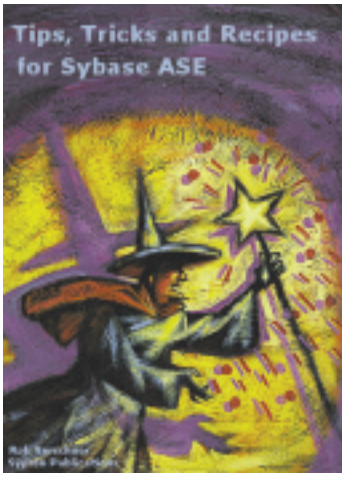
Tips, Tricks, and Recipes for Sybase ASE

By Rob Verschoor



Excerpts from a new book on getting the most out of ASE

Rob Verschoor is a freelance consultant in the Netherlands, specializing in Sybase ASE, and the author of several books. He can be reached at rob@sypron.nl.

Worldwide Uniqueness: GUIDs/UUIDs

Starting with ASE 12.5.0.3, it is possible to generate so-called GUIDs with the newid() built-in function. A GUID (Globally Unique Identifier) is a 16-byte (128-bit) value, generated according to the GUID/UUID specification. This specification was drafted by the Internet Engineering Task Force (IETF). It is widely accepted as a standard, and can be found at http://ftp.ics.uci.edu/pub/ietf/webdav/uuid-guid/. GUIDs are also known as UUIDs (Universally Unique Identifiers). In this book, GUID is used.

The uniqueness of a GUID is not limited to a particular database or server: multiple ASE servers running on the same host will produce different GUID values. By design, GUIDs generated by software products from different vendors will never be identical. Please refer to the GUID specification for details about the underlying principles.

Note that there is actually a limit to the GUID's uniqueness. According the GUID specification, the current design is valid until the year 3400; this seems sufficient for most practical requirements.

The 'newid()' Function

The **newid()** function generates a 16-byte GUID value, formatted as a hexadecimal varchar(36) expression:

```
1> select newid()
2> go
------------------------------------
fc1706e141e44c6a966842133ef94e73
```

Note that the above string contains only 32 characters. When specifying "1" (or any other non-zero integer value) as an argument to newid(), the GUID will be formatted with four additional dash ("-") characters:

```
1> select newid(1)
2> go
------------------------------------
10d1eaff-f116-45c7-95a9-44b5fc6dc351
```

This formatted version of a GUID may be easier to handle for a human. Note that other vendors use this formatting convention by default (for example, MS SQL Server). The positions of the dashes are prescribed by the GUID specification.

Why GUID Uniqueness is Guaranteed Worldwide

GUIDs can be useful because their values are guaranteed to be unique in all other systems as well. This means that GUIDs can safely be used as worldwide-unique keys when exchanging data between different systems or databases (obviously, all systems involved must support GUIDs in the first place). In a sense, a GUID can be thought of as a primary key that is valid across all existing applications and databases over a long period of time.

Note that the universal uniqueness of a GUID applies only when compared with other GUIDs. Nothing would stop you from generating an arbitrary, non-GUID compliant, 16-byte value, which might well be a duplicate of a GUID generated by some other system on this planet.

Generating Keys for Multiple-Row Inserts

Like other built-in functions, **newid(1)** produces a constant value for all rows in a result set:

```
1> select newid(1) GUID, EmpName from
Employees
2> go
------------------------------------
 GUID
EmpName
```

```
5e009fe6-fcff-44ac-94b2-0ece43f73d18   Gomez
5e009fe6-fcff-44ac-94b2-0ece43f73d18   Smith
5e009fe6-fcff-44ac-94b2-0ece43f73d18   Anderson
5e009fe6-fcff-44ac-94b2-0ece43f73d18   Stewart
5e009fe6-fcff-44ac-94b2-0ece43f73d18   McGuire
5e009fe6-fcff-44ac-94b2-0ece43f73d18   NULL
[...]
```

Unlike most other built-in functions, **newid( )** can also produce unique values for each row in the result set. The trick is to specify an argument to **newid( )** which always evaluates to > 0, but is not a constant. This is achieved by including a column reference in the expression.

The following is equivalent to **newid( )** but forces the function to be evaluated for each row:

```
1> select newid(ascii(EmpName)) GUID, EmpName
2> from Employees
3> go
------------------------------------
GUID                                 EmpName
------------------------------------
1384c4da-b584-4d7d-a8f8-abf49ec7e84f Gomez
1f00cfb7-6de3-4934-9417-50cb5470463e Smith
3330bbfb-323e-4c49-bc82-c5f6c37317c5 Anderson
------------------------------------
981eddb7-79bc-4943-80b1-96cdc22375d8 Stewart
f73f8542-7936-4c00-9caf-1c5b2a8784bd McGuire
7bd25b1663414e5682c1ab6a5923e1c9     NULL
[...]
```

In this example, the ASCII value of the first character of each employee name is specified as an argument to newid(). Since an ASCII value is always greater than 0, the resulting GUID will be formatted. More importantly, **newid( )** now produces a unique value for each row in the result set.

If you prefer an unformatted GUID, the argument to **newid( )** should be a non-constant expression evaluating to 0 or NULL. Example:

```
select newid(ascii(EmpName)/1000) GUID, EmpName [...]
```

For a formatted GUID, the **newid( )** argument must be not be 0 or NULL. The ASCII value of the first character of a column is sufficient for this, unless the column contains NULL (as illustrated by the last row in the example result set above). For a (var)char column allowing NULLs, use **isnull( )** to avoid a NULL result:

```
select newid(ascii(isnull(EmpName,'') )) GUID, EmpName [...]
```

Columns containing numbers can be used as an argument to **newid( )**, via **abs( )**. The following examples produce formatted and unformatted GUIDs, respectively:

```
select newid(abs(isnull(NumCol,1))+1) GUID, EmpName [...]----------------
select newid(abs(NumCol)-abs(NumCol)) GUID, EmpName [...]
```

Implicit Key Generation (Column Default)

A GUID key can also be generated implicitly by placing **newid( )** in a column default:

```
create table Customers
  (CustID   varchar(36) default newid(1) unique,
   CustName varchar(30))
-----------------------------------
-- Key value is generated implicitly by the column default
insert Customers (CustName) values ("Jones")
```

With this column default, only one row can be inserted at a time without an explicitly specified key value. As a column reference is not allowed in a default, multiple rows inserted with an **insert...select** statement will all have the same key (though the unique index will not allow this).

Note that key values can still be specified explicitly in **insert** statements, overriding the column default.

Notes and Remarks

- When using a GUID as a key column, you can choose between the 32-character unformatted value and the 36-character formatted value. While 32 characters require about 11% less space, 36 characters is perhaps a better choice when these values are ever exchanged with other systems, since other applications may use the formatted version by default.
- Two ASE servers on the same host should generate a disjoint set of GUID values. The GUID specification suggests using the host system's network card hardware address (MAC address) for this purpose, but ASE uses **@@nodeid** instead. **@@nodeid** is a unique, 12-character identifier used by ASTC/DTM for managing distributed transactions. It is unique for each ASE server.
- The built-in function **rand( )** produces predictable numbers when using a seed. In contrast, the results from **newid( )** will always be different. This makes **newid( )** probably the only feature in ASE whose behavior cannot be reproduced exactly. When you think about it, this is actually quite special…

Terminating Your Own Connection with syb_quit()

Sometimes there may be a requirement to terminate your own user process from T-SQL code. The **kill** command cannot be used for this purpose, because it will not operate on its own user process.

To terminate your own process, use the built-in function **syb_quit()**. When **syb_quit()** is executed, ASE will disconnect the connection immediately, and any subsequent statements in the stored procedure, trigger, or batch will not be executed anymore. An open transaction is rolled back.

As a built-in function, **syb_quit()** is executed by running a **select** statement on it:

```
% isql -Usa -Pbigsecret -SSYB125
1> select syb_quit()       -- life stops here ...
2> print "After syb_quit()"  -- this is never executed
3> go
CT-LIBRARY error:
   ct_results(): network packet layer: internal net library
   error: Net-Library operation terminated due to disconnect
```

In the above code, the **print** statement on line 2 will never be executed because the connection to ASE will already have been terminated as a result of **syb_quit()** on line 1. Note the error messages printed by the client, in this example isql (other clients may print different messages). The client does not know the effect of the statements sent to ASE, but it soon discovers that the server has terminated the client's connection. Since this is an unexpected event for **isql**, it will print an error message (something similar happens when executing **shutdown** from **isql**).

syb_quit() was introduced in ASE 12.0. No special permissions are required to execute it, so it is available to all users. It is unclear to me why this useful built-in function has remained undocumented. In my experience, **syb_quit()** is harmless and does not cause any problems. In fact, it is used in Sybase's own **installhasvss** (NT: **insthasv**) script, which installs the system stored procedures needed for the High Availability functionality.

Why Quit Your Own Session?

Abruptly terminating your own session may seem an odd thing to do. Nevertheless, this can be useful functionality. Here are some examples of how **syb_quit()** can be used:

- **Conditional script exit**

Suppose you're writing a T-SQL script to create some stored procedures. Before actually creating those procedures, you may want to verify that certain conditions are met, and abort the script if this is not the case. For example, it may be required that the script is only executed by a login with **sa_role** privilege, or that a database with a specific name must exist. Alternatively, you may want to ensure the script is executed against a certain ASE version: for example, when the code uses the "execute-immediate" feature, ASE 12.0 or later is required.

Without aborting the script, error messages would be printed when ASE runs into syntax errors or permission problems. Instead, aborting the script with a clear error message is a much more user-friendly solution. This example shows how to implement such functionality with **syb_quit()**:

```
set flushmessage on
go
-----------------------------------
-- Check: we must be on ASE 12.0 or later
if isnull(object_id("dbo.sysqueryplans"),99) >= 99
begin
  print "This script requires ASE 12.0+; quitting..."
-----------------------------------
  select syb_quit()
  -- this line will never be reached
end
go
-----------------------------------
-- Check: we must have 'sa_role' activated
if proc_role("sa_role") = 0
begin
  print "This script requires 'sa_role'; quitting'"
  select syb_quit()
  -- this line will never be reached
end
go
-----------------------------------
-- Check: database 'PROD_DB' must exist
if db_id("PROD_DB") = NULL
begin
  print "Database 'PROD_DB must exist; quitting'"
  select syb_quit()
  -- this line will never be reached
-----------------------------------
end
go
use PROD_DB  -- now we know this database exists!
go
-----------------------------------
-- All checks OK; now we can go ahead with our script...
create procedure my_proc
as
...T-SQL-commands...
go
```

Note that this code cannot be used in pre-12.0 versions because **syb_quit()** is a known function only in 12.0+. Hence, it will cause a syntax error in pre-12.0. If this code should also work for pre-12.0, use the method with **set background**, described below, instead.

Also note two other interesting features in this code: set **flushmessage on** ensures any printed warning messages will not be lost while disconnecting; the test for ASE 12.0 is based on the existence of a version-specific system table.

◆ Conditionally blocking ASE access

Under certain conditions, you may want to deny a client process access to the ASE server. For example, there could be a rule that employees may only run a certain application during office hours. This could be implemented by building a check into the application to determine whether the ASE login is currently allowed access; if not, it simply calls **syb_quit()** to terminate its connection, thus effectively blocking access to ASE:

```
if datepart(dw, getdate()) in (1,7)  -- Sunday, Saturday
  or datepart(hh, getdate())
    not between 9 and 18        -- outside 9AM to 6PM
begin
  set flushmessage on
  print "Access is allowed during working hours only!"
  select syb_quit()
end
```

In a variation on this theme, a login trigger is used to implement this kind of functionality. A login trigger is basically a stored procedure that is executed as part of the login process when a client connects to ASE. By executing **syb_quit()** in the login trigger, the login attempt fails. See www.sypron.nl/logtrig.html for further examples of using login triggers.

◆ Emergency exit from nested SQL code

Yet another way of using **syb_quit()** is to use it as an emergency exit from SQL code. For example, a weird, unlikely kind of error is encountered in a deeply nested piece of SQL—let's say there appear to be two rows for a certain primary key (someone has dropped the unique index on this key?). Due to the severity of the error, there may be no point in continuing execution, and all action in this session should be aborted immediately. This rather drastic exit can be implemented by simply calling **syb_quit()**, without backing out from each of the nested code levels.

Please note that I do not recommend this method as a standard practice: in general, I believe that error checking and backout actions deserve attention, and are worth spending code on. This application of **syb_quit()** is probably best suited for tool-style SQL code used by DBAs rather than for production code. To be used with care!

set background: syb_quit() for ASE Pre-12.0

syb_quit() was introduced in ASE 12.0, so it cannot be used in pre-12.0. Functionality equivalent to **syb_quit()** can be implemented in all ASE versions with the undocumented and unsupported command **set background**.

I have never seen any "official" documentation or application of **set background** by Sybase, so the information in this book was determined empirically. Still, **set background** seems reliable when used for the purposes described here.

The most obvious effect of **set background on** suppression of all output to the client, and sending some (but not all) output to the ASE errorlog instead. **set background off** resumes the normal output stream to the client.

Although there are some practical applications of suppressing output to the client, we'll be using a different aspect of **set background** in this section.

Terminating Your Connection with set background

set background on and **set background off** are expected to occur in the same batch or stored procedure together. Interestingly, when only **set background on** occurs, and **set background off** is omitted, ASE will terminate the connection, much like **syb_quit()** does:

```
1> set background on
2> go
CT-LIBRARY error:
  ct_results(): network packet layer: internal net library
  error: Net-Library operation terminated due to disconnect
```

This functionality can be wrapped in a stored procedure **sp_quit** as follows:

```
1> use sybsystemprocs
2> go
1> create procedure sp_quit
2> as set background on
3> go
1> use master
2> go
1> sp_quit
2> go
CT-LIBRARY error:
  ct_results(): network packet layer: internal net library
  error: Net-Library operation terminated due to disconnect
```

This trick is identical to **syb_quit()** in the way the connection is terminated, but there are a few differences. For example, executing **syb_quit()** will immediately terminate the connection; when using **set background on**, execution seems to continue until the end of the batch before terminating the connection. Consider the following code:

```
1> set flushmessage on
2> print "Before 'set background on'"
3> set background on              -- same as 'exec sp_quit'
4> print "After 'set background on'"  -- goes to the errorlog
5> select "After 'print'"         -- this output disappears
6> go
Before 'set background on'
CT-LIBRARY error:
        ct_results(): network packet layer: internal net library
  error: Net-Library operation terminated due to disconnect
```

When executing the above code, the following happens:

1. The **print** statement on line 2 is executed normally;
2. On line 3, **set background on** is executed, suppressing all further output to the client;
3. The **print** statement on line 4 is executed; because **set background on** is active, the output goes into the ASE errorlog instead of to the client;
4. The **select** statement on line 5 is executed; because **set background on** is active, all result sets disappear completely;
5. Finally, the end of the batch is reached and there are no more statements to be executed. Because **set background off** was not found in this batch, the connection is terminated and processing stops.

Note that the statements in the remainder of the batch are still executed after **set background on**. For this reason, this trick cannot be used to replace **syb_quit()** for quickly exiting nested SQL code as described earlier. It can be used for the other purposes described for **syb_quit()** (i.e. Conditionally blocking ASE access; Conditional script exit) as long as no further processing is performed in the batch after **set background on**.

This "application" of **set background** has been possible since at least ASE 11.0. No special permissions are required to execute it, so it is available to all users.

You should be aware that **set background** can have peculiar side effects in certain situations. I can imagine this is one of the reasons why **set background** has officially remained undocumented. Whenever you decide to use **set background**, do so with care. ■