SQL Transactions White Paper

A Guide to "Nested" Transactions.

by Paul Sweeney

This document produced 20-Sep-2012.

Latest Word, RTF & PDF versions can be found online at http://www.kolossi.co.uk/SqlWhitePaper

Introduction

This article is written about SQL Server 2005, but it is applicable to other versions and sql servers in general.

Despite the title, there are no such things as "nested" transactions in SQL Server. Let me say that again:

There are no such things as "Nested" Transactions.

TSQL gives us named transactions, but they are generally pointless. The only reason the named transaction is used is for code in an inner transaction to be able to force a commit of the "outermost" transaction. Aha, so there are such things as "nested" transactions right? Well very likely not the way you are thinking!

If anything in this article is wrong, please do not hesitate to contact me on sql.transactions@kolossi.co.uk. I have tried to verify everything I've said and The Grand Solution at the end of the document has held out well for me. If I find any errors, I'll update this document. I'm not perfect though, so I may have it hideously wrong, in which case please tell me!

Why Transactions

The widely-quoted canonical example of transactions is the transfer of money from one account to another. This consists of two actions:

- Debit money from source account
- Credit money to destination account

These 2 actions must either both fail or both succeed. They are said to be "atomic". This comes from the greek "atomos" meaning indivisible. To arrange that this is the case, we wrap the two actions in a transaction, and all is right with the world. As the insurance-selling Meerkat would say "Simples...schk"!

Nesting Transactions

OK, so that's all good, but say in my database design, I have it that in order to perform a debit or credit from an account, I actually have to perform 2 actions:

- Add the action to the "Account Actions" table
- Change the current balance in the "Account Balances" table

Let's leave aside the fact that this is a REALLY dumb database design for a minute, it's purpose is to serve as an example, not win design awards.

These are again, 2 actions which require to be "atomic". I'm assuming we put these actions, and the account transfer, in their own stored procedures. To add the "atomic" bit, in the stored procedure we just wrap them in a transaction which either commits or rolls back if there is an error, right?

That will work fine if all goes well, but when there is an error, things start to unravel. Let's see why.

SQL Server Transaction Handling

If we do not explicitly start a transaction, SQL Server usually makes each and every statement it's own transaction but then proceeds in the same way as described below.

Start Transaction

When the transaction is started, SQL Server increments an internal counter for your connection called "@@TRANCOUNT" so it knows how many transactions are open. Seriously, that's it.

Do Some SQL actions

Now we do some actions, usually more than one if we've explicitly opened a transaction. SQL Server keeps the necessary locks on the database so no-one else can edit data that would effect or be affected by our transaction. It keeps note of the changes we make, but they aren't committed to the database yet - or at least no-one else can see the changes. (This is a whole topic in itself, but feel free to look up transaction isolation levels and lock levels and hints in Google).

Committing a "Nested" Transaction

OK, so we've done some SQL actions in a "nested" transaction. So @@TRANCOUNT is more than one. As part of the account transfer, we've done the debit by updating the "Account Actions" and "Account Balances" tables. Now we are going to commit this piece of work. Are your ready for this explanation, it's complex, so listen carefully as we go through all the actions?:

- SQL Server decrements @@TRANCOUNT by one
- ... that's it

Seriously! If we are in a "nested" transaction, that's all there is to it. We've already said that the transfer of funds is an "atomic" operation. It's indivisible. This isn't like the nuclear atom where we keep on finding smaller and smaller bits (protons/neutrons, quarks etc). There isn't any smaller piece we care about, either the whole thing works or it doesn't. If... OK, you've got it!

The reason is, if we did somehow commit just this couple of actions somewhere, we still couldn't publish them in the database as we don't know if the credit is going to succeed. When we've finished the funds transfer, either everything is committed to the database, or all the changes we've made are chucked away when we "rollback" on an error.

Committing the outermost transaction

When the transfer, both debit and credit, has been completed we do a commit of the outer transaction which is the "real" commit. SQL Server decrements @@TRANCOUNT and finds it is now 0, so it updates the database with all the changes we've made since the outermost "BEGIN TRANSACTION", including everything done in our "nested" transactions, and releases all the locks relating to these actions. The account transfer has happened and it's all good.

Rolling Back the Whole Transaction

What if when we go to do the debit we find there aren't sufficient funds available in the source account? We need to abandon the whole piece of work. No problem, we just call "ROLLBACK TRANSACTION". This:

- rolls back all transactions, and throws away any changes we've made since the outermost BEGIN TRANSACTION
- sets @@TRANCOUNT to 0

Great.

Is the transaction count correct?

Sql Server keeps track of the Transaction count when it calls a stored procedure, and if it find an inconsistency because you forgot to commit one of your "nested" transactions, it helpfully throws an error.

Rolling back the "nested" transaction

The account transfer scenario is working great, and we used a "nested" transaction without problem, so we now understand "nested" transactions and can go on to more complicated coding. We do know that "nested" transactions don't really exist, it's just a case of incrementing @@TRANCOUNT by one, but if we think of them as "nested" transactions, everything seems to work ok.

But what if at some stage during our "nested" transaction we encounter an error and want to rollback just the steps in that "nested" transaction? Not everything, just these certain steps within our "nested" ttransaction.

As an example, we may have been asked to delete a user from a website system. It may be that this comprises two entities - a web user and an email contact. The database may have foreign keys that mean we can't delete a web user if there are any audit entries refering to them, and we can't delete an email contact if we have ever sent them an email, or there are any email audit entries referring to them. For whatever reason it's not possible or practicle to check this first (there may be a LOT of audit tables!).

So we go ahead and try to delete the user and if we get a foreign key constraint when trying to delete the email contact, we want to rollback the web user delete as well. We don't want to rollback everything - this might be part of an admin procedure to delete all users who haven't logged in for 60 days, and we have made this admin procedure a transaction so we know it has either succeeded or failed as an atomic operation.

So anyway, back to the code, we can just do "ROLLBACK TRANSACTION" which will roll back the inner transaction in the same way "COMMIT TRANSACTION" commits it right? Wrong!

Both "ROLLBACK TRANSACTION" and "ROLLBACK TRANSACTION {nested transaction name}" will rollback ALL transactions exactly the same as described in "Rolling back the whole transaction". It doesn't matter where you are in a "nested" transaction tree, a ROLLBACK will roll back EVERYTHING!

Huh!?... Yup!...This is where the whole "nested" transaction illusion we have created for ourselves unravels.

Where did it all go wrong?

We talked briefly above about the case where we commit the outermost transaction and how that is where the changes to the database really happen.

Well what if for some reason in our "nested" transaction we decided everything was good and the outer transaction should be committed (doesn't really apply with our account transfer example)? Provided the outermost transaction was stated using the syntax "BEGIN TRANSACTION {outer transaction name}" we could issue "COMMIT TRANSACTION {outer transaction name}" from within the "nested" transaction, which would commit everything. We'd need to be very clever with our logic though so that when we returned to the code that started the outermost transaction, it doesn't try to do a COMMIT TRANSACTION - as it's already committed, this woud cause an error.

This is the only possible use for named transactions, they are irrelevant when using the ROLLBACK command (see above), and it's not easy to see a use case which couldn't be designed better.

For me it's the ability to give a transaction name that really cements the idea that named "nested" transactions are possible. The Books OnLine BEGIN TRANSACTION page alludes to the problems with this with an obscure and confusing statement "Use transaction names only on the outermost pair of nested BEGIN...COMMIT or BEGIN...ROLLBACK statements", but this should be in large red flashing lights, with far more detailed explanation and a warning that:

There are no such things as "Nested" Transactions.

The solution to rolling back a "nested" transaction

Well of course the answer is that there is no solution since:

There are no such things as "Nested" Transactions.

But in practice, all we need to do is change our thinking a little. There is no need for nested transactions, it's a mental throw-back to nested procedure calls in "normal" code programming, but this is SQL.

In general, there's a block or work that needs doing that must all succeed or must all be abandoned together. Because we are writing nice modular stored procedures, ***we have no idea where the boundary of that block of work is***. There's only one transaction that matters and that's the WHOLE (up to now called the "outermost") transaction, started at the beginning of the piece of work.

So if there is already a transaction in existence we should carry on using that. If we get an error we consider fatal, we should signal that to the caller who can decide whether they consider it fatal and decide whether to rollback ***their*** transaction. I know, I know, there's lots of questions this raises!

Save Points

In our debit account transaction we had two actions which both needed to complete for the debit to work. If we succeed in the first action but fail in the second, we need to undo the first action. Absolutely, but the solution to this is not a "nested" transaction, it's a "Save Point" on an existing transaction.

To do this, instead of doing "BEGIN TRANSACTION {nested transaction name}", do "SAVE TRANSACTION {save point name}".

When the time comes to roll back just the debit, we do "ROLLBACK TRANSACTION {save point name}".

The "ROLLBACK TRANSACTION" command accepts a savepoint name as well as a transaction name. Tellingly though, the "COMMIT TRANSACTION" does not accept save point names, we just wait until the whole transaction is committed and our changes will take effect. It reinforces this idea of only having one effective transaction to cover all the work.

Starting a transaction

So for a save point to work, we must have a transaction. We know we don't want to have "nested" transactions since:

There are no such things as "Nested" Transactions.

What we need to do is at the start of each stored procedure, check if a transaction already exists by using @@TRANCOUNT. If there is a transaction already, do nothing and just create savepoints if necessary for us to rollback to, otherwise do all our actions which will be committed when the existing transaction is committed by the code that started it.

If there is no current transaction, we are top of the tree and need to start a transaction. It will also be our job to commit it at the end or roll it back if there is an error.

Error Handling

Error handling in nested stored procedure calls, is almost as tricky an issue as "nested" transactions. There's a great article on this on MSDN ("Error Handling in T-SQL: From Casual to Religious"), which should be read in full. There's just two problems with the article from my point of view:

Firstly, if you get a high-level error such as that the table you have referenced in a query doesn't exist, this causes @@ERROR to be set to NULL, which using the code in the article, would be allowed to continue since "IF (@ERROR = 0)" does not fire if @ERROR is NULL.

Secondly, if there is a problem with the stored proc call itself, e.g. you put "@Param1=Param1" (without the second '@') from SQL's point of view this is a compile time error picked up at runtime and the stored proc doesn't get called so @ERROR is not updated. In this case you need to refer to @@ERROR.

So a fix is required to the template code which should be used to call both a sql command and a stored procedure.

The code used should be as follows:

```
{a SQL command}
SELECT @ERROR=COALESCE(@@ERROR,1)
EXEC @ERROR=MyStoredProc @Param1=@Param1
SELECT @ERROR=CASE WHEN @@ERROR=0 THEN COALESCE(@ERROR,1)
                           ELSE COALESCE(@@ERROR,1) END
```

At the end of the article, we'll put all this together into a template stored procedure pattern.

Checking for existing rows

This isn't to do with transactions or error handling, but it's a nasty gotcha than devs who do an occasional bit of sql might fall into messing up. When checking for whether there's a matching existing record in SQL, it's tempting to write e.g.:

```
DECLARE @Exists bit

SELECT @Exists = CASE WHEN fooId IS NULL THEN 0 ELSE 1 END
FROM Foo
WHERE fooName=@newName

IF @Exists <> 1
BEGIN
    RAISERROR(100001,-1,-1)
    SELECT @ERROR=100001
    GOTO ERROR_HANDLER
END
```

Unfortunately there are two major gotchas in this code.

The first is that if there are no records matching fooName=@newName, then the columns of the select are not evaluated, so @Exists is not updated and remains NULL. When "@Exists <> 1" is evaluated, because @Exists is NULL, the expression evaluates to NULL, so the IF does not fire and the error code is not executed. The direct fix is to do "COALESCE(@Exists,0) <> 1". Having done this, there's no need for the CASE statement, so we can just set @Exists to 1 *if* we find a record, otherwise it will be 0.

But the second gotcha is that as described above, @Exists is not updated at all if no matching row is found. As such if there is a second test later in the Stored Proc reusing the @Exists variable, assuming the first @Exists test passed, if the second one failed @Exists would be left as true and so the error again wouldn't be detected. So we need to set @Exists to 0 before we start. Doing this means there's no need for the COALESCE.

We also need to do our usual error checking, so the complete pattern for checking for an existing row should always be:

```
DECLARE @Exists bit

...

SELECT @Exists=0
SELECT @Exists = 1 FROM Foo WHERE fooName=@newName

SELECT @ERROR=COALESCE(@@ERROR,1)
IF (@ERROR <> 0)
BEGIN
    GOTO ERROR_HANDLER
END

IF @Exists <> 1
BEGIN
    RAISERROR(100001,-1,-1)
    SELECT @ERROR=100001
    GOTO ERROR_HANDLER
END
```

External References

Some great articles on error handling and the use of nested stored procedures/transactions, the combination of which lead to this article are:

- TIP: Nested Stored Procedure Calls with SQL Server Transactions - http://www.developer.com/db/article.php/3768671
- Error Handling in T-SQL: From Casual to Religious - http://msdn.microsoft.com/en-us/library/aa175920(SQL.80).aspx
- SAVE TRANSACTION (Transact-SQL) - http://msdn.microsoft.com/en-us/library/ms188378(SQL.90).aspx
- ROLLBACK TRANSACTION (Transact-SQL) - http://msdn.microsoft.com/en-us/library/ms181299(SQL.90).aspx
- SAVE TRANSACTION (Transact-SQL) - http://msdn.microsoft.com/en-us/library/ms188378(SQL.90).aspx
- Not used directly in compiling this article, but an excellent resource for topics such as those discussed here are Erland Sommarskog's Texts on SQL - http://www.sommarskog.se

The Grand Solution

OK, here it is, my grand template of how to write a stored procedure including propagating errors through nested stored procedure calls, and NOT using "nested" transactions since

There are no such things as "Nested" Transactions.

In order to get the benefits of this approach, all the stored procedures in the system will need to be built this way to get the consistency, robustness, and just-plain-works features we are all after:

```
CREATE PROCEDURE [dbo].[MyStoredProc] AS
BEGIN
	DECLARE @ERROR int
	DECLARE @OwnTransaction bit
	SET @OwnTransaction = 0
	IF @@TRANCOUNT = 0
	BEGIN
		BEGIN TRANSACTION
		SET @OwnTransaction = 1
	END

-------------------------------------------------
-- Stored Procedure Template header code ended
-------------------------------------------------


	--
	-- actual sp code start
	--

--	-- to call a stored procedure:
--
--	EXEC @ERROR=MyOtherStoredProc @Param1=@Param1,@Param2=@Param2
--
--	SELECT @ERROR=CASE WHEN @@ERROR=0 THEN COALESCE(@ERROR,1) ELSE
--	                                       COALESCE(@@ERROR,1) END
--	IF (@ERROR <> 0)
```

```
--     BEGIN
--            GOTO ERROR_HANDLER
--     END
--
--
--     -- to call sql code:
--
--     INSERT INTO MyTable (x) VALUES (2)
--     SELECT @ERROR=COALESCE(@@ERROR,1)
--     IF (@ERROR <> 0)
--     BEGIN
--            GOTO ERROR_HANDLER
--     END
--
--
--     -- To exit cleanly:
--
--     GOTO FINISH
--
--
--     -- To raise an error:
--
--     RAISERROR(100002,-1,-1)
--     SELECT @ERROR=100002
--     GOTO ERROR_HANDLER
--
--
--     -- to run code which may be reverted:
--
--     SAVE TRANSACTION revert
--     --
--     -- revertable sql here
--     --
--     SELECT @ERROR=COALESCE(@@ERROR,1)
--     if (@ERROR <> 0)
--     begin
--            goto ERROR_HANDLER
--     end
--     -- if code needs reverting,
--     --ROLLBACK TRANSACTION revert
--
--
--     -- to use a cursor (which you should almost NEVER need to):
--
--     DECLARE @MyCursor CURSOR
--     SET @MyCursor = CURSOR FAST_FORWARD FOR
--         SELECT something FROM tblSomwhere
--             WHERE somethingElse = @aValue
--
--
--     -- to check for existence of a record:
--
--     DECLARE @Exists bit
--
--     ...
--
--     SELECT @Exists=0
--     SELECT @Exists = 1 FROM Foo WHERE fooName=@newName
--
--     SELECT @ERROR=COALESCE(@@ERROR,1)
--     IF (@ERROR <> 0)
--     BEGIN
--         GOTO ERROR_HANDLER
--     END
--
--     IF @Exists <> 1
--     BEGIN
--         RAISERROR(100001,-1,-1)
```

```
--          SELECT @ERROR=100001
--          GOTO ERROR_HANDLER
--      END



        --
        -- actual sp code end
        --


-------------------------------------------------
-- Stored Procedure Template footer code follows
-------------------------------------------------

FINISH:
        IF @OwnTransaction = 1
        BEGIN
                COMMIT TRANSACTION
        END
        RETURN 0

ERROR_HANDLER:
        IF @OwnTransaction = 1
        BEGIN
                ROLLBACK TRANSACTION
        END
        RETURN @ERROR
END

GO
```