

## Re: SQL 7 vs. 2000 issue –trigger and nulls

**Source:**

<http://www.tech-archive.net/Archive/SQL-Server/microsoft.public.sqlserver.mseq/2004-12/0024.html>

---

**From:** Hugo Kornelis (*hugo\_at\_pe\_NO\_rFact.in\_SPAM\_fo*)

**Date:** 12/24/04

Date: Fri, 24 Dec 2004 10:15:26 +0100

On Wed, 22 Dec 2004 11:34:54 –0800, Bill Polewchak wrote:

Hi Bill,

I kept your message on hold for a few days, hoping that someone else would answer. My knowledge of SQL Server 7.0 is limited, so I'm not 100% sure if all the details below are entirely correct.

You'll find my comments inline.

>We're trying to dump our remaining v7 SQL server and update to 2000.  
>We're having trouble with a trigger updating some tables.  
>  
>At the beginning, it has this statement:  
>COMMIT TRANSACTION --This unlocks the Lot table so I can update UDAs,  
>etc...  
>BEGIN TRANSACTION  
>  
>This is what throws the trigger off. I created a "play" table with an  
>Update trigger, and when I added this statement at the beginning, my  
>trigger stopped working (this is all on the new server). I took these  
>two lines out of Lot's Update trigger (again on the new server), ran an  
>Update query, and the trigger populated all the fields correctly. Does  
>anybody know what it means about unlocking the Lot table in order to  
>update? Do we need this in SQL2000 or can we take it out, or do we need  
>to replace it by something?

The comment about unlocking the table to allow updates doesn't make sense at all. The only rows affected by the update statements in the rest of the trigger are the rows in INSERTED, the ones that fired the update trigger (assuming that LotID is the primary key for the Lots table). They will of course be locked by the current transaction, but that keeps OTHERS from accessing the rows; the trigger can happily change these rows as it runs in the same transaction as the firing update statement.

In fact, you get the reverse effect! I just did some testing and it showed that committing and beginning a transaction in a trigger will open the

possibility that OTHER transactions lock the data the trigger needs to modify. See the repro script below:

```
-- First, execute this
create table test (id int identity, a int)
go
-- Create a row of data to play with
insert test values(1)
go
create trigger testtrig on test after update
as
-- Prevent recursion
if TRIGGER_NESTLEVEL() > 1 return
select getdate()
-- short wait so we can start simultaneous transaction
waitfor delay '00:00:04'
commit transaction
begin transaction
select getdate()
update test
set a = a + 1
select getdate()
go
```

-- Now, open two windows in Query Analyzer.

-- Execute this in the first window:

```
update test set a = 5
select * from test
```

-- Execute this in the second window; make sure to start it

-- before the first window is finished. If 4 seconds is not enough,

-- increase the delay in the trigger.

```
update test set a = 10
select * from test
```

If you check the output, you'll see that the second transaction starts locking data and updating rows when the COMMIT inside the trigger is executed, causing blocking for the first transaction. If you rerun the test with the COMMIT TRANSACTION / BEGIN TRANSACTION commented out, you'll see that the locks are held until the end of the transaction.

The reading of data from other table and the insert into WorkOrderDetail might be blocked by locks from other connections, but committing \*this\* transaction won't help to release locks from \*other\* transactions!

I seriously hope that SQL Server 7.0 simply disregarded these two statements. Because if it really did execute them, you'd find yourself facing non-atomic statement execution. What if an unexpected error causes a rollback in the second part of the trigger? What if the power fails and SQL Server has to rollback the uncommitted transaction on recovery? If the COMMIT in the trigger would really be executed, you'd only rollback the

last half of the trigger, not the first half or the firing statement; your database would lose it's integrity.

I am also curious what you mean with "my trigger stopped working". Did you get error messages? Were the results other than expected? If so, how did you test?

The reason I ask is that I just ran a short test and my conclusion is that SQL Server 2000 will in fact happily commit the transaction that fired the trigger, start a new transaction and continue trigger execution. See this repro:

```
create table test (a int)
go
create trigger testtrig on test after insert
as
select a, 'before commit', @@trancount from test
commit transaction
select a, 'after commit', @@trancount from test
begin transaction
select a, 'before delete', @@trancount from test
delete test
select a, 'after delete', @@trancount from test
rollback transaction
select a, 'after rollback', @@trancount from test
go
insert test values(1)
go
drop table test
go
```

But even though this test shows that it apparently does work, I still urge you to take these lines out. I don't think that you want your update statements to lose their atomic behaviour and the argument about unlocking the table is nonsense.

(later) I think I now found why your trigger apparently "stops working" in SQL Server 2000 with the commit / begin transaction sequence. When the transaction is committed and a new transaction started, the contents of the inserted and deleted pseudo-table are no longer available:

```
create table test (a int)
go
create trigger testtrig on test after insert
as
select * from inserted
commit transaction
begin transaction
select * from inserted
go
insert test values(1)
go
```

```
drop table test
go
```

```
>CREATE TRIGGER trLot_U ON dbo.Lot
>FOR UPDATE
>AS
(snip)
>/*****Add Check for site id*****/
>Select @SalesOrder = substring(LotID, 1, CHARINDEX('-', LotID)-1),
>@SiteID = SiteID
>from inserted
>
>IF @SiteID = NULL or @SiteID = ''
>select @SiteID = SiteID
>from SalesOrder
>where SalesOrder = @SalesOrder
```

You should revisit this part. The other statements in the trigger will continue to work if multiple rows are updated in one statement; this statement won't. @SalesOrder and @SiteID will be based on the "last" row in inserted, but since the order of rows is undetermined, you'll in fact execute these statements for one of the updated row, chosen at random.

```
>UPDATE Lot
>SET Lot.EUs = Round(0.0077*Lot.OrderLength*@nRunsToSlit/3.0
> + 9.6300*@nRunsToSlit
> + 0.8100*Lot.NumberUp
> + 5.0000, 1)
>
>FROM inserted
>WHERE Lot.LotID=inserted.LotID
```

This will work, but @nRunsToSlit is calculated based on ALL rows in the update statement. I'm not sure if that's what you want – looking at the trigger logic, I'd expect that you need a separate @nRunsToSlit for each Lot.

```
>SELECT @Lot = (SELECT i.LotID FROM inserted i)
>
>IF @Lot like 'E%%'
>BEGIN
> INSERT INTO WorkOrderDetailTemp(LotID) SELECT @Lot
>END
```

You should revisit this part. The other statements in the trigger will continue to work if multiple rows are updated in one statement; this statement won't. @Lot will be based on the "last" row in inserted, but since the order of rows is undetermined, you'll in fact execute these statements for one of the updated row, chosen at random.

Another suggestion for the complete trigger. You now have several update statements, all affecting the same set of rows. Your performance will probably go up if you rewrite your code to use just one update statement, setting the new values for all columns at once. This will be a more complicated statement, so if your performance is no problem, you might wish to leave it for now – but do keep it in mind for future improvement if your business grows!

```
>select..
>..
>case
> when @nDBRollNumber>=@MaximumRollNumber then 'yes'
> else 'no'
>end
>
>This would replace the existing
>
>select..
>..
>isnull(substring('yes', charindex(convert(varchar,@MaximumRollNumber)
>,convert(varchar,@nDBRollNumber)),255),'no')
```

Note that these constructions are not the same!!

The charindex function will be 0 if the two variables are unequal, regardless of which is the greater one (both 4/6 and 6/4 will result in the charindex being 0). Equal values and some special value pairs will result in a charindex equal to 1 (e.q. 5/5, but also 5/53); other special value pairs will yield a charindex greater than 1 (e.g. 5/35 will result in the charindex being 2 and the substring equal to 'es').

Depending on the range of allowed values for MaximumRollNumber and nDBRollNumber, this is either a very contrived and anti–documenting way to test for equality, or the implementation of a very strange business requirement.

Best, Hugo

--

(Remove \_NO\_ and \_SPAM\_ to get my e-mail address)