Paper 155-2019

Working with Sparse Matrices in SAS®

Andrew T. Kuligowski, Independent Consultant,
Lisa Mendez, Ph.D., IQVIA Government Solutions

ABSTRACT / INTRODUCTION

For the past couple of years, it seems that “Big Data” has been a buzzword in the industry. We have more and more data coming in from more and more places, and it is our job to figure out how best to handle it. One way to attempt to organize data is with arrays – but what do you do when the array you are attempting to populate is so large that it cannot be handled in memory. In addition, how do you handle a large array when most of the elements are missing?

This paper deals with the concept of a Sparse Matrix – that is, a large array with relatively few actual elements. We will address methods such a construct be handled while keeping memory, CPU, clock, and programmer time to their respective minimums.

Note that the material presented is a conceptual introduction, aimed at the Base SAS® programmer. Readers who are interested in more advanced areas of the topic – for example, the use of SAS/IML™, for “Interactive Matrix Language” – are encouraged to check other papers on the subject. A short list of related papers from other SAS conferences is provided at the end of this paper.

WHAT IS A SPARSE MATRIX?

In order to define “sparse matrix”, it is necessary to take a step backwards and define a few introductory concepts. According to Wikipedia, an **array** is a systematic arrangement of similar objects – although the concepts of “similar” and “systematic” can be rather loosely interpreted in some instances. A **multidimensional array** is an array that cannot be defined in a single column or row. The most common multidimensional array is the 2-dimensional array, arranged in rectagonal form; this is commonly referred to as a **matrix**. The world is full of examples of matrices – anyone who has ever launched MS-Excel™, for example, immediately has a matrix consisting of a series of rows and columns staring back at them from the computer screen.

If someone were to pick up a paper calendar, they would see several (12 if the calendar is for a single year) examples of matrices. In February 2015 and again in February 2026 the month begins on a Sunday – given 28 days, it follows that it ends on a Saturday. This is the rare example of when every day of a calendar month is completely populated. In matrix terms, each day would be considered a **cell** – the intersection of a column and row. In all other months in all other years, there will be a series of empty cells prior to the first day of the month and/or the final day.

Once it is realized that not every cell in a matrix need be populated, we can introduce the concept of a **sparse matrix,** which is when most of the cells are not populated. “Most” is an intentionally vague word, as there is no official point at which a matrix is declared sparse. Many definitions define “not populated” as “equal 0”. Typically, in SAS, 0 (zero) is considered a valid value, while ***missing*** is used to denote “not populated”. For example, looking at survey results, a zero may mean “No” while a missing could represent “declined to answer”. The zero should be used in analysis, while the missing can most likely be ignored.

Matrices are measured for **sparsity** and its’ opposite dimension, **density**, by simple ratios:

```
ARRAY_DENSITY  = Number_of____populated_cells / Total_number_of_cells;
ARRAY_SPARSITY = Number_of_unpopulated_cells / Total_number_of_cells;
where Total_number_of_cells = Number_of_Rows * Number_of_Columns;
```

Using a small example of an 8x6 array:

15	20	0	0	0	0	0	0
0	25	30	0	0	0	0	0
0	0	50	23	12	0	0	0
0	0	0	50	45	0	0	0
0	0	0	0	0	35	65	0
0	0	0	0	0	0	0	41

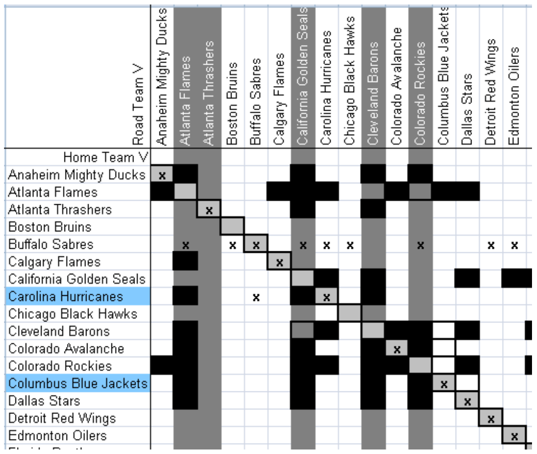
```
ARRAY_DENSITY  = 12 / 48;      25% DENSE
ARRAY_SPARSITY = 36 / 48;      75% SPARSE
```

Let us look at an actual example, tracking the travels of a hockey fanatic. If his goal is to visit each of the home arenas of the NHL, we would need to create an array containing 30 elements – then modify it to allow for the 31st team which was added a few seasons ago. (As of this writing, a 32nd team has been announced but has not yet begun play – this will require another modification.)

Anaheim Ducks	Boston Bruins	Buffalo Sabres	Calgary Flames	Carolina Hurricanes	Chicago Black Hawks	Colorado Avalanche	Columbus Blue Jackets	Dallas Stars	Detroit Red Wings	Edmonton Oilers	Florida Panthers	Los Angeles Kings	Minnesota Wild	Montreal Canadiens	Nashwille Predators	New Jersey Devils	New York Islanders	New York Rangers	Ottawa Senators	Philadelphia Flyers	Phoenix Coyotes	Pittsburgh Penguins	San Jose Sharks	St. Louis Blues	Tampa Bay Lightning	Toronto Maple Leafs	Vancouver Canucks	Vegas Golden Knights	Washington Capitals	Winnipeg Jets

```
ARRAY NHL_TEAMS (31) Team01-Team31;
```

The situation becomes complicated when we decide to track which teams he has seen compete against each other, requiring a two-dimensional array. This added tracking might include teams which have relocated or folded, such as the Atlanta Thrashers: Illustrated below is a subset of this matrix. Blackened cells represent a pair of teams that did not exist during the same time interval and could never have faced each other:



Road Team V / Home Team V	Anaheim Mighty Ducks	Atlanta Flames	Atlanta Thrashers	Boston Bruins	Buffalo Sabres	Calgary Flames	California Golden Seals	Carolina Hurricanes	Chicago Black Hawks	Cleveland Barons	Colorado Avalanche	Colorado Rockies	Columbus Blue Jackets	Dallas Stars	Detroit Red Wings	Edmonton Oilers
Anaheim Mighty Ducks	x	■					■			■		■				
Atlanta Flames	■					■	■	■			■		■	■		
Atlanta Thrashers			x				■			■						
Boston Bruins																
Buffalo Sabres		x		x	x		x	x	x			x			x	x
Calgary Flames		■				x										
California Golden Seals								■		■				■		■
Carolina Hurricanes		■			x		■	x		■						
Chicago Black Hawks																
Cleveland Barons		■						■			■	■		■		
Colorado Avalanche		■					■			■	x	■				
Colorado Rockies	■	■					■	■		■	■			■		
Columbus Blue Jackets		■					■			■		■	x			
Dallas Stars		■					■			■		■		x		
Detroit Red Wings															x	
Edmonton Oilers																x

```
ARRAY NHL_TEAMS (50,50) TeamCombo0001-TeamCombo2500;
```

The array gets much larger if we add in minor league ("farm teams"), college, junior teams, and such. If arenas are then factored into the tracking, noting which teams have changed buildings and/or played in temporary homes – our matrix gets significantly larger. Most of the cells will remain empty, because teams in different leagues are likely never to meet up. This team tracker example has become a sparse matrix. We can represent that in SAS pseudo-code as:

```
ARRAY HOCKEY_TEAMS (lots,lots) TeamCombo0001-TeamCombolotsandlots;
```

Unfortunately, once the matrix grows beyond a certain size, your SAS session will express its frustration in its usual fashion, with a message similar to:

```
ERROR: The SAS System stopped processing this step because of
       insufficient memory.
```

WORKING WITH AND AROUND THE SPARSE MATRIX

The main issue is not that we have a sparse matrix, but that we have a matrix which is too big for our current environment to handle. The fact it happens to be sparse may be coincidental, if we can solve the problem of dealing with the entire matrix!

APPROACH 1: BRUTE FORCE – DEAL WITH MEMORY

The quick-and-dirty (also known as "brute force") solution to this problem would be to allocate more memory to the problem. SAS will certainly allow this using the system option **MEMSIZE**. The default is 2G, but can be expanded – specified in bytes, kilobytes, megabytes, gigabytes, terabytes, or MAX. Note that MEMSIZE=0 is the same as MEMSIZE=MAX.

Increasing MEMSIZE is a viable solution, but it is not without issues and risks. One drawback is that memory is allocated at the beginning of a SAS session. This means that you cannot correct the situation without launching a new SAS session and repeating any steps required for this resolution. You will be cautioned if you attempt to override this option once you've initialized your session:

```
WARNING 30-12: SAS option MEMSIZE is valid only at startup
               of the SAS System. The SAS option is ignored.
```

The main problem with increasing MEMSIZE is that memory is a finite resource. Increasing the amount of memory allocated to SAS on your personal computer will reduce the amount of memory available to other applications. This may not be a major issue, as it may be possible – and necessary – to suspend all but the most critical tasks on your computer – and your personal schedule – in order to complete a requested run. However, when using a shared system, allocating additional memory for your own use will limit its availability to other applications and users on your system. This may not sit well with your fellow computer users and the team tasked with providing support to said system.

There are several SAS system options that control how memory is allocated. You can display them, with their current values, by using the **GROUP=MEMORY** option on the **OPTIONS** procedure:

```
65   PROC OPTIONS GROUP=MEMORY; RUN;

Group=MEMORY

SORTSIZE=1073741824   Specifies the amount of memory that is available to
                      the SORT procedure.
SUMSIZE=0             Specifies a limit on the amount of memory that is
                      available for data summarization procedures when class
                      variables are active.
MAXMEMQUERY=268435456 For certain procedures, specifies the maximum amount
```

```
                        of memory that can be allocated per request.
LOADMEMSIZE=0           Specifies a suggested amount of memory that is needed
                        for executable programs loaded by SAS.
MEMSIZE=42949672960     Specifies the limit on the amount of virtual memory
                        that can be used during a SAS session.
REALMEMSIZE=0           Specifies the amount of real memory SAS can expect
                        to allocate
```

APPROACH 2: SHRINK THE SIZE OF THE DATA(?)

It shouldn't be any surprise that True/False data can be represented as 1's and 0's. In the "Traveling hockey fan" example, the data are not being used to count occurrences, but simply to attest as to whether they occurred. However, it takes the same eight bytes to represent a 1 or a 0 as it would to represent an actual count.

The **LENGTH** statement will permit the allocation of numeric variables that are smaller than 8 bytes. SAS recommends that this only be considered when dealing with integer values, as a real numeric value's precision could, and will, be affected when fewer than eight bytes are used. Further, when performing mathematical processing, SAS will expand the value internally to eight bytes, which will at least partially defeat the purpose of storing variables in a shortened form.

There is another possibility to consider. A byte is made up of eight bits, each of which is either set to 1 or 0. If SAS will allow us to access and control the value of each individual bit, we can shrink our storage footprint by a factor of eight!

We can do bit comparisons in the SAS Data step. Care must be taken to ensure that the programmer uses either the specific or the generic value for each bit. Looking at an example:

```
IF  CHARVAR_LEN1 = '00001000'b  THEN …
```

In this example, every one of the eight bits are critical to the comparison. The fifth bit (using the left-to-right system that of prose writing, not right-to-left numeric nomenclature) must be TRUE (1), while the other seven must be FALSE (0).:

```
IF  CHARVAR_LEN1 = '0...1...'b  THEN …
```

On the other hand, this example is much less restrictive. The first bit must be FALSE, the fifth bit must be TRUE, and the settings of the other six bits are immaterial to the comparison and are ignored.

Unfortunately, the comparison requires a constant; variables are not permitted. This certainly limits the flexibility of using bit comparisons for our sparse matrix.

```
CHARVAR_LEN1 = '00001000'b;

ERROR 216-185: The use of a BIT string constant is not allowed in this
context.
```

The Data step *will* allow us to convert a bit string into an eight byte character field so that we can examine and work with the values stored in the original bits. However, having to expand our one byte field into an eight byte field defeats the eight-fold savings that we had anticipated.

INPUT will convert a character string into a numeric value, while PUT will do the opposite, accepting a numeric value into a character. The lengths in both cases must be multiples of eight.

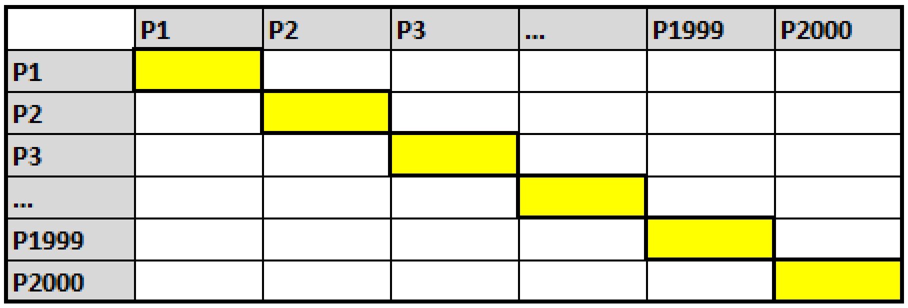
```
INPUT  NUMVAR_LEN1    binary8.;

PUT    CHARVAR_LEN1  $binary8.;
```

APPROACH 3: TRICKERY – MANEUVER AROUND THE EMPTY CELLS

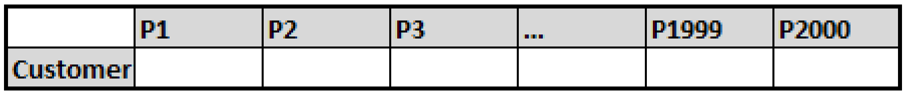
"If you can't go through something, see if you can go around it." This philosophy has been employed by such diverse groups as warriors, explorers, builders, and athletes. It's also applicable for programmers. At this point, since we have not been able to brute force our way through the array definition and processing, we are going to attempt to finesse our way around it.

In order to illustrate our approach, we will use a variation of the hockey attendance array described above. Let us assume that we are tracking 2000 tourist sites, and we'd like to track which combinations of those sites that each of our selected set of visitors have been to. Picture a 2000 by 2000 array, with the list of sites running across and running down. The intersection of those sites will contain a 1 for TRUE if someone has visited both sites. This will create a matrix of four million cells. This may not qualify as "large" in these days of big data, but it will suffice for this example.

The first thing we realize is that anyone who visited Site X ALSO visited Site X! (Remember, the matrix has the sites named in both the rows and columns.) Anyone who has stopped at any of our tracked sites will be counted in the cell where column = row. If this was all we were tracking, it would result in a **diagonal matrix**. It would also be much simpler and memory efficient to store as a single-dimension array – this fact will come into play a little later in our example.

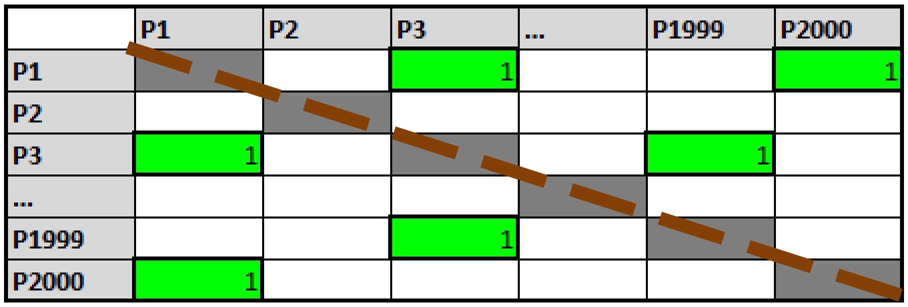


	P1	P2	P3	...	P1999	P2000
P1	🟨					
P2		🟨				
P3			🟨			
...				🟨		
P1999					🟨	
P2000						🟨



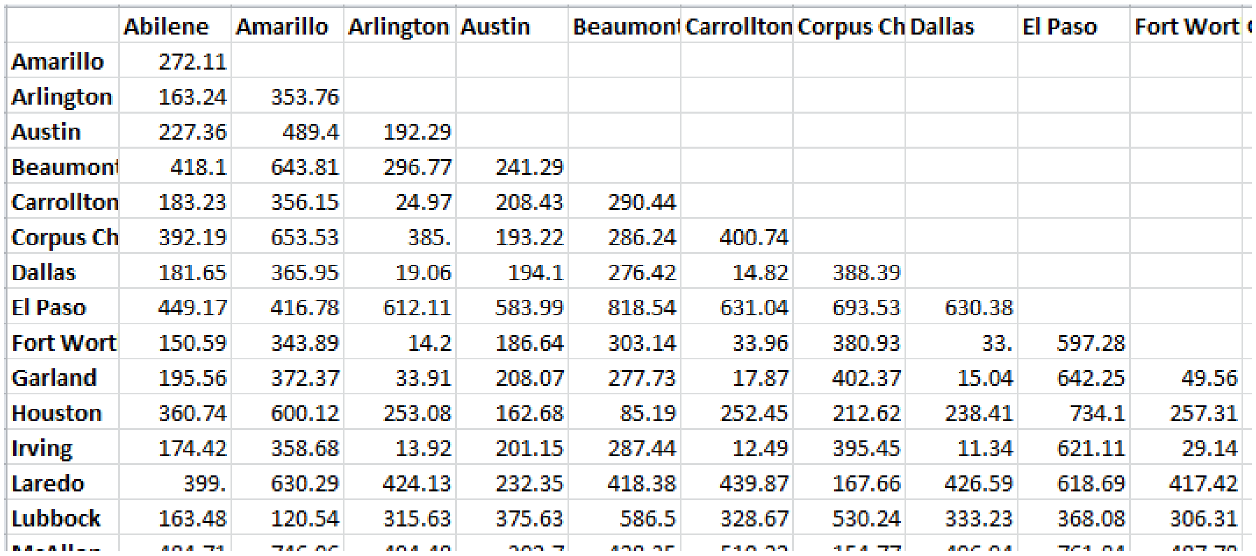
	P1	P2	P3	...	P1999	P2000
Customer						

Once we begin to encounter sample members who have visited multiple tourist sites, we can move away from the diagonal and populate other cells in our matrix. However, the truism mentioned for single site visitors has a parallel for those who see multiple sites: Anyone who visits Site X and Site Y ALSO visits Site Y and Site X. This will result in a **symmetrical matrix.**



	P1	P2	P3	...	P1999	P2000
P1	▨		1			1
P2		▨				
P3	1		▨		1	
...				▨		
P1999			1		▨	
P2000	1					▨

Road Atlases used to commonly contain a form of symmetrical matrix. They displayed the distances between two cities. Since the distance between City A and City B was the same as the distance between City B and City A, they simply didn't bother to print both mirror images, and left half of the matrix blank. (Of course, the distance between City A and City A was 0, so this was suppressed, as well.)



	Abilene	Amarillo	Arlington	Austin	Beaumont	Carrollton	Corpus Ch	Dallas	El Paso	Fort Wort
Amarillo	272.11									
Arlington	163.24	353.76								
Austin	227.36	489.4	192.29							
Beaumont	418.1	643.81	296.77	241.29						
Carrollton	183.23	356.15	24.97	208.43	290.44					
Corpus Ch	392.19	653.53	385.	193.22	286.24	400.74				
Dallas	181.65	365.95	19.06	194.1	276.42	14.82	388.39			
El Paso	449.17	416.78	612.11	583.99	818.54	631.04	693.53	630.38		
Fort Wort	150.59	343.89	14.2	186.64	303.14	33.96	380.93	33.	597.28	
Garland	195.56	372.37	33.91	208.07	277.73	17.87	402.37	15.04	642.25	49.56
Houston	360.74	600.12	253.08	162.68	85.19	252.45	212.62	238.41	734.1	257.31
Irving	174.42	358.68	13.92	201.15	287.44	12.49	395.45	11.34	621.11	29.14
Laredo	399.	630.29	424.13	232.35	418.38	439.87	167.66	426.59	618.69	417.42
Lubbock	163.48	120.54	315.63	375.63	586.5	328.67	530.24	333.23	368.08	306.31
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]

Again, this fact can also be used to shrink the amount of data that we must track, process, and store. If we realize that half of our data is redundant, we should be able to find a way to shrink the amount of this data by 50%.

The first thing we will do is identify and store our diagonal matrix. As stated earlier, we don't need to store this as a two-dimensional matrix, but rather as a one dimensional array. Further, we do not *necessarily* need to even track those customers who visited multiple sites, as we can get that information when dealing with their pairs of two-site occurrences. (If someone visited Site X and Site Y, they obviously visited Site X. They obviously visited Site Y.) We may decide to do it at this point for various processing reasons – or perhaps to make the code easier for subsequent support – but let us assume we are only tracking single site visitors – that is, when sorted by customer ID, the first record for a customer is also the last record for a customer.

```
ARRAY DIAGONAL (2000) DIAGONAL0001-DIAGONAL2000;
…
IF  LAST.CUSTOMER_ID THEN DO;
   IF  FIRST.CUSTOMER_ID  THEN DO;
      DIAGONAL( SITE_ID ) = 1;
      OUTPUT  DIAGONAL_MATRIX;  /* One cell */
   END;
END;
```

In fact, let's reduce our footprint even further. Rather than track a 2000 member True/False array, let's simply store the Customer ID with the Site ID. This will shrink our 2000 member array to a "one member" variable.

```
~~ARRAY DIAGONAL (2000) DIAGONAL0001-DIAGONAL2000;~~
…
IF  LAST.CUSTOMER_ID THEN DO;
   IF  FIRST.CUSTOMER_ID  THEN DO;
      ~~DIAGONAL( SITE_ID ) = 1;~~  DIAGONAL = SITE_ID;
      OUTPUT  DIAGONAL_MATRIX;  /* One cell */
   END;
END;
```

In order to test our process, we need some sample data. The code provided below is set up on the assumption that the majority of people will visit fewer than 12 of the sites we are tracking. However, every 75th tourist record MAY have visited a much larger subset of them.

```
data Tourist_Visits;
  DO Customer_ID = 1 TO 5000;
    MaxVisits = FLOOR( RanUni( 0 )*2000 + 1 ) /
                FLOOR( RanUni( 0 )* 125 + 1 );
    IF  ROUND( Customer_ID, 75 ) = Customer_ID  THEN
          Ceiling_Visits = MaxVisits;
    ELSE  Ceiling_Visits = 12 ;
    NumVisits = MAX( FLOOR( RanUni( 0 )*Ceiling_Visits + 1 ), 1 );
    NumVisitsMax = MAX( NumVisitsMax, NumVisits );
    DO J = 1 TO NumVisits;
      Site_ID = FLOOR( RanUni( 0 )*2000 + 1 ) ;
      OUTPUT;
    END;
  END;
  CALL  SYMPUT( "NumVisitsMax", NumVisitsMax );
run;
```

Executing this code for a sample run, we generated 32,964 observations, with one ambitious traveler having 482 visits to our 2000 sites. Note that given the variability of the program and RANUNI – or, if we had used updated random number generator CALL RAND – each execution will vary.

We will now populate TWO datasets. The first is a simple one, which contains single observations for tourists who visited only one tourist site on our list. The second contains multiple records for each of our tourists, one observation for each two-site combination on their list of visits. We populate an array with the list of sites that each of our tourists visited, then when the array is filled, as denoted by the last record for the ID in question, we write out our tuples. Note that the nested DO loops do not start and end on the same count.

```
DATA Tourist_Visit_Combos(KEEP=Customer_ID  NumVisits  Site_1  Site_2)
     Tourist_Visit_Single(KEEP=Customer_ID  NumVisits  Site_1);
   SET Tourist_Visits;
   BY  Customer_ID;

   ARRAY Site_ID (&NumBuysMax.) Site_ID1-Site_ID&NumBuysMax.;
   IF  NumVisits = 1  THEN  DO;
      Site_1 = Site_ID(1);
      OUTPUT Tourist_Visit_Single;
   END;
   ELSE  DO;
      ArrayPnt + 1;
      Site_ID( ArrayPnt ) = Site_ID;
   END;

   IF  LAST_Customer_ID  THEN DO;
     DO  I = 1 TO NumVisits - 1;
        Site_1 = Site_ID(I);
        DO J = (I + 1) TO NumVisits;
           Site_2 = Site_ID(J);
           OUTPUT Tourist_Visit_Combos;
        END;
     END;
  END;
run;
```

Cutting to the chase:

We run PROC MEANS against the Tourist_Visit_Combos dataset to get counts of the number of tourists who visited each combination of sites. This will create a symmetrical matrix – or rather, ½ of a symmetrical matrix. As we have cut down the number of unique combinations to only those that have actual values, we should find that our density has gone up, and sparsity fallen

CONCLUSION

This paper describes the approaches examined, and the one finally taken, when faced with the issue of system limitations allowing the processing of a sparse matrix. As with any SAS problem, there are several possible solutions, some of which may prove to be more efficient in terms of system resources or programming / testing time. This solution proved to be workable and ran for several years in a production environment, although the exact nature of the problem has been altered in order to protect potentially proprietary data.

REFERENCES

Kuligowski, Andrew and Mendez, Lisa A. (2016) “An Introduction to SAS Arrays”. Proceedings of the SAS® Global Forum 2009 Conference. Cary, NC: SAS Institute, Inc. .https://support.sas.com/resources/papers/proceedings16/6406-2016.pdf

SAS Institute, Inc. (2016) Base SAS 9.4 Procedures Guide, Seventh Edition Companion for Windows, Fifth Edition. https://documentation.sas.com/?docsetId=proc&docsetTarget=p1kwrbpvinbd5bn1cpha9brh92eg.htm&docsetVersion=9.4&locale=en

SAS Institute, Inc. (2016) SAS 9.4 Companion for Windows, Fifth Edition. http://support.sas.com/documentation/cdl/en/hostwin/69955/HTML/default/viewer.htm#p1k3q9i7yknv74n12dc8uqlnwzya.htm

Wikipedia. “Array”. (2019). https://en.wikipedia.org/wiki/Array

Wikipedia. “Matrix (Mathematics)”. (2019). https://en.wikipedia.org/wiki/Matrix_(mathematics)

Wikipedia. “Sparse Matrix”. (2019). https://en.wikipedia.org/wiki/Sparse_matrix

RELATED READING

As stated earlier, the authors are not statisticians and we did not delve into that aspect of the topic. A quick review of papers published at earlier SAS conferences produced the following subset:

Fenchel, Matthew C., McPhail, Gary L., and VanDyke, Rhonda D. (2010) Proceedings of the 10`0 MidWest SAS Users Group Conference. “Using HPMIXED with Other SAS® 9.2 Procedures to Efficiently Analyze Large Dimension Registry Data” Proceedings of the 2010 Midwest SAS Users Group.. https://www.lexjansen.com/mwsug/2010/health/MWSUG-2010-71.pdf

Kuss, Oliver. (2001) "A SAS/IML Macro for Goodness-of-Fit Testing in Logistic Regression Models With Sparse Data". Proceedings of the Twenty-Sixth Annual SAS® Users Group International Conference. Cary, NC: SAS Institute, Inc. https://support.sas.com/resources/papers/proceedings/proceedings/sugi26/p265-26.pdf

Luo, Sheng, and Lin, Xinsheng. “Using SAS Bitwise Functions to Scramble Data Fields with Key”, Proceedings of the Twenty-Second Annual SAS® Users Group International Conference. https://support.sas.com/resources/papers/proceedings/proceedings/sugi22/POSTERS/PAPER248.PDF

Wang, Tianlin and Tobias, Randy. (2009) “All the Cows in Canada: Massive Mixed Modeling with the HPMIXED Procedure in SAS® 9.2”. Proceedings of the SAS® Global Forum 2009 Conference. Cary, NC: SAS Institute, Inc. https://support.sas.com/resources/papers/proceedings09/256-2009.pdf

Zhao, Zheng, Albright, Russell, and Cox, James. (2014) "Processing and Storing Sparse Data in SAS Using SAS Text Miner Procedures". Proceedings of the SAS® Global Forum 2014 Conference. Cary, NC: SAS Institute, Inc. https://support.sas.com/resources/papers/proceedings14/SAS195-2014.pdf

This list is meant to whet the appetite. The reader is heartily encouraged to search for these and other fine papers on www.lexjansen.com

ACKNOWLEDGMENTS

This paper had its roots in a series of PL/I routines designed and written in the mid-1990s by Al Borawski, William Cormier, Cindy McKenzie, Bob Jablonski, and a few other programmers. The issues and potential solutions discussed at the time stayed with me and proved useful over a decade later when a related issue came up in an entirely different environment and context. Even then, it took another decade before the material was actually organized and formatted into a professional paper.

All of the folks who were involved in either of those efforts, whether or not specifically named above, deserve credit for their creativity, dedication, and for helping me to develop into the programmer I am STILL becoming,

CONTACT INFORMATION

In the event of any questions, comments, or whatever, you can contact the authors via email:

Andrew T. Kuligowski
KuligowskiConference@gmail.com

Lisa Mendez, Ph.D.
sasebmendez@gmail.com