Revision	Description
8/9/2011	Original
9/5/2011	Mention .xsd for intellisense, additional info related to data gen (new features)

SHCommand – Quality QA Data, Made Easy

Introduction

SQL-Hero has included the SHCommand command line tool (formerly known as "SHDBSync") for a number of releases now but some recent work has given it some new and extremely powerful capabilities. Rather than go through these in a reference guide format (which does exist in the document entitled SQL-Hero Command Line Tool), this paper approaches things from a scenario-based perspective. We're going to look at everything that's necessary to take data from a production database, scrambling sensitive data it may contain, then restoring it into a QA environment - while preserving on-going changes in schema that are present in the existing QA database. Additionally, e-mails are produced at the end, letting relevant developers know that the process has completed. What we arrive at is a QA database that's unchanged with respect to schema, but contains quality, production-like data - while not exposing sensitive data outside the production environment. The entire process is automated and can be launched by a simple, single command-line invocation of SHCommand.exe.

What's the real value in demonstrating this process? For one thing, it's not uncommon for organizations to have developers using development and QA databases that are poor reflections of reality: reality being what's in production. This isn't always a problem, but it can be especially bad when it's impractical to rebuild an entire development database from scratch, such as when we have hundreds of tables, many complex relationships, and potentially large volumes of data. (Having large data volumes can be very valuable as discussed in the slide deck Managing Test Data and Stress Testing Your SQL Applications.) It can also be a problem for users trying to do testing when their QA environment falls so far out-of-date with respect to production that they can no longer test effectively (especially when important reference data is no longer in sync). Another problem is based on an observation from the industry: developers will gravitate to where there's good data quality, and we've seen this lead to things like developers doing development work in QA regions instead of development regions - clearly not an ideal situation!

The process described in this document attempts to remove hurdles that many have thrown up to the notion of using production data in QA regions, including:

- **Security:** Many organizations would never want their production data to be visible to developers under any circumstances. However, most sensitive production data can be scrambled while retaining important relationships and cardinalities, yet obscuring attributes that are deemed most sensitive such as Social Security numbers, credit card numbers, customer names, customer addresses, and so on. In fact, SQL-Hero has a sophisticated data generation

and scrambling engine that will be covered in a different whitepaper. This scrambling process is integrated with SHCommand and is a central aspect of this use case. A second aspect of security is that some organizations will have separation of production and QA environments such that we may need to rely on things like FTP to move files between them.

- **"Lost updates":** In theory a QA database should be in sync with a particular revision level as captured in a source control system, meaning it could be replicated if there was a need to effectively "roll forward" from an older version (such as what's in production). Although it sounds easy in theory, in practice it can be easier to simply carry over schema differences that exist in the current version of QA at the time the QA region gets its data refreshed. This implies that the actively tested code-base in QA is not changed by restoring production data.

- **Ease:** This process sounds like a lot of steps – and it can be – but by reducing the complexity to the one-time set-up of the script to make it all work, it can be made repeatable and painless. If you're a DBA, you'll appreciate being able to launch the process and simply check for a success or failure email. Another aspect of "ease" is we want any process to be as non-disruptive as possible for developers: ideally a new version of the QA database "just appears" and we don't want to tell them, "well, you'll see the data first and we'll get you your schema updates later." Additionally, if there's a problem in the process we don't want things visible to the outside world left in an incomplete or corrupt state. In other words, it would be nice if the whole process at least *seems* atomic to outsiders, with as nearly "instant cut-over" as possible.

The solution we'll look at in detail addresses all of these points definitively and scales to even more complex processes your organization may face. New use cases are being supported with on-going releases, so check the document library regularly at http://www.codexframework.com/library.

The Pieces Involved

Let's look at the various pieces involved in making this use case work. First, you must have at least the Developer Edition of SQL-Hero installed on the machines which actually will be running the SHCommand tool. What SHCommand "runs" is a series of tasks that are specified in a file (by convention, with a .shcommand extension) - which is really just an XML file that conforms to a particular schema. Second, you must have at least one Advanced (or Enterprise) Edition of SQL-Hero installed somewhere, along with the server components. Why? The server components host global connection information which is used throughout the example presented – instead of connection strings, we use aliases which hide actual connection details and can be configured globally or by user.

A picture (Figure 1) helps show how the full installation of SQL-Hero relates to what we're trying to accomplish. The machine marked (1) represents where we might choose to run the SHCommand tool. It doesn't necessarily need to be the actual database server which hosts the Production or QA databases (2), but it does need sufficient rights to connect to these machines using UNC naming to get at backup files, for example. Another option is to run the tool directly on the database servers (2), which may be

practical if this process is being launched by a privileged DBA. This also implies you've installed at least the SQL-Hero client tools on the database servers. The command tool references the SQL-Hero application server (3), and this must be reachable regardless of where the tool is run from. It should be noted that all SQL-Hero components can be run from a single machine if desired, and the easiest way to install these is using the Express Installation option that's available when installing the Advanced Edition. More details about installation can be found in the whitepaper Installing SQL-Hero.

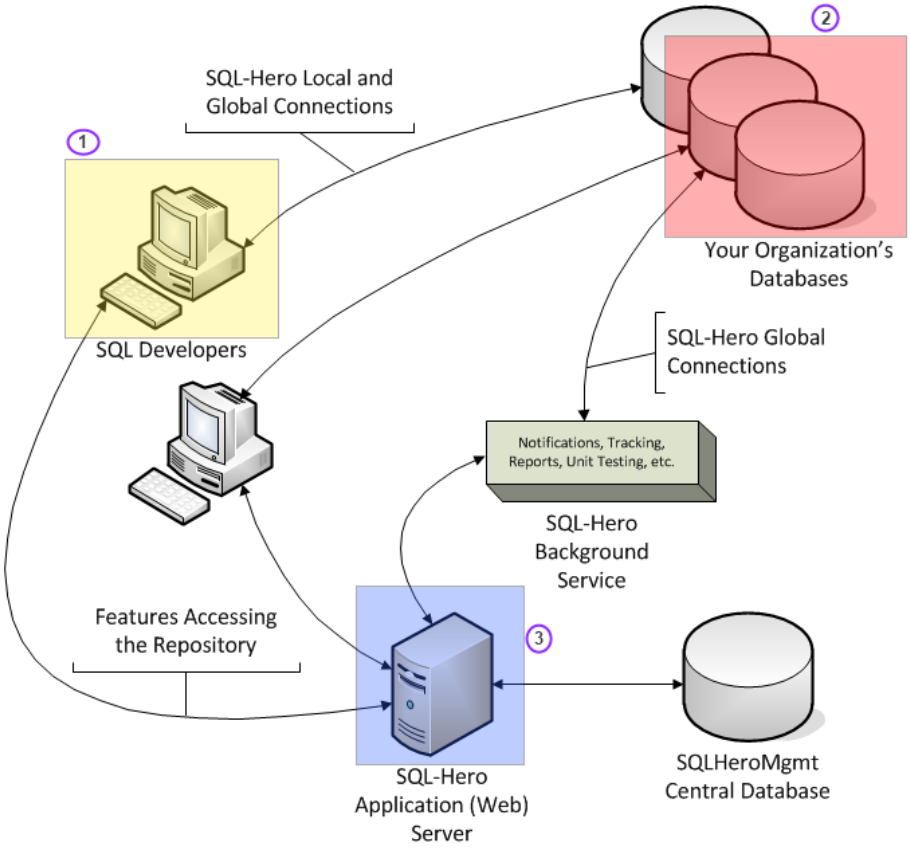


Figure 1 - Architectural elements part of this solution

Another consideration addressed in the scenario is whether it's practical to run all steps on one machine or whether it makes sense to transfer control to another machine to perform some steps. Why might this be useful? There can be issues with credentials: perhaps related to file system access, where we don't want to rely on UNC naming. SHCommand allows us to "hand-off" execution to another instance of the tool running on the machine that's running the SQL-Hero Background Service (shown in Figure 1). Your particular situation may not require this, but it's something to be aware of as a capability, and we will rely on it later in the scenario to transfer control from the production database server to the QA database server. In our example, we kick the process off by invoking:

SHCommand OverlayExampleProd.shcommand

Also note that we can double-click on the OverlayExampleProd.shcommand in Windows Explorer and it will be invoked as well since there's a file association for the shcommand file type.

We assume this is done on the production database machine. We also assume we have a number of global connections set up in SQL-Hero before using this script. These include:

Alias	Connection Definition
DemoScript-ProdRemote	Server = Production, Database = Production, Credentials allow for server operations such as backup / restore while running on production server
DemoScript-ProdMasterRemote	Server = Production, Database = master, Credentials allow for server operations such as backup / restore while running on production server
DemoScript-QA	Server = QA, Database = QA, Standard credentials
DemoScript-QAMaster	Server = QA, Database = master, Credentials needed to support backup / restore, etc.
DemoScript-QANewRemote	Server = Production, Database = QANew – this is the newly restored production database (on the production server) prior to it having its data scrambled; this as a temporary database that lasts only long enough to scramble sensitive data
DemoScript-QANew	Server = QA, Database = QANew – this is the scrambled production database, restored to the QA server; this as a temporary database that's eventually renamed to become the final QA database
DemoScript-QAOld	Server = QA, Database = QAOld – this is the original QA database, prior to this process being run, retained for reference

In building this scenario for testing, we started with all databases on a single machine and introduced a second machine to more accurately test file copies and such. The "Remote" aspect of naming became the second test machine which houses the potentially more physically and/or logically isolated production server.

The Script

Let's break down the .shcommand file, section by section, to see how we go from two standalone databases – Production and QA (see Figure 2) – to the final product which is the existing production database, untouched, and two versions of QA: "QAOld" (which is the current QA prior to running this process), and "QA" (which will be the newly restored QA environment).

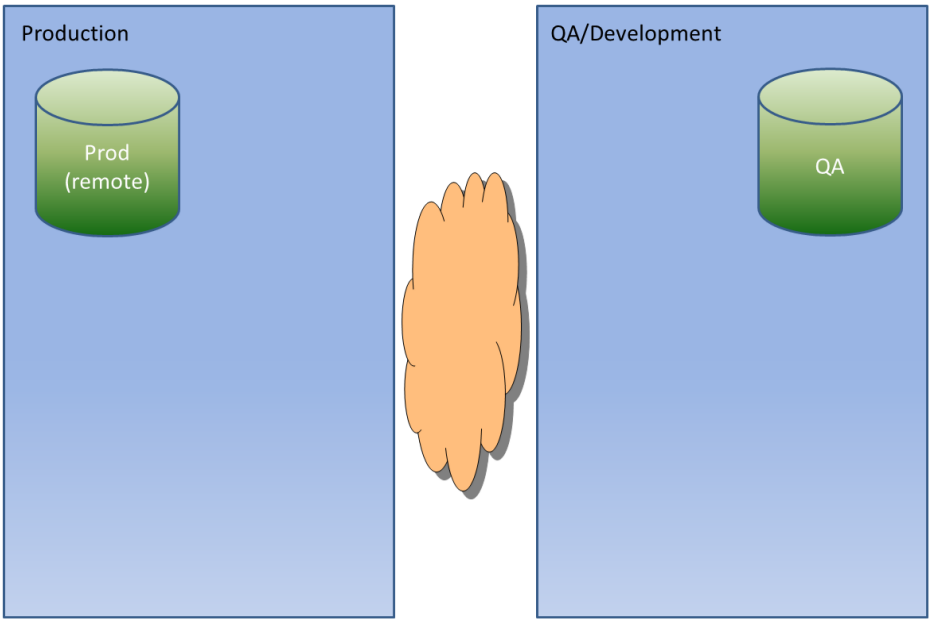


Figure 2 - Before starting

Looking at the configuration file we pass to SHCommand.exe (OverlayExampleProd.shcommand), under the TASKS root element, the SERVER child element points at the desired SQL-Hero application server. To use HTTP port 80, you can just supply the machine name, as is done here:

```
<TASKS xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="\program files
(x86)\sqlhero\SHCommandSchema.xsd">
  <SERVER>codex07</SERVER>
  <FAIL_CATCH>FailStep</FAIL_CATCH>
  <DEFAULT_FILE_TARGET>OverlayExampleOutput.log</DEFAULT_FILE_TARGET>
```

Note that by referencing the SHCommandSchema.xsd file as we've done here, we can get a measure of IntelliSense assistance if: a) we use Visual Studio as the file editor, b) we rename the file to use a .xml extension (at least for the duration of the time we're constructing it). Although it's nice to see a list of possible elements without having to consult external documentation, it doesn't work perfectly since IntelliSense will list elements that are not necessarily valid in a given context – however, when combined with some understanding of how the different tasks actually work, it can be a big time saver.

The FAIL_CATCH element identifies the TASK element which will be jumped to in the event of a task failure: in this case, we will be sending a failure e-mail to a fixed distribution list (probably the DBA who manages this process):

```
  <!-- email a fixed dist list based on ultimate failure (with details of failure) -->
  <TASK action="EMAIL" id="FailStep" stop="true">
    <MAILSERVER>codex06</MAILSERVER>
    <TO>admin@codexframework.com</TO>
    <FROM>SHCommand@SH.Null</FROM>
    <SUBJECT>Command line tool:  Error</SUBJECT>
    <BODY>The command line tool completed with an error.  Attaching output.</BODY>
    <ATTACHMENT>OverlayExampleOutput.log</ATTACHMENT>
  </TASK>
```

The DEFAULT_FILE_TARGET element lets you name a file which will receive a copy of all the console output that is shown by the tool. (Some additional fine tuning of output is possible, but this element acts as a global setting across all task types.)

Let's examine the first meaningful steps in the process. The first task type is SQLSCRIPT which is used to perform the backup of the production database to a backup file located on the production server. By "production environment" here we mean it could be a server or servers completely isolated from ordinary developers. We've structured the script in a way that assumes some production DBA has the rights necessary to initiate the running of SHCommand. We further assume that the SQL-Hero global connection called "DemoScript-ProdRemote" has been set up with enough permission to allow the DBA to connect and perform a backup. (One could even make it only visible to this one person within SQL-Hero.) The second task is another T-SQL script that performs a restore using the backup made in the first step. The restored database name is DemoScriptQANew, and we're assuming we can do this on the same server as production (although perhaps a different SQL instance) – the reality is we may wish to restore it to a different physical (or virtual) server entirely. This example could be enhanced to include a file copy step in between the backup and restore if this was a necessity.

```
  <!-- execute arbitrary SQL, input acting as a template so we have access to some especially relevant
markup tags if we can use them -->
  <TASK action="SQLSCRIPT" level="ErrorOnlyDetail">
    <IN>DemoScript-ProdRemote</IN>
    <TIMEOUT>3600</TIMEOUT>
    <COMMAND>
      <![CDATA[
BACKUP DATABASE [DemoScriptProd] TO  DISK = N'C:\Temp\DemoScriptProd.bak' WITH FORMAT, INIT,  NAME =
N'DemoScriptProd-Full Database Backup', SKIP, NOREWIND, NOUNLOAD,  STATS = 10, COMPRESSION
]]>
    </COMMAND>
  </TASK>

  <!-- restore as ProdCopy -->
  <TASK action="SQLSCRIPT" level="ErrorOnlyDetail">
    <IN>DemoScript-ProdMasterRemote</IN>
    <TIMEOUT>3600</TIMEOUT>
    <COMMAND>
      <![CDATA[
RESTORE DATABASE [DemoScriptQANew] FROM  DISK = N'C:\Temp\DemoScriptProd.bak' WITH  FILE = 1,  MOVE
N'DemoScriptProd' TO N'C:\Temp\DemoScriptQANew.mdf',  MOVE N'DemoScriptProd_log' TO
N'C:\Temp\DemoScriptQANew_log.ldf',  KEEP_REPLICATION,  NOUNLOAD,  REPLACE,  STATS = 10
GO
]]>
    </COMMAND>
  </TASK>
```

After these tasks are run, we have the state illustrated in Figure 3.

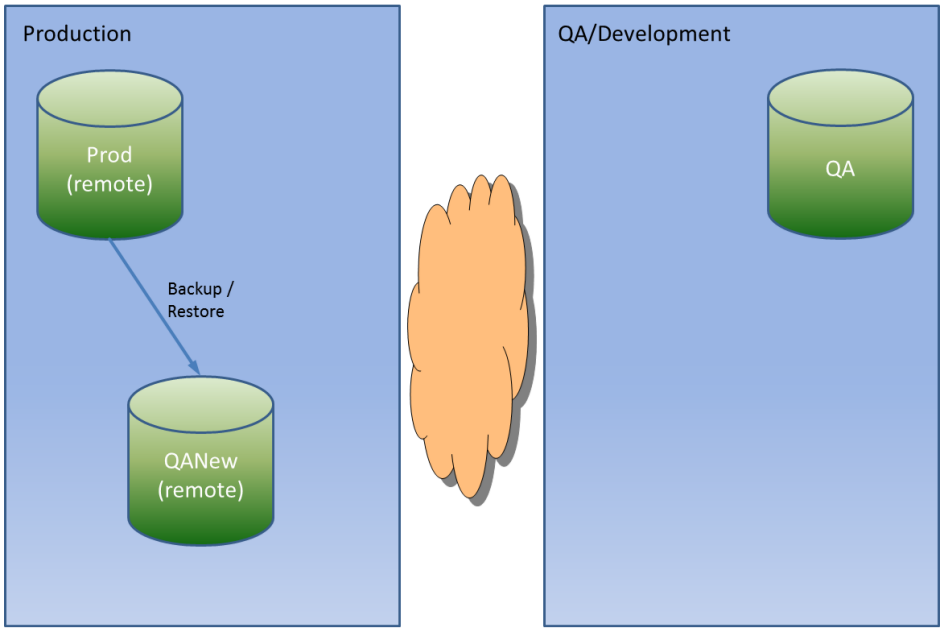
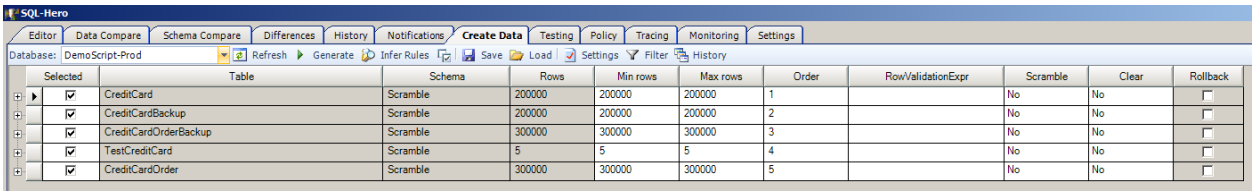


Figure 3 - Backup and Restore of Production

The next step is the CREATEDATA task. In our case, we're not creating new data but are using the "Generate Data" engine that's part of SQL-Hero to transform what we consider to be data too sensitive to release outside the production environment. The task details below name a second configuration file, DemoQANewScramble.gdcfg:

```
  <!-- scramble some data present in the restored database that we may consider too sensitive to share
with developers in the restored QA region -->
  <TASK action="CREATEDATA" level="Detail">
    <FILE>DemoQANewScramble.gdcfg</FILE>
    <IN>DemoScript-QANewRemote</IN>
  </TASK>
```

This file can be most easily built using the SQL-Hero client tool. When we switch to the "Create Data" tab and pick the source database that applies, we get a list of all tables:



Selected	Table	Schema	Rows	Min rows	Max rows	Order	RowValidationExpr	Scramble	Clear	Rollback
☑	CreditCard	Scramble	200000	200000	200000	1		No	No	☐
☑	CreditCardBackup	Scramble	200000	200000	200000	2		No	No	☐
☑	CreditCardOrderBackup	Scramble	300000	300000	300000	3		No	No	☐
☑	TestCreditCard	Scramble	5	5	5	4		No	No	☐
☑	CreditCardOrder	Scramble	300000	300000	300000	5		No	No	☐

In our case, let's assume the CreditCard table contains the sensitive data we don't want to allow developers to see. Here's a sample of the production data:

```
SELECT TOP 20 * FROM Scramble.CreditCard
```

	CreditCardID	CreditCardNumber	CardholderName	ExpiryMonth	ExpiryYear
1	1	0xC69EF289532DE09AEE7115FCA9403DC3D92CB2470F40644009	0x1B5DFBB3C9204BD388EBC6F7EF8EF	10	2014
2	2	0xD931EF8121435D5F039B862948D252B6FF70F2A1B25402C8353	0xF4255449246C56E9365138E6EFB742	1	2020
3	3	0x0C286D201BFEBA49082EDFC6D7CFAD4346649F736A35FA3ED	0x48E9622DC6E965B3F389B3D3A60AE	11	2016

So although this looks fairly secure because it's using columnar encryption, we still need to be able to write code that will eventually decrypt this into plain-text, such as for example leading to:

```
EXECUTE [dbo].[up_CreditCardOrderList]
        @MaxRow = @MaxRow
```

	OrderDate	OrderAmount	CreditCardNumber	CardholderName
1	11/12/2010 00:00:00.000	27.5200	5532324848536859	MIKAYLA COOK
2	3/25/2010 00:00:00.000	572.2500	2502030043746702	JOVAN ROY
3	2/17/2011 00:00:00.000	657.7100	8881649811794033	SHANTEL B. BYRD
4	1/10/2011 00:00:00.000	968.5100	8716605747875847	KIMBERLEE PENNINGTON
5	4/6/2010 00:00:00.000	741.6100	5872876062094668	ALONSO DELGADO

What we'd really like is for data in QA to have all credit card numbers changed to be from a fixed set of five "test card numbers" – perhaps given to us by our payment gateway provider as valid for testing purposes. We store these card numbers in unencryped form in a table called TestCreditCard:

```
SELECT * FROM Scramble.TestCreditCard
```

	TestCreditCardID	CreditCardNumber	CardholderName	ExpiryMonth	ExpiryYear
1	1	9999888877776666	TEST CARDHOLDER	12	2016
2	2	8888777766665555	CARDHOLDER TEST	1	2015
3	3	7777666655554444	TESTER CARD	6	2014
4	4	6666555544443333	CARD TEST	3	2013
5	5	5555444433332222	TESTING CARD	9	2012

We assume that although we have thousands of actual credit card records, it's acceptable to replace these "in place" with any one of the five test numbers. (If we had code which groups by the actual card number as opposed to the CreidtCardID, this may result in funny looking results – just five credit cards in the entire system but with many thousands of orders each - but most functions should still *work* and we might end up only observing this grouping in things like reports, for example.)

To facilitate this, we'll indicate that the CreditCard table should be "scrambled" (as opposed to generating new data for it):

Selected	Table	Schema	Rows	Min rows	Max rows	Order	RowValidationExpr	Scramble	Clear	Rollback
☑	CreditCard	Scramble	200000	200000	200000	1		Yes	No	☐

Column	Type	Length	PK	FK	Unique	Order	Rule	Rule parameters	Null Pct
CreditCardID	int		☑	☐	☑	0	Retain existing value (no scramble)		
CreditCardNumber	varbinary	(128)	☐	☐	☐	1	Lookup-based	Method=SQL Query; ExistingDistribution=False; XMLFile	
CardholderName	varbinary	(128)	☐	☐	☐	2	Lookup-based	Method=SQL Query; ExistingDistribution=False; XMLFile	
ExpiryMonth	int		☐	☐	☐	3	Lookup-based	Method=SQL Query; ExistingDistribution=False; XMLFile	
ExpiryYear	int		☐	☐	☐	4	Lookup-based	Method=SQL Query; ExistingDistribution=False; XMLFile	

Furthermore, we'll leave the CreditCardID alone (the primary key which is used in relationships elsewhere – it's important to keep it the same!), and the other columns will be transformed using

lookup-based rules. Since these attributes are dependent on each other, the order of computation is going to be important here:

Selected	Table	Schema	Rows	Min rows	Max rows	Order	RowValidationExpr	Scramble	Clear	Rollback
☑	CreditCard	Scramble	200000	200000	200000	1		Yes	No	☐

Column	Type	Length	PK	FK	Unique	Order	Rule	Rule parameters	Null Pct
CreditCardID	int		☑	☐	☑	0	Retain existing value (no scramble)		
CreditCardNumber	varbinary	(128)	☐	☐	☐	1	Lookup-based	Method=SQL Query; ExistingDistribution=False; XMLFile	
CardholderName	varbinary	(128)	☐	☐	☐	2	Lookup-based	Method=SQL Query; ExistingDistribution=False; XMLFile	
ExpiryMonth	int		☐	☐	☐	3	Lookup-based	Method=SQL Query; ExistingDistribution=False; XMLFile	
ExpiryYear	int		☐	☐	☐	4	Lookup-based	Method=SQL Query; ExistingDistribution=False; XMLFile	

For the CreditCardNumber field, our lookup rule could look like this:

Generate Data Parameters...

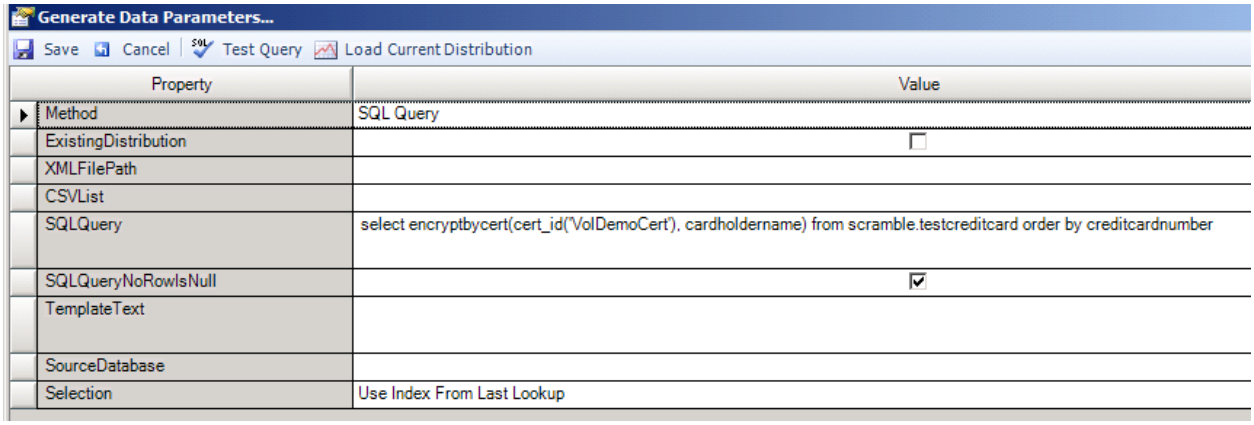
Save | Cancel | Test Query | Load Current Distribution

Property	Value
Method	SQL Query
ExistingDistribution	☐
XMLFilePath	
CSVList	
SQLQuery	select encryptbycert(cert_id('VolDemoCert'), creditcardnumber) from scramble.testcreditcard order by creditcardnumber
SQLQueryNoRowIsNull	☑
TemplateText	
SourceDatabase	
Selection	Random

For the CardholderName, we need to keep the name in sync with the number, so the lookup rule SQLQuery could be for that:

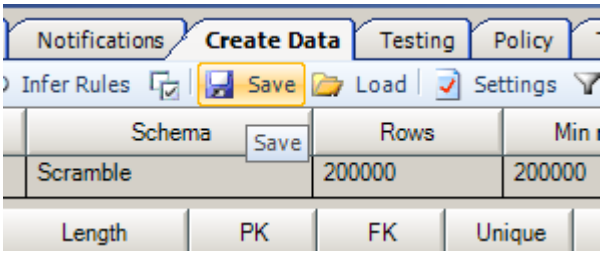
```
select encryptbycert(cert_id('VolDemoCert'), cardholdername) from
scramble.testcreditcard where creditcardnumber = convert(varchar(20),
decryptbycert(cert_id('VolDemoCert'), @CreditCardNumber, N'VolDem0'))
```

Note that partial transformation results *within the row that is being transformed* is available using "@" in front of the column name for columns with smaller order sequence. This is an important capability that addresses a wide range of use cases, not just the one presented here. There's a second way to even more simply achieve the same result: use the "Use Index From Last Lookup" selection rule. If we do this, then the ORDER BY clause shown above is critical since we need to ensure all lookups see *consistent* result set ordering. (It may not be obvious, but the order of a result set can be different depending on the query, presence of indexes, unique constraints, etc.)



Throughout this whole process of setting up scrambling, we assert that everything is relatively straightforward: there's no need to write custom .NET code or futz with things to achieve desired results - it's all laid out clearly in the main grid. A different whitepaper will be covering much greater detail around the data generation engine, including how you can create very realistic data, beyond that which Visual Studio (and many, many other products) can achieve.

Once we've specified all the transformation rules for the data scramble operation, we can save the configuration to a .gdcfg file – the DemoQANewScramble.gdcfg file mentioned above:



Note that SHCommand offers an alternative to having a separate file that needs to be distributed along with the .shcommand file: you can embed text files within the .shcommand file itself. We can achieve this using the EMBED_FILE element, such as illustrated here:

```
  <!-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -->
  <!-- allows us to include what would otherwise have to be external files within this one single file -
creates these as temporary files which are removed after run completes -->
  <!-- the contents of this embedded file come from a .gdcfg file that was saved from the generate data
SQL-Hero tool window -->
  <EMBED_FILE filename="DemoQANewScramble.gdcfg">
    <![CDATA[
<ROOT>
  <TABLE selected="True" name="CreditCard" owner="Scramble" maxrow="0" minrow="0" order="1" scramble="1"
clear="0" rollback="False">
…
]]>
  </EMBED_FILE>
```

After the CREATEDATA task completes, we have the situation shown in Figure 4:

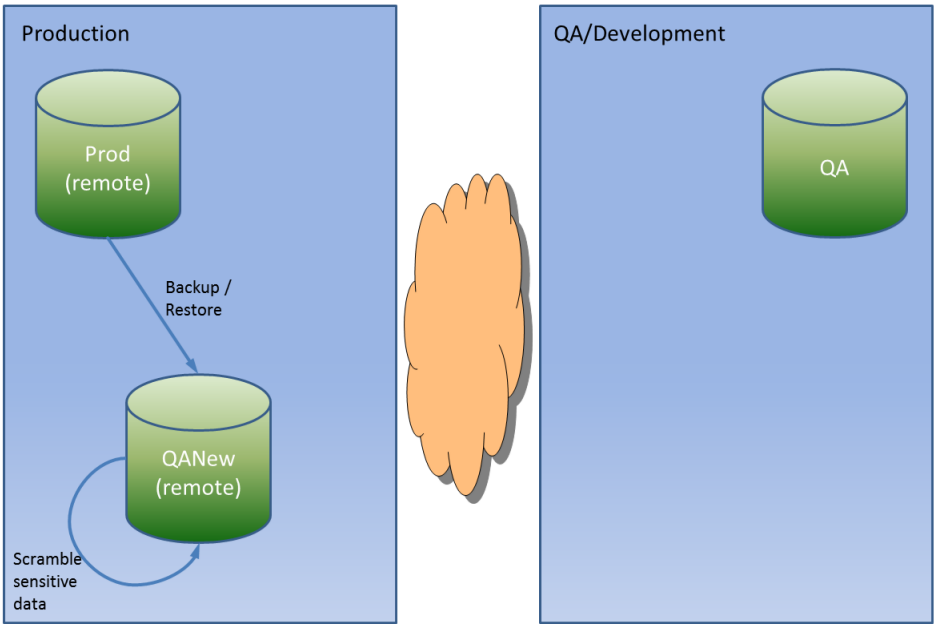


Figure 4 - Scramble Sensitive Data in Restored Copy

If the data transformation step fails for any reason, recall that the process will halt and transfer control to the email step which lets our DBA know there was a problem. However, if we want to be especially paranoid about possibly moving the production database to QA without the scrambling step having "worked", we can use an additional SQLSCRIPT step to validate the scrambling:

```
  <!-- to be extra cautious, we can perform a validation that the scramble did work -->
  <TASK action="SQLSCRIPT" level="ErrorOnlyDetail">
    <IN>DemoScript-QANewRemote</IN>
    <TIMEOUT>300</TIMEOUT>
    <COMMAND>
      <![CDATA[
IF EXISTS
  (SELECT 0
  FROM (SELECT TOP 500 CreditCardNumber FROM Scramble.CreditCard) cc
    LEFT OUTER JOIN Scramble.TestCreditCard tc
    ON tc.CreditCardNumber = convert(varchar(20), decryptbycert(cert_id('VolDemoCert'),
cc.CreditCardNumber, N'VolDem0'))
  WHERE
    tc.TestCreditCardID IS NULL)

  RAISERROR('At least 1 credit card number was left unscrambled or the process did not work as expected.',
16, 1)
]]>
    </COMMAND>
  </TASK>
```

This T-SQL will force a failure if a credit card record exists that is not in the TestCreditCard table (only 500 rows are sampled, but this could be changed to be all records at the expense of performance). This step is entirely optional.

The next step is to use another SQLSCRIPT task to create a backup of the transformed QANew database:

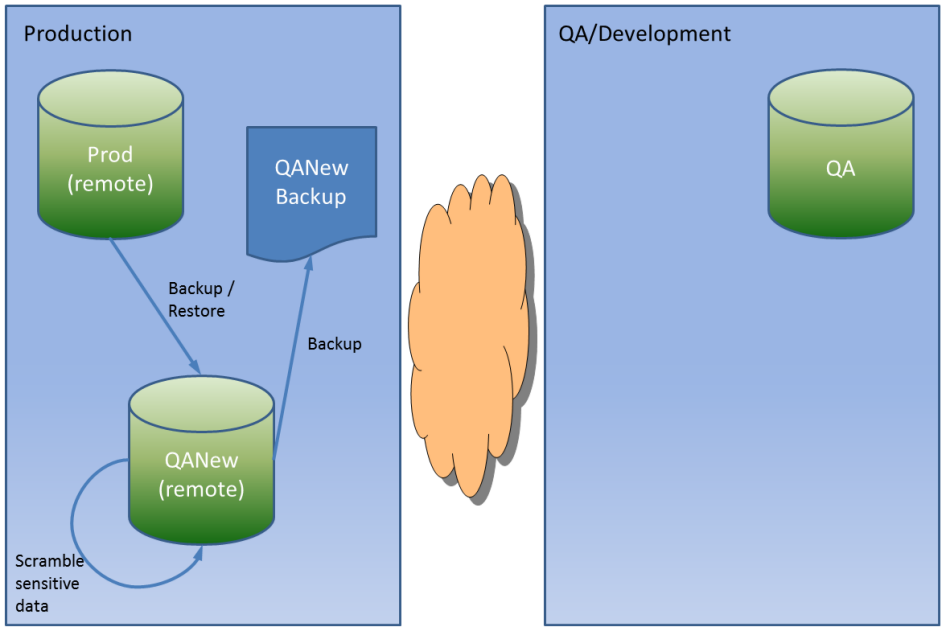


Figure 5 - Create Backup File with Scrambled Data

We allow this second backup overwrite the first backup file:

```
  <!-- backup newly scrambled QA database - replaces the prior backup in place -->
  <TASK action="SQLSCRIPT" level="ErrorOnlyDetail">
    <IN>DemoScript-ProdRemote</IN>
    <TIMEOUT>3600</TIMEOUT>
    <COMMAND>
      <![CDATA[
BACKUP DATABASE [DemoScriptQANew] TO  DISK = N'C:\Temp\DemoScriptProd.bak' WITH FORMAT, INIT,  NAME =
N'DemoScriptProdQA-Full Database Backup', SKIP, NOREWIND, NOUNLOAD,  STATS = 10, COMPRESSION
]]>
    </COMMAND>
  </TASK>
```

After the backup file is created, we can remove the QANew database in the production environment:

```
  <!-- take down the QA version in the prod environment, delete it -->
  <TASK action="SQLSCRIPT" level="ErrorOnlyDetail">
    <IN>DemoScript-ProdMasterRemote</IN>
    <TIMEOUT>3600</TIMEOUT>
    <COMMAND>
      <![CDATA[
ALTER DATABASE [DemoScriptQANew] SET SINGLE_USER WITH ROLLBACK IMMEDIATE;
GO
DROP DATABASE [DemoScriptQANew];
GO
]]>
    </COMMAND>
  </TASK>
```

In the next step, let's assume that due to the isolation of the production and QA regions, we need to perform a FTP copy of the backup file to the QA server. The SHCommand COPYTO task type supports a reliable FTP copy, including the ability to resume in the face of connection problems. There are a number of additional configuration parameters for this task type, but we'll use defaults here:

```
<!-- copy backup to ftp landing zone on QA server -->
<TASK action="COPYTO" level="ErrorOnlyDetail">
  <FROM>c:\temp\DemoScriptProd.bak</FROM>
  <TO>ftp://codex07/DemoScriptQANew.bak</TO>
  <USERNAME>demoscriptuser</USERNAME>
  <PASSWORD>demoDemo22</PASSWORD>
</TASK>
```

After this task completes, we have a situation as illustrated here:

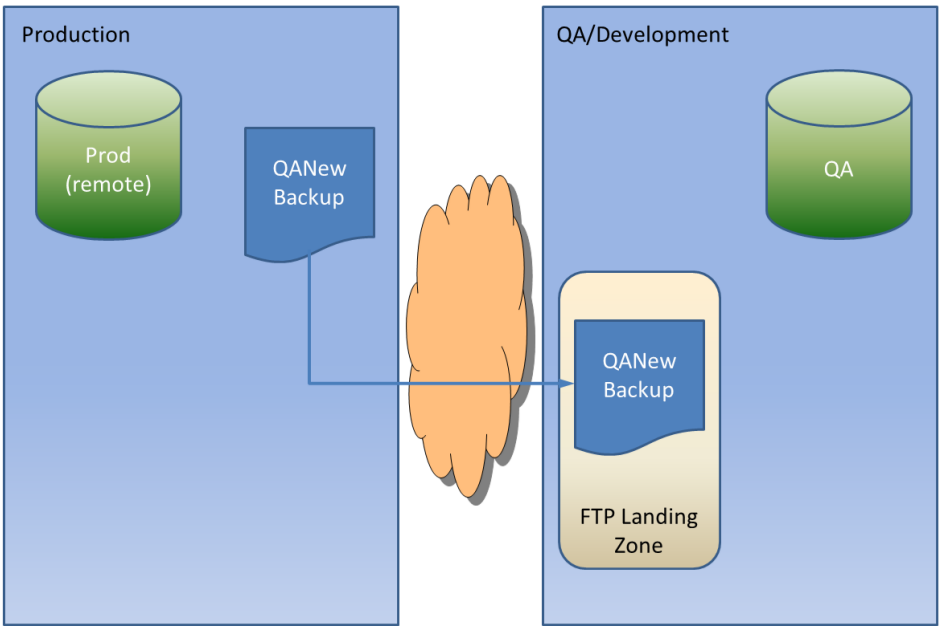


Figure 6 - Copy Backup File Across Environment Boundaries

Now let's suppose that the next steps should be carried out, running *on* the QA sever. SHCommand supports this using the HANDOFF task. The server named here is the target machine where SHCommand will be invoked using the specific configuration file (FILE), and using specific credentials (USERNAME and PASSWORD):

```
<TASK action="HANDOFF">
  <SERVER>codex07</SERVER>
  <FILE>c:\Project\VolumeDemo\OverlayExampleQA.shcommand</FILE>
  <USERNAME>CODEX07\Administrator</USERNAME>
  <PASSWORD></PASSWORD>
</TASK>
```

Here we assume that the QA server is named CODEX07, and SHCommand stops running on the production box until the SHCommand process completes on CODEX07. (Console output from CODEX07 is merged into the output on the calling machine.) Note that by omitting the password, we're prompted by SHCommand for the named user's password (not displayed as it is typed).

Let's take a look at the tasks inside OverlayExampleQA.shcommand:

```
<!-- restore to "new" QA version on QA server - fixup logins too -->
<TASK action="SQLSCRIPT" level="Summary">
  <IN>DemoScript-QAMaster</IN>
```

```
    <TIMEOUT>3600</TIMEOUT>
    <COMMAND>
      <![CDATA[
RESTORE DATABASE [DemoScriptQANew] FROM  DISK = N'C:\inetpub\ftproot\Temp\DemoScriptQANew.bak' WITH  FILE
= 1,  MOVE N'DemoScriptProd' TO N'C:\Project\VolumeDemo\DemoScriptQANew.mdf',  MOVE N'DemoScriptProd_log'
TO N'C:\Project\VolumeDemo\DemoScriptQANew_log.ldf',  KEEP_REPLICATION,  NOUNLOAD,  REPLACE,  STATS = 10
GO
USE DemoScriptQANew
GO
DECLARE @name sysname
DECLARE csr CURSOR FOR SELECT [name] FROM sys.sysusers u WHERE u.islogin=1 AND u.issqluser=1 AND u.uid>4
OPEN csr
FETCH csr INTO @name
WHILE @@FETCH_STATUS = 0
BEGIN
        BEGIN TRY
                EXEC sp_change_users_login 'update_one', @name, @name
        END TRY
        BEGIN CATCH
        END CATCH
        FETCH csr INTO @name
END
CLOSE csr
DEALLOCATE csr
GO
]]>
    </COMMAND>
  </TASK>
```

The first step is a straightforward restore of the backup file – this time on the QA server, to a database named QANew. The next bit of T-SQL warrants some explanation. The use of sp_change_users_login can be needed to deal with the fact that server-level internal ID's for logins won't necessarily match between the Production and QA machines, yet we need these to properly link with database-level user ID's, or our restored QA database may get errors such as "the login failed." What we've done to deal with this is include a small bit of T-SQL that attempts to match logins to users, *by name*, in the new QA database. The assumption that this is valid needs to be weighed based on your particular situation: it may be more appropriate to use a fixed list of calls to sp_change_users_login with hardcoded mappings, to be more secure. Also be aware that sp_change_users_login doesn't work with trusted logins (Windows accounts) – you'll have to handle these manually.

After the restore operation has completed, our overall state is:

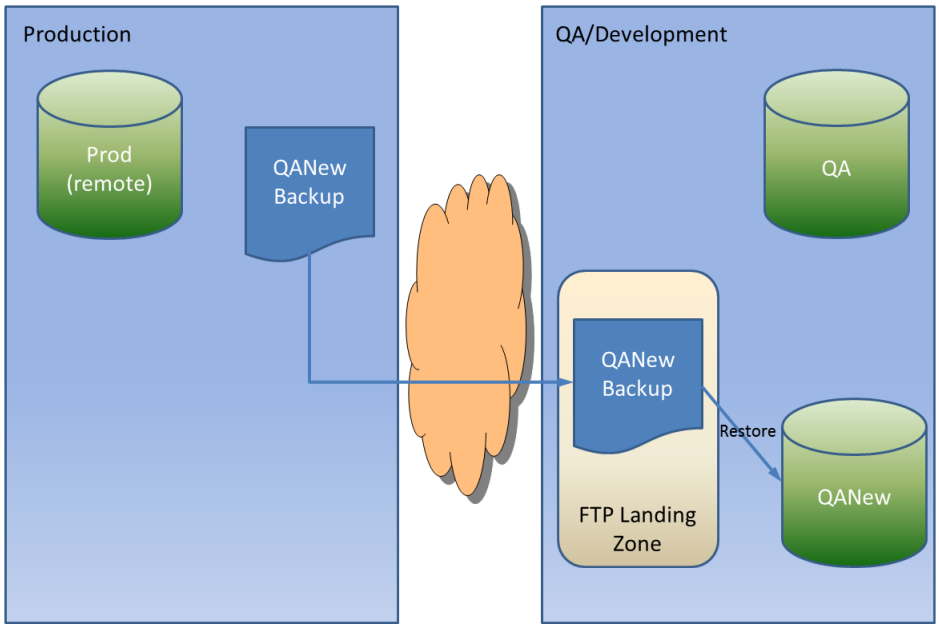


Figure 7 - Restore Database As QANew

The next series of steps take the existing QA database's schema and reconcile it to the QANew database. This might be useful if we have in-progress work in QA that's not in production yet and we don't want to have to go back and piece it together from the contents of a development database or a source control system. Using this approach we can at least arrive at a QA database that looks quite similar (in terms of schema) to the way it was prior to this whole operation. One downside is we may introduce back "bad objects" which are the result of model drift. (Model drift is where databases tend to get out-of-sync over time when people are allowed to make changes in a region like QA and not apply the same change to preceding regions like development – such as might happen with hot-fixes, or other changes which are not handled using traditional build processes.) Assuming this is a concern, it may be more favorable to: a) pick a time to run this process when there should be no "unpromoted" work in QA (get *full* buy-in that this is true!), b) restore the database as it is in production with no schema changes, c) advise developers that if there are any discrepencies, they can be found in the QAOld database. SQL-Hero's schema compare tool can be used to get a sense of what's acutally different, and these could be handled case-by-case. Similarly, the data compare tool can reconcile reference data differences easily.

In the real world, we've seen examples of why it can be a good idea to *not* perform this automatic reconciliation, with the price of some pain if people have been intentionally using the QA region to perform development. In this case it can be argued that at least QA has been "purified" – in a sense, model drift has been reversed. However, this can also be a disruptive approach and eventually the "new QA" can become as "polluted" as it was before, mainly since this kind of situation speaks to a lack of rigor around process, meaning that without curing the process issue, similar problems are likely going to recurr.

Assuming we *do* want to perform this reconciliation automatically, the first step is to compare objects in QA and QANew, with the intent of moving *from* QA *to* QANew. The SCHEMASCRIPT task actually builds

the T-SQL necessary to reconcile QANew with QA. SCHEMAEXECUTE actually invokes the script generated by the SCHEMASCRIPT step – we'll put any error output in a log file that can be attached to our final success/failure email. It's worth noting that SHCommand supports a BUILDFILTER task which allows us to affect the list of objects to process by criteria such as "changed in the last x days" or by regular expression matching against the object schema / name, by object type, etc.

If we were interested in automatically reconciling some static reference data from the "old QA" to the "new QA," there exist "DATACOMPARE", "DATASCRIPT" and "DATAEXECUTE" task types. We've omitted these from this demonstration, but they could be included along with the aforementioned BUILDFILTER task as well.

```
<!-- syncrhonize any in-progress work to the new database - first step is to compare -->
<TASK action="SCHEMACOMPARE" level="Summary">
  <FROM>DemoScript-QA</FROM>
  <TO>DemoScript-QANew</TO>
</TASK>

<!-- second step is to build the reconciliation script -->
<TASK action="SCHEMASCRIPT" level="Summary">
  <TARGET type="File">ScriptPendingQAToQANew[date:yyyyMMddHHmm].sql</TARGET>
</TASK>

<!-- third step is to actually apply changes -->
<TASK action="SCHEMAEXECUTE" level="Summary">
  <IN>DemoScript-QANew</IN>
  <OUTPUT>ScriptPendingQAToQANew[date:yyyyMMddHHmm].log</OUTPUT>
</TASK>
```

After these steps, the state has become:

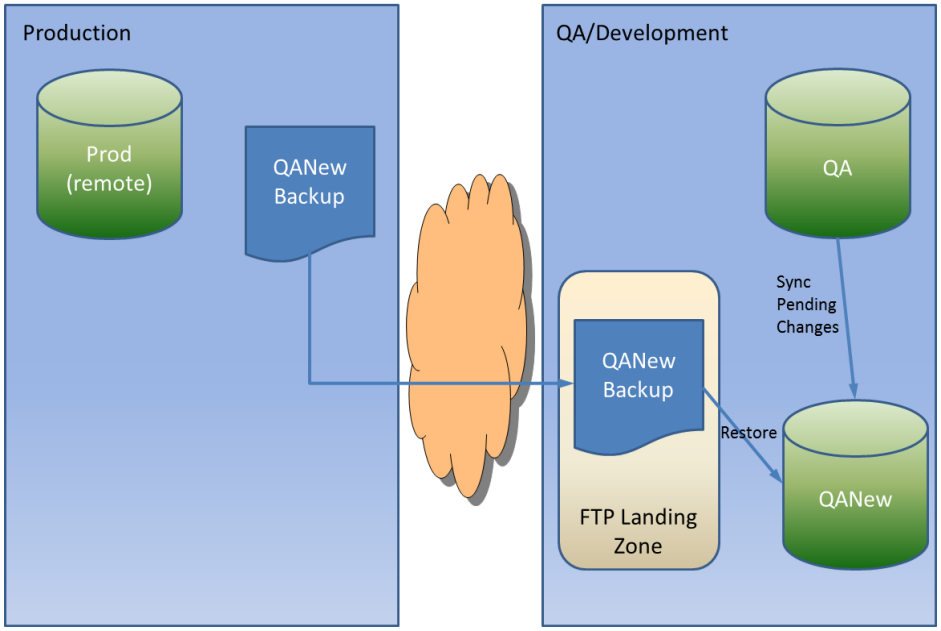


Figure 8 - Sync QANew Schema with Existing QA Schema

The next step – the UNITTEST task type - can be considered "optional" and has some value after we've synchronized objects. What this task does is takes all objects which are "visible" for testing (i.e., if you

had a filter applied using BUILDFILTER, the list of visible objects is restricted accordingly), are of a testable type (stored procedures, UDF's, etc.), and it builds a T-SQL script which is used to exercise those objects. It then executes that script, reporting any test failures as task output. What this means is if the act of running this backup/restore process has somehow changed what *would* have been a successful procedure call into a call that now *fails* due to a SQL error, there's a chance we'll get told about the problem now, as opposed to later when users are trying to do testing in the QA environment. This isn't necessarily a perfect form of testing since it only works with test cases that use preset parameters to test objects in isolation, but it really *is* better than nothing from our own experiences since we've see this approach catch things that would've otherwise fallen through the cracks on occassion.

To get a fuller sense of the inner working of unit testing supported by SQL-Hero, check out the whitepaper SQL-Hero Unit and Performance Testing.

In the task specification below, we're using the "Detail" output level, which means all objects tested have their names listed under either the "Success" or "Failure" heading. From this we can quickly pinpoint problems that could have been introduced by a number of things including:

- The schema synchronization step was incomplete or failed somewhere due to dependencies and as such, someone needs to be made aware of problems.
- The refreshed data has introduced an issue that now needs to be addressed.
- There are lingering problems that might have been previously introduced and have yet to be addressed.

The STOP element indicates whether any test failures are considered "fatal" and should stop SHCommand or not.

```
<!-- run touch tests, where failures should be informational to users in email below -->
<TASK action="UNITTEST" level="Detail">
  <IN>DemoScript-QA</IN>
  <STOP>false</STOP>
</TASK>
```

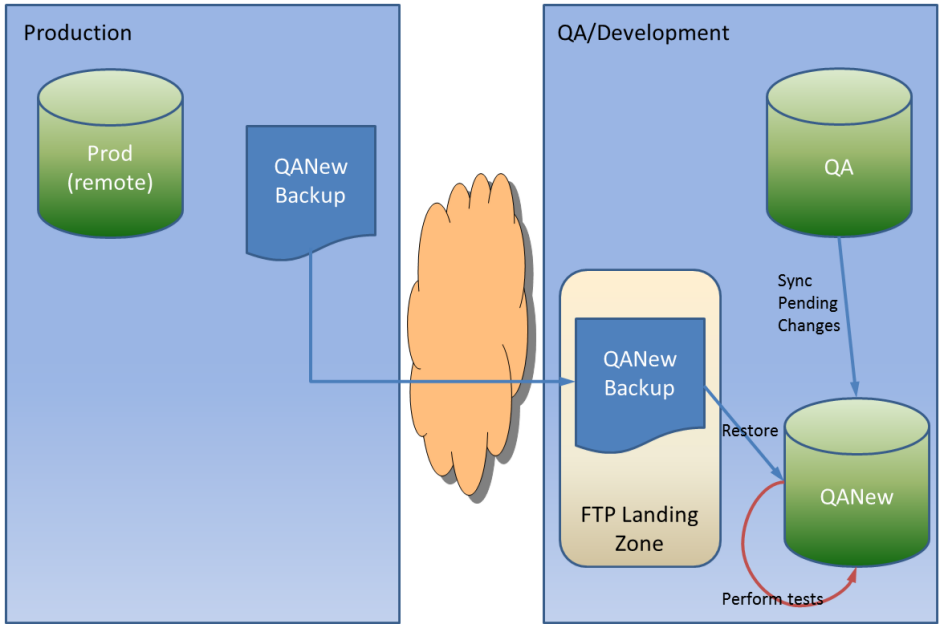


Figure 9 - Perform Unit Tests Against QANew - Ensure Functional

We now have a functional database called QANew that contains what we want to replace the current QA database. It would be ideal to retain the existing QA database (under a different name) so we have it for reference. To achieve this "swap," we can use detact / attach actions in SQL Server. This is actually T-SQL combined with some file copies to swap the physical database files:

```
  <!-- rename original database to "old", rename "new" to existing - first sub-step is to detach existing
databases -->
  <TASK action="SQLSCRIPT" level="Summary">
    <IN>DemoScript-QAMaster</IN>
    <COMMAND>
      <![CDATA[
ALTER DATABASE DemoScriptQA SET SINGLE_USER WITH ROLLBACK IMMEDIATE;
EXEC sp_detach_db 'DemoScriptQA'
GO
ALTER DATABASE DemoScriptQANew SET SINGLE_USER WITH ROLLBACK IMMEDIATE;
EXEC sp_detach_db 'DemoScriptQANew'
GO
BEGIN TRY
ALTER DATABASE DemoScriptQAOld SET SINGLE_USER WITH ROLLBACK IMMEDIATE;
EXEC sp_detach_db 'DemoScriptQAOld'
END TRY
BEGIN CATCH
END CATCH
GO
]]>
    </COMMAND>
  </TASK>

  <TASK action="COPYTO" level="Summary">
    <DELETE_AFTER>true</DELETE_AFTER>
    <FROM>\\codex07\C$\Program Files\Microsoft SQL
Server\MSSQL10_50.SQL2008R2\MSSQL\DATA\DemoScriptQA.mdf</FROM>
    <TO>\\codex07\C$\Program Files\Microsoft SQL
Server\MSSQL10_50.SQL2008R2\MSSQL\DATA\DemoScriptQAOld.mdf</TO>
  </TASK>
  <TASK action="COPYTO" level="Summary">
    <DELETE_AFTER>true</DELETE_AFTER>
```

```
    <FROM>\\codex07\C$\Project\VolumeDemo\DemoScriptQANew.mdf</FROM>
    <TO>\\codex07\C$\Program Files\Microsoft SQL
Server\MSSQL10_50.SQL2008R2\MSSQL\DATA\DemoScriptQA.mdf</TO>
  </TASK>
  <TASK action="COPYTO" level="Summary">
    <DELETE_AFTER>true</DELETE_AFTER>
    <FROM>\\codex07\C$\Program Files\Microsoft SQL
Server\MSSQL10_50.SQL2008R2\MSSQL\DATA\DemoScriptQA_log.ldf</FROM>
    <TO>\\codex07\C$\Program Files\Microsoft SQL
Server\MSSQL10_50.SQL2008R2\MSSQL\DATA\DemoScriptQAOld_log.ldf</TO>
  </TASK>
  <TASK action="COPYTO" level="Summary">
    <DELETE_AFTER>true</DELETE_AFTER>
    <FROM>\\codex07\C$\Project\VolumeDemo\DemoScriptQANew_log.ldf</FROM>
    <TO>\\codex07\C$\Program Files\Microsoft SQL
Server\MSSQL10_50.SQL2008R2\MSSQL\DATA\DemoScriptQA_log.ldf</TO>
  </TASK>

  <TASK action="SQLSCRIPT" level="Summary">
    <IN>DemoScript-QAMaster</IN>
    <COMMAND>
      <![CDATA[
EXEC sp_attach_db 'DemoScriptQA', 'C:\Program Files\Microsoft SQL
Server\MSSQL10_50.SQL2008R2\MSSQL\DATA\DemoScriptQA.mdf', 'C:\Program Files\Microsoft SQL
Server\MSSQL10_50.SQL2008R2\MSSQL\DATA\DemoScriptQA_log.ldf'
GO
EXEC sp_attach_db 'DemoScriptQAOld', 'C:\Program Files\Microsoft SQL
Server\MSSQL10_50.SQL2008R2\MSSQL\DATA\DemoScriptQAOld.mdf', 'C:\Program Files\Microsoft SQL
Server\MSSQL10_50.SQL2008R2\MSSQL\DATA\DemoScriptQAOld_log.ldf'
GO
]]>
    </COMMAND>
  </TASK>
```

At this point, we've completed what SHCommand has been requested to do on the QA server and execution control returns to the production server's instance of SHCommand (invoked prior to the HANDOFF task). We've also reached our end-state:

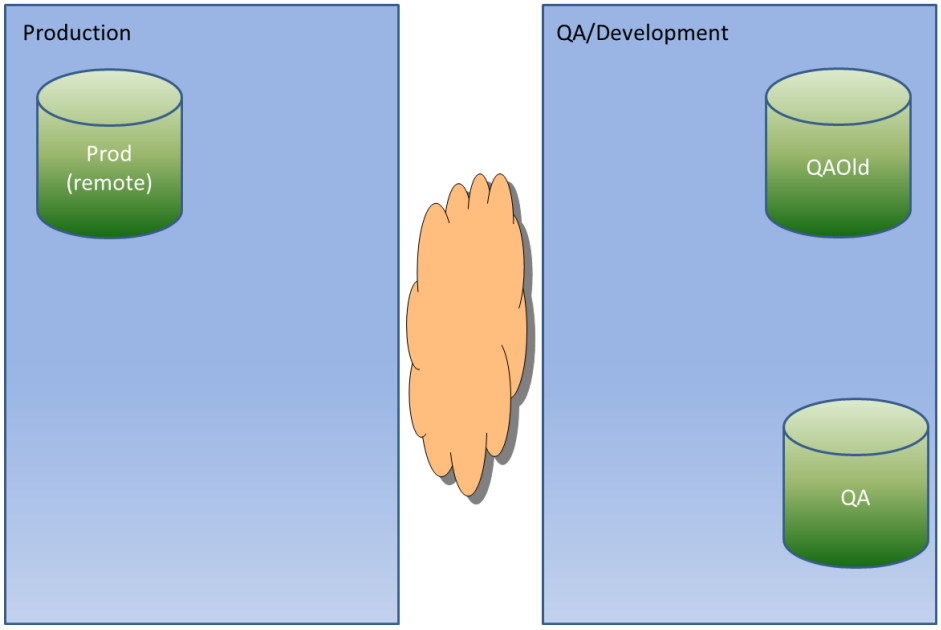


Figure 10 - Detach Databases, Move Files, Reattach Databases as QA and QAOld

As a final step, we want to send an automatic email to all developers who have ever worked in either the Development or QA databases. A more traditional approach might be to use a fixed distribution list, but SQL-Hero offers the option to leverage its central repository of all database changes, along with user names and email addresses to associate with those changes. To use a dynamic distribution list, we use the "anychage" attribute of the TO element to specify that changes within the last 180 days are to be used to arrive at the change list, and the text in the TO element names the database aliases which are checked. (Without the anychange attribute, the TO element contains a comma-separated list of email addresses – a fixed distribution list.)

```
  <!-- email dist list based on ultimate success (stop=true implies will not run the next step which is
the failure message which we're tailoring to be different) -->
  <TASK action="EMAIL" stop="true">
    <MAILSERVER>codex06</MAILSERVER>
    <TO anychange="180">DemoScript-QA,DemoScript-Dev</TO>
    <FROM>SHCommand@SH.Null</FROM>
    <SUBJECT>Command line tool:  Success</SUBJECT>
    <BODY>The command line tool completed successfully - QA database is restored with production data.
Attaching output.</BODY>
    <ATTACHMENT>OverlayExampleOutput.log</ATTACHMENT>
  </TASK>
```

In the MAILSERVER, we've named CODEX06 as our out-going SMTP server, and we're attaching the complete console output from this session.

The End Result

After all this has been run, what do we have? If we run the same procedure that in production gave us plain-text credit card numbers and names, in the newly restored QA database we see:

```
EXECUTE [dbo].[up_CreditCardOrderList]
      @MaxRow = @MaxRow
```

	OrderDate	OrderAmount	CreditCardNumber	CardholderName
1	11/12/2010 00:00:00.000	27.5200	7777666655554444	TESTER CARD
2	3/25/2010 00:00:00.000	572.2500	5555444433332222	TESTING CARD
3	2/17/2011 00:00:00.000	657.7100	7777666655554444	TESTER CARD
4	1/10/2011 00:00:00.000	968.5100	6666555544443333	CARD TEST
5	4/6/2010 00:00:00.000	741.6100	7777666655554444	TESTER CARD
6	4/18/2010 00:00:00.000	297.8700	6666555544443333	CARD TEST
7	6/28/2010 00:00:00.000	26.4200	5555444433332222	TESTING CARD

This is exactly what we were after: we still see related data such as credit card orders intact, but the sensitive credit card data has been replaced with the test card numbers and names from our TestCreditCard table.

Another thing to note about the final state is we have a copy of the "old" QA database remaining available for developers to reference. This can be very useful if the schema reconciliation step doesn't work perfectly (heaven forbid!), or there are questions about the data that "used to be in QA."

It's not hard to think of other possibilities that are afforded by SHCommand and in that spirit we're constantly adding new functionality. In fact, it was not needed for this demo but the T-SQL script contained in SQLSCRIPT tasks is actually treated like a SQL-Hero template which happens to *produce* T-SQL, which is what's actually executed. What this means is you're able to add markup tags to your T-SQL that transform into "real T-SQL" when executed.

Command line tools are very easy to integrate into larger scripting processes (and can be invoked on a schedule using the "AT" command), so although we may implement a .shcommand file designer in the future to make configuration easier, the tool itself will remain and become increasingly important in the automation use cases we're looking to address in the future.

Although the demo presented in this paper works like a charm in our own testing environment, your particular environment may have additional considerations when dealing with a substantial operation like restoring a production database. We have not covered issues with database encryption, for example, which can include some necessary leg work around certificates and master keys. A good resource that covers some of these points can be found at http://sqldbasupport.wordpress.com/2009/05/15/sql-server-databases-migration

Contrast with Other Tools

How does what's outlined in this document compare with what's available in other tools? We can't know every single tool available, but we'll look at some of the more commonly used. If you're aware of

others that have significant value, feel free to drop us a line at admin@codexframework.com: we may elect to build a comparison with it or help shape SQL-Hero in the future. By all means, challenge us with use cases you're facing and might be struggling with - there's a good chance we've either already solved the problem or can include the support to solve it very quickly!

For some specific steps that compose our solution, we might rely on a tool like Visual Studio 2010 – it could be used to handle, for example, our data transformation step. VS2010 includes Data Transformation Plans which are intended to do things similar to what we achieved with the credit card number scrambling. However, don't assume things are quite as easy for what we outlined using SQL-Hero! As discussed in the slide deck Managing Test Data and Stress Testing Your SQL Applications, the Transformation Plan chose to drop data in dependent tables, such as CreditCardOrder – clearly not what our intention is! Even if we assume we could have made this work with VS2010, we'd have to do more work, potentially write some .NET code to support it, and therefore spend more time to achieve the same result. In terms of performance, SQL-Hero uses bulk loading approaches similar to what you find VS2010 using internally – however, as we've mentioned, it addresses limitations you'll find in VS.

Another option that might be used to implement a solution similar to what's presented here is to use PowerShell. However, you'll still need some additional logic to handle data transformation, unit testing, object synchronization, and so on. You might wonder if you can invoke SQL-Hero components programmatically to do some of this. Although you can technically, we don't necessarily advocate this in the short-term: API's are not officially published since we reserve the right to change them, given that SQL-Hero is technically still a young product. In the end, the amount of script you'd need to write is very limited when using SHCommand versus what you'll find in other scripting technologies to achieve the same result.

In terms of performance, SQL-Hero relies on bulk operations and in doing so, has a rather good performance profile compared to other tools. Visual Studio may in fact be faster in cases, but we've traded some performance for what is clearly some rich functionality that lets you do more with less effort. Ultimately, the script we've presented here runs very quickly over 300,000 rows in the CreditCard table.

What about other third-party tools like SQLEdit which offers a tool to scramble data, and even a SDK to support this in a programmatic sense? With an API, you'll still end up writing code, so we assert the ideal situation is to have something which directly integrates into a scripting engine and therefore adapts to a variety of needs more easily. The task engine provided by SHCommand is something we've only really touched on – another powerful task type is "TEMPLATE". This task allows you to execute any template which is written in the SQL-Hero template language, which in turn supports the SCRIPT_C# markup tag, which allows you to inject C# (4.0) into your templates. The "Bulk Load Table From Stored Procedure Results" template is a good example of how this can be leveraged, where in less than 2Kb of an existing template, we can tie a stored procedure's result set (or multiple result sets) to a bulk copy operation which loads a target table (or tables). To illustrate, we could incorporate invocation of this bulk load into a larger SHCommand scripting operation to move a subset of data around very quickly – something that could easily be a part of automatically constructing a clean QA environment, depending

on your needs. More detail on the template engine and SHCommand are covered in other whitepapers, and we encourage you the check back often for new documentation – or follow @sqlheroguy on Twitter to get an up-to-date news feed when new documentation is released.