Tally OH! An Improved SQL 8K “CSV Splitter” Function

sqlservercentral.com/articles/tally-oh-an-improved-sql-8k-“csv-splitter”-function

Jeff Moden

Update! (5/12/2011)

You've just got to love this community!!! In the discussions found in the "Join the Discussion" link above, a couple of good folks came forward with some improvements to the code I built in the following article. "Nadrek" made a change that caused a good 10% improvement. That was followed by a post by "peter-757102" (aka Peter de Heer) and that brought a whopping 15-20% improvement to the code in the article.

I've modified the attachments (See "RESOURCES" near the very bottom of this article) to include the old tests and the new tests on the functions which contain Peter's modifications. The new tests also include the CLR that Paul White was kind enough to write for me for inclusion in this article.

Because of some confusion by some folks, I've also changed the old code in this article to the new code. I did, however, leave the rest of the article the same even though it's no longer quite correct for expediency sake and because it metaphorically shows what I did to solve a terrible performance problem.

WARNING!

There's also been a bit of confusion about the nature of the code. First, the code has been explicitly written to use only a single character delimiter. Change it at your own risk of performance degradation. Second, DO NOT change the data-types of the inputs to the function to VARCHAR(MAX) or NVARCHAR(MAX). Doing so will cause the code to run at least twice as slow because MAX data-types don't like to be joined to.

A word on the CLR... it looks like it does some pretty strange things at the bottom of each performance curve. That's because my tests don't "Prime" it for first usage. It's performance is actually VERY linear.

I've also included some new spreadsheets in the attachments which show the performance tests I did with Nadrek's and Peter's modifications.

Thanks to all who made for such a great discussion and also made it possible to squeeze even more performance out of the code.

Prologue

This article uses methods that rely heavily on a wonderful little tool known as a "Numbers" or "Tally" Table. More specifically, this article relies on its modern day, close-cousin, the cteTally. Both require more than just a casual knowledge if this article is to be easily understood. If you don't know what a Tally Table is or how it can be used to replace certain While Loops like those found in this article, I strongly recommend that you stop reading here and take the time to study the article at the following link: **http://www.sqlservercentral.com/articles/T-SQL/62867/**

If you've never used a Tally Table before, understanding how a Tally Table (or cteTally) works will greatly simplify your professional life as a DBA, an SQL Developer, or a front-end programmer that also has to write code in T-SQL.

Shifting gears, there are 3 "obviously named" attachments to this article. One is a fully documented stand-alone test script, one is a 5 sheet Excel spreadsheet with a performance chart on each sheet, and one is a zipped file which contains copies of the new VARCHAR(8000) and NVARCHAR(4000) splitters that I think you'll like.

VARCHAR(MAX) and NVARCHAR(MAX) splitters will be covered in a separate article.

Last and certainly not least, a special "Thanks" goes out to Ron McCullough, Wayne Sheffield, and Paul White for their invaluable help on this article and in testing the code associated with this article.

Introduction

I know, I know... The last thing we need is yet another article to add to the thousands already on the Internet on how to split things which probably shouldn't be in a database anyway. I think you're going to like this one, though.

Most folks know by now, that a "Tally Table" (or cteTally, which is a "table-less" version of the Tally Table) based CSV Splitter absolutely screams performance-wise beating all other "T-SQL Only" split methods quite handily. Here's a 1,000 row performance-curve chart of a cteTally splitter pitted against several other splitters. Each line, of course, represents a different type of splitter as indicated in the chart legend.

Figure 1: The Tally Table Wins for Short Splits

As you can see, except for that skinny little Black line, the cteTally based splitter (the fat Red line), soundly thrashes the other commonly known delimited string splitters.

Most folks also know by now (it's a very common complaint, actually), that Tally Table and cteTally (which I'll include in the term "Tally Table" from here on because it's easier to say) splitters are only good for relatively short strings. The chart above shows performance curves only for a very limited 1 to 64 randomly sized elements (string values located between delimiters in a string) of 10 to 20 characters each. That range of the number of elements is actually pretty small compared to some requirements for

splitting. Notice that the performance curve for the Tally Table is starting to look a little strange. It looks like it might be starting to curve up (slow down) a bit. What happens if we try to split strings with more than just 64 elements? Let's have a look at an expanded performance chart and see:

Figure 2: The Tally Table is a Sore Loser on Longer Strings

Yowch! It's true! The Tally Table splitter has a HUGE performance problem when more elements are involved and the overall length of the string has increased. Again, that big, fat Red line is the performance curve of the current "state-of-the-art" Tally Table (as an inline CTE) splitter and it's not nearly as linear as the others clearly are. The Tally Table splitter starts out by blowing the proverbial doors off all the standard collection of "T-SQL Only" splitters but then something strange begins to happen as the number of elements increase, which also means that the overall length of the string is increasing.

As you can see in the chart above, the Tally Table splitter begins to falter at about 125 random length elements of 10 to 20 characters each (about 2,000 characters including the delimiters). At 210 elements (about 3,360 characters including the delimiter), the Recursive CTE (rCTE), XML, and the Tally Table splitters are neck-and-neck. At 290 elements (about 4,640 characters including the delimiters), the Tally Table ties with two different types of While Loop splitters. Finally, stretching out to 480 elements (about 7,680 characters including the delimiters), the once proud Tally Table splitter is a sore loser even to the (gasp!) WHILE loop methods.

The problem is severely exacerbated when the element width increases to as little as 20 to 30 characters per element, as witnessed in the following performance chart.

Figure 3: Tally Table Performance Gets Even Worse for Wider Elements

What on this good green Earth is going on here and why is there such a performance problem with the Tally Table breed of splitters?

While we're asking questions, what do you suppose that skinny little Black line that's doing so very well in the charts above actually is? Raise your hand if you think it's an SQLCLR. Has the proverbial leader of the "Anti-RBAR Alliance" (thank you Mr. Tassin) finally given up the ghost and gone over to the "Dark Side" of SQLCLRs? Is this the end of high-speed "Pseudo Cursors" and the Tally Table?

BWAA-HAAA!!!! Do the words "Obese Opportunity" mean anything to you?

In this article, we'll first learn that there are really only two basic types of delimited string splitters in T-SQL and how they work. Up next, we'll learn a bit more about why Tally Table and cteTally splitters have such performance problems with longer strings and wider elements. Then, we'll learn how to solve those problems.

How a Splitter Works

As stated in the introduction, there are really only two basic types of delimited string splitters regardless of whether you're using some form of procedural code or set-based code to write the splitter. It's important to understand how they work because, without such an understanding, you might end up like me when I was trying to figure all of this out; Sitting in the corner with my favorite "binky" (a small and very comfortable quilt from childhood), knees withdrawn into a near fetal position facing away from the light, twiddling my hair with one hand and sucking on a beer popsicle with the other whilst uttering whiny little unintelligible mutters about why I couldn't solve the performance problem of Tally Table splitters. And, NO, Steve Jones, you may NOT have a picture of that!

The "Nibbler" Splitter

The "Nibbler" type of delimited string splitter isn't all that common and, as you'll soon see, doesn't play well with the "Pseudo-Cursors" formed when you join a Tally Table to a variable at the character level as such splitters require. Without taking the time to draw a pretty picture as I have with the "Inch Worm" family of splitters in the next section of this article, here's how it works.

1. The starting position is known to be character position 1 and is so assigned.
2. The ending position of the current element is found usually by using CHARINDEX to find the next delimiter (if there is one).
3. If a delimiter was found, the length of the element is simply calculated by subtracting 1 from the value determined in step 2 above.
4. The LEFT or SUBSTRING function extracts the data for the element using 1 as the starting position and the length calculated in Step 3 and inserts it into a table.
5. Then, the code simply removes (STUFF works well here) the entire first element and its trailing delimiter from the string leaving the next element at character position 1 in the now modified string.
6. The code loops back to Step 2 and continues to run until no more delimiters are found.
7. Since the last element has no delimiter, it's normally taken care of by a final bit of code outside the loop. This step also takes care of instances where the string only has 1 element.

Like I said, the "Nibbler" type of splitter doesn't play well in the set-based world of the Tally Table because a modification is made to the original string by each iteration. I mention its operation only so that fellow experimenters aren't quickly frustrated by trying to emulate this method using a Tally Table as the "looping" mechanism.

The "Inch Worm" Splitter

The "Nibbler" splitter notwithstanding, virtually all other splitters are nothing more than some variation of the "Inch Worm" type of splitter. There certainly are variations of this type of splitter but they all boil down to the same thing. Let's take a look at a 3 element string and how this type of splitter is used to separate the elements and put them into a result table.

Figure 4: Analysis of the "Inchworm" Splitter

Referring to Figure 4 above, in one form or another, all "Inchworm" types of splitters take the same basic steps:

1. (Point 1) A starting position is either assigned (@Start in all cases above) or the first delimiter (theoretically at Character Position 0 in the chart above) is found.
2. (Point 2) The ending position is found (@End in all cases above) usually by using CHARINDEX to find the next delimiter (if there is one).
3. Using @Start and @End, the SUBSTRING function extracts the data and inserts it into a table (or result set in the case of Tally Tables). Length is determined simply by @End-@Start. Because the "endpoint" isn't included is such simple subtractions, using a "-1) in the formula is not necessary to calculate the proper length.
4. Since the next @Start is the same as the current @End+1, it's typical to simply assign the current value of @End+1 to @Start in preparation for the next iteration of the loop.
5. A new @End is calculated using CHARINDEX.
6. The code loops back to Step 3 and continues to run until no more delimiters are found.
7. Since the last element has no trailing delimiter, it's normally taken care of by a final bit of code outside the loop. This step also takes care of instances where the string only has 1 element.

If you follow the code through, the "Inchworm" starts at Point 1 and then stretches out to Point 2 to return the first element. Then, for a moment, the "tail" of the "worm" is brought up to meet the "head" of the "worm" at Point 2 and then the "head" stretches out again to Point 3 to return the second element. The process continues in this Inchworm-like fashion until all of the elements but the final one has been returned. The final element is the only one that has no ending delimiter so it's normally handled by a separate bit of code outside the loop.

Obviously, the "Inchworm" While loop splitter I used in testing follows the above steps. What you may not realize is that the Recursive CTE (rCTE) method and the XML method (also included in the attached code) are nothing more than minor variations of the same theme. The rCTE uses a special formula (lordy, how I wish that I'd looked at that formula more closely earlier in the game) to handle the final element. The XML method, which is also an "Inchworm" type of splitter, first replaces existing delimiters and adds two "wrapper" delimiters around the original string with short "XML Tag"

markers (to form Points 1 through 4 in the inchworm chart above) . Once the proper delimiters are in place, the XML splitter uses FOR XML to do the split based on those tag markers.

Since the rCTE and the XML methods both use high speed formulas, they're both almost equally as fast with the rCTE method edging out the XML method in most cases. This is because the XML uses 1 concatenation to wrap the original string in delimiter tags and the XML method isn't a high performance Inline Table Valued Function. Both of those items turn out to be a matter of high importance with Tally Table based splitters as you'll see below.

The "Linearity" Problem with the Tally Table

The current breed of Tally Table splitters (as identified by the Red lines in Figures 1 through 3 in the Introduction) is very much like the XML splitter in that, in order to do its job, it needs both a leading and trailing delimiter to "wrap" the original string in. Further, it's generally well known that using an iTVF (Inline Table Valued Function) is usually faster than using an mTVF (Multi-line Table Valued Function). In order to be able to create an iTVF, everything in the splitter code must be done in "a single query" much like a view.

That's were the performance problem for conventional Tally Table splitters comes into play. In order to wrap the original string in delimiters, one of two things must be done; either a separate variable to hold the concatenation must be declared and used or the concatenation must be accomplished within the code of the single query of the splitter code. It turns out that both methods have a catastrophic effect on performance as the width of the string or the width of the individual elements increases.

If we use a separate variable to do the concatenation with, the splitter code can no longer be constructed as a high performance iTVF. Instead, it must be constructed as an mTVF which has all of the same performance problems as any Scalar UDF and would only be able to keep up with the XML method. This is part of the reason why the rCTE method edges out the usual XML method for performance not to mention the required explicit conversion from VARCHAR to XML that most (if not all, I haven't found any that deviate from this necessity) XML splitter methods require.

If we do the concatenation within a single SELECT using either a CTE, a derived table (like a CTE but in the FROM clause), or within the main SELECT itself, a different and rather large performance problem rears its ugly head. While one would normally expect the optimizer of SQL Server to treat such concatenation as a singularly calculated "run time constant", it does not. I've still not determined precisely why longer length elements take such a toll on performance as one would think that it *should* take less time if there are fewer delimiters in the string being split, yet the contrary seems to be true. That assumption, on my part, was part of the reason why it took me so long to finally figure out that all concatenation needed to be removed from Tally Table splitters for them to be performant.

Now you can fully appreciate why I was scrunched up in the corner twiddling my hair and sucking on a beer popsicle. Prior to my discovery that concatenation was the problem, "Mr. no-RBAR" had been beaten and I was contemplating publishing a retraction and full apology to anyone who may have ever listened to my Tally Table splitter rants against all forms of RBAR and XML versions of delimited string splitters. It really wasn't something I was looking forward to doing and so decided it was high time to wash the binky before I sat down in front of the computer.

An (OWCH!) Epiphany

I sat down in front of my computer to write the dreaded retraction that I knew I must write. Finally resigning myself to the task at hand, I opened MS Word to begin the chore. While waiting ever so briefly for it to load, I interlaced the fingers of one hand in the other and stretched mightily towards the ceiling to work out the kinks that spending so many hours in the corner of the room had caused. POP!!!! Dang it! Just what I needed. I had re-aggravated an old shoulder injury and it hurt like the dickens. Rubbing my shoulder in a vain attempt to relieve the pain, I chuckled to myself as I thought of the now ages old joke of going to the doctor and, while raising my arm over my head in a most ungainly fashion, exclaiming, *"Doc! It hurts when I do this."* Of course, the punch-line to that joke is the doctor smiling in the most caring fashion that could be mustered up and in a nearly fatherly tone replying, *"So don't do that."*

Hmmmm... *"So don't do that."* The words echoed in my head over and over. I looked over at a small piece of artwork that a friend had drawn for me. It's an artist's rendition of a big ol' fat cat sitting in a Kitty-Litter™ box with huge grin on its face. The words above the all too satisfied feline state *"Before you can think outside the box, you must first realize... **YOU'RE IN A BOX**"!*

THAT was the key! All this time, I'd been trying to save the *current* rendition of the Tally Table splitter. I was stuck in the proverbial box! Next to *"It Depends"*, one of my other favorite two word publishable sayings is *"What If"?* What if I just scrap the whole thing and start over?

I closed MS Word and loaded SSMS up for one final shot at resolving the Tally Table splitter performance problem.

Recreating the Tally Table Splitter

As I previously identified, I decided that the concatenation of delimiters was the cause of the non-linear performance problem for Tally Table splitters. The sole purpose of that concatenation is to make all the elements look similar to all the others. That is, each element will have both a leading and trailing delimiter available to key off of with a SUBSTRING join to the Tally Table at the character level to easily find all of the delimiters. That, of course, greatly simplifies the code. Could it be simplified even

more? Could I get rid of all the concatenations and some of the other math that supports them? Let's see what the original Tally Table splitter code looks like sans the code to build the inline cteTally just to keep focused on the problem:

Figure 5: A Typical Tally Table Based Splitter

```
--===== Do the split
 SELECT ItemNumber = ROW_NUMBER() OVER (ORDER BY t.N),
        Item       = SUBSTRING(@pString, t.N, CHARINDEX(@pDelimiter, @pString + @pDelimiter,
t.N) - t.N)
   FROM cteTally t
  WHERE t.N <= DATALENGTH(@pString)+1 --DATATLENGTH allows for trailing space
delimiters
    AND SUBSTRING(@pDelimiter + @pString, t.N, 1) = @pDelimiter
;
```

If you look back at the narrative for the "Inchworm" splitter earlier in this article, I say that *"Since the last element has no delimiters, it's normally taken care of by a final bit of code outside the loop. This step also takes care of instances where the string only has 1 element"*. Again, what's the difference between the final element and all the rest of the elements in a typical CSV string? All of the elements in a CSV string are all identical in structure in that each element is *followed* by a delimiter EXCEPT for the final (or only) element which has no trailing delimiter.

So , looking back at Figure 5, ***why*** was I concatenating a delimiter to the ***beginning*** of the string in the WHERE clause? Answer: Because my (and almost everyone else's) Tally Table starts at 1 and, to simplify the splitting formulas, I needed that delimiter to find the first character of the first element by using t.N+1. You can't actually see t.N+1 in the code above because of some optimizations that I and a whole lot of other people have done but, logically speaking, t.N+1 is where the first character of the first element is in the presence of a leading delimiter.

Obviously, I needed to get rid of that leading delimiter to get rid of that particular concatenation but that would also have meant a separate formula for all the other elements which would have leading delimiters. I couldn't just assign a value like what is done in the procedural code of a While loop. Hmmmm or could I? You're about to find out why part of the title of this article isn't the "phat-phingering" mistake you may have thought it was.

Tally OH!

Like many other folks, I've always used a "unit" or "one" based Tally Table. In other words, a Tally Table that starts with one. Let's look at a part of the chart I built for the "Inch Worm" technique with a few minor changes.

Figure 6: A Pattern Appears in the "Inchworm" Splitter

If you look at the colors I've used in Figure 6 above, a pattern develops for ALL of the

elements. If we identify the character BEFORE the first character of each element, we identify one of two things; either a character position that contains a delimiter or the "non-existent" character number zero (all of which are marked as "N" with a Light Green shading in Figure 6) or an "oh", as some people call a zero. And, *OH*, what an epiphany that was! The most difficult part was going to be to write a Tally CTE that started with "OH".

At nearly the same time, it dawned on me that several problems had arisen in the past because the number of "N" values required was calculated in the WHERE clause of the splitter SELECT. With that thought in mind, I did some testing and found that if I use a calculated TOP in the cteTally instead of in the WHERE clause of the splitter SELECT, I could shave a couple of very substantial seconds off a 1,000 row run.

Rather than bore you with the many previously failed experiments that took way too long to execute, let me show you what I ended up with for a zero based cteTally.

Figure 7: A "New" cteTally Row Source, Tally OH

```
--====== "Inline" CTE Driven "Tally Table" produces values from 0 up to
    -- 10,000... enough to cover VARCHAR(8000)
  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
              ),                          --10E+1 or 10 rows
       E2(N) AS (SELECT 1 FROM E1 a, E1 b), --10E+2 or 100 rows
       E4(N) AS (SELECT 1 FROM E2 a, E2 b), --10E+4 or 10,000 rows max
 cteTally(N) AS ( --=== This provides the "zero base" and limits the number of rows right up
front
                  -- for both a performance gain and prevention of accidental "overruns"
               SELECT 0 UNION ALL
               SELECT TOP (DATALENGTH(ISNULL(@pString,1))) ROW_NUMBER() OVER (ORDER BY
(SELECT NULL)) FROM E4
              ),
```

As you can see in Figure 7 above, the addition of the value "0" as one of the values of "N" in the correct order turned out to be a easy task simply by using a well placed and very fast UNION ALL. The TOP is applied only to the non-zero values of "N" because the zero position isn't actually a part of the string we're splitting and we actually do need the non-zero values of "N" to be as long as the actual string that is being split. Also, you've seen this before but it's worth pointing out again, DATALENGTH is being used instead of just LEN in case we ever want to use the SPACE character as the delimiter. LEN ignores trailing spaces where DATALENGTH does not.

So... starting an inline cteTally with an "OH" was solved and I moved on to the next discovery.

Finding the Starting Position of Each Element

If we look at the "Inch Worm" chart fragment again, we see that the start of each element is actually at N+1. In terms of our cteTally splitter, that would mean that we don't actually need to use any value of just "N" in our splitter code. We *always* need "N+1". Check out the positions highlighted in Orange in Figure 8 below to see what I mean.

Figure 8: Revisiting the "Inchworm" Pattern

Armed with that bit of knowledge and my previous discovery about character position "0", I added the code to solve only for the correct values of N+1:

Figure 9: Solving for N+1

```
--====== "Inline" CTE Driven "Tally Table" produces values from 0 up to
    -- 10,000... enough to cover VARCHAR(8000)
  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
              ),                          --10E+1 or 10 rows
       E2(N) AS (SELECT 1 FROM E1 a, E1 b), --10E+2 or 100 rows
       E4(N) AS (SELECT 1 FROM E2 a, E2 b), --10E+4 or 10,000 rows max
 cteTally(N) AS ( --=== This provides the "zero base" and limits the number of rows right up
front
                   -- for both a performance gain and prevention of accidental "overruns"
               SELECT 0 UNION ALL
               SELECT TOP (DATALENGTH(ISNULL(@pString,1))) ROW_NUMBER() OVER (ORDER BY
(SELECT NULL)) FROM E4
              ),
cteStart(N1) AS ( --=== This returns N+1 (starting position of each "element" just once for
each delimiter)
               SELECT t.N+1
                 FROM cteTally t
                WHERE (SUBSTRING(@pString,t.N,1) = @pDelimiter OR t.N = 0)
              )
```

Heh... and there I sat, once again, twiddling my hair and sucking my thumb (I'd previously exhausted my supply of beer popsicles). I was staring an "OR in the WHERE clause" straight in the face. Conventional wisdom and personal experience both sat on my shoulder telling me, no, nagging me, that it was a performance mistake.

Dang. There I go again... stuck in the same ol' box. After several minutes of agonizing, I convinced myself that this situation was somehow different. Throwing caution to the wind, I brushed the doubts from my shoulder (one of them bit me in the process) and continued with the code. I swear I heard the proverbial box I'd been trapped in for so long rip but it turned out to be one of my real cats scratching in the pan. I laughed out loud at the symbolism of the moment for I had also just buried a doubt.

Finding the Length of each Element

I hadn't yet figured out how I was going to handle the final element, which has no trailing delimiter. Now was time to run a sanity check on the code to help figure it all out. I declared some variables to hold a CSV string and a delimiter, assigned the appropriate values to each of them. What I ended up with can be seen in Figure 10 below as a first stab at finding the length of each element based on the position of the next delimiter.

Figure 10: First "Stab" at Determining the Length of Each Element

```
DECLARE @pString VARCHAR(8000),
        @pDelimiter CHAR(1)
;
-- Position        0123456789
 SELECT @pString = '1,22,333',
        @pDelimiter = ','
;
--====== "Inline" CTE Driven "Tally Table" produces values from 0 up to
     -- 10,000... enough to cover VARCHAR(8000)
  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
                ),                          --10E+1 or 10 rows
       E2(N) AS (SELECT 1 FROM E1 a, E1 b), --10E+2 or 100 rows
       E4(N) AS (SELECT 1 FROM E2 a, E2 b), --10E+4 or 10,000 rows max
 cteTally(N) AS ( --=== This provides the "zero base" and limits the number of rows right up
front
                    -- for both a performance gain and prevention of accidental "overruns"
                 SELECT 0 UNION ALL
                 SELECT TOP (DATALENGTH(ISNULL(@pString,1))) ROW_NUMBER() OVER (ORDER BY
(SELECT NULL)) FROM E4
                ),
cteStart(N1) AS ( --=== This returns N+1 (starting position of each "element" just once for
each delimiter)
                 SELECT t.N+1
                   FROM cteTally t
                  WHERE (SUBSTRING(@pString,t.N,1) = @pDelimiter OR t.N = 0)
                )
--====== Return all t.N+1 positions and calculate the length of the element for each starting
position
 SELECT ItemNumber    = ROW_NUMBER() OVER(ORDER BY s.N1),
        StartPosition = s.N1,
        ElementLength = CHARINDEX(@pDelimiter, @pString, s.N1) - s.N1
   FROM cteStart s
;
```

First, I marveled at how comparatively simple this new code was compared to all of the calculations necessary in the old Tally Table splitter. Then, I saw the result set and my heart sank.

Figure 11: Results from Figure 10

See the highlighted cell above? It's a negative length equal to the negative value of the last start position for the final element length. What the heck was I going to do with that!?

Conventional wisdom, once again, sprung to my shoulder and started screaming in a most aggravating, sing-songy voice, *"BWAA-HAAA!!!! YOU NEED A CASE STATEMENT! YOU'RE DOOMED! YOU'RE DOOMED! ALL FALL DOWN!"* He was right. Proverbial "Box" or not, I knew a CASE statement would do two things that I didn't want to do. First, it would make for a really ugly bit of code. Second, because of previous experiments in the same area, I knew it would slow the whole process down even if I wrote it so it would "short circuit" to finding delimiters which would occur more often than the end of the string occurred. There HAD to be a better way!

It was time to make a fresh batch of beer popsicles and take the binky out of the dryer. I remember hoping there were no dust-bunnies hopping about in my favorite dimly lit corner because I was sure I was going to be there a while.

OH! To the Rescue Again!

Despite my misgivings, I had nothing to lose by trying again. I won't list the code here but I did finish the code off using a CASE statement to drive the Length operand of a SUBSTRING just to be sure my misgivings were correct. I was right. The CASE statement slowed the otherwise high performance code down to where even the XML splitter edged past it in performance. Not good enough, but what was I going to do now?

Knowing that I was frustrated and not thinking clearly, I changed the "guts" of the code to show me only the *position* of the next delimiter in line from its related starting position, thusly:

Figure 12: Discovering the End Position

```
 SELECT ItemNumber    = ROW_NUMBER() OVER(ORDER BY s.N1),
        StartPosition = s.N1,
        EndPosition   = CHARINDEX(@pDelimiter, @pString, s.N1)
   FROM cteStart s
;
```

That code returned the following result set:

Figure 13: Results from Figure 12

There it was, again. A zero! Since no delimiter was found for the final element in the string, an end position of zero was returned. Using his best "Yoda" impersonation, personal experience leapt to my shoulder and whispered, *"Ahhhh! New best friend, OH, he is. To your other best friend, introduce him."*

I stared at my beer popsicle with disbelieving eyes wondering if I'd made a bad batch. Where did THAT come from?

Ah... then I remembered an old trick I had used on a bit of code where I didn't really want to use a CASE statement but still wanted to be able to make a CASE-like decision based on whether something was zero or not. I needed to replace *only* zero values with another value. That left REPLACE out because that would try to replace the "0" of numbers like "10" with the other value causing either an error or a bad number not to mention the multitude of implicit and explicit conversions that would be added. That brought up another problem. What value would I use for the Length parameter of a SUBSTRING that would mean *"to the end of the remaining string no matter how long it may be"?* Visions of complex and, therefore, slow length calculations jumped into my head. I started sucking on my beer popsicle with a fervor that would make the cartoon character, Maggie Simpson, look like an amateur.

I remembered something special about the Length operand of SUBSTRING that I had used for other things several times before. Here's the quote from Books Online.

Length: *Is a positive integer or bigint expression that specifies how many characters of the value_expression will be returned. If length_expression is negative, an error is generated and the statement is terminated.* ***If the sum of start_expression and length_expression is greater than the number of characters in value_expression, the whole value expression is returned.***

That's actually a bit of a misnomer and may be why a lot of people don't use it. Too many people think it will return the *whole* string. What it actually does is it returns the *remainder* of the string (without adding extra spaces) according to the Starting Position operand of SUBSTRING and THAT's exactly what I needed for the length of the final element! To cover any eventuality for the length of the final element (or maybe only element possibly), I needed to replace "0" with "8000" in a high performance manner and without using a CASE statement.

While I was thinking of it and to level the playing field a bit, I went back to the other functions I was testing and added that same SUBSTRING 8000 length optimization to the final element calculation of each. A couple of them previously had DATALENGTH(@pString) - @Start which works fine, but I was all about getting rid of as many formulas as I could even in the WHILE loop code.

My other Best Friend, NULL

The "other" friend that my personal experience was making reference to is "NULL" and the functions that manipulate it. Most people know that ISNULL will determine if a value in the first operand is NULL and, if it is, replace it with the value from the second operand. If the first operand is not NULL, then it will not be replaced. What we need now is something that will turn a "0" into a NULL without using a CASE statement.

As it turns out, there's a function in T-SQL which is the diametric opposite of ISNULL. That function is NULLIF. If the first operand matches the second operand, then NULLIF will return a NULL. If the first operand does not match the second operand, then the first operand will be returned unscathed.

As a side bar, some will suggest the use of COALESCE instead of ISNULL in an effort to make the code more "portable". Yes, that will work, but it turned out that ISNULL is just a little bit faster than COALESCE and, considering how many times the code snippet would be executed in a batch job of thousands of rows, I decided to go with the faster option of using ISNULL . Besides, I don't believe in the myth of truly portable batch code, anyway.

Finally, Putting the Length Together

It was time to put all of that together to calculate the length of every element including the final element. First, I needed the formula that finds the next delimiter (end position). That' was easy because I'd already done that.

Figure 14: Find the "Next" Delimiter

```
CHARINDEX(@pDelimiter, @pString, s.N1)
```

Now, the formula in Figure 14 will always return a non-zero position for all elements of a CSV string *except* for the final element. As we previously discovered, because there are no delimiters that follow it, the final element will ALWAYS return a zero for the CHARINDEX. Since I could bank on that little bit of knowledge, I needed to convert only zero values to a NULL leaving the original value only if it's not equal to zero using NULLIF.

As a bit of a sidebar, don't forget that s.N1 in the previous code is actually N+1 from the modified "Inch Worm" charts That means that the CHARINDEX won't start looking for the next delimiter until after the current delimiter which is at "N" in our charts.

In order to covert any zero values that the above formula returns to NULL whilst leaving non-zero values unscathed, I wrapped that whole formula in a NULLIF, thusly.

Figure 15: Converting "0" to a NULL

```
NULLIF(CHARINDEX(@pDelimiter, @pString, s.N1), 0)
```

Substituting that formula back into the larger code, I got a result set that looks like the following:

Figure 16: Results of Changes Added by Figure 15

One more change to the formula was required. I needed to change the NULL to 8,000. I simply wrapped the entire previous formula in an ISNULL to easily pull that off.

Figure 17: Converting NULL to 8,000

```
ISNULL(NULLIF(CHARINDEX(@pDelimiter, @pString, s.N1), 0), 8000)
```

THAT returned the following:

Figure 18: Results of Changes Added by Figure 17

Much to my chagrin, it turns out that I'm not the first person to ever do such a thing although I'm probably the only one that's figured it out while eating a beer popsicle with dust bunnies stuck all over my back. If you take a gander at the rCTE function in the attached code, you'll find that the ISNULL/NULLIF combination used in Figure 17 is used there, as well. The only thing I did, which may be faster and is certainly easier, is that I used a constant instead of a formula for the second operand of the ISNULL function.

I had everything I needed to complete the element extraction using SUBSTRING. All I needed to do next was to subtract the Starting Position from the End Position and I'd have the Length which I needed to complete a SUBSTRING function with.

Figure 19: The Final Length Formula

```
ElementLength = ISNULL(NULLIF(CHARINDEX(@pDelimiter, @pString, s.N1), 0), 8000) - s.N1
```

That bit of computational heaven produced the following result set when used in the larger code:

Figure 20: Results of Changes Added by Figure 19

Notice how it all worked. Referring to Figure 20 above, since the End Position calculated by CHARINDEX for rows 1 and 2 were not equal to zero, neither the NULLIF nor the ISNULL came into play. Those came into play only on the final element on row 3. The code made a CASE-like decision without using CASE and the code is much streamlined for the effort.

Putting it all Together in a New Splitter Function

I next put it all together in the larger code with the required SUBSTRING extraction. Since it's all done in "one query", I also took the opportunity to turn it into a high performance iTVF. Except for missing some comments and a header, this is the final function I ended up with and it's a direct replacement for the old DelimitedSplit8K function:

NOTICE!!!! THIS IS NOT THE ORIGINAL CODE FROM THE ORIGINAL ARTICLE!!! THIS IS THE UPDATED CODE WHICH INCLUDES THE 15-20% SPEED ENHANCEMENT IDENTIFIED IN

THE PROLOGUE OF THIS ARTICLE. THE 4K NVARCHAR VERSION CAN BE FOUND IN THE "RESOURCES" LINK (The New Splitter Functions.zip) AT THE END OF THIS ARTICLE.

Figure 21: The Final "New" Splitter Code, Ready for Testing

```
CREATE FUNCTION [dbo].[DelimitedSplit8K]
--====== Define I/O parameters
        (@pString VARCHAR(8000), @pDelimiter CHAR(1))
--WARNING!!! DO NOT USE MAX DATA-TYPES HERE!  IT WILL KILL PERFORMANCE!
RETURNS TABLE WITH SCHEMABINDING AS
 RETURN
--====== "Inline" CTE Driven "Tally Table" produces values from 1 up to 10,000...
     -- enough to cover VARCHAR(8000)
  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
                ),                          --10E+1 or 10 rows
       E2(N) AS (SELECT 1 FROM E1 a, E1 b), --10E+2 or 100 rows
       E4(N) AS (SELECT 1 FROM E2 a, E2 b), --10E+4 or 10,000 rows max
 cteTally(N) AS (--===== This provides the "base" CTE and limits the number of rows right up
front
                    -- for both a performance gain and prevention of accidental "overruns"
                 SELECT TOP (ISNULL(DATALENGTH(@pString),0)) ROW_NUMBER() OVER (ORDER BY
(SELECT NULL)) FROM E4
                ),
cteStart(N1) AS (--===== This returns N+1 (starting position of each "element" just once for
each delimiter)
                 SELECT 1 UNION ALL
                 SELECT t.N+1 FROM cteTally t WHERE SUBSTRING(@pString,t.N,1) = @pDelimiter
                ),
cteLen(N1,L1) AS(--===== Return start and length (for use in substring)
                 SELECT s.N1,
                        ISNULL(NULLIF(CHARINDEX(@pDelimiter,@pString,s.N1),0)-s.N1,8000)
                   FROM cteStart s
                )
--====== Do the actual split. The ISNULL/NULLIF combo handles the length for the final
element when no delimiter is found.
 SELECT ItemNumber = ROW_NUMBER() OVER(ORDER BY l.N1),
        Item       = SUBSTRING(@pString, l.N1, l.L1)
   FROM cteLen l
;
```

Functional Testing

I was elated. I'd managed to avoid all forms of concatenation in the new splitter, as well as severely limiting the number of "+1" and "-1" calculations required. Even the dust bunnies that were stuck to my back cheered.

Of course, I'd not accomplished a thing until I actually tested the function for correct operation. I used the following code to test all aspects of the splitter using a comma as a delimiter including when there were adjacent delimiters which represent empty strings.

Figure 22: Testing Functionality

```
--
============================================================

-- TEST 1:
-- This tests for various possible conditions in a string using a comma as the delimiter.  The
expected results are
-- laid out in the comments
--
============================================================

--===== Conditionally drop the test tables to make reruns easier for testing.
     -- (this is NOT a part of the solution)
     IF OBJECT_ID('tempdb..#JBMTest') IS NOT NULL DROP TABLE #JBMTest
;
--===== Create and populate a test table on the fly (this is NOT a part of the solution).
     -- In the following comments, "b" is a blank and "E" is an element in the left to right order.
     -- Double Quotes are used to encapsulate the output of "Item" so that you can see that all
blanks
     -- are preserved no matter where they may appear.
 SELECT *
   INTO #JBMTest
   FROM (                                    --# & type of Return Row(s)
         SELECT  0, NULL                   UNION ALL --1 NULL
         SELECT  1, SPACE(0)                 UNION ALL --1 b (Empty String)
         SELECT  2, SPACE(1)                 UNION ALL --1 b (1 space)
         SELECT  3, SPACE(5)                 UNION ALL --1 b (5 spaces)
         SELECT  4, ','                    UNION ALL --2 b b (both are empty strings)
         SELECT  5, '55555'                UNION ALL --1 E
         SELECT  6, ',55555'               UNION ALL --2 b E
         SELECT  7, ',55555,'              UNION ALL --3 b E b
         SELECT  8, '55555,'               UNION ALL --2 b B
         SELECT  9, '55555,1'               UNION ALL --2 E E
         SELECT 10, '1,55555'                UNION ALL --2 E E
         SELECT 11, '55555,4444,333,22,1'     UNION ALL --5 E E E E E
         SELECT 12, '55555,4444,,333,22,1'    UNION ALL --6 E E b E E E
         SELECT 13, ',55555,4444,,333,22,1,'  UNION ALL --8 b E E b E E E b
         SELECT 14, ',55555,4444,,,333,22,1,' UNION ALL --9 b E E b b E E E b
         SELECT 15, ' 4444,55555 '           UNION ALL --2 E (w/Leading Space) E (w/Trailing Space)
         SELECT 16, 'This,is,a,test.'              --E E E E
        ) d (SomeID, SomeValue)
;
--===== Split the CSV column for the whole table using CROSS APPLY (this is the solution)
 SELECT test.SomeID, test.SomeValue, split.ItemNumber, Item = QUOTENAME(split.Item,'"')
   FROM #JBMTest test
  CROSS APPLY dbo.DelimitedSplit8k(test.SomeValue,',') split
;
```

If you'll run the code from Figure 22 above, you'll see that it produces an output that's a

wee bit too lengthy to include in this article. It does, however return all the correct answers (leading commas indicate a missing leading element as do trailing commas). I've encapsulated the output on each row with double quotes so that you can see that all spaces are preserved (unless they're used as a delimiter, of course).

The test code in Figure 22 above has also been incorporated into the attached code as comments in the header of the new splitter. If you decide to test it using spaces for delimiters, you won't be disappointed. Just remember things like splitting on 2 spaces will return 3 rows because there are two delimiters.

A word on Collation

If you've never used anything other than a comma, pipe, semicolon, or a tab character as a delimiter, you'll never have seen what Collation can do ***to*** you. For example, the collation installed by SQL Server when you accept the defaults is usually a "case-insensitive" collation. That means that if you use a delimiter of "M", it will split on both "M" and "m".. Things get even weirder if you use some of the accented characters.

Most people stick to the "normal" delimiters of commas and tab characters. If you want to see how the new splitter will react to your particular collation for any given delimiter, run "Test 2" from Figure 23 below and check the output.

As with "Test 1", "Test 2 is also located in the attached code.

Figure 23: Code to Determine the Effects of Collation

```
 --
============================================================

-- TEST 2:
-- This tests for various "alpha" splits and COLLATION using all ASCII characters from 0 to 255
as a delimiter against
-- a given string.  Note that not all of the delimiters will be visible and some will show up as
tiny squares because
-- they are "control" characters.  More specifically, this test will show you what happens to
various non-accented
-- letters for your given collation depending on the delimiter you chose.
--
============================================================

WITH
cteBuildAllCharacters (String,Delimiter) AS
(
 SELECT TOP 256
        'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789',
        CHAR(ROW_NUMBER() OVER (ORDER BY (SELECT NULL))-1)
   FROM master.sys.all_columns
)
 SELECT ASCII_Value = ASCII(c.Delimiter), c.Delimiter, split.ItemNumber, Item =
QUOTENAME(split.Item,'"')
   FROM cteBuildAllCharacters c
  CROSS APPLY dbo.DelimitedSplit8k(c.String,c.Delimiter) split
  ORDER BY ASCII_Value, split.ItemNumber
;
```

Performance Testing

Isn't it wonderful how things come full circle? Performance was the original problem and, now that the Tally Table delimited string splitter function had been rewritten, I needed to test it and see if any improvements in performance have been made. Obviously, I've already tested it or this would have been an article of retraction and sincere apology. I've attached some commented code that I used to do the performance testing to create the performance charts at the beginning of this article. Feel free to run it. It all "lives" in TempDB during the run.

That test code also has some pretty neat "Pseudo Cursor" and randomizing tricks in it, as well. It's a lesson in itself.

But, let me possibly save you some time. How good and how performant is the new Tally Table splitter? Well, do you remember that *little Black line*?

What About a CLR Splitter?

As many will tell you, SQL Server just isn't very good at manipulating strings when you compare it to some well written C code. Paul While provided me with a CLR splitter and the performance curves for the CLR and the new DelimitedSplit8K splitter can be seen

in Figure 24 below. I used the same vertical scale of 30 as I did in Figure 3 (compared all but the CLR function for 20 to 30 characters) so that you can see that, although the CLR is still two to three times faster, the new DelimitedSplit8K function gives the CLR splitter a run for it's money.

Figure 24: New DelimitedSplit8K vs. a CLR Splitter.

Still, the CLR will handle both NVARCHAR(MAX) and VARCHAR(MAX) as is where we need two separate functions using the new breed of Tally Table functions just to handle the differences between NVARCHAR and VARCHAR.

Epilogue

Make no doubt about it. The current best way to split delimited strings in SQL Server is to use a CLR splitter. However, if, for whatever reason, you cannot use a CLR splitter, the new DelimitedSplit8K function provides a close second with both linear and stable performance across a wide range of string and individual element size. And, I promise, I didn't include any dust bunnies in the code.

Thanks for listening folks.

--Jeff Moden

"**RBAR** is pronounced *"ree-bar"* and is a "Modenism" for "**R**ow-**B**y-**A**gonizing-**R**ow"