Chapter 4

SQL Components

In This Chapter

- SQL's Basic Objects
- Data Types
- Transact-SQL Functions
- Scalar Operators
- NULL Values
This chapter introduces the elementary objects and basic operators supported by the Transact-SQL language. First, the basic language elements, including constants, identifiers, and delimiters, are described. Then, because every elementary object has a corresponding data type, data types are discussed in detail. Additionally, all existing operators and functions are explained. At the end of the chapter, NULL values are introduced.

SQL’s Basic Objects

The language of the Database Engine, Transact-SQL, has the same basic features as other common programming languages:

- Literal values (also called constants)
- Delimiters
- Comments
- Identifiers
- Reserved keywords

The following sections describe these features.

Literal Values

A literal value is an alphanumerical, hexadecimal, or numeric constant. A string constant contains one or more characters of the character set enclosed in two single straight quotation marks ("'") or double straight quotation marks (" "). (Single quotation marks are preferred due to the multiple uses of double quotation marks, as discussed in a moment.) If you want to include a single quotation mark within a string delimited by single quotation marks, use two consecutive single quotation marks within the string. Hexadecimal constants are used to represent nonprintable characters and other binary data. Each hexadecimal constant begins with the characters '0x' followed by an even number of characters or numbers. Examples 4.1 and 4.2 illustrate some valid and invalid string constants and hexadecimal constants.

**Example 4.1**

Some valid string constants and hexadecimal constants follow:

- 'Philadelphia'
- "Berkeley, CA 94710"
- '9876'
- 'Apostrophe is displayed like this: can’’t' (note the two consecutive single quotation marks)
- 0x53514C0D
EXAMPLE 4.2

The following are not string constants:

'AB'C' (odd number of single quotation marks)

'New York'' (same type of quotation mark—single or double—must be used at each end of the string)

The numeric constants include all integer, fixed-point, and floating-point values with and without signs (see Example 4.3).

EXAMPLE 4.3

The following are numeric constants:

130

–130.00

–0.357E5 (scientific notation—\(nE^m\) means \(n\) multiplied by \(10^m\))

22.3E-3

A constant always has a data type and a length, and both depend on the format of the constant. Additionally, every numeric constant has a precision and a scale factor. (The data types of the different kinds of literal values are explained later in this chapter.)

Delimiters

In Transact-SQL, double quotation marks have two meanings. In addition to enclosing strings, double quotation marks can also be used as delimiters for so-called delimited identifiers. Delimited identifiers are a special kind of identifier usually used to allow the use of reserved keywords as identifiers and also to allow spaces in the names of database objects.

NOTE

Differentiation between single and double quotation marks was first introduced in the SQL92 standard. In the case of identifiers, this standard differentiates between regular and delimited identifiers. Two key differences are that delimited identifiers are enclosed in double quotation marks and are case sensitive. (Transact-SQL also supports the use of square brackets instead of double quotation marks.) Double quotation marks are used only for delimiting strings. Generally, delimited identifiers were introduced to allow the specification of identifiers, which are otherwise identical to reserved keywords. Specifically, delimited identifiers protect you from using names (identifiers and variable names) that could be introduced as reserved keywords in one of the future SQL standards. Also, delimited identifiers may contain characters that are normally illegal within identifier names, such as blanks.
In Transact-SQL, the use of double quotation marks is defined using the QUOTED_IDENTIFIER option of the SET statement. If this option is set to ON, which is the default value, an identifier in double quotation marks will be defined as a delimited identifier. In this case, double quotation marks cannot be used for delimiting strings.

**Comments**

There are two different ways to specify a comment in a Transact-SQL statement. Using the pair of characters /* and */ marks the enclosed text as a comment. In this case, the comment may extend over several lines. Furthermore, the characters -- (two hyphens) indicate that the remainder of the current line is a comment. (The two -- comply with the ANSI SQL standard, while /* and */ are the extensions of Transact-SQL.)

**Identifiers**

In Transact-SQL, identifiers are used to identify database objects such as databases, tables, and indices. They are represented by character strings that may include up to 128 characters and may contain letters, numerals, or the following characters: _, @, #, and $. Each name must begin with a letter or one of the following characters: _, @, or #. The character # at the beginning of a table or stored procedure name denotes a temporary object, while @ at the beginning of a name denotes a variable. As indicated earlier, these rules don't apply to delimited identifiers (also known as quoted identifiers), which can contain, or begin with, any character (other than the delimiters themselves).

**Reserved Keywords**

Each programming language has a set of names with reserved meanings, which must be written and used in the defined format. Names of this kind are called reserved keywords. Transact-SQL uses a variety of such names, which, as in many other programming languages, cannot be used as object names, unless the objects are specified as delimited or quoted identifiers.

**NOTE**

In Transact-SQL, the names of all data types and system functions, such as CHARACTER and INTEGER, are not reserved keywords. Therefore, they can be used to denote objects. (Do not use data types and system functions as object names! Such use makes Transact-SQL statements difficult to read and understand.)
Data Types

All the data values of a column must be of the same data type. (The only exception specifies the values of the SQL_VARIANT data type.) Transact-SQL uses different data types, which can be categorized as follows:

- Numeric data types
- Character data types
- Temporal (date and/or time) data types
- Miscellaneous data types

The following sections describe all these categories.

Numeric Data Types

Numeric data types are used to represent numbers. The following table shows the list of all numeric data types:

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTEGER</td>
<td>Represents integer values that can be stored in 4 bytes. The range of values is $-2,147,483,648$ to $2,147,483,647$. INT is the short form for INTEGER.</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>Represents integer values that can be stored in 2 bytes. The range of values is $-32768$ to $32767$.</td>
</tr>
<tr>
<td>TINYINT</td>
<td>Represents nonnegative integer values that can be stored in 1 byte. The range of values is $0$ to $255$.</td>
</tr>
<tr>
<td>BIGINT</td>
<td>Represents integer values that can be stored in 8 bytes. The range of values is $-2^{63}$ to $2^{63} - 1$.</td>
</tr>
<tr>
<td>DECIMAL(p,[s])</td>
<td>Describes fixed-point values. The argument $p$ (precision) specifies the total number of digits with assumed decimal point $s$ (scale) digits from the right. DECIMAL values are stored, depending on the value of $p$, in 5 to 17 bytes. DEC is the short form for DECIMAL.</td>
</tr>
<tr>
<td>NUMERIC(p,[s])</td>
<td>Synonym for DECIMAL.</td>
</tr>
<tr>
<td>REAL</td>
<td>Used for floating-point values. The range of positive values is approximately $2.23E - 308$ to $1.79E + 308$, and the range of negative values is approximately $-1.18E - 38$ to $-1.18E + 38$ (the value zero can also be stored).</td>
</tr>
<tr>
<td>FLOAT[(p)]</td>
<td>Represents floating point values, like REAL. $p$ defines the precision, with $p &lt; 25$ as single precision (stored in 4 bytes) and $p \geq 25$ as double precision (stored in 8 bytes).</td>
</tr>
<tr>
<td>MONEY</td>
<td>Used for representing monetary values. MONEY values correspond to 8-byte DECIMAL values and are rounded to four digits after the decimal point.</td>
</tr>
<tr>
<td>SMALLMONEY</td>
<td>Corresponds to the data type MONEY but is stored in 4 bytes.</td>
</tr>
</tbody>
</table>
Character Data Types

There are two general forms of character data types. They can be strings of single-byte characters or strings of Unicode characters. (Unicode uses several bytes to specify one character.) Further, strings can have fixed or variable length. The following character data types are used:

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAR[(n)]</td>
<td>Represents a fixed-length string of single-byte characters, where n is the number of characters inside the string. The maximum value of n is 8000. CHARACTER(n) is an additional equivalent form for CHAR(n). If n is omitted, the length of the string is assumed to be 1.</td>
</tr>
<tr>
<td>VARCHAR[(n)]</td>
<td>Describes a variable-length string of single-byte characters (0 &lt; n ≤ 8000). In contrast to the CHAR data type, the values for the VARCHAR data type are stored in their actual length. This data type has two synonyms: CHAR VARYING and CHARACTER VARYING.</td>
</tr>
<tr>
<td>NCHAR[(n)]</td>
<td>Stores fixed-length strings of Unicode characters. The main difference between the CHAR and NCHAR data types is that each character of the NCHAR data type is stored in 2 bytes, while each character of the CHAR data type uses 1 byte of storage space. Therefore, the maximum number of characters in a column of the NCHAR data type is 4000.</td>
</tr>
<tr>
<td>NVARCHAR[(n)]</td>
<td>Stores variable-length strings of Unicode characters. The main difference between the VARCHAR and the NVARCHAR data types is that each NVARCHAR character is stored in 2 bytes, while each VARCHAR character uses 1 byte of storage space. The maximum number of characters in a column of the NVARCHAR data type is 4000.</td>
</tr>
</tbody>
</table>

**NOTE**

The VARCHAR data type is identical to the CHAR data type except for one difference: if the content of a CHAR(n) string is shorter than n characters, the rest of the string is padded with blanks. (A value of the VARCHAR data type is always stored in its actual length.)

Temporal Data Types

Transact-SQL supports the following temporal data types:

- DATETIME
- SMALLDATETIME
- DATE
- TIME
- DATETIME2
- DATETIMEOFFSET
The DATETIME and SMALLDATETIME data types specify a date and time, with each value being stored as an integer value in 4 bytes or 2 bytes, respectively. Values of DATETIME and SMALLDATETIME are stored internally as two separate numeric values. The date value of DATETIME is stored in the range 01/01/1753 to 12/31/9999. The analog value of SMALLDATETIME is stored in the range 01/01/1900 to 06/06/2079. The time component is stored in the second 4-byte (or 2-byte for SMALLDATETIME) field as the number of three-hundredths of a second (DATETIME) or minutes (SMALLDATETIME) that have passed since midnight.

The use of DATETIME and SMALLDATETIME is rather inconvenient if you want to store only the date part or time part. For this reason, SQL Server introduced the data types DATE and TIME, which store just the DATE or TIME component of a DATETIME, respectively. The DATE data type is stored in 3 bytes and has the range 01/01/0001 to 12/31/9999. The TIME data type is stored in 3–5 bytes and has an accuracy of 100 nanoseconds (ns).

The DATETIME2 data type stores high-precision date and time data. The data type can be defined for variable lengths depending on the requirement. (The storage size is 6–8 bytes). The accuracy of the time part is 100 ns. This data type isn't aware of Daylight Saving Time.

All the temporal data types described thus far don't have support for the time zone. The data type called DATETIMEOFFSET has the time zone offset portion. For this reason, it is stored in 6–8 bytes. (All other properties of this data type are analogous to the corresponding properties of DATETIME2.)

The date value in Transact-SQL is by default specified as a string in a format like 'mmm dd yyyy' (e.g., 'Jan 10 1993') inside two single quotation marks or double quotation marks. (Note that the relative order of month, day, and year can be controlled by the SET DATEFORMAT statement. Additionally, the system recognizes numeric month values with delimiters of / or –.) Similarly, the time value is specified in the format 'hh:mm' and the Database Engine uses 24-hour time (23:24, for instance).

NOTE

Transact-SQL supports a variety of input formats for DATETIME values. As you already know, both objects are identified separately; thus, date and time values can be specified in any order or alone. If one of the values is omitted, the system uses the default values. (The default value for time is 12:00 AM.)

Examples 4.4 and 4.5 show the different ways date and time values can be written using the different formats.
EXAMPLE 4.4

The following date descriptions can be used:

'28/5/1959' (with SET DATEFORMAT dmy)
'May 28, 1959'
'1959 MAY 28'

EXAMPLE 4.5

The following time expressions can be used:

'8:45 AM'
'4 pm'

Miscellaneous Data Types

Transact-SQL supports several data types that do not belong to any of the data type groups described previously:

- Binary data types
- BIT
- Large object data types
- CURSOR (discussed in Chapter 8)
- UNIQUEIDENTIFIER
- SQL_VARIANT
- TABLE (discussed in Chapters 5 and 8)
- XML (discussed in Chapter 26)
- Spatial (e.g., GEOGRAPHY and GEOMETRY) data types (discussed in Chapter 27)
- HIERARCHYID
- TIMESTAMP
- User-defined data types (discussed in Chapter 5)

The following sections describe each of these data types (other than those designated as being discussed in another chapter).
Binary and BIT Data Types

BINARY and VARBINARY are the two binary data types. They describe data objects being represented in the internal format of the system. They are used to store bit strings. For this reason, the values are entered using hexadecimal numbers.

The values of the BIT data type are stored in a single bit. Therefore, up to 8 bit columns are stored in 1 byte. The following table summarizes the properties of these data types:

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>BINARY[(n)]</td>
<td>Specifies a bit string of fixed length with exactly ( n ) bytes (0 &lt; n ≤ 8000).</td>
</tr>
<tr>
<td>VARBINARY[(n)]</td>
<td>Specifies a bit string of variable length with up to ( n ) bytes (0 &lt; n ≤ 8000).</td>
</tr>
<tr>
<td>BIT</td>
<td>Used for specifying the Boolean data type with three possible values: FALSE, TRUE, and NULL.</td>
</tr>
</tbody>
</table>

Large Object Data Types

Large objects (LOBs) are data objects with the maximum length of 2GB. These objects are generally used to store large text data and to load modules and audio/video files. Transact-SQL supports the following LOB data types:

- VARCHAR(max)
- NVARCHAR(max)
- VARBINARY(max)

Starting with SQL Server 2005, you can use the same programming model to access values of standard data types and LOBs. In other words, you can use convenient system functions and string operators to work with LOBs.

The Database Engine uses the max parameter with the data types VARCHAR, NVARCHAR, and VARBINARY to define variable-length columns. When you use max by default (instead of an explicit value), the system analyzes the length of the particular string and decides whether to store the string as a convenient value or as a LOB. The max parameter indicates that the size of column values can reach the maximum LOB size of the current system.

Although the database system decides how a LOB will be stored, you can override this default specification using the sp_tableoption system procedure with the LARGE_VALUE_TYPES_OUT_OF_ROW option. If the option's value is set to 1, the data in columns declared using the max parameter will be stored separately from all other data. If this option is set to 0, the Database Engine stores all values for the row size < 8060 bytes as regular row data.
Since SQL Server 2008, you can apply the new FILESTREAM attribute to a VARBINARY(max) column to store large binary data directly in the NTFS file system. The main advantage of this attribute is that the size of the corresponding LOB is limited only by the file system volume size. (This storage attribute will be described in the upcoming “Storage Options” section.)

**UNIQUEIDENTIFIER Data Type**

As its name implies, a value of the UNIQUEIDENTIFIER data type is a unique identification number stored as a 16-byte binary string. This data type is closely related to the globally unique identifier (GUID), which guarantees uniqueness worldwide. Hence, using this data type, you can uniquely identify data and objects in distributed systems.

The initialization of a column or a variable of the UNIQUEIDENTIFIER type can be provided using the functions NEWID and NEWSEQUENTIALID, as well as with a string constant written in a special form using hexadecimal digits and hyphens. (The functions NEWID and NEWSEQUENTIALID are described in the section “System Functions” later in this chapter.)

A column of the UNIQUEIDENTIFIER data type can be referenced using the keyword ROWGUIDCOL in a query to specify that the column contains ID values. (This keyword does not generate any values.) A table can have several columns of the UNIQUEIDENTIFIER type, but only one of them can have the ROWGUIDCOL keyword.

**SQL_VARIANT Data Type**

The SQL_VARIANT data type can be used to store values of various data types at the same time, such as numeric values, strings, and date values. (The only types of values that cannot be stored are TIMESTAMP values.) Each value of an SQL_VARIANT column has two parts: the data value and the information that describes the value. (This information contains all properties of the actual data type of the value, such as length, scale, and precision.)

Transact-SQL supports the SQL_VARIANT_PROPERTY function, which displays the attached information for each value of an SQL_VARIANT column. For the use of the SQL_VARIANT data type, see Example 5.5 in Chapter 5.

**NOTE**

Declare a column of a table using the SQL_VARIANT data type only if it is really necessary. A column should have this data type if its values may be of different types or if determining the type of a column during the database design process is not possible.
HIERARCHYID Data Type
The HIERARCHYID data type is used to store an entire hierarchy. (For instance, you can use this data type to store a hierarchy of all employees or a hierarchy of all folder lists.) It is implemented as a Common Language Runtime (CLR) user-defined type that comprises several system functions for creating and operating on hierarchy nodes. The following functions, among others, belong to the methods of this data type: GetLevel(), GetAncestor(), GetDescendant(), Read(), and Write(). (The detailed description of this data type is outside the scope of this book.)

TIMESTAMP Data Type
The TIMESTAMP data type specifies a column being defined as VARBINARY(8) or BINARY(8), depending on nullability of the column. The system maintains a current value (not a date or time) for each database, which it increments whenever any row with a TIMESTAMP column is inserted or updated and sets the_TIMESTAMP_ column to that value. Thus, TIMESTAMP columns can be used to determine the relative time at which rows were last changed. (ROWVERSION is a synonym for TIMESTAMP.)

**NOTE**
The value stored in a TIMESTAMP column isn't important by itself. This column is usually used to detect whether a specific row has been changed since the last time it was accessed.

Storage Options
Since SQL Server 2008, there are two different storage options, each of which allows you to store LOBs and to save storage space:

- FILESTREAM
- Sparse columns

The following subsections describe these options.

FILESTREAM Storage
As you already know, SQL Server supports the storage of LOBs using the VARBINARY(max) data type. The property of this data type is that binary large objects (BLOBs) are stored inside the database. This solution can cause performance problems if the stored files are very large, as in the case of video or audio files. In that case, it is common to store such files outside the database, in external files.
The FILESTREAM storage option supports the management of LOBs, which are stored in the NTFS file system. The main advantage of this type of storage is that the database system manages data, although the data is stored outside the database. Therefore, this storage type has the following properties:

- You use the CREATE TABLE statement to store FILESTREAM data and use the data modification statements (SELECT, INSERT, UPDATE, and DELETE) to query and update such data.
- The database system assures the same level of security for FILESTREAM data as for relational data stored inside the database.

The creation of FILESTREAM data will be described in detail in Chapter 5.

Sparse Columns
The aim of sparse columns as a storage option is quite different from the FILESTREAM storage support. Whereas FILESTREAM is Microsoft’s solution for the storage of LOBs outside the database, sparse columns help to minimize data storage space. These columns provide an optimized way to store column values, which are predominantly NULL. (NULL values are described at the end of this chapter.) If you use sparse columns, NULL values require no disk space, but on the other side, non-NULL data needs an additional 2 to 4 bytes, depending on the data type of the non-NULL values. For this reason, Microsoft recommends using sparse columns only when the overall storage space savings will be at least 20 percent.

You specify and access sparse columns in the same way as you specify and access all other columns of a table. This means that the statements SELECT, INSERT, UPDATE, and DELETE can be used to access sparse columns in the same way as you use them for usual columns. (These four SQL statements are described in detail in Chapters 6 and 7.) The only difference is in relation to creation of a sparse column: you use the SPARSE option (after the column name) to specify that a particular column is a sparse column: col_name data_type SPARSE.

If a table has several sparse columns, you can group them in a column set. Therefore, a column set is an alternative way to store and access all sparse columns in a table. For more information concerning column sets, see Books Online.

Transact-SQL Functions
Transact-SQL functions can be either aggregate functions or scalar functions. The following sections describe these function types.
Aggregate Functions

*Aggregate functions* are applied to a group of data values from a column. Aggregate functions always return a single value. Transact-SQL supports several groups of aggregate functions:

- Convenient aggregate functions
- Statistical aggregate functions
- User-defined aggregate functions
- Analytic aggregate functions

Statistical and analytic aggregate functions are discussed in Chapter 23. User-defined aggregates are beyond the scope of this book. That leaves the convenient aggregate functions, described next:

- **AVG**  Calculates the arithmetic mean (average) of the data values contained within a column. The column must contain numeric values.
- **MAX** and **MIN**  Calculate the maximum and minimum data value of the column, respectively. The column can contain numeric, string, and date/time values.
- **SUM**  Calculates the total of all data values in a column. The column must contain numeric values.
- **COUNT**  Calculates the number of (non-null) data values in a column. The only aggregate function that is not applied to columns is COUNT(*). This function returns the number of rows (whether or not particular columns have NULL values).
- **COUNT_BIG**  Analogous to COUNT, the only difference being that COUNT_BIG returns a value of the BIGINT data type.

The use of convenient aggregate functions with the SELECT statement can be found in Chapter 6.

Scalar Functions

In addition to aggregate functions, Transact-SQL provides several scalar functions that are used in the construction of scalar expressions. (A scalar function operates on a single
value or list of values, whereas an aggregate function operates on the data from multiple rows.) Scalar functions can be categorized as follows:

- Numeric functions
- Date functions
- String functions
- System functions
- Metadata functions

The following sections describe these function types.

**Numeric Functions**

Numeric functions within Transact-SQL are mathematical functions for modifying numeric values. The following numeric functions are available:

<table>
<thead>
<tr>
<th>Function</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABS(n)</td>
<td>Returns the absolute value (i.e., negative values are returned as positive) of the numeric expression n. Example: SELECT ABS(–5.767) = 5.767, SELECT ABS(6.384) = 6.384</td>
</tr>
<tr>
<td>ACOS(n)</td>
<td>Calculates arc cosine of n. n and the resulting value belong to the FLOAT data type.</td>
</tr>
<tr>
<td>ASIN(n)</td>
<td>Calculates the arc sine of n. n and the resulting value belong to the FLOAT data type.</td>
</tr>
<tr>
<td>ATAN(n)</td>
<td>Calculates the arc tangent of n. n and the resulting value belong to the FLOAT data type.</td>
</tr>
<tr>
<td>ATN2(n,m)</td>
<td>Calculates the arc tangent of n/m. n, m, and the resulting value belong to the FLOAT data type.</td>
</tr>
<tr>
<td>CEILING(n)</td>
<td>Returns the smallest integer value greater than or equal to the specified parameter. Examples: SELECT CEILING(4.88) = 5, SELECT CEILING(–4.88) = –4</td>
</tr>
<tr>
<td>COS(n)</td>
<td>Calculates the cosine of n. n and the resulting value belong to the FLOAT data type.</td>
</tr>
<tr>
<td>COT(n)</td>
<td>Calculates the cotangent of n. n and the resulting value belong to the FLOAT data type.</td>
</tr>
<tr>
<td>DEGREES(n)</td>
<td>Converts radians to degrees. Examples: SELECT DEGREES(PI()/2) = 90, SELECT DEGREES(0.75) = 42</td>
</tr>
<tr>
<td>EXP(n)</td>
<td>Calculates the value e^n. Example: SELECT EXP(1) = 2.7183</td>
</tr>
<tr>
<td>Function</td>
<td>Explanation</td>
</tr>
<tr>
<td>------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>FLOOR(n)</td>
<td>Calculates the largest integer value less than or equal to the specified value ( n ). Example: ( ) ( \text{SELECT FLOOR}(4.88) = 4 ) ( )</td>
</tr>
<tr>
<td>LOG(n)</td>
<td>Calculates the natural (i.e., base e) logarithm of ( n ). Examples: ( \text{SELECT LOG}(4.67) = 1.54 ) ( ) ( \text{SELECT LOG}(0.12) = -2.12 ) ( )</td>
</tr>
<tr>
<td>LOG10(n)</td>
<td>Calculates the logarithm (base 10) for ( n ). Examples: ( \text{SELECT LOG10}(4.67) = 0.67 ) ( ) ( \text{SELECT LOG10}(0.12) = -0.92 ) ( )</td>
</tr>
<tr>
<td>PI()</td>
<td>Returns the value of the number pi (3.14). ( )</td>
</tr>
<tr>
<td>POWER(x,y)</td>
<td>Calculates the value ( x^y ). Examples: ( \text{SELECT POWER}(3.12,5) = 295.65 ) ( \text{SELECT POWER}(81,0.5) = 9 ) ( )</td>
</tr>
<tr>
<td>RADIANS(n)</td>
<td>Converts degrees to radians. Examples: ( \text{SELECT RADIANS}(90.0) = 1.57 ) ( \text{SELECT RADIANS}(42.97) = 0.75 ) ( )</td>
</tr>
<tr>
<td>RAND()</td>
<td>Returns a random number between 0 and 1 with a FLOAT data type. ( )</td>
</tr>
<tr>
<td>ROUND(n, p,t)</td>
<td>Rounds the value of the number ( n ) by using the precision ( p ). Use positive values of ( p ) to round on the right side of the decimal point and use negative values to round on the left side. An optional parameter ( t ) causes ( n ) to be truncated. Examples: ( \text{SELECT ROUND}(5.4567,3) = 5.4570 ) ( \text{SELECT ROUND}(345.4567,-1) = 350.0000 ) ( \text{SELECT ROUND}(345.4567,-1,1) = 340.0000 )</td>
</tr>
<tr>
<td>ROWCOUNT_BIG</td>
<td>Returns the number of rows that have been affected by the last Transact-SQL statement executed by the system. The return value of this function has the BIGINT data type. ( )</td>
</tr>
<tr>
<td>SIGN(n)</td>
<td>Returns the sign of the value ( n ) as a number (+1 for positive, -1 for negative, and 0 for zero). Example: ( \text{SELECT SIGN}(0.88) = 1.00 ) ( )</td>
</tr>
<tr>
<td>SIN(n)</td>
<td>Calculates the sine of ( n ). ( n ) and the resulting value belong to the FLOAT data type. ( )</td>
</tr>
<tr>
<td>SQRT(n)</td>
<td>Calculates the square root of ( n ). Example: ( \text{SELECT SQRT}(9) = 3 ) ( )</td>
</tr>
<tr>
<td>SQUARE(n)</td>
<td>Returns the square of the given expression. Example: ( \text{SELECT SQUARE}(9) = 81 ) ( )</td>
</tr>
<tr>
<td>TAN(n)</td>
<td>Calculates the tangent of ( n ). ( n ) and the resulting value belong to the FLOAT data type. ( )</td>
</tr>
</tbody>
</table>
Date Functions

Date functions calculate the respective date or time portion of an expression or return the value from a time interval. Transact-SQL supports the following date functions:

<table>
<thead>
<tr>
<th>Function</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>GETDATE()</td>
<td>Returns the current system date and time. Example: SELECT GETDATE() = 2011-01-01 13:03:31.390</td>
</tr>
<tr>
<td>DATEPART(item, date)</td>
<td>Returns the specified part item of a date date as an integer. Examples: SELECT DATEPART(month, '01.01.2005') = 1 (1 = January) SELECT DATEPART(weekday, '01.01.2005') = 7 (7 = Sunday)</td>
</tr>
<tr>
<td>DATENAME(item, date)</td>
<td>Returns the specified part item of the date date as a character string. Example: SELECT DATENAME(weekday, '01.01.2005') = Saturday</td>
</tr>
<tr>
<td>DATEDIFF(item, dat1, dat2)</td>
<td>Calculates the difference between the two date parts dat1 and dat2 and returns the result as an integer in units specified by the value item. Example (returns the age of each employee): SELECT DATEDIFF(year, BirthDate, GETDATE()) AS age FROM employee</td>
</tr>
<tr>
<td>DATEADD(i,n,d)</td>
<td>Adds the number n of units specified by the value i to the given date d. (n could be negative, too.) Example (adds three days to the start date of employment of every employee; see the sample database): SELECT DATEADD(DAY,3,HireDate) AS age FROM employee</td>
</tr>
</tbody>
</table>

String Functions

String functions are used to manipulate data values in a column, usually of a character data type. Transact-SQL supports the following string functions:

<table>
<thead>
<tr>
<th>Function</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASCII(character)</td>
<td>Converts the specified character to the equivalent integer (ASCII) code. Returns an integer. Example: SELECT ASCII('A') = 65</td>
</tr>
<tr>
<td>CHAR(integer)</td>
<td>Converts the ASCII code to the equivalent character. Example: SELECT CHAR(65) = 'A'</td>
</tr>
<tr>
<td>CHARINDEX(z1,z2)</td>
<td>Returns the starting position where the partial string z1 first occurs in the string z2. Returns 0 if z1 does not occur in z2. Example: SELECT CHARINDEX('bl', 'table') = 3</td>
</tr>
<tr>
<td>DIFFERENCE(z1,z2)</td>
<td>Returns an integer, 0 through 4, that is the difference of SOUNDEX values of two strings z1 and z2. (SOUNDEX returns a number that specifies the sound of a string. With this method, strings with similar sounds can be determined.) Example: SELECT DIFFERENCE('spelling', 'telling') = 2 (sounds a little bit similar, 0 = doesn't sound similar)</td>
</tr>
<tr>
<td>Function</td>
<td>Explanation</td>
</tr>
<tr>
<td>-------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>LEFT(z, length)</td>
<td>Returns the first length characters from the string z.</td>
</tr>
<tr>
<td>LEN(z)</td>
<td>Returns the number of characters, instead of the number of bytes, of the specified string expression, excluding trailing blanks.</td>
</tr>
<tr>
<td>LOWER(z1)</td>
<td>Converts all uppercase letters of the string z1 to lowercase letters. Lowercase letters and numbers, and other characters, do not change. Example:</td>
</tr>
<tr>
<td>Function</td>
<td>Explanation</td>
</tr>
<tr>
<td>------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>STUFF(z1,a,length,z2)</td>
<td>Replaces the partial string $z_1$ with the partial string $z_2$ starting at position $a$, replacing $length$ characters of $z_1$. Examples: &lt;br&gt;SELECT STUFF('Notebook',5,0,' in a') = 'Note in a book'  &lt;br&gt;SELECT STUFF('Notebook',1,4,'Hand') = 'Handbook'</td>
</tr>
</tbody>
</table>
| SUBSTRING(z,a,length)  | Creates a partial string from string $z$ starting at the position $a$ with a length of $length$.  
Example: <br>SELECT SUBSTRING('wardrobe',1,4) = 'ward' |
| UNICODE                | Returns the integer value, as defined by the Unicode standard, for the first character of the input expression. |
| UPPER(z)               | Converts all lowercase letters of string $z$ to uppercase letters. Uppercase letters and numbers do not change.  
Example: <br>SELECT UPPER('loWer') = 'LOWER' |

**System Functions**

System functions of Transact-SQL provide extensive information about database objects. Most system functions use an internal numeric identifier (ID), which is assigned to each database object by the system at its creation. Using this identifier, the system can uniquely identify each database object. System functions provide information about the database system. The following table describes several system functions. (For the complete list of all system functions, please see Books Online.)

<table>
<thead>
<tr>
<th>Function</th>
<th>Explanation</th>
</tr>
</thead>
</table>
| CAST(a AS type ([length])) | Converts an expression $a$ into the specified data type $type$ (if possible). $a$ could be any valid expression.  
Example: <br>SELECT CAST(3000000000 AS BIGINT) = 3000000000 |
| COALESCE($a_1,a_2,...$) | Returns for a given list of expressions $a_1$, $a_2$,... the value of the first expression that is not NULL. |
| COL_LENGTH(obj,col)     | Returns the length of the column $col$ belonging to the database object (table or view) $obj$.  
Example:<br>SELECT COL_LENGTH('customers', 'cust_ID') = 10 |
| CONVERT(type([length]),a) | Equivalent to CAST, but the arguments are specified differently. CONVERT can be used with any data type. |
### Function | Explanation
--- | ---
CURRENT_TIMESTAMP | Returns the current date and time. Example:
SELECT CURRENT_TIMESTAMP = '2011-01-01 17:22:55.670'
CURRENT_USER | Returns the name of the current user.
DATALENGTH(z) | Calculates the length (in bytes) of the result of the expression z. Example (returns the length of each field):
SELECT DATALENGTH(ProductName) FROM products
GETANSINULL('dbname') | Returns 1 if the use of NULL values in the database dbname complies with the ANSI SQL standard. (See also the explanation of NULL values at the end of this chapter.) Example:
SELECT GETANSINULL('AdventureWorks') = 1
ISNULL(expr, value) | Returns the value of expr if that value is not null; otherwise, it returns value.
ISNUMERIC(expression) | Determines whether an expression is a valid numeric type.
NEWID() | Creates a unique ID number that consists of a 16-byte binary string intended to store values of the UNIQUEIDENTIFIER data type.
NEWSEQUENTIALID() | Creates a GUID that is greater than any GUID previously generated by this function on a specified computer. (This function can be used only as a default value for a column.)
NULLIF(expr1,expr2) | Returns the NULL value if the expressions expr1 and expr2 are equal. Example (returns NULL for the project with the project_no = 'p1'):
SELECT NULLIF(project_no, 'p1') FROM projects
SERVERPROPERTY(propertyname) | Returns the property information about the database server.
SYSTEM_USER | Returns the login ID of the current user. Example:
SELECT SYSTEM_USER = LTB13942\dusan
USER_ID([user_name]) | Returns the identifier of the user user_name. If no name is specified, the identifier of the current user is retrieved. Example:
SELECT USER_ID('guest') = 2
USER_NAME([id]) | Returns the name of the user with the identifier id. If no name is specified, the name of the current user is retrieved. Example:
SELECT USER_NAME(1) = 'dbo'

All string functions can be nested in any order; for example, REVERSE(CURRENT_USER).
Metadata Functions

Generally, metadata functions return information about the specified database and database objects. The following table describes several metadata functions. (For the complete list of all metadata functions, please see Books Online.)

<table>
<thead>
<tr>
<th>Function</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>COL_NAME(tab_id, col_id)</td>
<td>Returns the name of a column belonging to the table with the ID tab_id and column ID col_id. Example: SELECT COL_NAME(OBJECT_ID('employee'), 3) = 'emp_lname'</td>
</tr>
<tr>
<td>COLUMNPROPERTY(id, col, property)</td>
<td>Returns the information about the specified column. Example: SELECT COLUMNPROPERTY(object_id('project'), 'project_no', 'PRECISION') = 4</td>
</tr>
<tr>
<td>DATABASEPROPERTYEX(database, property)</td>
<td>Returns the named database property value for the specified database and property. Example (specifies whether the database follows SQL-92 rules for allowing NULL values): SELECT DATABASEPROPERTYEX('sample', 'IsAnsiNullDefault') = 0</td>
</tr>
<tr>
<td>DB_ID([db_name])</td>
<td>Returns the identifier of the database db_name. If no name is specified, the identifier of the current database is returned. Example: SELECT DB_ID('AdventureWorks') = 6</td>
</tr>
<tr>
<td>DB_NAME([db_id])</td>
<td>Returns the name of the database with the identifier db_id. If no identifier is specified, the name of the current database is displayed. Example: SELECT DB_NAME(6) = 'AdventureWorks'</td>
</tr>
<tr>
<td>INDEX_COL(table, i, no)</td>
<td>Returns the name of the indexed column in the table table, defined by the index identifier i and the position no of the column in the index.</td>
</tr>
<tr>
<td>INDEXPROPERTY(obj_id, index_name, property)</td>
<td>Returns the named index or statistics property value of a specified table identification number, index or statistics name, and property name.</td>
</tr>
<tr>
<td>OBJECT_NAME(obj_id)</td>
<td>Returns the name of the database object with the identifier obj_id. Example: SELECT OBJECT_NAME(453576654) = 'products'</td>
</tr>
<tr>
<td>OBJECT_ID(obj_name)</td>
<td>Returns the identifier of the database object obj_name. Example: SELECT OBJECT_ID('products') = 453576654</td>
</tr>
<tr>
<td>OBJECTPROPERTY(obj_id,property)</td>
<td>Returns the information about the objects from the current database.</td>
</tr>
</tbody>
</table>

Scalar Operators

Scalar operators are used for operations with scalar values. Transact-SQL supports numeric and Boolean operators as well as concatenation.
There are unary and binary arithmetic operators. Unary operators are + and – (as signs). Binary arithmetic operators are +, –, *, /, and %. (The first four binary operators have their respective mathematical meanings, whereas % is the modulo operator.)

Boolean operators have two different notations depending on whether they are applied to bit strings or to other data types. The operators NOT, AND, and OR are applied to all data types (except BIT). They are described in detail in Chapter 6.

The bitwise operators for manipulating bit strings are listed here, and Example 4.6 shows how they are used:

- \sim \ C\ c \quad \text{Complement (i.e., NOT)}
- \& \ C\ c \quad \text{Conjunction of bit strings (i.e., AND)}
- \| \ C\ c \quad \text{Disjunction of bit strings (i.e., OR)}
- \wedge \ C\ c \quad \text{Exclusive disjunction (i.e., XOR or Exclusive OR)}

**Example 4.6**

\(
\sim(1001001) = (0110110)
\)
\(
(11001001) | (10101101) = (11101101)
\)
\(
(11001001) \& (10101101) = (10001001)
\)
\(
(11001001) ^ (10101101) = (01100100)
\)

The concatenation operator + can be used to concatenate two character strings or bit strings.

**Global Variables**

Global variables are special system variables that can be used as if they were scalar constants. Transact-SQL supports many global variables, which have to be preceded by the prefix @@. The following table describes several global variables. (For the complete list of all global variables, see Books Online.)

<table>
<thead>
<tr>
<th>Variable</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>@@CONNECTIONS</td>
<td>Returns the number of login attempts since starting the system.</td>
</tr>
<tr>
<td>@@CPU_BUSY</td>
<td>Returns the total CPU time (in units of milliseconds) used since starting the system.</td>
</tr>
<tr>
<td>@@ERROR</td>
<td>Returns the information about the return value of the last executed Transact-SQL statement.</td>
</tr>
<tr>
<td>@@IDENTITY</td>
<td>Returns the last inserted value for the column with the IDENTITY property (see Chapter 6).</td>
</tr>
<tr>
<td>@@LANGID</td>
<td>Returns the identifier of the language that is currently used by the database system.</td>
</tr>
<tr>
<td>@@LANGUAGE</td>
<td>Returns the name of the language that is currently used by the database system.</td>
</tr>
<tr>
<td>@@MAX_CONNECTIONS</td>
<td>Returns the maximum number of actual connections to the system.</td>
</tr>
</tbody>
</table>
### Variable

<table>
<thead>
<tr>
<th>Variable</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>@@PROCID</td>
<td>Returns the identifier for the stored procedure currently being executed.</td>
</tr>
<tr>
<td>@@ROWCOUNT</td>
<td>Returns the number of rows that have been affected by the last Transact-SQL statement executed by the system.</td>
</tr>
<tr>
<td>@@SERVERNAME</td>
<td>Retrieves information about the local database server. This information contains, among other things, the name of the server and the name of the instance.</td>
</tr>
<tr>
<td>@@SPID</td>
<td>Returns the identifier of the server process.</td>
</tr>
<tr>
<td>@@VERSION</td>
<td>Returns the current version of the database system software.</td>
</tr>
</tbody>
</table>

### NULL Values

A NULL value is a special value that may be assigned to a column. This value normally is used when information in a column is unknown or not applicable. For example, in the case of an unknown home telephone number for a company’s employee, it is recommended that the NULL value be assigned to the `home_telephone` column.

Any arithmetic expression results in a NULL if any operand of that expression is itself a NULL value. Therefore, in unary arithmetic expressions (if A is an expression with a NULL value), both +A and –A return NULL. In binary expressions, if one (or both) of the operands A or B has the NULL value, A + B, A – B, A * B, A / B, and A % B also result in a NULL. (The operands A and B have to be numerical expressions.)

If an expression contains a relational operation and one (or both) of the operands has (have) the NULL value, the result of this operation will be NULL. Hence, each of the expressions A = B, A <> B, A < B, and A > B also returns NULL.

In the Boolean AND, OR, and NOT, the behavior of the NULL values is specified by the following truth tables, where T stands for true, U for unknown (NULL), and F for false. In these tables, follow the row and column represented by the values of the Boolean expressions that the operator works on, and the value where they intersect represents the resulting value.

<table>
<thead>
<tr>
<th>AND</th>
<th>T</th>
<th>U</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>U</td>
<td>F</td>
</tr>
<tr>
<td>U</td>
<td>U</td>
<td>U</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OR</th>
<th>T</th>
<th>U</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>U</td>
<td>T</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>U</td>
<td>F</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>NOT</th>
<th>T</th>
<th>U</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>F</td>
<td>U</td>
</tr>
<tr>
<td>U</td>
<td>U</td>
<td>U</td>
</tr>
</tbody>
</table>

Any NULL value in the argument of aggregate functions AVG, SUM, MAX, MIN, and COUNT is eliminated before the respective function is calculated (except for the function COUNT(*)). If a column contains only NULL values, the function
returns NULL. The aggregate function COUNT(*) handles all NULL values the same as non-NUL values. If the column contains only NULL values, the result of the function COUNT(DISTINCT column_name) is 0.

A NULL value has to be different from all other values. For numeric data types, there is a distinction between the value zero and NULL. The same is true for the empty string and NULL for character data types.

A column of a table allows NULL values if its definition explicitly contains NULL. On the other hand, NULL values are not permitted if the definition of a column explicitly contains NOT NULL. If the user does not specify NULL or NOT NULL for a column with a data type (except TIMESTAMP), the following values are assigned:

- **NULL** If the ANSI_NULL_DFLT_ON option of the SET statement is set to ON
- **NOT NULL** If the ANSI_NULL_DFLT_OFF option of the SET statement is set to ON

If the SET statement isn’t activated, a column will contain the value NOT NULL by default. (The columns of TIMESTAMP data type can be declared only as NOT NULL.)

**Summary**

The basic features of Transact-SQL consist of data types, predicates, and functions. Data types comply with data types of the ANSI SQL92 standard. Transact-SQL supports a variety of useful system functions.

The next chapter introduces you to Transact-SQL statements in relation to SQL’s data definition language. This part of Transact-SQL comprises all the statements needed for creating, altering, and removing database objects.

**Exercises**

**E.4.1**

What is the difference between the numeric data types INT, SMALLINT, and TINYINT?

**E.4.2**

What is the difference between the data types CHAR and VARCHAR? When should you use the latter (instead of the former) and vice versa?
E.4.3
How can you set the format of a column with the DATE data type so that its values can be entered in the form ‘yyyy/mm/dd’?

In the following two exercises, use the SELECT statement in the Query Editor component window of SQL Server Management Studio to display the result of all system functions and global variables. (For instance, SELECT host_id() displays the ID number of the current host.)

E.4.4
Using system functions, find the ID number of the test database (Exercise 2.1).

E.4.5
Using the system variables, display the current version of the database system software and the language that is used by this software.

E.4.6
Using the bitwise operators &, |, and ^, calculate the following operations with the bit strings:

(11100101) & (01010111)
(10011011) | (11001001)
(10110111) ^ (10110001)

E.4.7
What is the result of the following expressions? (A is a numerical expression and B is a logical expression.)

A + NULL
NULL = NULL
B OR NULL
B AND NULL

E.4.8
When can you use both single and double quotation marks to define string and temporal constants?

E.4.9
What is a delimited identifier and when do you need it?