[Avoiding Constraint Violations during SQL Server Deployments](https://www.red-gate.com/hub/product-learning/sql-compare/database-build-breakers-avoiding-constraint-violations-during-sql-server-deployments" \o "Database Build Breakers: Avoiding Constraint Violations during SQL Server Deployments)

[Phil Factor](https://www.red-gate.com/blog/author/phil-factor" \o "Posts by Phil Factor)

10 February 2020

FROM: <https://www.red-gate.com/hub/product-learning/sql-compare/database-build-breakers-avoiding-constraint-violations-during-sql-server-deployments>

A common database build breaker is data that violates the conditions of any of the CHECK, UNIQUE or FOREIGN KEY constraints, and unique non-clustered indexes, designed to protect the consistency and integrity of your data. Phil Factor explain how to avoid this problem, using SQL Compare and some custom stored procedures to discover which rows will cause violations, and fixing them, before running the build.

Guest post

This is a guest post from Phil Factor. Phil Factor (real name withheld to protect the guilty), aka Database Mole, has 30 years of experience with database-intensive applications.  
  
Despite having once been shouted at by a furious Bill Gates at an exhibition in the early 1980s, he has remained resolutely anonymous throughout his career.  
  
He is a regular contributor to [Simple Talk](http://www.simple-talk.com/) and [SQLServerCentral](http://www.sqlservercentral.com/).

Build errors, caused by constraint violations, can happen during development, where you will typically build the database, temporarily disable constraints, bulk import the test data and reenable constraints such that they are ‘trusted’. If someone has, for example, fiddled with the test data without updating the constraint definitions, then re-enabling constraints can lead to build errors. You can encounter similar problems if you’re deploying changes to table constraints in existing databases, and some ‘bad data’ has previously snuck into those tables. Dealing with problems, once a deployment has already failed will inevitably cause disruption, while someone identifies and fixes each row of bad data, error-by-error.

This article will show you how to prevent constraint violations from ‘breaking the build’, during SQL Compare deployments, by ferreting out any conflicting rows in advance, and developing a script to fix the data. The detective work is done by some custom stored procedures, which extract the constraint metadata from the source database (and can save it in a JSON constraint document), and then ‘test’ the data in the target database against the new constraint definitions. This produces an advance report listing which rows of data, if any, would cause constraint violations during a subsequent deployment, and why. Armed with this, you can develop the script that will fix the bad data and so avoid any errors.

During development builds, you can bulk load the data, test it and run the script to fix any conflicting rows, and the reenable constraints. When deploying changes to other databases, you can use the JSON constraint document to test the data in the target database. You can either fix the data using a SQL Compare pre-deployment script or, probably safer, deploy any new constraints in a ‘disabled’ state, initially, and then use a SQL Compare post-deployment script to fix the data and then enable them.

Why worry about constraints and bad data?

Constraints restrict in some way the values allowed in a column, or combination of columns. A FOREIGN KEY constraint restricts values to those that already exist in the referenced table. A UNIQUE constraint forbids duplicates (if users request the current sales tax rate in Arizona, they want one answer, not three). A CHECK constraint restricts values to those that have meaning for the business. In other words, there are there to protect the integrity and consistency of your data. They police the data to make sure that it doesn’t contain mistakes, ambiguities and misunderstandings.

However, when you build a new version of a database, during development, or deploy changes to existing databases, the constraints you’ve rightfully added to protect your data, can also slow you down.

Breaking the build during development

During development work, once you’ve built a new version of a database schema, using SQL Compare, you’ll then want to load in the test data. However, you can encounter problems. Firstly, the FOREIGN KEY constraints will force you to fill the tables in a reverse-dependency order, starting with tables that don’t reference other tables. Secondly, SQL Server will fastidiously check every row against existing constraints, as it’s inserted, and abort the mission at the first violation.

Msg 547, Level 16, State 0, Line 2

The ALTER TABLE statement conflicted with the CHECK constraint "CK\_Employee\_MaritalStatus". The conflict occurred in database "AdventureWorks", table "HumanResources.Employee", column 'MaritalStatus'

Uh-oh, it seems that the data contains some unexpected values in the MaritalStatus column. You’ll need to find and fix the conflicting row and try again.

It is much easier and faster to disable all constraints, and unique non-clustered indexes, before loading the data. This is the database equivalent of administering an anesthetic. You can load the data, and tinker with it, without protest, and then re-enable all the constraints, adding the option WITH CHECK, meaning that SQL Server will immediately check all the data to make sure it complies with all existing table constraints (you’d also need to rebuild any rebuild unique non-clustered indexes, if you disabled them).

|  |  |
| --- | --- |
| 1 | EXEC sp\_msforeachtable 'ALTER TABLE ? WITH CHECK CHECK CONSTRAINT all' |

This is still a moment that can strike fear into even the most placid DBA, dreading to see the errors that indicate constraints violations. If you are lucky, all is well, and each constraint is both enabled and trusted. If it doesn’t go well, SQL Server [flags the constraints as ‘not-trusted](https://www.red-gate.com/hub/product-learning/sql-prompt/the-whys-and-wherefores-of-untrusted-or-disabled-constraints)‘. If you decide to skip the checks, when enabling a constraint, using the WITH NOCHECK option, then it’s a lot faster but the constraint will be untrusted. This is bad news because, while all enabled constraints will apply to newly inserted rows, the optimizer cannot use untrusted constraints when devising execution plans for queries that access the table.

What would be useful is to have a list of all the rows that would fail the constraints, before reenabling all of them. During development builds, we’ll have a connection to the target database, so we simply run the new build, disable constraints, load the data, test it against the live constraint metadata and devise a script to fix things up, before reenabling constraints.

Breaking deployments to existing databases

Imagine that you’ve made some development changes that improve the constraints on some of the tables in AdventureWorks. It all works fine with your test data sets, so you generate the deployment script using SQL Compare, test and amend it where necessary, and deliver it to your Ops teams. They need to test it all out on Staging, which has a copy of the real data, much of which is necessarily restricted and isn’t available to development.

The build fails, and the Ops team send you a report that has an error like the one we saw before. It seems that some of the existing data violates the new constraint definitions, but the error message will tell you only of the first row that fails a constraint condition, not all of them. You fix that row and every time you try to enable the constraint WITH CHECK you get another error.

At this stage, you, or the Ops team if you can’t have full read and write access to the production data, would need to wade through each of these errors, row by row, and come up with a script to fix the data.

You could, alternately, avoid this, if the development team could be delivered a list of all the rows that would fail the new constraints, and unique non-clustered indexes, before running the deployment. The Ops team could test the live data against the constraint metadata for the forthcoming deployment, stored in JSON format. This will generate the report that lets the developers know which values would cause the problems, so that they can produce a ‘data-cleansing’ script to fix all the constraint problems, check constraints, unique constraints and foreign key constraints, well in advance of the release.

How to check and get reports on constraints

Constraint errors, duplicate errors and referential errors happen with surprising frequency, whether you are updating a database to a new version, while preserving the data, or building a new version from scratch and then loading the data. You’ve deployed a new database version, but the tables haven’t altered, and yet you are confronted with a message pane that is a sea of red. Perhaps someone, fed up with duplicate rows creeping in, has, reasonably and sensibly tightened, up the constraints.

To avoid all this, we need to run a range of slightly different tests, in order to catch:

Bad data checks – for rows that would violate the conditions of CHECK constraints

Duplicate checks – for rows that would violate UNIQUE consultants or non-clustered indexes.

Relational integrity checks – for rows that would violate FOREIGN KEY constraints

We extract the list of constraints from the source database, as a JSON constraint file, and then use it to test the data. We store the result of our tests in a JSON report file.

The bad news is that it needs to be scripted, mostly in SQL. The good news is that I’ve done it for you and put it in my public [Github repository](https://github.com/Phil-Factor/TestOutConstraintsInSQLServer). I’ve described how it works in a series of Simple-Talk articles: [But the Database Worked in Development! Preventing Broken Constraints](https://www.red-gate.com/simple-talk/sql/database-devops-sql/but-the-database-worked-in-development-preventing-broken-constraints/), But the [Database Worked in Development! Avoiding Duplicate Rows](https://www.red-gate.com/simple-talk/sql/database-devops-sql/but-the-database-worked-in-development-checking-for-duplicates/) and But the Database Worked in Development! Checking Referential Integrity.

For each type of check, there are just two stored procedures to do the work. A temporary stored procedure named #List\* will query the system catalog views for the index or constraint metadata and store it in JSON format. A corresponding temporary stored procedure named #Test\* executes its corresponding #List\* procedure, runs the checks and produces the report.

For example, getting a detailed report of which rows in a local target database violated which CHECK Constraints might look something like this:

|  |  |
| --- | --- |
| 1  2  3 | DECLARE @OurFailedConstraints  NVARCHAR(MAX)  EXECUTE #TestAllCheckConstraints @TheResult=@OurFailedConstraints OUTPUT  SELECT @OurFailedConstraints AS theFailedCheckConstraints |

In my GitHub repo, I’ve made things as simple as possible by providing three scripts, one for each type of check:

[TestLoadedDataForCheckConstraints.sql](https://github.com/Phil-Factor/TestOutConstraintsInSQLServer/blob/master/TestLoadedDataForCheckConstraints.sql)

[TestLoadedDataForFKConstraints.sql](https://github.com/Phil-Factor/TestOutConstraintsInSQLServer/blob/master/TestLoadedDataForFKConstraints.sql) and

[TestLoadedDataForUniqueConstraints.sql](https://github.com/Phil-Factor/TestOutConstraintsInSQLServer/blob/master/TestLoadedDataForUniqueConstraints.sql)

All each one does to execute the code in its two associated stored procedures. After the test is done, you get three JSON reports on how much work needs to be done to clean the data.

I’ve also included in the repo a PowerShell file called ExecuteConstraintsCode.ps1, which contains an Assert-Constraints function to make all these checks easier, once everything is set up. You simply place the above three files in a directory scoop and the function scoops these up and executes them. I’ll show some examples later.

A simple walk-through: CHECK constraint violations

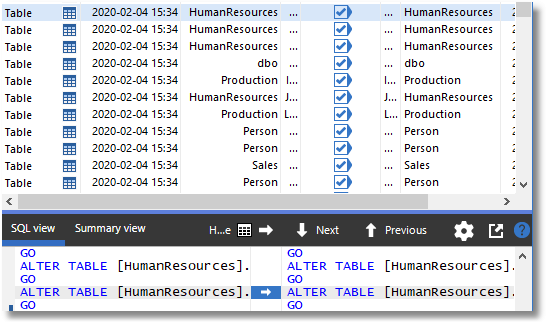
Let’s see what can go wrong, with a simple example, using SQL Compare. The AdventureWorks business, we’ll pretend, has told us that an employee has exceeded their annual allowance of 65 hours sick leave, without HR being aware. The existing constraint allows between 0 and 120 hours, so needs to be fixed.

Developer Dave alters the employee table to fix this, and checks in the code:

|  |  |
| --- | --- |
| 1 | ALTER TABLE [HumanResources].[Employee]  WITH CHECK ADD  CONSTRAINT [CK\_Employee\_SickLeaveHours] CHECK  (([SickLeaveHours]>=(0) AND [SickLeaveHours]<=(65))) |

Unfortunately, Dave is a bit absent-minded, and he forgot to check in the migration script that is needed, to deal with any existing data that has more than 65 hours, before disappearing off to an Agile conference. In his absence, and unaware of the problem or its implications, the AdventureWorks devs decide to deploy the change using SQL Compare. They set the source to be the source control directory, and in this case the destination is their staging copy of AdventureWorks, which contains all their test data. Equally, it could be an empty database, if they just need to build from scratch.

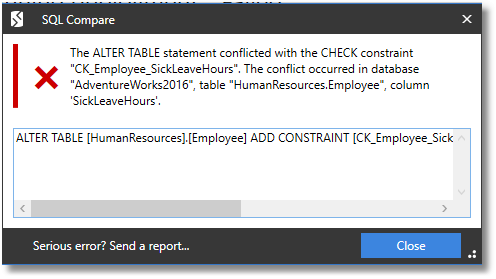
They check that the collation is set to be the same and then run SQL Compare; it detects Developer Dave’s change.



They generate the deployment script, which includes the following code to modify the constraint:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | PRINT N'Adding constraints to [HumanResources].[Employee]'  GO  ALTER TABLE [HumanResources].[Employee] ADD CONSTRAINT [CK\_Employee\_SickLeaveHours] CHECK (([SickLeaveHours]>=(0) AND [SickLeaveHours]<=(65)))  GO  IF @@ERROR <> 0 SET NOEXEC ON  GO |

They press on and go ahead with a deployment. Had developer Dave added the migration script to ensure that the maximum SickLeaveHours in the HumanResources.Employee was less than or equal to 65 then all would have gone well, but he didn’t, and the deployment fails:



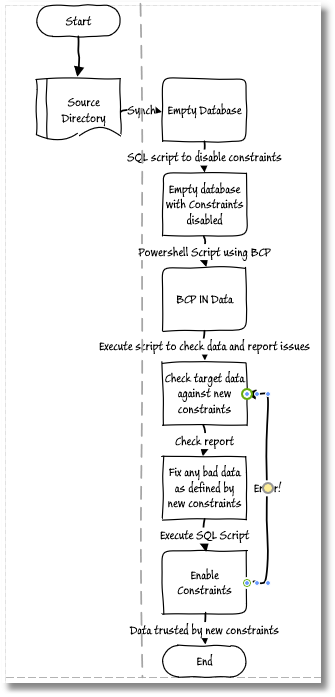
During a build from scratch, they would encounter a similar error on re-enabling constraints, after loading in their test data.

Let’s see now how, in either case, they can get the JSON report that will give you all the details of the rows that violate constraint conditions, from which you can prepare the migration script that will fix the data, before the errors occur.

Checking for constraint violations during development builds

We’ll assume that the team want a working database with data in it, starting from their scripts in source control. We’ll assume that they have a copy of the development data ready in a directory. [Refer here](https://www.red-gate.com/simple-talk/blogs/copying-of-all-data-between-sql-server-databases-and-servers/) on how to do that.

The process looks like this:

[](https://www.red-gate.com/wp-content/uploads/2020/02/BuildAndFill.png)

1. Build the database

When you deploy a SQL Server database, SQL Compare can be used for deployment in two rather different ways, build and synchronization. It can change the schema of an existing target database to match the source. If the devs are building a new database, then the target database is empty, and so the deployment script will modify the empty (model) database schema, so it matches the source. If you are scripting a build process for regular use, you’d want to use SQL Compare CLI, or for a more robust process, with documentation, code analysis checks and so on, use [SQL Change Automation](https://www.red-gate.com/hub/product-learning?product=sql-change-automation&topic=database-builds).

Here’s a very simple PowerShell script for building the new database, with the new constraint, using SQL Compare CLI. You’ll need to fill in the path to SQL Compare, and specify the server and database to get it from. If you use integrated security $MyUserId and $MyPassword to $null, otherwise fill them in:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | Set-Alias SQLCompare "${env:ProgramFiles(x86)}\Red Gate\SQL Compare 13\sqlcompare.exe" -Scope Script  $MyServerInstance='MyServer'#The SQL Server instance  $MySourceDatabase='MyDatabase' #The name of the database  $MySourceDatabasePath = "$($env:HOMEDRIVE)$($env:HOMEPATH)\Documents\GitHub\$MySourceDatabase"  $MyNewDatabase = 'TargetDatabase'  $MyUserId='MyUserID'  #fill this in if you need credentials  $MyPassword='MyPassword' #fill this in if you need credentials  #------  if ($MyUserId -ne $NULL)      {      SQLCompare /scripts1:$MySourceDatabasePath /username2:$MyUserId /Password2:$MyPassword `                 /database2:$MyNewDatabase /server2:$MyServerInstance /force      }  else      {      SQLCompare /scripts1:$MySourceDatabasePath `                 /database2:$MyNewDatabase /server2:$MyServerInstance /force      }  if ($?) {'That went well'}  else {"That Went Badly (code $LASTEXITCODE)"} |

2. Disable constraints and unique indexes

In order to avoid errors from new foreign key or check constraints until we’ve had time to identify and fix them, we need to disable all foreign key or check constraints.

|  |  |
| --- | --- |
| 1 | EXEC sp\_msforeachtable "ALTER TABLE ? NOCHECK CONSTRAINT all" |

The code to disable all non-clustered unique constraints and indexes is more complicated, but here is the code that generates the statements that can then be executed

|  |  |
| --- | --- |
| 1  2  3  4  5 | SELECT  'Alter Index '+IX.name+ ' ON '+QuoteName(Object\_Schema\_Name(tabs.object\_id))+'.'+QuoteName(tabs.name)+ ' DISABLE'    AS Statement           FROM sys.tables AS tabs            INNER JOIN sys.indexes AS IX              ON IX.object\_id = tabs.object\_id        WHERE (IX.is\_unique\_constraint =1) OR (IX.type=2 AND IX.is\_unique=1) |

The alternative to disabling all constraints and indexes as a separate stage, is to adopt the discipline of checking in constraints as disabled:

|  |  |
| --- | --- |
| 1  2 | ALTER TABLE [HumanResources].[Employee] WITH NOCHECK ADD CONSTRAINT [CK\_Employee\_SickLeaveHours]  CHECK (([SickLeaveHours]>=(0) AND [SickLeaveHours]<=(65))) |

Likewise, you can create an index and set its state to disabled:

|  |  |
| --- | --- |
| 1 | ALTER INDEX [AK\_SalesTaxRate\_StateProvinceID\_TaxType] ON [Sales].[SalesTaxRate] DISABLE |

In which case, when we use SQL Compare to do the build, in the previous stage, it will ensure that they are in that state at the end of the synchronization. Then you will have an automated step to re-enable constraints and rebuild indexes, after you’ve done the tests and fix the data.

3. Load the data

We now need to get the data in. We can use the script I’ve published as [Copying of all data between SQL Server databases and servers](https://www.red-gate.com/simple-talk/blogs/copying-of-all-data-between-sql-server-databases-and-servers/).

Although native BCP, using the command-line BCP utility, is the fastest approach, it is by no means the only way of doing it. I’ve published quite a bit on using JSON. Whatever the medium that the data comes in, you can use the bulk load library of Data.SqlClient.SqlBulkCopy which will allow you to import DataTables, Datareaders, or arrays of data rows. In layman’s terms, anything that can be represented as a table can be imported in bulk. I even use it to import CSV when it needs cleaning ‘on the fly’ and then enable them when you’ve finished.

4. Test the data for constraint violations

Development builds are the simplest case, because the team are checking a local target database that has been synchronised with, or built from, the new version but without the new constraints or unique non-clustered indexes being enabled.

This is a relatively simple process because you don’t need to fetch or store the JSON constraint document, because you can simply examine the metadata in the target database and run the tests. If you aren’t interested in saving the reports, load all the temporary stored procedures and execute:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | DECLARE @OurFailedConstraints  NVARCHAR(MAX)  EXECUTE #TestAllCheckConstraints @TheResult=@OurFailedConstraints OUTPUT  SELECT @OurFailedConstraints AS theFailedCheckConstraints  EXECUTE # #TestAllUniqueConstraints @TheResult=@OurFailedConstraints OUTPUT  SELECT @OurFailedConstraints AS theFailedCheckConstraints  EXECUTE #TestAllForeignKeyConstraints @TheResult=@OurFailedConstraints OUTPUT  SELECT @OurFailedConstraints AS theFailedCheckConstraints |

Alternatively, you run the checks using PowerShell, as I’ll demonstrate shortly, for deploying constraint changes to databases with existing data. Either way, from the resulting JSON reports, you’ll see immediately any data that will cause problem when you enable constraints and rebuild those unique constraints and unique non-clustered indexes!

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | [    {      "success": "There were 1  check constraints that would fail data and no errors",      "FailedChecks": [        {          "BadData": {            "RowsFailed": "25",            "ConstraintName": "[CK\_Employee\_SickLeaveHours]",            "ConstraintTable": "[HumanResources].[Employee]",            "Expression": "([SickLeaveHours]>=(0) AND [SickLeaveHours]<=(65))",            "BadDataSample": [              {                "BusinessEntityID": 1,                "NationalIDNumber": "295847284",                "LoginID": "adventure-works\\ken0",                "JobTitle": "Chief Executive Officer",                "BirthDate": "1969-01-29",                "MaritalStatus": "S",                "Gender": "M",                "HireDate": "2009-01-14",                "SalariedFlag": true,                "VacationHours": 99,                "SickLeaveHours": 69,                "CurrentFlag": true,                "rowguid": "F01251E5-96A3-448D-981E-0F99D789110D",                "ModifiedDate": "2014-06-30T00:00:00"              },  …etc…  ] |

5. Fix the data, enable constraints and indexes

Now you devise a script that will modify the reported rows of data so that they comply with the constant conditions, run it on the newly built, and filled, databases, and then reenable all foreign key or check constraints.

|  |  |
| --- | --- |
| 1 | EXEC sp\_msforeachtable "ALTER TABLE ? WITH CHECK CHECK CONSTRAINT all" |

You enable the UNIQUE indexes using the keyword REBUILD instead of DISABLE. Caution is required, particularly as some developer may have used the DISABLE keyword as a lazy way of deleting the index!

|  |  |
| --- | --- |
| 1 | ALTER INDEX [AK\_SalesTaxRate\_StateProvinceID\_TaxType] ON [Sales].[SalesTaxRate] REBUILD |

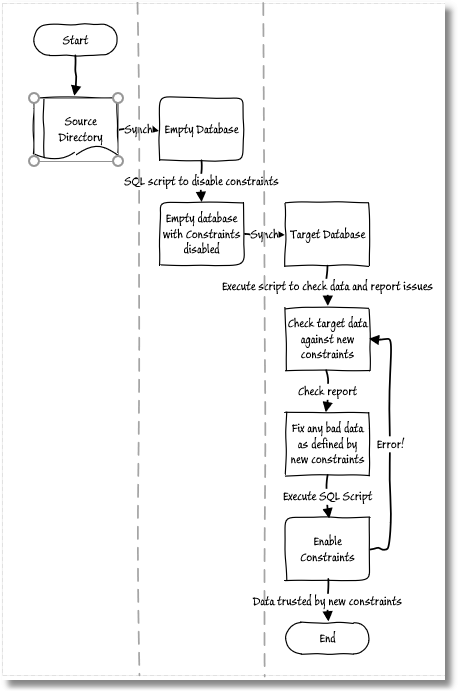
Testing and fixing existing data in a target database

Relational databases have a strong distinction between structure (schema), and data. This isn’t true of other types of database, and this can cause confusion. Although SQL Compare does its best to preserve existing data in a target database, it can’t do anything to make it match new constraints or unique indexes, and there is no guarantee that the data is now valid.

We’ll discuss how the process will work, both in cases where you do and don’t have a connection to the target database. We’ll see how to get the JSON reports, this time using PowerShell, which will tell in advance the full consequences for the data of the changes that are being deployed.

Finding the violating rows in a connected target database

The case where we are running a synchronization with a direct connection to the target database is simpler. Once again, we stat by building an empty database and then disabling its constraints and unique indexes. However, this database now becomes the source for a synchronization with a target data with existing data. As discussed earlier, SQL Compare will generate a deployment script that, when applied to the target, will leave constraints and unique indexes “off”. With this done the process proceeds as before:

[](https://www.red-gate.com/wp-content/uploads/2020/02/DeployToStaging.png)

Here is the PowerShell code to test all your check constraints, foreign key constraints, and unique indexes. There is no input because the script can get all the metadata from the synched database:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32 | Assert-Constraints @(  <# list of connection strings for each of the SQLservers that you need to execute code on #>      @{          'ServerConnectionString' = 'Server=MyServer;User Id=MyName;Persist Security Info=False';          #and a list of databases you wish the string-based (EG JSON report) from.          'Databases' = @('Shadrak', 'Meshak', 'Abednego'); # do all these databases          'RootDirectoryForOutputFile' = "$env:USERPROFILE\JSONDocumentation"; #the directory you want it in as subdirectories          'minimumCompatibilityLevel' = 130; #specify the minimum database compatibility level. We check!          'ScriptDirectory' = 'D:\Github\TestOutConstraints'; # where you store the project SQL files          'fileType' = 'json'; #the filetype of the files you save for each database for reports          # and now a list of all the temporary stored procedures you'll need.          'setupScripts' = @(            'ListAllUniqueIndexes.sql', 'ListAllForeignKeyConstraints.sql', 'ListAllCheckConstraints.sql',            'TestAllUniqueIndexes.sql', 'TestAllForeignKeyConstraints.sql', 'TestAllCheckConstraints.sql');      <#This lot are used process 1- for testing the loaded data to ensure it complies with the constraints#>          'FilesToExecute' = @(              @{                  'scriptFileName' = 'TestLoadedDataForCheckConstraints.sql'; `                  'OutputFileName' = 'CheckConstraintsReport'              },              @{                  'scriptFileName' = 'TestLoadedDataForUniqueConstraints.sql'; `                  'OutputFileName' = 'UniqueConstraintsReport'              },              @{                  'scriptFileName' = 'TestLoadedDataForFKConstraints.sql'; `                  'OutputFileName' = 'FKConstraintsReport'              }          )          'TearDownScripts' = @();      }  ) |

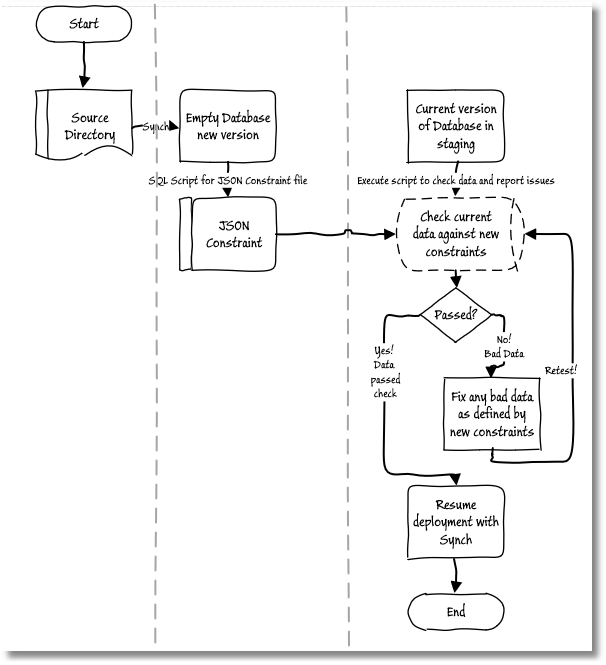
In this example here, there are three databases, Shedrak, Meshak and Abednego, all on MyServer, which have been upgraded to the latest version but with constraints disabled. You will get a report of all the problems that you’d have if the constraints were enabled WITH CHECK and disabled unique indexes were enabled with a REBUILD.

Armed with the report, you develop the data migration script, run it and re-enable all constraints and indexes such that they are trusted.

Finding violating rows using the JSON constraint file

There will be times when, of course, you can’t develop data migrations scripts and deploy changes in this way, with a direct connection to the target, such as when deploying to production. Also, in some cases, the development team can’t access Staging either and need to pass on the deployment scripts to an Ops team, for testing.

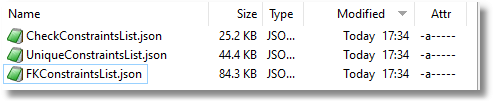
In such cases, the dev team would create the necessary JSON constraint script file from the new version of the database, and the Ops team would use it to test the data in the current version of Staging, and create a data migrations script, or send back a report to the development team so they can do it.

[](https://www.red-gate.com/wp-content/uploads/2020/02/DeployToProd.png)

To get the JSON constraint script file, we can run the following PowerShell on an existing built database at the new level. If there isn’t one, we can build an empty database from our source control directory. The only real difference is that this time we’re just saving the list of constraints into JSON:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32 | Assert-Constraints @(  <# list of connection strings for each of the SQLservers that you need to execute code on #>    @{      'ServerConnectionString' = 'Server=MyServer;User Id=PhilFactor;Persist Security Info=False';      #and a list of databases you wish the string-based (EG JSON report) from.      'Databases' = @('NewAdventureworks2016'); # the database we do it on      'RootDirectoryForOutputFile' = "$env:USERPROFILE\ConstraintDemo"; #the directory you want it in as subdirectories      'minimumCompatibilityLevel' = 130; #specify the minimum database compatibility level. We check!      'ScriptDirectory' = 'D:\Github\TestOutConstraints'; # where you store the project SQL files      'fileType' = 'json'; #the filetype of the files you save for each database for reports      # and now a list of all the temporary stored procedures you'll need.      'setupScripts' = @('ListAllUniqueIndexes.sql', 'ListAllForeignKeyConstraints.sql', 'ListAllCheckConstraints.sql',        'TestAllUniqueIndexes.sql', 'TestAllForeignKeyConstraints.sql', 'TestAllCheckConstraints.sql');        <#Save the lists of constraints as defined in the database #>      'FilesToExecute' = @(        @{          'scriptFileName' = 'GetListofAllForeignKeyConstraints.sql'; `          'OutputFileName' = 'FKConstraintsList.JSON'        },        @{          'scriptFileName' = 'GetListofAllUniqueConstraints.sql'; `          'OutputFileName' = 'UniqueConstraintsList.JSON'        },        @{          'scriptFileName' = 'GetListofAllCheckConstraints.sql'; `          'OutputFileName' = 'CheckConstraintsList.JSON'        }      )      'TearDownScripts' = @();    }  ) |

And here are the resulting JSON constraint files:



The Ops team can now check them against the data in the target database:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34 | Assert-Constraints @(  <# list of connection strings for each of the SQLservers that you need to execute code on #>    @{      'ServerConnectionString' = 'Server=MyOtherServer;User Id=PhilFactor;Persist Security Info=False';      #and a list of databases you wish the string-based (EG JSON report) from.      'Databases' = @('Adventureworks2016'); # do all these databases      'RootDirectoryForOutputFile' = "$env:USERPROFILE\ConstraintDemo"; #the directory you want it in as subdirectories      'minimumCompatibilityLevel' = 130; #specify the minimum database compatibility level. We check!      'ScriptDirectory' = 'D:\Github\TestOutConstraints'; # where you store the project SQL files      'fileType' = 'json'; #the filetype of the files you save for each database for reports      # and now a list of all the temporary stored procedures you'll need.      'setupScripts' = @('ListAllUniqueIndexes.sql', 'ListAllForeignKeyConstraints.sql', 'ListAllCheckConstraints.sql',        'TestAllUniqueIndexes.sql', 'TestAllForeignKeyConstraints.sql', 'TestAllCheckConstraints.sql');        <#Save the lists of constraints as defined in the database #>      'FilesToExecute' = @(            @{              'scriptFileName' = 'TestJSONForUniqueIndexes.sql'; `              'input' = @{ 'FileName' = 'UniqueConstraintsList.json' }; `              'OutputFileName' = 'DelayedUniqueConstraintsReport'            },              @{              'scriptFileName' = 'TestJSONForCheckConstraints.sql'; `              'input' = @{ 'FileName' = 'CheckConstraintsList.json' }; `              'OutputFileName' = 'DelayedCheckConstraintsReport'            },              @{              'scriptFileName' = 'TestJSONForForeignKeyConstraints.sql'; `              'input' = @{ 'FileName' = 'FKConstraintsList.json' }; `              'OutputFileName' = 'DelayedFKConstraintsReport'            }    )              'TearDownScripts' = @();            }  ) |

Here’s the output messages:

now running script S:\work\Github\TestOutConstraints\TestJSONForUniqueIndexes.sql on Adventureworks2016 database

WARNING: There were 2 indexes that does not match the data and no errors

now running script D:\Github\TestOutConstraints\TestJSONForCheckConstraints.sql on Adventureworks2016 database

WARNING: There were 1 check constraints that would fail data and no errors

now running script D:\Github\TestOutConstraints\TestJSONForForeignKeyConstraints.sql on Adventureworks2016 database

And here is there full report for rows that violate constraint conditions; I’ve added a couple of index duplicates to the check constraint violation:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69 | [    {      "success": "There were 1  check constraints that would fail data and no errors",      "FailedChecks": [        {          "BadData": {            "RowsFailed": "25",            "ConstraintName": "[CK\_Employee\_SickLeaveHours]",            "ConstraintTable": "[HumanResources].[Employee]",            "Expression": "([SickLeaveHours]>=(0) AND [SickLeaveHours]<=(65))",            "BadDataSample": [              {                "BusinessEntityID": 1,                "NationalIDNumber": "295847284",                "LoginID": "adventure-works\\ken0",                "JobTitle": "Chief Executive Officer",                "BirthDate": "1969-01-29",                "MaritalStatus": "S",                "Gender": "M",                "HireDate": "2009-01-14",                "SalariedFlag": true,                "VacationHours": 99,                "SickLeaveHours": 69,                "CurrentFlag": true,                "rowguid": "F01251E5-96A3-448D-981E-0F99D789110D",                "ModifiedDate": "2014-06-30T00:00:00"              },              {                "BusinessEntityID": 4,                "NationalIDNumber": "112457891",                "LoginID": "adventure-works\\rob0",                "OrganizationNode": "/1/1/1/",                "OrganizationLevel": 3,                "JobTitle": "Senior Tool Designer",                "BirthDate": "1974-12-23",                "MaritalStatus": "S",                "Gender": "M",                "HireDate": "2007-12-05",                "SalariedFlag": false,                "VacationHours": 48,                "SickLeaveHours": 80,                "CurrentFlag": true,                "rowguid": "59747955-87B8-443F-8ED4-F8AD3AFDF3A9",                "ModifiedDate": "2014-06-30T00:00:00"              },              {                "BusinessEntityID": 88,                "NationalIDNumber": "294148271",                "LoginID": "adventure-works\\betsy0",                "OrganizationNode": "/3/1/8/1/",                "OrganizationLevel": 4,                "JobTitle": "Production Technician - WC10",                "BirthDate": "1966-12-17",                "MaritalStatus": "S",                "Gender": "F",                "HireDate": "2009-12-18",                "SalariedFlag": false,                "VacationHours": 99,                "SickLeaveHours": 69,                "CurrentFlag": true,                "rowguid": "EBCDBA1C-6C1D-4D36-90F7-1893755C85E3",                "ModifiedDate": "2014-06-30T00:00:00"              }            ]          }        }      ]    }  ] |

The Ops team runs the resulting data migration script, essentially as a pre-deployment script, and then rerun the data checks. Hopefully, no further violating rows will be reported, and they can proceed to deploy the schema changes that will bring Staging up to the new version, with the new constraint definitions.

Running automated deployments

When deploying new constraints as part of an automated deployment process, then you have a couple of choices about when in the process to fix the data.

You can run the data migration script as a SQL Compare (or or SQL Change Automation) pre-deployment script. Be careful to obey the rules of pre-deployment code: do your own rollback on error and obey the convention of setting NOEXEC on if there is an error. However, what if the pre-deployment script runs, but then the subsequent deployment fails?

The alternative would be for the altered constraints and unique indexes to be committed to source control in a disabled state, as discussed earlier. In this case, you’d run the data migration script as part of a post-deployment script, using SQL Compare or SQL Change Automation, which then also re-enabled the constraints and indexes and checked that they were all ‘trusted’, before deployment is complete.

Summary

Even if your source and target databases have the same table names and columns, there is no guarantee that you can copy the data from one to the other successfully. If you are loading data from a different source, such as an external application, then all bets are off. Why is this? Well, it is due to constraints in the target database picking up duplicates, bad data and problems with referential integrity. These must be fixed, otherwise you will always face a performance deficit. You will probably also suffer a lot worse from the data problems too. It all gets worse if you, as a developer, don’t have direct access to the data, or you as an Ops person don’t have the time or expertise to do the job.

This code is designed to try to prevent this sort of problem from happening. It checks data against the constraints in the target database and gives you a list of the data that needs to be fixed before you enable constraints, or before you start an automated process such as an automated build.

Several times in my professional life, I’ve had to start an automated build before leaving work in the evening and hoping that, by the morning, the build process is finished. Almost all the time was spent loading the data. If the build is broken, then things can be a bit tense, while the team try to work out how to explain a day’s delay in a release. I’d have given a lot for this code, because I could have then built the new release without its data, copied out the constraint information, tested it on a previous version of the database and got a report of any likely problems