR. Unicode is the ISO standard 16-bit (2 byte) character set capable of representing every language in the world. Example Unicode literal: N'Sue'	page 102
NVARCHAR (*n*) NATIONAL CHAR VARYING NATIONAL CHARACTER VARYING	Variable-length Unicode character data with maximum length of *n* characters. *n* must be a value from 1 through 4,000. Storage size, in bytes, is two times the number of characters entered. The data entered can be 0 characters in length.	page 102
TEXT	Variable-length non-Unicode data and with a maximum length of $2^{31}-1$ (2 billion) characters.	page 107
NTEXT NATIONAL TEXT	Variable-length Unicode data with a maximum length of $2^{30}-1$ (1 billion) characters. Storage size, in bytes, is two times the number of characters entered.	page 107
IMAGE	Variable-length binary data from 0 through $2^{31}-1$ or 0 through 2 GB.	page 107
SYSNAME	System supplied user-defined data type defined as NVARCHAR(128).	page 107
BINARY (*n*)	Fixed-length binary data with a length of *n* bytes. *n* = 1 to 8,000. Default length with DDL is 1, with CAST function is 30. Stores (*n* + 4 bytes).	page 110
VARBINARY (*n*) BINARY VARYING(*n*)	Variable-length binary data with a maximum length of 8,000 bytes. Default length with DDL is 1, with CAST function is 30. Stores (actual length + 4 bytes).	page 110

The words in bold are the SQL Server 2K base data type name. The other word or words appearing in the same cell are synonyms, which may be used interchangeably with the data type name in Transact-SQL statements.

2.4.3 Other Data Type Overview

Table 2-18 broadly describes the characteristics of several other data types. The last column gives the page where each is described in detail with examples given. We should note that cursor and table data types may not be used as column data types but they may be used for local variables and function return types.

Table 2-18 Other Data Types

SQL Server 2000 Other Data Types	Description	Details and Examples
CURSOR	A data type for cursor variables or stored procedure OUTPUT parameters that contain a reference to a cursor variable. A cursor variable is a Transact-SQL variable capable of containing the result set of a query; it may be updatable. See Cursors, p. 498. Cursors cannot be used as the column data type of a table.	page 111
SQL_VARIANT	A data type that can store values of any SQL Server–supported data types, except TEXT, NTEXT, TIMESTAMP and SQL_VARIANT.	page 112
TABLE	A special data type to store table structured data such as a result set. **Table** data type cannot be used as the column data type of a table. It is somewhat similar to cursors: a table variable is for temporary storage of data whereas cursors have more programmatic control including the ability to cause updates back to the original base table.	page 120
TIMESTAMP ROWVERSION	A data type that exposes automatically generated binary numbers, which are guaranteed to be unique within a database. Timestamp is used typically as a mechanism for version-stamping table rows. The storage size is 8 bytes. ROWVERSION should be used in place of TIMESTAMP data type as the latter is slated to change behavior in a future release of SQL Server to correspond to ANSI SQL behavior.	page 123
UNIQUEIDENTIFIER	A globally unique identifier (GUID). The only operations that are allowed against a uniqueidentifier value are comparisons (=, <>, <, >, <=, >=) and checking for NULL (IS NULL and IS NOT NULL).	page 127
The words in bold are the SQL Server 2K base data type name. The other word appearing in the same cell is a synonym which may be used interchangeably with the data type name in Transact-SQL statements. It is the base data type and not the synonym that is stored and will be seen from operations such as **sp_help**.		

2.4.4 Transact-SQL Data Type Precedence

2.4.4.1 Implicit Data Type Conversions

Implicit data type conversions are those conversions that are done by SQL Server when neither the CAST or CONVERT function is specified. Implicit data type conversions use the Data Type Precedence List shown and are done to complete either of the following tasks.

- **comparing two expressions of different data types:** When comparing two expressions of different data types supported by implicit conversion, the expression of the lower precedence data type is implicitly converted to the data type of the higher precedence, and then the comparison is made. If implicit conversion is not supported, an error is returned. For a table containing all implicit data type conversions and which conversions are supported, see Books Online index: "CAST and CONVERT" and scroll to the table shown under "Remarks."
- **evaluating two operand expressions of different types joined by a binary operator:** When two expressions of different compatible data types are combined by a binary operator, the expression of lower precedence is implicitly converted to the data type of higher precedence, the operator's operation is performed and the resulting value is of the higher precedence data type. If implicit conversion is not supported, an error is returned.

Data Type Precedence List	
sql_variant (highest)	**bit**
datetime	**ntext**
smalldatetime	**text**
float	**image**
real	**timestamp**
decimal	**uniqueidentifier**
money	**nvarchar**
smallmoney	**nchar**
bigint	**varchar**
int	**char**
smallint	**varbinary**
tinyint	**binary** (lowest)

2.4.5 Transact-SQL Data Type Hierarchy

SQL Server documentation divides similar data types into three major categories.

- Numeric Data Types
- Character and Binary String Data Types
- Other Data Types

It further arranges them in the hierarchy shown in Figure 2-2.

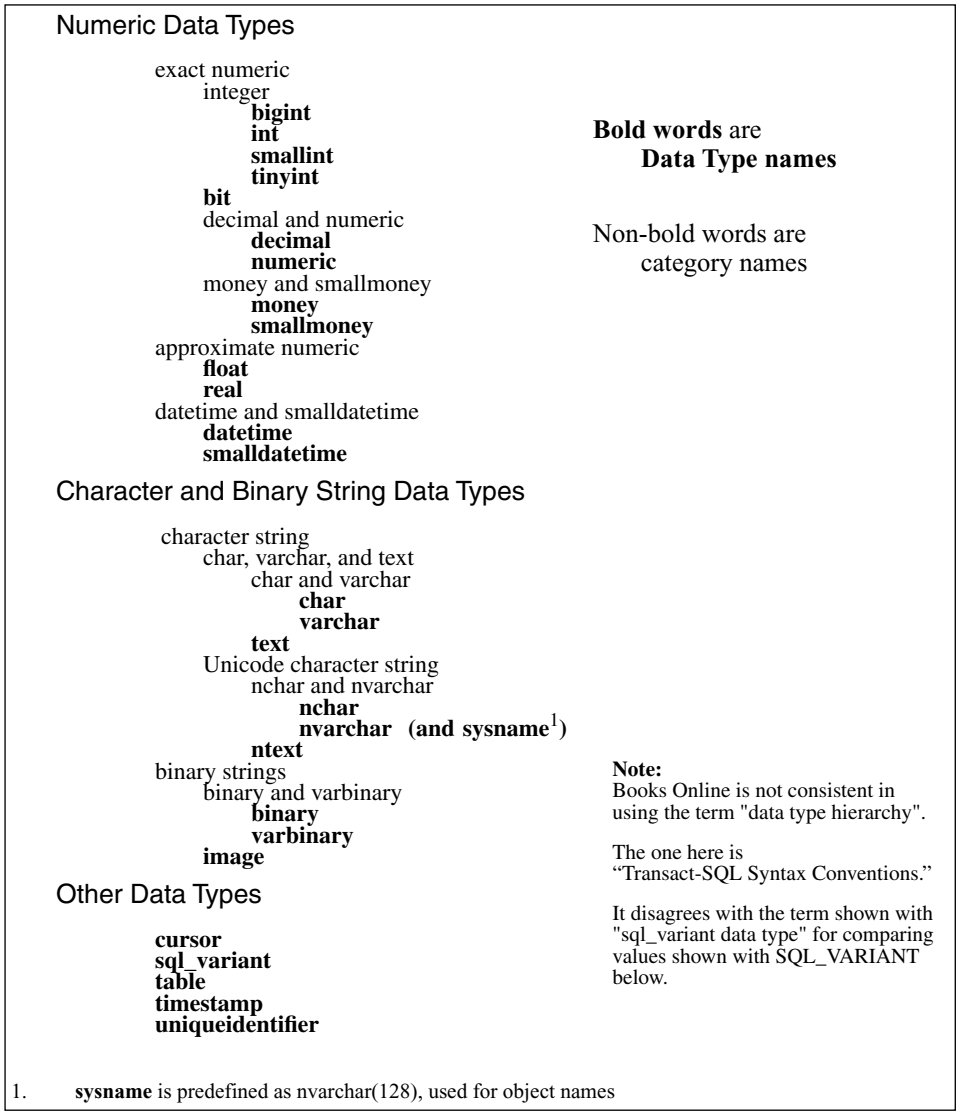

Figure 2-2 Data Type Hierarchy.

The three major categories of data type are described in the following sections.

2.4.5.1 sp_datatype_info

The **sp_datatype_info** system-stored procedure in Microsoft SQL Server returns information about the data types supported by the current environment. Table 2-19 gives a partial listing of output from **sp_datatype_info** run on SQL Server 2K. See Books Online for more details.

Table 2-19 sp_datatype_info

```
SQL EXEC sp_datatype_info
```

Result

TYPE_NAME	...	PRECISION	LITERAL_PREFIX	LITERAL_SUFFIX
sql_variant	...	8000	NULL	NULL
uniqueidentifier	...	36	'	'
ntext	...	1073741823	N'	'
nvarchar	...	4000	N'	'
sysname	...	128	N'	'
nchar	...	4000	N'	'
bit	...	1	NULL	NULL
tinyint	...	3	NULL	NULL
bigint	...	19	NULL	NULL
image	...	2147483647	0x	NULL
varbinary	...	8000	0x	NULL
binary	...	8000	0x	NULL
timestamp	...	8	0x	NULL
text	...	2147483647	'	'
char	...	8000	'	'
numeric	...	38	NULL	NULL
decimal	...	38	NULL	NULL
money	...	19	$	NULL
smallmoney	...	10	$	NULL
int	...	10	NULL	NULL
smallint	...	5	NULL	NULL
float	...	15	NULL	NULL
real	...	7	NULL	NULL
datetime	...	23	'	'
smalldatetime	...	16	'	'
varchar	...	8000	'	'

2.4.6 `BIGINT, INT, SMALLINT` and `TINYINT` Details

Table 2-20 provides a recap of the first four data types. Details and examples of these types follow.

Table 2-20 Recapping Data Types

Data Type	Description	Storage Size
`BIGINT`	Integer values from -2^{63} through $+(2^{63}-1)$ or -9.22×10^{18} through 9.22×10^{18}	8 Bytes
`INT` `INTEGER`	Integer values from -2^{31} through $+(2^{31}-1)$ or –2 billion through + 2 billion `INTEGER` is a synonym for data type INT.	4 Bytes
`SMALLINT`	Integer values from -2^{15} through $+(2^{15}-1)$ or (–32,768) through (+32,767)	2 Bytes
`TINYINT`	Integer values from 0 through +255	1 Byte

2.4.6.1 Integer Data Type Constants (Literals)

A constant (literal) is a representation of a specific, fixed scalar data value.The format of a constant depends on the data type of the value it represents. Integer literals consist of a sequence of numeric characters not enclosed in quotation marks, containing neither decimal point nor comma and optionally preceded by plus or minus prefix.

Example:

–13, 13, +13

13.89 is truncated to 13, not rounded. So a decimal point is permitted, but not used.

Any fractions generated by arithmetic operations on these four integer data types are truncated, not rounded.

Example:

3 / 4 evaluates to 0.

2.4.6.2 `BIGINT` — Special Considerations

The **bigint** data type is new with SQL Server 2K. It is intended for cases where the integer values may exceed the range supported by the **int** data type. The **int** data type remains the primary integer data type in SQL Server.

`BIGINT` with Expressions SQL Server will not automatically promote **tinyint**, **smallint** or **int** values to bigint, although it sometimes does automatically promote **tinyint** or **smallint** to int.

`BIGINT` with Functions Functions will return bigint only if the input parameter expression is a **bigint** data type.

Example:

SQL

```
CREATE TABLE a ( x int)
INSERT INTO a VALUES (2000000000)  -- 2 x 10^9, almost the max int value
INSERT INTO a VALUES (2000000000) -- the sum of exceeds int range

SELECT SUM(x)  FROM a -- Error, Input parameter is int, sum returns int
```

Result

```
Server: Msg 8115, Level 16, State 2, Line 1
Arithmetic overflow error converting expression to data type int.
```

SQL

```
SELECT CAST(SUM(x) AS bigint) FROM a  -- Error happens
          -- before the cast
```

Result

```
Server: Msg 8115, Level 16, State 2, Line 1
Arithmetic overflow error converting expression to data type int.
```

The solution is to CAST the column value to **bigint** before doing the SUM.

SQL

```
SELECT SUM( CAST(x AS bigint) ) FROM a  -- Correct result
```

Result

```
--------------------
4000000000
```

Unless explicitly stated in the documentation, functions and system-stored procedures with **int** expressions for their parameters have not been changed to support implicit conversion of **bigint** expressions to those parameters. For this reason, SQL Server implicitly converts **bigint** to **int** only 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.

When you provide **bigint** as input parameters and the return values are of type **bigint**, you may use the Transact-SQL functions shown in Table 2-21. See details under Functions, page 137.

Table 2-21 Functions

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

According to Books Online, certain aggregate functions will not return a **bigint** unless the input parameter expression is of type **bigint**.

Example:

SQL

```
CREATE TABLE b ( y bigint)
INSERT INTO b VALUES (2000000000) -- 2 x 109, almost the max int value
INSERT INTO b VALUES (2000000000)


SELECT AVG(y) FROM b  -- Input parameter y is bigint, returns in int
range
```

Result

```
-------------------
2000000000
```

```
SQL
SELECT SUM(y) FROM b  -- Input parameter y is bigint,
  sum returns bigint

Result
--------------------
4000000000
```

Recall previous example, x is **int** so it must be CAST to **bigint** as input parameter to SUM so that result will be bigint to accommodate the large result value.

Example:

```
SQL
SELECT SUM( CAST(x AS bigint) ) FROM a -- Correct result

Result
--------------------
4000000000
```

The functions shown in Table 2-22 may be used to reference **bigint** columns or variables though they do not return **bigint** values. See details under Functions, page 137.

Table 2-22 **bigint** References

@@IDENTITY	ISNULL	VARP
COL_LENGTH	ISNUMERIC	
DATALENGTH	STDEV[P]	

SQL Server provides two functions just for **bigint** values, COUNT_BIG and ROWCOUNT_BIG.

`COUNT_BIG` Function This function is used when counting the number of items in a very large group if the value might exceed the range supported by the **int** data type. It returns a **bigint** type.

Example:

```
SQL
SELECT COUNT_BIG( a_column )  FROM  verybigtable
  -- Works like count() but returns bigint
  -- The data type of column doesn't matter,
     the number of rows does

Result
--------------------
4000000000
```

Note: If the number of rows is within the **int** range then either COUNT_BIG() or COUNT() may be used though the return type differs.

`ROWCOUNT_BIG` Function Use this function when counting the number of rows affected in the last statement executed and when 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.

Example:

```
SQL
SELECT ROWCOUNT_BIG()  -- Like @@ROWCOUNT but returns bigint.
    Needed if table is HUGE.

Result
--------------------
4000000000
```

Note: If the number of rows returned by the last statement is within the **int** range, then either ROWCOUNT_BIG() or @@ROWCOUNT may be used though the return type differs.

BIGINT with Stored Procedures and Other Transact-SQL Elements

SQL Server will not automatically promote tinyint, smallint or int values to **bigint**, although sometimes it automatically promotes tinyint or smallint to int.

CAST and CONVERT support **bigint**, applying 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 CAST and CONVERT page 166.

When using the CASE expression, you will get a result of type **bigint** if either the *result_expression* or the *else_result_expression* if present evaluates to **bigint**. See CASE page 164.

You may use **bigint** for exact numeric data type in these Transact-SQL statements.

- CREATE TABLE
  ALTER TABLE
  Example:

```
CREATE TABLE c ( x int, y bigint)
INSERT INTO c  VALUES ( 20 , 3000000000 )
```

- CREATE PROC[EDURE]
  ALTER PROC[EDURE]
  Example:

```
CREATE PROC pr ( @parm bigint ) AS print @parm + 4
EXEC pr 3000000000
```

- DECLARE variable
  Example:

```
DECLARE  @var1   bigint
SET  @var1 = 3000000000 ;   print @var1
```

2.4.6.3 Specifying BIGINT Constants

Whole number constants outside the range supported by the **int** data type continue to be interpreted as **numeric**, with a scale of 0 and a precision sufficient for the value specified.

Example: The constant 3000000000 is interpreted by SQL Server as `NUMERIC(10,0)`.

These constants are implicitly convertible to **bigint** and can be assigned to **bigint** columns and variables. So in the examples from the previous section, the constant 3000000000 is seen by SQL Server as `NUMERIC(10,0)` and implicitly cast to **bigint**.

```
INSERT INTO c  VALUES ( 20 , 3000000000 )
```

and

```
EXEC pr 3000000000
```

To explicitly create a **bigint** constant use the CAST function,

```
CAST( 3000000000 AS BIGINT).
```

Example:

```
SQL
print CAST( 3000000000 AS BIGINT)

Result
3000000000
```

2.4.7 BIT Data Type Details

The **bit** data type is described in Table 2-23.

Table 2-23 BIT Data Type

Data Type	Description	Storage Size
BIT	Integer data with value either 0 or 1 (or NULL) Columns of type **bit** cannot have indexes on them.	1 Byte for bits 1 to 8 2 Bytes if NULLABLE

2.4.7.1 BIT Data Type Constants (Literals)

The numbers zero and one represent **bit** constants.. If a number larger than one is used, it is converted to one. (Empirically, any number other than 0 is stored as 1.)

No quote marks are used.

Example **bit** constants:

0 1

Example:

```
SQL
CREATE TABLE d ( x BIT )

INSERT INTO d VALUES (0);
INSERT INTO d VALUES (1);
INSERT INTO d VALUES (12);

SELECT * FROM d;
```

SQL (cont.)

Result

```
 x
----
0
1
1
```

2.4.8 `DECIMAL` and `NUMERIC` Data Type Details

Table 2-24 offers an overview of the DECIMAL and NUMERIC data types.

Table 2-24 DECIMAL and NUMERIC Data Type

Data Type	Description
`DECIMAL(`*p*`,`*s*`)`	Numeric data , fixed precision (*p*) and scale (*s*) from -10^{38} through $+(10^{38} - 1)$. *p* <= 38, *s* <= *p*
`DEC(`*p*`,`*s*`)`	DEC is a synonym for DECIMAL.
`NUMERIC (`*p*`,`*s*`)`	DECIMAL and NUMERIC are functionally equivalent. Storage size depends on precision; see Table 2-25.

DECIMAL and NUMERIC are exact numeric data types with fixed precision (*p*) and scale (*s*) containing numbers from $(-10^{38} + 1)$ through $(+10^{38} - 1)$. That is, they store an exact representation of any number up to 38 digits which may all be to the left or right of the decimal point (or some on the left and the rest on the right). FLOAT and REAL may store larger and smaller numbers, but they are approximate. The number for *p* (precision) specifies the maximum total number of decimal digits that can be stored, both to the left and to the right of the decimal point. The precision must be a value from 1 through the maximum precision which is 38. If not specified *p* defaults to 18. In SQL Server 7.0 the maximum value for *p* is 28 unless the server is started with the /p option, **sqlserver /p**, in which case the maximum precision is 38.

The number for *s* (scale) specifies the maximum number of decimal digits that can be stored to the right of the decimal point. Scale must be a value from 0 through *p*, so, 0 <= *s* <= *p*. The default scale is 0.

Legal declarations are as follows:

- DECIMAL(*p,s*) where 0 <= *p* <= 38 and 0 <= *s* <= p.
- DECIMAL(*p*) where 0 <= *p* <= 38 and *s* defaults to 0.
- DECIMAL where *p* defaults to 18 and *s* defaults to 0.

Note: DECIMAL may be replaced by either DEC or NUMERIC with equivalent results.

For DECIMAL(*p*,*s*) and NUMERIC(*p*,*s*), the **integer part may not exceed (p – s) digits**. The result will be an error. If more than *s* digits are specified for the **fractional part the value stored is rounded to *s* digits**. Table 2-25 below shows how the storage sizes vary based on the precision.

Table 2-25 Varying Storage Size

Precision	Storage bytes
1–9	5
10–19	9
20–28	13
29–38	17

2.4.8.1 Numeric Data Type Constants (Literals)

Table 2-26 summarizes the constant (literal) format for the numeric and decimal data types. Examples of these types follow.

Table 2-26 Numeric Data Type Constants

Data Type	Constant (Literal) Format
DECIMAL(*p, s*) DEC (*p, s*) **NUMERIC**(*p, s*)	A sequence of numbers not enclosed in quotation marks that may include decimal point but not a comma. Value must be within the range for the data type. Examples: For DEC(4,2): –13 or –13.24 or 13 or 13.24

Constants for numeric data types are represented by a sequence of numeric digits optionally preceded by a plus symbol or a minus symbol and are optionally followed by a decimal point and another sequence of numeric digits. No quote marks are used.

Example:

```
SQL
CREATE TABLE e ( x DEC(5,2) , y DEC(5) , z DEC)
EXEC SP_HELP e
```

Result

Column_name	Type	Length	Prec	Scale
x	decimal	... 5	5	2
y	decimal	... 5	5	0
z	decimal	... 9	18	0

Length represents the number of bytes of storage as given in Table 2-26.

This **sp_help** output shows that	is stored as
DEC(5,2)	DEC(5,2)
DEC(5)	DEC(5,0)
DEC	DEC(18,0)

Example: DEC(5,2) can store from –999.99 to +999.99.

```
SQL INSERT INTO e (x) VALUES (123.45)     -- Okay
```

Entering a number with more than ($p - s$) digits to the left of the decimal point, or (5 – 2) = 3 in this example, is an error.

```
SQL INSERT INTO e (x) VALUES(1234.00)-- Error - overflow
```

Any number of digits may be entered to the right of the decimal point. If more than s digits are entered, the number is rounded to have exactly s digits to the right of the decimal.

Example:

```
SQL
INSERT INTO e (x) VALUES (-999)       -- Okay
INSERT INTO e (x) VALUES (12.89)      -- Okay
INSERT INTO e (x) VALUES (123.899)    -- Okay
INSERT INTO e (x) VALUES (-123.899)   -- Okay

SELECT  x  FROM e
```

SQL (cont.)

```

```

Result

```
 x
-------
123.45
–999.00
12.89
123.90
–123.90
```

2.4.9 `MONEY` and `SMALLMONEY` Data Type Details

The data types **money** and **smallmoney** are distinguished by their range, as shown in Table 2-27.

Table 2-27 MONEY and SMALLMONEY Data Type

Data Type	Range	Accuracy	Storage Size
MONEY	From –922,337,203,685,477.5808 through +922,337,203,685,477.5807	4 Decimal Places	8 Bytes
SMALLMONEY	From -214,748.3648 through 214,748.3647	4 Decimal Places	4 Bytes

The monetary data types, **money** and **smallmoney**, can store positive or negative amounts of money. However, if more than four decimal places are required, use the decimal data type instead.

The data type is versatile enough to accept many national currency symbols including the British pound, the Euro, the Yen and many others. See details in Books Online, search for “Using Monetary Data.”

2.4.9.1 Monetary Data Type Constants (Literals)

The acceptable constant (literal) format for monetary data types is summarized in Table 2-28.

Table 2-28 Monetary Data Type

MS SQL Server 2K Data Type	Constant (Literal) Format
MONEY	A sequence of numbers not enclosed in quotation marks with one optional decimal point and one optional currency symbol as a prefix within the range from -2^{63} through $+(2^{63}-1)$ i.e., -9.22×10^{18} through 9.22×10^{18}. Ex: –13 or –13.24 or 13.24 or –$13.24 or –$922337203685477.5808
SMALLMONEY	Same as for MONEY with range from –214,748.3648 through +214,748.3647. Examples: -$13 or 13.24 or -214748.3648 or $214748.3647

Example:

SQL

```
CREATE TABLE f ( m MONEY, s SMALLMONEY )

INSERT INTO f VALUES ( 12, 12);          -- Okay
INSERT INTO f VALUES ( $12345678, 12);   -- Okay
INSERT INTO f VALUES ( 12, $12345678 );  -- Error --  s too large
INSERT INTO f VALUES ( $12,345,678, 12); -- Error --  no commas
INSERT INTO f VALUES
  ( CAST('$12,345,678' AS MONEY), 12);   -- Okay

SELECT * FROM f;
```

Result

```
m                s
---------------  -----------
12.0000          12.0000
12345678.0000    12.0000
12345678.0000    12.0000
```

2.4.10 `FLOAT` and `REAL` Data Type Details

The data types FLOAT and REAL allow numerical approximations. They are summarized in Table 2-29.

DECIMAL and NUMERIC contain an exact representation of any number up to 38 digits. MONEY and SMALLMONEY also store an exact representation of numbers with a smaller range than DECIMAL and NUMERIC. However,

Table 2-29 FLOAT and REAL Data Type

Data Type	Description	Storage Size and Precision
FLOAT(*n*) DOUBLE PRECISION	Floating point number data in [–1.79E + 308, 1.79E + 308] FLOAT(*n*) causes *n* bits to be used to store the mantissa, *n* in [1,53] *n* defaults to 53 so FLOAT is the same as FLOAT(53) DOUBLE PRECISION is a synonym for FLOAT(53)	If *n* in [1,24] then storage = 4 bytes, precision = 7 digits If *n* in [25,53] then storage = 8 bytes, precision = 15 digits
REAL FLOAT(24)	Floating point number data in [–3.40E + 38, 3.40E + 38] FLOAT(24) is a synonym for REAL	storage = 4 bytes, precision = 7 digits

FLOAT and REAL contain an approximate representation with a range of much larger and smaller fractional numbers and take less storage space than the other data types for the same number.

2.4.10.1 FLOAT and REAL Data Type Constants (Literals)

Constants for approximate numeric data types are represented by a sequence of numeric digits with an optional decimal point and optionally preceded by a plus symbol or a minus symbol and optionally suffixed with the letter *E* and an integer representing an exponent of 10. No quote marks are used. See Table 2-30.

Table 2-30 FLOAT and REAL Data Types

MS SQL Server 2K Data Type	Constant (Literal) Format
Floating Point Format	[+|–]<nums>[.<nums>][E[+|–]<nums>] where <nums> is sequence of one or more numeric characters and the value following E, if present, represents the exponent of 10. Example Format: [+|–]12[.34][E[+|–]5] Examples: –12 or 12 or 12E–1 (which is 1.2) or 12E1 (which is 120)
FLOAT[(*n***)]** DOUBLE PRECISION	Floating Point Format in the range [–1.79E + 308, 1.79E + 308]
REAL	Floating Point Format in the range [–3.40E + 38, 3.40E + 38].

Example: Create a table with a REAL and a FLOAT and enter exactly the same value for each though using different notation

SQL

```
CREATE TABLE g ( r REAL , f FLOAT )

INSERT INTO g VALUES ( 123, 1.23E2 );        -- Okay
INSERT INTO g VALUES ( .0123 , 1.23E-2 );    -- Okay
INSERT INTO g VALUES ( .00123 , 1.23E-3 );   -- Okay

SELECT * FROM g;
```

Result

r	f
123.0	123.0
0.0123	0.0123
1.2300001E-3	0.00123

Because the stored values of FLOAT and REAL, all other values are approximate. WHERE clause comparisons with FLOAT and REAL columns should use > , < , >= and <= relational operators and should avoid = and <> operators.

Example:

SQL

```
SELECT * FROM g -- For each row the same value was entered for r and f
WHERE  r = f; -- But "=" only finds the first row as the same
```

Result

r	f
123.0	123.0

SQL

```
SELECT * FROM g
WHERE NOT ( r < f  AND r > f ); -- This finds all three rows
```

Result

```
r                     f
--------------------  ----------------------------------------------------
123.0                 123.0
0.0123                0.0123
1.2300001E-3          0.00123
```

2.4.11 `DATETIME` and `SMALLDATETIME` Data Type Details

Microsoft SQL Server 2000 has only **datetime** and **smalldatetime** data types (Table 2-31) to store both date and time data.

Table 2-31 DATETIME and SMALLDATETIME Data Type

Data Type	Description	Storage Size
DATETIME	Date and time data from Jan 1, 1753, through Dec 31, 9999, accurate to three-hundredths of a second, or 3.33 milli-seconds.	8 Bytes
SMALLDATETIME	Date and time data from Jan 1, 1900, through Jun 6, 2079, accurate to one minute.	4 Bytes

There is no separate data type for just date or just time. A date with no time changes to the default time of midnight (12:00 AM). A time with no date changes to the default date of Jan 1, 1900.

Guidelines for searching for dates and/or times are as follows:

- To search for an exact match on both date and time, use an equal sign (=).
- The relational operators <, <=, >, >=, <> may be used to search for dates and times before or after a specified value.
- To search for a partial date or time value (year, day, hour), use the LIKE operator.

WHERE hiredate LIKE ‘Jan 200%’

However, 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. (See details under Functions, page 137.) For more information about searching for partial dates or times, see LIKE on page 64.

- To search for an exact match on both date and time, use an equal sign (=).
- SQL Server evaluates **datetime** constants at run time.
- A date string that works for the date formats expected by one language may not work if the query is executed by a connection using a different language and date format setting. For more information see Books Online. Search for “Writing International Transact-SQL Statements.”

2.4.11.1 DATETIME Data Type Constants (Literals) Formats—for INPUT

SQL Server recognizes date and time literals enclosed in single quotation marks (') in the formats shown in Tables 2-32 and 2-33 for DATE and TIME data. A DATETIME value may be specified as either DATE [TIME] or [DATE] TIME where DATE and TIME follow the formats below.

Table 2-32 DATE Formats

DATE only Format Names (no time defaults to 12:00 AM)	Formats - put inside single quotes ' '
Alphabetic date format Month may be any case, e.g., April or april or APRIL or any other mixed case	Apr[il] [22] [,] 2001 Apr[il] 22 [,] [20]01 Apr[il] 2001 [22] Apr[il] 01 22 [22] Apr[il] [,] 2001 22 Apr[il] [,] [20]01 22 [20]01 Apr[il] [22] 2001 Apr[il] 2001 Apr[il] [22] 2001 [22] Apr[il]

Table 2-32 DATE Formats **(cont.)**

DATE only Format Names (no time defaults to 12:00 AM)	Formats - put inside single quotes ' '
Numeric date format	[0]4/22/[20]01 -- (mdy) [0]4-22-[20]01 -- (mdy) [0]4.22.[20]01 -- (mdy) [04]/[20]01/22 -- (myd) 22/[0]4/[20]01 -- (dmy) 22/[20]01/[0]4 -- (dym) [20]01/22/[0]4 -- (ydm) [20]01/[04]/22 -- (ymd) 2001-22-04
Unseparated string format	[20]010422

DATETIME values may be subtracted to give the number of days between, but test it carefully to make sure it gives what you want.

Example:

```
SQL
SELECT CAST (
CAST( '1/3/2003' AS DATETIME ) - CAST( '1/1/2003'AS DATETIME )
  AS INT )

Result
-------
```

Table 2-33 TIME Formats

TIME only Format Names (no date defaults to 1 Jan, 1900)	Formats - put inside single quotes ' '
Time format AM and PM may be any case	17:44 17:44[:20:999] 17:44[:20.9] 5am 5 PM [0]5[:44:20:500]AM
Note: 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.	

Table 2-34 shows several examples of DATETIME Constants.

Table 2-34 DATETIME Constants

	Examples - Represent same date and/or time
Alphabetic date format	'April 22, 2001' or '22 April, 2001'
Numeric date format	'04/22/2001' or '4/22/2001' or '4/22/01'
Unseparated string format	'20010422' or '010422'
Time only format	'5:44PM' or '17:44' or '17:44:20.999'
Date and Time - any combination of the above forms 'DATE TIME' 'TIME DATE'	'April 22, 2001 5:44PM' '20010422 17:44' '5:44PM April 22, 2001' '17:44 20010422'

SET DATEFORMAT SET DATEFORMAT sets the order of the dateparts (month/day/year) for entering datetime or smalldatetime data values as a string of 3 numbers separated by slash, /. See the example below which changes the order from the default mdy to the European format of dmy.

Syntax

```
SET DATEFORMAT { mdy | dmy | ymd | ydm | myd | dym }
```

See Server, Database and Session Settings, page 175.

Remarks

In the U.S. English, the default is mdy. Remember that this setting is **used for input only**, that is, only in the interpretation of character strings as they are converted to date values. It has no effect on the display of date values. All users may use SET DATEFORMAT without special permission.

Examples:

SQL

```
DECLARE   @v_date   DATETIME
SET @v_date = '04/22/03'
PRINT @v_date
```

Result

```
Apr 22 2003 12:00AM
```

SQL

```
SET DATEFORMAT dmy

DECLARE   @v_date   DATETIME
SET @v_date = '22/04/03'

PRINT @v_date

SET DATEFORMAT mdy --  Set input date format back to the default
```

Result

```
Apr 22 2003 12:00AM
```

Notice that PRINT and SELECT use a different output format for datetime objects.

SQL

```
DECLARE   @v_date   DATETIME
SET @v_date = '04/22/03'

SELECT @v_date
```

Result

```
-----------------------------------------------------
2003-04-22 00:00:00.000
```

2.4.11.2 Formatting DATETIME Data Type Constants (Literals) for OUTPUT

The default display format for DATETIME in SQL Server is arguably ugly: 2001-04-22 17:44:20.999. Table 2-35 gives some alternatives to obtain a more presentable output for a datetime column, variable or function.

Perhaps the easiest are CONVERT(VARCHAR[(19)] , *datetimevariable*) and CAST(*datetimevariable* AS VARCHAR[(19)]).

Table 2-35 Formatting DATETIME

Format String	Output
SELECT CONVERT(VARCHAR(19), GETDATE())	Apr 22 2001 5:44PM
SELECT CAST(GETDATE() AS VARCHAR(19))	Apr 22 2001 5:44PM
SELECT CONVERT(VARCHAR, GETDATE())	Apr 22 2001 5:44PM
SELECT CAST(GETDATE() AS VARCHAR)	Apr 22 2001 5:44PM
SELECT CONVERT(VARCHAR(10), GETDATE(), 101) (see CONVERT for more options)	04/22/2001
MONTH(GETDATE())	4
DAY(GETDATE())	22
YEAR(GETDATE())	2001

Example:

```
SQL
SELECT CONVERT( VARCHAR(19), GETDATE() )

Result
-------------------
Apr 22 2001  5:44PM
```

2.4.11.3 Avoiding the Problems of Y2K

The primary way to avoid problems such as that caused by Y2K is to always enter datetime values with four digits for the year. This should be standard procedure by now. Nonetheless, SQL Server does provide a two-digit year cutoff option which tells how two-digit years should be interpreted. A two-digit year that is less than or equal to the last two digits of the cutoff year is in the same century as the cutoff year. A two-digit year that is greater than the last two digits of the cutoff year is in the century that precedes the cutoff year.

The default **two-digit year cutoff** for SQL Server is 2049. That means that a two-digit year of 40 is interpreted as 2040.A two-digit year of 60 is interpreted as 1960.

See Books Online and search for “cutoff” to learn how to change the **two-digit year cutoff** value for the entire server.

2.4.12 CHAR and VARCHAR Data Type Details

An overview of char and varchar is contained in Table 2-36.

Table 2-36 CHAR and VARCHAR Data Type

Data Type	Description	Storage Size
CHAR(*n*) CHARACTER(*n*)	Fixed-length, non-Unicode character data of *n* characters with a fixed length of *n* bytes. Default length *n* with DDL is 1. (See page 106) Default length *n* with CAST function is 30. (See page 107) Always stores *n* bytes, padding the right with blanks for strings less that *n* characters. Example non-Unicode literal: 'Sue'	*n* Bytes *n* = 1 to 8,000.
VARCHAR(*n*) CHAR VARYING(*n*) CHARACTER VARYING(*n*)	Variable-length, non-Unicode character up to *n* characters data with maximum length of *n* bytes. Default length *n* with DDL is 1. (See page 106) Default length *n* with CAST function is 30. (See page 107) Stores the actual number of bytes in the string up to the maximum of *n*. Example non-Unicode literal: 'Sue'	length of the data entered <= *n* Bytes *n* = 1 to 8,000.

Character string data types include CHAR, VARCHAR, TEXT, NCHAR, NVARCHAR, NTEXT and SYSNAME which is NVARCHAR(128).

String data types are for storing sequences of zero or more characters. Essentially any character on the keyboard may be stored in a string data type, including the following characters.

- Upper and lower case alphabetic characters such as a, b, c, ..., z, A, B, C, ..., Z
- Numeric characters such as 0, 1, ..., 9
- Punctuation and special characters such as ., ,, ;, [,], @, #, &, !, ...

2.4.12.1 CHAR and VARCHAR Data Type Constants (Literals)

Character string constants consist of a sequence of characters enclosed in single quotes in accordance with ANSI SQL standard. Essentially any alphabetic, numeric or punctuation character on the keyboard may be stored in a string data type.

A literal single quote (apostrophe) in a string is represented by two consecutive single quote characters, as in 'O''Reilly' for O'Reilly.

Example string constants:

'Mary Smith'

'O''Reilly'

2.4.12.2 Storage of CHAR and VARCHAR Data Types

CHAR(*n*) and VARCHAR(*n*) are data types for storing string data in one byte per character, non-Unicode. The main difference between the two is as follows.

- CHAR(*n*) always stores *n* bytes to contain a string up to *n* characters long, filling in by right padding with blanks if the data is less than n characters.
- VARCHAR(*n*) stores one byte for each of the actual number of characters up to *n*.
- Both CHAR(*n*) and VARCHAR(*n*) truncate data longer than *n* bytes to exactly *n* bytes for a local variable but return an error on INSERT or UPDATE into a table column. See “Truncation Examples” below.

Storage Size Examples

'Sue' inserted into CHAR(6) will be stored as "Sue " with three blanks on the right. 'Sue' inserted into VARCHAR(6) will be stored as "Sue" plus number 3 for the length.

An explicit trailing space inserted into a VARCHAR will be retained, space permitting.

'Sue ' will correctly insert all four characters into VARCHAR(6) as "Sue". 'Sue ' will store six characters as usual in CHAR(6) as 'Sue ".

An empty string of zero characters may be stored by entering '' (<single quote><single quote>). Such an empty string is very different from NULL, which is "no domain value entered" (see NULL, page 39).

Truncation Examples for a Local Variable

'Sammy' inserted into CHAR(3) or VARCHAR(3) variable will be truncated to "Sam."

Example:

```
SQL
DECLARE  @name   VARCHAR(3)          -- or CHAR(3)

SET @name  =   'Sammy'

PRINT   @name
```

```
Result
```

Sam

Truncation Examples for a Table Column

If the string 'Sammy' is inserted into CHAR(3) or VARCHAR(3) column it returns an error and fails.

Example:

```
SQL
CREATE  TABLE   t ( name  VARCHAR(3) )-- or CHAR(3)

INSERT INTO   t   VALUES  ( 'Sammy'  )
go
```

```
Result
```

Server: Msg 8152, Level 16, State 9, Line 1
String or binary data would be truncated.
The statement has been terminated.

```
SQL
SELECT  *   FROM  t
```

```
Result
```

name

(0 row(s) affected)

Concatenation Operator

The concatenation operator is the plus sign (+). It may be used with string concatenation types.

Example:

```
SQL
PRINT 'Mary' + ' ' + 'Smith'
--  The middle term is <single quote><space><single quote>

Result
Mary Smith
```

Relational Operators

These relational operators, shown in Table 2-37 may be used with string values as well as numeric or date values. They behave the same as if one is alphabetizing a list, “Al” is before “Alan.”

Table 2-37 Relational Operators

=	equal to
<>	not equal to
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to

Strings are compared according to the collation, and they generally behave according to dictionary order. So 'Mary Smith' > 'Ma' is TRUE and 'Mary Smith' < 'N' is TRUE.

2.4.12.3 Comparing CHAR() and VARCHAR() Values

If a value of CHAR() data type value is compared to a VARCHAR() value, then SQL Server trims trailing blanks, if any, from the CHAR() value to allow them to be compared as equal values if the leading strings of both are the same.

Note that Oracle will not compare the two as equal, so this behavior is RDBMS dependent.

This behavior is true whether the values are local variables or table column values. The following example demonstrates it using local variables.

Example: In this example, `@charname` will contain five characters, abc followed by two spaces, as can be seen from the first SELECT output where it is concatenated with `@vchrname` since there are spaces between the two, that is **abc abc**. But `@vchrname` contains only the three visible characters as shown by the second SELECT output with the two reversed, **abcabc**. Last, the IF statement comparison shows that the values do compare as equal.

SQL

```
DECLARE @charname CHAR(5) '@vchrname VARCHAR(5)
SET @charname = 'abc'    -- Declared as CHAR(5) so it
  will store 5 characters 'abc'
SET @vchrname = 'abc'    -- Declared as VARCHAR(5) so it
  will store 3 characters 'abc'

SELECT 'Char-HELLO:   ' + @charname  + 'HELLO'
SELECT 'Varchar- HELLO:   ' + @vchrname  + 'HELLO'

IF @charname = @vchrname
  PRINT 'Same'
ELSE
  PRINT 'Different' >
```

Result

```
--------------------------
Char-HELLO:   abc  HELLO

---------------------------
Varchar- HELLO:   abcHELLO

Same
```

This last behavior of comparing the two showing equality is caused by implicit data type conversion from CHAR() to VARCHAR() before doing the comparison. See the discussion of implicit data type conversion and precedence on pages 73 and 74.

2.4.12.4 String Functions

The scalar functions, shown in Table 2-38, perform an operation on a string input value and return a string or numeric value. See details under Functions, page 137.

Table 2-38 Scalar Functions

ASCII	NCHAR	SOUNDEX
CHAR	PATINDEX	SPACE
CHARINDEX	REPLACE	STR
DIFFERENCE	QUOTENAME	STUFF
LEFT	REPLICATE	SUBSTRING
LEN	REVERSE	UNICODE
LOWER	RIGHT	UPPER
LTRIM	RTRIM	

ANSI_PADDING ON I suggest heeding the following good advice from Books Online—**Warning: It is recommended that ANSI_PADDING always be set to ON.**

- `VARCHAR(n)` — not trimmed, not padded
  - data inserted or updated retain and store trailing blanks provided (not trimmed).
  - only the characters provided are stored (not padded to the length of the column).
  - The four character string 'Sue ' inserted or updated into a VARCHAR(6) will retain the trailing blank explicitly inserted.
- `CHAR(n)` — padded (never trimmed)

ANSI_PADDING OFF

- `VARCHAR(n)` — trimmed, not padded.
  - The four-character string 'Sue ' inserted (or updated) into a VARCHAR(6) would be truncated storing only three characters, 'Sue'.
- `CHAR(n)` — NULLable not padded, NOT NULLable always padded, (neither ever trimmed)
  - CREATE TABLE t (a CHAR(3) , b VARCHAR(3)):a and b behave the same.

You can display the current setting of the ANSI_PADDING option as follows.

```
SQL
SELECT SESSIONPROPERTY ( 'ANSI_PADDING' )  ANSI_PADDING

Result
ANSI_PADDING
------------------------
1
```

The OLE DB and ODBC libraries always set ANSI_PADDING option to ON when making a connection. For more on OLE DB and ODBC automatic settings, see page 215.

Collation A COLLATE clause (new in SQL Server 2K) may be applied to a string expression, column definition, or database definition to determine comparison (search) and sorting characteristics. See Collation on page 233. Objects using **CHAR** or **VARCHAR** are assigned the default collation of the database, unless a specific collation is assigned using the COLLATE clause. (The same is true for **NCHAR** or **NVARCHAR**.)

2.4.13 `NCHAR` and `NVARCHAR` Data Type Details

The data types of nchar and nvarchar can be summarized as shown in Table 2-39.

Table 2-39 NCHAR and NVARCHAR Data Types

Data Type	Description	Storage Size
NCHAR(*n*)	Fixed-length Unicode character data of *n* characters with a fixed length of 2*n* bytes.	*2n* Bytes
NATIONAL CHAR(*n*)	Default length *n* with DDL is 1. (See p. 106) Default length *n* with CAST function is 30. (See p. 107)	*n* = 1 to 4,000.
NATIONAL CHARACTER(*n*)	Always stores two times *n* bytes, padding the right with blanks for strings less that *n* characters. The ANSI SQL synonyms for **nchar** are **national char** and **national character**. Example Unicode literal: N'Sue'	

Table 2-39 NCHAR and NVARCHAR Data Types (cont.)

NVARCHAR(*n*) NATIONAL CHAR VARYING(*n*) NATIONAL CHARACTER VARYING(*n*)	Variable-length Unicode character data of *n* characters. Default length *n* with DDL is 1. (See p. 105) Default length *n* with CAST function is 30. (See p. 106) Storage size, in bytes, is two times the number of characters entered. The data entered can be zero characters in length. The ANSI SQL synonyms for **nvarchar** are **national char varying** and **national character varying**. Example Unicode literal: N'Sue'	2 x length of the data entered <= *2n* Bytes n = 1 to 4,000.

2.4.13.1 UNICODE

UNICODE Data Types use two bytes per character to enable handling of international character sets. UNICODE Data Types were introduced with SQL Server 7.0.

Single-byte ASCII is able to accommodate European languages (Latin1) including English and German (with umlauts), but UNICODE is required for languages such as those of Asian and Middle Eastern countries. See Books Online: "Collations, overview."

UNICODE string literals have N prefixed to the string enclosed in single quotes.

Example UNICODE Literal

```
N'This specifies a UNICODE string literal'
```

The server will still have a default character set determined during installation. But you may now specify a column of a table to be of NCHAR, NVARCHAR or NTEXT data type able to contain any character set defined by the Unicode Standard (that is, any of the National Character data types in the ANSI SQL standard).

NCHAR(*n*) behaves similarly to CHAR(*n*) in that (*n*) characters are always stored, but the storage size for CHAR is one byte per character whereas NCHAR is two bytes per character. See the discussion of CHAR above.

NVARCHAR(*n*) behaves similarly to VARCHAR(*n*) in that only the number of characters in the string up to a maximum of (*n*) characters are stored. See the discussion of VARCHAR in section 2.4.12 above.

UNICODE Data Type FUNCTIONS Two of the string functions listed in the previous section specifically support Unicode data types.

NCHAR (*integer_expression*) **r**eturns the Unicode character with the given integer code.

UNICODE (*'ncharacter_expression'*) returns the integer value for the first character of the input expression.

Example:

```
SQL
PRINT NCHAR(252)

Result
ü
```

```
SQL
PRINT UNICODE( 'ü' )

Result
252
```

Books Online provides other examples of the use of NCHAR, NVARCHAR and the functions NCHAR() and UNICODE().

2.4.13.2 NCHAR and NVARCHAR Data Type Constants (Literals)

Unicode character string constants consist of a capital **N** followed by a sequence of characters enclosed in single quotes in accordance with ANSI SQL standard. The capital N stands for the National Language support in the SQL-92 standard. The N prefix must be uppercase.

Example Unicode (NCHAR and NVARCHAR) String Constants

N'Mary Smith'
N'O''Reilly'

2.4.13.3 Storage of NCHAR and NVARCHAR Data Types

NCHAR(*n*) always stores *n* characters, right padding with blanks if the data is less than *n* characters long. Storage space for these *n* characters is 2*n* bytes.

Example: If a column declared as NCHAR(5) has 'abc' inserted, then 'abc ' is stored, five characters occupying ten bytes.

NVARCHAR(*n*) stores the actual number of characters up to *n*, using two bytes per character.

Example: If a column declared as NVARCHAR(5) has 'abc' inserted, then those three characters are stored, occupying six bytes (plus the number 3 indicating the number of characters).

Collation A COLLATE clause (new in SQL Server 2K) may be applied to a string expression, column definition or database definition to determine comparison (search) and sorting characteristics. See Collation on page 233.

Objects using **NCHAR** or **NVARCHAR** are assigned the default collation of the database, unless a specific collation is assigned using the COLLATE clause. The same is true for **CHAR** or **VARCHAR**.

2.4.13.4 Default Length for CHAR, VARCHAR, NCHAR, NVARCHAR with DDL

The default length of CHAR, VARCHAR, NCHAR and NVARCHAR with DDL is 1. This means that a DDL declaration of CHAR defaults to CHAR(1).

Example 1: CHAR with DDL

SQL

```
CREATE TABLE  t  (
  a  CHAR                -- Defaults to  CHAR(1)
)

INSERT INTO  t  ( a ) VALUES ( 'A' ) -- Succeeds
INSERT INTO  t  ( a ) VALUES ( 'AAA' ) --  Fails.   A column
input too long fails
SELECT * FROM  t
CREATE TABLE  t  (
  a  CHAR                -- Defaults to  CHAR(1)
)

INSERT INTO  t  ( a ) VALUES ( 'A' ) -- Succeeds
INSERT INTO  t  ( a ) VALUES ( 'AAA' ) --  Fails, column
                                           input too long
SELECT * FROM  t
```

Result

```
Server: Msg 8152, Level 16, State 9, Line 1
String or binary data would be truncated.
The statement has been terminated.
a               b
----            ----
A               B
```

Example 2: CHAR with DDL

```
SQL
DECLARE  @variable CHAR -- Defaults to  CHAR(1)

SET  @variable = 'V'    -- Succeeds
PRINT @variable

SET  @variable = 'WWW'  -- Succeeds. A string variable input
                           that is too long is truncated.
PRINT @variable
```

```
Result
```

```
V
W
```

2.4.13.5 Default Length for CHAR, VARCHAR, NCHAR, NVARCHAR with CAST

The Default length of CHAR, VARCHAR, NCHAR and NVARCHAR with CAST is 30. This means that CAST (object AS CHAR) is the same as CAST (object AS CHAR(30))

Example: CHAR with CAST

```
SQL
PRINT CAST(  GETDATE()   AS  CHAR ) -- Defaults to CHAR(30)
```

```
Result
```

```
Jul 22 2002  3:36PM
```

```
SQL
PRINT  CAST(  GETDATE()   AS  CHAR(30) )
```

```
Result
```

```
Jul 22 2002  3:36PM
```

2.4.14 SYSNAME Data Type Details

SYSNAME is a system-supplied user-defined data type as NVARCHAR(128). SYSNAME is used to reference database object names.

2.4.15 TEXT, NTEXT and IMAGE Data Type

TEXT, NTEXT and IMAGE data types, as summarized in Table 2-40, are fixed and variable-length data types for storing large non-Unicode and Unicode character and binary data. They are generally for large data values up to 2 GB in size, which are more efficiently stored on their own pages than on the same page as the other columns of the row. They serve the function of BLOBs (Binary Large OBjects) in some systems.

Restrictions on their use include the inability to refer to them directly in a WHERE clause. But they may be used in WHERE clauses as input parameter to functions such as ISNULL(), SUBSTRING(), PATINDEX() as well as in IS NULL, IS NOT NULL and LIKE expressions. They may not be used as variables, although they may be parameters to stored procedures.

Table 2-40 TEXT, NTEXT and IMAGE Data Types

Data Type	Description	Storage Size
TEXT	Variable-length non-Unicode data with a maximum length of 231-1 (2,147,483,647) characters.	Multiples of 8KB Pages Max storage is 2GB
NTEXT	Variable-length Unicode data with a maximum length of 2^{30} - 1 (1,073,741,823) characters. Storage size, in bytes, is two times the number of characters entered. The ANSI SQL synonym for **ntext** is **national text**.	
IMAGE	Variable-length binary data from 0 through 2^{31}-1 (2,147,483,647) bytes.	

2.4.15.1 Text and Image Functions and Statements

Text and image functions, summarized in Table 2-41, perform an operation on a text or image value or column and return information about the value. All are nondeterministic (see page 162).

Table 2-41 Text and Image Functions and Statements

Statement Name	Description and Syntax
DATALENGTH	Returns the number of bytes used to represent any expression. Syntax: DATALENGTH (expression)
PATINDEX	Returns the starting position of the first occurrence of a pattern in a specified expression or zero if the pattern is not found. All text and character data types. Syntax: PATINDEX ('%pattern%' , expression)
SUBSTRING	Returns part of a character, binary, text, or image expression. Syntax: SUBSTRING (expression , start , length)
TEXTPTR	Returns the text-pointer value that corresponds to a text, ntext, or image column as a varbinary value. The retrieved text pointer value can be used in READTEXT, WRITETEXT, and UPDATETEXT statements. Syntax: TEXTPTR (column)
TEXTVALID	Returns 1 if a given text, ntext, or image pointer is valid, 0 if not. Syntax: TEXTVALID ('table.column' , text_ ptr)
READTEXT	Reads text, ntext, or image values from a text, ntext, or image column, starting from a specified offset and reading the specified number of bytes. Syntax: READTEXT { table.column text_ptr offset size } [HOLDLOCK]
SET TEXTSIZE	Specifies the size of text and ntext data returned with a SELECT statement. Syntax: SET TEXTSIZE { number }
UPDATETEXT	Updates an existing text, ntext, or image field. Use UPDATETEXT to change only a portion of a text, ntext, or image column in place. Logging depends on recovery model in effect for the database. Syntax: UPDATETEXT { *table_name.dest_column_name dest_text_ptr* } { NULL | *insert_offset* } { NULL | *delete_length* } [WITH LOG] [*inserted_data* | *table_name.src_column_name src_text_ptr*]
WRITETEXT	Permits nonlogged, interactive updating of an existing text, ntext, or image column. This statement overwrites any existing data in the column it affects. (WRITETEXT cannot be used on text, ntext, and image columns in views.) Syntax: WRITETEXT { table.column text_ptr } [WITH LOG] { data }

Example: This example shows the use of TXTPTR and WRITETEXT.

SQL

```
CREATE TABLE t (  id    INT , txtcol TEXT )
INSERT INTO t ( id , txtcol ) VALUES ( 1 , 'txtcol initial data' )
SELECT  *  FROM  t
```

SQL (cont.)

Result

```
id          txtcol
---         --------------------
1           txtcol initial data
```

SQL

```
DECLARE   @b_ptr   VARBINARY(16)

SELECT  @b_ptr = TEXTPTR( txtcol ) FROM  t WHERE  id = 1

WRITETEXT t.txtcol @b_ptr  'This represents a very long text message.'


SELECT  *  FROM  t
```

Result

```
id          txtcol
---         ----------------------------------------------------
1           This represents a very long text message.
```

Arguments

Text in Row

If your data is 7000 bytes or less, you may set the "Text in Row" feature that lets you store the large object on the same page as the other columns. This feature is enabled for an entire table with **sp_tableoption**.

The next statement turns the feature on and sets an upper limit on data size to the default maximum size of 256 bytes. Objects larger than the maximum value are stored on separate pages, not in rows.

```
sp_tableoption  tablename , 'text in row', 'ON'
-- ON must be enclosed in single quotes.
```

The next statement both turns the feature on and sets an upper limit on data size. The value specified must be between 24 and 7000.

```
sp_tableoption  tablename , 'text in row', 2000
```

2.4.16 `BINARY(n)` and `VARBINARY(n)` Data Type Details

BINARY and VARBINARY data types, summarized in Table 2-42, store strings of bits up to a maximum of 8000 bytes. From Books Online:

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

Table 2-42 BINARY(n) and VARBINARY (n) Data Type

Data Type	Description	Storage Size
BINARY(*n*)	Fixed-length binary data of *n* bytes. *n* must be a value from 1 through 8,000. Storage size is *n*+4 bytes. Default length n with DDL is 1. Default length n with CAST function is 30.	*n*+4 bytes
VARBINARY(*n*) BINARY VARYING(*n*)	Variable-length binary data of *n* bytes. *n* must be a value from 1 through 8,000. Default length *n* with DDL is 1. Default length *n* with CAST function is 30. The data entered can be zero bytes in length. The ANSI SQL synonym for VARBINARY is BINARY VARYING.	length of the data entered + 4 bytes <= *n*+4 Bytes

2.4.16.1 BINARY and VARBINARY Data Type Constants (Literals)

Binary constants have a leading 0x (a zero and the lowercase letter x) followed by the hexadecimal representation of the bit pattern, each hex digit representing four bits.

So 0x3A (or 0x3a) represents the hexadecimal value of 3A or both four-bit "nibbles".

3 represents the four bits 0011
and A represents the four bits 1010

Therefore 0x3A represents the eight-bit Byte 00111010 which is equal to decimal 58.

Examples:

```
SQL
PRINT CAST( 0x3A AS INT)
```

```
SQL (cont.)
Result
58
```

```
SQL
PRINT CAST( 58 AS BINARY(2) )

Result
0x003A
```

```
SQL
PRINT CAST( 58 AS BINARY ) -- Default length n with CAST is 30

Result
0x0000000000000000000000000000000000000000000000000000003A
```

2.4.17 `CURSOR` Data Type Details

As summarized in Table 2-43, a cursor data type contains a pointer to a cursor object, which is required for writing code that accesses a result set including more than one row of data. The cursor object in a sense contains the entire result set but only allows access to one result set row at a time.

Table 2-43 CURSOR Data Type

Data Type	Description	Storage Size
CURSOR	Pointer to a cursor	1 byte for cursor pointer, variable size for the result set

CURSOR data type is for variables or stored procedure OUTPUT parameters that contain a reference to a cursor. Any variables created with the cursor data type are nullable.

Operations that can reference variables and parameters having a **cursor** data type are as follows. Remember that the **cursor** data type cannot be used for a column in a CREATE TABLE statement.

- DECLARE @local_variable and SET @local_variable statements.
- OPEN, FETCH, CLOSE, and DEALLOCATE cursor statements as well as UPDATE and DELETE.
- Stored procedure output parameters.
- The CURSOR_STATUS function.
- The **sp_cursor_list**, **sp_describe_cursor**, **sp_describe_cursor_tables**, and **sp_describe_cursor_columns** system stored procedures.

For more in-depth treatment of cursors, see Cursors, page 638.

For a thorough coverage of using cursors see *Professional SQL Server 2000 Programming* by Robert Vieira and *Advanced Transact-SQL for SQL Server 2000* by Itzik Ben-Gan and Tom Moreau.

2.4.18 `SQL_VARIANT` Data Type Details

SQL_VARIANT data type can be assigned to a column or variable into which you can put data with different base data types. (See Table 2-44.) Each SQL_VARIANT object stores both the data value and the data type (metadata) for the SQL_VARIANT value assigned, so extra space is required for using SQL_VARIANT. ODBC does not fully support SQL_VARIANT. See Books Online under SQL_VARIANT.

Table 2-44 SQL_VARIANT Data Type

Data Type	Description	Storage Size
SQL_VARIANT	A data type that stores values of any data type except TEXT, NTEXT, IMAGE, TIMESTAMP and SQL_VARIANT.	<= 8016 bytes

2.4.18.1 Using `SQL_VARIANT`

General Value Assignment

- SQL_VARIANT objects can hold data of any SQL Server data type except TEXT, NTEXT, IMAGE, TIMESTAMP and SQL_VARIANT.
- Predicates or assignments referencing SQL_VARIANT columns can include constants of any data type.
- An SQL_VARIANT object assigned a value of NULL does not have an underlying base data type.

- When assigning the value of an SQL_VARIANT object to a non-SQL_VARIANT data object, the SQL_VARIANT value must be explicitly cast to the data type of the destination.
  ```
  SET @intvar = CAST( @variantvar AS INT )
  ```
  There are no implicit conversions from SQL_VARIANT to non-SQL_VARIANT.
- When doing arithmetic operations with an SQL_VARIANT object, the SQL_VARIANT value must be explicitly cast to the appropriate numeric data type.
- Catalog objects such as the DATALENGTH function that report the length of SQL_VARIANT objects report the length of the data only (not including the length of the meta data contained in a SQL_VARIANT object).
- 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.

SQL_VARIANT in Tables

- SQL_VARIANT columns can be used in indexes and with 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 of primary or foreign keys.
- SQL_VARIANT columns cannot be used in a computed column.
- Use ALTER TABLE to change a column to SQL_VARIANT. All existing values (of the prior data type) are converted to SQL_VARIANT values.
- ALTER TABLE cannot be used to change the data type of an SQL_VARIANT column to any other data type.

Collation

- The COLLATE clause cannot be used to assign a column collation to an SQL_VARIANT column.
- 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 itself).

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

The following example shows how to use and assign SQL_VARIANT.

Example:

```
SQL
```

```
DECLARE @intvar INT, @chvar CHAR(4)
DECLARE @vrntvar SQL_VARIANT

SET @chvar = '123'
SET @intvar = @chvar  -- implicit conversion okay
PRINT @intvar

SET @vrntvar = @chvar
PRINT CAST( @vrntvar AS CHAR )-- must cast SQL_VARIANT

-- SET @intvar = @vrntvar -- Fails, no implicit conversion

SET @intvar = CAST( @vrntvar AS INT ) -- explicit
                                      conversion- ok
PRINT @intvar
```

```
Result
```

```
123
123
123
```

2.4.18.2 Comparisons with `SQL_VARIANT` Objects

Since an SQL_VARIANT object may contain any of various data types, it is always best to explicitly cast the SQL_VARIANT object to the data type you wish to compare against when doing your own comparisons, as in a WHERE search of in an IF statement.

For situations in which you can not cast each item, such as those on the following list, special comparison rules will apply.

- Using ORDER BY, GROUP BY
- Creating Indexes
- Using MAX and MIN aggregate functions
- Using UNION (without ALL)
- Evaluating CASE expressions and using comparison operators

For SQL_VARIANT comparisons, the SQL Server data type hierarchy order is grouped into data type families from highest (top of Table 2-45) to lowest.

Table 2-45 Data Type Families

Data type	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

For comparing SQL_VARIANT objects, the data type hierarchy shown in Table 2-45 is used.When comparing two SQL_VARIANT objects of *different* data type families, the object with the family higher in the table is deemed greater (regardless of data value). When comparing two SQL_VARIANT objects of the

same data type family, both objects are implicitly converted to the higher data type and are compared based on value.

When comparing two SQL_VARIANT objects with data type of CHAR, VARCHAR, NCHAR or NVARCHAR, the comparison is based on integer comparison of the following four values in this order: LCID (locale ID), LCID version, comparison flags and sort ID.

LCID has to do with regional language settings and both LCID and LCID versions will usually be the same for all string objects in a given database. In these cases the comparison seems to behave the same as for non-SQL_VARIANT comparisons of the underlying data type values. If you can, however, it is always safer to explicitly cast and do testing of sample cases.

Examples Comparing `SQL_VARIANT` Objects Since number families are a higher family than string families, the INT value 222 would evaluate as greater than CHAR value 444.

SQL

```
DECLARE @intvariant SQL_VARIANT,@charvariant SQL_VARIANT
SET @intvariant = 222
SET @charvariant = '444'

if @intvariant > @charvariant
  PRINT '@intvariant is greater'
else
  PRINT '@charvariant is greater'
```

Result

```
@intvariant is greater
```

An explicit cast of both to INT would, of course, reverse this result.

SQL

```
DECLARE @intvariant SQL_VARIANT,@charvariant SQL_VARIANT
SET @intvariant = 222
SET @charvariant = '444'

if CAST(@intvariant AS INT) > CAST(@charvariant AS INT)
  PRINT '@intvariant is greater'
else
  PRINT '@charvariant is greater'
```

SQL (cont.)

Result

@charvariant is greater

String comparison examples. Explicit casting is suggested.

SQL

```
DECLARE @chvariant1 SQL_VARIANT
DECLARE @chvariant2 SQL_VARIANT
SET @chvariant1 = 'AAA'
SET @chvariant2 = 'MMM'

if @chvariant1  > @chvariant2
  PRINT '@chvariant1 is greater'
else
  PRINT '@chvariant2 is greater'
```

Result

@chvariant2 is greater

This example was run on a server installed as case-insensitive and behaves as hoped.

SQL

```
DECLARE @chvariant1 SQL_VARIANT
DECLARE @chvariant2 SQL_VARIANT
SET @chvariant1 = 'AAA'
SET @chvariant2 = N'aaa'        -- UNICODE

if @chvariant1 = @chvariant2
  PRINT 'They''re Equal'
else
PRINT 'They''re Not Equal'
```

Result

They’re Equal

2.4.18.3 New Function `SQL_VARIANT_PROPERTY()`

The new function for an SQL VARIANT PROPERTY is used to obtain data type and other properties about an SQL_VARIANT value.

Syntax

```
SQL_VARIANT_PROPERTY ( expression, property )
```

Arguments

expression

Expression is the input expression of type SQL_VARIANT.

property

Property is the name from the Table 2-46 of the SQL_VARIANT property for which information is requested. *property* is VARCHAR(128)

Return Type

Return Type sql_variant—see base type in Table 2-46.

Table 2-46 SQL VARIANT Properties

Value	Description	Base type of sql_variant Returned
BaseType	The SQL Server data type CHARINTMONEY NCHARNTEXTNUMERIC NVARCHARREALSMALLDATETIME SMALLINT-SMALLMONEYTEXT TIMESTAMPTINYINTVARBI-NARY UNIQUEIDENTIFIERVARCHAR	**SYSNAME** Invalid input = NULL
Precision	Precision of the numeric base data type: DATETIME = 23 SMALLDATETIME = 16 FLOAT = 53 REAL = 24 DECIMAL (p,s) and NUMERIC (p,s) = p MONEY = 19 SMALLMONEY = 10 INT = 10 SMALLINT = 5 TINYINT = 3 BIT = 1 all other types = 0	**INT** Invalid input = NULL
Scale	Scale of the numeric base data type: DECIMAL (p,s) and NUMERIC (p,s) = s MONEY and SMALLMONEY = 4 DATETIME = 3 all other types = 0	**INT** Invalid input = NULL
TotalBytes	The number of bytes required to hold both the meta data and data of the value. If the value is greater than 900, index creation will fail.	**INT** Invalid input = NULL
Collation	Represents the collation of the particular SQL_VARIANT value.	**SYSNAME** Invalid input = NULL
MaxLength	The maximum data type length, in bytes. For example, **Max-Length** of NVARCHAR**(50)** is 100, **MaxLength** of INT is 4.	**INT** Invalid input = NULL

Examples Using SQL_VARIANT_PROPERTY() Function Since number families are a higher family than string families,the INT value 222 would evaluate as greater than CHAR value 444.

SQL

```
DECLARE @chvariant1 SQL_VARIANT
DECLARE @chvariant2 SQL_VARIANT
SET @chvariant1 = 'AAA'
SET @chvariant2 = N'aaa'

SELECT SQL_VARIANT_PROPERTY( @chvariant1 , 'BaseType' )
SELECT SQL_VARIANT_PROPERTY( @chvariant2 , 'BaseType' )
```

Result

```
-------------------------------------------
varchar

-------------------------------------------
nvarchar
```

It should be noted that the PRINT operation requires an explicit CAST if used in place of SELECT in the code segment above.

SQL

```
PRINT CAST ( SQL_VARIANT_PROPERTY( @chvariant1 , 'BaseType' ) AS SYSNAME)
```

Result

```
varchar
```

2.4.18.4 SQL_VARIANT Data with Functions

These Transact-SQL functions accept SQL_VARIANT parameters and return a SQL_VARIANT value when a SQL_VARIANT parameter is input.

COALESCE	MAX	MIN	NULLIF

These functions support references to SQL_VARIANT columns or variables but 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 input parameters.

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

The CAST and CONVERT functions support SQL_VARIANT.

2.4.19 `TABLE` Data Type Details

The TABLE data type, new inMSS 2000, allows the user to declare a local variable capable of storing any table-structured data, such as a query result, set in any code and especially contain the return value from a table-valued function.

Table 2-47 TABLE Data Type

Data Type	Description
TABLE	A special data type that can be used to store a result set for later processing in the current sequence of SQL statements. It's primarily used to hold the result set of a table-valued function.

Syntax:

```
DECLARE  @local_variable    table_type_definition

table_type_definition ::=
 TABLE ( { column_definition | table_constraint } [ ,...n ] )

column_definition ::=
 column_name scalar_data_type
 [ COLLATE collation_definition ]
 [ [ DEFAULT constant_expression ] | IDENTITY [ ( seed , increment ) ] ]
 [ ROWGUIDCOL ]
 [ column_constraint ] [ ...n ]
```

```
column_constraint ::=
 { [ NULL | NOT NULL ]
 | [ PRIMARY KEY | UNIQUE ]
 | CHECK ( logical_expression )
 }

table_constraint ::=
 { { PRIMARY KEY | UNIQUE } ( column_name [ ,...n ] )
 | CHECK ( search_condition )
 }
```

Arguments

table_type_definition

table_type_definition is the same subset of information used to define a table in CREATE TABLE. The table declaration includes column definitions, names, data types and constraints. The only constraint types allowed are PRIMARY KEY, UNIQUE KEY and NULL.
See also CREATE TABLE, CREATE FUNCTION, and DECLARE. @local_variable

collation_definition

collation_definition is the collation of the column consisting of a Microsoft Windowslocale and a comparison style, a Windows locale and the binary notation or a Microsoft SQL Server collation.

Comments

- Functions and variables can be declared to be of type TABLE.
- TABLE variables can be used in functions, stored procedures and batches.
- Use TABLE variables instead of temporary TABLEs, whenever possible.

TABLE variable benefits (over temporary tables)

- A TABLE variable behaves like a local variable in that it has a well-defined scope, which is the function, stored procedure or batch in which it is declared.
- Within its scope, a TABLE variable may be used like a regular TABLE. It may be applied anywhere a TABLE or TABLE expression may be used in SELECT, INSERT, UPDATE and DELETE statements.

- TABLE variables are automatically dropped at the end of the function, stored procedure or batch in which they are defined.
- TABLE variables used in stored procedures result in fewer recompilations of the stored procedures than when temporary tables are used.
- Table variables require fewer locking and logging resources because transactions involving table variables last only for the duration of the table variable update.

TABLE variable limitations

- TABLE may not be used in the following statements:
  - INSERT INTO *table_variable* EXEC stored_procedure
  - SELECT *select_list* INTO *table_variable statements*
- Assignment operation between table variables is not supported.
- Table variables are not impacted by transaction rollbacks because they have limited scope and are not part of the persistent database.

2.4.19.1 TABLE Data Type Examples

Example:

SQL

```
DECLARE @tablevar TABLE (
   id    INT            PRIMARY KEY,
   name  VARCHAR(10)    NOT NULL,
   age   TINYINT
     CHECK( age > 0 and age < 180 ) --    Table variables may even have
                                          Check constraints
INSERT INTO @tablevar VALUES ( 1 , 'Sue' , 35)
INSERT INTO @tablevar VALUES ( 2 , 'Sam' , 25)

SELECT * FROM @tablevar
```

Result

id	nam	age
1	Sue	35
2	Sam	25

2.4.20 `TIMESTAMP` (`ROWVERSION`) Data Type Details

Table 2-48 TIMESTAMP Data Type

Data Type	Description	Storage Size
`TIMESTAMP` `ROWVERSION`	A database-wide unique number that gets updated every time a row gets updated. Automatically generated binary numbers, which are guaranteed to be unique within a database. TIMESTAMP is used typically as a mechanism for version-stamping table rows.	8 bytes
Note: ROWVERSION should always be in place of TIMESTAMP.		

Although ROWVERSION (Table 2-48) is now a data type synonym for TIMESTAMP, it should always be used in place of TIMESTAMP as the latter may completely change definition and usage in a future release of Microsoft SQL Server. Books Online says the following.

> The Transact-SQL **timestamp** data type is not the same as the **timestamp** data type defined in the ANSI SQL standard. The ANSI SQL **timestamp** data type is equivalent to the Transact-SQL **datetime** data type.
>
> A future release of Microsoft SQL Server may modify the behavior of the Transact-SQL **timestamp** data type to align it with the behavior defined in the standard. At that time, the current **timestamp** data type will be replaced with a **rowversion** data type.
>
> Microsoft SQL Server 2000 introduces a **rowversion** synonym for the **timestamp** data type. Use **rowversion** instead of **timestamp** wherever possible in DDL statements.

TIMESTAMP is as of now the SQL Server 2K base data type name. ROWVERSION is a synonym, which may be used interchangeably with the TIMESTAMP in Transact-SQL statements. It is the base data type TIMESTAMP and not the synonym that is stored and will be seen from operations such as **sp_help**. Nonetheless, in this case it is recommended to always use ROWVERSION and avoid TIMESTAMP as its definition is likely to change.

This book will use the term ROWVERSION exclusively except to note when the word TIMESTAMP has a different behavior.

2.4.20.1 Using ROWVERSION (TIMESTAMP) Data Type

A ROWVERSION (TIMESTAMP) data type column contains a database-wide unique number that gets updated every time a row gets updated. This column

can act as a version number for the row, which gives some control over optimistic locking.

Example using ROWVERSION (TIMESTAMP):

```
SQL
CREATE TABLE table1 (
       data    INT,
       rowver  ROWVERSION
    )

    INSERT INTO table1 (data) VALUES (1)
    INSERT INTO table1 (data) VALUES (2)
    INSERT INTO table1 (data) VALUES (3)

    SELECT * FROM table1
```

```
Result
data        rowver
----------- ------------------
1           0x00000000000003F3
2           0x00000000000003F4
3           0x00000000000003F5
```

Notice that the ROWEVERSION (TIMESTAMP) values under the `rowver` column have nothing to do with dates or time values but are unique integer values in the current database. When any updateable column for a row is updated, the ROWVERSION column is updated by the system to a new value as shown in the next code segment. The new value is not necessarily sequential within the table depending on what else is going on within the database. What's important is the fact that any update on a row causes the system to change its ROWVERSION value. Its use is discussed next under Optimistic Locking.

Example:

```
SQL
UPDATE table1 SET data = 20 WHERE data = 2
SELECT * FROM table1
```

SQL (cont.)

Result

```
data        rowver
----------  ------------------
1           0x00000000000003F3
20          0x00000000000003F7
3           0x00000000000003F5
```

2.4.20.2 Optimistic Locking

Generally speaking "pessimistic locking" (pessimistic concurrency) is the scheme in which exclusive locks are obtained as they are required when data is to be changed. This ensures consistency within the database, but it introduces the possibility of deadlock and has overhead due to doing the locking. It also decreases concurrency by holding exclusive (write) locks for a relatively long time.

Under optimistic locking (optimistic concurrency) schemes locking is deferred or omitted, and a check is made to see if a data value has been changed by another process between the time it was read and the time a new value is to be written. If not changed by another process, the writing may proceed. If changed by another process, the programmer can choose to abort the current attempt, try again or even offer the user the choice to overwrite the data value, abort or start over. Optimistic locking is useful in high transaction environments in which the chance of conflict on the same piece of data is small. Savings by less locking overhead and increase in concurrency can make up for the very rare conflicts that do occur.

One optimistic locking scheme is for a program to use a ROWVERSION column. The program reads the target row and releases the shared read lock immediately allowing other processes to access the row. The data value read may be used in doing some work and then calculating the new value. When the new value is ready for update, the row is again read and then, if the ROWVERSION column has not changed, the data value is updated to the new value.

This scheme is often used with cursors where the client program may obtain several rows into a cursor (read operation) releasing the lock. The user may then take several minutes to study the data and decide to change a value on one row.

Having a ROWVERSION column and declaring the cursor as OPTIMISTIC, when a row has been changed between read time and update time the system generates a 16934 error, which reads as follows: "Optimistic concurrency check failed. The row was modified outside of this cursor." Your program then refetches

the row in question and may either abort, start over or overwrite the value with your new value or present the changed data to the user for a decision.

For a further discussion of the subjects of ROWVERSION (TIMESTAMP) data type, using cursors and optimistic locking, see *Advanced Transact-SQL for SQL Server 2K* by Itzik Ben-Gan and Tom Moreau. Also see Books Online under DECLARE CURSOR and Cursor Concurrency.

Comments

- In a CREATE TABLE or ALTER TABLE statement, you do not have to supply a column name for the TIMESTAMP data type:

```
CREATE TABLE ExampleTable (
  PriKey      INT    PRIMARY KEY,
    TIMESTAMP  -- column name defaults to "timestamp"
)
```

- If you do not supply a column name, SQL Server generates a column name of TIMESTAMP. The ROWVERSION data type synonym does not follow this behavior. You must supply a column name when you specify ROWVERSION.

```
CREATE TABLE ExampleTable (
  PriKey         INT    PRIMARY KEY,
  timestamp_col  ROWVERSION  -- column name must be entered
)
```

- A table can have only one ROWVERSION column.
- The value in a ROWVERSION (TIMESTAMP) column, like an IDENTITY column, is set by the system and cannot be updated by the user via the UPDATE statement. However, the value in the ROWVERSION column is updated every time a row containing a ROWVERSION column is inserted or updated. Do not use a ROWVERSION column as a primary key and do not put an index on it because the continual changes of ROWVERSION column value cause many problems.
- The only comparison operators allowed with ROWVERSION data types are the relational operators for equality or inequality. Usually the programmer doesn't do the comparison but lets the system raise an exception if the row has had a data change.

A nonnullable ROWVERSION column is semantically equivalent to a BINARY(8) column.A nullable ROWVERSION column is semantically equivalent to a VARBINARY(8) column.

2.4.21 `UNIQUEIDENTIFIER` Data Type Details

The `UNIQUEIDENTIFIER` data type (Table 2-49) lets you manage globally unique identifiers (GUID). It is used with the NEWID() function.

Table 2-49 `UNIQUEIDENTIFIER` Data Type

Data Type	Description	Storage Size
`UNIQUEIDENTIFIER`	A globally unique identifier (GUID), which is a 16-byte binary number unique on any computer in the world. Used to hold such an identifier that must be unique throughout the entire corporate network and beyond. Used in conjunction with the NEWID() function which generates such a `UNIQUEIDENTIFIER` value.	16 bytes

The `UNIQUEIDENTIFIER` data type used with the NEWID() function is similar to an integer data type with the IDENTITY property, although the latter just guarantees uniqueness within the table.

2.4.21.1 Using `UNIQUEIDENTIFIER`

- `UNIQUEIDENTIFIER` objects
  - may be compared using the relational operators (=, <>, <, >, <=, >=)
  - may be checked for `NULL` (`IS NULL` and `IS NOT NULL`)
  - allow no other arithmetic operators
- All column constraints and properties except `IDENTITY` are allowed on the `UNIQUEIDENTIFIER` data type.
- Multiple columns within a table may be assigned as `UNIQUEIDENTIFIER` data type.
- Declaring a column as `UNIQUEIDENTIFIER` data type does not preclude manually inserting the same value again.
- To make the values unique, it is suggested that the column also be specified as `PRIMARY KEY` and always be given a new value using the `NEWID()` function.

Suggested usage of `UNIQUEIDENTIFIER` To have a column unique within the table and worldwide, declare the column as

columnname UNIQUEIDENTIFIER PRIMARY KEY DEFAULT NEWID()

The `NEWID()` Function The function generates a unique value of type `UNIQUEIDENTIFIER` each time it's called.

SQL

```
DECLARE @uid UNIQUEIDENTIFIER
SET @uid = NEWID()
PRINT '@uid is:  '+ CONVERT(varchar(255), @uid)
```

Result

```
@uid is:  C24922A8-51B6-40DA-B53B-E40A81516C60
```

The values generated by `NEWID()` are not sequential from one call to the next but instead have a random appearance.

Example of a table using `UNIQUEIDENTIFIER` as Primary Key To generate a new `UNIQUEIDENTIFIER` for each new row, give a DEFAULT as the `NEWID()` function.

Example:

SQL

```
CREATE TABLE table1 (
  uid    UNIQUEIDENTIFIER  PRIMARY KEY  DEFAULT NEWID(),
  data   INT
)

INSERT INTO table1 (data) VALUES (1)
INSERT INTO table1 (data) VALUES (2)
INSERT INTO table1 (data) VALUES (3)

SELECT * FROM table1
```

Result

```
uid                                  data
-----------------------------------  -----------
00516380-0291-4B90-A113-C10B92F2622B  1
64A88B51-1BCC-4FE0-81E4-69BC65A3E957  2
FF8BD9CB-8793-4E87-80F0-1AF46036C288  3
```

The UNIQUEIDENTIFIER value is certainly bulky and awkward to work with, but when you need a world-wide globally unique identifier, it fills the bill.

Example:

SQL

```
SELECT * FROM table1
WHERE uid = '64A88B51-1BCC-4FE0-81E4-69BC65A3E957'
```

Result

uid	data
64A88B51-1BCC-4FE0-81E4-69BC65A3E957	2

2.4.21.2 The ROWGUIDCOL Property

The ROWGUIDCOL column property is primarily used by SQL Server replication, but otherwise it does not seem to add much value.

- ROWGUIDCOL property can be assigned to only one column in a table, and that must be a UNIQUEIDENTIFIER data type.
- The table may contain other UNIQUEIDENTIFIER columns.
- ROWGUIDCOL property can only be assigned to a UNIQUEIDENTIFIER column, but neither ROWGUIDCOL property nor the UNIQUEIDENTIFIER data type ensures uniqueness within the table. So either a UNIQUE or PRIMARY KEY constraint (recommended) is still required to get that result.
- The OBJECTPROPERTY function can be used to determine if a table has a ROWGUIDCOL column, and the COLUMNPROPERTY function can be used to determine the name of the column.
- A column declared with the ROWGUIDCOL property can be referenced in a SELECT list either by the word ROWGUIDCOL or by the column name itself. This is similar to using the IDENTITYCOL keyword to reference an IDENTITY column.

Examples with UNIQUEIDENTIFIER and ROWGUIDCOL Property

SQL

```
CREATE TABLE table2 (
uid1    UNIQUEIDENTIFIER  ROWGUIDCOL ,
uid2    UNIQUEIDENTIFIER
)

INSERT INTO table2 (uid1,uid2) VALUES ( NEWID() , NEWID() )

SELECT * FROM table2
```

Result

```
B63E88-1B19-42E9-BADF-814CE00656A0      5E30BAC7-FEF6-4217-BEA9-ED78C247E273
```

Re-insert the same values to demonstrate that neither column has a uniqueness constraint.

SQL

```
INSERT INTO table2 (uid1,uid2) VALUES
      ( 'A7B63E88-1B19-42E9-BADF-814CE00656A0' ,
       '5E30BAC7-FEF6-4217-BEA9-ED78C247E273'
      )

SELECT uid1, uid2 FROM table2
```

Result

```
uid1                                    uid2
--------------------------------------  --------------------------------------
A7B63E88-1B19-42E9-BADF-814CE00656A0    5E30BAC7-FEF6-4217-BEA9-ED78C247E273
A7B63E88-1B19-42E9-BADF-814CE00656A0    5E30BAC7-FEF6-4217-BEA9-ED78C247E273
```

SQL

```
SELECT ROWGUIDCOL FROM table2
```

SQL (cont.)

```
SELECT ROWGUIDCOL FROM table2
```

Result

```
uid1
------------------------------------
A7B63E88-1B19-42E9-BADF-814CE00656A0
A7B63E88-1B19-42E9-BADF-814CE00656A0
```

2.4.21.3 `UNIQUEIDENTIFIER` Data Type Constants (Literals)

For completeness, the two ways to specify a `UNIQUEIDENTIFIER` constant are shown here, although typically the `NEWID()` function is always used to generate a new `UNIQUEIDENTIFIER` value.

- Character string format

  'FF8BD9CB-8793-4E87-80F0-1AF46036C288'

  '6F9619FF-8B86-D011-B42D-00C04FC964FF'

- Binary format

  0xff19966f868b11d0b42d00c04fc964ff

  0x46463842443943422D383739332D 344538372D383046302D314146343630

You almost never enter your own `UNIQUEIDENTIFIER` value as a constant because the `UNIQUEIDENTIFIER` column even with `ROWGUIDCOL` property does impose a uniqueness constraint; therefore, entering a duplicate value would not be detected. But if you use the `NEWID()` function, then uniqueness of the new value is guaranteed.

2.5 USER-DEFINED DATA TYPES

User-defined data types are data types based on intrinsic system types given a name by the user, which may then be used in future DDL statements within the database where they were created.

It is particularly useful to create a user-defined data type for a unique key that has foreign key columns (usually in another table) that refer to it. Creating a user-defined type and using it for the unique key and foreign key column definitions ensures that they are the same data type.

The following parts of user-defined data types must be provided when created.

- **Name.**
- **Underlying System Data Type**—see Table 2-50.
- **Nullability**—'NULL', 'NOT NULL', 'NONULL' in single quotes, see below. For a discussion of default nullability see page 230.

2.5.1 Enterprise Manager—Create and Manage a User-Defined Data Type

- **Create**
  1. Expand the console tree to the database in which you want the new type.
  2. Right click on **User-Defined Types** and select **New User-Defined Data Type**.
  3. Enter the desired values.
- **Drop, Copy, Rename, Properties or Generate Script**
  4. Expand the console tree to the database and **User Defined Types**.
  5. Right click on the user-defined data type and select the operation desired.

2.5.2 Transact-SQL—Create and Manage a User-Defined Data Type

Four phrases can be used to create and manage a user-defined data type:

- **sp_addtype:** Creates a user-defined data type in the current database.
- **sp_droptype:** Deletes a user-defined data type from the current database.
- **sp_rename:** Changes the name of the user-created object.
- **sp_help** <*typename*>: Displays the definition of the user-defined data type (or system data type).

sp_addtype The addtype phrase creates a user-defined data type in the current database.

Syntax

```
sp_addtype [ @typename = ] type,
[ @phystype = ] system_data_type
[ , [ @nulltype = ]{ 'NULL' | 'NOT NULL' | 'NONULL' }  ]
[ , [ @owner = ] 'owner_name' ] -- defaults to current user
```

Arguments

[@typename =] type

This is the name of the user-defined data type to be created; it must be unique in the database.

[@phystype =] system_data_type

This is the physical SQL Server data type being defined as the underlying or base data type. It has no default and must be one of the values given in Table 2-50.

Table 2-50 Base Data Type Values

'BINARY(*n*)'	IMAGE	SMALLDATETIME
BIT	INT	SMALLINT
'CHAR(*n*)'	'NCHAR(*n*)'	TEXT
DATETIME	NTEXT	TINYINT
DECIMAL	NUMERIC	UNIQUEIDENTIFIER
'DECIMAL[(*p* [, *s*])]'	'NUMERIC[(*p* [, *s*])]'	'VARBINARY(*n*)'
FLOAT	'NVARCHAR(*n*)'	'VARCHAR(*n*)'
'FLOAT(*n*)'	REAL	
Quotation marks are required if there are embedded blank spaces or punctuation marks including parenthesis, (), or square brackets, [].		

```
[ , [ @nulltype = ]  { 'NULL' | 'NOT NULL' | 'NONULL' }  ]
```

If not specified in **sp_addtype** the nullability is set to the current default nullability for the database as can be seen with the GETANSINULL() system function and which can be changed using SET or ALTER DATABASE.

I suggest that nullability be explicitly specified in **sp_addtype**. If specified in **sp_addtype,** the setting becomes the default nullability for the user-defined data type but can be set to a different value as with CREATE TABLE or ALTER TABLE.

For a discussion of nullability see page 230.

[@owner =] '*owner_name*'

This specifies the owner or creator of the new data type. When not specified, *owner_name* is the current user.

Return Code Values

0 (success) or 1 (failure)

Comments

Note that the main features that can be set in a user-defined data type are the

- base data type
- size
- nullability
- owner

Constraints to limit permissible values cannot be assigned as in the ANSI SQL notion of domain. Nonetheless, user-defined data type is a useful concept to improve readability and consistency among related tables using foreign keys.

Permissions

Execute permissions default to the public role.

sp_droptype The droptype phrase deletes a user-defined data type from the current database.

Syntax

```
sp_droptype [ @typename = ] 'type'
```

Arguments

```
[@typename =] type
```

This is the name of the user-defined data type to be dropped.

sp_rename The rename function changes the name of the user-created object.

Syntax

```
sp_rename [ @objname = ] 'object_name' -- Current name of the type to be renamed
, [ @newname = ] 'new_name' -- New name of the type
[ , [ @objtype = ] 'object_type' ]  -- USERDATATYPE for a user-defined data
                                       type

sp_rename [ @objname = ] 'object_name' ,
[ @newname = ] 'new_name'
[ , [ @objtype = ] 'object_type' ]  -- USERDATATYPE for a user-defined data
                                       type
```