Transaction and Error Handling Basics

One of the most important and most basic responsibilities of any modern RDBMS is transaction handling and guaranteeing transactional consistency. It's crucial, absolutely fundamental that when you get a green traffic light to cross in front of me that I get a red light first, for instance, or that no money be debited to my bank account without being credited to yours. This is a basic principle of data management: blocks of work or actions that logically belong together are carried out in full or reversed in full.

The idea is, you declare your intention to begin a unit of work consisting of one or more actions (a batch). If all the actions complete successfully, that batch is finalized and you can proceed, but if anything goes wrong, you want the system to react, either by trying to continue and finish the unit of work, or by undoing all previous actions so that your data is restored to its original state.

SQL Server has full transaction support built in, but invoking and using transactions properly seems to cause people more problems than it should, given its importance.

Error trapping and handling is, obviously, an essential aspect of transaction handling and this is an area that was relatively poorly served in SQL versions up to SQL Server 2005. Since then, though, it's been simple and easy to implement good transaction handling combined with solid error trapping.

Here's a script with a transaction encompassing three statements:

```
use Tempdb
go

if object_id('TestTable') is not null drop table TestTable;
if object_id('TestTable2') is not null drop table TestTable2;


begin transaction AllOrNothing

	create table TestTable(Field1 int not null);

	select 1/0 as GuaranteedError;

	create table TestTable2(Field1 int not null);

commit transaction AllOrNothing
```

The second produces an error, so the 'commit' never gets called, right? Wrong.

```
select *
from   Tempdb.sys.objects
where  name like     N'TestTable%';
```

...and you'll see that the first and third statement still get executed.

Whoops! What we see here is that the 'COMMIT' keyword doesn't do enough on its own to ensure transactional integrity. What you want is to detect if an error has occurred and then roll the transaction back.

So, checking the system function @@error should be the way forward, right?

```
use Tempdb
go

if object_id('TestTable') is not null drop table TestTable;
if object_id('TestTable2') is not null drop table TestTable2;


begin transaction AllOrNothing

      create table TestTable(Field1 int not null);

      select 1/0 as GuaranteedError;

      create table TestTable2(Field1 int not null);

if    @@error > 0
begin

      print  'Big fat error!'
      rollback transaction AllOrNothing
end

else
commit transaction AllOrNothing;
```

Nope. If you run the sys.objects query again you'll see our two TestTables. What's gone wrong? The problem is, @@error returns the status of the last executed statement only, so if a subsequent statement is executed successfully @@error will return a zero. What this means is, you need to check the status of @@error after each statement in your batch.

```
use Tempdb
go

declare      @v_ErrorStateint;
select @v_ErrorState =     0;

if object_id('TestTable') is not null drop table TestTable;
if object_id('TestTable2') is not null drop table TestTable2;


begin transaction AllOrNothing

      create table TestTable(Field1 int not null);

      select @v_ErrorState =     @v_ErrorState + @@error;   -- Check

      select 1/0 as GuaranteedError;

      select @v_ErrorState =     @v_ErrorState + @@error;   -- Check

      create table TestTable2(Field1 int not null);

      select @v_ErrorState =     @v_ErrorState + @@error;   -- Check

      select @v_ErrorState as Variable;

if    @v_ErrorState > 0
      rollback transaction AllOrNothing

else
commit transaction AllOrNothing;
```

This works; the divide by zero error occurs and the entire batch is reverted, so no tables are created. However, it's not exactly elegant. There's more error and transaction handling code in this short script than actual work being done. You can imagine that this becomes tiresome if you're dealing with complex or long transaction batches.

However, if you're working with SQL Server 2000 or earlier versions (say it's not true!) then this, or variations on it, is what you're stuck with.

The picture is a lot rosier for SQL Server 2005 and SQL Server 2008, however. SQL 2005 introduced the TRY...CATCH construct, a real HALLELUJAH! moment for those of us who'd been left struggling with the limitations of earlier versions.

What happens these days is, you enclose your batch in a BEGIN TRY...END TRY block, and that's all the error trapping you need to do. It's that simple. Error handling is then taken care of by an adjoining BEGIN CATCH...END CATCH block. If no error occurs, all the statements in the TRY block are executed and the CATCH block is skipped, otherwise the CATCH block assumes control immediately, as soon as the error occurs. You can use the CATCH block to report on the nature of your errors, rollback your transaction, or perhaps call some other statement or stored procedure (for instance) in order to rescue the situation.

How does that look in our example, then?

```
use Tempdb
go

if object_id('TestTable') is not null drop table TestTable;
if object_id('TestTable2') is not null drop table TestTable2;


begin  try

       begin transaction AllOrNothing

              create table TestTable(Field1 int not null);

              select 1/0 as GuaranteedError;

              create table TestTable2(Field1 int not null);

       commit transaction AllOrNothing

end    try
begin  catch

       rollback transaction AllOrNothing;

end    catch
```

Run this script, and you'll see very little happen. The DIV/0 error occurs halfway, the CATCH block takes over, the whole lot gets rolled back.

A more verbose example illustrates better what's happening behind the scenes:

```
use Tempdb
go

set nocount on

if object_id('TestTable') is not null drop table TestTable;
if object_id('TestTable2') is not null drop table TestTable2;


begin  try

       begin transaction AllOrNothing

              create table TestTable(Field1 int not null);

              select 1/0 as GuaranteedError;

              create table TestTable2(Field1 int not null);

       commit transaction AllOrNothing

end    try
begin  catch

       print  'Whoops! An error has occurred...';

       select error_message(), error_line();

       select name as [TempTablesCreated]
       from   Tempdb.sys.objects
       where  name like     N'TestTable%';

       print  'Rolling back your transaction...';

       rollback transaction AllOrNothing;

       print  'All changed, changed utterly.'
       print  '';

end    catch;

select name as [TempTablesCreated]
from   Tempdb.sys.objects
where  name like     N'TestTable%';
```

As you can see, the first table gets created, then the error statement is executed and control passes immediately to the CATCH block - the second table is never created. Also, it's useful to observe that once the CATCH block has completed executing (or the TRY block, if no error has occurred) execution continues with the first statement after END CATCH.

This can be useful in stored procedures, where you perhaps want to accomplish more than just one logical block of work, but be careful to stop execution by including a RETURN after the ROLLBACK in the CATCH block if you don't want the stored procedure code to continue executing.

It's also worth noting that all the foregoing is based on the default SQL settings for options like XACT_ABORT and IMPLICIT_TRANSACTIONS. A discussion of these settings and how they affect your transactions will have to wait for some other time, but this article should have made the basics at least a little clearer.

Paul Clancy
360Data