

Day 3: Working with Columns

Yesterday, you took your first step in understanding the unique language of database access. Learning how to filter and sort the data returned by a `SELECT` statement provides you with the skills to get data from your database. Of course, this has only scratched the surface of what you can do with a `SELECT` statement.

In today's lesson, you will extend your knowledge of the `SELECT` statement by working with the columns in the database tables. We will cover a lot of information today, so prepare yourself. As you read this, you might ask yourself, "What will this do for me?" The answer to that question might not become clear until later in this book. Much of this lesson will help you throughout your career as a SQL programmer. The topics that will be discussed today are

- Data types
- Column manipulation
- The `CONVERT` and `CAST` functions
- Arithmetic and string operators
- String functions
- Date functions
- Numeric functions
- System functions
- The `CASE` statement

Column Characteristics

When a column is added to a database table, you set several parameters that define the characteristics of that column. These include the name of the column, its data type, its length, and a default value. In this section, you will learn all about data types, what they are and how you should use them. In addition, you will see how empty or null columns are treated by the SQL processor.

Data Types

In a database, every column, variable, expression, and parameter has a related data type associated with it. A *data type* is an attribute that specifies the type of data (integer, string, date, and so on) that object can contain. All the data that a specific column holds must be of the same data type. A data type also determines how the data for a particular column is accessed, indexed, and physically stored on the server.

Tables 3.1 and 3.2 describe each of the available data types in Microsoft SQL Server 2000 and

provide an example of what each might be used for. If the data you are working with is of different lengths, such as names, addresses, and other text, you should use variable-length data types. Fixed-length data types are best used for data, such as phone numbers, Social Security numbers, and ZIP Codes.

Table 3.1 Data Types in SQL Server 2000

Long Name	Syntax	Example	Description
Variable character	<code>varchar(6)</code>	"John"	Variable-length character fields are best for most strings.
Character	<code>char(6)</code>	"John"	Fixed-length character fields are best for most strings.
National variable characters	<code>nvarchar(6)</code>	"John"	Variable-length Unicode data with a maximum length of 4,000 characters.
Datetime	<code>datetime</code>	Jan 1, 2000 12:15:00.000 pm	Datetime fields are used for precise storage of dates and times. Datetimes can range from Jan 1, 1753 to Dec 31, 9999. Values outside this range must be stored as character.
Small datetime	<code>smalldatetime</code>	Jan 2, 2000 12:15pm	Small datetimes are half the size of datetimes. They use increments of one minute and represent dates from Jan 1, 1900 to Jun 6, 2079.
Precise decimal	<code>decimal(4,2)</code> or <code>numeric(4,2)</code>	13.22	Decimal/numeric data types store fractional numerics precisely. The first parameter specifies how many digits are allowed in the field. The second parameter specifies how many digits may come after the decimal. In this example, I could represent numbers from –99.99 to 99.99.
Big floating point	<code>float(15)</code>	64023.0134	Floating-point numbers are not guaranteed to be stored precisely. SQL Server rounds up numbers that binary math can't handle. Floats take a parameter specifying the total number of digits.
Little float	<code>real(7)</code>	16.3452	Half the size of a float; the same rules apply.
Integer	<code>int</code>	683423	Integers are four bytes wide and store numbers between plus or minus two billion.
Small integer	<code>smallint</code>	12331	Small integers are half the size of integers, ranging from –32,768 through 32,767.
Tiny integer	<code>tinyint</code>	5	Tiny integers are half again the size of small integers, a single byte, and may not be negative. Values run from 0 to 255. Perfect for an age column.

Bit	bit	1	Bits are the smallest data type available today. They are one bit in size, one-eighth of a byte. Bits may not be null and can have a value of 0 or 1. This is the actual language of all computers.
Binary	binary	0x00223FE2...	Fixed-length binary data with a maximum length of 8,000 bytes.
Money	money	\$753.1132	Money types range from +/- 922 trillion. Money types store four digits to the right of the decimal and are stored as fixed-point integers.
Small money	smallmoney	\$32.50	Small money can handle about +/- \$214,000, with four digits to the right of the decimal. Half the size of Money.
Text	text	"We the people..."	Text fields can be up to 2GB in size. Text fields are treated as Binary Large Objects (BLOBs) and are subject to a great many limitations. They cannot be used in an ORDER BY, indexed, or grouped, and handling the inside an application program takes some extra work. (BLOBs will be discussed on Day 21, "Handling BLOBs in T-SQL.")
Image	image	0x00223FE2...	Image data can be used to store any type of binary data, including images (gif, jpg, and so on), executables, or anything else you can store on your disk drive. Images are also BLOBs, subject to the same limitations.

Table 3.2 New Data Types in SQL Server 2000

Long Name	Use	Example	Description
Big integer	bigint	983422348	A large integer that can hold a number +/- 2 raised to the 63rd power. Twice as large as an integer.
Sql_variant	sql_variant		A data type that stores values of other supported data types, except text, ntext, and sql_variant. You can use this to hold data from any other data type without having to know the data type in advance.
Table	table		This is a special data type that can be used to store a result set for later processing. It is primarily used for temporary storage of a set of rows.

Data Type Precedence

When two expressions of different data types are combined using one or more operators or functions, the data type precedence rules specify which data type is converted to the other. The data type with

the lower precedence is converted to the data type with the higher precedence. If the conversion is not a supported implicit conversion, an error is returned. If both expressions have the same data type, the resulting object has the same data type. The order of precedence for the data types are shown in Table 3.3.

Table 3.3 Data Type Order of Precedence

Precedence Number	Data Type
1	sql_variant
2	datetime
3	smalldatetime
4	float
5	real
6	decimal
7	money
8	smallmoney
9	bigint
10	int
11	smallint
12	tinyint
13	bit
14	ntext
15	image
16	timestamp
17	uniqueidentifier
18	nvarchar
19	nchar
20	varchar
21	char
22	varbinary
23	binary

An example of the precedence can be seen when you add two unlike numbers together as shown here:

```
Select quantity * price as Sale_Total from "Order Details"
```

The quantity column is an integer data type, whereas the price column is a money data type. When this calculation is performed, the result would be a money data type.

Note - Don't worry about how to use the multiplication operator, or what the `as Sale_Total` means. We will cover these issues later in this lesson.

Using Null Data

When a column is empty, it is treated differently than when a column is blank or zero. These might sound the same to you, but to the computer, they are very different. A blank is an actual character that takes a position in the column. Of course, zero is its own explanation. A null means that the column actually contains nothing. To see how a null is displayed, try executing the following SQL statement in the Query Analyzer:

```
use pubs
select title_id, advance
from titles
order by title_id
```

Results:

```
title_id  advance
-----  -
BU1032    5000.0000
BU1111    5000.0000
BU2075    10125.0000
BU7832    5000.0000
MC2222     .0000
MC3021    15000.0000
MC3026    NULL
PC1035    7000.0000
PC8888    8000.0000
PC9999    NULL
...
TC4203    4000.0000
TC7777    8000.0000

(18 row(s) affected)
```

You should see that null values are displayed using the string `NULL`. However, most standard comparisons will not recognize the null value. The next example shows you what happens when I add a `where` clause to the `SELECT` statement. I am looking for all title IDs where the advance amount is less than \$5,000. You might think that 'null' or 'empty' are identical, but they're not. Although you will be covering functions later in this lesson, I want to cover one in this section.

The `isnull()` function permits a way to include null values in aggregate calculations. The function requires two arguments and its syntax is shown here:

```
Isnull(<expression>, <value>)
```

The first argument is the expression on which the calculation will be performed; usually this is a column from the database. The second argument is the value that will replace the null value for display or calculation purposes. If the expression contains a null, the second parameter is returned by this function. If the expression is not null, the value in the expression is returned.

The following SQL query shows the effect of the `isnull()` function on the `titles` table in the `pubs` database.

```
use pubs
select title_id, price, isnull(price, $45)
from titles
order by price
```

The output of this query shows which prices were null:

```
title_id price
-----
MC3026    NULL          45.0000
PC9999    NULL          45.0000
MC3021    2.9900        2.9900
BU2075    2.9900        2.9900
PS2106    7.0000        7.0000
...
PS3333    19.9900       19.9900
PC8888    20.0000       20.0000
TC3218    20.9500       20.9500
PS1372    21.5900       21.5900
PC1035    22.9500       22.9500

(18 row(s) affected)
```

As you can see in the output, there are two titles in the table that have null in their `price` column. When the `isnull()` function evaluates those columns, it returns \$45 for the third column in the result set. For the other columns, it simply returns the value of the column.

You can use the `isnull()` function inside an aggregate function. If you want to know the average price of a book, you would ask for the `avg(price)` on the `titles` table. However, the two books that have null for a price would be excluded from the calculation. Suppose that you know that those books will be priced at \$29 each. You could use the `isnull()` function to include those books in the calculation. The example shows you the query without the `isnull()` function and then with the `isnull()` function.

```
select avg(price)
from titles
```

Results:

```
-----
14.7662

(1 row(s) affected)

Warning: Null value is eliminated by an aggregate or other SET operation.
```

As you can see, the server displays a warning message informing you that there were null values found and they were not used.

```
select avg(isnull(price, $29))
from titles
```

Results:

```
-----
16.3477
(1 row(s) affected)
```

In the second version of the query, you can see that the average returned is different because the two books with nulls were included in the calculation. The `isnull()` function doesn't change the value of the row in the table; it only assumes a value for the purposes of a single query.

Changing Result Sets

So far, every column you have requested has been retrieved and displayed as the data was stored. However, there will be many times when you will need to change the way the data is displayed. This next section will discuss the different ways that you can change the labels of the displayed columns or modify the data in the columns by using operators and functions.

Using Column Aliases

You will find that the names of the columns as defined in the database are not exactly easy to read as shown in the following example:

```
use pubs
select emp_id, job_id, job_lvl, pub_id
from employee
```

Results:

emp_id	job_id	job_lvl	pub_id
PMA42628M	13	35	0877
PSA89086M	14	89	1389
VPA30890F	6	140	0877
H-B39728F	12	35	0877
L-B31947F	7	120	0877
F-C16315M	4	227	9952

As you can see, the column names are not very descriptive. SQL provides you with the ability to change the names of the columns in the `SELECT` statement to make it easier for you to understand the output. To change a column name, you would use the following syntax:

```
SELECT <column name> as <new name> FROM <table>
```

Using the previous SQL example, I could change it to make the output more readable as shown in the next example.

```
use pubs
select emp_id as 'Employee Number',
       job_id as 'Job Number',
       job_lvl as 'Job Pay Level',
       pub_id as 'Publisher ID'
```

```
from employee
```

Results:

```
Employee Number Job Number Job Pay Level Publisher ID
-----
PMA42628M      13    35    0877
PSA89086M      14    89   1389
VPA30890F       6   140    0877
H-B39728F      12    35    0877
```

You can also see that I used single quotes to define the new column names that include blanks in the SQL statement.

Tip - Using column aliases is one of the best ways of creating easy to understand T-SQL scripts, especially long, complex scripts.

Using String Constants

Now that you have seen how to change the names of the columns that you select, you will see how to add new columns of text to your output that are not included in the database. String constants enable you to add labels and comments to your output easily. You can use a string constant in your SQL statement in the same way you would reference a column. The following example shows how to add a constant to your SQL query:

```
use northwind
select 'Hi! My Name is: ', FirstName
from employees
```

Results:

```

                FirstName
-----
Hi! My Name is: Nancy
Hi! My Name is: Andrew
Hi! My Name is: Janet
Hi! My Name is: Margaret
Hi! My Name is: Steven
Hi! My Name is: Michael
Hi! My Name is: Robert
Hi! My Name is: Laura
Hi! My Name is: Anne

(9 row(s) affected)
```

You can see from the output that there are two columns being displayed. The first column has the words, `Hi! My name is:` for every row in the `Employees` table. The second column contains the first name of each employee. Adding these strings to your SQL enables you to produce more readable output. In fact, you will see in the next section how you can manipulate strings even further using operators and functions.

Using Operators

All programming languages, including T-SQL, contain a set of operators that enable you to specify an action to be performed on one or more expressions. These operators fall into several different categories, which are listed in Table 3.4.

Table 3.4 Available Operators in T-SQL

Category	Operator	Description
Arithmetic	+ (Add)	Addition.
	- (Subtract)	Subtraction.
	* (Multiply)	Multiplication.
	/ (Divide)	Division.
	% (Modulo)	Returns the integer remainder of a division.
Bitwise	&	Bitwise AND.
		Bitwise OR.
	^	Bitwise exclusive OR.
Comparison	=	Equal to.
	>	Greater than.
	<	Less than.
	>=	Greater than or equal to.
	<=	Less than or equal to.
	<>	Not equal to.
Logical		Tests for the truth of some specified condition. All the logical operators return a Boolean data type with a value of TRUE or FALSE. These operators are ALL, AND, ANY, BETWEEN, EXISTS, IN, LIKE, NOT, OR, and SOME.
Unary	+ (Positive)	Positive number.
	- (Negative)	Negative number.
	~ (Bitwise NOT)	Returns the ones complement of the number.
Assignment	= (Equal sign)	Used to assign a value to a variable.
String Concatenation	+ (Concatenation)	Appends two strings together to form one string.

In the next section, you will see how to manipulate the columns you request in a SQL statement.

Using the Addition Operator

If we want to add two or more numbers together, we can use the addition sign. This will add the

numbers together and display the single result as shown in the following example:

```
use northwind
select UnitPrice, Discount,
       UnitPrice + Discount as Total
from "order details"
```

Results:

UnitPrice	Discount	Total
14.0000	0.0	14.0
9.8000	0.0	9.8000002
34.8000	0.0	34.799999
42.4000	0.15000001	42.550003
16.8000	0.15000001	16.949999
16.8000	0.0	16.799999

Note - You can see how using an alias here helps to make the output more readable.

You can use any of the arithmetic operators in this fashion.

What about string values? If you want to add two or more strings together, you can use the addition operator for this as well. When used between two strings, the addition operator enables you to concatenate the strings (that is, to add the second string on to the end of the first). This functionality gives you the ability to create new strings using the data in multiple columns.

This next example requires both the string and the first name from the database table to appear as the column called `Introduction`. To make the output pretty, you have to add an extra space in the constant string. Otherwise, the first names of all the employees will be pushed up against the word `is`.

```
use northwind
select 'Hello, my name is ' + firstname as Introduction
from employees
```

Results:

```
Introduction
-----
Hello, my name is Nancy
Hello, my name is Andrew
Hello, my name is Janet
Hello, my name is Margaret
Hello, my name is Steven
Hello, my name is Michael
Hello, my name is Robert
Hello, my name is Laura
Hello, my name is Anne

(9 row(s) affected)
```

Although this looks a lot like the example using string constants earlier in this section, in this query

we retrieved the static string plus the data in a single column instead of two columns. Then we used the + sign to concatenate the strings.

Manipulating Multiple Columns and Using Constants

Besides adding a constant to one column value, you can use constants to manipulate multiple columns to display information in standard format styles. A common example of this is when you need to print usernames. In the `Employees` table, the first name is stored in the `FirstName` column and the last name in `LastName`. Suppose that you wanted a list of employee names, listed with last name first, first name last, and separated by a comma. You also want to call this column `Employee Names`. The following example shows how to perform this request:

```
use northwind
select lastname + ', ' + firstname as 'Employee Names'
from employees
Order by "employee names"
```

Results:

```
Employee Names
-----
Buchanan, Steven
Callahan, Laura
Davolio, Nancy
Dodsworth, Anne
Fuller, Andrew
King, Robert
Leverling, Janet
Peacock, Margaret
Suyama, Michael

(9 row(s) affected)
```

Using the + operator with strings enables you to put multiple columns and constants together in a single column of the result set. I used an alias to name the column, and then I used the alias in the `ORDER BY` clause.

Tip - I could have sorted this last example by using the original column names as shown:

```
use northwind
select lastname + ', ' + firstname as 'Employee Names'
from employeesOrder by lastname, firstname
```

When you ask for an ordering to occur on a calculated result, such as the concatenated string, the server first builds the result set and then sorts it.

Using Functions

You have briefly used a couple of functions in yesterday's lesson on filtering dates. In this next section, you will learn about the different types of functions available to you in SQL Server 2000. There are four types of built-in functions that you can use in T-SQL. The types of functions are

- Mathematical
- String
- System
- Niladic

I will provide an example of each of these functions in the next section. *Niladic* functions work exclusively with the `INSERT` statement, which will be covered on Day 7, "Adding, Changing, and Deleting Rows." They include functions such as `CURRENT_TIMESTAMP()` to insert the current date and time automatically.

Note - Whenever a function is performed on a column, the server cannot use an index to resolve a query based on that column. To answer your query in the least amount of time, the server must act based on information available to it before the query runs. Because a function will take some action on a column, the server can't know what the data will be until after the query runs. Using functions in the `SELECT` list is okay, but using them in the `WHERE` clause can cause performance problems.

The purpose of this section is to provide you with a list of examples in order to understand how each set of functions are used in real-world situations. In each of the following groups, you will see how to use several of the more useful functions available. Remember that there are more functions than what I will cover in this section.

Note - As you work with these functions, remember that you can use them with columns you are retrieving, or you can use them in the `WHERE` clause to modify the data you are comparing.

Using String Functions

String functions enable you to manipulate character data from the database. These functions perform an operation on a string value and then return either a string or numeric value, depending on the type of function being used.

`UPPER()` and `LOWER()`

These two functions are the direct opposites of each other. The `UPPER()` function will return a character expression with any lowercase characters converted to uppercase. The `LOWER()` function will return a character expression with any uppercase characters converted to lowercase. The following example shows how to use the `UPPER()` function. Remember that the syntax for the `LOWER()` function is exactly the same.

```

use northwind
select firstname,
       upper(firstname) as "All Caps"
from employees
where lastname = 'Fuller'

```

Results:

```

firstname  All Caps
-----
Andrew     ANDREW

(1 row(s) affected)

```

SUBSTRING() and RIGHT()

The `SUBSTRING()` function enables you to extract a string from anywhere in a larger string, whereas the `RIGHT()` function returns a string from the end of another string. There is no `LEFT()` function. Both functions are useful for manipulating strings in columns where several fields have been concatenated into a single column. Although this is a bad database design practice, this type of data often occurs in the real world.

Suppose that you have a table of product information where a column contains a combination of two unique values, as we can see in the `title_id` column in the `titles` table of the `pubs` database. The first two positions in the column represent the book type and the rest of the column represents the unique book number. You might need to separate these values out in order to work with them independently. To do this, you could use the `SUBSTRING()` or `RIGHT()` function as shown in the following example:

```

use pubs
select title_id,
       type,
       substring(title_id,1,2) as Book_Type_Code
from titles

```

Results:

title_id	type	Book_Type_Code
BU1032	business	BU
BU1111	business	BU
BU2075	business	BU
BU7832	business	BU
MC2222	mod_cook	MC
MC3021	mod_cook	MC
MC3026	UNDECIDED	MC
PC1035	popular_comp	PC
PC8888	popular_comp	PC
PC9999	popular_comp	PC
PS1372	psychology	PS
PS2091	psychology	PS
PS2106	psychology	PS
PS3333	psychology	PS
PS7777	psychology	PS
TC3218	trad_cook	TC
TC4203	trad_cook	TC

```
TC7777 trad_cook TC
(18 row(s) affected)
```

`SUBSTRING()` takes three arguments: the source string, where to start (1 is the first character), and how many characters to take. The `RIGHT()` function takes two arguments: the source string and how many characters to take. The syntax for each follows:

```
SUBSTRING(source, start, length)
RIGHT(source, start)
```

RTRIM and LTRIM

These two functions trim any extra blanks off the beginning of a string (`LTRIM`) or the end of a string (`RTRIM`). There is no function that performs both at the same time. However, you can nest functions, so that if I were inserting some data into a table, and that data included blanks at both the beginning and end, I could trim off the extra blanks.

This example doesn't insert anything into the database table (that is covered in Day 7). Instead, this is how you might trim the blanks off the front and back of a column at the same time:

```
Select ltrim(rtrim(title))
From Titles
```

Both of these functions accept a single parameter, which is the string to be trimmed.

STR

The `STR()` function will convert numeric data into string data. There is no `VAL()` function to convert strings to numbers as you might have seen in Visual Basic, but the generic `CONVERT()` function will accomplish both goals. The `STR()` function accepts three parameters: source number, length of the new string to be created, and number of digits to the right of the decimal. The following example shows how to use this function:

```
use pubs
select 'The price is ' + str(price, 6,3)
from titles
```

Results:

```
-----
The price is 19.990
The price is 11.950
The price is 2.990
...
The price is 7.990
The price is 20.950
The price is 11.950
The price is 14.990

(18 row(s) affected)
```

Caution - If you try to execute the preceding SQL query without using the `STR` function,

you will receive the following error from the server:

```
Server: Msg 260, Level 16, State 1, Line 2
Disallowed implicit conversion from data type
varchar to data type money, table 'pubs.dbo.titles', column 'price'.Use t
```

As you can see from the message, the server is telling you to use the `CONVERT()` function instead of the `STR()` function. The `CONVERT()` function will convert numeric data to a string, but it won't add insignificant zeroes or line up all the decimal points.

The syntax for the `STR()` function is

```
STR(<source number>, [length of output], [scale])
```

The `STR()` function is useful for converting numeric data in a way that looks good when displayed.

ASCII() and CHAR()

These functions convert a single character into its ASCII code or an ASCII code into its character. If you pass more than one character to the `ASCII` function, it will return the code for only the first letter. The following example shows both the `ASCII()` and `CHAR()` functions in one query:

```
select ascii('B'), char(66)
```

```
----- ----
66    B
```

```
(1 row(s) affected)
```

The `CHAR()` function can be useful for inserting keycodes that cannot be easily typed in, such as the Tab and CR/LF codes.

PATINDEX() and CHARINDEX()

`PATINDEX()` returns the first occurrence of a string inside a larger string. You can use this function to search a `BLOB` text field. Wildcards are available for use in the pattern field. `CHARINDEX()` works the same way, but wildcard characters are not permitted, and it cannot be used to search text types.

The `titles` table contains the text field `notes`. The query that follows will search for books that have a note containing the word `Computer` in it, and return the character position where that word appears.

```
select title_id,
       patindex('%Computer%', notes) as 'Starts at'
from titles
where patindex('%Computer%', notes) <> 0
```

Results:

```
title_id Starts at
-----
BU7832    28
```

```
PC8888    45
PC9999    17
PS1372    94
PS7777    97
```

```
(5 row(s) affected)
```

As you can see, the `PATINDEX()` function was used in two places in the query. The use in the `WHERE` clause searches through all the text data looking for the word `Computer`. If a match is found, the title ID is returned along with the starting position of the word. The syntax for both of these functions is the same:

```
PATINDEX(<pattern>, <source string>)
CHARINDEX(<pattern>, <source string>)
```

The `CAST` and `CONVERT` Functions

To combine columns and constants, they must be of the same underlying data type. The server will convert `varchar` and `char` columns without your help. However, if you tried a query where you wanted a numeric and character type in the same column, you would have a problem, as shown in the following example:

```
use pubs
select title_id + ' is priced at $' + price
from titles
```

Results:

```
Server: Msg 260, Level 16, State 1, Line 2
Disallowed implicit conversion from data type
varchar to data type money, table 'pubs.dbo.titles',
column 'price'.
Use the CONVERT function to run this query.
```

To overcome this, as the error message suggests, you must use the `CONVERT` function. There are actually two functions that you could use: the `CONVERT()` and `CAST()` functions. These functions provide similar functionality in T-SQL.

The `CAST()` function will change the data type of the expression being passed to it. It will not allow you to specify the style or length of the new expression. The syntax for the `CAST()` function is

```
CAST (expression AS data_type)
```

The `CONVERT()` function, besides allowing you to specify the new data type as the `CAST()` function does, also enables you to specify the length and style of the new expression.

Both functions are useful for many more applications than the example. You can also convert character data to numeric data for use in mathematical calculations. To perform the concatenation in the current example, you must convert the `price` column (datatype `money`) into a variable character format:

```
use pubs
select title_id +
```

```

    ' is priced at $' +
    convert(varchar(10), price)
from titles

```

Results:

```

-----
BU1032 is priced at $19.99
BU1111 is priced at $11.95
BU2075 is priced at $2.99
...
TC3218 is priced at $20.95
TC4203 is priced at $11.95
TC7777 is priced at $14.99

(18 row(s) affected)

```

In this example, I converted the `price` column, containing small money data, into a variable character. When a data type is converted to a variable character, the length of the resulting string is only as long as it needs to be, up to the maximum specified in the function call. In this case, I asked for a `varchar(10)`. So, if the price were 132.55, the converted `varchar` string would be exactly six characters long.

However, if I had converted the price to a fixed-length character string (a `char(10)`), the total string length would have been the maximum size of 10 characters no matter how many characters were needed. The data is left justified, and spaces are added on the end to round out the remainder of the field.

In either case, if the price data were too large to fit in the maximum length, the server would place an asterisk in the character field to indicate that it didn't have enough room to convert the data. If you convert a long string to a shorter string, the data is truncated, meaning that only as much of the string as fits in the new length will be converted; the rest is thrown away.

Note - If you do not specify a length for data types, such as `char` and `varchar` fields, the server will default to the data type specific value. In the case of `char` and `varchar` types, this default is 30 characters.

In addition to converting data types, the `CONVERT()` function also helps you format date fields in several different ways. By default, when you select a `datetime` value, the output would be displayed as

```
Oct 6 2000 3:00PM
```

As you know, `datetime` values are stored out to the millisecond. This presents you with some pretty ugly output. The `CONVERT()` function can be used to request some alternative date formats. When using the `CONVERT()` to format the `datetime` values, you should always convert the data to a `char` or `varchar` data type. Table 3.5 summarizes the different date formats that are usable with the `CONVERT()` function.

Table 3.5 Date Formats Available with `CONVERT()`

Code	Description
0	(default style) mon dd yyyy hh:mmAM
1	mm/dd/yy
2	yy.mm.dd
3	dd/mm/yy
4	dd.mm.yy
5	dd-mm-yy
6	dd mon yy
7	mon dd, yy
8	hh:mm:ss
9	mon dd yyyy hh:mm:ss:mmmAM
10	mm-dd-yy
11	yy/mm/dd
12	yymmdd
13	dd mon yyyy hh:mm:ss:mmmm (military time)
14	hh:mm:ss:mmm (military time)

Tip - These codes display the year without a century prefix. To get a century prefix, add 100 to the code. Note that styles 9 and 13 always provide a four-digit year.

Other String Functions

Other string functions are available for you to use in T-SQL; I have listed them in Table 3.6 as a reference.

Table 3.6 Other Available String Functions

Function	Description
SPACE(<int>)	Returns a string of <i>n</i> spaces.
REVERSE(<source string>)	Reverses the source string (that is, "Ben" becomes "neB"). Not sure what you would use this for.
REPLICATE (<pattern>, <int>)	Returns a string with the pattern repeated <i>n</i> times.
SOUNDEX(<source string>)	Returns a four-digit soundex code to represent the phonemes in the source string. This function would probably be used when searching for a string that sounds like..., as in a case when you are looking for a

	phone number. If you wanted to find the number for someone named Smith, the soundex code would match for the following names: Smythe, Smyth, Sumith, and Smith. The function knocks out any vowels in the string and uses the first letter to form its code.
DIFFERENCE (<i><char1></i> , <i><char2></i>)	Evaluates the soundex difference between <i>char1</i> and <i>char2</i> on a scale of 0 to 4, where 4 is an exact match and 0 is a complete mismatch.

Using Arithmetic Functions

Most math functions are used in very specific applications having something to do with engineering. However, there are times when you might need to use one in some more standard application areas. What follows is an accounting example. The function being used is

`SIGN(numeric)` -Returns 1 if the number is positive, 0 if 0, and -1 if the number is negative.

In this example, you have a table with two columns in it: `CustID` and `InvAmt`. The `InvAmt` column contains negative values for the amounts that you owe (credits) and positive values for amounts that are owed to you (debits). This query will display a report with four columns, labeled `CustID`, `Debit`, `Credit`, and `Invoice Amount`.

```
select CustID,
       InvAmt * ($0.5 *(sign(InvAmt) + 1)) as 'Debit',
       InvAmt * ($0.5 *(sign(InvAmt) - 1)) as 'Credit',
       InvAmt
from invoice
```

Note - This example uses a table that I have in my personal database; it will not work on a standard installation of SQL Server 2000.

Results:

CustID	Debit	Credit	InvAmt
1	0.00	500.00	-500.00
1	0.00	250.00	-250.00
2	75.00	0.00	75.00
3	35.00	35.00	35.00

This query uses some math functions to arrive at a clever answer to the problem of identifying which values are debits and which are credits.

Let's start from the inside out. First, get the sign of the data. For the `Debit` column, I want data that is positive. So, take the sign and add 1 to it. Positive values go to 2, negative values to 0, and zeroes to 1.

Now, multiply the result by .5. By using .5, the server thinks I want a float result, but if I multiply by 50 cents, the data type remains a `smallmoney`. Positive numbers go back to 1 ($2 * .5 = 1$). Negative numbers go to zero because zero times anything is zero and zeroes go to zero ($1 * .5 = 0$). Now,

multiply this product by the data. Only positive numbers retain a non-zero value.

In the `Credit` column, we start off the same way. This time, we are subtracting 1 from the sign. If you follow the logic through, this means only negative numbers retain a non-zero value.

Using Date Functions

Only nine date functions are available in T-SQL. Table 3.7 lists each of them along with a description.

Table 3.7 Date Functions

Function	Description
<code>DateAdd</code>	Adds a specified number of dateparts to the date
<code>DateDiff</code>	Calculates the difference between two dates
<code>DatePart</code>	Extracts the specified datepart from the date
<code>DateName</code>	Returns a part of the date as a string
<code>Day</code>	Returns the day number of the date
<code>GetDate</code>	Returns the current date and time of the computer
<code>Month</code>	Returns the month number of the date
<code>Year</code>	Returns the year of the date
<code>GetUTCDate</code>	Returns the current UTC (universal time, GMT) date and time

The first four date functions in Table 3.7 take an argument (the first argument) to tell it on what part of the date to operate. Table 3.8 shows the different date parts available for use in the date functions.

Table 3.8 Datepart Codes Used in Date Functions

Name	Usage	Sample Values
Year	yy	1999
Quarter	qq	1, 2, 3, 4
Month	mm	1–12
Day of year	dy	1–365 (366 in leap years)
Day	dd	1–31
Week	wk	1–53
Weekday	dw	1–7, where Sunday is

		1
Hour	hh	0–23, midnight is 0
Minute	mi	0–59
Second	ss	0–59
Millisecond	ms	0–999

Refer to this table if you have any questions about these codes while you are working with the examples in this section. When providing arguments to these functions, do not use quotation marks around the datepart codes, but if you are specifying a constant date, use quotation marks around the codes.

GETDATE()

This is one function that you will find yourself using quite often, in many different applications. The `GETDATE()` function returns the current date and time as a `datetime` value. The following example shows only one of the many uses of this function:

```
Select GETDATE()
```

Results:

```
-----
2000-10-06 20:48:10.500
```

This function is most often used in stored procedures and triggers to test the validity of an operation. For example, an inserted row might be checked to see whether a due date is five days from now or if it has passed.

DATEADD()

To get the date that is 30 days from today, you would first need to use the `GETDATE()` function to get today's date. You would then use `DATEADD()` to add the 30 days. The function would then return the date that is 30 days later. The syntax of the functions is

```
DATEADD(<datepart>, <increment>, <source>)
```

The example shown adds 30 days to today.

```
Select Dateadd(dd,30,getdate())
```

Note - You should note the use of the `dd` as the datepart to request that the addition be done in days. If you used `mm` accidentally, you would have added 30 months to today instead.

Results:

```
-----
2000-11-05 20:53:44.670
(1 row(s) affected)
```

Note - The `DATEADD()` function takes into account the different days in a month and even a leap year when doing its calculations.

DATEDIFF()

This function calculates the difference between `date1` and `date2`, based on the datepart specified. The syntax of this function is

```
DATEDIFF(<datepart>, <date1>, <date2>)
```

The following example shows the difference between today and New Year's Eve:

```
Select getdate(), datediff(dd, getdate(), '12/31/2000')
```

Results:

```
-----
2000-10-06 20:58:03.483      86
(1 row(s) affected)
```

If the second date is earlier than the first date, you will receive a negative number as the result.

DATEPART()

The `DATEPART()` function returns an integer that represents a specified datepart of the source date. The following query sums the total number of books sold in June, regardless of the year:

```
use pubs
select sum(qty) as 'June Sales'
from sales
where datepart(mm,ord_date) = 6
```

Results:

```
June Sales
-----
80
(1 row(s) affected)
```

DATENAME()

This function works the same way as the `DATEPART()` function, except that it returns a string instead of an integer. It is most useful when you need to get the days of the week or months of the year. The

following example shows how to get the day of the week from today's date:

```
Select 'Today is ' + datename(dw, getdate()) + '.'
```

Results:

```
-----
Today is Friday.
```

GETUTCDATE()

The `GETUTCDATE()` function returns the datetime value representing the current UTC time (Universal Time Coordinate or Greenwich mean time). The current UTC time is derived from the current local time and the time zone setting in the operating system of the computer on which SQL Server is running. The example shows the current time and the corresponding UTC time.

```
Select getdate() as 'Today', getutcdate() as 'UTC'
```

Results:

```
Today                UTC
-----
2000-10-06 21:10:51.067    2000-10-07 01:10:51.067
```

DAY(), MONTH(), YEAR()

These three functions return an integer that represents the day, month, or year part of a specified date. The following example shows all three functions at once:

```
SELECT DAY(getdate()) as 'day',
       MONTH(getdate()) as 'month',
       YEAR(getdate()) as 'year'
```

Results:

```
day      month      year
-----
6        10        2000
```

```
(1 row(s) affected)
```

Using the CASE Statement

The last topic we will discuss today is the `CASE` statement. The `CASE` expression provides great performance and allows me to design very fast SQL code. The `CASE` expression compares a specified column in the `SELECT` statement against a list of possibilities and returns different results depending on which `WHEN` expression in the `CASE` statement is matched.

There are two ways that you can use the `CASE` keyword. The first way is the simplest; it enables you to take different actions against the same expression. The second method, which is called a searched `CASE` expression, enables you to specify a Boolean expression and to take actions on that expression depending on whether it is true or false.

The syntax for the `CASE` statement is as follows:

```
CASE input_expression
  WHEN when_expression THEN result_expression
  [...n]
  [
  ELSE else_result_expression
  ]
END
```

Using a Simple `CASE` Statement

A simple `CASE` statement uses a single column in the `CASE` and returns an expression based on the contents in the column. The following example uses the `CASE` function to alter the display of book categories to make them more understandable:

```
use pubs
SELECT Category =
  CASE type
    WHEN 'popular_comp' THEN 'Popular Computing'
    WHEN 'mod_cook' THEN 'Modern Cooking'
    WHEN 'business' THEN 'Business'
    WHEN 'psychology' THEN 'Psychology'
    WHEN 'trad_cook' THEN 'Traditional Cooking'
    ELSE 'Not yet categorized'
  END,
  CAST(title AS varchar(25)) AS 'Shortened Title',
  price AS Price
FROM titles
WHERE price IS NOT NULL
ORDER BY type, price
```

Results:

Category	Shortened Title	Price
Business	You Can Combat Computer S	2.9900
Business	Cooking with Computers: S	11.9500
Business	The Busy Executive's Data	19.9900
Business	Straight Talk About Compu	19.9900
Modern Cooking	The Gourmet Microwave	2.9900
...		
Psychology	Computer Phobic AND Non-P	21.5900
Traditional	Cooking Fifty Years in Buckingham	11.9500
Traditional	Cooking Sushi, Anyone?	14.9900
Traditional	Cooking Onions, Leeks, and Garlic	20.9500

(16 row(s) affected)

The `CASE` expression takes the place of a single column in an otherwise quite ordinary query. For each row in the `titles` table, the `CASE` expression is evaluated. The order of the `WHEN` expressions is significant: If a row satisfies the Boolean `WHEN` expression, the code following the `WHEN'S THEN` clause is performed, and the next row is evaluated. If the row matches none of the `WHEN` clauses, the `ELSE` is evaluated.

The `CASE` expression has five keywords associated with it: The `CASE` keyword starts the statement;

`WHEN` presents the Boolean expression to test; `THEN` precedes the expression return value that the `CASE` expression represents for a particular row; `ELSE` gives an expression that is returned when none of the `WHEN` conditions is true; the `END` keyword marks the end of the `CASE` statement. After a `CASE` statement satisfies an expression, the remaining `CASE` statements in the grouping are skipped.

Using a Searched `CASE` Statement

This type of `CASE` statement is basically the same as the simple `CASE`, except that the `WHEN` expression contains a complete Boolean expression as shown in the following example:

```
use pubs
select lname, job_lvl,
       case
         when job_lvl<100 then 'Dear Fellow Worker'
         when job_lvl<200 then 'Dear Honored Guest'
         else 'Dear Executive'
       end as 'Greeting'
from employee
order by Greeting, lname
```

As you can see, the `WHEN` keyword is followed by a Boolean expression which includes a column name, operator, and value to be tested.

Summary

Well, you made it through the day. Today, you learned how to manipulate data using operators, functions, constants, and the `CASE` statement. In addition, you learned how to create aliases for columns in the `SELECT` statement. These techniques will serve you well throughout the rest of this book and throughout your career in T-SQL programming.

You also learned how to manipulate and search date values. You explored how to use the `CONVERT` and `CAST` functions to change the data type of a value from one type to another, and to format dates differently.

Finally, you discovered how to use the `CASE` expression to evaluate the values in selected columns and return a different value based on those contents. If you find yourself asking, "How do I do that in SQL?", remember the `CASE` expression.

Q&A

Q. If the server won't convert my data type to hold fractions, what happens when I get the sum of something too big to be held?

A. The server is smart enough to promote the data type to the next larger data type to hold the result of an aggregate or calculation.

Q. Functions look really interesting, but it seems like a lot of work to type in all that stuff just for a simple result set.

A. Functions are frequently used in the production of stored procedures and triggers. They enable

you to provide consistent code for particular actions that will be used in many places in an application.

Workshop

The Workshop provides quiz questions to help you solidify your understanding of the material covered and exercises to provide you with experience in using what you've learned. Try to answer the quiz and exercise questions before checking the answers in Appendix A, "Answers to Quizzes and Exercises," and make sure you understand the answers before continuing to the next lesson.

Quiz

1. What function enables you to provide a value for nulls?
2. What is the result of this query?

```
Use Pubs
Select 'My name is ' + au_fname
From authorsWhere au_id = '172-32-1176'
```

3. What column name is displayed by this query?

```
Use Pubs
Select au_fname as 'First Name'From authors
```

4. What would be the result of the following query?

```
Use Pubs
Select 'My price is ' + price
From titlesWhere title like '%Computer%'
```

Exercises

1. When was the first employee hired by Northwind?
2. What is today's date and what day of the week is it?
3. How many hours until New Year's Day?

© Copyright Pearson Education. All rights reserved.