



Hugo
Kornelis

14 January
2014

24763 views

0



Hugo Kornelis

14 January 2014

24763 views

13

0

Generating Test Data in TSQL

To test SQL, you need test data. There are usually many reasons why you can't use production data. Although it is usually enough to use a utility to generate test data, sometimes your requirements will compel you to resort to code to supplement this. Hugo shows how he used SQL and C# to generate large volumes of test data involving related columns and complex distributions.



**DevOps, Continuous Delivery
& Database Lifecycle**

Management

Continuous Integration

Generating test data can be a pain. And there's no simple one-size-fits-all solution, since different problems require different types and amounts of test data. To check a simple query for correct results, just a few rows to cover the various cases will do. For testing more complex queries or complete applications, the same basic principle applies, except that you cannot cover all combinations of all possible cases, because the test set would grow too large. Limiting that is a difficult task – and not the subject of this article.

This article is about what is arguably the most difficult type of test data generation: test data for a performance test. The problem here is not the size – okay, you do need a pretty big database for any serious performance test, but it is easy to generate a big database: Time-consuming, maybe, but easy. Just set up a simple script to keep adding either identical or random rows to the database and let it run as long as needed. Unfortunately, SQL Server is too smart to let us get away with such a simple approach. Because of the way that the optimizer works, running the exact same query on tables with the exact same number of rows but with a different *distribution* of the data can result in different execution plans. Unless your test database is very similar to production in both size and distribution, all your testing efforts may well be in vain!

The easiest way to get a test database that has the same size and the same data distribution as the production database is of course to use a complete copy of the production database. But that is often not an option – for instance because it's blocked by legislation (privacy laws) or company policy (data protection protocols); or because you are developing a new application or anticipating huge growth, so the production database doesn't exist, or it is not representative of the nature of the data that you are expecting.


There are tools available to fall back on. I know of one: [Red Gate's SQL Data Generator](#). This product has some great features such as, for instance, the option to generate names and addresses that are completely fake, but still believable. I was also impressed to see how smart it is at filling out default generation algorithms for each column. For some purposes, the provided generators and the distribution of data values may be good enough. Even better, if you can get permission to use an

anonymized version of the production database, is to use SQL Data Generator to generate sensitive data such as names, credit card numbers to replace the real data [as described here](#).

But this tool has its limitations too. Out of the box, it will not generate data with specific distributions such as normal distribution, log-normal distribution or the Pareto distribution. All data it generates gives a flat or uniform distribution where every value is equally likely. And every column is populated independently: There is not a built-in option to, for instance, populate a `ShipDate` column with a date that is between 5 and 15 days from the `OrderDate` value in the same row, or to make sure that every row that uses product number 17319 uses the same product name and unit price – essential when working with denormalized tables in a data warehouse. You can, of course, [create a new generator for special requirements](#) using a .NET language such as C#. SQL Data Generator [does support Python scripts](#) to allow you to tweak the data generation (and I have been told that this support has been improved since the last version I worked with), so if my next data generation involves a situation where the Red Gate tool can handle most of the columns, I will definitely look into this and see if I can learn enough Python to complete the task. But in a situation where the tool falls short for the majority of the columns and needs extensive tweaking, I decided it is better instead to code the entire data generation logic in my favorite language: T-SQL.

This article describes how I created one such data generation script. I'm not describing this because I think that you would ever need the same database or the same script – you probably won't. I'm showing you some techniques in this script because I have solved various problems that, in one form or

another, are relevant to many data generation tasks. When you have to generate data, your challenges will be different – but perhaps, they are similar enough to mine that you can adapt my solution to your situation, instead of having to reinvent a completely new set of wheels.



The SQL Code to go with this article can be downloaded by clicking the link at the bottom of the article, [or by clicking here](#).

My particular use case

I decided to roll my own solution, rather than to use SQL Data Generator, because the particular nature of my use case. I wanted to fill a table that has no names, addresses, bank accounts, or credit card numbers at all (which are the areas where the tool really shines), but that does have a lot of related columns and denormalized data (which SQL Data Generator doesn't handle well).

Two of the presentations that I regularly deliver at community events are both about the SQL Server 2012 [Columnstore Index](#) feature. This feature is most useful in a Data Warehousing (DW) environment – as such, it really needs a very big table if you want to see where it shines, where its limitations start to hurt, and how to work around those limitations to make it shine again. And with “very big”, I am not talking about thousands of rows, and not even about a few million – I like to have at least a hundred million rows when I demonstrate the Columnstore Index.

I have previously delivered these presentations using a database that was given to me by Microsoft, with permission to use but not to redistribute, but that never felt right to me. When I do a presentation,

I want the attendees to be able to download and play with my scripts afterwards. Well, the scripts are available for download – but without the database they run on, opportunities for playing with them are rather limited! When, therefore, I was asked to deliver [one of my Columnstore Index presentations](#) at the [SQL Server Connections conference](#) (Las Vegas, Sep 30-Oct 4, 2013), and I found myself in the rare possession of enough hours to spare, I could finally do what I wanted to do all the time – create my own demo database to use in my presentations on the Columnstore Index.

Decision time

My first decisions were easy to make. I decided to base my big demo database on [AdventureWorksDW2012](#) which is the Data Warehouse version of `AdventureWorks2012`, and to make a very big version of just one of the fact tables. I wished to merely restore a copy of that database to my new database (`AdventureWorksDW2012XL`) and add that single table only, with the data in all dimension tables already there.

A normal “big” DW database would probably have small dimension tables, but lots of rows in `all` fact tables, so it wasn’t entirely realistic to increase only one fact table. But I plan to base all my demos on a single fact table only. I could in fact even decide to drop all the other fact tables and leave only the dimension tables and this single big fact table.

The table I use as my starting point is `FactResellerSales`. Again: a simple choice. This was the table used for the same purpose in the Microsoft-supplied demo database I used so far, so I was already familiar with it. Its columns are varied enough to allow me to set up some nice queries that

can showcase the Columnstore Index features and limitations nicely.

So I created a new table, `FactResellerSalesXL`, with the exact same schema, indexes, and constraints as the original. With one exception – a table the size I intend it to be should be partitioned, and I decided to stick with the more or less standard pattern of partitioning by date. Unfortunately, this made it impossible to enforce the actual primary key (on order number and order line number), because a partitioned table requires that the partitioning column is included in the PRIMARY KEY and in all UNIQUE constraints.

Valid dates

The data in the `AdventureWorks` sample databases is already a few years old. The actual date range is not really relevant for most demos, but since I'll generate the data anyway, I opted to add an extra touch of realism by moving the data nearer to the present. The data I generate spans the period from January 2009 up to and including August 2013. That's a total of 56 months.

That does require me to make some changes to two other tables in my database. The immediate obvious one is `DimDate`, the date dimension table. I had to add dates for the calendar years 2011, 2012, and 2013. The query that is used to do that is big and ugly, because of the large number of columns used in this dimension table, but also very straightforward. I could have made it simpler by populating dummy data in the columns I will never use in my demos anyway – but again, I like to be as realistic as possible.

A more unexpected change is required in the `DimProduct` table. This table is set up as a [type 2 slowly changing dimension table](#), which means that

when properties of a product change, the old row is kept with an end-date equal to yesterday; and a new row with start-date equal to today is entered. The data in this table was already inconsistent, because some of the end-dates were earlier than the start-dates – and all start and end dates predate the period for which I will generate data. So to make this a bit more realistic, I modified the start and end dates such that all data is consistent, and the valid products and their properties actually change during the relevant period.

The orders

The `FactResellerSales` table is a fact table that combines data for order headers and for order lines. I could of course decide to generate completely random order numbers and order line numbers (especially since I had to drop the PRIMARY KEY constraint on the combination of those columns), but that would not be very realistic. Instead, I decided to generate actual orders, each with a random number of order lines. I'll discuss the generation of order lines later, for now it's sufficient to understand that the *average* number of order lines will be five per order – this is relevant, because I want to ensure that I end up with approximately a hundred million rows in the `FactResellerSalesXL` table, which means I'll have to generate approximately twenty million orders.

After playing with the numbers a bit, I decided to mimic a business that is growing at a steady pace. I start with 5,000 orders on the first day (January 1st, 2009) and grow that number by 0.1% per day in 2009, 2010, and 2011. In 2012 and 2013, the growth increases to 0.15% per day. A quick calculation reveals that this will produce a total of almost 24.8 million orders over the whole period – close enough;

but this algorithm produces a very unrealistic pattern of orders per day. Actual businesses see large variations in the business day over day. So I use the above algorithm to calculate a *trend* of the average orders per day, but then use a random factor to set the actual number of orders to somewhere between 90% and 110% of that average.

Depending on the type of business you want to mimic, this can be okay (with maybe a smaller or larger variation), or you may need to tweak the algorithm for season peaks or for higher or lower sales on some days in the week.

Order number

Orders need an order number, of course. The original `FactResellerSales` table has order numbers in the format "SO" + a five-digit number. (Yeah, I agree, anything that has letters in it should not be called a "number". I blame Microsoft. It's not even the worst column name they've come up with). This format allows for 100,000 different values – not enough. But a small variation fixes this – instead of prefixing the numbers with the letters "SO", I'll prefix them with any combination of two letters. Since there are $26 * 26$ two-letter combinations, this gives me 67.6 million different values: This should be enough.

My first attempt used some simple logic to concatenate a five-digit counter with two letters; after each order I would increase the counter; when it hits 100,000, I reset it, increase the first letter until it goes beyond "Z", at which point I increase the second letter and reset the first to "A". Simple, effective – but it forced me to process each order individually. SQL Server is not optimized for that, and I try to avoid it. In this case, I already had a loop to process each individual date in the relevant period; creating a second loop to iterate over orders

and nesting that in the other loop was not acceptable.

The solution I found was to calculate the order number from an actual count of the number of orders already generated. This means that now, after all those years, I finally found a good use for the time I spent in college trying to understand [non-decimal numeral systems](#), such as binary (base 2), hexadecimal (base 16), octal (base 8), duodecimal (base 12), or even sexagesimal (base 60). In this case, I could interpret the order number as a number in a weird combination of decimal (our familiar base 10 numerals) and hexavigesimal (base 26 – and yes, that word actually exists, I did not make it up, though I wish I had).

Another way to put it is to say that I divide the order count by 100,000, use the remainder (always a number between 0 and 99,999) for the five numbers at the end, and use quotient for the two letters – by representing it in base-26. That is, I divide this quotient by 26, use the remainder of THAT division (0-25) to find one letter, and the quotient for the other. A nice extra feature of this approach is that it is now extremely easy to add extra letters – in the code, I have added a comment to show how to change the expression to add a third letter, boosting the available number of orders to over 17.5 billion.

Order lines

I have already mentioned that every order will have a random number of order lines. In my quest for realism, I decided not to simply pick a number between 1 and a maximum, because that would be a very unrealistic distribution. In a real company, you will have a relatively large number of orders with only a few lines, and a much smaller number of big orders. I was unable to find a way to get such a distribution of random numbers using set-based T-

SQL code only, so I decided to make a quick detour to .Net for this – I wrote a simple CLR user-defined function to start with 1, and then keep adding 1 in a loop with a 80% to keep looping and a 20% chance to stop. This will result in 20% of the results being 1, 16% being 2, 12.8% being 3, 10.24% being 4, and so on. I added a hard cap at 1000, because you have to draw a line somewhere. (I could have chosen 32,000 and still be able to use a smallint for the results, but 1,000 seemed a nice number). The chance of actually really needing this cap is not very big – the chance of staying in the loop for a thousand times in a row is only 1.23×10^{-97} . Or to put it in another way, if all seven billion people on the earth run the script to generate this database every second (which means they will all need a computer that is a LOT faster than mine!), then the number of years that will pass until it is likely that the limit has been reached at least once is still a 72-digit number that I will not try to pronounce.

Since I am not very experienced at C#, writing the code was a challenge. Okay, the loop itself was simple enough, even for me, but how to deal with the randomizer object? My first attempt had the declaration of the randomizer as part of the method itself. That appeared to work well when I did some test calls – but when I used the function in a set-based query, I often got the same result on every line. The reason for this turned out to be that the randomizer was reseeded on every call, and the new seed was based on the internal system clock – so if the calls were quick enough, they'd be the same every time. Now there may have been a way around this, but I wanted to avoid the overhead of reseeding the randomizer on every call anyway, so I decided to move the randomizer outside the function body and make it a global object instead. When Visual Studio started to complain that static objects have to be marked read only, I had my doubts whether this was

going to work – somewhere in the black box that is called “randomizer”, there is a seed that is changed on each call, so I expected the “readonly” attribute to cause errors. However, much to my surprise, this turned out not to be the case. I now got nice random numbers with the expected distribution, and they were different on every call – both for individual calls and within a set-based query. I must admit that I still don’t really understand *why* this works – but it does, and that’s what counts! (Readers, please do feel free to use the comments section if you can explain to a C# noob like me how and why this works).

Order header columns

The AdventureWorksDW2012 database uses a typical data warehouse design – that is, the tables are heavily *denormalized*. For the `FactResellerSales` table, this means that, even though there are rows for every order line, most of the columns are actually order-header information, and should be the same for each order-line of the same order. That’s why I generate these values, or at least the base values from which they can be computed, at the order level.

Six of the order-related columns are date-related. We have order date, ship date, and due date; and then we have the same three columns again, now as integer instead of datetime, to implement the foreign key into the `DimDate` dimension table. Data warehouse designs usually suggest using meaningless integer numbers as surrogate keys for the dimension tables. The date dimension in the AdventureWorksDW2012 does indeed use integer numbers, but they are far from meaningless – they are the YYYYMMDD representation of the date, as an integer number. This can be quite confusing – when I see a column `OrderDateKey` and the

contents look like 20130815 and 20091205, I tend to think of them as dates in the twenty-first century rather than integer values of a little over twenty million. This misunderstanding has at times resulted in painful errors in my demo code – making me glad I always test my code before presenting! Anyway, armed with this knowledge, it is easy to generate the correct “key” values from the actual dates. The order date is also already known, since I generate my orders by date. For the other dates, the original `FactResellerSales` table always has a ship date that is either 7 or 8 days after the order date, and a due date that is always 5 days after the ship date. I decided to increase the spread a little – I generate the ship date to be anywhere between 5 and 14 days from the order date, and I retained the five day period between ship date and due date.

For the sales territory, I decided to introduce some (admittedly very simple) change through time. All orders until 2010, are for one of the sales territories 1 – 5, which correspond to the five regions in the United States. In 2011, business expands outside the US, so all sales territories 1 – 10 can be selected by the randomizer for orders in 2011 and 2012; as of 2013 the office in the North-West of the United States closes, so now only sales territories 2 – 10 can be selected. It’s probably not totally realistic to spread all sales evenly across all territories, especially when the business has just expanded into new areas, but I had to draw the line between realism and readable code *somewhere*.

I did not use a randomizer for the currency. In the original `FactResellerSales` table, there is a logical relationship between sales territory – for instance, all orders with sales territory equal to 6 (Canada) have `CurrencyKey` 19 (CAD). The only exception is that in sales territory 7 (France), only 13 % of sales use Euros, the remainder uses U.S.

Dollars. I have no idea if that was deliberate or a mistake when Microsoft created this database, but I decided that for my own data, all orders would use the “natural” currency for the sales territory they are in.

The rest of the columns are fairly simple. There are a total of 701 resellers, and I use a simple distribution where each of them has the same chance. Given the total number of orders, that will mean that in the end they will all have almost the same amount of orders. Again, a tradeoff between effort and realism. The same applies for the `EmployeeKey` column – the existing table uses numbers 272 and 281 through 296 – and if you inspect the `DimEmployee` table, you will see that these are exactly the employees with the `SalesPersonFlag` column equal to 1: So no coincidence. There are a total of 17 sales persons, and I simply generate a random number from 1 to 17 and use a CASE expression to translate that to the corresponding `EmployeeKey` number. If you need more realism for either of these columns, feel free to modify the script (using any of the techniques described in this article, or something you come up with yourself).

For the revision number, the original `FactResellerSales` table had a distribution where roughly 99.9% of all orders have revision number 1, and the remaining 0.1% have revision number 2. I decided that this is good enough for me.

The carrier tracking number in the original table looked like a ten-digit hexadecimal number with two dashes inserted. So I generate a random number in a range that includes all possible ten-digit hexadecimal numbers, convert that first to binary and then (using a little known feature of the `CONVERT` statement) to the string representation of

the hexadecimal equivalent. I wrapped it up by using STUFF to insert the two dashes. Stuff is generally intended to replace a substring with another substring, but you can make it insert data by “replacing” a zero-length string.

Finally, the `CustomerPONumber` always starts with “PO”, followed by a ten-digit number in the range 1000000000 – 9999999999; this is very easy to generate. I did not attempt to enforce uniqueness for either this column or for the carrier tracking number, because I am not even planning to use any of these columns at all in my demos.

Order line columns

There are a lot of columns that correspond to the order line. But most of them are determined using formulas and the other columns. The only thing I had to randomly determine are the product, the quantity, and the promotion. With the product known, the unit price (dealer price) and standard cost can both be fetched from the `DimProduct` dimension table. The extended amount is the multiplication of the unit price and the quantity; the discount amount can be found by multiplying that by the discount percentage for the order (see previous paragraph), and tax and freight are 8% and 2.5% of the total amount minus discount; and so on.

I spend quite some time thinking about the selection of the product for each order line. This was an area where I really wanted to go for realism – I went through the effort to create a realistic population for the `DimProduct` table; I will not let that go to waste! So my first version would just at random pick one of the products that are valid on the order date. To do that, I created a temporary table of valid products, plus a second temporary holding only the dates of a change in the `DimProduct` table – the latter table ensures that I only have to rebuild the

former after an actual change, instead of for every day in the main loop for order date. (In relation to the total running time of the script, this is probably a futile optimization, but I simply cannot not optimize when I see the chance).

This version turned out to make all products with the same period of validity to appear in approximately the same number of orders – totally unrealistic! My second version modified this by adding a bit of realism – I assigned weights to the articles that correspond to their price, such that cheaper products are included more often than expensive products. But now, the total amount (quantity * price) for each product was about the same. Again, not what I wanted, for this is what some of my demo queries report on, and it looks weird if all rows report approximately the same amount! So, for my third version, the square root of the standard cost is used as the modifier instead of just the standard cost for each product- and to ensure some variation among items of the same price (fourth version!), I added the absolute value of the cosine of the `ProductKey` (a more or less random value between 0 and 1, but always the same for each product) as an additional modifier.

I am aware that there will probably be some orders with the same item on different order lines. Normally, these would be combined on a single line. I could have added logic for that, or changed the logic to make sure no two order lines of the same order will ever be for the same item, but I decided that this would complicate the script and slow down the generation too much. If I ever need data where no order can have two order lines for the same amount, I will probably change the script to first run the generation as is, then repeat the same random data generation for only the unwanted duplications, in a loop that runs until there are no more

duplicates. Obviously, this will normally go relatively fast if the number of available products is sufficiently higher than the number of order lines – but if you have 75 order lines and only 80 available products, this method spells disaster!

I mentioned using weights to make some products appear more often than others. For these “weighted random picks”, I used an interesting method that I first found in the C code of [an open source game](#). The principle of these weighted random picks is that some items appear more often than others. If you have a bag with one red, one blue, and one green ball, each ball will be picked one third of the time. Add two reds and a blue, and now half the picks will be a red ball, blue will still be one third, and green is down to one in six. The red ball has weight three, the blue ball has weight two, and green has weight one. When all weights are integer numbers, this can be mimicked on a computer by adding multiple copies of the item to a table and then picking one random row. But an easier method is to add up all weights to create numerical ranges. One to three for red, four and five for blue, and six for green. Generate a random number between one and six, and pick the corresponding ball. The next step after this is to use a random number that is not an integer, but a floating point. Now, the ranges are 0-3 for red, 3-5 for blue, and 5-6 for green (all with lower bound inclusive and upper bound exclusive). The random number is a floating point number between 0 (inclusive) and 6 (exclusive), and again you get a distribution of 50% red, 33.3% blue, and 16.7% green. This algorithm has the advantage that it can also be used with weights that are not an integer – which is great, because the weights I use are absolutely not integers! The base value is 100 divided by the square root of the price, with an additional modifier ranging from 0 to 1 (the absolute value of the cosine of the `ProductKey`).

For the quantity of the item ordered. I have considered going fancy with a formula that would favor higher numbers for cheaper items, but I was afraid that this might undo the spread in total (price * quantity) over all the generated rows. So I simply used the same CLR function that I also used for the number of order lines per order, giving a fairly high number of low values and a lower number of high values. I did consider creating a second version of the CLR with the chance to stay in the loop increased from 80% to 85% or 90% to get overall higher numbers, or creating a version that would take the percentage as a parameter, but I decided that the version I already had was good enough.

The final choice was for the `PromotionKey` column. Looking at the `DimPromotion` table it references, I see that there are five rows for volume-based discount, and a bunch of rows for very specific discounts. I did not want to randomly set completely inapplicable discounts, so I decided to ignore those specific discounts and assign the `PromotionKey` for the volume-based discount that matches the quantity of the item ordered.

The actual script

With all these decisions made, writing the script was a fairly basic exercise. Sure, it is a long script so it took some time (and some bug hunting and error fixing) to get it completely right. But the code used to tie all my decisions and implementation choices together is all fairly basic. The full script is attached to this article.

When you download and run the script, there are three things in particular to be aware of.

Firstly, the path names of the data and log file and of the AdventureWorksDW2012 backup that will be restored at the start of the script are all hardcoded,

and will probably not work on your system. You will have to change them before running the script.

Secondly, you will need a lot of available disk space. The data file is allocated at 22 GB. Since I do not run the script in a transaction and I set the database to simple recovery, the log file does not need more than just a single GB. (I have added autogrow settings for both files, but I have not seen them being used when running the script.)

The third (and maybe most important) consideration is that you will need time – a lot of time! The script runs for almost five and a half hours on my laptop! (And unless you limit the resources available to SQL Server, it will use up all your memory and most of the available CPU cycles; my laptop became totally useless for any other work when I first executed this script, and remained so until I forced SQL Server to release resources by restarting the service).

There are a lot of comments in the script, so I will only highlight a few things here. For instance, the indexes on the table – it is usually faster to load data first and only then create indexes, and this script is no exception. I didn't write down the numbers, but I did run a performance comparison between creating indexes before or after the population, and this version won. I have not run a similar test for the various constraints, so maybe the execution time can be brought down a bit by postponing the constraint creation to the end of the script as well. (If you do, then don't forget to create the constraints with the `WITH CHECK` option!). You will also notice that I included various `RAISERROR` statements to report the progress of the script. I always like to do this for long-running scripts, so that I can see it is still busy, and also see which parts are running slow (in case I need to do further optimizations). If you switch SSMS to "output to

text” (Ctrl-T) before running the script, you will see them appear. I did notice that, after some time, the progress report messages appear to stop for a while – and then a whole bunch appears. This is normal behavior for `RAISERROR` with default settings, but should not happen when using the `WITH NOWAIT` option. But apparently, that option stops having effect after using it too often in a single batch.

Where I have to generate random numbers, I do not simply call `RAND()` – that will return the same number for all rows in the query, so would in this case only be usable if I wrote the entire script as purely procedural logic. Instead, I pass in an argument (that will be used as the seed). That argument has to be different for each row and for each occurrence of the function in the query; that is a property exhibited by `NEWID()`. Note that `NEWID()` itself also looks a bit random, but that generation algorithm is not random enough for my purpose. That’s why I combine `NEWID()` (for getting a new seed for each call) with `RAND` (for getting a better random distribution). I also have to add in `CHECKSUM`, because the seed passed in to `RAND` has to be an integer, and the result of `NEWID()` is a `uniqueidentifier` – there is no way to actually convert a `uniqueidentifier` to an integer, but `CHECKSUM` is able to generate an integer out of each `uniqueidentifier`.

Conclusion

After running this script, I had the demo database I needed. I immediately made a backup, because it is far quicker to restore from backup than to re-run the script. And because the backup can be compressed, it takes far less space. I now simply drop the database when I don’t need it, and restore it from the backup when I do a presentation on columnstore indexes.

You can run the same script, and get a similar database – though not exactly the same, because of the random numbers. But more importantly, you can use the various methods I used to influence the distribution of random data when you need to generate test data that is random, yet exposes specific patterns.

One final note – if you want to experiment or demo with Data Mining features that try to find correlations by analyzing data, this method is probably not ideal. Most of the data I generated is random, and unless there are serious weaknesses in the random number algorithm used, the numbers generated will not be correlated. So the only correlations you will find are those I explicitly coded in – and those are probably way too obvious. Generating a good sample data set to use for Data Mining is probably only possible by either writing very smart code to force very subtle correlations, or by acquiring and anonymizing an existing real data set.

**DevOps,
Continuous
Delivery &
Database
Lifecycle
Management**



[Go to the Simple Talk library](#) to find more articles, or visit www.red-gate.com/solutions for more information on the benefits of extending DevOps practices to SQL Server databases.

