Generate a set or sequence without loops – part 1

sqlperformance.com/2013/01/t-sql-queries/generate-a-set-1

Aaron Bertrand

January 16, 2013

There are many use cases for generating a sequence of values in SQL Server. I'm not talking about a persisted `IDENTITY` column (or the new `SEQUENCE` in SQL Server 2012), but rather a transient set to be used only for the lifetime of a query. Or even the simplest cases – such as just appending a row number to each row in a resultset – which might involve adding a `ROW_NUMBER()` function to the query (or, better yet, in the presentation tier, which has to loop through the results row-by-row anyway).

I'm talking about slightly more complicated cases. For example, you may have a report that shows sales by date. A typical query might be:

```
SELECT
  OrderDate  = CONVERT(DATE, OrderDate),
  OrderCount = COUNT(*)
FROM dbo.Orders
GROUP BY CONVERT(DATE, OrderDate)
ORDER BY OrderDate;
```

The problem with this query is that, if there are no orders on a certain day, there will be no row for that day. This can lead to confusion, misleading data, or even incorrect calculations (think daily averages) for the downstream consumers of the data.

So there is a need to fill those gaps with the dates that are not present in the data. And sometimes people will stuff their data into a #temp table and use a `WHILE` loop or a cursor to fill in the missing dates one-by-one. I won't show that code here because I don't want to advocate its use, but I've seen it all over the place.

Before we get too deep into dates, though, let's first talk about numbers, since you can always use a sequence of numbers to derive a sequence of dates.

Numbers table

I've long been an advocate of storing an auxiliary "numbers table" on disk (and, for that matter, a calendar table as well).

Here is one way to generate a simple numbers table with 1,000,000 values:

```
SELECT TOP (1000000) n = CONVERT(INT, ROW_NUMBER() OVER (ORDER BY s1.[object_id]))
INTO dbo.Numbers
FROM sys.all_objects AS s1 CROSS JOIN sys.all_objects AS s2
OPTION (MAXDOP 1);

CREATE UNIQUE CLUSTERED INDEX n ON dbo.Numbers(n)
-- WITH (DATA_COMPRESSION = PAGE)
;
```

Why MAXDOP 1? See Paul White's blog post and his Connect item relating to row goals.

However, many people are opposed to the auxiliary table approach. Their argument: why store all that data on disk (and in memory) when they can generate the data on-the-fly? My counter is to be realistic and think about what you're optimizing; computation can be expensive, and are you sure that calculating a range of numbers on the fly is always going to be cheaper? As far as space, the Numbers table only takes up about 11 MB compressed, and 17 MB uncompressed. And if the table is referenced frequently enough, it should always be in memory, making access fast.

Let's take a look at a few examples, and some of the more common approaches used to satisfy them. I hope we can all agree that, even at 1,000 values, we don't want to solve these problems using a loop or a cursor.

Generating a sequence of 1,000 numbers

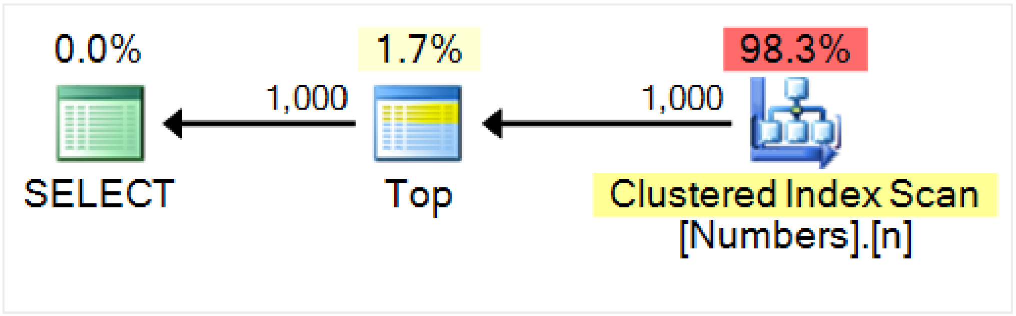
Starting simple, let's generate a set of numbers from 1 through 1,000.

Numbers table

Of course with a numbers table this task is pretty simple:

```
SELECT TOP (1000) n FROM dbo.Numbers ORDER BY n;
```

Plan:

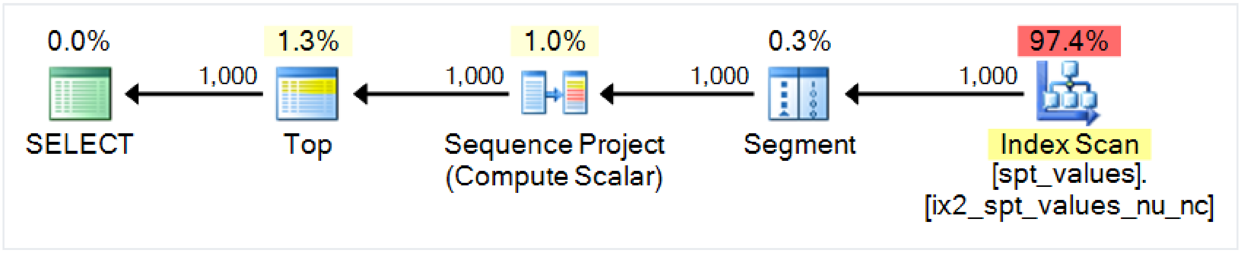


spt_values

This is a table that is used by internal stored procedures for various purposes. Its use online seems to be quite prevalent, even though it is undocumented, unsupported, it may disappear one day, and because it only contains a finite, non-unique, and non-contiguous set of values. There are 2,164 unique and 2,508 total values in SQL Server 2008 R2; in 2012 there are 2,167 unique and 2,515 total. This includes duplicates, negative values, and even if using `DISTINCT`, plenty of gaps once you get beyond the number 2,048. So the workaround is to use `ROW_NUMBER()` to generate a contiguous sequence, starting at 1, based on the values in the table.

```
SELECT TOP (1000) n = ROW_NUMBER() OVER (ORDER BY number)
  FROM [master]..spt_values ORDER BY n;
```

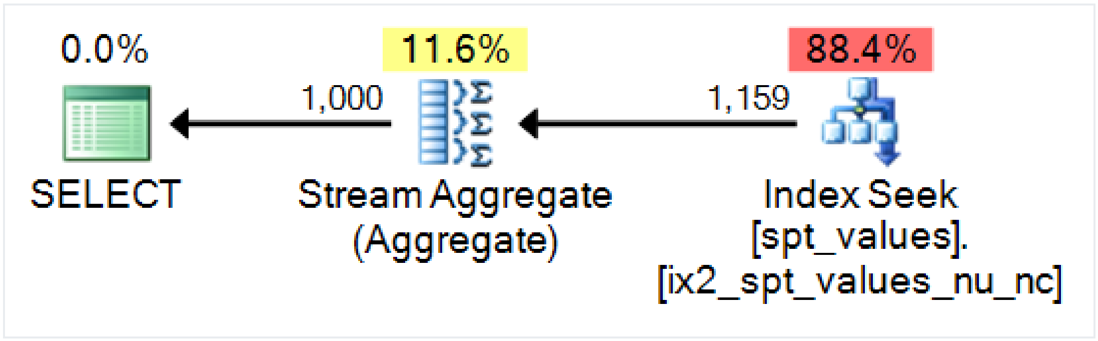
Plan:



That said, for only 1,000 values, you could write a slightly simpler query to generate the same sequence:

```
SELECT DISTINCT n = number FROM master..[spt_values] WHERE number BETWEEN 1 AND
1000;
```

This leads to a simpler plan, of course, but breaks down pretty quickly (once your sequence has to be more than 2,048 rows):



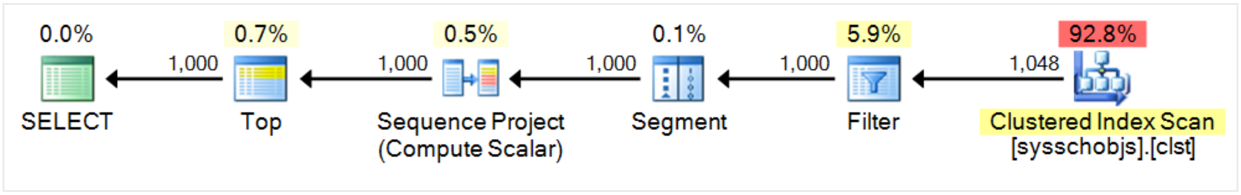
In any case, I do not recommend the use of this table; I'm including it for comparison purposes, only because I know how much of this is out there, and how tempting it might be to just re-use code you come across.

sys.all_objects

Another approach that has been one of my favorites over the years is to use `sys.all_objects`. Like `spt_values`, there is no reliable way to generate a contiguous sequence directly, and we have the same issues dealing with a finite set (just under 2,000 rows in SQL Server 2008 R2, and just over 2,000 rows in SQL Server 2012), but for 1,000 rows we can use the same `ROW_NUMBER()` trick. The reason I like this approach is that (a) there is less concern that this view will disappear anytime soon, (b) the view itself is documented and supported, and (c) it will run on any database on any version since SQL Server 2005 without having to cross database boundaries (including contained databases).

```
SELECT TOP (1000) n = ROW_NUMBER() OVER (ORDER BY [object_id]) FROM sys.all_objects
ORDER BY n;
```

Plan:

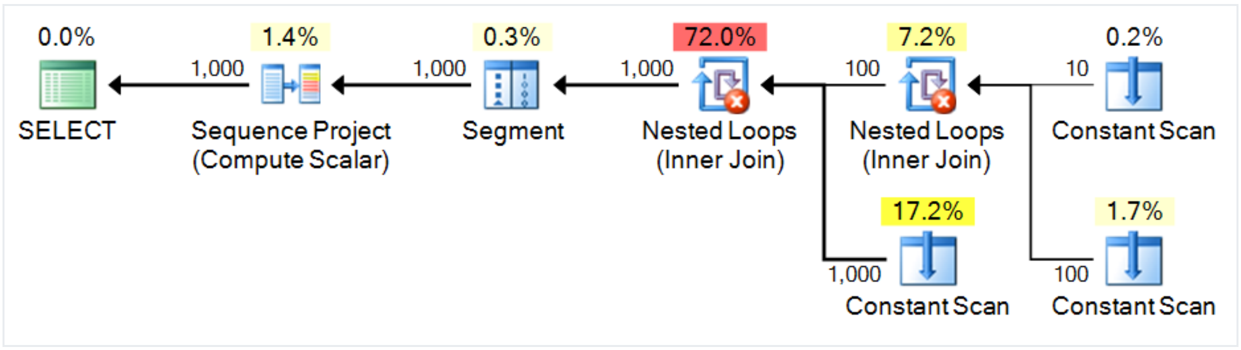


Stacked CTEs

I believe Itzik Ben-Gan deserves the ultimate credit for this approach; basically you construct a CTE with a small set of values, then you create the Cartesian product against itself in order to generate the number of rows you need. And again, instead of trying to generate a contiguous set as part of the underlying query, we can just apply `ROW_NUMBER()` to the final result.

```
;WITH e1(n) AS
(
    SELECT 1 UNION ALL SELECT 1 UNION ALL SELECT 1 UNION ALL
    SELECT 1 UNION ALL SELECT 1 UNION ALL SELECT 1 UNION ALL
    SELECT 1 UNION ALL SELECT 1 UNION ALL SELECT 1 UNION ALL SELECT 1
), -- 10
e2(n) AS (SELECT 1 FROM e1 CROSS JOIN e1 AS b), -- 10*10
e3(n) AS (SELECT 1 FROM e1 CROSS JOIN e2) -- 10*100
  SELECT n = ROW_NUMBER() OVER (ORDER BY n) FROM e3 ORDER BY n;
```

Plan:

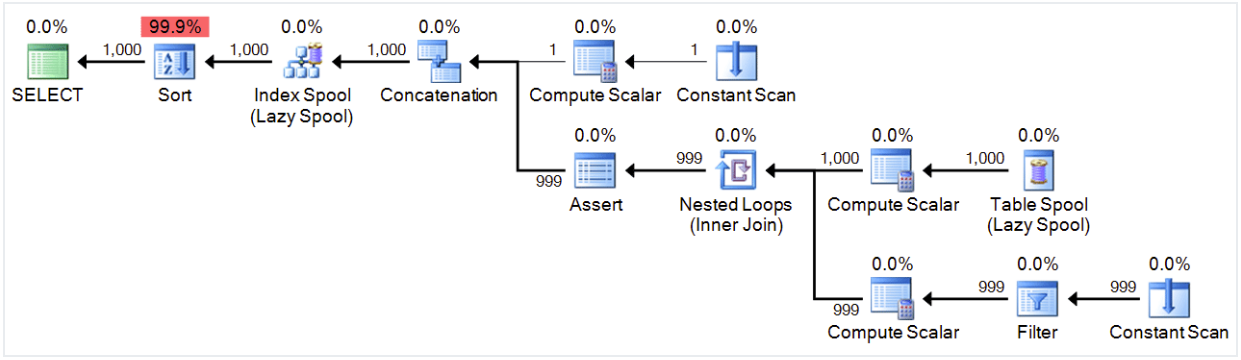


Recursive CTE

Finally, we have a recursive CTE, which uses 1 as the anchor, and adds 1 until we hit the maximum. For safety I specify the maximum in both the `WHERE` clause of the recursive portion, and in the `MAXRECURSION` setting. Depending on how many numbers you need, you may have to set `MAXRECURSION` to `0`.

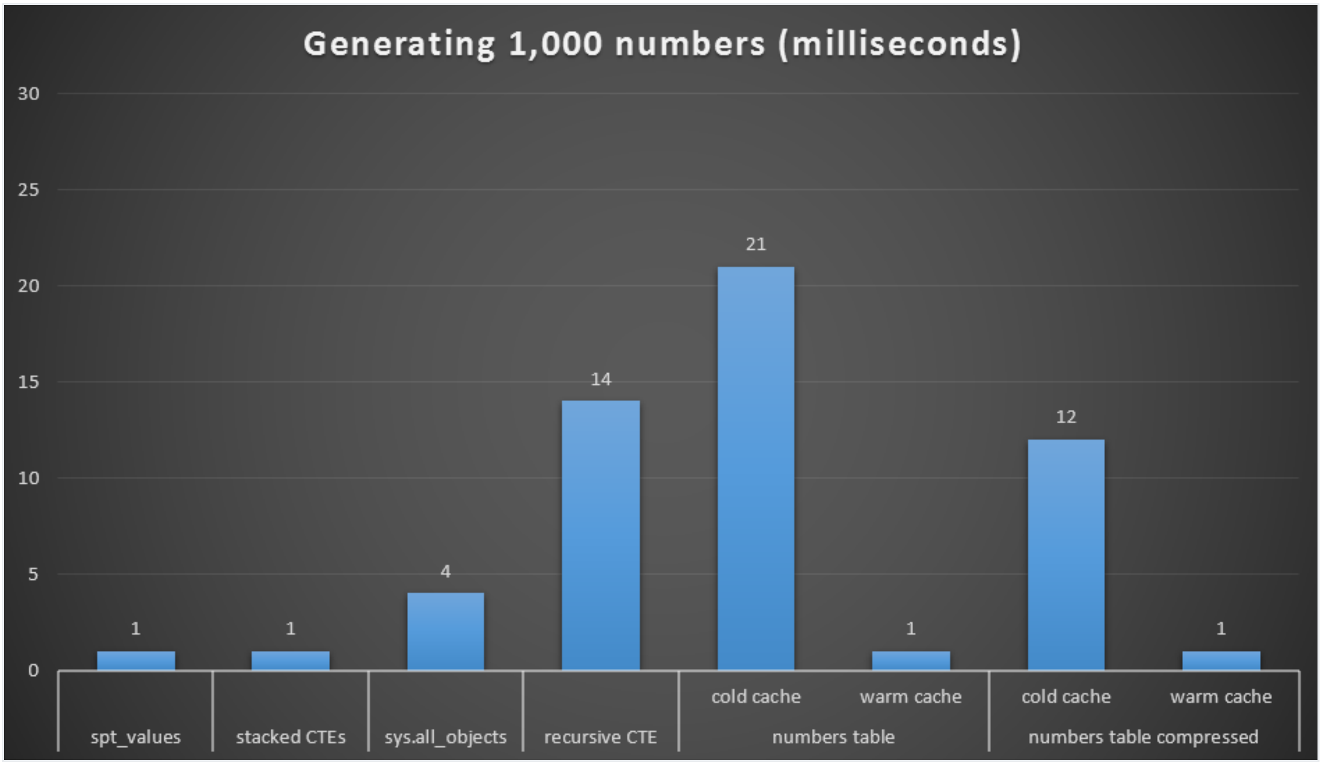
```
;WITH n(n) AS
(
    SELECT 1
    UNION ALL
    SELECT n+1 FROM n WHERE n < 1000
)
SELECT n FROM n ORDER BY n
OPTION (MAXRECURSION 1000);
```

Plan:



Performance

Of course with 1,000 values the differences in performance is negligible, but it can be useful to see how these different options perform:



Runtime, in milliseconds, to generate 1,000 contiguous numbers

I ran each query 20 times and took average runtimes. I also tested the `dbo.Numbers` table, in both compressed and uncompressed formats, and with both a cold cache and a warm cache. With a warm cache it very closely rivals the other fastest options out there (`spt_values` , not recommended, and stacked CTEs), but the first hit is relatively expensive (though I almost laugh calling it that).

To Be Continued...

If this is your typical use case, and you won't venture far beyond 1,000 rows, then I hope I have shown the fastest ways to generate those numbers. If your use case is a larger number, or if you are looking for solutions to generate sequences of dates, stay tuned. Later in this series, I will explore generating sequences of 50,000 and 1,000,000 numbers, and of date ranges ranging from a week to a year.

[Part 1 | Part 2 | Part 3]