Generate a set or sequence without loops – part 2

sqlperformance.com/2013/01/t-sql-queries/generate-a-set-2

Aaron Bertrand January 17, 2013

In my previous post, I talked about ways to generate a sequence of contiguous numbers from 1 to 1,000. Now I'd like to talk about the next levels of scale: generating sets of 50,000 and 1,000,000 numbers.

Generating a set of 50,000 numbers

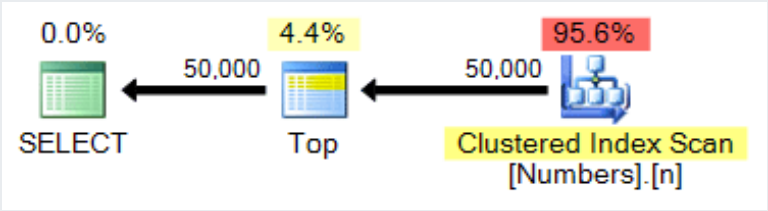
When starting this series, I was genuinely curious how the different approaches would scale to larger sets of numbers. At the low end I was a little dismayed to find that my favorite approach – using `sys.all_objects` – was not the most efficient method. But how would these different techniques scale to 50,000 rows?

Numbers table

Since we have already created a Numbers table with 1,000,000 rows, this query remains virtually identical:

```
SELECT TOP (50000) n FROM dbo.Numbers ORDER BY n;
```

Plan:



spt_values

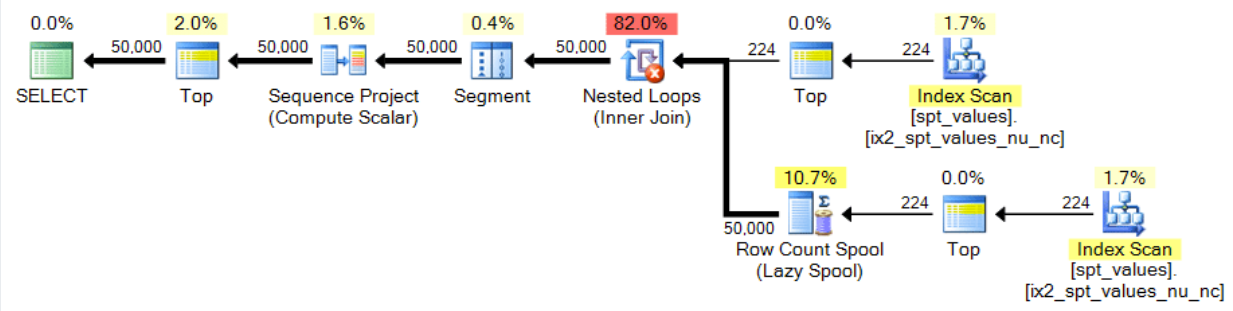
Since there are only ~2,500 rows in `spt_values`, we need to be a little more creative if we want to use it as the source of our set generator. One way to simulate a larger table is to `CROSS JOIN` it against itself. If we did that raw we'd end up with ~2,500 rows squared (over 6 million). Needing only 50,000 rows, we need about 224 rows squared. So we can do this:

```
;WITH x AS
(
  SELECT TOP (224) number FROM [master]..spt_values
)
SELECT TOP (50000) n = ROW_NUMBER() OVER (ORDER BY x.number)
FROM x CROSS JOIN x AS y
ORDER BY n;
```

Note that this is equivalent to, but more concise than, this variation:

```
SELECT TOP (50000) n = ROW_NUMBER() OVER (ORDER BY x.number)
FROM
(SELECT TOP (224) number FROM [master]..spt_values) AS x
CROSS JOIN
(SELECT TOP (224) number FROM [master]..spt_values) AS y
ORDER BY n;
```

In both cases, the plan looks like this:

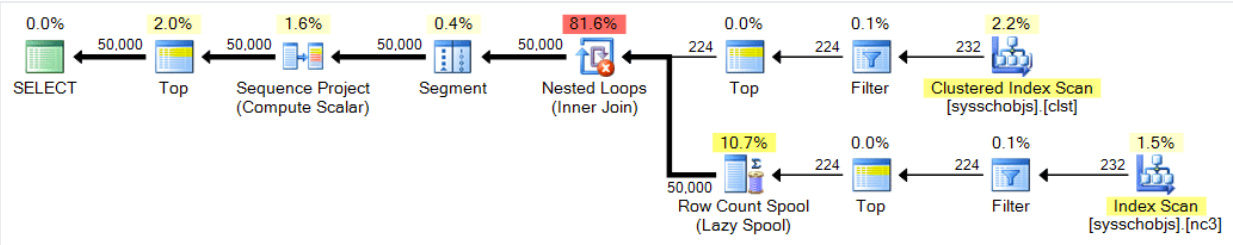


sys.all_objects

Like `spt_values`, `sys.all_objects` does not quite satisfy our 50,000 row requirement on its own, so we will need to perform a similar `CROSS JOIN`.

```
;;WITH x AS
(
  SELECT TOP (224) [object_id] FROM sys.all_objects
)
SELECT TOP (50000) n = ROW_NUMBER() OVER (ORDER BY x.[object_id])
FROM x CROSS JOIN x AS y
ORDER BY n;
```

Plan:

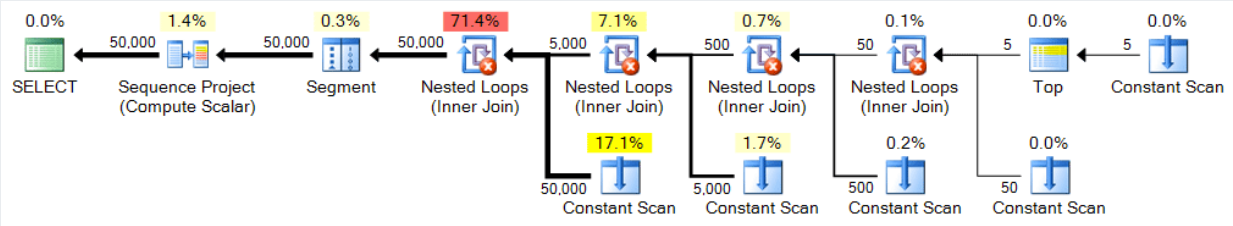


Stacked CTEs

We only need to make a minor adjustment to our stacked CTEs in order to get exactly 50,000 rows:

```
;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 e2 CROSS JOIN e2 AS b), -- 100*100
e4(n) AS (SELECT 1 FROM e3 CROSS JOIN (SELECT TOP 5 n FROM e1) AS b)  -- 5*10000
  SELECT n = ROW_NUMBER() OVER (ORDER BY n) FROM e4 ORDER BY n;
```

Plan:

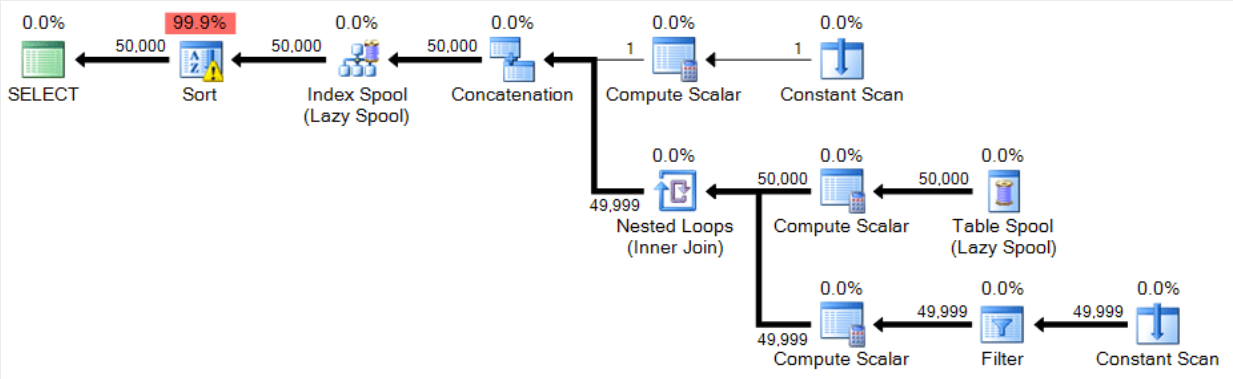


Recursive CTEs

An even less substantial change is required to get 50,000 rows out of our recursive CTE: change the `WHERE` clause to 50,000 and change the `MAXRECURSION` option to zero.

```
;WITH n(n) AS
(
    SELECT 1
    UNION ALL
    SELECT n+1 FROM n WHERE n < 50000
)
SELECT n FROM n ORDER BY n
OPTION (MAXRECURSION 0);
```

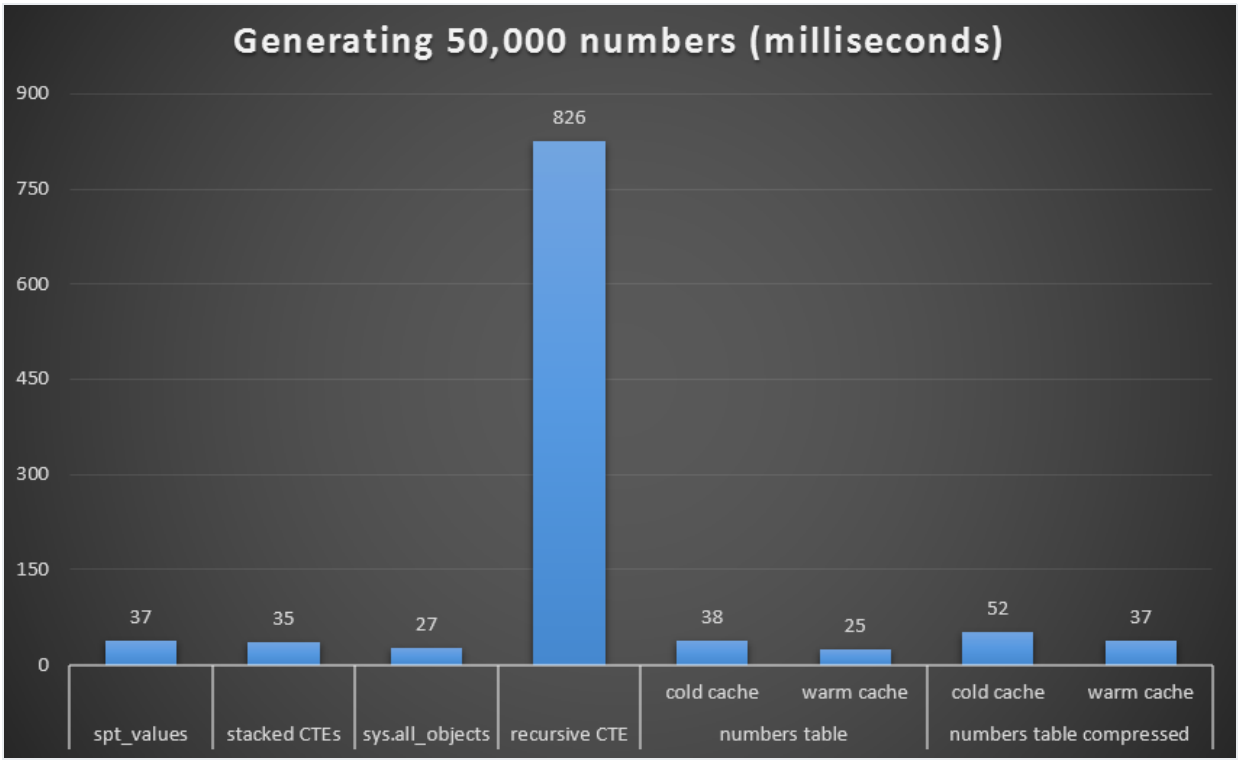
Plan:



In this case there is a warning icon on the sort – as it turns out, on my system, the sort needed to spill to tempdb. You may not see a spill on your system, but this should be a warning about the resources required for this technique.

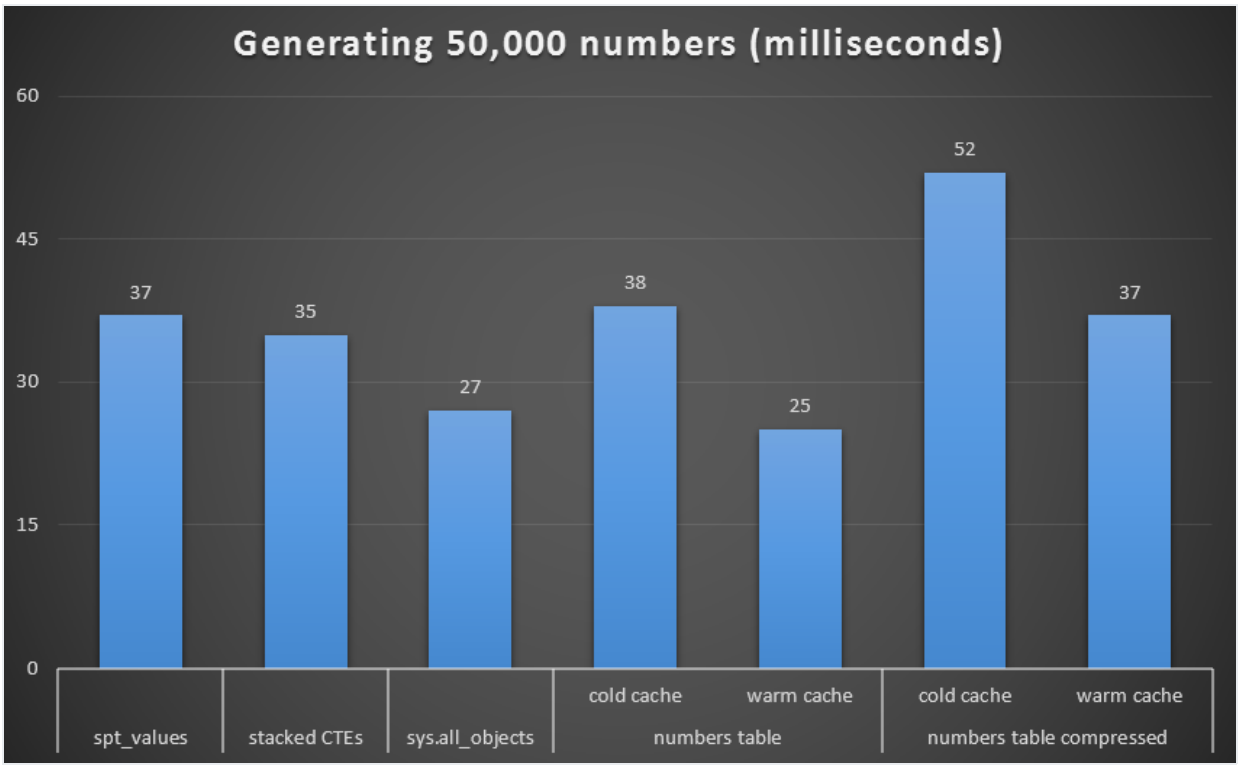
Performance

As with the last set of tests, we'll compare each technique, including the Numbers table with both a cold and warm cache, and both compressed and uncompressed:



Runtime, in milliseconds, to generate 50,000 contiguous numbers

To get a better visual, let's remove the recursive CTE, which was a total dog in this test and which skews the results:



Runtime, in milliseconds, to generate 50,000 contiguous numbers (excluding recursive CTE)

At 1,000 rows, the difference between compressed and uncompressed was marginal, since the query only needed to read 8 and 9 pages respectively. At 50,000 rows, the gap widens a bit: 74 pages vs. 113. However, the overall cost of decompressing the data seems to outweigh the savings in I/O. So, at 50,000 rows, an uncompressed numbers table seems to be the most efficient method of deriving a contiguous set – though, admittedly, the advantage is marginal.

Generating a set of 1,000,000 numbers

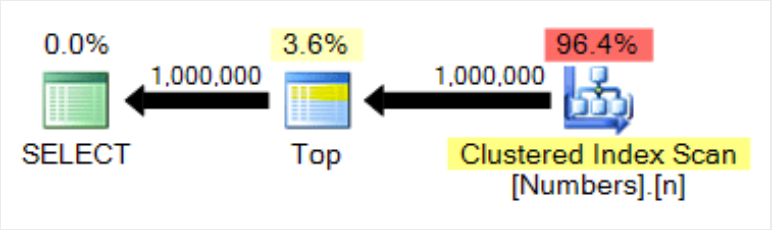
While I can't envision very many use cases where you'd need a contiguous set of numbers this large, I wanted to include it for completeness, and because I did make some interesting observations at this scale.

Numbers table

No surprises here, our query is now:

```
SELECT TOP 1000000 n FROM dbo.Numbers ORDER BY n;
```

The `TOP` isn't strictly necessary, but that's only because we know that our Numbers table and our desired output have the same number of rows. The plan is still quite similar to previous tests:

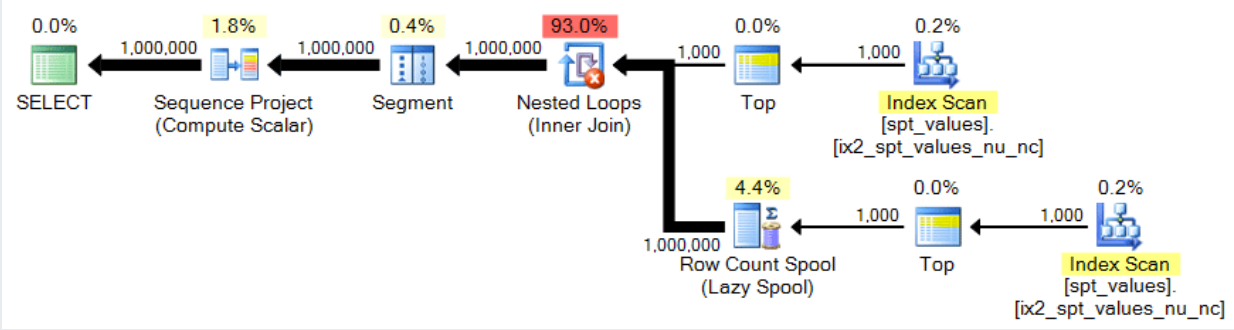


spt_values

To get a `CROSS JOIN` that yields 1,000,000 rows, we need to take 1,000 rows squared:

```
;WITH x AS
(
  SELECT TOP (1000) number FROM [master]..spt_values
)
SELECT n = ROW_NUMBER() OVER (ORDER BY x.number)
FROM x CROSS JOIN x AS y ORDER BY n;
```

Plan:

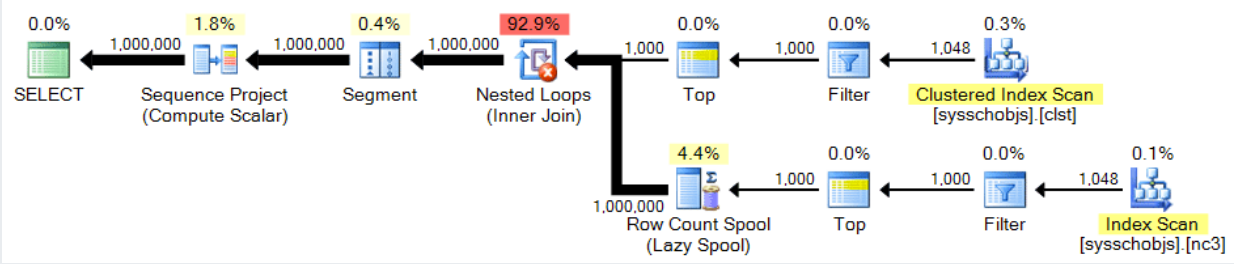


sys.all_objects

Again, we need the cross product of 1,000 rows:

```
;WITH x AS
(
  SELECT TOP (1000) [object_id] FROM sys.all_objects
)
SELECT n = ROW_NUMBER() OVER (ORDER BY x.[object_id])
FROM x CROSS JOIN x AS y ORDER BY n;
```

Plan:

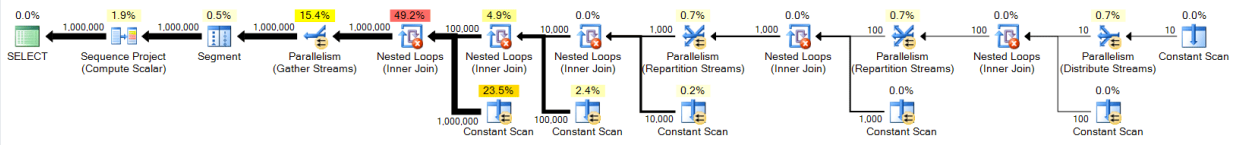


Stacked CTEs

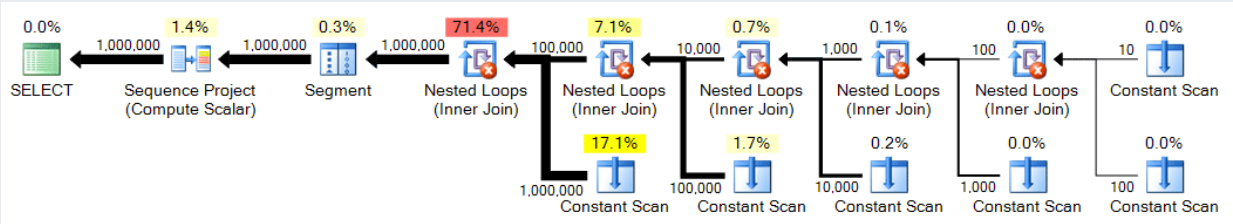
For the stacked CTE, we just need a slightly different combination of `CROSS JOIN`s to get to 1,000,000 rows:

```
;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 AS b), -- 10*100
e4(n) AS (SELECT 1 FROM e3 CROSS JOIN e3 AS b)  -- 1000*1000
 SELECT n = ROW_NUMBER() OVER (ORDER BY n) FROM e4 ORDER BY n;
```

Plan:



At this row size, you can see that the stacked CTE solution goes parallel. So I also ran a version with `MAXDOP 1` to get a similar plan shape as before, and to see if parallelism really helps:

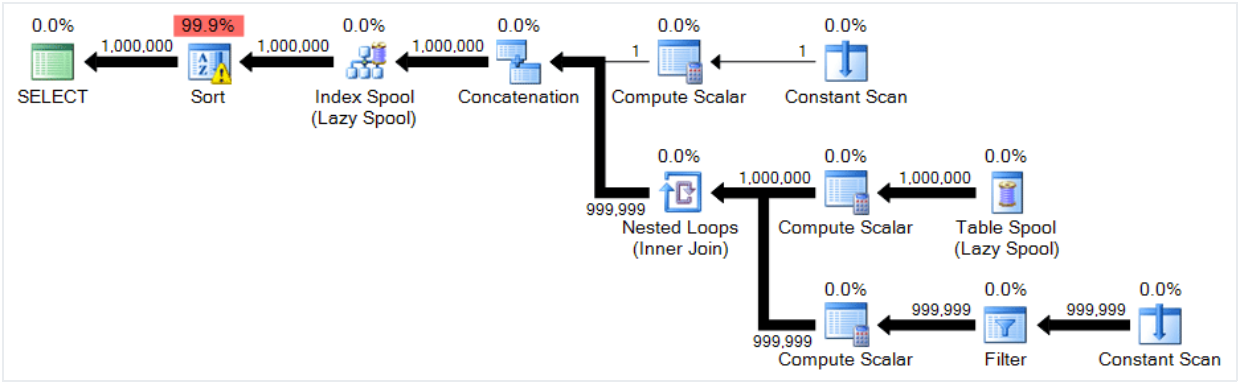


Recursive CTE

The recursive CTE again has just a minor change; only the `WHERE` clause needs to change:

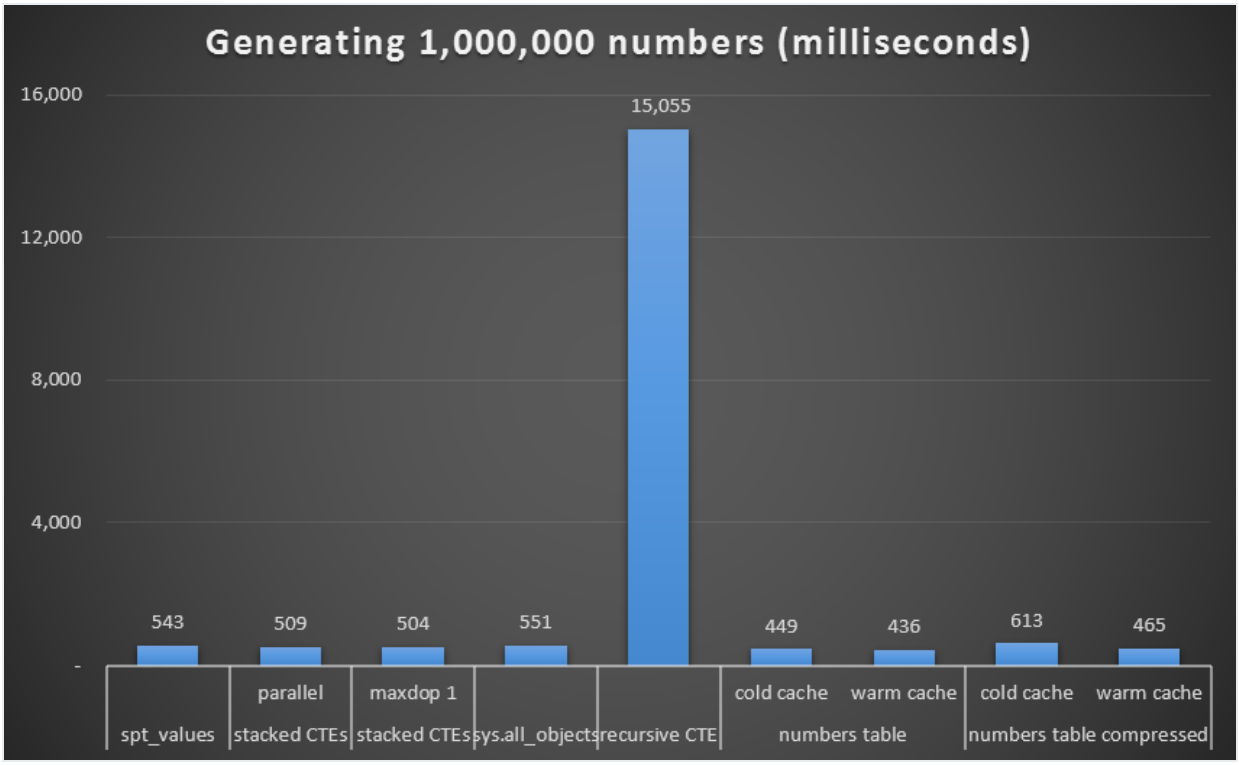
```
;WITH n(n) AS
(
   SELECT 1
   UNION ALL
   SELECT n+1 FROM n WHERE n < 1000000
)
SELECT n FROM n ORDER BY n
OPTION (MAXRECURSION 0);
```

Plan:



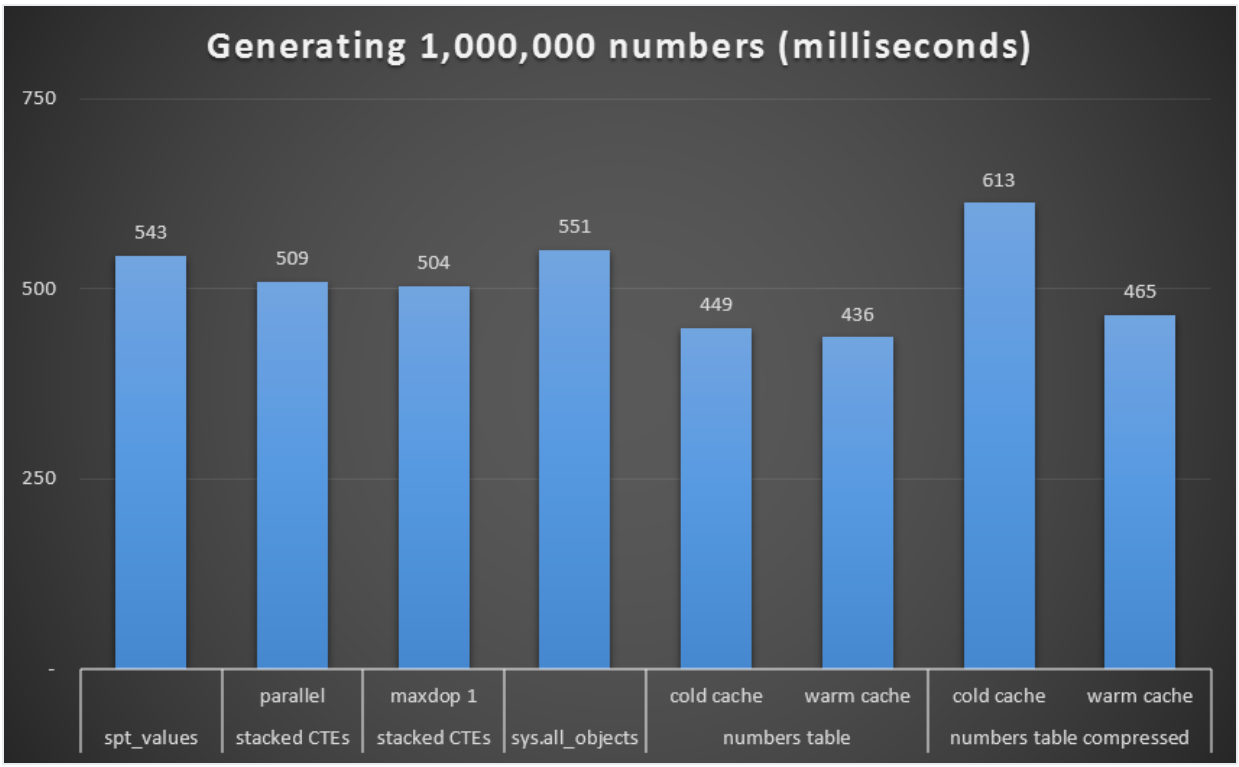
Performance

Once again we see the performance of the recursive CTE is abysmal:



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

Removing that outlier from the graph, we get a better picture about performance:



Runtime, in milliseconds, to generate 1,000,000 contiguous numbers (excluding recursive CTE)

While again we see the uncompressed Numbers table (at least with a warm cache) as the winner, the difference even at this scale is not all that remarkable.

To be continued...

Now that we've thoroughly explored a handful of approaches to generating a sequence of numbers, we'll move on to dates. In the final post of this series, we'll walk through the construction of a date range as a set, including the use of a calendar table, and a few use cases where this can be handy.

[Part 1 | Part 2 | Part 3]

Appendix : Row counts

You may not be trying to generate an exact number of rows; you may instead just want a straightforward way to generate a lot of rows. The following is a list of combinations of catalog views that will get you various row counts if you simply `SELECT` without a `WHERE` clause. Note that these numbers will depend on whether you are at an RTM or a service pack (since some system objects do get added or modified), and also whether you have an empty database.

Source	Row counts		
	SQL Server 2008 R2	**SQL Server 2012**	**SQL Server 2014**
master..spt_values	2,508	2,515	2,519
master..spt_values CROSS JOIN master..spt_values	6,290,064	6,325,225	6,345,361
sys.all_objects	1,990	2,089	2,165
sys.all_columns	5,157	7,276	8,560
sys.all_objects CROSS JOIN sys.all_objects	3,960,100	4,363,921	4,687,225
sys.all_objects CROSS JOIN sys.all_columns	10,262,430	15,199,564	18,532,400
sys.all_columns CROSS JOIN sys.all_columns	26,594,649	52,940,176	73,273,600

Table 1: Row counts for various catalog view queries