1 System Functions

SQL Server includes a number of "system functions" as well as more typical functions with the product. Some of these are used often and are fairly clear right from the beginning in terms of how to use them. Others, though, are both rarer in use and more cryptic in nature.

In this appendix, I'll try to clarify the use of most of these functions in a short, concise manner.

NOTE *Just as an FYI, in prior releases, many system functions were often referred to as* global variables. *This was a misnomer, and Microsoft has attempted to fix it over the last few releases — changing the documentation to refer to them by the more proper* system function *name. Just keep the old terminology in mind in case any old fogies (such as myself) find themselves referring to them as globals.*

The T-SQL functions available in SQL Server 2012 fall into the following categories:

- System statistical functions
- Aggregate functions
- Configuration functions
- Cryptographic functions
- Cursor functions
- Date and time functions
- Mathematical functions
- Metadata functions
- Ranking functions

- Rowset functions
- Security functions
- String functions
- System functions
- Text and image functions

> ***NOTE*** *In addition, there is the* `OVER` *operator, which, while largely working as a ranking tool, can be applied to other forms of T-SQL functions (most notably aggregates). Although I discuss it only as part of the ranking functions, you may see it referenced several other places in this appendix.*

SYSTEM STATISTICAL FUNCTIONS

The first section in this appendix contains system functions related to operational state and system statistics. You'll notice that most of these look different than other functions in that their names begin with "@@"; these are the system functions formerly known as global variables.

@@CONNECTIONS

Returns the number of connections attempted since the last time your SQL Server was started.

This one is the total of all connection attempts made since the last time your SQL Server was started. The key thing to remember here is that I am talking about attempts, not actual connections, and that I am talking about connections as opposed to users.

Every attempt made to create a connection increments this counter regardless of whether that connection was successful. The only catch with this is that the connection attempt has to have made it as far as the server. If the connection failed because of NetLib differences or some other network issue, your SQL Server wouldn't even know that it needed to increase the count — it only counts if the server saw the connection attempt. Whether the attempt succeeded or failed does not matter.

It's also important to understand that I'm talking about connections instead of login attempts. Depending on your application, you may create several connections to your server, but you'll probably only ask the user for information once. Indeed, even Query Analyzer does this. When you click for a new window, it automatically creates another connection based on the same login information.

> ***NOTE*** *This, like a number of other system functions, is often better served by a system stored procedure,* `sp_monitor`*. This procedure, in one command, produces the information from the number of connections, CPU busy, through to the total number of writes by SQL Server. So, if basic information is what you're after,* `sp_monitor` *may be better — if you need discrete data that you can manipulate,* `@@CONNECTIONS` *provides a nice, neat, scalar piece of data.*

@@CPU_BUSY

Returns the time in milliseconds that the CPU has been actively doing work since SQL Server was last started. This number is based on the resolution of the system timer — which can vary — and can therefore vary in accuracy.

This is another of the "since the server started" kind of functions. This means that you can't always count on the number going up as your application runs. It's possible, based on this number, to determine a CPU percentage that your SQL Server is taking up. Realistically though, I'd rather tap right into the Performance Monitor for that if I had some dire need for it. The bottom line is that this is one of those really cool things from a "gee, isn't it swell to know that" point of view, but doesn't have all that many practical uses in most applications.

sys.fn_virtualfilestats

Calling `sys.fn_virtualfilestats` lets you know the total reads, writes, and wait times on a database file since the database instance started. Unlike the system functions here that look like they're actually global variables, this function is called like an ordinary table-valued function. It takes two parameters — the database ID and the file ID — and each can be `NULL` to indicate you want to see unfiltered results.

If you've accepted all defaults for a DB, `fileid 1` is the data file and `fileid 2` is the primary log. Thus a call to `sys.fn_virtualfilestats(NULL, 2)` will result in statistics for all the primary log files. The data returned includes:

- **`Dbid`:** The database ID for this output row
- **`FileID`:** The file ID
- **`TimeStamp`:** The number of seconds your instance has been running
- **`NumberReads`:** The total number of reads on this file since the instance started
- **`NumberWrites`:** The total number of writes on this file since the instance started
- **`BytesRead`:** The total bytes read from this file
- **`BytesWritten`:** The total bytes written to this file
- **`IoStallMs`:** The number of milliseconds that processes had to wait for IO to return

Things like reads and writes are great for getting an idea of general IO volume, but `IoStallMs` can be invaluable in locating performance bottlenecks.

@@IDLE

Returns the time in milliseconds (based on the resolution of the system timer) that SQL Server has been idle since it was last started.

You can think of this one as being something of the inverse of `@@CPU_BUSY`. Essentially, it tells you how much time your SQL Server has spent doing nothing. If anyone finds a programmatic use for this one, send me an email — I'd love to hear about it (I can't think of one).

@@IO_BUSY

Returns the time in milliseconds (based on the resolution of the system timer) that SQL Server has spent doing input and output operations since it was last started. This value is reset every time SQL Server is started.

This one doesn't really have any rocket science to it, and it is another one of those that I find falls into the "no real programmatic use" category.

@@PACK_RECEIVED and @@PACK_SENT

Respectively return the number of input packets read/written from/to the network by SQL Server since it was last started.

Primarily, these are network troubleshooting tools.

@@PACKET_ERRORS

Returns the number of network packet errors that have occurred on connections to your SQL Server since the last time the SQL Server was started.

Primarily a network troubleshooting tool.

@@TIMETICKS

Returns the number of microseconds per tick. This varies by machines and is another of those that falls under the category of "no real programmatic use."

@@TOTAL_ERRORS

Returns the number of disk read/write errors encountered by the SQL Server since it was last started.

Don't confuse this with runtime errors or as having any relation to `@@ERROR`. This is about problems with physical I/O. This one is another of those of the "no real programmatic use" variety. The primary use here would be more along the lines of system diagnostic scripts. Generally speaking, I would use Performance Monitor for this instead.

@@TOTAL_READ and @@TOTAL_WRITE

Respectively return the total number of disk reads/writes by SQL Server since it was last started.

The names here are a little misleading, as these do not include any reads from cache — they are only physical I/O.

AGGREGATE FUNCTIONS

Aggregate functions are applied to sets of records rather than a single record. The information in the multiple records is processed in a particular manner and then is displayed in a single record answer. Aggregate functions are often used in conjunction with the `GROUP BY` clause.

The aggregate functions are:

- `AVG`
- `CHECKSUM`
- `CHECKSUM_AGG`
- `COUNT`
- `COUNT_BIG`
- `GROUPING`
- `MAX`
- `MIN`
- `STDEV`
- `STDEVP`
- `SUM`
- `VAR`
- `VARP`

In most aggregate functions, the `ALL` or `DISTINCT` keywords can be used. The `ALL` argument is the default and will apply the function to all the values in the `expression`, even if a value appears numerous times. The `DISTINCT` argument means that a value will be included in the function only once, even if it occurs several times.

Aggregate functions cannot be nested. The `expression` cannot be a subquery.

AVG

`AVG` returns the average of the values in `expression`. The syntax is as follows:

```
AVG([ALL | DISTINCT] <expression>)
```

The `expression` must contain numeric values. `NULL` values are ignored. This function supports the `OVER` operator described in the ranking functions section of this appendix.

CHECKSUM

This is a basic hash algorithm normally used to detect changes or consistency in data. This particular function accepts either an expression as an argument or a `*` (which implies that you want all columns in all the joined tables to be included). The basic syntax is:

```
CHECKSUM(<expression>, [...n] | * )
```

Note that the order of your expression — or in the case of a `*`, the join order — will affect the checksum value, so, for example:

```
CHECKSUM(SalesOrderID, OrderDate)
```

would not give the same result as:

```
CHECKSUM(OrderDate, SalesOrderID )
```

This function is *NOT* compatible with the `OVER` operator.

NOTE *`CHECKSUM` isn't really like the other aggregate functions. They all aggregate multiple rows, and `CHECKSUM` works against only one. However, it does (or can) aggregate all the data within the row, so it can arguably be included here. My real reason for including it now, though, is that `CHECKSUM_AGG` is much easier to understand if you already know `CHECKSUM`.*

CHECKSUM_AGG

Like `CHECKSUM`, this is a basic hash usually used to detect changes or consistency in data. The primary difference is that `CHECKSUM` is oriented around rows, whereas `CHECKSUM_AGG` is oriented around columns. The basic syntax is:

```
CHECKSUM_AGG( [ALL | DISTINCT] <expression>)
```

The expression value can be virtually anything, including, if you want, concatenation of columns (just remember to cast as necessary); however, remember that expression order does matter, so if you're concatenating, `Col1 + Col2` does not equal `Col2 + Col1`.

COUNT

`COUNT` returns the number of items in `expression`. The data type returned is of type `int`. The syntax is as follows:

```
COUNT
(
   [ALL | DISTINCT] <expression> | *
)
```

The expression cannot be of the `uniqueidentifier`, `text`, `image`, or `ntext` data types. The `*` argument returns the number of rows in the table; it does not eliminate duplicate or `NULL` values.

This function supports the `OVER` operator described in the ranking functions section of this appendix.

COUNT_BIG

`COUNT_BIG` returns the number of items in a group. This is very similar to the `COUNT` function just described, with the exception that the return value has a data type of `bigint`. The syntax is as follows:

```
COUNT_BIG
(
   [ALL | DISTINCT ] <expression> | *
)
```

Like COUNT, this function supports the OVER operator described in the ranking functions section of this appendix.

GROUPING

GROUPING adds an extra column to the output of a SELECT statement. The GROUPING function is used in conjunction with CUBE or ROLLUP to distinguish between normal NULL values and those added as a result of CUBE and ROLLUP operations. Its syntax is:

```
GROUPING (<column name>)
```

GROUPING is only used in the SELECT list. Its argument is a column that is used in the GROUP BY clause and that is to be checked for NULL values.

This function supports the OVER operator described in the ranking functions section of this appendix.

MAX

The MAX function returns the maximum value from expression. The syntax is as follows:

```
MAX([ALL | DISTINCT] <expression>)
```

MAX ignores any NULL values.

This function supports the OVER operator described in the ranking functions section of this appendix.

MIN

The MIN function returns the smallest value from expression. The syntax is as follows:

```
MIN([ALL | DISTINCT] <expression>)
```

MIN ignores NULL values.

This function supports the OVER operator described in the ranking functions section of this appendix.

STDEV

The STDEV function returns the standard deviation of all values in expression. The syntax is as follows:

```
STDEV(<expression>)
```

STDEV ignores NULL values.

This function supports the OVER operator described in the ranking functions section of this appendix.

STDEVP

The STDEVP function returns the standard deviation for the population of all values in expression. The syntax is as follows:

```
STDEVP(<expression>)
```

STDEVP ignores NULL values and supports the OVER operator described in the ranking functions section of this appendix.

SUM

The SUM function will return the total of all values in expression. The syntax is as follows:

```
SUM([ALL | DISTINCT] <expression>)
```

SUM ignores NULL values. This function supports the OVER operator described in the ranking functions section of this appendix.

VAR

The VAR function returns the variance of all values in expression. The syntax is as follows:

```
VAR(<expression>)
```

VAR ignores NULL values. This function supports the OVER operator described in the ranking functions section of this appendix.

VARP

The VARP function returns the variance for the population of all values in expression. The syntax is as follows:

```
VARP(<expression>)
```

VARP ignores NULL values. This function supports the OVER operator described in the ranking functions section of this appendix.

ANALYTIC FUNCTIONS

SQL Server 2012 includes a new class of aggregate functions made for helping you perform statistical analysis. These are aggregates, but where a typical aggregate (like SUM) can only return one value for the rows it operates across, these can return multiple rows. They're especially useful for computing things like moving averages or running totals.

CUME_DIST

```
CUME_DIST( )
    OVER ( [ partition_by_clause ] order_by_clause )
```

Returns the percentage (a number between 0 and 1) of rows within the partition that have a value (the value is chosen by the ORDER BY clause) less than or equal to the current value.

FIRST_VALUE, LAST_VALUE

```
[ FIRST_VALUE | LAST_VALUE ] ( [scalar_expression )
OVER (
    [ partition_by_clause ] order_by_clause [ rows_range_clause ]
)
```

First and last values are something that SQL programmers have been developing tricks to compute for a long time, and you finally have a direct function to grab them for you in SQL 2012. These are cool because previously, the only ways to do this involved (at some level) a self-join. If you missed Chapter 4, a self-join is when you join a table back to itself; in any case, joins are expensive, and this function is less so.

Note that only the order by clause is required; omitting the other two represents the simplest case, whereby you only want the first out of the whole result set. Adding a partition by clause can get you the first per partition, and the rows range clause can tweak your results even further by allowing you to narrow the range of rows within the partition from which you'll select the first or last.

LAG, LEAD

```
[LAG | LEAD] (scalar_expression [,offset] [,default])
    OVER ( [ partition_by_clause ] order_by_clause )
```

Whereas `FIRST_VALUE` and `LAST_VALUE` can grab you a row from either end of your result set, `LAG` and `LEAD` can give you one that's a few steps before or after your current row. This can be useful, say, for computing a delta over time; using `LAG`, each row can be aware of the value of its previous row, allowing you to compute the change in value.

While the `OVER` clause is identical to what's been used elsewhere, the other parameters aren't so self-explanatory.

- `scalar_expression`: The value from the offset row you want to return.
- `offset`: How many rows prior (for `LAG`) or after (for `LEAD`) you want to look at; the default is 1.
- `default`: The value to be returned if there is no row at the specified offset. This must be implicitly convertible to the data type of `scalar_expression`.

PERCENTILE_CONT, PERCENTILE_DISC

```
 [ PERCENTILE_CONT | PERCENTILE_DISC ] ( numeric_literal )
WITHIN GROUP ( ORDER BY order_by_expression [ ASC | DESC ] )
OVER ( [ <partition_by_clause> ] )
```

These functions return the value of a row that is in the percentile specified by `numeric_literal` within the group described. `PERCENTILE_DISC` finds an actual row value — the closest available to the percentile break — and returns that, whereas `PERCENTILE_CONT` assumes an even distribution and returns the value at which the percentile would break (even if that value doesn't exist within the group).

You can think of these functions as using the `CUME_DIST` function. `PERCENTILE_DISC` finds the row with the lowest `CUME_DIST` below `numeric_literal`, and returns it.

PERCENT_RANK

```
PERCENT_RANK( )
OVER ( [ partition_by_clause ] order_by_clause )
```

Computes the percentile into which the rank of each value falls. This is kind of like `CUME_DIST`, which gives you the percentage of rows that fall at or below this value, but with a twist; identical rows reduce the number of distinct ranks available, so `PERCENT_RANK` tells you what percentage of ranks fall below this rank. So if there were only three distinct values in a column, and the bottom represented half of the values, `CUME_DIST` would return 0.5 ("half of the values are at or below this value"), while `PERCENT_RANK` would return zero ("none of the ranks are below this value"). The middle value would always return a `PERCENT_RANK` of 0.5, and the top of 1, regardless of how many rows there are in each rank.

CONFIGURATION FUNCTIONS

Well, I'm sure it will come as a complete surprise (okay, not really…), but configuration functions are those functions that tell you about options as they are set for the current server or database (as appropriate).

@@DATEFIRST

Returns the numeric value that corresponds to the day of the week that the system considers the first day of the week.

The default in the United States is `7`, which equates to Sunday. The values convert as follows:

- `1`: Monday (the first day for most of the world)
- `2`: Tuesday
- `3`: Wednesday
- `4`: Thursday
- `5`: Friday
- `6`: Saturday
- `7`: Sunday

This can be really handy when dealing with localization issues, so you can properly lay out any calendar or other day-of-week-dependent information you have.

NOTE *Use the* `SET DATEFIRST` *function to alter this setting.*

@@DBTS

Returns the last used timestamp for the current database.

At first look, this one seems to act an awful lot like `@@IDENTITY` in that it gives you the chance to get back the last value set by the system (this time, it's the last timestamp instead of the last identity value). The things to watch out for on this one include:

- The value changes based on any change in the database, not just the table you're working on.
- *Any* timestamp change in the database is reflected, not just those for the current connection.

Because you can't count on this value truly being the last one that you used (someone else may have done something that would change it), I personally find very little practical use for this one.

@@LANGID and @@LANGUAGE

Respectively return the ID and the name of the language currently in use.

These can be handy for figuring out if your product has been installed in a localization situation, and if so, what language is the default.

For a full listing of the languages currently supported by SQL Server, use the system stored procedure, `sp_helplanguage`.

@@LOCK_TIMEOUT

Returns the current amount of time in milliseconds before the system will time out waiting for a blocked resource.

If a resource (a page, a row, a table, whatever) is blocked, your process will stop and wait for the block to clear. This determines just how long your process will wait before the statement is canceled.

The default time to wait is `0` (which equates to indefinitely) unless someone has changed it at the system level (using `sp_configure`). Regardless of how the system default is set, you will get a value of `–1` from this global unless you have manually set the value for the current connection using `SET LOCK_TIMEOUT`.

@@MAX_CONNECTIONS

Returns the maximum number of simultaneous user connections allowed on your SQL Server.

Don't mistake this one to mean the same thing as you would see under the Maximum Connections property in the Management Console. This one is based on licensing and will show a very high number if you have selected "per seat" licensing.

NOTE *Note that the actual number of user connections allowed also depends on the version of SQL Server you are using and the limits of your application(s) and hardware.*

@@MAX_PRECISION

Returns the level of precision currently set for decimal and numeric data types.

The default is 38 places, but the value can be changed by using the `/p` option when you start your SQL Server. The `/p` can be added by starting SQL Server from a command line or by adding it to the Startup parameters for the MSSQLServer service in the Windows 2000, 2003, XP, or 2008 services applet.

@@NESTLEVEL

Returns the current nesting level for nested stored procedures.

The first stored procedure (sproc) to run has an `@@NESTLEVEL` of `0`. If that sproc calls another, the second sproc is said to be nested in the first sproc (and `@@NESTLEVEL` is incremented to a value of `1`). Likewise, the second sproc may call a third, and so on up to maximum of 32 levels deep. If you go past the level of 32 levels deep, not only will the transaction be terminated, but you should revisit the design of your application.

@@OPTIONS

Returns information about options that have been applied using the `SET` command.

Because you only get one value back, but can have many options set, SQL Server uses binary flags to indicate which values are set. In order to test whether the option you are interested in is set, you must use the option value together with a bitwise operator. For example:

```
IF (@@OPTIONS & 2)
```

If this evaluates to `True`, you would know that `IMPLICIT_TRANSACTIONS` had been turned on for the current connection. The values are shown in Table B-1.

TABLE B-1: @@OPTIONS Bitmask Values

BIT	SET OPTION	DESCRIPTION
1	`DISABLE_DEF_CNST_CHK`	Interim versus deferred constraint checking.
2	`IMPLICIT_TRANSACTIONS`	A transaction is started implicitly when a statement is executed.
4	`CURSOR_CLOSEON_COMMIT`	Controls behavior of cursors after a `COMMIT` operation has been performed.
8	`ANSI_WARNINGS`	Warns of truncation and `NULL` in aggregates.
16	`ANSI_PADDING`	Controls padding of fixed-length variables.
32	`ANSI_NULLS`	Determines handling of nulls when using equality operators.

BIT	SET OPTION	DESCRIPTION
64	`ARITHABORT`	Terminates a query when an overflow or divide-by-zero error occurs during query execution.
128	`ARITHIGNORE`	Returns `NULL` when an overflow or divide-by-zero error occurs during a query.
256	`QUOTED_IDENTIFIER`	Differentiates between single and double quotation marks when evaluating an expression.
512	`NOCOUNT`	Turns off the row(s) affected message returned at the end of each statement.
1024	`ANSI_NULL_DFLT_ON`	Alters the session's behavior to use ANSI compatibility for nullability. Columns created with new tables or added to old tables without explicit null option settings are defined to allow nulls. Mutually exclusive with `ANSI_NULL_DFLT_OFF`.
2048	`ANSI_NULL_DFLT_OFF`	Alters the session's behavior not to use ANSI compatibility for nullability. New columns defined without explicit nullability are defined not to allow nulls. Mutually exclusive with `ANSI_NULL_DFLT_ON`.
4096	`CONCAT_NULL_YIELDS_NULL`	Returns a `NULL` when concatenating a `NULL` with a string.
8192	`NUMERIC_ROUNDABORT`	Generates an error when a loss of precision occurs in an expression.

@@REMSERVER

Returns the value of the server (as it appears in the login record) that called the stored procedure.

Used only in stored procedures. This one is handy when you want the sproc to behave differently depending on what remote server (often a geographic location) the sproc was called from.

@@SERVERNAME

Returns the name of the local server that the script is running from.

If you have multiple instances of SQL Server installed (a good example would be a web hosting service that uses a separate SQL Server installation for each client), `@@SERVERNAME` returns the local server name information, shown in Table B-2,if the local server name has not been changed since setup.

TABLE B-2: Local Server Name Information that @@SERVERNAME Returns

INSTANCE	SERVER INFORMATION
Default instance	*<servername>*
Named instance	*<servername\instancename>*
Virtual server — default instance	*<virtualservername>*
Virtual server — named instance	*<virtualservername\instancename>*

@@SERVICENAME

Returns the name of the registry key under which SQL Server is running. This will be MSSQLService if it is the default instance of SQL Server, or the instance name if applicable.

@@SPID

Returns the server process ID (SPID) of the current user process.

This equates to the same process ID that you see if you run `sp_who`. What's nice is that you can tell the `SPID` for your current connection, which can be used by the DBA to monitor, and if necessary terminate, that task.

@@TEXTSIZE

Returns the current value of the `TEXTSIZE` option of the `SET` statement, which specifies the maximum length, in bytes, returned by a `SELECT` statement when dealing with text or image data.

The default is 4,096 bytes (4KB). You can change this value by using the `SET TEXTSIZE` statement.

@@VERSION

Returns the current version of SQL Server as well as the processor type and OS architecture.

For example:

```
SELECT @@VERSION
```

gives:

```
-----------------------------------------------------------------------------
Microsoft SQL Server 2012 RC0 - 11.0.1750.32 (X64)
    Nov  4 2011 17:54:22
    Copyright (c) Microsoft Corporation
    Enterprise Evaluation Edition (64-bit) on Windows NT 6.1 <X64>
(Build 7601: Service Pack 1) (Hypervisor)


(1 row(s) affected)
```

Unfortunately, this doesn't return the information into any kind of structured field arrangement, so you have to parse it if you want to use it to test for specific information.

Consider using the `xp_msver` system sproc instead — it returns information in such a way that you can more easily retrieve specific information from the results.

CONVERSION FUNCTIONS

Wouldn't it be handy if SQL Server could freely convert data between any two types? I've heard beginning developers scream with frustration because what they want should be easy and obvious ("I concatenated that number into a string…of COURSE I wanted it converted to a string!"), even though SQL requires you to be a bit more explicit. In fact, it's rarely easy or obvious when you start considering other cultures. For example, the date "01/02/03" could mean one thing to an American (January 2nd, 2003), another to most of the rest of the world (February 1st, 2003). The functions in the following sections help you to make these and other conversions more explicit and clear without overburdening you with details when you don't have to use them.

CAST and CONVERT

These two functions provide similar functionality in that they both convert one data type into another type. `CAST` is the simplest, making assumptions where it can, whereas `CONVERT` requires you to define from and to formats when `CAST` wouldn't be deterministic.

➤ Using `CAST`

```
CAST(<expression> AS <data type>)
```

➤ Using `CONVERT`:

```
CONVERT (<data type>[(<length>)], <expression> [, <style>])
```

Where `style` refers to the style of date format when converting to a character data type.

PARSE

`PARSE` is a lot like `CAST`, but its only source is a string value and it's culture-aware. `CAST` always uses the system's current culture settings, whereas `PARSE` lets you specify a cultural setting to apply to this conversion (it'll use the system setting as its default).

```
PARSE ( string_value AS data_type [ USING culture ] )
```

So, to use the introductory example this code:

```
PARSE ( '01/02/03' AS DATE USING 'en-US' )
```

returns a date value of January 2nd, 2003, whereas this code

```
PARSE ('01/02/03' AS DATE USING 'fr-FR' )
```

returns a date value of February 1st, 2003.

TRY_CONVERT

TRY_CONVERT does exactly what CONVERT does, with one exception. Whereas CONVERT throws an error if the conversion is impossible (try converting a string value of '99/99/NONE' to a date, for example), TRY_CONVERT returns a converted value on success and a NULL on failure with no error thrown.

```
TRY_CONVERT ( data_type [ ( length ) ], expression [, style ] )
```

TRY_PARSE

Like TRY_CONVERT, TRY_PARSE attempts to parse the string provided into the target data type, and fails silently with a NULL return value if it can't.

```
TRY_PARSE ( string_value AS data_type [ USING culture ] )
```

CRYPTOGRAPHIC FUNCTIONS

These functions help support the encryption, decryption, digital signing, and digital signature validation. Some of these were new with SQL Server 2008, and some came with SQL Server 2005; SQL Server 2012 hasn't added much here. Notice that there are duplicates of most functions from a general use point of view, but that they are different in that one supports a symmetric key and the duplicate (usually with an Asym in the name) supports an asymmetrical key.

Now, you may ask "why would I need these?" The answer is as varied as the possible applications for SQL Server. The quick answer though is this: Any time you're sending or accepting data that you want to protect during transport. For example, because SQL Server supports HTTP endpoints, and, from that, hosting of its own web services, you may want to accept or return encrypted information with a client of your web service. Perhaps a more basic example is simply that you've chosen to encrypt the data in your database, and now you need to get it back out in a useful manner.

AsymKey_ID

Given the name of an asymmetric key, this function returns an int that corresponds to the related ID from the database. The syntax is simple:

```
AsymKey_ID('<Asymmetric Key Name>')
```

You must have permissions to the key in question to use this function.

Cert_ID

Similar to AsymKey_ID, this returns an ID that relates to the name of a certificate name. The syntax is simple:

```
Cert_ID('<Certificate Name>')
```

You must have permissions to the certificate in question to use this function.

CertProperty

Allows you to fetch various properties of a given certificate (as identified by the certificate's ID). Valid properties include the start date, expiration date, certificate issuer's name, serial number, security ID (the "SID," which can also be returned as a string), and the subject of the certificate (who or what is being certified). The syntax looks like this:

```
CertProperty ( <Cert_ID> ,
    '<Expiry Date>' | '<Start Date>' | '<Issuing Authority>' |
    '<Certificate Serial Number>' | '<Subject>' | '<Security ID>' |
    '<SID as a String>'  )
```

The data type returned will vary depending on the specific property you're looking for (`datetime`, `nvarchar`, or `varbinary`, as appropriate).

DecryptByAsmKey

As you can imagine by the name, this one decrypts a chunk of data utilizing an asymmetric key. It requires the key (by ID), the encrypted data (either as a literal string or a string coercible variable), and the password used to encrypt the asymmetric key (if one was used when the key was created). The syntax is straightforward enough:

```
DecryptByAsymKey(<Asymmetric Key ID>, {'<encrypted string>'|<string variable>}
    [, '<password>'])
```

Keep in mind that, if a password was utilized when creating the asymmetric key, the same password is going to be required to properly decrypt data utilizing that key.

DecryptByCert

This is basically the same as `DecryptByAsmKey`, except that it expects a certificate rather than an asymmetric key. Like `DecryptByAsmKey`, this one decrypts a chunk of data utilizing a key. It requires the certificate (by ID), the encrypted data (either as a literal string or a string coercible variable), and the password used to further encrypt the data (if one was used). The syntax looks almost just like `DecryptByAsymKey`:

```
DecryptByCert(<Certificate ID>, {'<encrypted string>'|<string variable>}
    [, '<password>'])
```

Again, any password utilized when encrypting the data will be needed to properly decrypt it.

DecryptByKey

Like its asymmetric and certificate based brethren, this one decrypts a chunk of data utilizing a key. What's different is that this one not only expects a symmetric key (instead of the other types of key), but it also expects that key to already be "open" (using the `OPEN SYMMETRIC KEY` command). Other than that, it is fairly similar in use, with the encrypted data (either as a literal string or a string coercible variable) fed in as a parameter and, in this case, a hash key optionally accepted as an authenticator:

```
DecryptByKey({'<encrypted string>'|<string variable>},
    [<add authenticator value>, '<authentication hash>'|<string variable>])
```

Note that if you provide an add authenticator value (in the form of an `int`), that value must match the value supplied when the string was encrypted, and you must also supply a hash value that matches the hash supplied at encryption time.

DecryptByPassPhrase

Like the name says, this one decrypts data that was encrypted not by a formal key, but by a passphrase. Other than accepting a passphrase parameter instead of assuming an open key, `DecryptByPassPhrase` works almost exactly like `DecryptByKey`:

```
DecryptByPassPhrase({'<passphrase>'|<string variable>},
    {'<encrypted string>'|<string variable>},
    [<add authenticator value>, '<authentication hash>'|<string variable>])
```

As with `DecryptByKey`, if you provide an add authenticator value (in the form of an `int`), that value must match the value supplied when the string was encrypted, and you must also supply a hash value that matches the hash supplied at encryption time.

EncryptByAsmKey

Encrypts a chunk of data utilizing an asymmetric key. It requires the key (by ID) and the data to be encrypted (either as a literal string or a string coercible variable). The syntax is straightforward enough:

```
EncryptByAsymKey(<Asymmetric Key ID>, {'<string to encrypt>'|<string variable>})
```

Keep in mind that if a password was utilized when the asymmetric key was added to the database, the same password will be required to properly decrypt any data encrypted using that key.

EncryptByCert

This is basically the same as `EncryptByAsmKey`, except that it expects a certificate rather than an asymmetric key. Like `EncryptByAsmKey`, this one encrypts a chunk of data utilizing the provided key. It requires the certificate (by ID), the data to be encrypted (either as a literal string or a string coercible variable), and optionally, the password to be used to further encrypt the data. The syntax looks almost just like `EncryptByAsymKey`:

```
EncryptByCert(<Certificate ID>, {'<string to be encrypted>'|<string variable>}
    [, '<password>'])
```

Again, any password utilized when encrypting the data will be needed to properly decrypt it.

EncryptByKey

This one not only expects a symmetric key (instead of the other types of key), but it also expects that key to already be "open" (using the `OPEN SYMMETRIC KEY` command) and a GUID to be available to reference that key by. Other than that, it is fairly similar in use, with the data to be encrypted (either as a literal string or a string coercible variable) fed in as a parameter and, in this case, a hash key optionally accepted as an authenticator):

```
EncryptByKey({<Key GUID>, '<string to be encrypted>'|<string variable>},
    [<add authenticator value>, '<authentication hash>'|<string variable>])
```

Note that if you provide an add authenticator value (in the form of an `int`), that value must be supplied when the string is decrypted, and you must also supply a hash value (which again will be needed at decryption time).

EncryptByPassPhrase

This one encrypts data not by using a formal key, but by a passphrase. Other than accepting a passphrase parameter instead of assuming an open key, `EncryptByPassPhrase` works almost exactly like `EncryptByKey`:

```
EncryptByPassPhrase({'<passphrase>'|<string variable>},
    {'<string to be encrypted>'|<string variable>},
    [<add authenticator value>, '<authentication hash>'|<string variable>])
```

As with `EncryptByKey`, if you provide an add authenticator value (in the form of an `int`), that value must be supplied when the string is decrypted, and you must also supply a hash value.

Key_GUID

Fetches the GUID for a given symmetric key in the current database:

```
Key_GUID('<Key Name>')
```

Key_ID

Fetches the ID for a given symmetric key in the current database:

```
Key_ID('<Key Name>')
```

SignByAsymKey

Adds an asymmetric key signature to a given plain text value:

```
SignByAsymKey(<Asymmetric Key ID>, <string variable> [, '<password>'])
```

SignByCert

Returns a `varbinary(8000)` containing the resulting signature provided a given certificate and plain text value:

```
SignByCert(<Certificate ID>, <string variable> [, '<password>'])
```

VerifySignedByAsymKey

Returns an `int` (again, I think this odd because it is functionally a bit) indicating successful or failed validation of a signature against a given asymmetric key and plain text value:

```
VerifySignedByAsymKey(<Asymmetric Key ID>, <plain text> , <signature>)
```

VerifySignedByCert

Returns an `int` (though, personally I think this odd because it is functionally a bit) indicating successful or failed validation of a signature against a given asymmetric key and plain text value:

```
VerifySignedByCert(<Certificate ID>, <signed plain text> , <signature>)
```

CURSOR FUNCTIONS

These functions provide information on the status or nature of a given cursor.

@@CURSOR_ROWS

Returns how many rows are currently in the last cursor set opened on the current connection. Note that this is for cursors, not temporary tables.

Keep in mind that this number is reset every time you open a new cursor. If you need to open more than one cursor at a time, and you need to know the number of rows in the first cursor, you'll need to move this value into a holding variable before opening subsequent cursors.

It's possible to use this to set up a counter to control your `WHILE` loop when dealing with cursors, but I strongly recommend against this practice — the value contained in `@@CURSOR_ROWS` can change depending on the cursor type and whether SQL Server is populating the cursor asynchronously. Using `@@FETCH_STATUS` is going to be far more reliable and at least as easy to use.

If the value returned is a negative number larger than –1, you must be working with an asynchronous cursor, and the negative number is the number of records so far created in the cursor. If, however, the value is `-1`, the cursor is a dynamic cursor, in that the number of rows is constantly changing. A returned value of `0` informs you that either no cursor has been opened, or the last cursor opened is no longer open. Finally, any positive number indicates the number of rows within the cursor.

NOTE *To create an asynchronous cursor, set* `sp_configure cursor threshold` *to a value greater than* `0`*. Then, when the cursor exceeds this setting, the cursor is returned, while the remaining records are placed into the cursor asynchronously.*

@@FETCH_STATUS

Returns an indicator of the status of the last cursor `FETCH` operation.

If you're using cursors, you're going to be using `@@FETCH_STATUS`. This one is how you know the success or failure of your attempt to navigate to a record in your cursor. It will return a constant depending on whether SQL Server succeeded in your last `FETCH` operation, and, if the `FETCH` failed, why. The constants are:

- `0`: Success.
- `-1`: Failed. Usually because you are beyond either the beginning or end of the cursorset.
- `-2`: Failed. The row you were fetching wasn't found, usually because it was deleted between the time when the cursorset was created and when you navigated to the current row. Should occur only in scrollable, non-dynamic cursors.

For purposes of readability, I often will set up some constants prior to using `@@FETCH_STATUS`.

For example:

```
DECLARE @NOTFOUND int
DECLARE @BEGINEND int

SELECT @NOTFOUND = -2
SELECT @BEGINEND = -1
```

I can then use these in my conditional in the `WHILE` statement of my cursor loop instead of just the row integer. This can make the code quite a bit more readable.

CURSOR_STATUS

The `CURSOR_STATUS` function allows the caller of a stored procedure to determine whether that procedure has returned a cursor and result set. The syntax is as follows:

```
CURSOR_STATUS
   (
      {'<local>', '<cursor name>'}
      | {'<global'>, '<cursor name>'}
      | {'<variable>', '<cursor variable>'}
   )
```

`local`, `global`, and `variable` all specify constants that indicate the source of the cursor. `local` equates to a local cursor name, `global` to a global cursor name, and `variable` to a local variable.

If you are using the `cursor name` form, there are four possible return values:

- `1`: The cursor is open. If the cursor is dynamic, its result set has zero or more rows. If the cursor is not dynamic, it has one or more rows.
- `0`: The result set of the cursor is empty.
- `-1`: The cursor is closed.
- `-3`: A cursor of `cursor name` does not exist.

If you are using the `cursor variable` form, there are five possible return values:

- `1`: The cursor is open. If the cursor is dynamic, its result set has zero or more rows. If the cursor is not dynamic, it has one or more rows.
- `0`: The result set is empty.
- `-1`: The cursor is closed.
- `-2`: There is no cursor assigned to the `cursor variable`.
- `-3`: The variable with name `cursor variable` does not exist, or if it does exist, has not had a cursor allocated to it yet.

DATE AND TIME FUNCTIONS

This is an area with several items introduced in SQL Server 2008. In addition to working with timestamp data (which is actually more oriented toward versioning than anything to do with a clock or calendar), date and time functions perform operations on values that have any of the various date and time data types supported by SQL Server.

When working with many of these functions, SQL Server recognizes eleven "dateparts" and their abbreviations, as shown in Table B-3.

TABLE B-3: Dateparts and their abbreviations

DATEPART	ABBREVIATIONS
`Year`	`yy, yyyy`
`Quarter`	`qq, q`
`Month`	`mm, m`
`Dayofyear`	`dy, y`
`Day`	`dd, d`
`Week`	`wk, ww`
`Weekday`	`dw`
`Hour`	`hh`

DATEPART	ABBREVIATIONS
Minute	mi, n
Second	ss, s
Millisecond	ms

CURRENT_TIMESTAMP

The `CURRENT_TIMESTAMP` function simply returns the current date and time as a `datetime` type. It is equivalent to `GETDATE()`. The syntax is as follows:

```
CURRENT_TIMESTAMP
```

DATEADD

The `DATEADD` function adds an interval to a date and returns a new date. The syntax is as follows:

```
DATEADD(<datepart>, <number>, <date>)
```

The *`datepart`* argument specifies the time scale of the interval (day, week, month, and so on) and may be any of the dateparts recognized by SQL Server. The `number` argument is the number of dateparts that should be added to the `date`.

DATEDIFF

The `DATEDIFF` function returns the difference between two specified dates in a specified unit of time (for example: hours, days, weeks). The syntax is as follows:

```
DATEDIFF(<datepart>, <startdate>, <enddate>)
```

The `datepart` argument may be any of the dateparts recognized by SQL Server and specifies the unit of time to be used.

DATETIMEFROMPARTS

You can assemble any of the datetime data types from parts. Versions of this function and their return types include:

- **DATEFROMPARTS** returns `DATE`
- **DATETIMEFROMPARTS** returns `DATETIME`
- **DATETIME2FROMPARTS** returns `DATETIME2`
- **DATETIMEOFFSETFROMPARTS** returns `DATETIMEOFFSET`
- **SMALLDATETIMEFROMPARTS** returns `SMALLDATETIME`
- **TIMEFROMPARTS** returns `TIME`

Each of these functions assembles its output from some combination of the following parts, passed as arguments.

- **`year`:** Integer expression specifying a year.
- **`month`:** Integer expression specifying a month.
- **`day`:** Integer expression specifying a day.
- **`hour`:** Integer expression specifying hours.
- **`minute`:** Integer expression specifying minutes.
- **`seconds`:** Integer expression specifying seconds.
- **`fractions`:** Integer expression specifying fractions (`milliseconds` for `DATETIMEFROMPARTS`).
- **`hour_offset`**: Integer expression specifying the hour portion of the time zone offset.
- **`minute_offset`**: Integer expression specifying the minute portion of the time zone offset.
- **`precision`:** Integer literal specifying the precision of the `datetimeoffset` value to be returned. If the precision argument is omitted, a default precision of 7 is used.

Most of these are self-explanatory, but `fractions` are unique. `Fractions` are fractions of a second, and the size of a fraction depends on the `precision` parameter you pass. A `TIME` with a `precision` of 7 has a grain of 100 nanoseconds (the seventh decimal place), so `fractions` in that context are in 100 nanosecond increments. A `precision` of 3 is in milliseconds (three decimal places), so `fractions` are milliseconds. If you specify a `precision` of 0, though, you must specify `fractions` as 0 as well or an error will result.

As you can see, the syntax for each version of this function is similar:

```
DATEFROMPARTS ( year, month, day )

DATETIMEFROMPARTS ( year, month, day, hour, minute, seconds, milliseconds )

DATETIME2FROMPARTS ( year, month, day, hour, minute, seconds, fractions,
precision )

DATETIMEOFFSETFROMPARTS ( year, month, day, hour, minute, seconds, fractions,
hour_offset, minute_offset, precision )

SMALLDATETIMEFROMPARTS ( year, month, day, hour, minute )

TIMEFROMPARTS ( hour, minute, seconds, fractions, precision )
```

DATENAME

The `DATENAME` function returns a string representing the name of the specified `datepart` (for example: 1999, Thursday, July) of the specified `date`. The syntax is as follows:

```
DATENAME(<datepart>, <date>)
```

DATEPART

The `DATEPART` function returns an integer that represents the specified `datepart` of the specified `date`. The syntax is as follows:

```
DATEPART(<datepart>, <date>)
```

The `DAY` function is equivalent to `DATEPART(dd, <date>)`; `MONTH` is equivalent to `DATEPART(mm, <date>)`; `YEAR` is equivalent to `DATEPART(yy, <date>)`.

DAY

The `DAY` function returns an integer representing the day part of the specified date. The syntax is as follows:

```
DAY(<date>)
```

The `DAY` function is equivalent to `DATEPART(dd, <date>)`.

EOMONTH

EOMONTH returns the last date in the month found in its `date` parameter. It looks like this:

```
EOMONTH( date_value )
```

GETDATE

The `GETDATE` function returns the current system date and time. The syntax is as follows:

```
GETDATE()
```

GETUTCDATE

The `GETUTCDATE` function returns the current UTC (Universal Time Coordinate) time. In other words, this returns GMT (Greenwich Mean Time). The value is derived by taking the local time from the server, and the local time zone, and calculating GMT from this. Daylight saving is included. `GETUTCDATE` cannot be called from a user-defined function. The syntax is as follows:

```
GETUTCDATE()
```

ISDATE

The `ISDATE` function determines whether an input expression is a valid date. The syntax is as follows:

```
ISDATE(<expression>)
```

MONTH

The MONTH function returns an integer that represents the month part of the specified date. The syntax is as follows:

```
MONTH(<date>)
```

The MONTH function is equivalent to DATEPART(mm, <date>).

SYSDATETIME

Much like the more venerable GETDATE function, SYSDATETIME returns the current system date and time. The differences are twofold: First, SYSDATETIME returns a higher level of precision. Second, the newer function returns the newer datetime2 data type (to support the higher precision — a precision of 7 in this case). The syntax is as follows:

```
SYSDATETIME()
```

SYSDATETIMEOFFSET

Similar to SYSDATETIME, this returns the current system date and time. Instead of the simple datetime2 data type, however, SYSDATETIMEOFFSET returns the time in the new datetimeoffset data type (with a precision of 7), thus providing offset information versus universal time. The syntax is as follows:

```
SYSDATETIMEOFFSET()
```

SYSUTCDATETIME

Much like the more venerable GETUTCDATE function, SYSDATETIME returns the current UTC date and time. SYSDATETIME, however, returns the newer datetime2 data type (to a precision of 7). The syntax is as follows:

```
SYSUTCDATETIME()
```

SWITCHOFFSET

Returns a datetimeoffset value that is changed from the stored time zone offset to a specified new time zone offset.

So, for example, you can see how the offset gets applied in an example:

```
  SWITCHOFFSET ( <datetime to offset>, <time zone offset amount> )
CREATE TABLE dbo.test
   (
   ColDatetimeoffset datetimeoffset
   );
GO
```

```
INSERT INTO dbo.test
VALUES ('1998-09-20 7:45:50.71345 -5:00');
GO
SELECT SWITCHOFFSET (ColDatetimeoffset, '-08:00')
FROM dbo.test;
GO
--Returns: 1998-09-20 04:45:50.7134500 -08:00
SELECT ColDatetimeoffset
FROM dbo.test;
--Returns: 1998-09-20 07:45:50.7134500 -05:00
```

TODATETIMEOFFSET

Accepts a given piece of date/time information and adds a provided time offset to produce a `datetimeoffset` data type. The syntax is:

```
TODATETIMEOFFSET(<data that resolves to datetime>, <time zone>)
```

So, for example:

```
DECLARE @OurDateTimeTest datetime;
SELECT @OurDateTimeTest = '2008-01-01 12:54';
SELECT TODATETIMEOFFSET(@OurDateTimeTest, '-07:00');
```

Yields:

```
----------------------------------
1/1/2008 12:54:00 PM -07:00

(local)(sa): (1 row(s) affected)
```

YEAR

The `YEAR` function returns an integer that represents the year part of the specified date. The syntax is as follows:

```
YEAR(<date>)
```

The `YEAR` function is equivalent to `DATEPART(yy, <date>)`.

HIERARCHY FUNCTIONS

As mentioned before, the new implementation of hierarchyIDs is somewhat beyond the scope of this title. For detailed explanations on the use of the following functions, please refer to the HierarchyId data type method reference in Books Online. This section simply lists, and briefly describes, the functions provided by SQL Server 2012 to assist in working with the hierarchyIDs and the trees associated with them.

GetAncestor

The `GetAncestor` function returns the hierarchyID of n^{th} ancestor of the item in question. The syntax is as follows:

```
GetAncestor(<numeric expression>)
```

The parameter passed in determines how many levels up the hierarchy chain you want to retrieve descendants from. For example, 1 returns children of the item in question, 2 returns the grandchildren, and 0 returns the hierarchyID of the item in question.

GetDescendant

Returns a child node of the parent.

GetLevel

Returns an integer that represents the depth of the node in the tree.

GetRoot

Returns the root of the hierarchy tree.

IsDescendantOf

Returns `true` if the child is a descendant of the item in question.

Parse

Converts the canonical string representation of a hierarchyID to a hierarchyID value. Parse is called implicitly when a conversion from a string type to hierarchyID occurs.

GetReparentedValue

Returns a node whose path from the root is the path to the `newRoot`, followed by the path from the `oldRoot` to the current item.

ToString

The functional equivalent of using a `CAST` function, this returns a string typed value of the hierarchy node you apply the function to.

MATHEMATICAL FUNCTIONS

The mathematical functions perform calculations. They are:

- `ABS`
- `ACOS`

- ASIN
- ATAN
- ATN2
- CEILING
- COS
- COT
- DEGREES
- EXP
- FLOOR
- LOG
- LOG10
- PI
- POWER
- RADIANS
- RAND
- ROUND
- SIGN
- SIN
- SQRT
- SQUARE
- TAN

ABS

The `ABS` function returns the positive, absolute value of `numeric expression`. The syntax is as follows:

```
ABS(<numeric expression>)
```

ACOS

The `ACOS` function returns the angle in radians for which the cosine is the `expression` (in other words, it returns the arccosine of `expression`). The syntax is as follows:

```
ACOS(<expression>)
```

The value of `expression` must be between `-1` and `1` and be of the `float` data type.

ASIN

The `ASIN` function returns the angle in radians for which the sine is the `expression` (in other words, it returns the arcsine of `expression`). The syntax is as follows:

```
ASIN(<expression>)
```

The value of `expression` must be between `-1` and `1` and be of the `float` data type.

ATAN

The `ATAN` function returns the angle in radians for which the tangent is `expression` (in other words, it returns the arctangent of `expression`). The syntax is as follows:

```
ATAN(<expression>)
```

The `expression` must be of the `float` data type.

ATN2

The `ATN2` function returns the angle in radians for which the tangent is between the two expressions provided (in other words, it returns the arctangent of the two expressions). The syntax is as follows:

```
ATN2(<expression1>, <expression2>)
```

Both `expression1` and `expression2` must be of the `float` data type.

CEILING

The `CEILING` function returns the smallest integer that is equal to or greater than the specified expression. The syntax is as follows:

```
CEILING(<expression>)
```

COS

The `COS` function returns the cosine of the angle specified in `expression`. The syntax is as follows:

```
COS(<expression>)
```

The angle given should be in radians and `expression` must be of the `float` data type.

COT

The `COT` function returns the cotangent of the angle specified in `expression`. The syntax is as follows:

```
COT(<expression>)
```

The angle given should be in radians and `expression` must be of the `float` data type.

DEGREES

The DEGREES function takes an angle given in radians (expression) and returns the angle in degrees. The syntax is as follows:

```
DEGREES(<expression>)
```

EXP

The EXP function returns the exponential value of the value given in expression. The syntax is as follows:

```
EXP(<expression>)
```

The expression must be of the float data type.

FLOOR

The FLOOR function returns the largest integer that is equal to or less than the value specified in expression. The syntax is as follows:

```
FLOOR(<expression>)
```

LOG

The LOG function returns the natural logarithm of the value specified in expression. The syntax is as follows:

```
LOG(<expression>)
```

The expression must be of the float data type.

LOG10

The LOG10 function returns the base10 logarithm of the value specified in expression. The syntax is as follows:

```
LOG10(<expression>)
```

The expression must be of the float data type.

PI

The PI function returns the value of the constant. The syntax is as follows:

```
PI()
```

POWER

The `POWER` function raises the value of the specified `expression` to the specified `power`. The syntax is as follows:

```
POWER(<expression>, <power>)
```

RADIANS

The `RADIANS` function returns an angle in radians corresponding to the angle in degrees specified in `expression`. The syntax is as follows:

```
RADIANS(<expression>)
```

RAND

The `RAND` function returns a random value between `0` and `1`. The syntax is as follows:

```
RAND([<seed>])
```

The optional `seed` value is an integer expression, which specifies a start value to be used in random number generation. You can allow SQL Server to generate its own random seed value each time you call it, or you can specify your own.

NOTE *Be careful when specifying a literal seed value. Given a specific seed value, SQL Server will always return the same result (meaning your* `RAND` *is suddenly not random at all). If you truly need a random number generated, either allow SQL Server to select a random seed for you, or use a constantly changing seed such as the number of nanoseconds since a particular point in time.*

ROUND

The `ROUND` function takes a number specified in `expression` and rounds it to the specified length:

```
ROUND(<expression>, <length> [, <function>])
```

The `length` parameter specifies the precision to which `expression` should be rounded. The `length` parameter should be of the `tinyint`, `smallint`, or `int` data type. The optional `function` parameter can be used to specify whether the number should be rounded or truncated. If a `function` value is omitted or is equal to `0` (the default), the value in `expression` will be rounded. If any value other than `0` is provided, the value in `expression` will be truncated.

SIGN

The `SIGN` function returns the sign of the `expression`. The possible return values are `+1` for a positive number, `0` for zero, and `-1` for a negative number. The syntax is as follows:

```
SIGN(<expression>)
```

SIN

The `SIN` function returns the sine of an angle. The syntax is as follows:

```
SIN(<angle>)
```

The `angle` should be in radians and must be of the `float` data type. The return value will also be of the `float` data type.

SQRT

The `SQRT` function returns the square root of the value given in `expression`. The syntax is as follows:

```
SQRT(<expression>)
```

The `expression` must be of the `float` data type.

SQUARE

The `SQUARE` function returns the square of the value given in `expression`. The syntax is as follows:

```
SQUARE(<expression>)
```

The `expression` must be of the `float` data type.

TAN

The `TAN` function returns the tangent of the value specified in `expression`. The syntax is as follows:

```
TAN(<expression>)
```

The `expression` parameter specifies the number of radians and must be of the `float` or `real` data type.

BASIC METADATA FUNCTIONS

The metadata functions provide information about the database and database objects. They are:

- `COL_LENGTH`
- `COL_NAME`
- `COLUMNPROPERTY`

- `DATABASEPROPERTY`
- `DATABASEPROPERTYEX`
- `DB_ID`
- `DB_NAME`
- `FILE_ID`
- `FILE_NAME`
- `FILEGROUP_ID`
- `FILEGROUP_NAME`
- `FILEGROUPPROPERTY`
- `FILEPROPERTY`
- `FULLTEXTCATALOGPROPERTY`
- `FULLTEXTSERVICEPROPERTY`
- `INDEX_COL`
- `INDEXKEY_PROPERTY`
- `INDEXPROPERTY`
- `OBJECT_ID`
- `OBJECT_NAME`
- `OBJECTPROPERTY`
- `OBJECTPROPERTYEX`
- `@@PROCID`
- `SCHEMA_ID`
- `SCHEMA_NAME`
- `SQL_VARIANT_PROPERTY`
- `TYPE_ID`
- `TYPE_NAME`
- `TYPEPROPERTY`

COL_LENGTH

The `COL_LENGTH` function returns the defined length of a column. The syntax is as follows:

```
COL_LENGTH('<table>', '<column>')
```

The `column` parameter specifies the name of the column for which the length is to be determined. The `table` parameter specifies the name of the table that contains that column.

COL_NAME

The `COL_NAME` function takes a table ID number and a column ID number and returns the name of the database column. The syntax is as follows:

```
COL_NAME(<table_id>, <column_id>)
```

The `column_id` parameter specifies the ID number of the column. The `table_id` parameter specifies the ID number of the table that contains that column.

COLUMNPROPERTY

The `COLUMNPROPERTY` function returns data about a column or procedure parameter. The syntax is as follows:

```
COLUMNPROPERTY(<id>, <column>, <property>)
```

The `id` parameter specifies the ID of the table/procedure. The `column` parameter specifies the name of the column/parameter. The `property` parameter specifies the data that should be returned for the column or procedure parameter. The `property` parameter can be one of the following values:

- ➤ `AllowsNull`: Allows `NULL` values.
- ➤ `IsComputed`: The column is a computed column.
- ➤ `IsCursorType`: The procedure is of type `CURSOR`.
- ➤ `IsFullTextIndexed`: The column has been full-text indexed.
- ➤ `IsIdentity`: The column is an `IDENTITY` column.
- ➤ `IsIdNotForRepl`: The column checks for `IDENTITY NOT FOR REPLICATION`.
- ➤ `IsOutParam`: The procedure parameter is an output parameter.
- ➤ `IsRowGuidCol`: The column is a `ROWGUIDCOL` column.
- ➤ `Precision`: The precision for the data type of the column or parameter.
- ➤ `Scale`: The scale for the data type of the column or parameter.
- ➤ `UseAnsiTrim`: The ANSI padding setting was `ON` when the table was created.

The return value from this function will be `1` for `True`, `0` for `False`, and `NULL` if the input was not valid — except for `Precision` (where the precision for the data type will be returned) and `Scale` (where the scale will be returned).

DATABASEPROPERTY

The `DATABASEPROPERTY` function returns the setting for the specified database and property name. The syntax is as follows:

```
DATABASEPROPERTY('<database>', '<property>')
```

The `database` parameter specifies the name of the database for which data on the named property will be returned. The `property` parameter contains the name of a database property and can be one of the values in Table B-4.

TABLE B-4: Database Property Values

VALUE	DESCRIPTION
`IsAnsiNullDefault`	The database follows the ANSI-92 standard for `NULL` values.
`IsAnsiNullsEnabled`	All comparisons made with a `NULL` cannot be evaluated.
`IsAnsiWarningsEnabled`	Warning messages are issued when standard error conditions occur.
`IsAutoClose`	The database frees resources after the last user has exited.
`IsAutoShrink`	Database files can be shrunk automatically and periodically.
`IsAutoUpdateStatistics`	The autoupdate statistics option has been enabled.
`IsBulkCopy`	The database allows non-logged operations (such as those performed with the Bulk Copy Program).
`IsCloseCursorsOnCommitEnabled`	Any cursors that are open when a transaction is committed will be closed.
`IsDboOnly`	The database is only accessible to the `dbo`.
`IsDetached`	The database was detached by a `detach` operation.
`IsEmergencyMode`	The database is in emergency mode.
`IsFulltextEnabled`	The database has been full-text enabled.
`IsInLoad`	The database is loading.
`IsInRecovery`	The database is recovering.
`IsInStandby`	The database is read-only and restore log is allowed.
`IsLocalCursorsDefault`	Cursor declarations default to `LOCAL`.
`IsNotRecovered`	The database failed to recover.
`IsNullConcat`	Concatenating to a `NULL` results in a `NULL`.
`IsOffline`	The database is offline.
`IsQuotedIdentifiersEnabled`	Identifiers can be delimited by double quotation marks.
`IsReadOnly`	The database is in a read-only mode.

VALUE	DESCRIPTION
`IsRecursiveTriggersEnabled`	The recursive firing of triggers is enabled.
`IsShutDown`	The database encountered a problem during startup.
`IsSingleUser`	The database is in single-user mode.
`IsSuspect`	The database is suspect.
`IsTruncLog`	The database truncates its logon checkpoints.
`Version`	The internal version number of the SQL Server code with which the database was created.

The return value from this function will be `1` for true, `0` for false, and `NULL` if the input was not valid — except for `Version` (where the function will return the version number if the database is open and `NULL` if the database is closed).

DATABASEPROPERTYEX

The `DATABASEPROPERTYEX` function is basically a superset of `DATABASEPROPERTY`, and also returns the setting for the specified database and property name. The syntax is pretty much just the same as `DATABASEPROPERTY` and is as follows:

```
DATABASEPROPERTYEX('<database>', '<property>')
```

`DATABASEPROPERTYEX` just has a few more properties available, including:

- `Collation`: Returns the default collation for the database (remember, collations can also be overridden at the column level).
- `ComparisonStyle`: Indicates the Windows comparison style (for example, case sensitivity) of the particular collation.
- `IsAnsiPaddingEnabled`: Whether strings are padded to the same length before comparison or insert.
- `IsArithmaticAbortEnabled`: Whether queries are terminated when a data overflow or divide-by-zero error occurs.

The `database` parameter specifies the name of the database for which data on the named property will be returned. The `property` parameter contains the name of a database property and can be one of the following values.

DB_ID

The `DB_ID` function returns the database ID number. The syntax is as follows:

```
DB_ID(['<database name>'])
```

The optional `database_name` parameter specifies which database's ID number is required. If the `database_name` is not given, the current database will be used instead.

DB_NAME

The `DB_NAME` function returns the name of the database that has the specified ID number. The syntax is as follows:

```
DB_NAME([<database_id>])
```

The optional `database_id` parameter specifies which database's name is to be returned. If no `database_id` is given, the name of the current database will be returned.

FILE_ID

The `FILE_ID` function returns the file ID number for the specified filename in the current database. The syntax is as follows:

```
FILE_ID('<file_name>')
```

The `file_name` parameter specifies the name of the file for which the ID is required.

FILE_NAME

The `FILE_NAME` function returns the filename for the file with the specified file ID number. The syntax is as follows:

```
FILE_NAME(<file_id>)
```

The `file_id` parameter specifies the ID number of the file for which the name is required.

FILEGROUP_ID

The `FILEGROUP_ID` function returns the filegroup ID number for the specified filegroup name. The syntax is as follows:

```
FILEGROUP_ID('<filegroup_name>')
```

The `filegroup_name` parameter specifies the filegroup name of the required filegroup ID.

FILEGROUP_NAME

The `FILEGROUP_NAME` function returns the filegroup name for the specified filegroup ID number. The syntax is as follows:

```
FILEGROUP_NAME(<filegroup_id>)
```

The `filegroup_id` parameter specifies the filegroup ID of the required filegroup name.

FILEGROUPPROPERTY

The `FILEGROUPPROPERTY` returns the setting of a specified filegroup property, given the filegroup and property name. The syntax is as follows:

```
FILEGROUPPROPERTY(<filegroup_name>, <property>)
```

The `filegroup_name` parameter specifies the name of the filegroup that contains the property being queried. The `property` parameter specifies the property being queried and can be one of the following values:

- `IsReadOnly`: The filegroup name is read-only.
- `IsUserDefinedFG`: The filegroup name is a user-defined filegroup.
- `IsDefault`: The filegroup name is the default filegroup.

The return value from this function is `1` for `True`, `0` for `False`, and `NULL` if the input was not valid.

FILEPROPERTY

The `FILEPROPERTY` function returns the setting of a specified filename property, given the filename and property name. The syntax is as follows:

```
FILEPROPERTY(<file_name>, <property>)
```

The `file_name` parameter specifies the name of the filegroup that contains the property being queried. The `property` parameter specifies the property being queried and can be one of the following values:

- `IsReadOnly`: The file is read-only.
- `IsPrimaryFile`: The file is the primary file.
- `IsLogFile`: The file is a log file.
- `SpaceUsed`: The amount of space used by the specified file.

The return value from this function is `1` for `True`, `0` for `False`, and `NULL` if the input was not valid, except for `SpaceUsed` (which will return the number of pages allocated in the file).

FULLTEXTCATALOGPROPERTY

The `FULLTEXTCATALOGPROPERTY` function returns data about the full-text catalog properties. The syntax is as follows:

```
FULLTEXTCATALOGPROPERTY(<catalog_name>, <property>)
```

The `catalog_name` parameter specifies the name of the full-text catalog. The `property` parameter specifies the property that is being queried. The properties that can be queried are:

- `PopulateStatus`: For which the possible return values are `0` (idle), `1` (population in progress), `2` (paused), `3` (throttled), `4` (recovering), `5` (shutdown), `6` (incremental population in progress), and `7` (updating index).

- `ItemCount`: Returns the number of full-text indexed items currently in the full-text catalog.
- `IndexSize`: Returns the size of the full-text index in megabytes.
- `UniqueKeyCount`: Returns the number of unique words that make up the full-text index in this catalog.
- `LogSize`: Returns the size (in bytes) of the combined set of error logs associated with a full-text catalog.
- `PopulateCompletionAge`: Returns the difference (in seconds) between the completion of the last full-text index population and 01/01/1990 00:00:00.

FULLTEXTSERVICEPROPERTY

The `FULLTEXTSERVICEPROPERTY` function returns data about the full-text service-level properties. The syntax is as follows:

```
FULLTEXTSERVICEPROPERTY(<property>)
```

The `property` parameter specifies the name of the service-level property that is to be queried. The `property` parameter may be one of the following values:

- `ResourceUsage`: Returns a value from `1` (background) to `5` (dedicated).
- `ConnectTimeOut`: Returns the number of seconds that the Search Service will wait for all connections to SQL Server for full-text index population before timing out.
- `IsFulltextInstalled`: Returns `1` if Full-Text Service is installed on the computer and a `0` otherwise.

INDEX_COL

The `INDEX_COL` function returns the indexed column name. The syntax is as follows:

```
INDEX_COL('<table>', <index_id>, <key_id>)
```

The `table` parameter specifies the name of the table, `index_id` specifies the ID of the index, and `key_id` specifies the ID of the key.

INDEXKEY_PROPERTY

This function returns information about the index key.

```
INDEXKEY_PROPERTY(<table_id>, <index_id>, <key_id>, <property>)
```

The `table_id` parameter is the numerical ID of data type `int`, which defines the table you want to inspect. Use `OBJECT_ID` to find the numerical `table_id`. `index_id` specifies the ID of the index, and is also of data type `int`. `key_id` specifies the index column position of the key; for example, with a key of three columns, setting this value to `2` will determine that you are wanting to inspect the middle column. Finally, the `property` is the character string identifier of one of two properties you want to find the setting of. The two possible values are `ColumnId`, which will return the physical column ID, and `IsDescending`, which returns the order that the column is sorted (`1` is for descending and `0` is ascending).

INDEXPROPERTY

The `INDEXPROPERTY` function returns the setting of a specified index property, given the table ID, index name, and property name. The syntax is as follows:

```
INDEXPROPERTY(<table ID>, <index>, <property>)
```

The `property` parameter specifies the property of the index that is to be queried. The `property` parameter can be one of these possible values:

- `IndexDepth`: The depth of the index.
- `IsAutoStatistic`: The index was created by the autocreate statistics option of `sp_dboption`.
- `IsClustered`: The index is clustered.
- `IsStatistics`: The index was created by the `CREATE STATISTICS` statement or by the auto-create statistics option of `sp_dboption`.
- `IsUnique`: The index is unique.
- `IndexFillFactor`: The index specifies its own fill factor.
- `IsPadIndex`: The index specifies space to leave open on each interior node.
- `IsFulltextKey`: The index is the full-text key for a table.
- `IsHypothetical`: The index is hypothetical and cannot be used directly as a data access path.

The return value from this function will be `1` for `True`, `0` for `False`, and `NULL` if the input was not valid, except for `IndexDepth` (which will return the number of levels the index has) and `IndexFillFactor` (which will return the fill factor used when the index was created or last rebuilt).

OBJECT_ID

The `OBJECT_ID` function returns the specified database object's ID number. The syntax is as follows:

```
OBJECT_ID('<object>')
```

OBJECT_NAME

The `OBJECT_NAME` function returns the name of the specified database object. The syntax is as follows:

```
OBJECT_NAME(<object id>)
```

OBJECTPROPERTY

The `OBJECTPROPERTY` function returns data about objects in the current database. The syntax is as follows:

```
OBJECTPROPERTY(<id>, <property>)
```

The `id` parameter specifies the ID of the object required. The `property` parameter specifies the information required on the object. The following `property` values are allowed:

CnstIsClustKey	HasUpdateTrigger
CnstIsColumn	IsAnsiNullsOn
CnstIsDeleteCascade	IsCheckCnst
CnstIsDisabled	IsConstraint
CnstIsNonclustKey	IsDefault
CnstIsNotRepl	IsDefaultCnst
CnstIsNotTrusted	IsDeterministic
CnstIsUpdateCascade	IsExecuted
ExecIsAfterTrigger	IsExtendedProc
ExecIsAnsiNullsOn	IsForeignKey
ExecIsDeleteTrigger	IsIndexable
ExecIsFirstDeleteTrigger	IsIndexed
ExecIsFirstInsertTrigger	IsInlineFunction
ExecIsFirstUpdateTrigger	IsMSShipped
ExecIsInsertTrigger	IsPrimaryKey
ExecIsInsteadOfTrigger	IsProcedure
ExecIsLastDeleteTrigger	IsQueue
ExecIsLastInsertTrigger	IsQuotedIdentOn
ExecIsLastUpdateTrigger	IsReplProc
ExecIsQuotedIdentOn	IsRule
ExecIsStartup	IsScalarFunction
ExecIsTriggerDisabled	IsSchemaBound
ExecIsTriggerNotForRepl	IsSystemTable
ExecIsUpdateTrigger	IsTable
HasAfterTrigger	IsTableFunction
HasDeleteTrigger	IsTrigger
HasInsertTrigger	IsUniqueCnst
HasInsteadOfTrigger	IsUserTable

```
IsView
OwnerId
TableDeleteTrigger
TableDeleteTriggerCount
TableFullTextBackgroundUpdateIndexOn
TableFulltextCatalogId
TableFullTextChangeTrackingOn
TableFulltextDocsProcessed
TableFulltextFailCount
TableFulltextItemCount
TableFulltextKeyColumn
TableFulltextPendingChanges
TableFulltextPopulateStatus
TableHasActiveFulltextIndex
TableHasCheckCnst
TableHasClustIndex
TableHasDefaultCnst
TableHasDeleteTrigger
TableHasForeignKey
```

```
TableHasForeignRef
TableHasIdentity
TableHasIndex
TableHasInsertTrigger
TableHasNonclustIndex
TableHasPrimaryKey
TableHasRowGuidCol
TableHasTextImage
TableHasTimestamp
TableHasUniqueCnst
TableHasUpdateTrigger
TableInsertTrigger
TableInsertTriggerCount
TableIsFake
TableIsLockedOnBulkLoad
TableIsPinned
TableTextInRowLimit
TableUpdateTrigger
TableUpdateTriggerCount
```

The return value from this function is `1` for `True`, `0` for `False`, and `NULL` if the input was not valid, except for:

- `OwnerId`: Returns the database user ID of the owner of that object — note that this is different from the SchemaID of the object and will likely not be that useful in SQL Server 2005 and beyond.
- `TableDeleteTrigger`, `TableInsertTrigger`, `TableUpdateTrigger`: Return the ID of the first trigger with the specified type. Zero is returned if no trigger of that type exists.
- `TableDeleteTriggerCount`, `TableInsertTriggerCount`, `TableUpdateTriggerCount`: Return the number of the specified type of trigger that exists for the table in question.
- `TableFulltextCatalogId`: Returns the ID of the full-text catalog if there is one, and zero if no full-text catalog exists for that table.
- `TableFulltextKeyColumn`: Returns the ColumnID of the column being utilized as the unique index for that full-text index.
- `TableFulltextPendingChanges`: The number of entries that have changed since the last full-text analysis was run for this table. Change tracking must be enabled for this function to return useful results.

- `TableFulltextPopulateStatus`: This one has multiple possible return values:
    - `0`: Indicates that the full-text process is currently idle.
    - `1`: A full population run is currently in progress.
    - `2`: An incremental population is currently running.
    - `3`: Changes are currently being analyzed and added to the full-text catalog.
    - `4`: Some form of background update (such as that done by the automatic change tracking mechanism) is currently running.
    - `5`: A full-text operation is in progress, but has either been throttled (to allow other system requests to perform as needed) or has been paused.

  You can use the feedback from this option to make decisions about what other full-text-related options are appropriate (to check whether a population is in progress so you know whether other functions, such as `TableFulltextDocsProcessed`, are valid).
- `TableFulltextDocsProcessed`: Valid only while full-text indexing is actually running, this returns the number of rows processed since the full-text index processing task started. A zero result indicates that full-text indexing is not currently running (a `NULL` result means full-text indexing is not configured for this table).
- `TableFulltextFailCount`: Valid only while full-text indexing is actually running, this returns the number of rows that full-text indexing has, for some reason, skipped (no indication of reason). As with `TableFulltextDocsProcessed:` A zero result indicates the table is not currently being analyzed for full text, and a `NULL` indicates that full text is not configured for this table.
- `TableIsPinned`: This is left in for backward compatibility only and will always return “0” in SQL Server 2005 and beyond.

OBJECTPROPERTYEX

`OBJECTPROPERTYEX` is an extended version of the `OBJECTPROPERTY` function.

```
OBJECTPROPERTYEX(<id>, <property>)
```

Like `OBJECTPROPERTY`, the `id` parameter specifies the ID of the object required. The `property` parameter specifies the information required on the object. `OBJECTPROPERTYEX` supports all the same property values as `OBJECTPROPERTY` but adds the following `property` values as additional options:

- `BaseType`: Returns the base data type of an object.
- `IsPrecise`: Indicates that your object does not contain any imprecise computations. For example, an `int` or `decimal` is precise, but a `float` is not — computations that utilize imprecise data types must be assumed to return imprecise results. Note that you can specifically mark any .NET assemblies you produce as being precise.
- `IsSystemVerified`: Indicates whether the `IsPrecise` and `IsDeterministic` properties can be verified by SQL Server itself (as opposed to just having been set by the user).

- `SchemaId`: Just what it sounds like — returns the internal system ID for a given object. You can then use `SCHEMA_NAME` to put a more user-friendly name on the schema ID.
- `SystemDataAccess`: Indicates whether the object in question relies on any system table data.
- `UserDataAccess`: Indicates whether the object in question utilizes any of the user tables or system user data.

@@PROCID

Returns the stored procedure ID of the currently running procedure.

Primarily a troubleshooting tool when a process is running and using up a large amount of resources. Is used mainly as a DBA function.

SCHEMA_ID

Given a schema name, returns the internal system ID for that schema. Utilizes the syntax:

```
SCHEMA_ID( <schema name> )
```

SCHEMA_NAME

Given an internal schema system ID, returns the user-friendly name for that schema. The syntax is:

```
SCHEMA_NAME( <schema id> )
```

SQL_VARIANT_PROPERTY

`SQL_VARIANT_PROPERTY` is a powerful function and returns information about an `sql_variant`. This information could be from `BaseType`, `Precision`, `Scale`, `TotalBytes`, `Collation`, or `MaxLength`. The syntax is:

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

`Expression` is an expression of type `sql_variant`. `Property` can be any one of the values listed in Table B-5.

TABLE B-5: Properties for SQL_VARIANT PROPERTY

VALUE	DESCRIPTION	BASE TYPE OF SQL_VARIANT RETURNED
`BaseType`	Data types include `char`, `int`, `money`, `nchar`, `ntext`, `numeric`, `nvarchar`, `real`, `smalldatetime`, `smallint`, `smallmoney`, `text`, `timestamp`, `tinyint`, `uniqueidentifier`, `varbinary`, `varchar`	`Sysname`

continues

TABLE B-5 *(continued)*

VALUE	DESCRIPTION	BASE TYPE OF SQL_VARIANT RETURNED
`Precision`	The precision of the numeric base data type: `datetime` = 23 `smalldatetime` = 16 `float` = 53 `real` = 24 `decimal` (p,s) and `numeric` (p,s) = p `money` = 19 `smallmoney` = 10 `int` = 10 `smallint` = 5 `tinyint` = 3 `bit` = 1 All other types = 0	`Int`
`Scale`	The number of digits to the right of the decimal point of the numeric base data type: `decimal` (p,s) and `numeric` (p,s) = s `money` and `smallmoney` = 4 `datetime` = 3 All other types = 0	`Int`
`TotalBytes`	The number of bytes required to hold both the metadata and data of the value. If the value is greater than 900, index creation will fail.	`Int`
`Collation`	The collation of the particular `sql_variant` value.	`Sysname`
`MaxLength`	The maximum data type length, in bytes.	`int`

TYPEPROPERTY

The `TYPEPROPERTY` function returns information about a data type. The syntax is as follows:

```
TYPEPROPERTY(<type>, <property>)
```

The `type` parameter specifies the name of the data type. The `property` parameter specifies the property of the data type that is to be queried; it can be one of the following values:

- `Precision`: Returns the number of digits/characters.
- `Scale`: Returns the number of decimal places.

- AllowsNull: Returns 1 for True and 0 for False.
- UsesAnsiTrim: Returns 1 for True and 0 for False.

ROWSET FUNCTIONS

The rowset functions return an object that can be used in place of a table reference in a T-SQL statement. The rowset functions are:

- CHANGETABLE
- CONTAINSTABLE
- FREETEXTTABLE
- OPENDATASOURCE
- OPENQUERY
- OPENROWSET
- OPENXML

CHANGETABLE

Returns change tracking information for a table. You can use this statement to return all changes for a table or change tracking information for a specific row. The syntax is as follows:

```
CHANGETABLE (
        { CHANGES table , last_sync_version
        | VERSION table , <primary_key_values> } )
[AS] table_alias [ ( column_alias [ ,...n ] )
```

CONTAINSTABLE

The CONTAINSTABLE function is used in full-text queries. The syntax is as follows:

```
CONTAINSTABLE (<table>, {<column> | *}, '<contains_search_condition>')
```

FREETEXTTABLE

The FREETEXTTABLE function is used in full-text queries. The syntax is as follows:

```
FREETEXTTABLE (<table>, {<column> | *}, '<freetext_string>')
```

OPENDATASOURCE

The OPENDATASOURCE function provides ad hoc connection information. The syntax is as follows:

```
OPENDATASOURCE (<provider_name>, <init_string>)
```

The provider_name is the name registered as the ProgID of the OLE DB provider used to access the data source. The init_string should be familiar to VB programmers, as this is the initialization string to the OLE DB provider. For example, the init_string could look like:

```
"User Id=wonderison;Password=JuniorBlues;DataSource=MyServerName"
```

OPENQUERY

The OPENQUERY function executes the specified pass-through query on the specified linked_server. The syntax is as follows:

```
OPENQUERY(<linked_server>, '<query>')
```

OPENROWSET

The OPENROWSET function accesses remote data from an OLE DB data source. The syntax is as follows:

```
OPENROWSET('<provider_name>'
   {
    '<datasource>';'<user id>';'<password>'
    | '<provider_string>'
   },
   {
      [<catalog.>][<schema.>]<object>
      | '<query>'
   })
```

The provider_name parameter is a string representing the friendly name of the OLE DB provided as specified in the registry. The data_source parameter is a string corresponding to the required OLE DB data source. The user_id parameter is a relevant username to be passed to the OLE DB provider. The password parameter is the password associated with the user_id.

The provider_string parameter is a provider-specific connection string and is used in place of the datasource, user_id, and password combination.

The catalog parameter is the name of catalog/database that contains the required object. The schema parameter is the name of the schema or object owner of the required object. The object parameter is the object name.

The query parameter is a string that is executed by the provider and is used instead of a combination of catalog, schema, and object.

OPENXML

By passing in an XML document as a parameter, or by retrieving an XML document and defining the document within a variable, OPENXML allows you to inspect the structure and return data, as if the XML document were a table. The syntax is as follows:

```
OPENXML(<idoc int> [in],<rowpattern> nvarchar[in],[<flags> byte[in]])
[WITH (<SchemaDeclaration> | <TableName>)]
```

The idoc_int parameter is the variable defined using the sp_xml_prepareddocument system sproc. Rowpattern is the node definition. The flags parameter specifies the mapping between the XML document and the rowset to return within the SELECT statement. SchemaDeclaration defines the XML schema for the XML document; if there is a table defined within the database that follows the XML schema, TableName can be used instead.

Before being able to use the XML document, you must prepare it by using the sp_xml_preparedocument system procedure.

SECURITY FUNCTIONS

The security functions return information about users and roles. They are:

- HAS_DBACCESS
- IS_MEMBER
- IS_SRVROLEMEMBER
- SUSER_ID
- SUSER_NAME
- SUSER_SID
- USER
- USER_ID
- USER_NAME

HAS_DBACCESS

The HAS_DBACCESS function determines whether the user who is logged in has access to the database being used. A return value of 1 means the user does have access, and a return value of 0 means that he or she does not. A NULL return value means the database_name supplied was invalid. The syntax is as follows:

```
HAS_DBACCESS ('<database_name>')
```

IS_MEMBER

The IS_MEMBER function returns whether the current user is a member of the specified Windows NT group/SQL Server role. The syntax is as follows:

```
IS_MEMBER ({'<group>' | '<role>'})
```

The group parameter specifies the name of the NT group and must be in the form domain\group. The role parameter specifies the name of the SQL Server role. The role can be a database fixed role or a user-defined role but cannot be a server role.

This function will return a `1` if the current user is a member of the specified group or role, a `0` if the current user is not a member of the specified group or role, and `NULL` if the specified group or role is invalid.

IS_SRVROLEMEMBER

The `IS_SRVROLEMEMBER` function returns whether a user is a member of the specified server role. The syntax is as follows:

```
IS_SRVROLEMEMBER ('<role>' [,'<login>'])
```

The optional `login` parameter is the name of the login account to check — the default is the current user. The `role` parameter specifies the server role and must be one of the following possible values:

- `sysadmin`
- `dbcreator`
- `diskadmin`
- `processadmin`
- `serveradmin`
- `setupadmin`
- `securityadmin`

This function returns a `1` if the specified login account is a member of the specified role, a `0` if the login is not a member of the role, and a `NULL` if the role or login is invalid.

SUSER_ID

The `SUSER_ID` function returns the specified user's login ID number. The syntax is as follows:

```
SUSER_ID(['<login>'])
```

The `login` parameter is the specified user's login ID name. If no value for `login` is provided, the default of the current user will be used instead.

NOTE *The* `SUSER_ID` *system function has long since been replaced by* `SUSER_SID`*, and remains in the product purely for backward compatibility purposes. Avoid using* `SUSER_ID`*, as the internal value it returns may change from server to server (the SID is much more reliable when you consider a database may be restored to a new server where a given login may have a different* `SUSER_ID`*).*

SUSER_NAME

The SUSER_NAME function returns the specified user's login ID name. The syntax is as follows:

```
SUSER_NAME([<server user id>])
```

The server_user_id parameter is the specified user's login ID number. If no value for server_ user_id is provided, the default of the current user will be used instead.

NOTE *The SUSER_NAME system function is included in SQL Server 2012 for backward compatibility only, so if possible you should use SUSER_SNAME instead.*

SUSER_SID

The SUSER_SID function returns the security identification number (SID) for the specified user. The syntax is as follows:

```
SUSER_SID(['<login>'])
```

The login parameter is the user's login name. If no value for login is provided, the current user will be used instead.

SUSER_SNAME

The SUSER_SNAME function returns the login ID name for the specified security identification number (SID). The syntax is as follows:

```
SUSER_SNAME([<server user sid>])
```

The server_user_sid parameter is the user's SID. If no value for the server_user_sid is provided, the current user's value is used instead.

USER

The USER function allows a system-supplied value for the current user's database username to be inserted into a table if no default has been supplied. The syntax is as follows:

```
USER
```

USER_ID

The USER_ID function returns the specified user's database ID number. The syntax is as follows:

```
USER_ID(['<user>'])
```

The user parameter is the username to be used. If no value for user is provided, the current user is used.

USER_NAME

The `USER_NAME` function is the functional reverse of `USER_ID`, and returns the specified user's username in the database given a database ID number. The syntax is as follows:

```
USER_NAME(['<user id>'])
```

The `user id` parameter is the ID of the user you want the name for. If no value for `user id` is provided, the current user is assumed.

STRING FUNCTIONS

The string functions perform actions on string values and return strings or numeric values. The string functions are:

- `ASCII`
- `CHAR`
- `CHARINDEX`
- `CONCAT`
- `DIFFERENCE`
- `FORMAT`
- `LEFT`
- `LEN`
- `LOWER`
- `LTRIM`
- `NCHAR`
- `PATINDEX`
- `QUOTENAME`
- `REPLACE`
- `REPLICATE`
- `REVERSE`
- `RIGHT`
- `RTRIM`
- `SOUNDEX`
- `SPACE`
- `STR`

- STUFF
- SUBSTRING
- UNICODE
- UPPER

ASCII

The ASCII function returns the ASCII code value of the leftmost character in character_ expression. The syntax is as follows:

```
ASCII(<character expression>)
```

CHAR

The CHAR function converts an ASCII code (specified in expression) into a string. The syntax is as follows:

```
CHAR(<expression>)
```

The expression can be any integer between 0 and 255.

CHARINDEX

The CHARINDEX function returns the starting position of an expression in a character_string. The syntax is as follows:

```
CHARINDEX(<expression>, <character string> [, <start location>])
```

The expression parameter is the string, which is to be found. The character_string is the string to be searched, usually a column. The start_location is the character position to begin the search, if this is anything other than a positive number, the search will begin at the start of character_string.

CONCAT

New in SQL Server 2012, CONCAT gives you a new way to assemble a string from parts that is simpler to use than the '+' operator. With CONCAT, you provide two or more parameters — they can be strings, or anything that can implicitly convert to a string — and CONCAT will concatenate them into a string, performing all the data type conversions for you.

```
CONCAT ( string_value1, string_value2 [, string_valueN ] )
```

DIFFERENCE

The DIFFERENCE function returns the difference between the SOUNDEX values of two expressions as an integer. The syntax is as follows:

```
DIFFERENCE(<expression1>, <expression2>)
```

This function returns an integer value between `0` and `4`. If the two expressions sound identical (for example, blue and blew) a value of `4` will be returned. If there is no similarity, a value of `0` is returned.

FORMAT

`FORMAT` takes data from most numeric or date/time data types and returns a formatted string based on the provided format parameter. The format parameter uses the same format strings as you can use in .NET, so not only is there a great variety, there are in fact so many possibilities that I'm not going to begin to list them here. Use Books Online or search MSDN for Formatting Types to get a complete online reference.

```
FORMAT ( value, format [, culture ] )
```

LEFT

The `LEFT` function returns the leftmost part of an expression, starting a specified number of characters from the left. The syntax is as follows:

```
LEFT(<expression>, <integer>)
```

The `expression` parameter contains the character data from which the leftmost section will be extracted. The `integer` parameter specifies the number of characters from the left to begin — it must be a positive integer.

LEN

The `LEN` function returns the number of characters in the specified `expression`. The syntax is as follows:

```
LEN(<expression>)
```

LOWER

The `LOWER` function converts any uppercase characters in the `expression` into lowercase characters. The syntax is as follows:

```
LOWER(<expression>)
```

LTRIM

The `LTRIM` function removes any leading blanks from a `character_expression`. The syntax is as follows:

```
LTRIM(<character_expression>)
```

NCHAR

The NCHAR function returns the Unicode character that has the specified integer_code. The syntax is as follows:

```
NCHAR(<integer_code>)
```

The integer_code parameter must be a positive whole number from 0 to 65,535.

PATINDEX

The PATINDEX function returns the starting position of the first occurrence of a pattern in a specified expression or zero if the pattern was not found. The syntax is as follows:

```
PATINDEX('<%pattern%>', <expression>)
```

The pattern parameter is a string that will be searched for. Wildcard characters can be used, but the % characters must surround the pattern. The expression parameter is character data in which the pattern is being searched for — usually a column.

QUOTENAME

The QUOTENAME function returns a Unicode string with delimiters added to make the specified string a valid SQL Server delimited identifier. The syntax is as follows:

```
QUOTENAME('<character_string>'[, '<quote_character>'])
```

The character_string parameter is a Unicode string. The quote_character parameter is a one-character string that will be used as a delimiter. The quote_character parameter can be a single quotation mark ('), a left or a right bracket ([]), or a double quotation mark (") — the default is for brackets to be used.

REPLACE

The REPLACE function replaces all instances of second specified string in the first specified string with a third specified string. The syntax is as follows:

```
REPLACE('<string_expression1>', '<string_expression2>', '<string_expression3>')
```

The string_expression1 parameter is the expression in which to search. The string_expression2 parameter is the expression to search for in string_expression1. The string_expression3 parameter is the expression with which to replace all instances of string_expression2.

REPLICATE

The REPLICATE function repeats a character_expression a specified number of times. The syntax is as follows:

```
REPLICATE(<character_expression>, <integer>)
```

REVERSE

The `REVERSE` function returns the reverse of the specified `character_expression`. The syntax is as follows:

```
REVERSE(<character_expression>)
```

RIGHT

The `RIGHT` function returns the rightmost part of the specified `character_expression`, starting a specified number of characters (given by `integer`) from the right. The syntax is as follows:

```
RIGHT(<character_expression>, <integer>)
```

The `integer` parameter must be a positive whole number.

RTRIM

The `RTRIM` function removes all the trailing blanks from a specified `character_expression`. The syntax is as follows:

```
RTRIM(<character_expression>)
```

SOUNDEX

The `SOUNDEX` function returns a four-character (SOUNDEX) code, which can be used to evaluate the similarity of two strings. The syntax is as follows:

```
SOUNDEX(<character_expression>)
```

SPACE

The `SPACE` function returns a string of repeated spaces, the length of which is indicated by `integer`. The syntax is as follows:

```
SPACE(<integer>)
```

STR

The `STR` function converts numeric data into character data. The syntax is as follows:

```
STR(<numeric_expression>[, <length>[, <decimal>]])
```

The `numeric_expression` parameter is a numeric expression with a decimal point. The `length` parameter is the total length including decimal point, digits, and spaces. The `decimal` parameter is the number of places to the right of the decimal point.

STUFF

The `STUFF` function deletes a specified length of characters and inserts another set of characters in their place. The syntax is as follows:

```
STUFF(<expression>, <start>, <length>, <characters>)
```

The `expression` parameter is the string of characters in which some will be deleted and new ones added. The `start` parameter specifies where to begin deletion and insertion of characters. The `length` parameter specifies the number of characters to delete. The `characters` parameter specifies the new set of characters to be inserted into the `expression`.

SUBSTRING

The `SUBSTRING` function returns part of an expression. The syntax is as follows:

```
SUBSTRING(<expression>, <start>, <length>)
```

The `expression` parameter specifies the data from which the substring will be taken, and can be a character string, binary string, text, or an expression that includes a table. The `start` parameter is an integer that specifies where to begin the substring. The `length` parameter specifies how long the substring is.

UNICODE

The `UNICODE` function returns the Unicode number that represents the first character in `character_expression`. The syntax is as follows:

```
UNICODE('<character_expression>')
```

UPPER

The `UPPER` function converts all the lowercase characters in `character_expression` into uppercase characters. The syntax is as follows:

```
UPPER(<character_expression>)
```

SYSTEM FUNCTIONS

The system functions — the more longstanding way of referring to what Microsoft is now referring to simply as "other" — can be used to return information about values, objects, and settings with SQL Server. The functions are as follows:

- `APP_NAME`
- `CASE`
- `CAST` and `CONVERT`

- `COALESCE`
- `COLLATIONPROPERTY`
- `CURRENT_TIMESTAMP`
- `CURRENT_USER`
- `DATALENGTH`
- `FORMATMESSAGE`
- `GETANSINULL`
- `HOST_ID`
- `HOST_NAME`
- `IDENT_CURRENT`
- `IDENT_INCR`
- `IDENT_SEED`
- `IDENTITY`
- `ISDATE`
- `ISNULL`
- `ISNUMERIC`
- `NEWID`
- `NULLIF`
- `PARSENAME`
- `PERMISSIONS`
- `ROWCOUNT_BIG`
- `SCOPE_IDENTITY`
- `SERVERPROPERTY`
- `SESSION_USER`
- `SESSIONPROPERTY`
- `STATS_DATE`
- `SYSTEM_USER`

APP_NAME

The `APP_NAME` function returns the application name for the current session if one has been set by the application as an `nvarchar` type. It has the following syntax:

```
APP_NAME()
```

CASE

The CASE function evaluates a list of conditions and returns one of multiple possible results. It also has two formats:

- The simple CASE function compares an expression to a set of simple expressions to determine the result.
- The searched CASE function evaluates a set of Boolean expressions to determine the result.

NOTE *Both formats support an optional* ELSE *argument.*

Simple CASE function:

```
CASE <input expression>
   WHEN <when expression> THEN <result expression>
   ELSE <else result expression>
END
```

Searched CASE function:

```
CASE
    WHEN <Boolean expression> THEN <result expression>
    ELSE <else result expression>
END
```

COALESCE

The COALESCE function is passed an undefined number of arguments and it tests for the first non-null expression among them. The syntax is as follows:

```
COALESCE(<expression> [,...n])
```

If all arguments are NULL then COALESCE returns NULL.

COLLATIONPROPERTY

The COLLATIONPROPERTY function returns the property of a given collation. The syntax is as follows:

```
COLLATIONPROPERTY(<collation_name>, <property>)
```

The collation_name parameter is the name of the collation you want to use, and property is the property of the collation you want to determine. This can be one of three values:

- **CodePage:** The non-Unicode code page of the collation.

- **`LCID`:** The Windows LCID of the collation. Returns `NULL` for SQL collations.
- **`ComparisonStyle`:** The Windows comparison style of the collation. Returns `NULL` for binary or SQL collations.

CURRENT_USER

The `CURRENT_USER` function simply returns the current user as a `sysname` type. It is equivalent to `USER_NAME()`. The syntax is as follows:

```
CURRENT_USER
```

DATALENGTH

The `DATALENGTH` function returns the number of bytes used to represent `expression` as an integer. It is especially useful with `varchar`, `varbinary`, `text`, `image`, `nvarchar`, and `ntext` data types because these data types can store variable-length data. The syntax is as follows:

```
DATALENGTH(<expression>)
```

@@ERROR

Returns the error code for the last T-SQL statement that ran on the current connection. If there is no error, the value will be zero.

If you're going to be writing stored procedures or triggers, this is a bread-and-butter kind of system function — you pretty much can't live without it.

NOTE *The thing to remember with* `@@ERROR` *is that its lifespan is just one statement. This means that, if you want to use it to check for an error after a given statement, you either need to make your test the very next statement, or you need to move it into a holding variable. In general, I recommend using* `ERROR_NUMBER()` *in a* `TRY...CATCH` *block unless you need to support pre-SQL Server 2005 code.*

A listing of all the system errors can be viewed by using the `sys.messages` system table in the `master` database.

To create your own custom errors, use `sp_addmessage`.

FORMATMESSAGE

The `FORMATMESSAGE` function uses existing messages in `sysmessages` to construct a message. The syntax is as follows:

```
FORMATMESSAGE(<msg_number>, <param value>[,...n])
```

Where `msg_number` is the ID of the message in `sysmessages`.

> ***NOTE*** `FORMATMESSAGE` *looks up the message in the current language of the user. If there is no localized version of the message, the U.S. English version is used.*

GETANSINULL

The `GETANSINULL` function returns the default nullability for a database as an integer. The syntax is as follows:

```
GETANSINULL(['<database>'])
```

The `database` parameter is the name of the database for which to return nullability information.

When the nullability of the given database allows `NULL` values and the column or data type nullability is not explicitly defined, `GETANSINULL` returns `1`. This is the `ANSI NULL` default.

HOST_ID

The `HOST_ID` function returns the ID of the workstation. The syntax is as follows:

```
HOST_ID()
```

HOST_NAME

The `HOST_NAME` function returns the name of the workstation. The syntax is as follows:

```
HOST_NAME()
```

IDENT_CURRENT

The `IDENT_CURRENT` function returns the last identity value created for a table, within any session or scope of that table. This is exactly like `@@IDENTITY` and `SCOPE_IDENTITY`; however, this has no limit to the scope of its search to return the value.

The syntax is as follows:

```
IDENT_CURRENT('<table_name>')
```

The `table_name` is the table for which you want to find the current identity.

IDENT_INCR

The `IDENT_INCR` function returns the increment value specified during the creation of an identity column in a table or view that has an identity column. The syntax is as follows:

```
IDENT_INCR('<table_or_view>')
```

The `table_or_view` parameter is an expression specifying the table or view to check for a valid identity increment value.

IDENT_SEED

The `IDENT_SEED` function returns the seed value specified during the creation of an identity column in a table or a view that has an identity column. The syntax is as follows:

```
IDENT_SEED('<table_or_view>')
```

The `table_or_view` parameter is an expression specifying the table or view to check for a valid identity increment value.

@@IDENTITY

Returns the last identity value created by the current connection.

If you're using identity columns and then referencing them as a foreign key in another table, you'll find yourself using this one all the time. You can create the parent record (usually the one with the identity you need to retrieve), and then select `@@IDENTITY` to know what value you need to relate child records to.

If you perform inserts into multiple tables with identity values, remember that the value in `@@IDENTITY` will only be for the *last* identity value inserted — anything before that will have been lost, unless you move the value into a holding variable after each insert. Also, if the last column you inserted into didn't have an identity column, `@@IDENTITY` will be set to `NULL`.

IDENTITY

The `IDENTITY` function is used to insert an identity column into a new table. It is used only with a `SELECT` statement with an `INTO` table clause. The syntax is as follows:

```
IDENTITY(<data type>[, <seed>, <increment>]) AS <column name>
```

Where:

- `data type` is the data type of the identity column.
- `seed` is the value to be assigned to the first row in the table. Each subsequent row is assigned the next identity value, which is equal to the last `IDENTITY` value plus the `increment` value. If neither `seed` nor `increment` is specified, both default to `1`.
- `increment` is the increment to add to the `seed` value for successive rows in the table.
- `column name` is the name of the column that is to be inserted into the new table.

ISNULL

The `ISNULL` function checks an expression for a `NULL` value and replaces it with a specified replacement value. The syntax is as follows:

```
ISNULL(<check expression>, <replacement value>)
```

ISNUMERIC

The ISNUMERIC function determines whether an expression is a valid numeric type. The syntax is as follows:

```
ISNUMERIC(<expression>)
```

NEWID

The NEWID function creates a unique value of type uniqueidentifier. The syntax is as follows:

```
NEWID()
```

NULLIF

The NULLIF function compares two expressions and returns a NULL value. The syntax is as follows:

```
NULLIF(<expression1>, <expression2>)
```

PARSENAME

The PARSENAME function returns the specified part of an object name. The syntax is as follows:

```
PARSENAME('<object_name>', <object_piece>)
```

The object_name parameter specifies the object name from the part that is to be retrieved. The object_piece parameter specifies the part of the object to return. The object_piece parameter takes one of these possible values:

- 1: Object name
- 2: Owner name
- 3: Database name
- 4: Server name

PERMISSIONS

The PERMISSIONS function returns a value containing a bitmap, which indicates the statement, object, or column permissions for the current user. The syntax is as follows:

```
PERMISSIONS([<object_id> [, '<column>']])
```

The object_id parameter specifies the ID of an object. The optional column parameter specifies the name of the column for which permission information is being returned.

@@ROWCOUNT

Returns the number of rows affected by the last statement.

One of the most used globals, my most common use for this one is to check for non–runtime errors — that is, items that are logical errors in your program but that SQL Server isn't going to see any problem with. An example is a situation in which you are performing an update based on a condition, but you find that it affects zero rows. Odds are that, if your client submitted a modification for a particular row, it was expecting that row to match the criteria given — zero rows affected is indicative of something being wrong.

However, if you test this system function on any statement that does not return rows, you will also return a value of `0`.

ROWCOUNT_BIG

The `ROWCOUNT_BIG` function is very similar to `@@ROWCOUNT` in that it returns the number of rows from the last statement. However, the value returned is of a data type of `bigint`. The syntax is as follows:

```
ROWCOUNT_BIG()
```

SCOPE_IDENTITY

The `SCOPE_IDENTITY` function returns the last value inserted into an identity column in the same scope (that is, within the same sproc, trigger, function, or batch). This is similar to `IDENT_CURRENT`, discussed earlier, although that was not limited to identity insertions made in the same scope.

This function returns a `sql_variant` data type, and the syntax is as follows:

```
SCOPE_IDENTITY()
```

SERVERPROPERTY

The `SERVERPROPERTY` function returns information about the server you are running on. The syntax is as follows:

```
SERVERPROPERTY('<propertyname>')
```

The possible values for `propertyname` are found in Table B-6.

TABLE B-6: propertyname Values

PROPERTY NAME	VALUES RETURNED
`Collation`	The name of the default collation for the server.
`Edition`	The edition of the SQL Server instance installed on the server. Returns one of the following `nvarchar` results:`'Desktop Engine''Developer Edition''Enterprise Edition''Enterprise Evaluation Edition''Personal Edition''Standard Edition'`.

PROPERTY NAME	VALUES RETURNED
`Engine Edition`	The engine edition of the SQL Server instance installed on the server: `1` — Personal or Desktop Engine, `2` — Standard, or `3` — Enterprise (returned for Enterprise, Enterprise Evaluation, and Developer).
`InstanceName`	The name of the instance to which the user is connected..
`IsClustered`	Will determine whether the server instance is configured in a failover cluster: `1` — Clustered, `0` — Not clustered, and `NULL` — Invalid input or error.
`IsFullTextInstalled`	To determine whether the full-text component is installed with the current instance of SQL Server: `1` — Full-text is installed, `0` — Full-text is not installed, and `NULL` — Invalid input or error.
`IsIntegratedSecurityOnly`	To determine whether the server is in integrated security mode: `1` — Integrated security, `0` — Not integrated security, and `NULL` — Invalid input or error.
`IsSingleUser`	To determine whether the server is a single-user installation: `1` — Single user, `0` — Not single user, and `NULL` — Invalid input or error.
`IsSyncWithBackup`	To determine whether the database is either a published database or a distribution database, and can be restored without disrupting the current transactional replication: `1` — `True` and `0` — `False`.
`LicenseType`	What type of license is installed for this instance of SQL Server: `PER_SEAT` — Per-seat mode, `PER_PROCESSOR` — Per-processor mode, `DISABLED` — Licensing is disabled.
`MachineName`	Returns the Windows NT computer name on which the server instance is running. For a clustered instance (an instance of SQL Server running on a virtual server on Microsoft Cluster Server), it returns the name of the virtual server.
`NumLicenses`	Number of client licenses registered for this instance of SQL Server, if in per-seat mode. Number of processors licensed for this instance of SQL Server, if in per-processor mode.
`ProcessID`	Process ID of the SQL Server service. (The `ProcessID` is useful in identifying which `sqlservr.exe` belongs to this instance.)
`ProductVersion`	Very much like Visual Basic projects, in that the version details of the instance of SQL Server are returned, in the form of `'`*`major.minor.build`*`'`.
`ProductLevel`	Returns the value of the version of the SQL Server instance currently running. Returns:`'RTM'` — Shipping version`'SPn'` — Service pack version`'Bn'` — Beta version
`ServerName`	Both the Windows NT server and instance information associated with a specified instance of SQL Server.

NOTE *The* `SERVERPROPERTY` *function is very useful for multi-sited corporations where developers need to find out information from a server.*

SESSION_USER

The `SESSION_USER` function allows a system-supplied value for the current session's username to be inserted into a table if no default value has been specified. The syntax is as follows:

```
SESSION_USER
```

SESSIONPROPERTY

The `SESSIONPROPERTY` function is used to return the `SET` options for a session. The syntax is as follows:

```
SESSIONPROPERTY (<option>)
```

This function is useful when there are stored procedures that are altering session properties in specific scenarios. This function should rarely be used, as you should not alter too many of the `SET` options during runtime.

STATS_DATE

The `STATS_DATE` function returns the date that the statistics for the specified index were last updated. The syntax is as follows:

```
STATS_DATE(<table id>, <index id>)
```

SYSTEM_USER

The `SYSTEM_USER` function allows a system-supplied value for the current system username to be inserted into a table if no default value has been specified. The syntax is as follows:

```
SYSTEM_USER
```

@@TRANCOUNT

Returns the number of active transactions — essentially the transaction nesting level — for the current connection.

This is a very big one when you are doing transactioning. I'm not normally a big fan of nested transactions, but there are times when they are difficult to avoid. As such, it can be important to know just where you are in the transaction-nesting side of things (for example, you may have logic that starts a transaction only if you're not already in one).

If you're not in a transaction, @@TRANCOUNT is 0. From there, take a look at a brief example:

```
SELECT @@TRANCOUNT As TransactionNestLevel      --This will be zero
                                                --at this point

BEGIN TRAN
SELECT @@TRANCOUNT As TransactionNestLevel      --This will be one
                                                --at this point

  BEGIN TRAN
    SELECT @@TRANCOUNT As TransactionNestLevel  --This will be two
                                                --at this point
  COMMIT TRAN
SELECT @@TRANCOUNT As TransactionNestLevel      --This will be back to one
                                                --at this point
ROLLBACK TRAN
SELECT @@TRANCOUNT As TransactionNestLevel      --This will be back to zero
                                                --at this point
```

Note that, in this example, the @@TRANCOUNT at the end would also have reached zero if you had a COMMIT as the last statement.

TEXT AND IMAGE FUNCTIONS

The text and image functions perform operations on text or image data. They are:

- TEXTPTR
- TEXTVALID

TEXTPTR

The TEXTPTR function checks the value of the text pointer that corresponds to a text, ntext, or image column and returns a varbinary value. The text pointer should be checked to ensure that it points to the first text page before running READTEXT, WRITETEXT, and UPDATE statements. The syntax is as follows:

```
TEXTPTR(<column>)
```

TEXTVALID

The TEXTVALID function checks whether a specified text pointer is valid. The syntax is as follows:

```
TEXTVALID('<table.column>', <text_pointer>)
```

The table.column parameter specifies the name of the table and column to be used. The text_pointer parameter specifies the text pointer to be checked.

This function will return 0 if the pointer is invalid and 1 if the pointer is valid.