simple-talk

DATA, DEV, ADMIN, CHOP SUEY

LATEST SQL, .NET AND SYS ADMIN ARTICLES, OPINION AND NEWS

What Counts For a DBA: Imagination

Published Thursday, May 10, 2012 10:36 PM

"Imaginationâ€¦One little spark, of inspirationâ€¦ is at the heart, of all creation."

â€“ From the song "One Little Spark", by the Sherman Brothers

I have a confession to make. Despite my great enthusiasm for databases and programming, it occurs to me that every database system I've ever worked on has been, in terms of its inputs and outputs, downright *dull*. Most have been glorified e-spreadsheets, many replacing manual systems built on actual spreadsheets. I've created a lot of database-driven software whose main job was to "count stuff"; phone calls, web visitors, payments, donations, pieces of equipment and so on. Sometimes, instead of counting stuff, the database recorded values from other stuff, such as data from sensors or networking devices. *Yee hah!*

So how do we, as DBAs, maintain high standards and high spirits when we realize that so much of our work would fail to raise the pulse of even the most easily excitable soul? The answer lies in our *imagination*.

To understand what I mean by this, consider a role that, in terms of its output, offers an extreme counterpoint to that of the DBA: the *Disney Imagineer*. Their job is to design Disney's Theme Parks, of which I'm a huge fan. To me this has always seemed like a fascinating and exciting job. What must an Imagineer do, every day, to inspire the feats of creativity that are so clearly evident in those spectacular rides and shows? Here, if ever there was one, is a role where "dull moments" must be rare indeed, surely?

I wanted to find out, and so parted with a considerable sum of money for my wife and I to have lunch with one; I reasoned that if I found one small way to apply their secrets to my own career, it would be money well spent. Early in the conversation with our Imagineer (Cindy Cote), the job did indeed sound magical. However, as talk turned to management meetings, budget-wrangling and insane deadlines, I came to the strange realization that, in fact, her job was a lot more like mine than I would ever have guessed.

Much like databases, all those spectacular Disney rides bring with them a vast array of complex plumbing, lighting, safety features, and all manner of other "boring bits", kept well out of sight of the end user, but vital for creating the desired experience; and, of course, it is these "boring bits" that take up much of the Imagineer's time.

Naturally, there is still a vital part of their job that is spent testing out new ideas, putting themselves in the place of a park visitor, from a 9-year-old boy to a 90-year-old grandmother, and trying to imagine what experiences they'd like to have. It is these small, but vital, sparks of imagination and creativity that have the biggest impact. The real feat of a successful Imagineer is clearly to never to lose sight of this fact, in among all the rote tasks.

It is the same for a DBA. Not matter how seemingly dull is the task at hand, try to put yourself in the shoes of the end user, and imagine how your input will affect the experience he or she will have with the database you're building, and how that may affect the world beyond the bits stored in your database. Then, despite the inevitable rush to be "done", find time to go the extra mile and hone the design so that it delivers something as close to that imagined experience as you can get.

OK, our output still can't and won't reach the same spectacular heights as the *"Journey into The Imagination"* ride at EPCOT Theme Park in Orlando, where I first heard "One Little Spark". However, our imaginative sparks and efforts can, and will, make a difference to the user who now feels slightly more at home with a database application, or to the manager holding a report presented with enough clarity to drive an interesting decision or two.

They are small victories, but worth having, and appreciated, or at least that's how I imagine it.

by drsql

Handling Deadlocks in SQL Server

10 May 2012
by Jonathan Kehayias

> In this excerpt from his book *Troubleshooting SQL Server: A Guide for the Accidental DBA*, Jonathan Kehayias provides a guide to identifying the causes, reacting to, and ultimately preventing the dreaded deadlock.

A deadlock is defined in the dictionary as "*a standstill resulting from the action of equal and opposed forces*," and this turns out to be a reasonable description of a deadlock in SQL Server: two or more sessions inside of the database engine end up waiting for access to locked resources held by each other. In a deadlock situation, none of the sessions can continue to execute until one of those sessions releases its locks, so allowing the other session(s) access to the locked resource. Multiple processes persistently blocking each other, in an irresolvable state, will eventually result in a halt to processing inside the database engine.

A common misconception is that DBAs need to intervene to "kill" one of the processes involved in a deadlock. In fact, SQL Server is designed to detect and resolve deadlocks automatically, through the use the **Lock Monitor**, a background process that is initiated when the SQL Server instance starts, and that constantly monitors the system for deadlocked sessions. However, when deadlocks are reported, the DBA must investigate their cause immediately. Many of the same issues that cause severe blocking in the database, such as poor database design, lack of indexing, poorly designed queries, inappropriate isolation level and so on, are also common causes of deadlocking. This article will provide the tools, techniques and tweaks you need to diagnose and prevent deadlocks, and to ensure that they are handled gracefully if they ever do occur. Specifically, it will cover:

- how to capture deadlock graphs using a variety of techniques, including Trace Flags, the Profiler deadlock graph event, and service broker event notifications
- how to read deadlock graphs to locate the sessions, queries and resources that are involved
- common types of deadlock and how to prevent them
- using server- or client-side `TRY…CATCH` error handling for deadlocks, to avoid `UnhandledException` errors in the application.

The Lock Monitor

When the Lock Monitor performs a deadlock search and detects that one or more sessions are embraced in a deadlock, one of the sessions is selected as a deadlock victim and its current transaction is rolled back. When this occurs, all of the locks held by the victim's session are released, allowing any previously blocked other sessions to continue processing. Once the rollback completes, the victim's session is terminated, returning a 1205 error message to the originating client.

SQL Server selects the deadlock victim based on the following criteria:

1. **Deadlock priority** – the assigned `DEADLOCK_PRIORITY` of a given session determines the relative importance of it completing its transactions, if that session is involved in a deadlock. The session with the lowest priority will always be chosen as the deadlock victim. Deadlock priority is covered in more detail later in this article.
2. **Rollback cost** – if two or more sessions involved in a deadlock have the same deadlock priority, then SQL Server will choose as the deadlock victim the session that has lowest estimated cost to roll back.

Capturing Deadlock Graphs

When 1205 errors are reported, it is important that the DBA finds out why the deadlock happened and takes steps to prevent its recurrence. The first step in troubleshooting and resolving a deadlocking problem is to capture the **deadlock graph** information.

A deadlock graph is an output of information regarding the sessions and resources that were involved in a deadlock. The means by which you can capture a deadlock graph have diversified and improved over recent versions of SQL Server. If you are still running SQL Server 2000, then you are stuck with a single, somewhat limited, Trace Flag (**1204**). SQL Server 2005 added a new Trace Flag (**1222**), provided the **XML Deadlock Graph** event in SQL Server Profiler, and enabled deadlock graph capture via Service Broker event notifications, and the WMI (Windows Management Instrumentation) Provider for Server Events. In each case, the deadlock graph contains significantly more information about the nature of the deadlock than is available through Trace Flag 1204. This minimizes the need to gather, manually, additional information from SQL Server in order to understand why the deadlock occurred; for example, resolving the `pageid` for the locks being held to the `objectid` and `indexid`, using `DBCC PAGE`, and using SQL Trace to walk the deadlock chain and find out which currently executing statements are causing the problem. SQL Server 2008 provides all of these facilities, plus the **`system_health`** Extended Events Session.

To allow you to work through each section, and generate the same deadlock graphs that are presented and described in the text, the resource materials for this article include example code to generate a deadlock in SQL Server.

Trace Flag 1204

Trace Flags in SQL Server enable alternate "code paths" at key points inside the database engine, allowing additional code to execute when necessary. If you are seeing queries failing with deadlock errors on a SQL Server instance, Trace Flags can be enabled for a single session or for all of the sessions on that instance. When Trace Flag 1204 is enabled for all sessions on a SQL Server instance, any deadlock detected by the deadlock monitor will cause a deadlock graph to be written to the SQL Server error log.

In SQL Server 2000, this Trace Flag is the only means by which to capture a deadlock graph, which makes troubleshooting deadlocking in SQL Server 2000 quite challenging, though still possible. In later SQL Server versions, this Trace Flag is still available although superseded by Trace Flag 1222.

Trace Flag 1204, like all Trace Flags, can be enabled and disabled on an ad hoc basic using the `DBCC TRACEON` and `DBCC TRACEOFF` database console commands. Listing 1 shows how to enable Trace Flag 1204, for a short term, at the server-level (specified by the `-1` argument) so that all subsequent statements run with this Trace Flag enabled.

```
DBCC TRACEON(1204, -1)
```

Listing 1: Turning on Trace Flag 1204 for all sessions.

Alternatively, Trace Flags can be turned on automatically, using the `-T` startup parameter. To add a startup parameter to SQL Server, right-click on the Server Node in **Enterprise Manager** and open the **Server Properties** page. Under the **General** tab, click the **Startup Parameters** button, and then add the startup parameter to the server as shown in Figure 1.

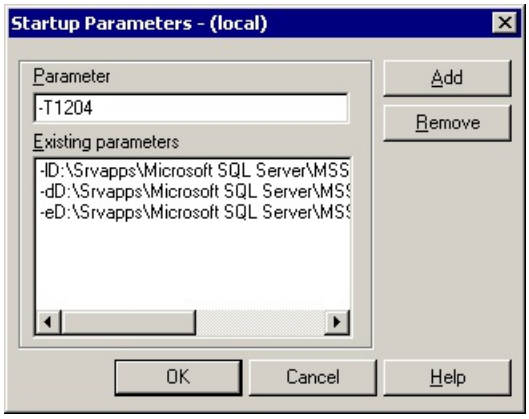


Figure 1: Using the `-T` startup parameter.

In cases where it is possible to perform an instance restart, using a startup parameter can be helpful when you want to capture every deadlock that occurs from the server, over a long period of time. However, once deadlock troubleshooting has been completed, the Trace Flag should be removed from the startup parameters. Since the Trace Flag enables the instance to write the deadlock graph to the SQL Server error log, the only way to retrieve the graph is to read the error log file and then extract the events from the log file for analysis.

Trace Flag 1222

SQL Server 2005 added Trace Flag 1222 to capture the deadlock graphs in an easier-to-read and more comprehensive format than was available with the 1204 flag. It captures and presents the information in a manner that makes it much easier to identify the deadlock victim, as well as the resources and processes involved in the deadlock (covered in detail in the *Reading Deadlock Graphs* section).

Trace Flag 1204 is still available, for backwards compatibility reasons, but when using Trace Flags to capture deadlock graphs in SQL Server 2005 or later, you should always use Trace Flag 1222 in preference to Trace Flag 1204. Trace Flag 1222 is enabled in the same manner as 1204, using `DBCC TRACEON()`, as shown in Listing 1 or the `-T` startup parameter, as shown in Figure 1.

SQL Profiler XML Deadlock Graph event

New to SQL Server 2005, the `Deadlock Graph` event in SQL Trace captures the deadlock graph information, without writing it to the SQL Server Error Log. The `Deadlock Graph` event is part of the **Locks** event category and can be added to a SQL Server Profiler trace by selecting the event in Profiler's **Trace Properties** dialog, as shown in Figure 2.

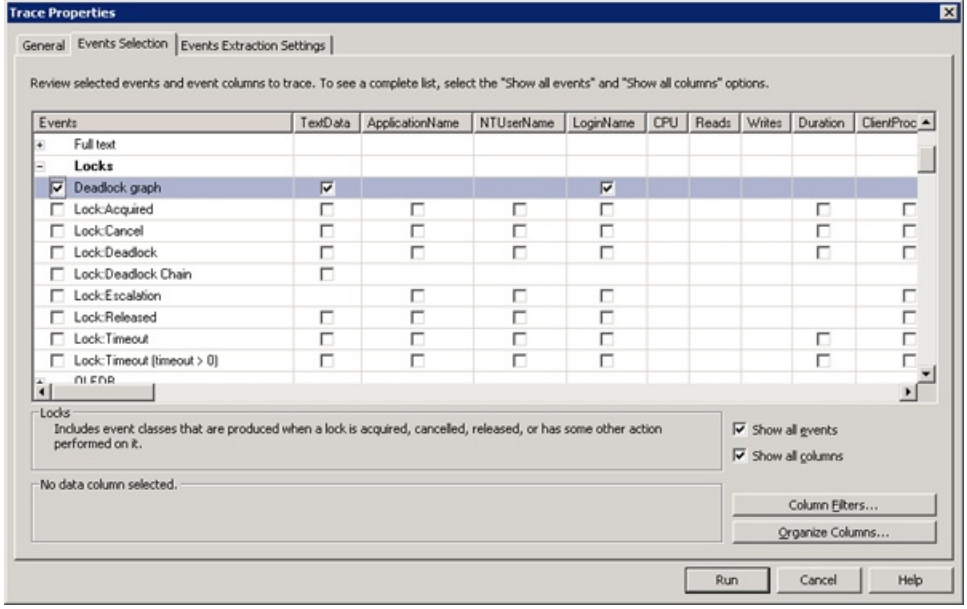


Figure 2: Selecting `Deadlock Graph` event in the Trace Properties dialog.

SQL Profiler can be configured to save the deadlock graphs separately, into XDL files, as shown in Figure 3.

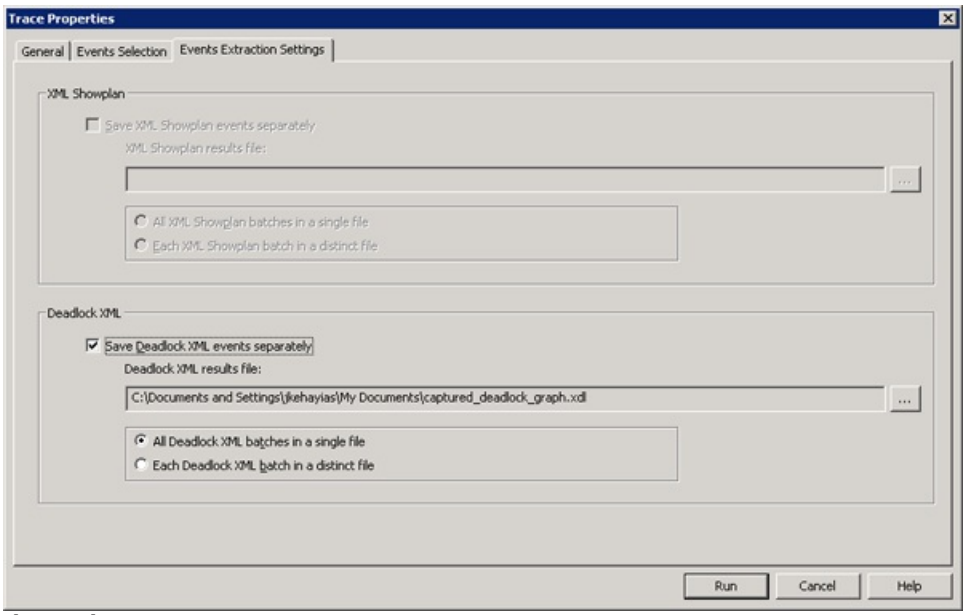


Figure 3: Saving deadlock graphs.

An XDL file is a standard XML file. Management Studio recognizes the file extension when opening the file and displays the deadlock information graphically, rather than as XML.

If you prefer to work directly with server-side traces, removing the overhead of the Profiler client, then you can capture the deadlock graph information directly from your scripts, using the `SP_TRACE_*` set of system stored procedures. The captured graphs will be written to a SQL Trace file on the SQL Server. The easiest way to generate a script for a server-side trace is to first create the trace in SQL Profiler, and then export it to a script using **File** | **Export** | **Script Trace Definition**.

A server-side trace file can be read using the system function `fn_trace_gettable`, or by opening it inside of SQL Profiler. When using SQL Profiler to view the trace file contents, the deadlock events can be exported to individual XDL files that can be opened up graphically using SQL Server Management Studio, through the **File** | **Export** | **Extract SQL Server Events** | **Extract deadlock Events** menu item.

Service Broker event notifications

Also new in SQL Server 2005, event notifications allow the capture of deadlock graph information using SQL Server Service Broker, by creating a **service** and **queue** for the `DEADLOCK_GRAPH` trace event. The information contained in the deadlock graph captured by event notifications is no different than the information contained in the deadlock graph captured by SQL Trace; the only difference is the mechanism of capture.

Setting up an event notification to capture deadlock graph information requires three Service Broker objects:

- A **`QUEUE`** to hold the `DEADLOCK_GRAPH` event messages
- A **`SERVICE`** to route the messages to the queue

- An **`EVENT NOTIFICATION`** to capture the deadlock graph and package it in a message that is sent to the Service.

Listing 2 shows how to create these objects using T-SQL. Note that you need to create the objects in a broker-enabled database, like `msdb`. The `Master` database is not enabled for broker, by default.

```
USE msdb;

--  Create a service broker queue to hold the events
CREATE QUEUE DeadlockQueue
GO

--  Create a service broker service receive the events
CREATE SERVICE DeadlockService
ON QUEUE DeadlockQueue ([http://schemas.microsoft.com/SQL/Notifications/PostEventNotification])
GO

-- Create the event notification for deadlock graphs on the service
CREATE EVENT NOTIFICATION CaptureDeadlocks
ON SERVER
WITH FAN_IN
FOR DEADLOCK_GRAPH
TO SERVICE 'DeadlockService', 'current database' ;
GO
```

Listing 2: Creating the Service Broker service, queue, and event notification objects.

With the objects created, deadlock graphs will be collected in the queue, as deadlocks occur on the server. While the queue can be queried using a `SELECT` statement, just as if it were a table, the contents remain in the queue until they are processed using the `RECEIVE` command, as demonstrated in Listing 3.

```
USE msdb ;
-- Cast message_body to XML and query deadlock graph from TextData
SELECT  message_body.valuequery('(/EVENT_INSTANCE/TextData/
                                 deadlock-list)[1]', 'varchar(128)')
                                 AS DeadlockGraph
FROM    ( SELECT    CAST(message_body AS XML) AS message_body
          FROM      DeadlockQueue
        ) AS sub ;
GO

-- Receive the next available message FROM the queue
DECLARE @message_body XML ;

RECEIVE TOP(1) -- just handle one message at a time
@message_body=message_body
FROM DeadlockQueue ;

-- Query deadlock graph from TextData
SELECT  @message_body.valuequery('(/EVENT_INSTANCE/TextData/
                                  deadlock-list)[1]','varchar(128)')
                                  AS DeadlockGraph
GO
```

Listing 3: Query and processing `DEADLOCK_GRAPH` event messages in the queue.

Since Event Notifications utilize a service broker queue for processing, additional actions can be performed when the deadlock event fires. When a deadlock event occurs, Service Broker can "activate" a stored procedure that processes the message and responds appropriately, for example, by sending an email notification using Database Mail, logging the event in a table, or gathering additional information, like the execution plans for both statements, from SQL Server, based on the information contained inside of the deadlock graph. Full coverage of this topic is beyond the scope of this article. However, a full example of how to use queue activation to completely automate deadlock collection can be found in the code download file for this book.

WMI Provider for server events

Also new to SQL Server 2005, the WMI Provider for Server Events allows WMI to be used to monitor SQL Server events as they occur. Any event that can be captured through event notifications has a corresponding **WMI Event Object**, and any WMI management application can subscribe to these event objects.

SQL Server Agent was updated to manage WMI events, through the use of WMI Query Language (WQL), a query language similar to T-SQL that is used with WMI and Agent Alerts for WMI events.

A full example of how to create a SQL Agent alert to capture and store deadlock graphs is out of scope for this article, and can be found in the Books Online Sample: (*Creating a SQL Server Agent Alert by Using the WMI Provider for Server Events*). However, in essence, it involves creating, via the WMI Event Provider, a SQL Agent alert to monitor deadlock graph events. The alert queries for events using WQL, and when it receive notification that one has occurred, it fires a job that captures the deadlock graph in a designated SQL Server table.

To capture deadlock graphs using the WMI Event Provider and a SQL Agent alert in this manner requires that the "Replace tokens for all job responses to alerts" in SQL Server Agent Alert System properties must be enabled. It also requires that Service Broker (which processes the notification messages) is enabled in `msdb` as well as the database in which the deadlock graphs are stored.

WMI event provider bug

It is worth noting that there is a known bug in the WMI Event Provider for server names that exceed fourteen characters; this was fixed in Cumulative Update 5 for SQL Server 2005 Service Pack 2.

Extended Events

Prior to SQL Server 2008, there was no way to retroactively find deadlock information. Obtaining deadlock graphs required that a SQL Trace was actively running, or that Trace Flag 1222 or 1205 was turned on for the instance. Since tracing deadlocks by either of these methods can be resource intensive, this usually meant that a series of deadlocks had to occur to prompt starting a trace or enabling the Trace Flags.

SQL Server 2008 includes all of the previously discussed techniques for capturing deadlock graphs, and adds one new one, namely collecting the deadlock information through the `system_health` default event session in Extended Events. This default event session (akin, in concept, to the default trace) is running by default on all installations of SQL Server 2008 and collects a range of useful troubleshooting information for errors that occur in SQL Server, including deadlocks. Deadlock graphs captured by Extended Events in SQL Server 2008 have the unique ability to contain information about multi-victim deadlocks (deadlocks where more than session was killed by the Lock Monitor to resolve the conflict).

> **Multi-victim Deadlock Example**
> We can't cover Extended Events in detail in this article but, for a good overview of the topic, read Paul Randal's article, "SQL 2008: Advanced Troubleshooting with Extended Events". Also, an example of how to create a multi-victim deadlock in SQL Server can be found on my blog post from 2009, "Changes to the Deadlock Monitor for the Extended Events xml_deadlock_report and Multi-Victim Deadlocks".

The `system_health` session uses a `ring_buffer` target which stores the information collected by events firing in memory as an XML document in the `sys.dm_xe_session_targets` DMV. This DMV can be joined to the `sys.dm_xe_sessions` DMV to get the session information along with the data stored in the `ring_buffer` target, as shown in Listing 4.

```
SELECT  CAST(target_data AS XML) AS TargetData
FROM    sys.dm_xe_session_targets st
        JOIN sys.dm_xe_sessions s ON s.address = st.event_session_address
WHERE   name = 'system_health'
```

Listing 4: Retrieving `system_health` session information.

The query in Listing 5a shows how to retrieve a valid XML deadlock graph from the default `system_health` session using XQuery, the `target_data` column, and a `CROSS APPLY` to get the individual event nodes. Note that, due to changes in the deadlock graph to support multi-victim deadlocks, and to minimize the size of the event data, the resulting XML cannot be saved as an XDL file for graphical representation.

```
SELECT  CAST(event_data.value('(event/data/value)[1]',
                              'varchar(max)') AS XML) AS DeadlockGraph
FROM    ( SELECT    XEvent.query('.') AS event_data
          FROM      (    -- Cast the target_data to XML
                     SELECT    CAST(target_data AS XML) AS TargetData
                     FROM      sys.dm_xe_session_targets st
                               JOIN sys.dm_xe_sessions s
                                ON s.address = st.event_session_address
                     WHERE     name = 'system_health'
                               AND target_name = 'ring_buffer'
                    ) AS Data -- Split out the Event Nodes
                    CROSS APPLY TargetData.nodes('RingBufferTarget/
                                  event[@name="xml_deadlock_report"]')
                    AS XEventData ( XEvent )
        ) AS tab ( event_data )
```

Listing 5a: Retrieving an XML deadlock graph in SQL Server 2008

Note, also, that there is a bug in the RTM release of SQL Server 2008 that causes deadlock graphs not to be captured and retained in an Extended

Events session. This bug was fixed in Cumulative Update 1 for SQL Server 2008 and is also included in the latest Service Pack. An additional bug exists for malformed XML in the deadlock graph generated by Extended Events, which was corrected in Cumulative Update Package 6 for SQL Server 2008 Service Pack 1. It is still possible to generate a valid XML document in these earlier builds, by hacking the deadlock graph being output by Extended Events. However, since the fix to SQL Server has already been released, the specifics of the work-around will not be covered in this article.

In SQL Server 2012 there are changes associated to how the Extended Events targets store XML data inside of the `value` element of the Event XML output. Listing 5 shows the use of the `.value()` method from XML in SQL Server, but in SQL Server 2012, a `.query()` method has to be used to retrieve the deadlock graph from the Event XML output.

```
-- Retrieve from Extended Events in 2012
SELECT  XEvent.query('(event/data/value/deadlock)[1]') AS DeadlockGraph
FROM    ( SELECT    XEvent.query('.') AS XEvent
          FROM      ( SELECT    CAST(target_data AS XML) AS TargetData
                      FROM      sys.dm_xe_session_targets st
                                JOIN sys.dm_xe_sessions s
                                 ON s.address = st.event_session_address
                      WHERE     s.name = 'system_health'
                                AND st.target_name = 'ring_buffer'
                    ) AS Data
                    CROSS APPLY TargetData.nodes
                   ('RingBufferTarget/event[@name="xml_deadlock_report"]')
                    AS XEventData ( XEvent )
        ) AS src;
```

Listing 5b: Retrieving an XML deadlock graph in SQL Server 2012

Reading Deadlock Graphs

The precise format of the deadlock graph in SQL Server has changed from version to version, and mainly for the better. In general, it now contains better information in an easier-to-digest format, such as the graphical display provided in SQL Server Management Studio and SQL Profiler, so allowing us to more easily troubleshoot deadlocks.

Even with the changes to the deadlock graph XML that is output by Extended Events, in SQL Server 2008, the fundamentals of how to interpret the graph are the same as for any other XML deadlock graph.

Interpreting Trace Flag 1204 deadlock graphs

Perhaps one of the most difficult aspects of troubleshooting deadlocks in SQL Server 2000 is interpreting the output of Trace Flag 1204. The process is complicated by the need to query the `sysobjects` and `sysindexes` system tables to find out exactly what objects are involved in the deadlock.

Listing 6 shows an example deadlock graph that was generated by enabling Trace Flag 1204, and then creating a deadlock situation (the code to do this is provided as part of the code download for this book).

```
Deadlock encountered .... Printing deadlock information

Wait-for graph

Node:1
KEY: 13:1993058136:2 (08009d1c9ab1) CleanCnt:2 Mode: S Flags: 0x0
 Grant List 0::
   Owner:0x567e7660 Mode: S         Flg:0x0 Ref:1 Life:00000000 SPID:54 ECID:0
   SPID: 54 ECID: 0 Statement Type: SELECT Line #: 3
   Input Buf: Language Event: WHILE (1=1)
BEGIN
    INSERT INTO #t1 EXEC BookmarkLookupSelect 4
    TRUNCATE TABLE #t1
END

 Requested By:
   ResType:LockOwner Stype:'OR' Mode: X SPID:55 ECID:0 Ec:(0x26F7DBD8) Value:0x58f80880
Cost:(0/3C)

Node:2
KEY: 13:1993058136:1 (040022ae5dcc) CleanCnt:2 Mode: X Flags: 0x0
 Grant List 1::
```

```
   Owner:0x58f80940 Mode: X        Flg:0x0 Ref:0 Life:02000000 SPID:55 ECID:0
   SPID: 55 ECID: 0 Statement Type: UPDATE Line #: 4
   Input Buf: Language Event: SET NOCOUNT ON
WHILE (1=1)
BEGIN
    EXEC BookmarkLookupUpdate 4
END

 Requested By:
   ResType:LockOwner Stype:'OR' Mode: S SPID:54 ECID:0 Ec:(0x2F881BD8) Value:0x567e76c0 Cost:(0/0)

Victim Resource Owner:
 ResType:LockOwner Stype:'OR' Mode: S SPID:54 ECID:0 Ec:(0x2F881BD8) Value:0x567e76c0 Cost:(0/0)
```

Listing 6: Sample deadlock graph from Trace Flag 1204, involving `KEY` locks.

The first thing to pay attention to in the graph output is that there are two **nodes**, each node representing a locked resource. The first line of output for each node shows the resource on which the lock is held, and then the **Grant List** section provides details of the deadlocking situation, including:

- `Mode` of the lock being held on the resource
- `SPID` of the associated process
- `Statement Type` the `SPID` is currently running
- `Line #` (line number) that marks the start of the currently executing statement
- `Input Buf`, the contents of the input buffer for that `SPID` (the last statement sent).

So, for Node 1, we can see that a shared read (`S`) lock is being held by `SPID 54` on an index `KEY` of a non-clustered index (`:2`) on an object with ID `1993058136`. Node 2 shows that an exclusive (`X`) lock is being held by `SPID 55` on an index key of the clustered index (`:1`) of the same object.

Further down, for each node, is the **Requested By** section, which details any resource requests that cannot be granted, due to blocking. For Node 1, we can see that that `SPID 55` is waiting for an exclusive lock on the non-clustered index key (it is blocked by the `S` lock held by `SPID 54`). For Node 2, we can see that `SPID 54` is waiting to acquire a shared read lock on the clustered index key (it is blocked by the exclusive lock held by `SPID 55`).

Furthermore, back in the **Grant List** section, we can see that `SPID 54` has issued the `SELECT` statement on Line # 3 of the `BookmarkLookupSelect` stored procedure (but is unable to acquire a shared read lock) and `SPID 55` has issued the `UPDATE` statement on Line # 4 of the `BookmarkLookupUpdate` stored procedure (but is unable to acquire an exclusive lock).

This is a classic deadlock situation, and happens to be one of the more common types of deadlock, covered in more detail later in this article, in the section titled *Bookmark lookup deadlock*.

Finally, in the **Victim Resource Owner** section we can find out which `SPID` was chosen as the deadlock victim, in this case, `SPID 54`. Alternatively, we can identify the deadlock victim by matching the binary information in the `Value` to the binary information in the `Owner` portion of the Grant List.

We've discussed a lot about the deadlocking incident, but so far we know only that it occurred on an object with an ID of `1993058136`. In order to identify properly the object(s) involved in the deadlock, the information in the `KEY` entry for each node needs to be used to query `sysobjects` and `sysindexes`. The `KEY` entry is formatted as `databaseid:objected:indexid`. So, in this example, `SPID 54` was holding a Shared (`S`) lock on `index id 2`, a non-clustered index, with `objectID 1993058136`. The query in Listing 7 shows how to determine the table and index names associated with the deadlock.

```
SELECT  o.name AS TableName ,
        i.name AS IndexName
FROM    sysobjects AS o
        JOIN sysindexes AS i ON o.id = i.id
WHERE   o.id = 1993058136
        AND i.indid IN ( 1, 2 )
```

Listing 7: Finding the names of the objects associated with the deadlock.

If a deadlock involves a `PAG` lock instead of a `KEY` lock, the deadlock graph might look as shown in Listing 8.

```
Wait-for graph

Node:1
PAG: 8:1:96                    CleanCnt:2 Mode: X Flags: 0x2
 Grant List 0::
   Owner:0x195fb2e0 Mode: X        Flg:0x0 Ref:1 Life:02000000
```

```
   SPID: 56 ECID: 0 Statement Type: UPDATE Line #: 4
   Input Buf: Language Event: SET NOCOUNT ON
WHILE (1=1)
BEGIN
    EXEC BookmarkLookupUpdate 4
END

 Requested By:
   ResType:LockOwner Stype:'OR' Mode: S SPID:55 ECID:0 Ec:(0x1A5E1560) Value:0x1960dba0 Cost:(0/0)
```

Listing 8: Page lock example, generated by Trace Flag 1204.

Notice now that the lock reference is of the form `databaseid:fileid:pageid`. In order to identify the object to which this page belongs, we need to enable Trace Flag 3604, dump the page header information to the client message box `DBCC PAGE()`, and then disable the Trace Flag, as shown in Listing 9.

```
DBCC TRACEON(3604)
DBCC PAGE(8,1,96,1)
DBCC TRACEOFF(3604)
```

Listing 9: Identifying the objects involved in a deadlock involving page locks.

The output of the `DBCC PAGE()` command will include a `PAGE HEADER` section, shown in Listing 10, which contains the IDs of the object (`m_objId` field) and index (`m_indexId`) to which the page belongs.

```
Page @0x1A5EC000
----------------
m_pageId = (1:96)          m_headerVersion = 1        m_type = 1
m_typeFlagBits = 0x0       m_level = 0                m_flagBits = 0x4
m_objId = 1977058079       m_indexId = 0              m_prevPage = (0:0)
m_nextPage = (1:98)        pminlen = 116              m_slotCnt = 66
m_freeCnt = 110            m_freeData = 7950          m_reservedCnt = 0
m_lsn = (912:41:3)         m_xactReserved = 0         m_xdesId = (0:0)
m_ghostRecCnt = 0          m_tornBits = 2
```

Listing 10: Page Header section from the output of the `DBCC PAGE()`.

Understanding the statements that are being executed along with the indexes and objects involved in the deadlock is critical to troubleshooting the problem. However, there are situations where the currently executing statement may not be the actual statement that caused the deadlock. Multi-statement stored procedures and batches that enlist an explicit transaction will hold all of the locks acquired under the transaction scope until the transaction is either committed or rolled back. In this situation, the deadlock may involve locks that were acquired by a previous statement that was executed inside the same transaction block. To completely troubleshoot the deadlock it is necessary to look at the executing batch from the `Input Buf` as a whole, and understand when locks are being acquired and released.

Interpreting Trace Flag 1222 deadlock graphs

The format of the information, as well as the amount of information, returned by Trace Flag 1222 is very different than the output from Trace Flag 1204. Listing 11 shows the Trace Flag 1222 output, in SQL Server 2005, for an identical deadlock to the one previously seen for the Trace Flag 1204 output, from SQL Server 2000.

```
deadlock-list
 deadlock victim=process8d8c58

  process-list
   process id=process84b108 taskpriority=0 logused=220 waitresource=KEY: 34:72057594038452224
(0c006459e83f) waittime=5000 ownerId=899067977 transactionname=UPDATE lasttranstarted=2009-12-
13T00:22:46.357 XDES=0x157be250 lockMode=X schedulerid=1 kpid=4340 status=suspended spid=102
sbid=0 ecid=0 priority=0 transcount=2 lastbatchstarted=2009-12-13T00:13:37.510
lastbatchcompleted=2009-12-13T00:13:37.507 clientapp=Microsoft SQL Server Management Studio -
Query hostname=SQL2K5TEST hostpid=5516 loginname=sa isolationlevel=read committed (2)
xactid=899067977 currentdb=34 lockTimeout=4294967295 clientoption1=673187936 clientoption2=390200
    executionStack
     frame procname=DeadlockDemo.dbo.BookmarkLookupUpdate line=4 stmtstart=260 stmtend=394
sqlhandle=0x03002200e7a4787d08a10300de9c0000010000000000000000
UPDATE BookmarkLookupDeadlock SET col2 = col2-1 WHERE col1 = @col2
     frame procname=adhoc line=4 stmtstart=82 stmtend=138
sqlhandle=0x020000002a7093322fbd674049d04f1dc0f3257646c4514b
EXEC BookmarkLookupUpdate 4
```

```
    inputbuf
SET NOCOUNT ON
WHILE (1=1)
BEGIN
    EXEC BookmarkLookupUpdate 4
END
   process id=process8d8c58 taskpriority=0 logused=0 waitresource=KEY: 34:72057594038386688
(0500b49e5abb) waittime=5000 ownerId=899067972 transactionname=INSERT EXEC lasttranstarted=2009-
12-13T00:22:46.357 XDES=0x2aebba08 lockMode=S schedulerid=2 kpid=5864 status=suspended spid=61
sbid=0 ecid=0 priority=0 transcount=1 lastbatchstarted=2009-12-13T00:22:46.347
lastbatchcompleted=2009-12-13T00:22:46.343 clientapp=Microsoft SQL Server Management Studio -
Query hostname=SQL2K5TEST hostpid=5516 loginname=sa isolationlevel=read committed (2)
xactid=899067972 currentdb=34 lockTimeout=4294967295 clientoption1=673187936 clientoption2=390200
    executionStack
     frame procname=DeadlockDemo.dbo.BookmarkLookupSelect line=3 stmtstart=118 stmtend=284
sqlhandle=0x03002200ae80847c07a10300de9c00000100000000000000
SELECT col2, col3 FROM BookmarkLookupDeadlock WHERE col2 BETWEEN @col2 AND @col2+1
     frame procname=adhoc line=3 stmtstart=50 stmtend=146
sqlhandle=0x02000000e00b66366c680fabe2322acbad592a896dcab9cb
INSERT INTO #t1 EXEC BookmarkLookupSelect 4
    inputbuf
WHILE (1=1)
BEGIN
    INSERT INTO #t1 EXEC BookmarkLookupSelect 4
    TRUNCATE TABLE #t1
END

  resource-list
   keylock hobtid=72057594038386688 dbid=34 objectname=DeadlockDemo.dbo.BookmarkLookupDeadlock
indexname=cidx_BookmarkLookupDeadlock id=lock137d65c0 mode=X associatedObjectId=72057594038386688
owner-list
     owner id=process84b108 mode=X
    waiter-list
    waiter id=process8d8c58 mode=S requestType=wait
   keylock hobtid=72057594038452224 dbid=34 objectname=DeadlockDemo.dbo.BookmarkLookupDeadlock
indexname=idx_BookmarkLookupDeadlock_col2 id=lock320d5900 mode=S
associatedObjectId=72057594038452224
    owner-list
    owner id=process8d8c58 mode=S
    waiter-list
    waiter id=process84b108 mode=X requestType=wait
```

Listing 11: Sample deadlock graph, generated by Trace Flag 1222.

The new format breaks a deadlock down into sections that define the deadlock victim, the processes involved in the deadlock (`process-list`), and the resources involved in the deadlock (`resource-list`). Each process has an assigned `process id` that is used to uniquely identify it in the deadlock graph. The deadlock victim lists the process that was selected as the victim and killed by the deadlock monitor. Each process includes the `SPID` as well as the `hostname` and `loginname` that originated the request, and the isolation level under which the session was running when the deadlock occurred. The `execution stack` section, for each process, displays the entire execution stack, starting from the most recently executed (deadlocked) statement backwards to the start of the call stack. This eliminates the need to perform additional steps to identify the statement being executed.

The `resource-list` contains all of the information about the resources involved in the deadlock and is generally the starting point for reading a deadlock graph. The index names are included in the output and each resource displays the owner process and the type of locks being held, as well as the waiting process and the type of locks being requested.

The definitive source for understanding the output from Trace Flag 1222 is Bart Duncan. He has a three-part series on troubleshooting deadlocks with the output from Trace Flag 1222 on his blog, starting with (*Deadlock Troubleshooting, Part 1*).

Using the same technique employed in these posts, we can construct a description of the deadlock described, as shown in Listing 12.

```
SPID 102 (process84b108) is running this query (line 4 of the BookmarkLookupUpdate sproc):
   UPDATE BookmarkLookupDeadlock SET col2 = col2-1 WHERE col1 = @col2

SPID 61 (process8d8c58 )is running this query (line 3 of BookmarkLookupSelect sproc):
   SELECT col2, col3 FROM BookmarkLookupDeadlock WHERE
col2                                                BETWEEN @col2 AND @col2+1

SPID 102 is waiting for an Exclusive KEY lock on the idx_BookmarkLookupDeadlock_col2 index (on the
```

```
BookmarkLookupDeadlock table).
   (SPID 61 holds a conflicting S lock)

SPID 61 is waiting for a Shared KEY lock on the index cidx_BookmarkLookupDeadlock (on the
BookmarkLookupDeadlock table)..
   (SPID 102 holds a conflicting X lock)
```

Listing 12: Deadlock analysis, constructed from the Trace Flag 1222 deadlock graph.

As we can see from the deadlock list section of Listing 11, SPID 61, attempting to run the `SELECT` statement against `cidx_BookmarkLookupDeadlock`, is chosen as the deadlock victim.

Interpreting XML deadlock graphs

The information contained in XML deadlock graph, obtained from SQL Profiler, or Service Broker Event notifications, and so on, is essentially the same as that obtained from the output of Trace Flag 1222, and it is interpreted in exactly the same way. However, the format in which the information is presented is very different. The XML deadlock graph can be displayed graphically in Management Studio by saving the XML to a file with a `.XDL` extension and then opening the file in Management Studio (although, as discussed earlier, the XML generated by Extended Events can't be displayed graphically, in this manner).

Figure 4 displays graphically the same deadlock graph that we saw for the two Trace Flags.

Figure 4: SSMS graphical deadlock graph.

In the graphical display, the deadlock processes are displayed as ovals. The process information is displayed inside of the oval, and includes a tooltip, which pops up when the mouse hovers over the process, and displays the statement being executed, as shown in Figure 5. The deadlock victim process is shown crossed out.

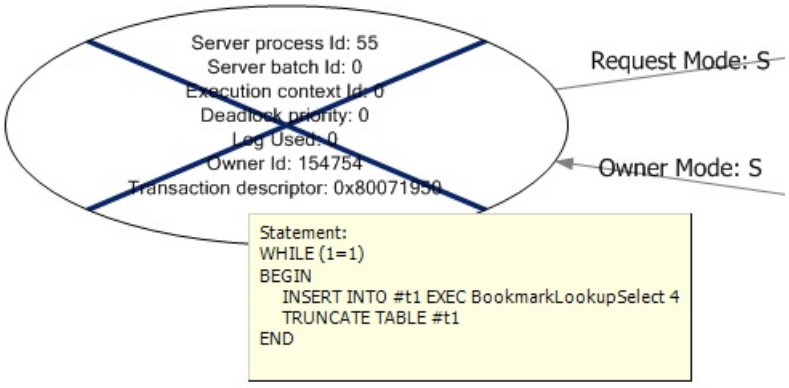


Figure 5: SSMS graphical deadlock graph: the victim process.

The resources contributing to the deadlock are displayed in rectangular boxes in the center of the graphical display. The locks, and their respective modes, are displayed by arrows between the processes and the resources. Locks owned by a process are shown with the arrow pointed towards the process, while locks being requested are shown with the arrow pointed towards the resource as shown in Figure 6.

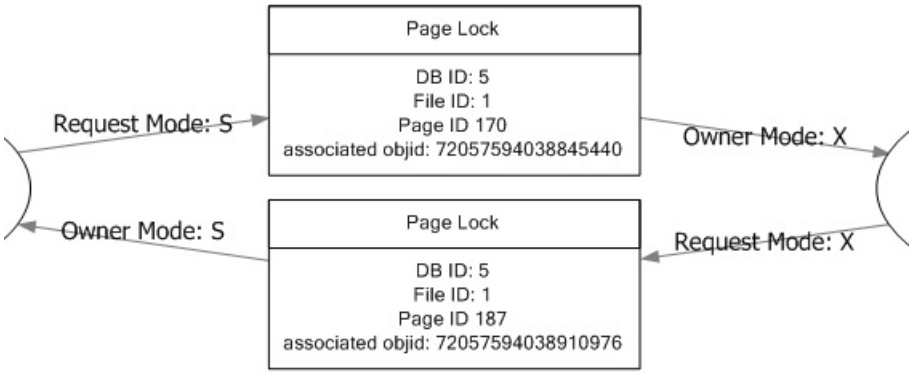


Figure 6: SSMS graphical deadlock graph: processes and resources.

A visual display like this makes it much easier to understand the circular blocking that caused the deadlock to occur.

Common types of deadlock and how to eliminate them

When troubleshooting any type of problem in SQL Server, you learn with experience how to recognize, from a distance, the particular varieties of problem that tend to crop up on a regular basis. The same is true of deadlocks; the same types of deadlock tend to appear with predictable regularity and, once you understand what patterns to look for, resolving the deadlock becomes much more straightforward.

This section assumes knowledge of basic locking mechanisms inside SQL Server and examines how to resolve the most common types of deadlock, namely the **bookmark lookup** deadlock, the **serializable range scan** deadlock, the **cascading constraint** deadlock, the **intra-query parallelism** deadlock and the **accessing objects in different orders** deadlock.

Bookmark lookup deadlock

Bookmark lookup deadlocks are one of the most common deadlocks in SQL Server. Fortunately, although they have a habit of appearing randomly, without any changes to the database or the code inside of it, they are also one of the easiest types of deadlock to troubleshoot.

Bookmark lookup deadlocks generally have a `SELECT` statement as the victim, and an `INSERT`, `UPDATE`, or `DELETE` statement as the other contributing process to the deadlock. They occur partly as a general consequence of SQL Server's pessimistic locking mechanisms for concurrency, but mainly due to the lack of an appropriate covering index for the `SELECT` operation.

When a column is used in the `WHERE` clause to filter the `SELECT` statement and a non-clustered index exists on that column, then the database engine takes a shared lock on the required rows or pages in the non-clustered index. In order to return any additional columns from the table, not covered by the non-clustered index, the database engine performs an operation known as KEY, or RID, lookup (in SQL Server 2000, the term "bookmark lookup" was used). This operation uses either the Clustered Index Key or RID (in the case of a heap) to look up the row in the table data and retrieve the additional columns.

When a lookup operation occurs, the database engine takes additional shared locks on the rows or pages needed from the table. These locks are held for the duration of the `SELECT` operation, or until lock escalation is triggered to increase the lock granularity from row or page to table.

The deadlock occurs, as we have seen in previous sections, when an operation that changes the data in a table (for example, an `INSERT`, `UPDATE`, or `DELETE` operation) occurs simultaneously with the `SELECT`. When the data-changing session executes, it acquires an exclusive lock on the row or page of the clustered index or table, and performs the data change operation. At the same time the `SELECT` operation acquires a shared lock on the non-clustered index. The data-changing operation requires an exclusive lock on the non-clustered index to complete the modification, and the `SELECT` operation requires a shared lock on the clustered index, or table, to perform the bookmark lookup. Shared locks and exclusive locks are incompatible, so if the data-changing operation and the `SELECT` operation affect the same rows then the data-changing operation will be blocked by the `SELECT`, and the `SELECT` will be blocked by the data change, resulting in a deadlock.

One of the most common online recommendations for curing this type of deadlock is to use a `NOLOCK` table hint in the `SELECT` statement, to prevent it from acquiring shared locks. This is bad advice. While it might prevent the deadlock, it can have unwanted side effects, such as allowing operations to read uncommitted changes to the database data, and so return inaccurate results.

The correct fix for this type of deadlock is to change the definition of the non-clustered index so that it contains, either as additional key columns or as `INCLUDE` columns, all the columns it needs to cover the query. Columns returned by the query that are not used in a `JOIN`, `WHERE`, or `GROUP BY` clause, can be added to the index as `INCLUDE` columns. Any column used in a `JOIN`, the `WHERE` clause, or in a `GROUP BY` should ideally be a part of the index key but, in circumstances where this exceeds the 900-byte limit, addition as an `INCLUDE` column may work as well. Implementing the covering index will resolve the deadlock without the unexpected side effects of using `NOLOCK`.

A shortcut to finding the appropriate covering index for a query is to run it through the Database Engine Tuning Advisor (DTA). However, the DTA recommendations are only as good as the supplied workload, and repeated single-query evaluations against the same database can result in an excessive number of indexes, which often overlap. Manual review of any index recommendation made by the DTA should be made to determine if modification of an existing index can cover the query without creating a new index. A good video example, *Using the DTA to Assist in*

Performance Tuning, can be found on the SQL Share website.

Range scans caused by SERIALIZABLE isolation

The `SERIALIZABLE` isolation level is the most restrictive isolation level in SQL Server for concurrency control, ensuring that every transaction is completely isolated from the effects of any other transaction.

To accomplish this level of transactional isolation, **range locks** are used when reading data, in place of the row or page level locking used under `READ COMMITTED` isolation. These range locks ensure that no data changes can occur that affect the result set, allowing the operation to be repeated inside the same transaction with the same result. While the default isolation level for SQL Server is `READ COMMITTED`, certain providers, like COM+ and BizTalk, change the isolation to `SERIALIZABLE` when connections are made.

Range locks have two components associated with their names, the lock type used to lock the range and then the lock type used for locking the individual rows within the range. The four most common range locks are shared-shared (`RangeS-S`), shared-update (`RangeS-U`), insert-null (`RangeI-N`), and exclusive (`RangeX-X`). Deadlocks associated with `SERIALIZABLE` isolation are generally caused by lock conversion, where a lock of higher compatibility, such as a `RangeS-S` or `RangeS-U` lock, needs to be converted to a lock of lower compatibility, such as a `RangeI-N` or `RangeX-X` lock.

A common deadlock that occurs under `SERIALIZABLE` isolation has a pattern that involves a transaction that checks if a row exists in a table before inserting or updating the data in the table. A reproducible example of this deadlock is included in the code examples for this article. This type of deadlock will generally produce a deadlock graph with a `resource-list` similar to the one shown in Listing 13.

```
    <resource-list>
      <keylock hobtid="72057594050969600" dbid="5"
objectname="AdventureWorks.Sales.SalesOrderHeader" indexname="IX_SalesOrderHeader_CustomerID"
id="lock35bcc80" mode="RangeS-U" associatedObjectId="72057594050969600">
        <owner-list>
          <owner id="processad4d2e8" mode="RangeS-U" />
          <owner id="process9595b8" mode="RangeS-S" />
        </owner-list>
        <waiter-list>
          <waiter id="processad4d2e8" mode="RangeI-N" requestType="convert" />
          <waiter id="process9595b8" mode="RangeI-N" requestType="convert" />
       </waiter-list>
     </keylock>
    </resource-list>
```

Listing 13: Extract from a deadlock graph for a `SERIALIZABLE` range scan deadlock.

In this example, two processes have acquired compatible shared locks, `RangeS-S` and `RangeS-U`, on the `SalesOrderHeader` table. When one of the processes requires a lock conversion to a lock type that is incompatible with the lock being held by the other process, in this case a `RangeI-N`, it is blocked. If both processes require a lock conversion to `RangeI-N` locks, the result is a deadlock since each session is waiting on the other to release its high compatibility lock.

There are several possible solutions to this type of deadlock and the most appropriate one depends on the database and the application it supports. If it is not necessary for the database to maintain the range locks acquired during the `SELECT` operation that checks for row existence and the `SELECT` operation can be moved outside of the transaction that performs the data change, then the deadlock can be prevented.

If the operation doesn't require the use of `SERIALIZABLE` isolation, then changing the isolation level to a less restrictive isolation level, for example `READ COMMITTED`, will prevent the deadlock and allow a greater degree of concurrency.

If neither of these solutions is appropriate, the deadlock can be resolved by forcing the `SELECT` statement to use a lower-compatibility lock, through the use of an `UPDLOCK` or `XLOCK` table hint. This will block any other transactions attempting to acquire locks of higher compatibility. This fix is specific to this particular type of deadlock due to the usage of `SERIALIZABLE` isolation. Using `UPDLOCK` hints under `READ COMMITTED` may result in deadlocks occurring more frequently under certain circumstances.

Cascading constraint deadlocks

Cascading constraint deadlocks are generally very similar to a Serializable Range Scan deadlock, even though the isolation level under which the victim transaction was running isn't `SERIALIZABLE`. To enforce cascading constraints, SQL Server has to traverse the `FOREIGN KEY` hierarchy to ensure that orphaned child records are not left behind, as the result of an `UPDATE` or `DELETE` operation to a parent table. To do this requires that the transaction that modifies the parent table be isolated from the effects of other transactions, in order to prevent a change that would violate `FOREIGN KEY` constraints, when the cascade operation subsequently completes.

Under the default `READ COMMITTED` isolation, the database engine would acquire and hold, for the duration of the transaction, Exclusive locks on all rows that had to be changed. This blocks users from reading or changing the affected rows, but it doesn't prevent another session from adding a new row into a child table for the parent key being deleted. To prevent this from occurring, the database engine acquires and holds range locks,

which block the addition of new rows into the range affected by the cascading operation. This is essentially an under-the-cover use of `SERIALIZABLE` isolation, during the enforcement of the cascading constraint, but the isolation level for the batch is not actually changed; only the type of locks used for the cascading operation are changed.

When a deadlock occurs during a cascading operation, the first thing to look for is whether or not non-clustered indexes exist for the `FOREIGN KEY` columns that are used. If appropriate indexes on the `FOREIGN KEY` columns do not exist, the locks being taken to enforce the constraints will be held for longer periods of time, increasing the likelihood of a deadlock between two operations, if a lock conversion occurs.

Intra-query parallelism deadlocks

An intra-query parallelism deadlock occurs when a single session executes a query that runs with parallelism, and deadlocks itself. Unlike other deadlocks in SQL Server, these deadlocks may actually be caused by a bug in the SQL Server parallelism synchronization code, rather than any problem with the database or application design. Since there are risks associated with fixing some bugs, it may be that the bug is known and won't be fixed, since it is possible to work around it by reducing the degree of parallelism for that the query, using the `MAXDOP` query hint, or by adding or changing indexes to reduce the cost of the query or make it more efficient.

The deadlock graph for a parallelism deadlock will have the same `SPID` for all of the processes, and will have more than two processes in the `process-list`. The `resource-list` will have threadpool, exchangeEvent, or both, listed as resources, but it won't have lock resources associated with it. In addition, the deadlock graph for this type of deadlock will be significantly longer than any other type of deadlock, depending on the degree of parallelism and the number of nodes that existed in the execution plan.

Additional information about this specific type of deadlock can be found on Bart Duncan's blog post, (*Today's Annoyingly-Unwieldy Term: "Intra-Query Parallel Thread Deadlocks"*.

Accessing objects in different orders

One of the easiest deadlocks to create, and consequently one of the easiest to prevent, is caused by accessing objects in a database in different operation orders inside of T-SQL code, inside of transactions, as shown in Listings 14 and 15.

```
BEGIN TRANSACTION

UPDATE  TableA
SET     Column1 = 1

SELECT  Column2
FROM    TableB
```

Listing 14: Transaction1 updates `TableA` then reads `TableB`.

```
BEGIN TRANSACTION

UPDATE  TableB
SET     Column2 = 1

SELECT  Column1
FROM    TableA
```

Listing 15: Transaction2 updates `TableB` then reads `TableA`.

Transaction1's `UPDATE` against `TableA` will result in an exclusive lock being held on the table until the transaction completes. At the same time, Transaction2 runs an `UPDATE` against `TableB`, which also results in an exclusive lock being held until the transaction completes. After completing the `UPDATE` to `TableA`, Transaction1 tries to read `TableB` but is blocked and unable to acquire the necessary shared lock, due to the exclusive lock being held by Transaction2. After completing its `UPDATE` to `TableB`, Transaction2 reads `TableA` and is also blocked, unable to acquire a shared lock due to the exclusive lock held by Transaction1. Since the two transactions are both blocking each other, the result is a deadlock and the Lock Monitor will kill one of the two sessions, rolling back its transaction to allow the other to complete.

When using explicit transactions in code, it is important that objects are always accessed in the same order, to prevent this type of deadlock from occurring.

Handling Deadlocks to Prevent Errors

In most cases, the same issues that cause severe blocking in the database, such as poor database design, lack of indexing, poorly designed queries, inappropriate isolation level and so on, are also the common causes of deadlocking. In most cases, by fixing such issues, we can prevent deadlocks from occurring. Unfortunately, by the time deadlocks become a problem, it may not be possible to make the necessary design changes to correct them.

Therefore, an important part of application and database design is defensive programming; a technique that anticipates and handles exceptions as a part of the general code base for an application or database. Defensive programming to handle deadlock exceptions can be implemented in two different ways:

- database-side, through the use of T-SQL `TRY…CATCH` blocks
- application-side, through the use of application `TRY…CATCH` blocks.

In either case, proper handling of the 1205 exception raised by SQL Server for the deadlock victim can help avoid `UnhandledException` errors in the application and the ensuing end-user phone calls to Help Desk or Support.

T-SQL TRY…CATCH blocks

Depending on how an application is designed, and whether there is separation between application code and database code, the simplest implementation of deadlock error handling could be via the use of `BEGIN TRY/CATCH` blocks inside of the T-SQL being executed.

This technique is most applicable in cases where an application calls stored procedures for all of its data access. In such cases, changing the code in a stored procedure so that it handles the deadlock exception doesn't require changes to application code, or recompiling and redistribution of the application. This greatly simplifies the implementation of such changes.

The best way to deal with a deadlock, within your error handling code, will depend on your application and its expected behavior in the event of a deadlock. One way of handling the deadlock would be to retry the transaction a set number of times before actually raising an exception back to the application for handling. The cross-locking situation associated with a deadlock generally only lasts a very short duration, usually timed in milliseconds so, more often than not, a subsequent attempt at executing the T-SQL code selected as a victim will succeed, and there will be no need to raise any exceptions to the application.

However, it is possible that the deadlock will continue to occur, and we need to avoid getting into an infinite loop, attempting repeatedly to execute the same failing code. To prevent this, a variable is used to count down from a maximum number of retry attempts; when zero is reached, an exception will be raised back to the application. This technique is demonstrated in Listing 16.

```
DECLARE @retries INT ;
SET @retries = 4 ;

WHILE ( @retries > 0 )
    BEGIN
        BEGIN TRY
            BEGIN TRANSACTION ;

         -- place sql code here
            SET @retries = 0 ;

            COMMIT TRANSACTION ;
        END TRY
        BEGIN CATCH
        -- Error is a deadlock
            IF ( ERROR_NUMBER() = 1205 )
                SET @retries = @retries - 1 ;

        -- Error is not a deadlock
            ELSE
                BEGIN
                    DECLARE @ErrorMessage NVARCHAR(4000) ;
                    DECLARE @ErrorSeverity INT ;
                    DECLARE @ErrorState INT ;

                    SELECT  @ErrorMessage = ERROR_MESSAGE() ,
                            @ErrorSeverity = ERROR_SEVERITY() ,
                            @ErrorState = ERROR_STATE() ;

                    -- Re-Raise the Error that caused the problem
                    RAISERROR (@ErrorMessage, -- Message text.
                       @ErrorSeverity, -- Severity.
                       @ErrorState -- State.
                       ) ;
                    SET @retries = 0 ;
                END

            IF XACT_STATE() <> 0
                ROLLBACK TRANSACTION ;
```

```
        END CATCH ;
    END ;
GO
```

Listing 16: `TRY…CATCH` handling of deadlock exceptions, in T-SQL.

Handling ADO.NET SqlExceptions in .NET code

While it is possible to handle deadlocks in SQL Server 2005 and 2008, using `BEGIN TRY` and `BEGIN CATCH` blocks, the same functionality doesn't exist in SQL Server 2000, and in any event it may not be acceptable to have the database engine retry the operation automatically. In either case, the client application should be coded to handle the deadlock exception that is raised by SQL Server.

There isn't much difference between the error handling in .NET and the error handling in T-SQL. A `TRY…CATCH` block is used to execute the SQL call from the application and catch any resulting exception raised by SQL Server. If the code should reattempt the operation in the event of a deadlock, a maximum number of retries should be set by a member variable that is decremented each time a deadlock is encountered.

The example in Listing 17 shows how to catch the `SqlException` in C#, but can be used as a model to handle deadlocks in other languages as well.

```
int retries = 4;
while (retries > 0)
{
  try
  {
     // place sql code here
     retries = 0;
  }
  catch (SqlException exception)
  {
     // exception is a deadlock
     if (exception.Number == 1205)
     {
          // Delay processing to allow retry.
          Thread.Sleep(500);
          retries --;
     }
     // exception is not a deadlock
     else
     {
      throw;
     }
  } }
```

Listing 17: `TRY…CATCH` handling of deadlock exceptions, in C#.

Rather than retrying the operation, it may be desirable to log the exception in the Windows Application Event Log, or perhaps display a `MessageBox` dialog and determine whether or not to retry the operation, based on user input. These are two examples of how handling the deadlock exception in the application code allows for more flexibility over handling the deadlock in the database engine.

Controlling Deadlock Behavior with Deadlock Priority

There are circumstances (for example, a critical report that performs a long running `SELECT` that must complete even if it is the ideal deadlock victim) where it may be preferable to specify which process will be chosen as the deadlock victim in the event of a deadlock, rather than let SQL Server decide based purely on the cost of rollback. As demonstrated in Listing 18, SQL Server offers the ability to set, at the session or batch level, a deadlock priority using the `SET DEADLOCK PRIORITY` option.

```
-- Set a Low deadlock priority
SET DEADLOCK_PRIORITY LOW ;
GO

-- Set a High deadlock priority
SET DEADLOCK_PRIORITY HIGH ;
GO

-- Set a numeric deadlock priority
SET DEADLOCK_PRIORITY 2 ;
```

Listing 18: Setting deadlock priority.

A process running in a batch or session with a low deadlock priority will be chosen as the deadlock victim over one that is running with a higher deadlock priority. Like all other `SET` options in SQL Server, the `DEADLOCK PRIORITY` is only in effect for the current execution scope. If it is set inside of a stored procedure, then when the stored procedure execution completes, the priority returns to the original priority of the calling execution scope.

Note that SQL Server 2000 offers only two deadlock priorities; `Low` and `Normal`. This allows the victim to be determined by setting its priority to `Low`. SQL Server 2005 and 2008 however, have three named deadlock priorities; `Low`, `Normal`, and `High`, as well as a numeric range from -10 to +10, for fine-tuning the deadlock priority of different operations.

The deadlock priority is set at execution time, and all users have the permission to set a deadlock priority. This can be a problem if users have ad hoc query access to SQL Server, and set their deadlock priority higher than other processes, in order to prevent their own process from being selected as a victim.

Summary

This article has covered how to capture and interpret deadlock graph information in SQL Server to troubleshoot deadlocking. The most common deadlocks have also been covered to provide a foundation for troubleshooting other types of deadlocks that might occur. Most often, deadlocks are the result of a design problem in the database or code that can be fixed to prevent the deadlock from occurring. However, when changes to the database are not possible to resolve the deadlock, adding appropriate error handling in the application code reduces the impact caused by a deadlock occurring. The information included in this article should allow rapid and efficient troubleshooting of most deadlocks in SQL Server.

DIVs of equal height

Published Sunday, April 29, 2012 9:35 AM

It's the same old old problem you want to make a set of columns the same height but life it too short for the CSS only version. It's technically possible to do but nowadays you can't run the web without having javascript turned on. There must be an easier way.

After a short amount of googling I came across a few solutions. A couple were GPL'd which ruled them straight out as I want Red Gate to pay for my mortgage.

The best simple solution was found at.

http://www.cssnewbie.com/equal-height-columns-with-jquery/

```
function equalHeight(group) {
   var tallest = 0;
   group.Each(function() {
       var thisHeight = $(this).height();
       if(thisHeight > tallest) {
           tallest = thisHeight;
       }
   });
   group.height(tallest);
}
```

Now this worked for my particular situation but wasn't quite nice enough for me. So I thought I'd put it forward to our internal web development e-mail list to see what they made of the issue. This is a list made of developers, designers, marketers who use HTML so can be quite a useful forum for getting good ideas. This threw up the beautiful code from Matt Lee.

> *Or, if you like a bit of code golf (and are using underscore.js, which you should be!):*
>
> ```
> $.fn.equalize = function () {
>       return this.css("height", _.max(this.map(function () { return $(this).height(); })));
>   };
> ```
>
> *See the whole thing at: http://jsbin.com/isixib/edit#javascript,html,live*
>
> *Also, I don't think Richard's version worked if the elements were in increasing height order: http://jsbin.com/abovoq/edit#javascript,html,live*

This seems to do the job really nicely and I thought it was so elegant that it should be shared. enjoy. The way I call this is to on every page perform the following code to equalize groups of divs to be the same height - it will also work for multiple rows and columns of divs: http://jsbin.com/isixib/2/edit#javascript,html,live

```
$("div.height1").equalize();
$("div.height2").equalize();
$("div.height3").equalize();
```

by Richard Mitchell

The PoSh DBA: Grown-Up PowerShell Functions

10 May 2012
by Laerte Junior

> Laerte goes step-by-step through the process of tidying up and making more reusable an untidy collection of PowerShell routines, showing how pipelines and advanced functions can make PowerShell more effective in helping to automate many of the working DBA's chores.

There comes a time after you've got used to using PowerShell to automate your DBA chores that you look at all those PowerShell functions you've written, and feel slightly dissatisfied. You realize, although everything works, you could, be using PowerShell in way that is more easily enhanced, reused and maintained. With some fairly simple techniques, our PowerShell functions can become more reusable. The be made to behave much more like the built-in cmdlets that are provided with PowerShell. You will, for example, want your functions to participate in pipelines, so that you can use filters, sort the order, group objects, write out the data and take advantage of other useful PowerShell facilities. You'd probably benefit from other features of cmdlets such as standard parameters, debug, 'whatIf', and command-line Help. This article will be explaining how these features can help with scripting.

Reusable Scripts

If you take a look at my old scripts on Simple-Talk, you will see some functions that I wrote assuming that I needed to send the object populated with all the information. Something like :

```
Function Get-MSSQLServerInfo ($ServersList)
{
       [reflection.assembly]::LoadWithPartialName("Microsoft.SqlServer.Smo") | out-null
       $FinalResult = @()
       $ServersList = Get-Content $ServersList
        foreach ($svr in  $ServersList ) {
              $Server=New-Object "Microsoft.SqlServer.Management.Smo.Server" "$svr"
              $Object = New-Object PSObject   -Property
     @{         ServerNname = $svr
                           EngineEdition   = $Server.EngineEdition
                           Product = $Server.Product
                           ProductLevel = $Server.ProductLevel }
              $FinalResult += $Object
       }
       Write-Output $FinalResult
}
```

This code will run perfectly. However, it isn't a reusable function. I need to send the function a list of servers as a parameter but, if I need to get any information about my servers within other function, I cannot use this function to do it; I would need to write all this again inside the other function.

I would need a series of filters, the first of which outputted a stream of server objects. The next one would add the server information, the next one filtering just the information I needed, and the next one, maybe displaying it. I need to think in terms of pipelines

How does a Pipeline Work?

I suspect that you are already familiar with the phrase "I am piping.." or "pipe to". Other Shell languages use this 'pipe' concept, but PowerShell is different in that you are "piping" objects and not their text-representations. The 'pipe' is more like an assembly line, where a stream of objects is passed from the start to the end of the line, and may be changed, filtered out, or added to. The output of one filter becomes the input of the next. It is represented by the conventional 'pipe' character "|"

In this line …

```
Get-ChildItem c:\posh -Filter "*.ps1"
```

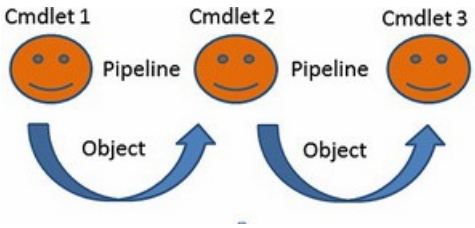
… I am Looking for files with the PowerShell (PS1) file-type in the c:\posh folder and my output is a directory listing like this :

```
    Directory: C:\posh


Mode                LastWriteTime     Length Name
```

```
----                 -------------     ------ ----
-a---        26/04/2012     14:40       2422 eventhandler.ps1
-a---        26/04/2012     14:40        697 filesystemwatcher.ps1
-a---        15/07/2007     22:06        157 install.ps1
-a---        26/04/2012     14:40       1532 sqlbackup.ps1
```

Now , let's send the output to a text file :

```
Get-ChildItem c:\posh -Filter "*.txt" | Out-File c:\temp\FilesTXT.txt
```



PowerShell uses the pipeline to send a stream of objects, thereby encapsulating all the information and methods within the object in an easily accessible way. In this case, the get-ChildItem cmdlet outputs an object called **`System.IO.FileInfo`** with all the information (properties, methods etc.) and the pipeline sends this package information to Out-File. The Out-File Cmdlet can, for example, get the date and length parameter without having to do any parsing of a text stream. It can determine the properties of the **`FileInfo`** object and access them directly.

This would suggest that, if I want to write a function that will be used with pipeline, I would just need to populate an object. However, real-life isn't that simple. I would need to understand how to use pipes in order to write effective PowerShell scripts.

It was then that I was reminded of something I'd read in the Microsoft TechNet

> *"The question, however, is whether you'd want to write PowerShell scripts without using pipelines. You can order a banana split and ask them to hold the ice cream; that's fine, but at that point you don't really have a banana split, do you? The same is true of the PowerShell pipeline: you can write scripts without a pipeline. But, at that point, do you really have a PowerShell script? "*

I needed to clear my head by doing an experiment. Let's see this example to illustrate what I mean

```
function foo1() {
        Write-Host "Foo1 Showing $_"
              $_
}

function foo2() {

        Write-Host "Foo2 Showing $_"
              $_

}

function foo3() {
        Write-Host "Foo3 Showing $_"

}
```

... and run ...

```
1,2,3 |  foo1 | foo2 | foo3
```

We've created three functions and passed them an array of integers. What Are you expecting? I'd assumed that the process was waiting until the first function had populated the entire object and sent it to the next cmdlet. I thought I'd see something like this…

```
Foo1 Showing 1
Foo1 Showing 2
Foo1 Showing 3
Foo2 Showing 1
Foo2 Showing 2
Foo2 Showing 3

Foo3 Showing 1
Foo3 Showing 2
Foo3 Showing 3
```

For my First Surprise the output really was ...

```
Foo1 Showing
Foo2 Showing
Foo3 Showing
```

Doh! I'd forgotten that these objects would be passed in an enumeration, and that I should have iterated over the $input variable with a foreach loop.

```
function foo1() {
                        foreach($element in $input)
        { Write-Host "Foo1 Showing $element"
                        $element
                        }
}

function foo2() {
                        foreach($element in $input)
        { Write-Host "Foo2 Showing $element"
                        $element
                        }

}

function foo3() {
                        foreach($element in $input)
        { Write-Host "Foo3 Showing $element"
                        }
}

1,2,3 |  foo1 | foo2 | foo3
```

and the output is this…

```
Foo1 Showing 1
Foo1 Showing 2
Foo1 Showing 3
Foo2 Showing 1
Foo2 Showing 2
Foo2 Showing 3
Foo3 Showing 1
Foo3 Showing 2
Foo3 Showing 3
```

I could do a lot better than this, since I'm forcing PowerShell to batch up the input to every function until it is all processed and then pass it as an enumeration. In fact, a function/advanced function can be written to accept an input element, process it and then and pass output to the pipeline before the next in the sequence is processed. This gives you better performance and always saves on memory. I should have written code to handle the input stream this way. To do so, we'll need some other language constructs. These are

Begin Block : The code inside this block, will be executed first.
Process Block : The code inside this block will execute once FOR EACH value piped
End Block : When everything is processed, this block execute once the code inside it.

The Syntax is something like this:

```
Function Foo {

      Begin {}
      Process {}
      End {}

}
```

So firstly, let's see what happens when our example is rewritten just using the Process Block :

```
function foo1() {
```

```
        PROCESS {
         Write-Host "Foo1 Showing $_"
               $_
        }
}

function foo2() {
        Process {

         Write-Host "Foo2 Showing $_"
               $_
        }
}

function foo3() {
        Process {
         Write-Host "Foo3 Showing $_"

        }
}


1,2,3 |  foo1 | foo2 | foo3
```

And the Output is :

```
Foo1 Showing 1
Foo2 Showing 1
Foo3 Showing 1
Foo1 Showing 2
Foo2 Showing 2
Foo3 Showing 2
Foo1 Showing 3
Foo2 Showing 3
Foo3 Showing 3
```

Each value in the sequence is being streamed down the pipeline, one at a time.

Now we'll modify the functions using all the blocks that I've listed just to show you the sequence of events :

```
function foo1() {
    begin {
        Write-Host "Foo1 begin"
    }
    process {
        Write-Host "Foo1 Process $_"
              $_
    }
    end {
        Write-Host "Foo1 end"
    }
}

function foo2() {
    begin {
        Write-Host "Foo2 begin"
    }
    process {
        Write-Host "Foo2 Process $_"
              $_
    }
    end {
        Write-Host "Foo2 end"
    }
}
```

```
function foo3() {
    begin {
        Write-Host "Foo3 begin"
    }
    process {
        Write-Host "Foo3 Process $_"

    }
    end {
        Write-Host "Foo3 end"
    }
}


1,2,3 |  foo1 | foo2 | foo3
```

And the output is :

```
Foo1 begin
Foo2 begin
Foo3 begin
Foo1 Process 1
Foo2 Process 1
Foo3 Process 1
Foo1 Process 2
Foo2 Process 2
Foo3 Process 2
Foo1 Process 3
Foo2 Process 3
Foo3 Process 3
Foo1 end
Foo2 end
Foo3 end
```

At this point I have discovered how I can write functions that use pipelines. I can put this to use immediately in my function, **`Get-MSSQLServerInfo`** that gets information about servers. but as I've already said, my function is still not reusable. How can I change this?

A function should do one thing well.

We can see from our code that I have functionality that can be generalized for a lot of other operations that I will need to perform in SQL Server. The task of making a connection is one of those.. Why not write a function to do this?

```
function Get-MSSQLServerConnection
{
    param([Parameter(Position=0, Mandatory=$true)] [string]$sqlserver)

   $sqlconn = new-object ("Microsoft.SqlServer.Management.Common.ServerConnection") $sqlserver

    $sqlconn.Connect()

    Write-Output $sqlconn

}
```

Then we could change the code in our **`Get-MSSQLServerInfo`** function to use our new **`Get-MSSQLServerConnection`** :

```
function Get-MSSQLServerInfo
{
    param(
    [Parameter(Position=0, Mandatory=$true)] [string]$sqlserver)


    $sqlconn = Get-MSSQLServerConnection $sqlserver
```

```
        $Server=New-Object "Microsoft.SqlServer.Management.Smo.Server" $sqlconn
        $Object = New-Object PSObject -Property @{ServerNname = $server.name
                                                   EngineEdition= $Server.EngineEdition
                                 Product = $Server.Product
                        ProductLevel = $Server.ProductLevel}
        Write-Output $Object
}
```

As you can see, *I* am setting the parameter **$sqlserver** as Mandatory (required) and with Position 0 or "named". (for more information about parameters, type **help about_parameters** at the PowerShell prompt)

Now I have a function that makes a connection to a server, and it can be used in all other functions that require a connection to SQL Server.

I can, of course, get information about several servers :

```
$ServersList = Get-Content c:\posh\Servers.txt
foreach ($svr in  $ServersList ) {
      Get-MSSQLServerInfo $svr
}
```

But why use **foreach** if I can receive the instance or server name by pipeline? You'll probably be thinking that you just need to use **ValueFromPipeline** in the parameters options. (for more information, type **help About_pipeline**)

Let's try if it works.. Type the same server 3 times and pipe to the function : (in my case the name is **7-PC)**

```
"7-pc","7-pc","7-pc" | Get-MSSQLServerInfo
```

Are you expecting 3 rows with the same info correct ? Wrong.. the output is :

```
EngineEdition ProductLevel       ServerNname           Product
------------- ------------       -----------           -------
EnterpriseOrDeveloper RTM        7-pc                  Microsoft SQL Server
```

Just one line..Why? Remember the process block ? That is what is missing . So :

```
function Get-MSSQLServerInfo
{
    param(
    [Parameter(Position=0, Mandatory=$true,ValueFromPipeline = $true)] [string]$sqlserver)

       process {

           $sqlconn = Get-MSSQLServerConnection $sqlserver
              $Server=New-Object "Microsoft.SqlServer.Management.Smo.Server" $sqlconn
              $Object = New-Object PSObject   -Property @{    ServerNname = $server.name
                                                 EngineEdition=$Server.EngineEdition
                                                 Product = $Server.Product
                                                 ProductLevel = $Server.ProductLevel}
              Write-Output $Object
       }
}
```

Finally we have the right output

```
EngineEdition ProductLevel           ServerNname         Product
------------- ------------            -----------        -------
EnterpriseOrDeveloper RTM             7-pc               Microsoft SQL Server
EnterpriseOrDeveloper RTM             7-pc               Microsoft SQL Server
EnterpriseOrDeveloper RTM             7-pc               Microsoft SQL Server
```

You'll be wondering why I am putting the **Get-MSSQLServerConnection** in the process block rather than the Begin block. It is because I need a new connection to each server. In fact, I would just use the begin block to do initialization tasks such as creating variables or loading namespaces.

At this point I'm still not entirely happy with the code. I was trying to do several processes in the one function. For each server, I was making a connection, getting the information, and then outputting it. In PowerShell, it is easier to write functions that do one discrete operation at a time.

Don Jones has several articles about this, and I've provided links at the end of this article.

Now my function is reusable and can now be used in any application that needs to connect to a server via SMO

But something is still missing…..

Hey Function, now behave as a cmdlet

You can do more. If you add the **`Cmdletbinding()`** attribute to your function, it will then have some of the characteristics of cmdlets. We are now well on the road to making an advanced function. You can write functions that can perform operations similarly to a built-in cmdlet. Advanced functions are written in PowerShell rather than by using a Microsoft .NET Framework language.

You can, for example

- use Write-Debug and Write-Verbose in your function and you can control this using –verbose and –debug parameters in your function. This parameters are automatically added by the **`[Cmdletbinding()]`**
- make the common parameters available for your function. To more info type **`help About_commomparameters`**
- implement **`-whatif`** This allows the user of your function to find out what would happen if he executed the function. In other words, it is something like "If I drop all these tables, what would happen?". By typing **`-whatif`** , you can have an idea that what would happen before you did it for real. (I will show how to do this in my next article)

Finally you can see more details of the power of the **`[Cmdletbinding()]`** by typing :

- **`about_Functions_Advanced`**
- **`about_Functions_Advanced_CmdletBindingAttribute`**
- **`about_Functions_Advanced_Methods`**
- **`about_Functions_Advanced_Parameters`**

Our (almost) final code is :

```
function Get-MSSQLServerConnection
{
    param([Parameter(Position=0, Mandatory=$true)] [string]$sqlserver)
      $sqlconn = new-object ("Microsoft.SqlServer.Management.Common.ServerConnection") $sqlserver
    $sqlconn.Connect()
    Write-Output $sqlconn

}

function Get-MSSQLServerInfo
{
       [Cmdletbinding()]
    param(
    [Parameter(Position=0, Mandatory=$true,ValueFromPipeline = $true)] [string]$sqlserver)

       begin {
              [reflection.assembly]::LoadWithPartialName("Microsoft.SqlServer.Smo") | out-null
       }

       process {

           $sqlconn = Get-MSSQLServerConnection $sqlserver
           $Server=New-Object "Microsoft.SqlServer.Management.Smo.Server" $sqlconn
           $Object = New-Object PSObject   -Property @{ServerNname=$server.name
                                                       EngineEdition= $Server.EngineEdition
                                                       Product =$Server.Product
                                                       ProductLevel = $Server.ProductLevel}
           Write-Output $Object
       }
}
```

Handling Errors.

Basic error handling in PowerShell is simple. There is the try/catch/finally block.
For details, see **about_try_catch_finally**) and this link from Steven Murawski - Error Handling

One obvious source of problems in our function could be in connection to SQL Server, possibly if I pass the name incorrectly or the service is out. This would be a serious candidate for placing in an error block.

```
function Get-MSSQLServerInfo
{
       [Cmdletbinding()]
    param(
    [Parameter(Position=0, Mandatory=$true,ValueFromPipeline = $true)] [string]$sqlserver)

       begin {
              [reflection.assembly]::LoadWithPartialName("Microsoft.SqlServer.Smo") | out-null
       }

       process {

              Try {
               $sqlconn = Get-MSSQLServerConnection $sqlserver
               $Server=New-Object "Microsoft.SqlServer.Management.Smo.Server" $sqlconn
$Object = New-Object PSObject -Property @{ServerNname=$server.name
                                                      EngineEdition=$Server.EngineEdition
                                                      Product = $Server.Product
                                                      ProductLevel = $Server.ProductLevel}
               Write-Output $Object
              } Catch {

                     Write-Error $Error[0]

              }
       }
}
```

The Comment-based Help

The first thing I do when I want to know more about a cmdlet is to get the help about it. So if my function needs to act as a cmdlet, it must be a supplier of help via comment-based help.

> *Comment-based Help is written as a series of comments. You can type a comment symbol (#) before each line of comments, or you can use the "<#" and "#>" symbols to create a comment block. All the lines within the comment block are interpreted as comments.*
>
> **about_Comment_Based_Help**

So, with a help comment-block …

```
#######################
<#
.SYNOPSIS
Gets an Information about the SQL Server.
.DESCRIPTION
The Get-MSSQLServerInfo function  gets a collection Custom Object about the SQL Server Server.
.EXAMPLE
get-content servers.txt | Get-MSSQLServerInfo
.EXAMPLE
"Server1","Server2","Server3" | Get-MSSQLServerInfo
.EXAMPLE
Get-MSSQLServerInfo Server1
.LINK
```

```
Get-MSSQLServerInfo
#>
function Get-MSSQLServerInfo …..
```

…our function can provide help just like a cmdlet. If you type **Get-Help** Get-MSSQLServerInfo you will see the magic!.

For a complete list that what sections you can use in your help, type

```
help about_Comment_Based_Help
```

The output

I suspect you will have asked yourself 'Why is Laerte using a custom object (**PSObject)** to output the Server properties? The answer is that I was doing it only for the first example. It is better practice to use the live object in this case, so that you are passing into the pipe all the information (properties, methods) .from the $Server Object, allowing you to then filter out whatever information you actually need for your purpose.

The process block would then be :

```
process {

           $sqlconn = Get-MSSQLServerConnection $sqlserver
           $Server=New-Object "Microsoft.SqlServer.Management.Smo.Server" $sqlconn
           Write-Output $Server

       }
```

And then I could use it simply in a neat pipeline

```
"7-pc" | Get-MSSQLServerInfo | select *
```

Or

```
Get-MSSQLServerInfo "7-pc" | select *
```

However, the custom objects are very useful when you need to combine more than one object in a single output. In this example I need to join some information about the SQL Server and the Configurations (WMI Managed Computer) of the SQL Instance to provide a single object as output.

```
function Get-MSSQLServerInfo
{
       [Cmdletbinding()]
    param(
    [Parameter(Position=0, Mandatory=$true,ValueFromPipeline = $true)] [string]$sqlserver)

       begin {
              [reflection.assembly]::LoadWithPartialName("Microsoft.SqlServer.Smo") | out-null
              [reflection.assembly]::LoadWithPartialName("Microsoft.SqlServer.SqlWmiManagement ") | Out-Null
       }

       process {
              try {

           $sqlconn = Get-MSSQLServerConnection $sqlserver
              $Server=New-Object "Microsoft.SqlServer.Management.Smo.Server" $sqlconn
              $ManagedComputer = New-Object "Microsoft.SqlServer.Management.Smo.Wmi.managedcomputer"
$sqlconn.server
              $Object = New-Object PSObject   -Property
 @{    ServerName = $server.name
       EngineEdition    = $Server.EngineEdition
       Product = $Server.Product
       ProductLevel = $Server.ProductLevel
       ClientProtocols = ($ManagedComputer.ClientProtocols | Select displayname)
       ConnectionSettings = $ManagedComputer.ConnectionSettings.machinename }
```

```
Write-Output $Object
        } catch {
                Write-error $Error[0]
        }
        }
}
```

You can see that, in order to combine the information about the two objects, the **$server** and **$ManagedComputer**, I am using a Custom Object (**$object**)

Displaying Object Status Updates

When your function is performing operations, it helps to have a mechanism that allows you to opt to display information on the screen that reports on progress. You can use to debug the command processing as well. In times past, this was called 'Printfing'. With advanced functions, we can use the **Write-Verbose** cmdlets instead of using **Write-Host.**

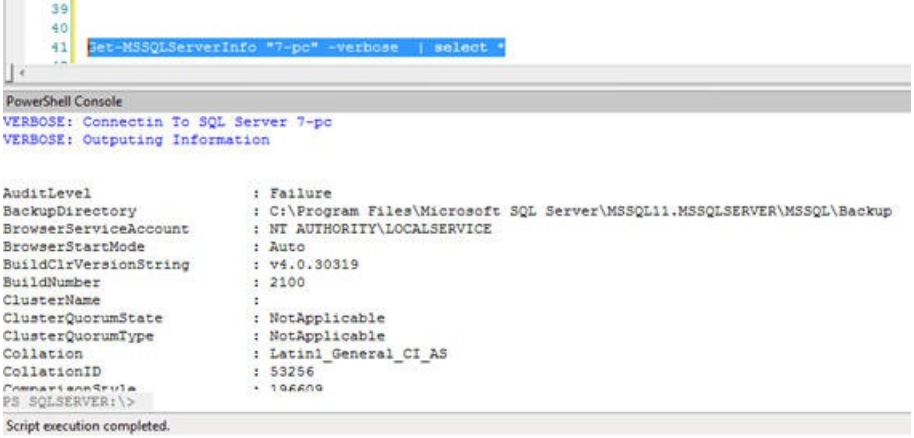
This cmdlet writes text to the verbose message stream, and this can be switched in or out as we wish.

So we can write our code using this feature :

```
function Get-MSSQLServerConnection
{
        param([Parameter(Position=0, Mandatory=$true)] [string]$sqlserver)
        try {
                $sqlconn = new-object ("Microsoft.SqlServer.Management.Common.ServerConnection") $sqlserver
        $sqlconn.Connect()
        Write-Output $sqlconn
        } catch {
                Write-Error $Error[0]
        }

}
function Get-MSSQLServerInfo
{
        [Cmdletbinding()]
    param(
    [Parameter(Position=0, Mandatory=$true,ValueFromPipeline = $true)] [string]$sqlserver)

        begin {
                [reflection.assembly]::LoadWithPartialName("Microsoft.SqlServer.Smo") | out-null
        }

        process {

            $sqlconn = Get-MSSQLServerConnection $sqlserver
               try {
                        Write-Verbose "Connectin To SQL Server $($sqlserver)"
               $Server=New-Object "Microsoft.SqlServer.Management.Smo.Server" $sqlconn
                        write-verbose "Outputing Information"
               Write-Output $Server
               } catch {
                        Write-Error $Error[0]
               }

        }
}
```

Do you remember what I said below the characteristics of an Advanced Function ? One of them is to enable the –verbose common parameter. By default, the message will not be displayed if we not pass the –verbose or change the value of the $VerbosePreference variable. Type about_Preference_Variables to further information.

```
Get-MSSQLServerInfo "7-pc" -verbose  | select *
```



Working with Modules

Once you have your powerful set of functions, you'll want them to behave as if they were built in to your copy of PowerShell. To do this, you can use a script module. The script module is a PowerShell Script file that has a .PSM1 Extension. It is a useful way of sharing your functions and using them by calling them the same way as you would a native cmdlet . When you start a PowerShell Session, your modules will be loaded and you could then, for example, invoke **`Get-MSSQLServerInfo`** by typing directly from the PowerShell command-line.

One of the characteristics of a module are you can choose whether functions are exported so that they would, for example, show up in a get-module cmdlet. You can also load your assemblies just once so it is never necessary to do it within a function. You can also create aliases for your functions.

I can see that the **`Get-MSSQLServerConnection`** function is only useful in order to connect a SQL Server from inside other function, so I don't want it to show up in the list of cmdlets and functions that can be used.

First we create a .psm1 file empty. Then we load the Assemblies.

In our function we use SQL Server SMO assembly. We just add this type :

```
add-type -AssemblyName "Microsoft.SqlServer.Smo"
```

We can now remove the begin block in **`Get-MSSQLServerInfo`** Function because we were only using it to load the assembly.

Now we add our functions :

```
function Get-MSSQLServerConnection
{
	param([Parameter(Position=0, Mandatory=$true)] [string]$sqlserver)
	try {
		$sqlconn = new-object ("Microsoft.SqlServer.Management.Common.ServerConnection") $sqlserver
	$sqlconn.Connect()
	Write-Output $sqlconn
	} catch {
		Write-Error $Error[0]
	}

}

function Get-MSSQLServerInfo
{
	[Cmdletbinding()]
    param(
    [Parameter(Position=0, Mandatory=$true,ValueFromPipeline = $true)] [string]$sqlserver)

	process {

	    $sqlconn = Get-MSSQLServerConnection $sqlserver
		try {
			Write-Verbose "Connectin To SQL Server $($sqlserver)"
		$Server=New-Object "Microsoft.SqlServer.Management.Smo.Server" $sqlconn
			write-verbose "Outputing Information"
		Write-Output $Server
		} catch {
```

```
                Write-Error $Error[0]
            }

        }
}
```

How about creating an alias for **`Get-MSSQLServerInfo`** . We can use gsql instead :

```
Set-Alias -Name gsql -Value Get-MSSQLServerInfo
```

As I said before, I only want **`Get-MSSQLServerInfo`** to be exported as a cmdlet so we add :

```
Export-ModuleMember -Function "Get-MSSQLServerInfo" -Alias gsql
```

As you can see I am exporting the alias as well.

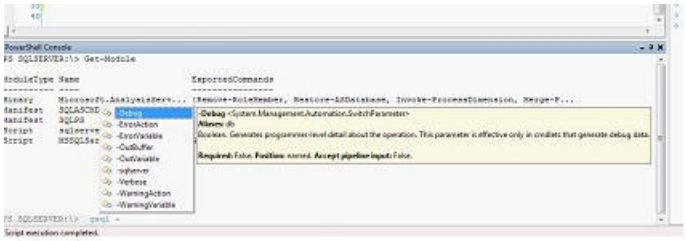
Ufa. Our module is done. The final code is :

```
add-type -AssemblyName "Microsoft.SqlServer.Smo"

function Get-MSSQLServerConnection
{
       param([Parameter(Position=0, Mandatory=$true)] [string]$sqlserver)
       try {
              $sqlconn = new-object ("Microsoft.SqlServer.Management.Common.ServerConnection") $sqlserver
       $sqlconn.Connect()
       Write-Output $sqlconn
       } catch {
              Write-Error $Error[0]
       }

}

function Get-MSSQLServerInfo
{
       [Cmdletbinding()]
    param(
    [Parameter(Position=0, Mandatory=$true,ValueFromPipeline = $true)] [string]$sqlserver)

       process {

           $sqlconn = Get-MSSQLServerConnection $sqlserver
              try {
                     Write-Verbose "Connectin To SQL Server $($sqlserver)"
              $Server=New-Object "Microsoft.SqlServer.Management.Smo.Server" $sqlconn
                     write-verbose "Outputing Information"
              Write-Output $Server
              } catch {
                     Write-Error $Error[0]
              }

       }
}

Set-Alias -Name gsql -Value Get-MSSQLServerInfo
Export-ModuleMember -Function "Get-MSSQLServerInfo" -Alias gsql
```

Now it is only to put in the module path and import into our profile (see Jonathan article) and after that, typing GET-MODULE :

Here it is. Name MSSQLServer. Note that only **`Get-MSSQLServerInfo`** is showed because it was all we exported.

And we can type gsql too rather than the whole name . But let's see something interesting : if I type gsql - , as I am using a Script Editor, all parameters will be shown



Wooowww .. What is this ? My function has only a Server Name parameter. Why is it showing a bunch of parameters ?

Ha! It is an advanced function remember ? **`Cmdletbinding`** ? Now we can use all the common parameters as well.

That is it, guys. I've explained why I believe that you can actually make life easier by introducing a little about reusable functions, how it can help your life too in order to avoid rework and some of the featured of advanced functions into your PowerShell scripts and how to create as a module. I've described how the use of pipes can make scripts run faster and cut out some of the complexity.

In the next article, I will show to you some of the other features of Advanced Functions can help to add more power to your PowerShell scripting studding some examples of CODEPLEX - SQLPSX library (SQL Server PowerShell Extensions) . I hope that you have had as much fun reading this as I've had in writing it. Any comments or ideas are very welcome

Acknowlegements

These are the Resources to this article. Not to PowerShell. I am not listing event 1% of the PowerShell Jedis that I like to read.

Chad Miller – (@cmille19)
Sev17 – SQL Server, PowerShell and so on
Codeplex – SQLPSX (SQL Server PowerShell Extensions)
Sean Kearney (@energizedtech)
Implementing the **`-WHATIF`** into an Advanced Function in Windows PowerShell
Don Jones (@concentrateddon)
Jeffery Hicks (@JeffHicks)
Ravikanth Chaganti (@ravikanth)
Joel Bennet (@Jaykul) A guide to PowerShell's Advanced Functions (Excellent read)

.NET Demon 1.0 Released

Published Thursday, May 03, 2012 1:43 PM

Today we're officially releasing version 1.0 of .NET Demon, the Visual Studio Extension Alex Davies and I have been working on for the last 6 months. There have been beta versions available for a while, but we have now released the first "official" version and made it available to purchase. If you haven't yet tried the tool, it's all about reducing the time between when you write a line of code and when you are able to try it out so you don't have to wait:

- **Continuous compilation**
  We use spare CPU cycles on your machine to compile your code in the background when you make changes, so assemblies are up to date whenever you want to run them. Some clever logic means we only recompile code which may have been affected by your changes.
- **Continuous save**
  .NET Demon can perform background saving, so you don't lose any work in case of crashes or power failures, and are less likely to forget to commit changed files.
- **Continuous testing (Experimental)**
  The testing tool in .NET Demon watches which code you change in your solution, and automatically reruns tests which are impacted, so you learn about any breaking changes as quickly as possible. It also gives you inline test coverage information inside Visual Studio. Continuous testing is still experimental - it will work fine in many cases, but we know it's not yet perfect.

Releasing version 1.0 doesn't mean we're pausing development or pushing out improvements. We will still be regularly providing new versions with improved functionality and fixes for any bugs people come across.

Visit the .NET Demon product page to download

by theo.spears

SQL VIEW Basics

10 May 2012
by Joe Celko

SQL Views are essential for the database developer. However, it is common to see them misued, or neglected. Joe Celko tackles an introduction to the subject, but there is something about the topic that makes it likely that even the experienced developer will find out something new from reading it.

A **VIEW** is a virtual table, defined by a query, that does not exist until it is invoked by name in an SQL statement. This may sound simple enough, but some developers have difficulties with the concept. Because of this, they tend to leave out **VIEW**s because they do not appreciate their value. It is easy to write a query and not bother to put it into a **VIEW**, because there is no performance boost. 'If I am going to save code,' they reason, 'I want it in a stored procedure that can take a parameter list instead.'

In fact, a **VIEW** definition can be copied as in-line text, just like a CTE. But with a good optimizer, the SQL engine can decide that enough sessions are using the same **VIEW** and materialize it as a shared table. The CTE, in contrast, is strictly local to the statement in which it is declared.

VIEWs are often named incorrectly. A **VIEW** is a table, so it is named just like any other table. The name tells us what set of things it represents in the data model. The most common offender is the “Volkswagen” coder who prefixes or suffixes the **VIEW** name with “vw_” or “**VIEW**_” in violation of ISO-11179 rules. We do not mix data and meta data in a name. This is as silly as prefixing every noun in a novel with “n_” so the reader will know that the word is a noun in English grammar.

The ANSI Standard SQL syntax for the **VIEW** definition is

```
CREATE VIEW <table name> [(<VIEW column list>)]
AS <query expression>
[WITH [<levels clause>] CHECK OPTION]
<levels clause> ::= CASCADED | LOCAL
```

the **WITH CHECK OPTION** is a little known and less used feature that has been around since the SQL-86 Standards. The **<levels clause>** option in the **WITH CHECK OPTION** did not exist in Standards before SQL-92. It is not implemented in T-SQL dialect, so I will skip it. If you see it, just remember T-SQL defaults to **CASCADED** behavior.

A **VIEW** has no physical existence in the database until it is invoked. You cannot put constraints on a **VIEW** for that reason. The name of the **VIEW** must be unique within the entire database schema, like a base table name. The **VIEW** definition cannot reference itself, since it does not exist yet. Nor can the definition reference only other **VIEW**s; the nesting of **VIEW**s must eventually resolve to underlying base tables. This only makes sense; if no base tables were involved, what would you be **VIEW**ing?

You can either build a column name list in the **VIEW** header or inherit the column names from the **SELECT** statement. Building this list is usually just one quick “cut & paste” and well worth it. This is why we do not ever use "**SELECT** *" in a **VIEW** definition in production code. When the columns of a base tables change, the definition of the "star" will also change. If you are lucky, you will get an error when the **VIEW** has too many or too few columns when it is invoked. If you are not so lucky, the **VIEW** will run and give you unexpected results. If you are unlucky, the **VIEW** will run and give you wrong answers that you use.

Every few months, someone will post to a SQL forum asking how to use a parameter in a **VIEW**. They would never ask how to use a parameter in a base table. The sight of a **SELECT** statement instead of a list of column declarations throws their mindset in the wrong direction.

Mullins Heuristics for VIEWS

Programmers have rules and standards for creating base tables. The data element names should follow the ISO-11179 rules. We have to have a key. We can have all kinds of constraints. We can have Declarative Referential Integrity actions among other base tables. But how do you design a **VIEW**?

Craig Mullins, a DB2 expert and author, gave the following rule to ensure that **VIEW**s are created in a responsible and useful manner. Simply stated, the **VIEW** creation strategy should be goal-oriented. **VIEW**s should be created only when they achieve a specific, reasonable goal. Each **VIEW** should have a specific application or business requirement that it fulfills before it is created. That requirement should be documented somewhere, preferably in a data dictionary.

Although this rule seems obvious, **VIEW**s are implemented at some shops without much thought as to how they will be used. This can cause the number of **VIEW**s that must be supported and maintained to increase until so many **VIEW**s exist that it is impossible to categorize their uses. Nobody wants to take a chance and drop a **VIEW**, so they just write a new one. Whenever a base table used by a **VIEW** definition is changed, then all those **VIEW**s have to be re-compiled and checked. Since **VIEW**s can be built on top of **VIEW**s, this can be tricky.

Unlike other virtual tables, a **VIEW** is defined in the schema information tables and its definition (not its content!) is persisted. This implies some privileges are needed to use, create, alter and drop **VIEW**s. The first question is do you need to have privileges on the base tables that build a **VIEW**? Yes, but not full privileges. The minimal privileges would be to use the base tables, so you can build the **VIEW**. But that does not mean that the user needs to be able to directly query or modify the base tables.

The ideal design should give each user a set of **VIEW**s that make it look as if the schema was designed for just his or her use, without regard to the rest of the enterprise.

This is most often done for security and privacy. The payroll clerk can see the salaries of other personnel and change them. But he cannot give himself a pay raise and try to get out of town before the police find out. He can see the minimum, maximum and average salary in each department, but not who is making which salary.

The Data Control Language (DCL) is the third sub-language in SQL after DDL and DML. This is where the DBA can **GRANT**, **REVOKE** or **DENY** all kinds of schema object privileges. We spend almost no time on it in training classes, and failure to do it right can destroy your enterprise. As a generalization, the DBA ought to start with a list of roles users can play in the enterprise and create a script for the privileges each role needs. A new user can then be assigned a role and you do not have to repeat the script over and over.

Do not grant **VIEW** creation privileges to everyone. The "nearly the same" **VIEW**s are a special problem. One user might have read the spec "Employees must be over 21 years of age to serve alcohol" to mean strictly over 21 as of today or can they pour a drink on their 21-st birthday? If **VIEW** creation had been left to just one data modeler, only one of these **VIEW**s would exist and it would have the correct business rule.

Tricky Queries and Computations

Not all programers are equal, so you can make sure that the **VIEW**s preserve the best work in your shop. The other advantage is that if someone finds a better query for the current state of the database, you keep the **VIEW** header, drop the **SELECT** statement in the body, replace it and then re-compile your code. The programmer needs no knowledge of how the **VIEW** works. This technique becomes more useful as the SQL becomes more complex.

In T-SQL, we used to write complicated code to get sequence numbers and pure dates without time. This code was often hidden in **VIEW**s. The numbering can now be done with ROW_NUMBER() and we have a DATE data type since SQL Server 2008. In many cases, procedures and functions that used loops and fancy string manipulations can be replaced with **VIEW**s.

Updatable and Read-Only **VIEW**s

Unlike base tables, **VIEW**s are either updatable or read-only, but not both. **INSERT**, **UPDATE**, and **DELETE** operations are allowed on updatable **VIEW**s and base tables, subject to other constraints. **INSERT**, **UPDATE**, and **DELETE** are not allowed on read-only **VIEW**s, but you can change their base tables, as you would expect.

An updatable **VIEW** is one that can have each of its rows associated with exactly one row in an underlying base table. When the **VIEW** is changed, the changes pass through the **VIEW** to that underlying base table unambiguously. Updatable **VIEW**s in Standard SQL are defined only for queries that meet these criteria

1. They are built on only one table
2. No **GROUP BY** clause
3. No **HAVING** clause
4. No aggregate functions
5. No calculated columns
6. No **UNION**, **INTERSECT** or **EXCEPT**
7. No **SELECT DISTINCT** clause
8. Any columns excluded from the **VIEW** must be **NULL**-able or have a **DEFAULT** clause in the base table, so that a whole row can be constructed for insertion.

By implication, the **VIEW** must also contain a key of the table. In short, we are absolutely sure that each row in the **VIEW** maps back to one and only one row in the base table. The major advantage of this limited definition is that it is based on syntax and not semantics. For example, these **VIEW**s are logically identical:

```
CREATE VIEW Foo1 (a, b, ..) -- updatable, has a key!
AS SELECT (a, b, ..)
   FROM Foobar
  WHERE x IN (1,2);
```

```
CREATE VIEW Foo2 (a, b, ..)-- not updateable!
AS SELECT (a, b, ..)
```

```
   FROM Foobar
  WHERE x = 1
  UNION ALL
  SELECT (a, b, ..)
   FROM Foobar
  WHERE x = 2;
```

But Foo1 is updateable and Foo2 is not. While I know of no formal proof, I suspect that determining if a complex query resolves to an updatable query for allowed sets of data values possible in the table is an NP-complete problem.

The **INSTEAD OF** trigger was the ANSI Standards Committee letting the Data Modeler decide on how to resolve the **VIEW** updating problem. These triggers are added to a **VIEW** and are executed on base tables that make up the **VIEW**. The user never sees them fire and work their magic.

As an example, consider a **VIEW** that builds the total compensation for each employee by joining the personnel, employee stock holdings, bonuses and **salary_amt** tables in one **VIEW**. An **INSTEAD OF** trigger can update the total compensation using a hidden formula and complex business rules that the user never sees.

The use of **INSTEAD OF** triggers gives the user the effect of a single table, but there can still be surprises. Think about three tables; A, B and C. Table C is disjoint from the other two. Tables A and B overlap. So I can always insert into C and may or may not be able to insert into A and B if I hit overlapping rows.

Going back to my Y2K consulting days, I ran into a version of such a partition by calendar periods. Their Table C was set up on Fiscal quarters and got leap year wrong because one of the fiscal quarters ended on the last day of February.

Nested VIEWs

A point that is often missed, even by experienced SQL programmers, is that a **VIEW** can be built on other **VIEW**s. The only restrictions are that circular references within the query expressions of the **VIEW**s are illegal and that a **VIEW** must ultimately be built on base tables. One problem with nested **VIEW**s is that different updatable **VIEW**s can reference the same base table at the same time. If these **VIEW**s then appear in another **VIEW**, it becomes hard to determine what has happened when the highest-level **VIEW** is changed. As an example, consider a table with two keys:

```
CREATE TABLE CanadianDictionary
(english_id INTEGER UNIQUE,
 french_id INTEGER UNIQUE,
 eng_word CHAR(30),
 french_word CHAR(30),
 CHECK (COALESCE (english_id, french_id) IS NOT NULL);
```

The table declaration is a bit strange. It allows an English-only or French-only word to appear in the table. But the **CHECK()** constraint requires that a word must fall into one or both type codes.

```
INSERT INTO CanadianDictionary
VALUES (1, 2, 'muffins', 'croissants'),
    (2, 1, 'fish bait', 'escargots');
CREATE VIEW EnglishWords
AS SELECT english_id, eng_word
   FROM CanadianDictionary
  WHERE eng_word IS NOT NULL;
CREATE VIEW FrenchWords
AS SELECT french_id, french_word
   FROM CanadianDictionary
  WHERE french_word IS NOT NULL);
```

We have now tried the escargots and decided that we wish to change our opinion of them:

```
UPDATE EnglishWords
  SET eng_word = 'appetizer'
 WHERE english_id = 2;
```

Our French user has just tried Haggis and decided to insert a new row for his experience:

```
UPDATE FrenchWords
  SET french_word = 'tripoux'
 WHERE french_id = 3;
```

The row that is created is **(NULL, 3, NULL, 'tripoux')**, since there is no way for the **VIEW FrenchWords** to get to the **VIEW EnglishWords** columns. Likewise, the English **VIEW** user can construct a row to insert his translation, **(3, NULL, 'Haggis', NULL)**, but neither of them can consolidate the two rows into a meaningful piece of data.

To delete a row is also to destroy data; the French-speaker who drops 'croissants' from the table also drops 'muffins' from **VIEW EnglishWords.**

WITH CHECK OPTION Clause

If **WITH CHECK OPTION** is specified, the **VIEW**ed table has to be updatable. This is actually a fast way to check how your particular SQL implementation handles updatable **VIEW**s. Try to create a version of the **VIEW** in question using the **WITH CHECK OPTION** and see if your product will allow you to create it. The **WITH CHECK OPTION** was part of the SQL-89 Standard, but nobody seems to know about it! Consider this skeleton:

```
CREATE VIEW V1
AS SELECT key_col, col1, col2
   FROM Foobar
  WHERE col1 = 'A';
```

and now **UPDATE** it with

```
UPDATE V1 SET col1 = 'B';
```

The **UPDATE** will take place without any trouble, but the rows that were previously seen now disappear when we use V1 again. They no longer meet the **WHERE** clause condition! Likewise, an **INSERT INTO** statement with **VALUES (col1 = 'B')** would insert just fine, but its rows would never be seen again in this **VIEW**. This might be the desired behavior. For example, you can set up a **VIEW** of rows in a jobs table with a status code of 'to be done', work on them, and change a status code to 'finished', and the rows will disappear from your **VIEW**. The important point is that the **WHERE** clause condition was checked only at the time when the **VIEW** was invoked.

The **WITH CHECK OPTION** makes the system check the **WHERE** clause condition upon **INSERT** and **UPDATE**. If the new or changed row fails the test, the change is rejected and the **VIEW** remains the same. The **WITH CHECK OPTION** clause does not work like a CHECK constraint.

```
CREATE TABLE Foobar (col_a INTEGER);
CREATE VIEW TestView (col_a)
AS
SELECT col_a FROM Foobar WHERE col_a > 0
WITH CHECK OPTION;
INSERT INTO TestView VALUES (NULL); -- This fails!
CREATE TABLE Foobar_2 (col_a INTEGER CHECK (col_a > 0));
INSERT INTO Foobar_2(col_a)
VALUES (NULL); -- This succeeds!
```

The **WITH CHECK OPTION** must be **TRUE** while the CHECK constraint can be either **TRUE** or **UNKNOWN**. This is an example of the differences in DDL and DML in SQL. Once more, you need to watch out for **NULL**s.

T-SQL checks all the underlying levels that built the **VIEW**, as well as the **WHERE** clause condition in the **VIEW** itself. If anything causes a row to disappear from the **VIEW**, the **UPDATE** is rejected. Consider two **VIEW**s built on each other from the Personnel table:

```
CREATE VIEW Low_Paid_Personnel (emp_id, salary_amt)
AS SELECT emp_id, salary_amt
   FROM Personnel
  WHERE salary_amt <= 250.00;
CREATE VIEW Medium_Paid_Personnel (emp_id, salary_amt)
AS SELECT emp_id, salary_amt
   FROM Low_Paid_Personnel
  WHERE salary_amt >= 100.00;
```

If neither **VIEW** has a **WITH CHECK OPTION**, the effect of updating **Medium_Paid_Personnel** by increasing every **salary_amt** by $1,000 will be passed without any check to **Low_Paid_Personnel** . **Low_Paid_Personnel** will pass the changes to the underlying Personnel table. The next time **Medium_Paid_Personnel** is used, **Low_Paid_Personnel** will be rebuilt in its own right and **Medium_Paid_Personnel** rebuilt from it, and all the employees will disappear from **Medium_Paid_Personnel** .

If only **Medium_Paid_Personnel** has a **WITH CHECK OPTION** on it, the **UPDATE** will fail. **Medium_Paid_Personnel** has no problem with such a large **salary_amt**, but it would cause a row in **Low_Paid_Personnel** to disappear, so **Medium_Paid_Personnel** will reject it. However, if only **Medium_Paid_Personnel** has a **WITH LOCAL CHECK OPTION** on it, the **UPDATE** will succeed. **Medium_Paid_Personnel** has no problem with such a large **salary_amt**, so it passes the change along to **Low_Paid_Personnel** . **Low_Paid_Personnel** , in turn, passes the change to the Personnel table and the **UPDATE** occurs. If both **VIEW**s have a **WITH CHECK OPTION**, the effect is a set of conditions, all of which have to be met. The Personnel table can accept **UPDATE**s or **INSERT**s only where the **salary_amt** is between $100 and $250.

WITH CHECK OPTION as Constraints

Lothar Flatz, an instructor for Oracle Software Switzerland made the observation that while Oracle cannot put subqueries into **CHECK()**() constraints and triggers would not be possible because of the mutating table problem, you can use a **VIEW** that has a **WITH CHECK OPTION** to enforce subquery constraints.

For example, consider a hotel registry that needs to have a rule that you cannot add a guest to a room that another is or will be occupying. Instead of writing the constraint directly, like this:

```
CREATE TABLE Hotel
(room_nbr INTEGER NOT NULL,
 arrival_date DATE NOT NULL,
 departure_date DATE NOT NULL,
 guest_name CHAR(30) NOT NULL,
 CONSTRAINT valid_stay_dates
 CHECK (H1.arrival_date <= H1.departure_date),
 CONSTRAINT no_overlaps
 CHECK (NOT EXISTS
    (SELECT *
      FROM Hotel AS H1, Hotel AS H2
     WHERE H1.room_nbr = H2.room_nbr
      AND H2.arrival_date < H1.arrival_date
      AND H1.arrival_date < H2.departure_date)));
```

The **valid_stay_dates** constraint is fine, since it has no subquery, but will choke on the **no_overlaps** constraint. Leaving the **no_overlaps** constraint off the table, we can construct a **VIEW** on all the rows and columns of the Hotel base table and add a **WHERE** clause which will be enforced by the **WITH CHECK OPTION**.

```
CREATE VIEW Valid_Hotel (room_nbr, arrival_date, departure_date, guest_name)
AS SELECT H1.room_nbr, H1.arrival_date, H1.departure_date, H1.guest_name
   FROM Hotel AS H1
  WHERE NOT EXISTS
     (SELECT *
       FROM Hotel AS H2
      WHERE H1.room_nbr = H2.room_nbr
       AND H2.arrival_date < H1.arrival_date
       AND H1.arrival_date < H2.departure_date)
   AND H1.arrival_date <= H1.departure_date
  WITH CHECK OPTION;
```

For example,

```
INSERT INTO Valid_Hotel
VALUES (1, '2012-06-01', '2012-06-03', 'Ron Coe');
GO
INSERT INTO Valid_Hotel
VALUES (1, '2012-06-03', '2012-06-05', 'John Doe');
```

will give a **WITH CHECK OPTION** clause violation on the second **INSERT INTO** statement, as we wanted.

Dropping VIEWs

VIEWs, like tables, can be dropped from the schema. The T-SQL syntax for the statement is:

```
DROP VIEW <table name list>;
```

The use of the <table name list> is dialect and it gives you a shorthand for repeating drop statements. The drop behavior depends on your vendor. The usual way of storing **VIEW**s was in a schema information table is to keep the **VIEW** name, the text of the **VIEW**, but dependencies. When you drop a **VIEW**, the engine usually removes the appropriate row from the schema information tables. You find out about dependencies when you try to use something that wants the dropped **VIEW**s. Dropping a base table could cause the same problem when the **VIEW** was accessed. But the primary key/foreign key dependencies among base tables will prevent dropping some base tables.

Table Expression **VIEW**s

An old usage for **VIEW**s was to do the work of CTEs when there were no CTEs. The programmers created **VIEW**s, and then used them. Of course they wasted space, caused disk reads and were only used in one statement. It might be worth looking at old code for **VIEW**s that are not shared.

Today the reverse is true. Programmers create the same CTE code over and over in different queries and give it local names for each appearance. Those local names are seldom the same and are often “place markers” like “X” or “CTE_1” and give no hint as to what the table expression means in the data model.

It can be hard to factor out common table expressions across multiple queries. One query uses an infixed JOIN operators and another uses a FROM list, the predicates are equivalent but written slightly different and so forth.

I recommend that you sit down and think of useful **VIEW**s, write them and then see if you can find places where they would make the code easier to read and maintain. As an example, our hotel application will probably need to find vacant rooms by calendar date, compute an occupancy ratio by calendar date and other basic facts.

Another bad use is the one **VIEW** per Base Table myth that was poplar with DB2 programmers years ago. The reasoning behind this myth was the applaudable desire to insulate application programs from database changes. All programs were to be written against **VIEW**s instead of base tables. When a change is made to the base table, the programs would not need to be modified because they access a **VIEW**, not the base table.

This does not work in the long run. All you do is accumulate weird orphaned **VIEW**s. Consider the simplest type of database change – adding a column to a table. If you do not add the column to the **VIEW**, no programs can access that column unless another **VIEW** is created that contains the new column. But if you create a new **VIEW** every time you add a new column it will not take long for your schema to be swamped with **VIEW**s. Even more troublesome is the question of which **VIEW** should be used by which program. Similar arguments can be made for any structural change to the tables.

Subterranean IL: The ThreadLocal type

Published Thursday, May 03, 2012 12:11 PM

I came across `ThreadLocal<T>` while I was researching `ConcurrentBag`. To look at it, it doesn't really make much sense. What's all those extra `Cn` classes doing in there? Why is there a `GenericHolder<T,U,V,W>` class? What's going on? However, digging deeper, it's a rather ingenious solution to a tricky problem.

Thread statics

Declaring that a variable is thread static, that is, values assigned and read from the field is specific to the thread doing the reading, is quite easy in .NET:

```
[ThreadStatic]
private static string s_ThreadStaticField;
```

`ThreadStaticAttribute` is not a pseudo-custom attribute; it is compiled as a normal attribute, but the CLR has in-built magic, activated by that attribute, to redirect accesses to the field based on the executing thread's identity.

`TheadStaticAttribute` provides a simple solution when you want to use a single field as thread-static. What if you want to create an arbitary number of thread static variables at runtime? Thread-static fields can only be declared, and are fixed, at compile time. Prior to .NET 4, you only had one solution - thread local data slots. This is a lesser-known function of `Thread` that has existed since .NET 1.1:

```
LocalDataStoreSlot threadSlot = Thread.AllocateNamedDataSlot("slot1");

string value = "foo";
Thread.SetData(threadSlot, value);

string gettedValue = (string)Thread.GetData(threadSlot);
```

Each instance of `LocalStoreDataSlot` mediates access to a single slot, and each slot acts like a separate thread-static field.

As you can see, using thread data slots is quite cumbersome. You need to keep track of `LocalDataStoreSlot` objects, it's not obvious how instances of `LocalDataStoreSlot` correspond to individual thread-static variables, and it's not type safe. It's also relatively slow and complicated; the internal implementation consists of a whole series of classes hanging off a single thread-static field in `Thread` itself, using various arrays, lists, and locks for synchronization. `ThreadLocal<T>` is far simpler and easier to use.

ThreadLocal

`ThreadLocal` provides an abstraction around thread-static fields that allows it to be used just like any other class; it can be used as a replacement for a thread-static field, it can be used in a `List<ThreadLocal<T>>`, you can create as many as you need at runtime. So what does it do? It can't just have an instance-specific thread-static field, because thread-static fields have to be declared as `static`, and so shared between all instances of the declaring type. There's something else going on here.

The values stored in instances of `ThreadLocal<T>` are stored in instantiations of the `GenericHolder<T,U,V,W>` class, which contains a single `ThreadStatic` field (`s_value`) to store the actual value. This class is then instantiated with various combinations of the `Cn` types for generic arguments.

In .NET, each separate instantiation of a generic type has its own static state. For example, `GenericHolder<int,C0,C1,C2>` has a completely separate `s_value` field to `GenericHolder<int,C1,C14,C1>`. This feature is (ab)used by `ThreadLocal` to emulate instance thread-static fields.

Every time an instance of `ThreadLocal` is constructed, it is assigned a unique number from the static `s_currentTypeId` field using `Interlocked.Increment`, in the `FindNextTypeIndex` method. The hexadecimal representation of that number then defines the specific `Cn` types that instantiates the `GenericHolder` class. That instantiation is therefore 'owned' by that instance of `ThreadLocal`.

This gives each instance of `ThreadLocal` its own `ThreadStatic` field through a specific unique instantiation of the `GenericHolder` class. Although `GenericHolder` has four type variables, the first one is always instantiated to the type stored in the `ThreadLocal<T>`. This gives three free type variables, each of which can be instantiated to one of 16 types (`C0` to `C15`). This puts an upper limit of 4096 (16^3) on the number of `ThreadLocal<T>` instances that can be created for each value of T. That is, there can be a maximum of 4096 instances of `ThreadLocal<string>`, and separately a maximum of 4096 instances of `ThreadLocal<object>`, etc.

However, there is an upper limit of 16384 enforced on the total number of `ThreadLocal` instances in the AppDomain. This is to stop too much memory being used by thousands of instantiations of `GenericHolder<T,U,V,W>`, as once a type is loaded into an AppDomain it cannot be unloaded, and will continue to sit there taking up memory until the AppDomain is unloaded. The total number of `ThreadLocal` instances created is

tracked by the `ThreadLocalGlobalCounter` class.

So what happens when either limit is reached? Firstly, to try and stop this limit being reached, it recycles `GenericHolder` type indexes of `ThreadLocal` instances that get disposed using the `s_availableIndices` concurrent stack. This allows `GenericHolder` instantiations of disposed `ThreadLocal` instances to be re-used. But if there aren't any available instantiations, then `ThreadLocal` falls back on a standard thread local slot using `TLSHolder`. This makes it very important to dispose of your `ThreadLocal` instances if you'll be using lots of them, so the type instantiations can be recycled.

The previous way of creating arbitary thread-static variables, thread data slots, was slow, clunky, and hard to use. In comparison, `ThreadLocal` can be used just like any other type, and each instance appears from the outside to be a non-static thread-static variable. It does this by using the CLR type system to assign each instance of `ThreadLocal` its own instantiated type containing a thread-static field, and so delegating a lot of the bookkeeping that thread data slots had to do to the CLR type system itself! That's a very clever use of the CLR type system.

by Simon Cooper
Filed Under: Subterranean IL

SQL Scripts Manager and IronPython

04 May 2012
by Timothy Wiseman

> As well as running TSQL and PowerShell, the free SQL Scripts Manager tool can also run Python scripts. This allows even more sophisticated scripting possibilities for managing automated database tasks.

Last year, Red Gate released the free tool SQL Scripts Manager (SSM) to provide a convenient way to organize scripts for SQL Server. Most scripting that DBAs do require a small number of forms, such as keying in credentials, and these are already-made within SSM and therefore consistent between scripts. SSM makes it easy to create a graphical front end to T-SQL Scripts, but it will also support PowerShell and IronPython scripts.

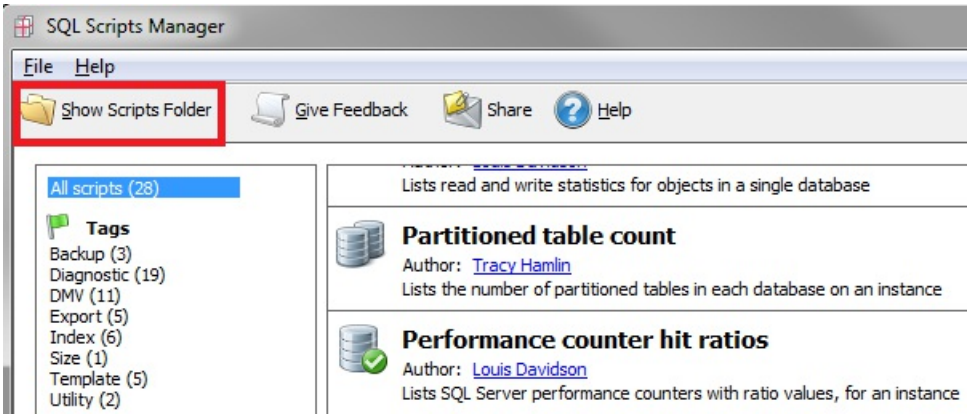
Why use IronPython with SSM?

IronPython is a .Net implementation of Python, which means it comes with Python's ease of readability and ease of learning, but adds the advantages of access to the .Net libraries, including the SQL Management Objects (SMO). By using IronPython with SSM, a DBA can easily create and organize a series of administrative scripts with a simple, consistent user interface that will automate many routine tasks. IronPython has an amazing range of libraries and modules, and so it is sometimes the quickest way of getting some of the more unusual functionality into a script.

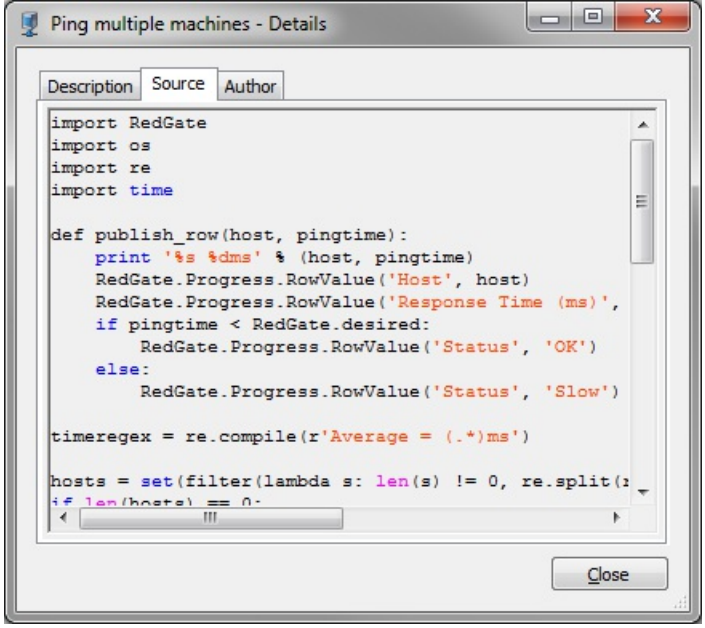
Creating an IronPython Script for SSM

In order to integrate with SSM, a script must be wrapped in XML which provides extra information to be displayed to the user, as well as a description of the interface that SSM will provide. As with many things, the best way to get a feeling for the XML format is to look at examples.

SSM comes with a library of scripts written by experts that are useful by themselves, but also provide examples of what can be done with SSM and how it can be done. Some of the scripts that come with the package are written in IronPython and provide a good template for new utilities. For instance, the entire Export series by Phil Factor is written in IronPython, as is the Ping Multiple Machines script. To examine the complete source code for one of these scripts, select the "Show Scripts Folder" Icon in SSM and then open the corresponding .rgtools file in a text editor.



There is an option within SSM to view the details of a script and the details include the source code. This provides an opportunity to review exactly what a script would do before executing it. But it is not the best place to find templates for creating new scripts since it only shows the code that will be executed and strips out the XML.



The complete reference for the XML Schema used by SSM is detailed in The SQL Scripts Manager XML schema. Along with providing metadata about the script, the XML template describes the user interface and provides a variety of controls that allow the user to provide inputs for the script. The script itself can interact with the controls through the RedGate.control_identifier after importing the RedGate module. Once the XML file, including the IronPython script, is completed it can be added to SSM just by saving it in the scripts folder (which can be set in File – Application Options) with the *.rgtools* extensions.

Example

As an example, here is a simple script that will identify Orphan Users with SMO:

```
<?xml version="1.0" encoding="utf-8"?>
<tool version="1">
  <description name="Find Orphan Users" icon="data_next" version="1"><![CDATA[
<p><b>Provides a list of all orphan users.</b></p>
<p>Uses IronPython and SMO to provide a list of all databases on an instance
and any users which do not have a login associated with them but have access
to the DB.</p>
]]></description>
  <author name="Tim Wiseman"
          url="http://www.simple-talk.com/author/timothy-wiseman/" img="" />
  <tags>
          <tag>Diagnostic</tag>
          <tag>users</tag>
  </tags>

  <ui>
          <output displaytype=" Message | ForceLog" />
          <block>
            <control type="header" label="Server">
                     <font name="Arial" size="12" />
            </control>
            <control type="server" id="Connection" />
          </block>
  </ui>


  <script type="python"><![CDATA[
import RedGate #allows interaction with the UI

import sys #to add to the Python Path to find the assemblies
import clr #To bring in .NET libraries

#May or may not be needed depending on configuration, adjust path as needed
sys.path.insert(0,
    r"C:\Program Files (x86)\Microsoft SQL Server\100\SDK\Assemblies")

#bring in the SMO assemblies and import
```

```
clr.AddReferenceToFile('Microsoft.SqlServer.Smo.dll')
clr.AddReferenceToFile('Microsoft.SqlServer.SqlEnum.dll')
clr.AddReferenceToFile('Microsoft.SqlServer.ConnectionInfo.dll')
import Microsoft.SqlServer.Management.Smo as SMO
import Microsoft.SqlServer.Management.Common as Common

#connect with SMO, one of several methods
if RedGate.Connection.Credentials.IntegratedSecurity:
            sqlServer = SMO.Server(RedGate.Connection.DataSource)
else:
            srvConn = Common.ServerConnection(RedGate.Connection.DataSource)
            srvConn.LoginSecure = 0
            srvConn.Login = RedGate.Connection.Credentials.UserID
            srvConn.Password = RedGate.Connection.Credentials.Password
            sqlServer = SMO.Server(srvConn)

for db in sqlServer.Databases:
            RedGate.Progress.Message = 'DB: {0}'.format(db.Name)
            for user in db.Users:
                        #Orphan user if it is not a system account (like guest)
                        #and it has no Login
                        if user.Login == '' and user.IsSystemObject == False:
                                    RedGate.Progress.Message = '\t{0}'.format(user.Name)
            RedGate.Progress.Message = ' ' #Insert a blank line between DBs

RedGate.Progress.Success = True

            ]]></script>
  <signature></signature>
</tool>
```

When this is saved in the scripts folder with a *.rgtool* extension, it will automatically appear in the SSM menu under the title, tags, and author given in the XML document itself.

It shows up in red to show that there is no valid signature in the XML document, which also makes it stand out from the Red Gate provided scripts, which can be convenient at times.

Some Technical Details

Generally, Python is known for its readability and the XML formatting used by SSM is generally self-explanatory, but there are a few things worth noting about the interactions.

SSM comes with its own IronPython interpreter, currently at version 2.6. This means that it can run IronPython scripts whether the user has another IronPython interpreter or not. But it also means that a script for SSM cannot use the new features of later versions of IronPython. The SSM interpreter also maintains its own path. So, if a script needs to import libraries not included in the SSM IronPython path, then either those libraries need to be copied into the SSM folder or the folder they are in needs to be explicitly added to sys.path.

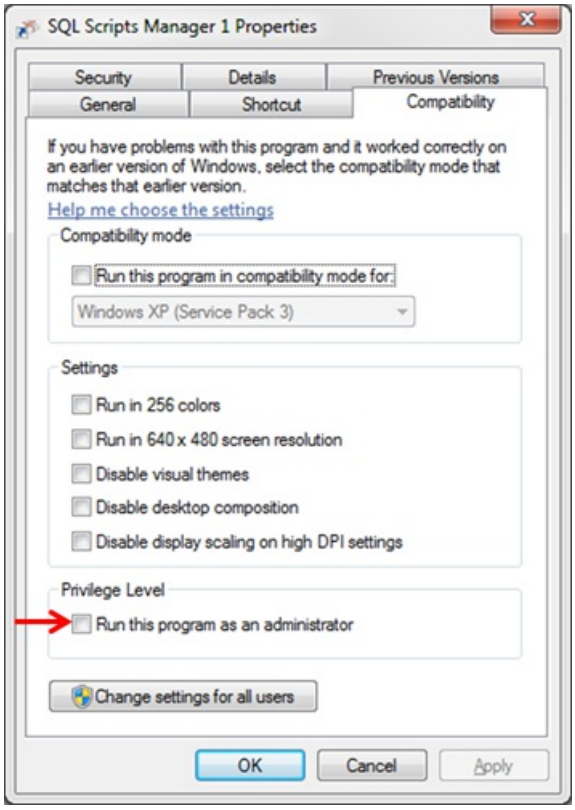
In some cases, it may be necessary to specifically insert a new folder at the beginning of sys.path instead of appending it to the end. For instance, if there is more than one version of SQL Server installed, the SSM IronPython may put all of their assembly directories in the default path. The script can be forced to use a specific version of the assembly by inserting the folder with the right version at the beginning of the list, like:

```
sys.path.insert(0, r"C:\Program Files (x86)\Microsoft SQL Server\100\SDK\Assemblies")
```

Also, the signature block at the bottom is there to authenticate scripts and there is currently no way for the end user to sign their own scripts. Scripts with a blank signature block will appear red in the SSM interface, but that will not affect their use in any other way.

SSM generally does not run with elevated operating system permissions unless specifically opened with elevated permissions. It can still get

elevated permissions inside of SQL Server, so most scripts which interact primarily with SQL Server will be unaffected with this. However, this may cause issues for scripts which make calls to the operating system or reach out to other programs. One way to ensure that SSM has full permissions is to change the compatibility settings it runs with:



Naturally, this does permit scripts run through SSM to cause more problems than they would otherwise if they are malicious or poorly written, so this should be used with care.

Using SSM with other tools

IronPython is well adapted to interacting with other programs which expose a command line interface. For instance, Red Gate's SQL Backup Pro can be run from the command line on the server. A simple SSM script to make a backup with SQL Backup would look like:

```
<?xml version="1.0" encoding="utf-8"?>
<tool version="1">
<description name="Sql Backup Pro Backup" icon="data_floppy_disk" version="1">
<![CDATA[
<p>
<b>Sql Backup Pro Database Backup</b>
</p>
<p>Uses the CLI for Red Gate Sql Backup Pro to take a database backup.</p>
<p>Must be run on the server where SQL Backup Pro is installed.</p>
<p>Using SQLBackupC this way requires elevated permissions and this
script will fail if SQL Scripts Manager is run without the "Run as
Administrator" option.</p>

]]></description>

<author name="Tim Wiseman"
url="http://www.simple-talk.com/author/timothy-wiseman/" img="" />

<tags>
    <tag>Backup</tag>
</tags>
<ui>
    <output displaytype="Message|ForceLog" />
    <block>
        <control type="header" label="Database">
            <font name="Arial" size="12" />
        </control>
        <control type="database" id = "dbConn" />
            <control type="header" label="Backup to:" />
```

```
        <control type="savefile" id = "sfile">
                        <filter>SQB Files (*.sqb)|*.sqb|All files(*.*)|*.*</filter>
                        <title>Backup File...</title>
        </control>
    </block>
</ui>
<script type="python"><![CDATA[
import RedGate #allows interaction with the UI
from os import popen #subprocess.check_output is a better way
#but it is not a default part of the SSM python interpreter

#Set this to the directory containing Sql Backup Pro
rgDir = r'C:\Program Files (x86)\Red Gate\SQL Backup 6\(LOCAL)'

rgCMD = '"{0}\SQLBackupC" -SQL "BACKUP DATABASE {1} TO DISK=\'{2}\'"'.format(
        rgDir, RedGate.dbConn.Database, RedGate.sfile.File)

RedGate.Progress.Message = rgCMD +'\n'

#This command may fail silently if run without administrator
#permissions.
results = popen(rgCMD).read()

RedGate.Progress.Message = results

RedGate.Progress.Success = True
]]></script>
<signature></signature>
</tool>
```

This script is only intended to illustrate how easy it is to call command-line applications. It must be run on the server on which the backup is to be made, it routes around the user-friendly GUI built in to SQL Backup Pro, and it requires that SSM be run with administrator permissions. It could however serve as a foundation for something larger which is useful for repeated series of operations which require only slight user-interaction. For instance, it would be fairly simple to write a script which invoked additional programs to take a backup and transfer it by secure FTP to a server on a different network. Or, if PsExec was available, it would be simple to write a script to make a fresh backup on one server, transfer it to another, and restore it there.

Additional References

i. Using SMO to manage a MS SQL database
ii. The SQL Scripts Manager XML Schema
iii. The SQL Server Central SSM Repository

List of resources for database continuous integration

Published Wednesday, May 09, 2012 12:14 PM

Because there is so little information on database continuous integration out in the wild, I've taken it upon myself to aggregate as much as possible and post the links to this blog. Because it's my area of expertise, this will focus on SQL Server and Red Gate tooling, although I am keen to include any quality articles that discuss the topic in general terms.

Please let me know if you find a resource that I haven't listed!

General database Continuous Integration

Â· What is Database Continuous Integration? (David Atkinson)

Â· Continuous Integration for SQL Server Databases (Troy Hunt)

Â· Installing NAnt to drive database continuous integration (David Atkinson)

Â· Continuous Integration Tip #3 - Version your Databases as part of your automated build (Doug Rathbone)

Â· How the "migrations" approach makes database continuous integration possible (David Atkinson)

Â· Continuous Integration for the Database (Keith Bloom)

Setting up Continuous Integration with Red Gate tools

Â· Continuous integration for databases using Red Gate tools - A technical overview (White Paper, Roger Hart and David Atkinson)

Â· Continuous integration for databases using Red Gate SQL tools (Product pages)

Â· Database continuous integration step by step (David Atkinson)

Â· Database Continuous Integration with Red Gate Tools (video, David Atkinson)

Â· Database schema synchronisation with RedGate (Vincent Brouillet)

Â· Database continuous integration and deployment with Red Gate tools (David Duffett)

Â· Automated database releases with TeamCity and Red Gate (Troy Hunt)

Â· How to build a database from source control (David Atkinson)

Â· Continuous Integration Automated Database Update Process (Lance Lyons)

Other

Â· Evolutionary Database Design (Martin Fowler)

Â· Recipes for Continuous Database Integration: Evolutionary Database Development (book, Pramod J Sadalage)

Â· Recipes for Continuous Database Integration (book, Pramod Sadalage)

Â· The Red Gate Guide to SQL Server Team-based Development (book, Phil Factor, Grant Fritchey, Alex Kuznetsov, Mladen Prajdic)

Â· Using SQL Test Database Unit Testing with TeamCity Continuous Integration (Dave Green)

Â· Continuous Database Integration (covers MySQL, Perason Education)

Technorati Tags: SQL Server,Continous Integration
by David Atkinson

.NET Memory Management and Finalization

02 May 2012
by Jean-Philippe Gouigoux

> In this excerpt from his new book, *Practical Performance Profiling: Improving the Efficiency of .NET Code*, Jean-Phillipe Gouigoux discusses the Dispose mechanism and the finalization process in the context of .NET Garbage Collection

The `System.GC` class provides access to several methods related to memory management. Unless you have a great deal of experience in .NET memory management and know precisely what you are doing, it is strongly recommended that you simply forget about its very existence.

There is a lot of temptation for beginner developers to explicitly call the `Collect` method at the end of a process which they know is memory-intensive. Of all the cases that the author is aware of since the early versions of .NET, none has ever shown that calling the GC mechanism explicitly could give better results in memory consumption or performance than simply letting the GC do its job.

Of course, in use cases where the usage does not vary at all, explicit garbage collection can allow control of memory use in one's application, but this comes with a high price tag:

- Sooner or later, the use of the application will change, and the memory consumption will vary.
- Explicit GC calls at regular intervals have the natural consequence of increasing the number of garbage collections. In the end, this increases processing time.
- As a general rule, there is no need to limit memory consumption of an application as long as the system keeps enough memory available. In the case of a system with a high memory pressure, the GC will adapt by running passes more frequently than on a system well equipped with RAM.

In short, it is essential to let the GC do its job, and not try to improve memory use by directing the collection process in any way.

We can have much more impact by helping the GC to recycle memory efficiently, by taking care of resources as explained below.

> **The question of high memory use**
> A question is often asked by beginner developers: is it normal that such and such a .NET process uses so much memory? The feeling that the CLR does not release enough of the memory used is very common, but one must understand that, as long as memory is available, it is normal for .NET to use it. And why would it do any differently? As long as the OS can provide it with memory, there is no sense in .NET limiting its use of it: running the GC more often would take time and thus slow down the application. The only important point to check is that the same process can also run in a smaller memory set.

Releasing external resources when the GC is fired

Firstly, let us make it clear that we are only talking about external resources here, such as connections to a database, memory spaces not controlled by managed code (typically in COM interoperability), or any other resources that are not controlled directly by the CLR. Indeed, contrary to what happens with C++, an object in .NET does not need to worry about whether the managed objects it references should be recycled or not. The CLR will check whether this object is referenced by any others, and if not, it will free them as well.

By contrast, in the examples below, it is important to release resources correctly. In the case of a database connection, this means calling the closing method on the corresponding API. But in most cases, as below, this operation is explicit, and there is no need to wait for the end of the object's life to close the connection.

```
using System;
using System.Data;
using System.Data.SqlClient;

namespace FreeingResources
{
    class Program
    {
        static void Main(string[] args)
        {
            SqlConnection Connection = new SqlConnection();
            SqlCommand Command = new SqlCommand("SELECT * FROM TEST", Connection);
            IDataReader Reader = null;
            try
            {
                Connection.Open();
                Reader = Commande.ExecuteReader(CommandBehavior.CloseConnection);
```

```
                if (Reader.Read())
                    Console.WriteLine(Reader.GetString(0));
            }
            finally
            {
                if (Reader != null && ! Reader.IsClosed)
                    Reader.Close();
            }
        }
    }
}
```

Listing 1

In the example above, the `CommandBehavior.CloseConnection` parameter used in the `ExecuteReader` method guarantees that the connection closing operation will be called automatically upon closure of the associated reader.

By contrast, we can imagine a .NET object for which we would need to initialize a connection during construction, and to close the connection only when the object is at the end of its life. To do so, there exists a way of informing the CLR that it should execute an action when freeing an object. Typically, this works like this:

```
using System;
using System.Data;
using System.Data.SqlClient;

namespace FreeingResources
{
    public class Tracer
    {
        SqlConnection Connection = null;

        public Tracer()
        {
            Connection = new SqlConnection();
            Connection.Open();
        }

        ~Tracer()
        {
            Connection.Close();
        }

        public void Log()
        {
            SqlCommand Command = new SqlCommand("UPDATE TABLE SET nb = nb + 1",
            Connection);
            Commande.ExecuteNonQuery();
        }
    }
}
```

Listing 2

Obviously, this example is over-simplified: keeping the connection open throughout the life of this object would only make sense if it was destined to be called extremely frequently on the `Log()` method. In the more plausible case of the method being called irregularly, it would definitely be better to open the connection and close it at the end of the function call.

This would remove the need to deal with closing the connection upon disposing of the instance, and would also free database connections for other uses, making the code more capable of handling high loads. But this is not the end of the matter, and one should remember that performance handling is often about choosing where to strike the balance between two extremes. In this example, one could argue that opening and closing the connection at each call takes processing time and slows the process down. In particular, opening a database connection is a heavy operation, which involves starting a new thread, calculating authorization levels, and several other complex operations.

So, how does one choose? Quite simply, by knowing the mechanisms used in database connection management. In practice, SQL Server will pool the connections, bringing better performance even if they are opened and closed frequently. When the `Close` instruction is called on an ADO.NET connection, the underlying object that deals with the actual database connection is in fact not abandoned, but only deactivated, and marked as available for another user. If the object is then taken from the pool, the opening of a connection is much less complex, since the object exists and the code only has to reactivate it for another use, usually only having to re-authorize it.

In short, since we have no need to deal with the object finalizer, we can write:

```
using System;
using System.Data;
using System.Data.SqlClient;

namespace FreeingResources
{
    public class Tracer
    {
        SqlConnection Connection = null;

        public Tracer()
        {
            Connection = new SqlConnection();
        }

        public void Log()
        {
            SqlCommand Command = new SqlCommand("UPDATE TABLE SET nb = nb + 1",
            Connection);
            try
            {
                Connection.Open();
                Command.ExecuteNonQuery();
            }
            finally
            {
                Connection.Close();
            }
        }
    }
}
```

Listing 3

Early release of resources

The method described above (releasing a resource upon object recycling) still has a major drawback: if the resource is precious, it is a waste to wait minutes or even hours for the GC to release it.

This is the reason behind yet another .NET mechanism: the `IDisposable` interface. Implementing this interface forces a class to have a `Dispose()` method, allowing the class instances to release resources as soon as the developer calls the method, whether it be explicitly or through the `using` keyword. Let us take an example:

```
using System;
using System.Data;
using System.Data.SqlClient;

namespace FreeingResources
{
    public class Tracer : IDisposable
    {
        SqlConnection Connection = null;

        public Tracer()
        {
            Connection = new SqlConnection();
            Connection.Open();
        }

        public void Log()
        {
            SqlCommand Command = new SqlCommand("UPDATE TABLE SET nb = nb + 1",
            Connexion);
            Command.ExecuteNonQuery();
        }
```

```
        #region IDisposable Members

        public void Dispose()
        {
            Connection.Close();
        }

        #endregion
    }
}
```

Listing 4

The user of such an object would work with a code that calls the method like this:

```
using System;
using System.Data;
using System.Data.SqlClient;

namespace FreeingResources
{
    class Program
    {
        static void Main(string[] args)
        {
            using (Tracer Logger = new Tracer())
            {
                Logger.Log();
            }
        }
    }
}
```

Listing 5

For readers that are not used to the `using` keyword, the code above is exactly equivalent to this:

```
using System;
using System.Data;
using System.Data.SqlClient;

namespace FreeingResources
{
    class Program
    {
        static void Main(string[] args)
        {
            Tracer Logger = new Tracer();
            try
            {
                Logger.Log();
            }
            finally
            {
                Logger.Dispose();
            }
        }
    }
}
```

Listing 6

By the use of `Dispose` the caller guarantees that the resources will be released as soon as possible.

Combining both operations

At this point in the evolution of our example code, something is still missing: what happens if the caller does not use the `Dispose` mechanism, by forgetting to include the `using` keyword or to call the equivalent method? Resources will not be released, even when the GC recycles the object, and there will be a resource leak.

It is thus necessary to apply both of the mechanisms we have described above, in a combined way:

```
using System;
using System.Data;
using System.Data.SqlClient;

namespace FreeingResources
{
    public class Tracer : IDisposable
    {
        SqlConnection Connection = null;

        public Tracer()
        {
            Connection = new SqlConnection();
            Connection.Open();
        }

        ~Tracer()
        {
            Connection.Close();
        }

        public void Log()
        {
            SqlCommand Command = new SqlCommand("UPDATE TABLE SET nb = nb + 1",
            Connexion);
            Command.ExecuteNonQuery();
        }

        #region IDisposable Members

        public void Dispose()
        {
            Connection.Close();
        }

        #endregion
    }
}
```

Listing 7

This way, the `Dispose` mechanism can be called explicitly to release the associated resource as soon as possible, but if for some reason this is overlooked, the GC will eventually call the finalizer. This will be done later, but it is still better than never.

Nonetheless, a seasoned developer will notice the code duplication: the finalizer and the `Dispose` function use the same code, which is contrary to a well-known best practice. As a result, we should combine the resource freeing code, like this:

```
using System;
using System.Data;
using System.Data.SqlClient;

namespace FreeinsResources
{
    public class Tracer : IDisposable
    {
        SqlConnection Connection = null;

        public Tracer()
        {
            Connection = new SqlConnection();
            Connection.Open();
        }
```

```
        ~Tracer()
        {
            FreeResources();
        }

        private void FreeResources()
        {
            Connection.Close();
        }

        public void Log()
        {
            SqlCommand Command = new SqlCommand("UPDATE TABLE SET nb = nb + 1",
            Connexion);
            Command.ExecuteNonQuery();
        }

        #region IDisposable Members

        public void Dispose()
        {
            FreeResources();
        }

        #endregion
    }
}
```

Listing 8

We are getting there, but there are still a few potential problems we have to deal with:

- If `Dispose` is called explicitly, there is no use for the finalizer anymore, because we know it will not do anything: the resource has already been freed.
- We should make sure that calling the method to free resources several times will not cause any problems.
- We should take into account the fact that, when `Dispose` is called, the `Dispose` method for other managed resources should be called as well. Generally, the CLR takes care of this by using the finalizer, but in this case, we have to do it ourselves.

The final code is:

```
using System;
using System.Data;
using System.Data.SqlClient;

namespace FreeingResources
{
    public class Tracer : IDisposable
    {
        SqlConnection Connection = null;

        public Tracer()
        {
            Connection = new SqlConnection();
            Connection.Open();
        }

        ~Tracer()
        {
            FreeResources(false);
        }

        private bool Disposed = false;

        private void FreeResources(bool Disposing)
        {
            // If the object has already released its resources,
            // there is no need to continue
```

```
            if (Disposed) return;

            if (Disposing)
            {
                // This is where the Dispose would be called if there
                // were managed resources in this class
            }

            Connection.Close();

            // To avoid coming back to this code several times
            Disposed = true;
        }

        public void Log()
        {
            SqlCommand Command = new SqlCommand("UPDATE TABLE SET nb = nb + 1",
            Connexion);
            Command.ExecuteNonQuery();
        }

        #region IDisposable Members

        public void Dispose()
        {
            FreeResources(true);
            // The following lines tell the GC that there is no use
            // in calling the finalizer, when it recycles the current object
            GC.SuppressFinalize(this);
        }

        #endregion
    }
}
```

Listing 9

This code structure is known as the “Dispose” pattern, and is quite a standard form. Despite all the effort we have put into it, it is still not 100% complete. If we want to take care of all the possible situations, we should add one more safety feature: once `Dispose` has been called, the object cannot have its `Log` method called. A traditional modification is to set the connection to `null`, and then check its value in `Log` or any method that could use it.

Further details can be found by searching for “Dispose” and “Pattern” on the internet. There are numerous discussions on side-effects and how to avoid them, memory performance of each variant of the pattern, etc. The goal of this article is not to provide the reader with a state-of-the-art summary of these discussions, but to show the link between this pattern and the performance of an application. If it is not correctly implemented, there are risks of massively reducing the access to unmanaged resources.

A last note

It is essential to stress that the memory use of a process has absolutely nothing to do with the fact it cannot release it. This is a common misunderstanding of .NET memory management. As long as the OS does not restrict the CLR in its memory consumption, .NET has no reason whatsoever to run the GC at the risk of generating a drop in performance in the application.

It is perfectly normal for an application to grow in memory up until it reaches hundreds of megabytes. Even if one pass of the GC could make this drop to ten megabytes, as long as no other process needs memory, the CLR should not sacrifice even a small percentage of its time to freeing this memory. This is the origin of the reputation of .NET and Java as “memory hogs”. In fact, they are only using available resources as much as possible, while still maintaining a process to release them as much and as quickly as possible should the operating system ask for them.

Application In real life

A developer in my team created an application that processed XML in bulk. Each file was a few hundred kilobytes at most, and the corresponding instance of XmlDocument around one megabyte. The developer, who was watching memory consumption out of curiosity, was alarmed by the fact that is was growing consistently, for each file processed, and asked me whether he should cancel the process before reaching an OutOfMemoryException. After growing to 700 megabytes or so, it suddenly dropped to around 100 megabytes, and this cycle repeated itself like clockwork until the end of the application. This case is a good example of how .NET works: on this machine, that had 2 gigabytes RAM and almost no other active applications, it would have been counter-productive to have more GC activity, since the whole process would have taken a few more minutes, whereas reducing peak memory use would have made no difference at all. It is also revealing about the difficulty of grasping the GC

mechanism for a developer that has not had it explained, which can cause performance issues, as explained above.

.NET Performance Profiling: learn the skills, write better code
If you've enjoyed this article, you can download the complete eBook covering a huge range of performance profiling topics for free here.

Towards the Perfect Build

30 April 2012
by Matt Wrock

> An automated build and deployment system is no longer a dream. Now that PowerShell has matured as a product, and since the emergence of new tools such as Psake and Chocolatey, the ambition of the perfect build process has come nearer to reality.

About two and a half years ago I wrote a series of blog posts documenting the work my team had done to automate our build process. We had completed a migration from VSS to SVN and used a combination of nAnt and CruiseControl to facilitate continuous integration and push-button deployments to any of our environments including production.

Over the last couple months, I've had the opportunity to put together an automated deployment process for my current organization at Microsoft. Throughout my career, I've worked on a few projects that were essentially a rewrite of a similar project that I had worked on in the past for a different employer. What I love about this type of project is that it is a great opportunity to do so many things better. I can remember those architectural decisions I had made and regretted; but was too far in to easily change (usually a smell of an architectural weakness itself). Well now I can avoid them and approach it from the angle I wished I had before. This was an opportunity to do the same thing with the system for automating our builds.

Although I was pleased with the system that I had put together previously, I now had better tools at my disposal. I still think nAnt and CruiseControl are fine tools, but now I'm using PowerShell with PSake instead of nAnt, TeamCity instead of CruiseControl and our source code is now in Mercurial instead of SVN. The other major difference between the system I'm building now and the one I had worked on before is that this system also includes the automation of server setup and configuration, bringing a clean OS to a full functioning application node serving any tier in the app (web, database, admin, etc.)

This article is intended to provide an overview of the new system without diving into the detail.

Do you really need an automated build and deployment system?

Yes. You do.

You may be thinking that, while an automated system sounds neat, you simply don't have time to build one. While I tend to be very pragmatic in my approach to software architecture, I definitely see automated deployments as being essential rather than a luxury. The reason I say this is that by using manual, rather than automated, deployments, you will cause more time to be lost in the mechanics of deploying and there is far more risk of a bad deployment and there is more difficulty and time spent in troubleshooting deployments.

Often, teams do not recognize the value of automated deployments until they experience them. Once they work with one, they can't imagine going back. With automated build and deployments, the drama of deployments is reduced to a simple routine task and teams have more time to focus on building features. The business has more confidence that their features will move forward reliably and consistently. If you want to release more often and perhaps extend continuous integration to continuous deployment, you simply must automate the deployment process.

If they are so important, why did it take you over two years to start building one?

This is a fair question. I don't intend to enumerate the political reasons, of which there are many. That will have to wait for my memoire due out in 2042, "My life, a love song," please keep an eye out for that one.

Throughout my tenure in the MSDN/Technet organization at Microsoft, deployments have been managed by a combination of test and a "build team" in the Ops group. Although I have certainly been vocal in pushing for more automation, I was wary of forcing the pace of change because other people do most of the work and that there was resistance from some to the idea of automating the process. There were certainly pain points along the way. There was a lot of ceremony involved in preparing for a deployment and in scheduling "hot fixes" with the build team. When there were problems with a deployment, it could sometimes be difficult to determine where things went wrong.

Recently, we made a transition to a new offshore vendor company. One of their responsibilities would be deployments and setting up new environments. Because these were mostly done manually, the logistics involved were often communicated verbally and via large step by step Word documents.

Without going into the details, a lot fell through the cracks as the new team came on-board. I do not fault the people on this team, I wouldn't expect anyone to be able to build an environment for a complex app that they have never worked on before based on a few phone conversations and a SharePoint wiki. Our environment setups and deployments suddenly started having problems. Because a large part of the code I am involved with spans over several apps, I am often approached when things go wrong here and before long I found myself spending most of my time troubleshooting and fixing environments and their deployments. It soon became crystal clear that, until an automated system was in place, this would continue to stand in the way of me getting real feature work done; and instead of whining and complaining about it, I decided to just do it.

What exactly does a automated build and deployment system do?

For the system I set out to build, the following key components are included:

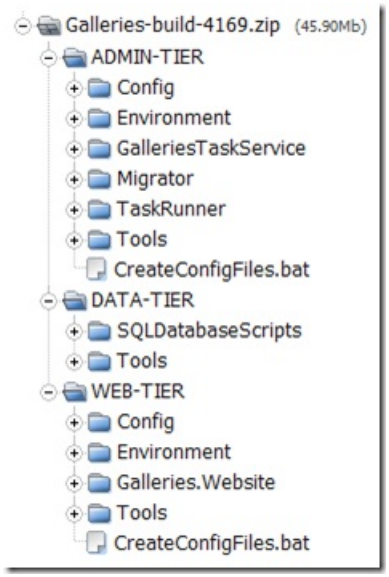
1. Application compilation and packaging
2. Deployment of application packages to various environments
3. Bootstrap scripts for setting up a new server or environment

The last one has inspired a new personal side project, Autobox, that sets out to automate the building of a developer machine (or any kind of personal machine) from bare OS via a single command line. After all, if I can create a test server with SQL Server, app fabric caching, various Windows services, and web applications along with all the file permissions and firewall rules involved, certainly I can create my own machine with all my preferred apps and settings ready to go.

Lets examine each of these individually.

Application compilation and packaging

This is, essentially, the process that transforms the raw application bits with all of its code files, static assets, SQL scripts, configuration files, and other application-specific files into a zip file that can be consumed by the deployment scripts. This package, in our case, is typically composed of a directory for each application tier. Here is the package for our Galleries application:



The packaging process is responsible for the actual compilation which typically involves a call to MsBuild and which invokes the appropriate MsBuild tasks from the original Visual Studio solution. In addition to transforming the source files to compiled DLLs, the packaging process copies everything that is needed in order to deploy the application into a coherent directory structure and nothing more. This typically includes PowerShell scripts and various command line tools that run SQL scripts to update the database with any schema changes, adds metadata to lookup tables or migrates old data to conform to new schema or logic. It may also include scripts responsible for transforming web.config and app.configs with settings that are appropriate for the environment.

This first step of the build and deployment process had been in place for quite some time, so I just had to make some minor tweaks here and there. The individual application teams in my group are responsible for keeping the packaging scripts up to date and it is wired into our continuous Integration process. Every push of source code to the central Mercurial repository forces our build server, Teamcity, to invoke a set of scripts that include compilation, running unit tests and finally packaging. TeamCity then saves the zipped package and makes it available to the deployment scripts. If you are familiar with Teamcity, you know that this is the build “Artifacts.”

Deployment of application packages to various environments

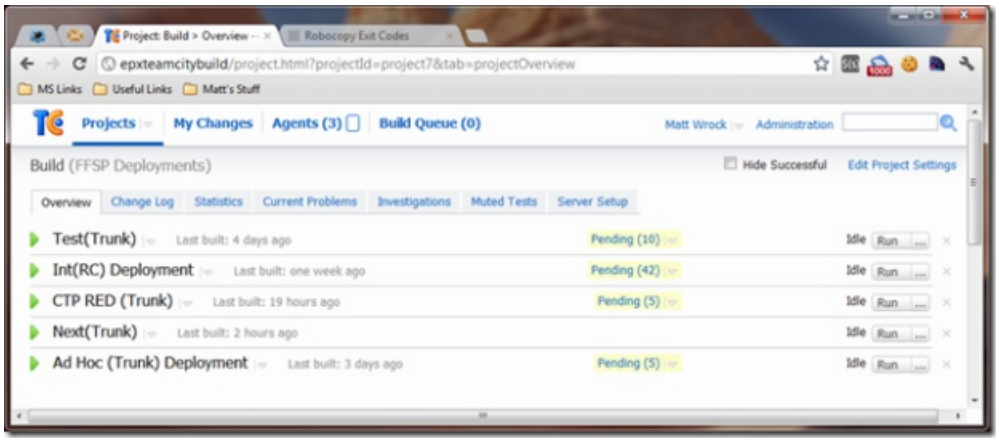
Here is where my work largely started. Until recently, we had a script that TeamCity would invoke twice a day which would collect the packages of each app and aggregate them into another package for each deployable environment. This uses the “dependencies” feature of TeamCity. In your TeamCity build settings, you can state that your build is dependent on another build or group of builds. In our case and as illustrated below, we had six depedent builds that would be aggregated into a master build package. The “artifacts” of the last successful build of each dependent build is unzipped and placed in a named directory relative to the working directory of the master build. For example, the deployable artifacts of the Forums application are unzipped and placed in a directory named Forums that the master build script can reference via “forums\...”

Artifacts source	Artifacts paths
AutoComplete :: trunk *(inherited)* (Last successful build)	*.zip!**=>Autocomplete
Chameleon :: trunk *(inherited)* (Last successful build)	*.zip!**=>Chameleon
Forums :: trunk *(inherited)* (Last successful build)	*.zip!**=>Forums
Framework :: main *(inherited)* (Last successful build)	*.zip!**=>CommunityPlatform
Profile :: trunk *(inherited)* (Last successful build)	*.zip!**=>Profile
Search :: trunk *(inherited)* (Last successful build)	*.zip!**=>Search

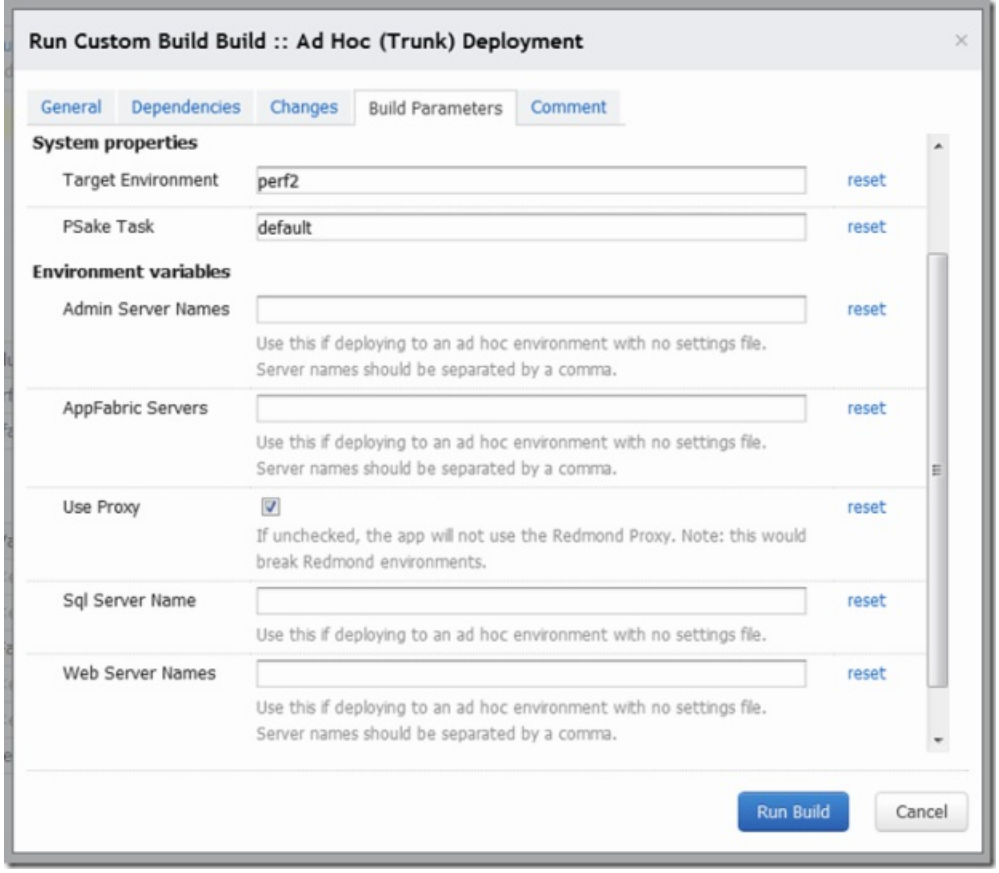
So in our case, we would have application packages for Forums, Search, Profile and various internal services as seen above and these would all be rolled into a single 7z file for each environment including test, staging, production, etc. This packaging script was also responsible for the final transformation of the configuration files. It would merge those settings that are specific to each environment into the web and app configs so that the final package, say prod-7791.7z (7791 being the build number), had the exact web and app configs that would end up in production.

Well this would take two and a half hours to run. Back in the day, it was fairly fast; but as environments got added, the process took longer and longer. It would then take the build team a couple hours to take this package and deploy its bits to each server, run the database upgrade scripts, stop and restart services, smoke test, etc. This could become more and more painful the closer we got to release. This was because, as or when the developers would fix bugs, it could take one to two days before they received feedback from test on those bugs.

Revamping this was fairly straightforward. I rewrote this script to transform the configs for only a single environment which it would receive via a command parameter from TeamCity. I created a separate build config in TeamCity to make this very clear:



Each of these build configurations run the exact same script but they each pass different command line arguments to the build script so as to indicate their environment. Also, some are wired to different Version Control branches. For example, our Int (Integration) environment builds off of the Release Candidate branch while the others build off of Trunk. Finally there is an “Ad Hoc” config where anyone can run a custom build with custom command line parameters. If the Ad Hoc build fails no one is notified and we don’t get particularly alarmed. Here is how the command line parameters are wired up for custom builds in TeamCity:



The script is a normal PowerShell script that gets called via psake. Psake provides a very nice PowerShell-based container for running builds. Think of it as an alternative to writing an MSBuild script. While MSBuild is more XML-based and very declarative in nature, PSake allows you to script out all of your build tasks in PowerShell. This makes a lot of sense for the type of things that a build script does - such as copying files around. I'm not going to dive into a PSake tutorial here but here is a snippet of my PSake script:

```
properties {
    $current = Resolve-Path .\default.ps1 | Split-Path
    $path = $current
    $BuildNumber = 0
    $ConfigDrop = ".\_configs"
    $WebDrop = "HTTP"
    $Environment = "DEFAULT"
    $configVariables = New-Object System.Collections.Queue
}

Include .\psake\teamcity.ps1

task default -depends Package
task Configs -depends Copy-readme, SetupEnvironment, Configure-Social,
    Configure-StoApps, Configure-Services, Configure-SocialServices
task Package -depends SetupEnvironment, Clean, Configs, Database, preparesearch,
    SocialSites, StoApps, SocialServices, StopServices, Services, CopyConfigs,
    StartServices, FlushRequestReduce, RestartIIS, RestartAppFabric, TestPages,
    CleanEnvironment

TaskSetup {
    TeamCity-ReportBuildStart "Starting task $($psake.context.Peek().currentTaskName)"
}

TaskTearDown {
    TeamCity-ReportBuildFinish "Finishing task $($psake.context.Peek().currentTaskName)"
}

Task SetupEnvironment {
    .\Utilities\EnvironmentSetup.ps1 $current $Environment $configVariables
}
```

This is not any kind of special scripting language. It is normal PowerShell. PSake provides a PowerShell module which exposes several functions like Task, Properties, etc. Many of these take script blocks as parameters. The PSake module really is not very large and therefore it does not take

much investment to understand what it does and what functionality it provides. It really does not provide much “functionality” at all, in terms of utility methods, but it provides a very nice framework for organizing the various parts of your build script and for specifying dependencies.

The snippet above is the beginning of my deployment script. The Properties section defines and sets script-wide variables that can be overridden via command line parameters when calling PSake. Next are my tasks. Tasks might actually do something like the **SetupEnvironment** task at the bottom. Or they might alias a group of tasks to be run in a specific order like the default, Configs and Package tasks. If you are familiar with MSBuild, these are simply the equivalent of MSBuild targets.

When you call PSake, you can tell it to run a specific task or, if you do not, it will run the default task. Even though I am only including a small part of my script here, it is easy to tell what the deployment script does by simply looking at the dependencies of the default task. It first sets up the environment by calling another PowerShell script that will set a bunch of global environment variables specific to the Environment property. It performs a task to clean traces of any previous build, it transforms the configs, and runs the database scripts. Then it executes several tasks that copy different directories to the web server, stops some windows services, copies the services code, starts the services, restarts IIS, runs some quick tests to make sure the apps are loading and finally cleans up after itself.

One nice thing about this script is that it does not use any kind of remoting. This can be important in some environments. The script can be run directly from the build agent (the server running the TeamCity Build Agent service) and target any environment. Instead of remoring, it requires that the Service Identity under which TeamCity runs is an administrator on the target web servers and SQL Servers. To give you a glimpse into what is going on here, I specify all of the server names that are specific to each environment in a config file named after the environment. So our Next (daily build) environment has a file called Next.ps1 that among many other things contains:

```
$global:WebServers     = "RR1STOCSAVWB18", "RR1STOCSAVWB17"
$global:ServicesServer = "RR1STOCSAVWB17"
```

Then my RestartIIS task looks like this:

```
Task RestartIIS {
    Restart-IIS $global:WebServers
}

function Restart-IIS([array] $servers) {
    foreach ($server in $servers) {
        .\Utilities\RemoteService.ps1 ($server -split "\\")[0] restart -service "W3SVC"
    }
}
```

RemoteServices.ps1 contains a bunch of functions to make working with services on remote servers less painful.

Did the deployment succeed?

At any point in the scripts, if an error occurs, the build will fail. However, I also want to have some way to quickly check each application and ensure they can at least load. It is very possible that the build script will complete just fine, but there may be something in the latest app code or some change to the environment that would cause an application to fail. If this happens, I want to know which app failed, fail the build and provide straight forward reporting to testers to discover where things broke down. Yes, each app build has its own set of unit tests. Most apps have thousands but there are a multitude of issues, both code-related and server or network-related, that can slip through the cracks and cause the app to fail.

At the end of every deployment, a series of URLs are “pinged” and expected to return a 200 HTTP status code. Currently we have 28 URLs in our tests. Now a big reason for overhauling this system was to make it faster. This meant that we were concerned that we would profoundly slow the build process by launching a bunch of app URLs. To try to make this as efficient as possible, we use PowerShell jobs to multi-thread the HTTP requests and set a 5 minute timeout that will automatically fail all tests that do not complete before the timeout.

Here is the testing script:

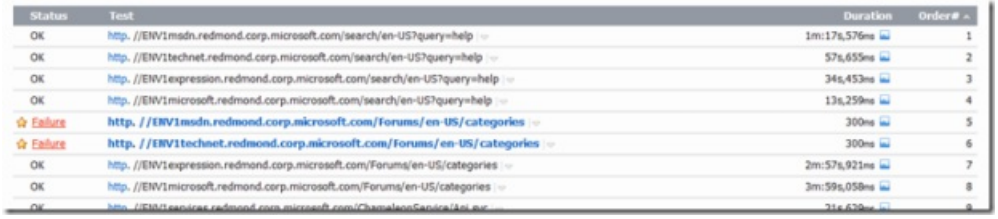
```
task TestPages -depends SetupEnvironment {
    . .\tests.ps1
    Wait-Job -Job $openRequests -Timeout 300
    foreach ($request in $openRequests) {
        TeamCity-TestStarted $request.Name
        $jobOutput = (Receive-Job $request)
        if($jobOutput -is [system.array]) {$jobOutput = $jobOutput[-1]}
        $testParts = $jobOutput -split " ::: "
        if($testParts.Length -eq 2) {
            $testMessage=$testParts[1]
            $testTime=$testParts[0]
        }
        else {
```

```
            $testMessage=$testParts[0]
            $testTime=300
        }
        if($request.state -ne "Completed") {
            TeamCity-TestFailed $request.Name "Timeout" "Test did not complete within timeout."
        }
        Remove-Job $request -Force
        if ($testMessage -like "Failed*") {
            TeamCity-TestFailed $request.Name "Did not Recive a 200 Response" $testMessage
        }
        TeamCity-TestFinished $request.Name $testTime
    }
}

function Ping ([string] $pingUrl) {
    $jobArray = @()
    $job = Start-Job -scriptblock {param($url)
        $host.UI.RawUI.BufferSize = New-Object System.Management.Automation.Host.Size(8192,50)
        $ms = (Measure-Command {
            $web=[net.HTTPwebrequest]::create($url)
            $web.AllowAutoRedirect = $true
            $web.PreAuthenticate = $true
            $web.Timeout = 300000
            $systemWebProxy = [net.webrequest]::GetSystemWebProxy()
            $systemWebProxy.Credentials = [net.CredentialCache]::DefaultCredentials
            $web.Proxy = $systemWebProxy
            $web.Credentials = [net.CredentialCache]::DefaultCredentials
            try {
                $resp=$web.GetResponse()
            }
            catch [System.Net.WebException]{
                $resp=$_.Exception.Response
                $outerMessage = $_.Exception.Message
                $innerMessage = $_.Exception.InnerException
            }
        }).TotalMilliseconds
        $status = [int]$resp.StatusCode
        if ($status -ne 200) {
            $badServer = $resp.Headers["Server"]
            Write-Output "$ms ::: Failed to retrieve $url in $ms ms with status code:
                $status from server: $badServer"
            Write-Output $outerMessage
            Write-Output $innerMessage
        }
        else {
            Write-Output "$ms ::: Succeeded retrieving $url in $ms ms"
        }
    } -name "$pingUrl" -ArgumentList $pingUrl
    $jobArray += $Job
    return $jobArray
}
```

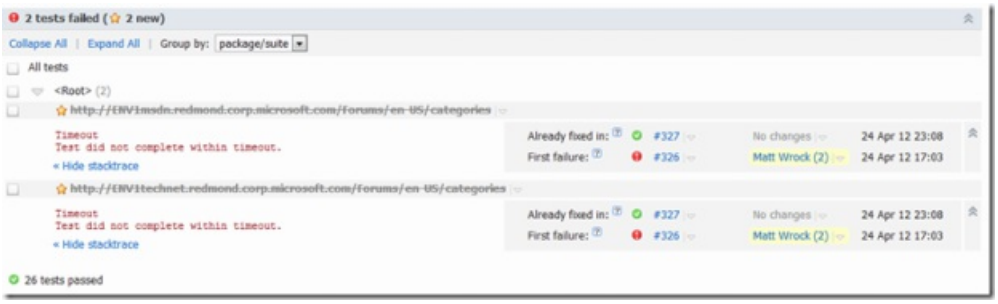
The individual test URLs are in the dot sourced tests.ps1:

```
$openRequests += Ping "HTTP://$global:ServicesUrl/ChameleonService/Api.svc"
$openRequests += Ping "HTTP://$global:AdminUrl/ChameleonAdmin/"
$openRequests += Ping "HTTP://$global:ServicesUrl/SearchProviderServices/SearchProviderService.svc"
$openRequests += Ping "HTTP://$global:ProfileApiUrl/ProfileApi/v1/profile/displayname/vamcalerts"
$openRequests += Ping HTTP://$global:UserCardLoaderUrl
...
```

An interesting thing to note here is the use of the functions beginning with TeamCity-. These are functions coming from a module provided by the pake-contrib project that exposes several functions that allow you to interact with TeamCity's messaging infrastructure. The functions I am using here create standard output messages formatted in such a way that TeamCity will treat them like test output reporting when a test starts and finishes, as well as if it succeeded or failed, and how long it took. What is really nice about all of this is that now these tests light up in TeamCity's test reporting:



I can zoom in on my failed tests to see why they failed:



Pretty slick eh?

Bootstrap scripts for setting up a new server or environment

In my original Perfect Build series of blog posts, I did not include automation around setting up servers or environments. However one of the habits I picked up from the teams I work with at Microsoft is the inclusion of a build.bat file at the root of every source code repo that can build a development environment from scratch. In the past I had never followed this practice. I had not really used PowerShell and was not aware of all the possibilities available. Basically, you can do pretty much anything in PowerShell. I'll admit that there is a learning curve involved but it is well worth climbing it. Being able to fire up a development environment for an app with a single command has proven to be a major time saver and a great way to "document" application requirements.

Now, it's one thing to get a development environment up and running, but getting a true server environment up can be more challenging. Since many organizations don't give developers access to the server environments, setting these up often becomes a responsibility of server operations. This may involve dev sending ops instructions, or sitting down with an ops engineer to get a server up and running. A lot of time can be lost here and it's easy to forget to properly update these instructions. I have, personally, spent an aggregate of weeks troubleshooting environments that have not been set up correctly.

One solution that is commonly employed here is to use VM images. Once you get an environment set up the way it is supposed to be inside of a VM, take a snapshot and simply apply that snapshot whenever you need to setup a new server. I don't like this approach. It is too easy for VM images to become stale and they don't serve well to "document" all of the requirements of an application. The fact is, just about anything can be scripted in PowerShell and, in my opinion, if it cannot be scripted then you have probably made a poor choice in technology. PowerShell scripts can replace "deployment documents" or server setup documents. They should be readable by both developers and server support engineers. Even if one is not well versed in PowerShell, I believe any technical professional should at least be able to read a PowerShell script and deduce the gist of what it is doing.

For my applications, I put together a script, again in psake format, that can build any application tier from a bare OS. It can also build a complete environment on a stand alone server. To provide an idea of what my script can do, here is the head of the psake script:

```
properties {
    $currentDir = Resolve-Path .\cleansetup.ps1 | Split-Path
    $profile = "$currentDir\CommApps\Profile"
    $forums = "$currentDir\CommApps\Forums"
    $search = "$currentDir\CommApps\Search"
    $chameleon = "$currentDir\CommApps\Chameleon"
    $configVariables = New-Object System.Collections.Queue
    IF ( TEST-PATH d:\) { $HTTPShare="d:\HTTP" } else { $HTTPShare="c:\HTTP" }
    $env = "test"
    $appFabricShareName = "velocity"
    $buildServerIdentity = "Redmond\Idiotbild"
    $domain = "redmond.corp.microsoft.com"
    $buildServer = "EpxTeamCityBuild.redmond.corp.microsoft.com"
    $buildServerQueue = "bt51"
    $doNotNeedsBits = $false
    $addHostFileEntries = $false
    $sqlServer = $env:computername
    $appFabricServer = $env:computername
    $AdminServer = $env:computername
    $restartSuffix = ""
```

```
    $noProxy = $true
}

Include .\psake\teamcity.ps1

task default -depends standalone
task standalone -depends Setup-Proxy, Set-EnvironmentParams, Pull-Bits, Setup-Roles,
    Disable-InternetExplorerESC, Database-server, Install-IIS-Rewrite-Module,
    Install-Velocity, Setup-MSDTC, Install-Event-Sources, Install-Certificates,
    Setup-Response-Headers, Register-ASP, Wait-For-Bits, Setup-IIS, Add-DB-Perms,
    Configure-Velocity, Install-WinServices, Set-Queue-Perms
task WebAppCache-Server -depends Setup-Proxy, Set-EnvironmentParams, Pull-Bits,
    Setup-Roles, Configure-Group-Security, Install-IIS-Rewrite-Module, Install-Velocity,
    Setup-MSDTC, Install-Event-Sources, Install-Certificates, Setup-Response-Headers,
    Register-ASP, Wait-For-Bits, Setup-IIS, Add-DB-Perms, Configure-Velocity,
    Install-WinServices, Set-Queue-Perms
task AppFabric-Server -depends Setup-Proxy, Set-EnvironmentParams, Setup-Roles,
    Configure-Group-Security, Install-Velocity, Configure-Velocity
task Web-server -depends Setup-Proxy, Set-EnvironmentParams, Pull-Bits, Setup-Roles,
    Configure-Group-Security, Install-IIS-Rewrite-Module, Setup-MSDTC, Install-Event-Sources,
    Install-Certificates, Setup-Response-Headers, Register-ASP, Wait-For-Bits,
    Setup-IIS, Add-DB-Perms
task Admin-Server -depends Setup-Proxy, Set-EnvironmentParams, Pull-Bits, Setup-Roles,
    Configure-Group-Security, Install-IIS-Rewrite-Module, Setup-MSDTC, Install-Event-Sources,
    Setup-Response-Headers, Register-ASP, Wait-For-Bits, Setup-IIS, Add-DB-Perms,
    Install-WinServices, Set-Queue-Perms
task Database-Server -depends Set-EnvironmentParams, Configure-Group-Security,
    Install-SqlServer, Create-Databases
task Post-Restart-Full -depends Set-EnvironmentParams, Remove-Startup,
    Configure-Velocity, Install-WinServices, Set-Queue-Perms
task Post-Restart -depends Remove-Startup, Configure-Velocity
task Get-Bits -depends Set-EnvironmentParams, Pull-Bits, Wait-For-Bits
```

By looking at the tasks, you can get a feel for all that's involved at each tier. First let me say that this script took about 20x more effort to write than the deployment script. I'm proud to report that I mastered file-copying long ago. Once I finally managed to figure out the difference between source and destination, it's been smooth sailing ever since. This script really taught me a lot about not only PowerShell but also a lot about how the windows OS and many of the administrative apps work together.

If I had to identify the step that was the biggest pain in the butt to figure out, by far and away it was installing and configuring AppFabric. This is Microsoft's distributed caching solution formerly known as Velocity. One thing that makes it tricky is that, at least in my case, it requires a reboot after installation and before configuration. I certainly do not want to include our entire server setup script here but let me include the AppFabric portion. Again keep in mind this is coming from a psake-consumable script. So the tasks can be thought of as the "entry points" of the script while the functions serve as "private" helper methods to those from more formal programming languages.

```
task Install-Velocity -depends Install-DotNet4 {
    $global:restartNeeded = $false
    Start-Service -displayname "Windows Update"
    if (!(Test-Path "$env:windir\system32\AppFabric")){
        $dest = "appfabric.exe"
        if (Is64Bit){
            $url = "HTTP://download.microsoft.com/download/1/A/D/
                1ADC8F3E-4446-4D31-9B2B-9B4578934A22/WindowsServerAppFabricSetup_x64_6.1.exe"
        } else{
            $url = "HTTP://download.microsoft.com/download/1/A/D/
                1ADC8F3E-4446-4D31-9B2B-9B4578934A22/WindowsServerAppFabricSetup_x86_6.1.exe"
        }
        Download-File $url (join-path $currentDir $dest)
        ./appfabric.exe /i "cachingservice,cacheclient,cacheadmin"

        Start-Sleep -s 10
        $p = Get-Process "appfabric"
        $p.WaitForExit()
        $global:restartNeeded = $true
    } else
    {
        Write-Host "AppFabric - Already Installed..." -ForegroundColor Green
    }
}
```

```
task Configure-Velocity -depends Create-Velocity-Share, Install-Velocity {
    if($global:restartNeeded -eq $true -or $global:restartNeededOverride -eq $true) {
        RebootAndContinue
    }
    Load-Module DistributedCacheConfiguration
    $clusterInfo = Get-CacheClusterInfo "XML" "\\$env:computername\$appFabricShareName"
    if( $clusterInfo.IsInitialized -eq $false ) {
        new-CacheCluster "XML" "\\$env:computername\$appFabricShareName" "Medium"
        Register-CacheHost -Provider XML -ConnectionString "\\$env:computername\$appFabricShareName"
            -CachePort 22233 -ClusterPort 22234  -ArbitrationPort 22235 -ReplicationPort 22236
            -HostName $env:computername -Account "NT AUTHORITY\Network Service"
        Add-CacheHost -Provider XML -ConnectionString "\\$env:computername\$appFabricShareName"
            -Account "NT AUTHORITY\Network Service"
        Load-Module DistributedCacheAdministration
        use-cachecluster -Provider XML -ConnectionString "\\$env:computername\$appFabricShareName"
        New-Cache ForumsCache -TimeToLive 1440
        Set-CacheClusterSecurity -SecurityMode None -ProtectionLevel None

        start-cachecluster
        netsh firewall set allowedprogram
            $env:systemroot\system32\AppFabric\DistributedCacheService.exe APPFABRIC enable
    }
}

function Is64Bit
{
    [IntPtr]::Size -eq 8
}
function Download-File([string] $url, [string] $path) {
    Write-Host "Downloading $url to $path"
    $downloader = new-object System.Net.WebClient
    $downloader.DownloadFile($url, $path)
}
function RebootAndContinue {
    $global:restartNeededOverride = $false
    Copy-Item "$currentDir\post-restart-$restartSuffix.bat"
        "$env:appdata\Microsoft\Windows\Start Menu\programs\startup"
    Restart-Computer -Force
}
```

Now there are several ways to configure AppFabric and this just demonstrates one approach. This uses the XML provider and it only installs the caching features of AppFabric.

Installing applications with Chocolatey

One “rediscovery” I made throughout this process is an open source project built on top of Nuget called Chocolatey. This is the brain child of Rob Reynolds who is one of the original creators of what we know of as Nuget today and was once called Nu before development was handed off to Microsoft and Outercurve. I say “rediscovery” because I stumbled upon this a year ago but didn’t really get it. However it really makes sense when it comes to build/setup automation whether that is an application server or your personal machine.

Chocolatey is a framework around the processes of installing and setting up applications via silent installations. Many of the apps that you and I are accustomed to manually download then launch the installer and click next, next, next, finish, are available via Chocolatey’s public feed. In addition to its own feed, it exposes the web platform installer’s command line utility so that any application available via the web platform installer can be silently installed with Chocolatey. Since it really just sits on top of Nuget, you can provide your own private feed as well.

So lets look at exactly how this works by exploring my setup script’s bootstrapper:

```
param(
    [string]$task="standalone",
    [string]$environment="test",
    [string]$sqlServer = $env:computername,
    [string]$appFabricServer = $env:computername,
    [string]$AdminServer = $env:computername,
    [string]$domain = "redmond.corp.microsoft.com",
    [switch]$doNotNeedBits,
    [switch]$addHostFileEntries,
    [switch]$skipPrerequisites,
    [switch]$noProxy
```

```
)
iex ((new-object net.webclient).DownloadString('HTTP://bit.ly/psChocInstall'))

if(-not $skipPrerequisites) {
    .$env:systemdrive\chocolatey\chocolateyinstall\chocolatey.cmd install hg
    if( test-path "$env:programfiles\Mercurial" ) {
        $mPath="$env:programfiles\Mercurial"
    }
    else {
        $mPath = "${env:programfiles(x86)}\Mercurial"
    }

    if( -not( test-path $env:systemdrive\dev )) { mkdir $env:systemdrive\dev }
    set-location $env:systemdrive\dev
    if( test-path socialbuilds ) {
        set-location socialbuilds
        .$mPath\hg pull
        .$mPath\hg update
    }
    else {
        .$mPath\hg clone HTTPs://epxsource/SocialBuilds
        set-location socialbuilds
    }
}
if($task -eq "standalone") {$addHostFileEntries=$true}
if($task -ne "AppFabric-Server") {$restartSuffix="Full"}
./psake/psake.ps1 cleansetup.ps1 -tasklist $task -properties @{env=$environment;sqlServer=$sqlServer;
    appFabricServer=$appFabricServer;AdminServer=$AdminServer;domain=$domain;
    doNotNeedBits=$doNotNeedBits;addHostFileEntries=$addHostFileEntries;
    restartSuffix=$restartSuffix;noProxy=$noProxy}
```

Notice these key lines:

```
iex ((new-object net.webclient).DownloadString('HTTP://bit.ly/psChocInstall'))
```

This Downloads and installs Chocolatey and then here is an example of using chocolatey to download the Mercurial source control client:

```
.$env:systemdrive\chocolatey\chocolateyinstall\chocolatey.cmd install hg
```

I should point out that under most circumstances, the above line could simply be:

```
cinst hg
```

Chocolatey's install puts itself in your path, and creates some aliases that makes this possible; but because I use Chocolatey here in the same script that installs Chocolatey, the environment variables it sets are not available to me yet. I'd need to open a new shell.

As a side note, I use chocolatey all the time now. If I need to hop on a random box and install a tool or set of tools, I now just launch a few lines of PowerShell and its all there. At Microsoft I often get asked for source code to my repos by fellow employees who are unfamiliar with Mercurial. I have found that sending an email like this is very effective:

"

Hi Phil,

You can get that from HTTPs://epxsource/Galleries. We use Mercurial. The easiest way to get everything you need is to launch this from PowerShell as admin:

```
iex ((new-object net.webclient).DownloadString('HTTP://bit.ly/psChocInstall'))
.$env:systemdrive\chocolatey\chocolateyinstall\chocolatey.cmd install hg
$env:programfiles\Mercurial\hg clone HTTPs://epxsource/Galleries
```

This will install Mercurial and clone the galleries repo.

Matt

How cool is that? No Mercurial tutorial needed and sometimes I get a reply back telling me what a cool script that is. I should really forward the compliment to Rob Reynolds since he was the one who, basically, wrote it.

So this really makes the consumption of my server setup script simple. As you can see, it basically clones (or updates) my script repo on the target machine where the script runs. This also means that, if I commit changes to my script, rerunning this script on the box will automatiucally pull in those

changes. To simplify things further, I provide a batch file wrapper so that the script can be launched from any command line:

```
@echo off

PowerShell -NonInteractive -NoProfile -ExecutionPolicy bypass
    -Command "& '%~dp0bootstrap\bootstrap.ps1' %*"
```

The only thing this does is to call the PowerShell bootstrap.ps1 script (the one listed before) but key to this call is:

```
-ExecutionPolicy bypass
```

Without this and assuming this script is being run on a fresh box, the user would get an error trying to run most PowerShell scripts. This prevents any scripts from blocking and suppresses all warnings regarding the security of the scripts. Often you will see advice that suggests that you use "unrestricted". However, I have found that "bypass" is better especially since I have had issues with setting the execution policy to 'unrestricted' on Windows 8. According to the documentation on execution policies:

> **Bypass**
> *- Nothing is blocked and there are no warnings or prompts.*
>
> *- This execution policy is designed for configurations in which a Windows PowerShell script is built in to a a larger application or for configurations in which Windows PowerShell is the foundation for a program that has its own security model.*

This seems to match the use case here.

The one liner setup call

So now, as long as I put my batch file and bootstrap.ps1 on a network share accessible to others who need to use it, simply typing this at any command prompt will kick off the script:

```
\\server\share\bootstrap.bat
```

By default, with no command line parameters passed in, a standalone setup will be installed. In my case, it takes about an hour to complete and I have a fully-functioning set of applications when finished.

Making this personal

I've been very impressed with what I can get done in PowerShell, and by the ease with which I can install many applications using Chocolatey. This has inspired me to create a personal bootstrapper which I have been tweaking over the past several weeks. It is still very rough and there is much I want to add but I'd like to craft it into a sort of framework allowing individuals to create sort of "recipes" that will serve up an environment to their liking. We are all VERY particular about how our environments are laid out and there really is no 'one size fits all'.

If you are interested in seeing where I am going with this, I have been keeping it at Codeplex here. Right now this is really about setting up MY box, but it does do some interesting things such as to download and install windows updates, turn off UAC (that dialog box that you may have never clicked "no" on) and makes Windows Explorer usable by changing the defaults and showing me hidden files and known extensions. Here is the script for the windows explorer "fix":

```
function Configure-ExplorerOptions([switch]$showHidenFilesFoldersDrives,
                                   [switch]$showProtectedOSFiles,
                                   [switch]$showFileExtensions) {
    $key = 'HKCU:\Software\Microsoft\Windows\CurrentVersion\Explorer\Advanced'
    if($showHidenFilesFoldersDrives) {Set-ItemProperty $key Hidden 1}
    if($showFileExtensions) {Set-ItemProperty $key HideFileExt 0}
    if($showProtectedOSFiles) {Set-ItemProperty $key ShowSuperHidden 1}
    Stop-Process -processname explorer -Force
}
```

So I hope you have found this helpful.

> See also Matt's blog posts in the same vein...
> The Perfect Build Part 3: Continuous Integration with CruiseControl.net and NANT for Visual Studio Projects
> *A couple months after migrating to subversion, Matt's team took another significant step to improve their build...*
> The Perfect Build Part 1
> *A year ago, Matt's team was using Visual Source Safe as their version control repository...*
> The Perfect Build Part 2: Version Control
> *Over a year ago, Matt's team was using Visual Source Safe (VSS) for version control!.*

.NET Demon support for VS 11 dark theme

Published Thursday, May 10, 2012 11:15 AM

I'm pleased to announce that .NET Demon will be shipping simultaneously with Visual Studio 11, whenever it ends up being released. That means we're going to make sure that a version of .NET Demon is released very near to the Visual Studio 11 final release which supports the new version of VS fully.

The interesting part of this support is going to be the new dark theme of VS, which I'm looking forward to using. I'm told dark colours reduce eye strain for developers. It's important that extensions like .NET Demon switch to a dark theme when the rest of the IDE changes, or the dark theme will look silly. Unfortunately, none of my favourite extensions look right in the dark theme yet, so even though I use Visual Studio 11 beta for my day-to-day development already, I can't use the dark theme.

Luckily .NET Demon uses WPF throughout, and the team at Microsoft are helping us to use the WPF Style system to make it easy for me to implement the support without having to add colour attributes to all the controls manually. We should have dark theme support in .NET Demon in the next month or so.

by Alex.Davies

I'm blogging again, and about time too

Published Saturday, May 12, 2012 3:02 AM

No, seriously, this one is about time.

I recently had an issue in a work database where a query was giving random results, sometimes the query would return a row and other times it wouldn't. There was quite a bit of work distilling the query down to find the reason for this and I'll try to explain by demonstrating what was happening by using some sample data in a table with rather a contrived use case.

Let's assume we have a table that is designed to have a start and end date for something, maybe a working session needs to be recorded, or a maybe its a set of rules that have start and end dates, the key thing is we are working with dates and times; A person starts working and then stops working, a rule becomes enforced at a certain time and ceases to be relevant at a later time. We can safely use a SMALLDATETIME data type for the Start and End columns as we don't need more accuracy.

Let's use our first example and create a table to log when a person is working.

```
IF OBJECT_ID('Attendance') IS NOT NULL
    DROP TABLE [dbo].[Attendance]
go

CREATE TABLE [dbo].[Attendance]
      (
        [ID] [int] NOT NULL
                   IDENTITY(1, 1) ,
        [PersonIn] [smalldatetime] NOT NULL ,
        [PersonOut] [smalldatetime] NULL ,
        [Empno] [int] NULL
      )
ON  [PRIMARY]
GO
```

We have a very simple design that enforces us to have an employee number and a start time but the end time can be NULL to allow for a person starting a session but not yet having completed it. Those sessions would be the ones that are in progress.

I use SQL Data Generator from RedGate* to create some random data in the table and then we can start analysing the contents.

With this information we can query the table to see who is currently on their shift or who *was* on a shift by using an historical date in a WHERE clause as in these two examples

```
31  DECLARE @SessionDate DATETIME
32  SET @SessionDate = GETDATE()
33
34  SELECT empno ,
35          [a].[PersonIn] ,
36          [a].[PersonOut]
37      FROM [dbo].[Attendance] AS a
38      WHERE @sessiondate BETWEEN [a].[PersonIn] AND [a].[PersonOut]
39
40  SET @SessionDate = '2012-05-02 12:00:00'
41
42  SELECT empno ,
43          [a].[PersonIn] ,
44          [a].[PersonOut]
45      FROM [dbo].[Attendance] AS a
46      WHERE @sessiondate BETWEEN [a].[PersonIn] AND [a].[PersonOut]
47  GO
48
49
50
100 %
Results  Messages

(0 row(s) affected)

(14 row(s) affected)
```

The results show 0 records started before the current date and ended after it and 14 records started before 2012-05-02 12:00:00 and ended after it.

That doesn't seem right.

I generated the data with a fixed percentage of nulls to be inserted into the PersonOut column, there must be some records where the session is still in progress. Lets allow for NULLs in our TSQL ...

```
DECLARE @SessionDate DATETIME
SET @SessionDate = GETDATE()

SELECT empno ,
        [a].[PersonIn] ,
        [a].[PersonOut]
    FROM [dbo].[Attendance] AS a
    WHERE @sessiondate BETWEEN [a].[PersonIn]
                       AND     ISNULL([a].[PersonOut], GETDATE())

GO
```

```
(0 row(s) affected)
```

There, that's better, we handle the NULL by substituting the @SessionDate value. No, hang on, still no rows?

Let's try that again ...

```
DECLARE @SessionDate DATETIME
SET @SessionDate = GETDATE()

SELECT empno ,
        [a].[PersonIn] ,
        [a].[PersonOut]
    FROM [dbo].[Attendance] AS a
    WHERE @sessiondate BETWEEN [a].[PersonIn]
                       AND     ISNULL([a].[PersonOut], GETDATE())

GO
```

```
(13 row(s) affected)
```

I kid you not, this is the exact same query, executed against the exact same data only moments after the one above.

What is happening is very subtle and could cause a lot of upset if it occurs in a critical system.

We can see the issue more simply by executing a query based solely on the ISNULL function and passing in different data types.

```
SET NOCOUNT ON
go
DECLARE @Nullval1 SMALLDATETIME
DECLARE @Nullval2 DATETIME
DECLARE @Nullval3 DATE

SELECT ISNULL(@Nullval1, GETDATE()) AS [smalldatetime]
SELECT ISNULL(@Nullval2, GETDATE()) AS [datetime]
SELECT ISNULL(@Nullval3, GETDATE()) AS [date]
```

```
smalldatetime
-----------------------
2012-05-10 22:39:00

datetime
-----------------------
2012-05-10 22:38:55.167

date
----------
2012-05-10
```

We don't assign values to the variables so they are all NULL which means that the ISNULL returns the value if GETDATE(). However you will notice that the value of smalldatetime is greater than datetime. Two things are happening:

1. First, the ISNULL function is having an effect on the data type of GETDATE. This is because it can only return one data type and that is dictated by the data type of the first parameter inside the parenthesis. Where the first and second parameters have different data types ISNULL carries out an implicit conversion between them. There is no error here, it silently changes the value if GETDATE into a SMALLDATETIME data type.
2. Secondly the IMPLICIT CONVERSION of the GETDATE() value from it's default of DATETIME to SMALLDATETIME means that the seconds part of the time is taken and used to either round up or round down the minutes part of the time. Anything from 0 to 29.998s means the minutes stay the same, anything over that and the minutes value is increased by 1.

There is full details of these two effects in the BOL pages here;

- ISNULL function - http://msdn.microsoft.com/en-us/library/ms184325
- SMALLDATETIME data type - http://msdn.microsoft.com/en-us/library/ms182418.aspx.

Going back to our code that gives us different results, we get the different results depending on when in the minute we execute the query but, why?

Well, let's substitute fixed values into the code and see

```
-- when the ss value is < 30 GETDATE() gets rounded down
-- if GETDATE() is a value of '2012-05-10 22:49:22'
-- the full value gets stored in @sessiondate
SELECT empno ,
        [a].[PersonIn] ,
        [a].[PersonOut]
    FROM [dbo].[Attendance] AS a
WHERE '2012-05-10 22:49:22' BETWEEN [a].[PersonIn] AND '2012-05-10 22:49:00'

-- this clearly isnt going to return any rows as '2012-05-10 22:49:22' is outside
-- the range specified by the BETWEEN
--------------------------------------------------------------------------------



-- when the ss value is >= 30 GETDATE() gets rounded up
-- if GETDATE() is a value of '2012-05-10 22:49:32'
-- the full value gets stored in @sessiondate
SELECT empno ,
        [a].[PersonIn] ,
        [a].[PersonOut]
    FROM [dbo].[Attendance] AS a
WHERE '2012-05-10 22:49:22' BETWEEN [a].[PersonIn] AND '2012-05-10 22:50:00'

-- this clearly will return rows as '2012-05-10 22:49:22' is within
-- the range specified by the BETWEEN
--------------------------------------------------------------------------------
```

So, there we have it, an IMPLICIT CONVERSION occurring within a function could easily catch out the unwary SQL developer. Knowing what data types are in use in your databases and how you write code to interact with them is important. It's also important to have a thorough understanding of how functions handle data types and may cause unexpected issues to you if you are not paying attention.

* - Disclosure: I am a Friend of RedGate and therefore have a bias towards their products. There are ways to produce demo data using just TSQL and there are other products available. All are very useful but I chose SQL Data Generator this time.

by fatherjack
Filed Under: SSMS, Tips and Tricks, TSQL, RedGate, DBA

SQL Server source control from Visual Studio

Published Monday, April 30, 2012 1:16 PM

Developers have long since had to context switch between two IDEs, Visual Studio for application code development and SQL Server Management Studio for database development. While this is accepted, especially given the richness of the database development feature set in SSMS, loading a separate tool can seem a little overkill.

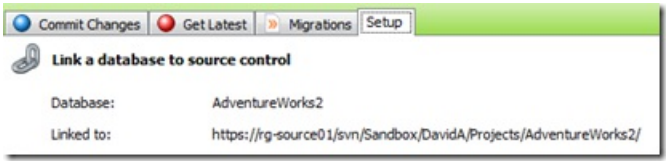
This is where SQL Connect comes in. This is an add-in to Visual Studio that provides a ***connected*** development experience for the SQL Server developer. Connected database development involves modifying a development sandbox database, as opposed to ***offline*** development, where SQL text files are modified independently of the database. One of the main complaints of Data Dude (VS DBPro) is that it enforces the offline approach. This gripe is what SQL Connect addresses.

If you don't already use SQL Source Control, you can get up and running with SQL Connect by adding a new project to your Visual Studio solution as follows:

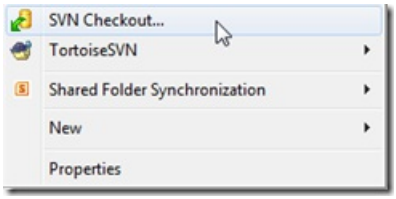
Then choose your existing development database and you're ready to go.

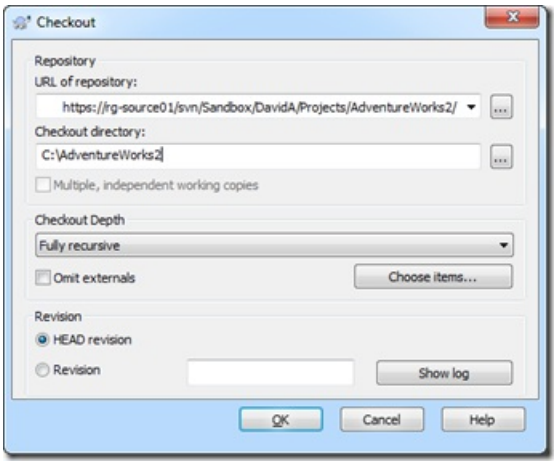
If you already use SQL Source Control, you will need to link SQL Connect to your existing database scripts folder repository, so SQL Connect and SQL Source Control can be used collaboratively (note that SQL Source Control v.3.0.9.18 or later is required).

Locate the repository (this can be found in the Setup tab in SQL Source Control).



.and create a working folder for it (here I'm using TortoiseSVN).

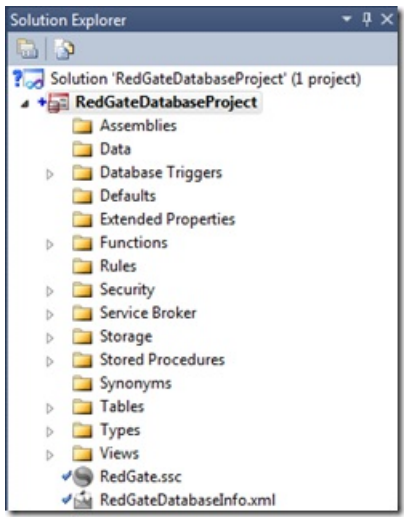




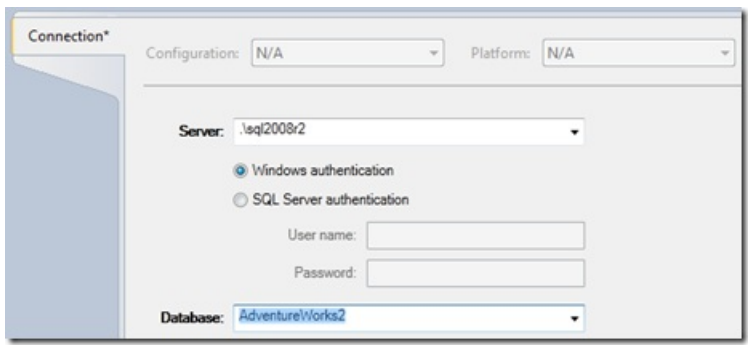
Back in Visual Studio, locate the SQL Connect panel (in the View menu if it hasn't auto loaded) and select Import SQL Source Control project

Locate your working folder and click Import.

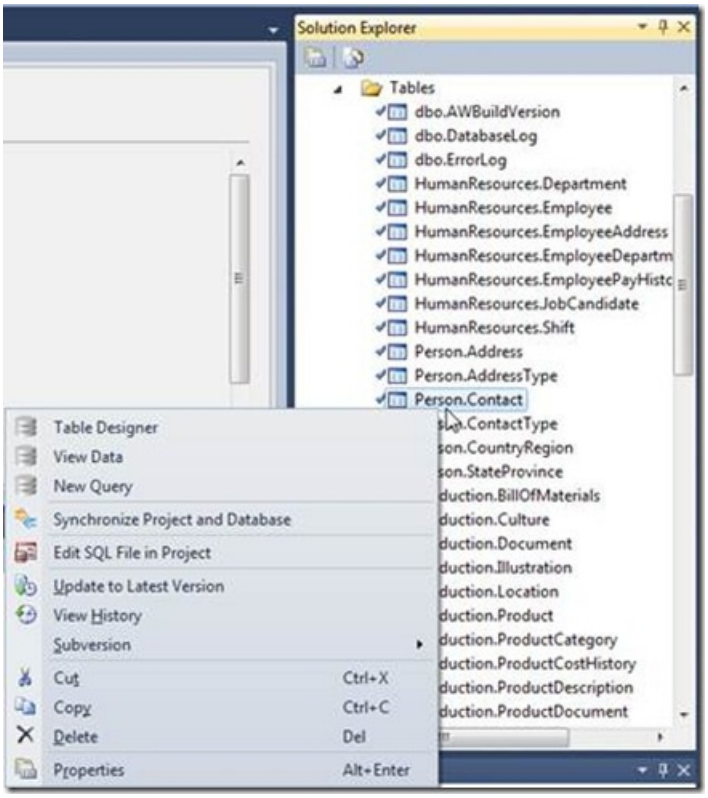
This creates a Red Gate database project under your solution:



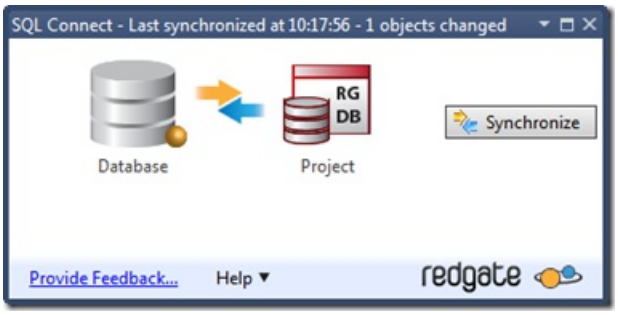
From here you can modify your development database, and manage your changes in source control. To associate your development database with the project, right click on the project node, select Properties, set the database and **Save**.



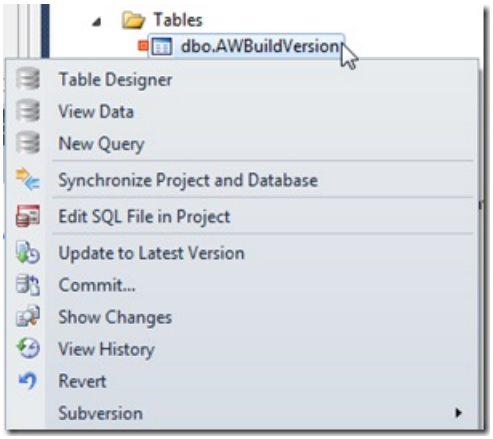
Now you're ready to make some changes. Locate the object you'd like to modify in the Solution Explorer, and double click it to invoke a query window or table designer. You also have the option to edit the creation SQL directly using Edit SQL File in Project.



Keeping the development database and Visual Studio project in sync is as easy as clicking on a button.



One you've made your change, you can use whichever mechanism you choose to commit to source control. Here I'm using the free open-source AnkhSVN to integrate Subversion with Visual Studio.



Maintaining your database in a Visual Studio solution means that you can commit database changes and application code changes in the same changeset. This is desirable if you have continuous integration set up as you want to ensure that all files related to a change are committed atomically, so you avoid an interim "broken build".

More discussion on SQL Connect and its benefits can be found in the following article on Simple Talk: No More Disconnected SQL Development in Visual Studio

The SQL Connect project team is currently assessing the backlog for the next development effort, and they'd appreciate your feature suggestions, as well as your votes on their suggestions site:

http://redgate.uservoice.com/forums/140800-sql-connect-for-visual-studio-

A 28-day free trial of SQL Connect is available from the Red Gate website.

Technorati Tags: SQL Server
by David Atkinson

Metrics - A little knowledge can be a dangerous thing (or 'Why you're not clever enough to interpret metrics data')

Published Thursday, May 03, 2012 3:23 PM

At RedGate Software, I work on a .NET obfuscatorÂ called SmartAssembly.Â Various features of it use a database to store various things (exception reports, name-mappings, etc.) The user is given the option of using either a SQL-Server database (which requires them to have Microsoft SQL Server), or a Microsoft Access MDB file (which requires nothing). MDB is the default option, but power-users soon switch to using a SQL Server database because it offers better performance and data-sharing.

In the fashionable spirit of optimization and metrics, an obvious product-management question is 'Which is the most popular? SQL Server or MDB?'

We've collected data about this fact, using our 'Feature-Usage-Reporting' technology (available as part of SmartAssembly) and more recently our 'Application Metrics' technology:

Parameter	Number of users	% of total users	Number of sessions	Number of usages
SQL Server	28	19.0	8115	8115
MDB	114	77.6	1449	1449

(*As a disclaimer, please note than SmartAssembly has far more than 132 users . This data is just a selection of one build)*

So, it would appear that SQL-Server is used by fewer users, but more often. Great.

But here's why these numbers are useless to me:

Only the original developers understand the data

What does a single 'usage' of 'MDB' mean? Does this happen once per run? Once per option change? On clicking the 'Obfuscate Now' button? When running the command-line version or just from the UI version? Each question could skew the data 10-fold either way, and the answers only known by the developer that instrumented the application in the first place. In other words, only the original developer can interpret the data - product-managers cannot interpret the data unaided.

Most of the data is from uninterested users

About half of people who download and run a free-trial from the internet quit it almost immediately. Only a small fraction use it sufficiently to make informed choices. Since the MDB option is the default one, we don't know how many of those 114 were people CHOOSING to use the MDB, or how many were JUST HAPPENING to use this MDB default for their 20-second trial.

This is a problem we see across all our metrics: Are people are using X because it's the default or are they using X because they *want* to use X? We need to segment the data further - asking what percentage of each percentage meet our criteria for an 'established user' or 'informed user'. You end up spending hours writing sophisticated and dubious SQL queries to segment the data further. Not fun.

You can't find out *why* they used this feature

Metrics can answer the when and what, but not the why. Why did people use feature X? If you're anything like me, you often click on random buttons in unfamiliar applications just to explore the feature-set. If we listened uncritically to metrics at RedGate, we would eliminate the most-important and more-complex features which people actually buy the software for, leaving just big buttons on the main page and the About-Box.

"Ah, that's interesting!" rather than "Ah, that's actionable!"

People do love data. Did you know you eat 1201 chickens in a lifetime? But just 4 cows? Interesting, but useless. Often metrics give you a nice number: '5.8% of users have 3 or more monitors' . But unless the statistic is both SUPRISING and ACTIONABLE, it's useless.

Most metrics are collected, reviewed with lots of cooing. and then forgotten. Unless a piece-of-data could change things, it's useless collecting it.

People get obsessed with significance levels

The first things that lots of people do with this data is do a t-test to get a significance level (*"Hey! We know with 99.64% confidence that people prefer SQL Server to MDBs!"*) Believe me: other causes of error/misinterpretation in your data are FAR more significant than your t-test could ever comprehend.

Confirmation bias prevents objectivity

If the data appears to match our instinct, we feel satisfied and move on. If it doesn't, we suspect the data and dig deeper, plummeting down a rabbit-hole of segmentation and filtering until we give-up and move-on. Data is only useful if it can change our preconceptions. Do you trust this dodgy data more than your own understanding, knowledge and intelligence?Â I don't.

There's always multiple plausible ways to interpret/action any data

Let's say we segment the above data, and get this data:

Post-trial users (i.e. those using a paid version after the 14-day free-trial is over):

Parameter	Number of users	% of total users	Number of sessions	Number of usages
SQL Server	13	9.0	1115	1115
MDB	5	4.2	449	449

Trial users:

Parameter	Number of users	% of total users	Number of sessions	Number of usages
SQL Server	15	10.0	7000	7000
MDB	114	77.6	1000	1000

How do you interpret this data? It's one of:

1. Mostly SQL Server users buy our software. People who can't afford SQL Server tend to be unable to afford or unwilling to buy our software. Therefore, ditch MDB-support.
2. Our MDB support is so poor and buggy that our massive MDB user-base doesn't buy it.Â Therefore, spend loads of money improving it, and think about ditching SQL-Server support.
3. People 'graduate' naturally from MDB to SQL Server as they use the software more. Things are fine the way they are.
4. We're marketing the tool wrong. The large number of MDB users represent uninformed downloaders. Tell marketing to aggressively target SQL Server users.

To choose an interpretation you need to segment again. And again. And again, and again.

Opting-out is correlated with feature-usage

Metrics tends to be opt-in. This skews the data even further. Between 5% and 30% of people choose to opt-in to metrics (often called 'customer improvement program' or something like that). Casual trial-users who are uninterested in your product or company are less likely to opt-in. This group is probably also likely to be MDB users. How much does this skew your data by? Who knows?

It's not all doom and gloom.

There are some things metrics can answer well.

1. Environment facts. How many people have 3 monitors? Have Windows 7? Have .NET 4 installed? Have Japanese Windows?
2. Minor optimizations.Â Is the text-box big enough for average user-input?
3. Performance data. How long does our app take to start? How many databases does the average user have on their server?

As you can see, questions about who-the-user-is rather than what-the-user-does are easier to answer and action.

Conclusion

1. Use SmartAssembly. If not for the metrics (called 'Feature-Usage-Reporting'), then at least for the obfuscation/error-reporting.
2. Data raises more questions than it answers.
3. Questions about environment are the easiest to answer.

by Jason Crease

Inside Red Gate - Ricky Leeks

Published Friday, May 04, 2012 3:30 PM

So, one of our profilers has a problem. Red Gate produces two .NET profilers - ANTS Performance Profiler (APP) and ANTS Memory Profiler (AMP). Both products help .NET developers solve problems they are virtually guaranteed to encounter at some point in their careers - slow code, and high memory usage, respectively.

Everyone understands slow code - the symptoms are very obvious (an operation takes 2 hours when it should take 10 seconds), you know when you've solved it (the same operation now takes 15 seconds), and everyone understands how you can use a profiler like APP to help solve your particular problem. High memory usage is a much more subtle and misunderstood concept.

How can .NET have memory leaks?

The garbage collector, and how the CLR uses and frees memory, is one of the most misunderstood concepts in .NET. There's hundreds of blog posts out there covering various aspects of the GC and .NET memory, some of them helpful, some of them confusing, and some of them are just plain wrong. There's a lot of misconceptions out there. And, if you have got an application that uses far too much memory, it can be hard to wade through all the contradictory information available to even get an idea as to what's going on, let alone trying to solve it.

That's where a memory profiler, like AMP, comes into play. Unfortunately, that's not the end of the issue. .NET memory management is a large, complicated, and misunderstood problem. Even armed with a profiler, you need to understand what .NET is doing with your objects, how it processes them, and how it frees them, to be able to use the profiler effectively to solve your particular problem.

And that's what's wrong with AMP - even with all the thought, designs, UX sessions, and research we've put into AMP itself, some users simply don't have the knowledge required to be able to understand what AMP is telling them about how their application uses memory, and so they have problems understanding & solving their memory problem.

Ricky Leeks

This is where Ricky Leeks comes in. Created by one of the many...colourful...people in Red Gate, he headlines and promotes several tutorials, pages, and articles all with information on how .NET memory management actually works, with the goal to help educate developers on .NET memory management. And educating us all on how far you can push various vegetable-based puns. This, in turn, not only helps them understand and solve any memory issues they may be having, but helps them proactively code against such memory issues in their existing code.

Ricky's latest outing is an interview on .NET Rocks, providing information on the Top 5 .NET Memory Management Gotchas, along with information on a free ebook on .NET Memory Management. Don't worry, there's loads more vegetable-based jokes where those came from...

by Simon Cooper
Filed Under: Inside Red Gate