Paper 241-2011

Numeric Length in SAS®: A Case Study in Decision Making

Paul Gorrell, IMPAQ International LLC, Columbia, MD, USA

ABSTRACT

Storage space for data is always at a premium, so any effort to reduce the size of SAS® data sets is easy to justify. One common option is to reduce the length of numeric variables. SAS documentation includes 'largest integer' tables to guide length specification, and macros have been created to automate the process. But reducing numeric length is rarely the most effective option. Understanding why requires a practical understanding of (i) how SAS stores numeric variables, (ii) how variable attributes are inherited by other data sets via merges or concatenation, and (iii) the difference between actual and permissible variable values.

This paper reviews these aspects of SAS data sets and argues that the compress=binary option provides both superior file size reduction and improved i/o times for more efficient data processing. In contrast, reducing numeric length cannot match sas compression in reducing file size—and it has no effect on computational efficiency. Concrete examples illustrate each point as well as the decision-making process-- and make it easy to apply to your work.

INTRODUCTION

It's often the case that decisions in one, apparently isolated, area of a project will have implications in other areas. This paper focuses on a common situation encountered by projects that store large amounts of data as SAS® data sets: under what circumstances should the default length of 8 bytes be reduced in order to save storage space?

Even these days when memory is comparatively cheap, storage space for data is always at a premium. So any effort to reduce storage space requirements is intuitively easy to justify. We will consider the decision to reduce (or, not reduce) numeric length as a case study of the various factors that should be evaluated before implementation.

For numeric variables in SAS, the LENGTH statement is an instruction about how to store the values of specified variables. Different systems use different methods to store numbers. SAS uses floating point representation and, by default, stores numeric values using eight bytes. Before discussing specific recommendations, we will first discuss some of the basics of floating point representation and some of its underlying concepts. The goal of this sketch is not to exhaustively explain how to translate base 10 numbers into floating point representation, but rather to give you a more-concrete sense of how the SAS system interprets a statement such as:

(1) LENGTH NVAR1 3 ;

One reason for not delving into the minutiae of floating point representation is that it is rarely of practical value in the day-to-day work of SAS programming and making decisions concerning the LENGTH of numeric data. It is better to have a small set of guiding principles for making decisions in a timely manner. Of course it is always good to know where to go to get detailed information if you need it, and the References section at the end of this paper contains useful resources on numeric precision and related topics.

It is important to distinguish how SAS *displays* numeric data from how it *stores* numeric data. SAS displays the values of numeric variables using either a default FORMAT or one that has been specified by the programmer. Because formats (i.e. the display properties of numeric variables) may be less precise than the stored properties, it's important to keep this distinction in mind when working with, and making decisions about, numeric variables.

We will look in some detail at the COMPRESS=BINARY option in SAS and compare it with the option of reducing the length specification for numbers with the LENGTH statement. In general, compression algorithms save storage space by detecting row-level redundancies, e.g. repeating sequences of zeroes, and encoding the information in a way that takes less space than the original sequence. We'll show that the COMPRESS=BINARY option is more efficient than reducing numeric length and that, for situations where COMPRESS=BINARY is not counter-indicated, is the preferred option. As noted, regardless of the value of the specified stored length, SAS uses the full 8-byte representation of numbers in PROCS and DATA steps. Because saving storage space is the only reason to reduce numeric length, the focus here will be on the relative merits of length reduction and compression with respect to its effect on file size.

There are two perspectives on this: one can look specifically at data-set properties, e.g. the maximum value for a particular variable on a particular data set; but one can also take a step back and think about the content and context in which the data is being used. For example, it is not unusual on many projects for data sets to be updated, or to be the input for updates, on a regular basis. In these situations, what are the consequences of decisions based solely

on the properties of a particular data set? We'll return to this and related questions after reviewing the basics of how SAS stores numbers, i.e. of floating point representation.

We will discuss particular examples later in this paper that show the advantages of using the COMPRESS=BINARY option, bur remember, you need to evaluate the recommendations made here with respect to your own data and project requirements.

HOW SAS STORES NUMBERS

There's quite a bit of detail in this section. In general, there would seem to be little reason to pay attention to this level of detail in making day-to-day programming decisions. So why go to the trouble of devoting time to it? For two reasons: (i) so that you are in a better position to evaluate the general claims made here in terms of your particular situation; (ii) you may, at some point, encounter a programming problem that requires a detailed understanding of how SAS stores numbers—and how this differs from how SAS displays numbers.

FLOATING POINT REPRESENTATION

Most papers on floating-point representation and numeric precision only tell part of the story. The part of the story I'm going to tell focuses on giving you enough information to understand, or, at least give you a jumpstart in understanding, more-detailed discussions of these topics.[1] I also hope it allows you to evaluate the guidelines that I give at the end of this paper (see also Thacher 2011[2] for a detailed discussion of floating-point arithmetic in SAS).

The basic unit of storage is the *bit* (binary digit). As the term *binary* suggests, there are two possible values: 0 and 1. A sequence of 8 bits is called a *byte*.

SAS stores numeric data using 64 bits (eight bytes). The eight bytes are divided among 3 different types of information: the *sign*, the *exponent*, and the *mantissa*. An important fourth type of information is the *base*, i.e. the number raised to a power. These are terms for the parts of a number in scientific notation. Floating point representation is just one form of scientific notation. In this system, each number is represented as a value (the mantissa) between 0 and 1 raised to a power of 2 (in a binary, base 2, system). The term *decimal* in 'decimal point' assumes a base 10 system, so the term '*radix* point' is often used when the base is not 10. The radix point for a number is moved (i.e. it *floats* in the picturesque speech of computer science) to the left until the number is a fraction between 0 and 1.

If we use the more-familiar base 10 system, the number 234 would be represented as $.234 \times 10^3$. Placing the decimal point to the left of the number is called 'normalizing the value.' This normalization yields a value between 0 and 1. In addition to the mantissa (.234) and the exponent (3), a full representation of 234 would include the sign (negative or positive).

Here's the byte layout for a 64-bit number in the IEEE system used by Windows (where S = sign; E = exponent; and M = mantissa):

(2) SEEEEEEE EEEEMMMM MMMMMMMM MMMMMMMM MMMMMMMM MMMMMMMM MMMMMMMM MMMMMMMM

byte 1 byte 2 byte 3 byte 4 byte 5 byte 6 byte 7 byte 8

That is, there's 1 bit for the sign, 11 bits for the exponent, and 52 bits for the mantissa. The number of exponent bits determines the magnitude of the numbers that can be represented. The number of mantissa bits determine the precision. For example, an IBM mainframe system will use 56 bits for the mantissa, allowing for greater precision than you get on a PC. In the following discussion I will focus on the mantissa since that is the part of the representation affected by the LENGTH statement.

For 52-bit systems, the functional equivalent of 53 bits is achieved by assuming an 'implied' bit. That is, since the only possible non-zero digit is 1, we can just assume an initial mantissa bit with a value of 1. We'll see that the existence of this implied bit solves an apparent puzzle when we look at numbers in binary where all the mantissa bits have values of 0.

When you use a LENGTH statement such as (1), you are telling the SAS system to only use the first 3 bytes to store the number, i.e. bytes 1-3 in (2). You save 5 bytes per file record this way, but you are potentially losing precision because you are sacrificing 40 mantissa bits. Note that you cannot specify length in terms of bits, or specify which bytes to use. If LENGTH is 3 then you only have the first 3 bytes in (2).

We will look at integers and fractions, in turn, before discussing length reduction and data set compression.

INTEGERS

Below is a table showing (what is often referred to as) the largest integer that can be represented accurately for a given LENGTH specification. The numbers in the table will be different for IBM and VAX computers (consult the *SAS Companion* for your operating system). Table 1 shows largest integers for specified lengths for SAS numeric variables under Windows and Unix systems.

Table 1 Length Specification and Largest Integer Value for SAS Numeric Variables

LENGTH IN BYTES	LARGEST INTEGER (PC/UNIX)
2	NOT ALLOWED
3	8,192
4	2,097,152
5	536,870,912
6	137,438,953,472
7	35,184,372,088,832
8	9,007,199,254,740,992

Now let's look at the 64-bit representation of 8,192 and 8,193 to see why 8,192 is the 'largest integer' that can be accurately represented. You can create 64-bit output by using the BINARY64. format (The spaces between bytes were added later).

(3) The 64-bit representation of 8,192:

01000000 11000000 00000000 00000000 00000000 00000000 00000000 00000000

(4) The 64-bit representation of 8,193:

01000000 11000000 00000000 **10000000** 00000000 00000000 00000000 00000000

The first thing to notice is that the difference between 8,192 and 8,193 is in byte 4. If you are limited to the first 3 bytes then this difference is eliminated and, as far as the computer is concerned, the numbers are identical (because bytes 1-3 are identical).

We can show this with the SAS program in (5).

(5)
```
DATA ONE ;
        LENGTH VAR1 VAR2 3 ;
        VAR1 = 8192 ;
        VAR2 = 8193 ;
RUN;

DATA TWO;
        SET ONE;
        PUT VAR1=22.16 ;
        PUT VAR1=BINARY64. ;
        PUT VAR2=22.16 ;
        PUT VAR2=BINARY64. ;
RUN;
```

Here's the LOG output you'd get:

(6)
```
VAR1= 8192.0000000000000000
VAR1= 01000000 11000000 00000000 00000000 00000000 00000000 00000000 00000000

VAR2= 8192.0000000000000000
VAR2= 01000000 11000000 00000000 00000000 00000000 00000000 00000000 00000000
```

There are a couple of things going on here we need to unpack. First, VAR2 was initially assigned a value of 8,193, but in data set TWO it's 8,192. The reason is that the LENGTH specification of 3 has removed the distinguishing information in byte 4. When SAS stored this variable it only used three bytes.

Second, and this is a point we'll come back to later, in the DATA step (i.e. the program data vector [PDV]), SAS uses the full 8-byte representation of numbers (Note that, for character variables, the LENGTH statement applies both to the PDV and the output data set). If a variable was stored as LENGTH 3, then bytes 4 through 8 are 'filled in' with zeros. This is what has happened with VAR1 and VAR2. When data set ONE is stored, VAR1 and VAR2 are stored with 3 bytes. When they are read for the DATA step that creates TWO, they are expanded to eight bytes, with the last 5 bytes all zeros.

For VAR1 (8,192) this does not result in any loss of precision because the last 5 bytes are all zeros in the original 8-byte representation. Note that there are no mantissa bits with a value of 1. This is because 8,192 is a power of 2 (2^{13}) and so only the implied bit needs a value of 1.

For VAR2 (8,193) there *is* a loss of precision when stored as LENGTH 3 because the 4th byte contains a "1", and only zeroes are used to restore the 8-byte representation used in processing. This information (the "1" in the 4th byte) is lost when the data set is stored. In this case the filling in of bytes 4-8 with zeros results in a value equivalent to 8,192—so the original value of VAR1 (8,193) has been changed to 8,192, i.e. the "1" in the 4th byte has been replaced by a zero.

Let's expand on the DATA step in (5) to illustrate a couple of points.

```
(7)     DATA ONE ;
            LENGTH VAR1 VAR2 VAR3 3 ;
            VAR1 = 8192 ;
            VAR2 = 8193 ;
            VAR3 = 16384 ;
            PUT VAR1=22.16 ;
            PUT VAR1=BINARY64. ;
            PUT VAR2=22.16 ;
            PUT VAR2=BINARY64. ;
            PUT VAR3=22.16 ;
            PUT VAR3=BINARY64. ;
        RUN;

        VAR1= 8192.0000000000000000
        VAR1= 01000000 11000000 00000000 00000000 00000000 00000000 00000000 00000000

        VAR2= 8193.0000000000000000
        VAR2= 01000000 11000000 00000000 10000000 00000000 00000000 00000000 00000000

        VAR3= 16384.0000000000000000
        VAR3= 01000000 11010000 00000000 00000000 00000000 00000000 00000000 00000000
```

The first thing to notice is that the LENGTH statement in (7) has no effect on the length of the variables created in the DATA step. That's why the output of the PUT statement for VAR2 (8,193) shows the correct value. Note also that for VAR3 (16,384) there are only zeroes in bytes 4 through 8.

This demonstrates that it's not, in fact, the case that 8,192 is the largest integer that can be accurately represented with LENGTH 3 (on a PC or UNIX system). What *is* true is that 8,192 is the largest integer *of a continuous range of integers* that can be accurately represented with 3 bytes. The same is true for the other integers in the right-hand column in the table. These 'largest integer' tables are still handy references for quick decisions, but it's important to know that exceptions exist, and that these exceptions (which, actually, do prove the rule!) may be relevant to your own project requirements.

As shown in (8) the LENGTH statement takes effect when the variables are stored. The PUT statements below show the effect of LENGTH = 3 specification on the stored values for data set ONE.

```
(8) DATA TWO;
        SET ONE;
        PUT VAR1=22.16 ;
        PUT VAR1=BINARY64. ;
        PUT VAR2=22.16 ;
        PUT VAR2=BINARY64. ;
        PUT VAR3=22.16 ;
        PUT VAR3=BINARY64. ;
    RUN;
```

```
VAR1= 8192.0000000000000000
VAR1= 01000000 11000000 00000000 00000000 00000000 00000000 00000000 00000000

VAR2= 8192.0000000000000000
VAR2= 01000000 11000000 00000000 00000000 00000000 00000000 00000000 00000000


VAR3= 16384.0000000000000000
VAR3= 01000000 11010000 00000000 00000000 00000000 00000000 00000000 00000000
```

Because, unlike with VAR2 (8,193), bytes 4 through 8 are all zeroes for VAR3 (16,384), there is no loss of precision when this value is stored. In fact, because 16,384 is a power of 2 (2^{14}), only the implied mantissa bit has information.

FRACTIONS

It's also important to remember that these tables apply only to *integers*. The situation is somewhat different when we look at fractions. Let's compare incrementing by 1/10 (0.1) with incrementing by 1/2 (0.5), with nine repetitions each.

```
(9)     DATA ONE;
            VAR1 = 1 ;
            VAR2 = 0 ;
            DO X = 1 TO 10 ;
                    VAR2 + 0.1 ;
            END;
            VAR3 = -4 ;
            DO X = 1 TO 10 ;
                    VAR3 + 0.5 ;
            END;
            IF VAR1 = VAR2
                    THEN PUT VAR1 'EQ ' VAR2 ;
            ELSE PUT VAR1 'NE ' VAR2 ;
            IF VAR1 = VAR3
                    THEN PUT VAR1 'EQ ' VAR3 ;
            ELSE PUT VAR1 'NE ' VAR3 ;
        RUN;
```

If you run this DATA step, the LOG output will be:

(10)

```
1 NE 1          [for 0.1]

1 EQ 1          [for 0.5]
```

The first LOG message looks like a clear contradiction. How can the result be that 1 does not equal 1? The answer, of course, is that the stored value of VAR2 differs from its display form. Remember, all displayed numeric output is formatted output, so what you see isn't necessarily what you have. See the SAS Technical Support Note 654 (TS-654) for more details on fractions and other numeric precision issues. Here's the 8-byte representation of VAR1 (1) and VAR2:

(11)

```
VAR1= 00111111 11110000 00000000 00000000 00000000 00000000 00000000 00000000

VAR2= 00111111 11101111 11111111 11111111 11111111 11111111 11111111 11111111
```

Once we see the 8-byte representation, it's clear that VAR1 and VAR2 have different stored values. The output of the comparison also shows that SAS is comparing stored values and not display values. The reason that VAR2 does not equal 1 is because 0.1 cannot be represented precisely in a binary system (though it's straightforward in a decimal system) and this imprecision iterates with each addition in the DO loop.

Now let's compare VAR1 with VAR3:

(12)

VAR1= 00111111 11110000 00000000 00000000 00000000 00000000 00000000 00000000

VAR3= 00111111 11110000 00000000 00000000 00000000 00000000 00000000 00000000

Here we see that incrementing by 0.5 results in a stored value that is equal to 1. Why here and not with 0.1? Notice that, in contrast to VAR2, where we saw repeating "1s", bytes 3-8 are all zeros. In fact, in binary representation, 0.5 looks a lot like 8,192 and 16,384. This is because these numbers are all powers of 2: $0.5 = 2^{-1}$.

For a good discussion of comparisons with numeric variables see Go 2008[3]. Go discusses apparent puzzles where SAS returns FALSE when evaluating "0.3 = (3*0.1)", as well as rounding by SAS formats and the use of FUZZ and the 'zero fuzz' functions (CEILZ, FLOORZ, INTZ, MODZ, ROUNDZ).

SPECIFYING NUMERIC LENGTH

It's clear that the magnitude of a number is an imperfect predictor of whether or not it can be accurately represented with a reduced number of bytes. The numbers 1/2 and 16,384 can be accurately represented with 3 bytes, but 8,193 cannot.

Let's consider a numeric variable HOSPITAL_DAYS that indicates how many days in a year a person was in the hospital. The largest possible value is 365 in non-leap years. But let's assume that half-days (but no other fractions) are possible, so values such as 189.5 are allowed. This is a variable that, despite the non-integer values, is a candidate for LENGTH 3.

Recognizing candidates for reduced LENGTH is an important part of the decision process, and knowing a bit (no pun intended) about how SAS stores numbers is essential for this. The larger question concerns other factors that are involved in making such a decision.

Here's an example that illustrates how you often need to consider a larger context when making, what appear to be, narrow, technical decisions. Assume that you have a SAS data set with a variable for annual out-of-pocket dental expenditures. The values are all integers and the largest value for this variable on the file is 8,192. Should you set LENGTH 3?

Let's say you do. Fast forward a year or so and the task is to input this file, increase dental expenditures by 13% (to account for inflation), and round the result to the nearest integer. Here's a DATA step that shows the potential problem.

(13)
```
DATA ONE;
        LENGTH DENTOOP1 3 ;
        DENTOOP1 = 8192;
        DENTOOP2 = 8192;
RUN;

DATA TWO;
        SET ONE;
        DENTOOP1 = ROUND((DENTOOP1*(1.13));
        DENTOOP2 = ROUND((DENTOOP2*(1.13));
        PUT DENTOOP1= ;
        PUT DENTOOP2= ;
RUN;

LOG Output:     DENTOOP1=9256;
                DENTOOP2=9257;
```

In data set ONE the variable DENTOOP1 has a LENGTH of 3 whereas DENTOOP2 has a LENGTH of 8 (the default). After the 13% increase and rounding in the second DATA step, we see that the modified values depend on the LENGTH specification. The consequence of setting LENGTH=3 for DENTOOP1, even though it didn't cause any problems for the original data set values, is inaccurate output down the road. Tracking down this type of error can be a nightmare because the reason for the inaccuracy isn't within the program generating the data set—it's back a year or so when the original data set was created and the decision was made to set LENGTH based only on the particular data set properties. Because the LENGTH specification of variables is implicitly transferred from one data set to another, being conservative and thinking outside the data set can save a lot of headaches.

Of course this problem could have been avoided if the programmer updating dental expenditures had simply

- run a PROC CONTENTS and noticed that the LENGTH is 3 bytes
- realized that variable attributes like LENGTH can be inherited by one data set from another
- known the integer values for which accuracy is preserved with this LENGTH
- determined that there are values that, if increased by 13%, would exceed 8,192
- known how to stop the default attribute inheritance for this variable
- placed a new LENGTH statement in the DATA step that updated the variable's values

If the steps listed above are second nature to you, everyone you work with, and everyone you will ever hire, then this would favor a decision to reduce numeric length based simply on the values in the data set.

I would argue, however, for a more conservative approach. This approach would first distinguish between *actual* and *permissible* values. By actual values I mean the set of values that happen to be true for a variable on a particular data set. Given reasonable QC, the set of actual values should be a subset of the set of permissible values. But the range of permissible values cannot be determined by examining properties of a single file. They require that you think outside the data set.

Let's take the dental expenditure variable as an example. On the data set I discussed earlier, the maximum value was 8,192. But this just happened to be true. It could have been different. We can contrast this with a numeric variable such as BIRTH_MONTH, which might have a permissible range of 1-12 and be restricted to integers. For various reasons it is possible that, on a particular data set, not all of these are actual values. But the point is that, if LENGTH 3 is set for this variable, it's hard to imagine a situation where values for this variable would ever create imprecision issues. Of course the variable might have a value of 99 or whatever to indicate "UNKNOWN", but, unlike the dental expenditure variable discussed earlier, the permissible range is well defined and clearly within the reduced LENGTH specification.

For the HOSPITAL_DAYS variable, given the permissible range, and the fact that the only permissible non-integer is 0.5, this variable could be stored with LENGTH=3 without any concern for immediate or down-the-road problems.

One reason to be tempted by a less-conservative approach is the potential saving in storage space. In the next section I will discuss an alternative way of reducing the storage space required for numeric variables. It is important to remember that the only reason to reduce LENGTH with numeric variables is to save disk space. There is no increase in computational efficiency. All numeric variables are expanded to 8 bytes for computations performed in DATA and PROC steps.

COMPRESS=BINARY

Beginning with SAS Version 8, one of the options for compressing SAS data sets is COMPRESS=BINARY. This is a compression method that is recommended for numeric data. It is efficient in two ways: (i) it can dramatically reduce storage requirements; (ii) it can decrease the time it takes to read in the data set. You can use this as a system or data-set option. The latter is recommended as it is more specific and allows you to evaluate its use on a case-by-case basis. The syntax is straightforward:

(14)
```
DATA TWO (COMPRESS=BINARY);
        SET ONE;
....
RUN;
```

Intuitively, a compression algorithm detects patterns, e.g. repeating sequences of single or multiple digits, and encodes them in a more-efficient way. For example, the file size of this paper would be reduced somewhat if every instance of the sequence *number* in this paper were replaced by *num*. In SAS, compression operates on rows (observations) so only patterns within a particular row are detected and encoded. In particular, COMPRESS=BINARY is a form of Ross Data Compression (RDC) but a discussion of the specifics of RDC is beyond the scope of this paper.

It's important to remember that there's no such thing as a free lunch, so there is typically some added CPU time involved with processing data where either numeric length has been reduced by the LENGTH statement or COMPRESS=BINARY has been used. Remember, SAS always uses the full 8-byte representation of numbers for all DATA steps and PROCs.

Let's look at some examples. We'll use publicly-available data so the results presented here can be tested and verified. The Agency for Healthcare Research and Quality (AHRQ, part of the U.S. Department of Health and Human Services) conducts the annual Medical Expenditure Panel Survey (MEPS). Each year it releases a series of data files (in ASCII or SAS transport format) focused on different areas of healthcare use and cost. The 2005 Medical Expenditure Panel Survey [MEPS] public use file HC-094A ("The 2005 Prescribed Medicines File") has 70 variables and 317,587 observations. Observation length is 560. Just over 77% of the variables are numeric (54). Of these 37 are LENGTH 3, 1 is LENGTH 4 and 16 are LENGTH 8.

This file has information about prescribed medicine purchases in the United States during the year, including medication name, national drug code, medicine properties (quantity, form, strength, therapeutic class), pharmacy type, amount paid by source of payment, etc.[4] At first glance it doesn't appear to be an ideal candidate for compression because its observation length is only 560. But it's clear from the LOG Note below that there is a substantial reduction in disk space usage.[5] The uncompressed file is 171 MB; compressed with the COMPRESS=BINARY option, it uses 66 MB. As the observation (row) length of a file increases so do the potential benefits of compression. For a typical project or study, this benefit will be multiplied many times over.

(15)

```
NOTE: There were 317587 observations read from the data set TEST.HC094A.
NOTE: The data set TEST.HC094A_C has 317587 observations and 70 variables.
NOTE: Compressing data set TEST.HC094A_C decreased size by 61.27 percent.
      Compressed is 4242 pages; un-compressed would require 10952 pages.
```

Consider another MEPS data file, the 2005 Full-Year Consolidated Data File, public use file HC-097.[6] This file has 1,654 variables and 33,961 observations. Observation length is 7,744. Just over 99% of the variables are numeric (1,639). Of these 1,071 are LENGTH 3, 14 are LENGTH 4 and 554 are LENGTH 8. Obviously a much wider file--and a great candidate for compression.

Uncompressed this file uses 266 MB of disk space. The compressed file uses 69 MB. Also, the number of data set pages is reduced from 16,995 to 4,410. This latter reduction, in part, allows for faster I/O times because, as described in SAS documentation of the BUFSIZE= option, "The page size is the amount of data that can be transferred from a single input/output operation to one buffer." So, the fewer pages per data set, the fewer I/O operations required. In my experience this saving in I/O time more than offsets any increase in CPU time. But of course this benefit may be outweighed if your project incurs charges for CPU time but not for I/O time (or, as is usually the case, if the charge per unit of time is less for I/O then CPU).

In many project contexts this creates a win-win situation: you save both storage space and the time it takes to read in a file. I have run numerous tests on a variety of data sets and the following generalization holds:

(16) If the COMPRESS=BINARY option reduces file size, then SAS takes less time to read in the file.

We can see the benefits of compressing this wider file in the following LOG Note:

(17)

```
NOTE: There were 33961 observations read from the data set TEST.HC097.
NOTE: The data set TEST.HC097C has 33961 observations and 1654 variables.
NOTE: Compressing data set TEST.HC097C decreased size by 74.05 percent.
      Compressed is 4410 pages; un-compressed would require 16995 pages.
```

Obviously COMPRESS=BINARY is an option with lots of advantages. More CPU resource may be required, but this cost is usually more than offset by the benefits: a significant reduction in file size and number of data set pages.

It is important to note that there are cases where compression will be enabled despite it increasing data set size. As a test of this a long, narrow, data set was compressed with the BINARY option. This data set, as the LOG note below shows, has only two variables (both numeric). Compressing this data set actually increases its size, as the LOG NOTE also shows (yet another reason to always read the LOG carefully, even if the output appears correct!).

(18)

```
NOTE: There were 343960 observations read from the data set TEST.XWLK.
NOTE: The data set TEST.XWLK_C has 343960 observations and 2 variables.
NOTE: Compressing data set TEST.XWLK_C increased size by 74.23 percent.
      Compressed is 2380 pages; un-compressed would require 1366 pages.
```

The general rule of thumb is that the effectiveness of compression increases with observation length. The overhead associated with a SAS-compressed data set is 12-24 bytes per observation, so this must be taken into account when making decisions about compression.

One final advantage of using SAS-internal compression: there is no need for an explicit utility or command to uncompress the data set. SAS will automatically uncompress the file.

In general, the potential advantages of using the COMPRESS=BINARY option are:

- Significant reduction in disk storage usage
- Significant reduction in I/O time
- No need for an explicit 'uncompress' command
- Avoidance of inappropriate use of length reduction

Some potential drawbacks include:

- Overhead of approximately 12-24 bytes per observation
- Compressing 'narrow' data sets may increase file size
- May require additional CPU time
- Cost of additional CPU time may be greater than savings due to reduced I/O time
- Cannot be input to conversion utilities such as Stat/Transfer®

Considering the relative costs and benefits of reduced CPU and I/O time is just another example of the need to think outside the data set. For decisions such as the use of the COMPRESS=BINARY option, a set of inter-related factors usually needs to be considered. Similarly if the data set need to be imported by some external application or utility (such as Stat/Transfer), this would argue against the use of SAS-internal compression.

Note that, for situations where storage space is not an issue, the use of SAS indexes may be appropriate for reducing program run time (for a comprehensive discussion see Raithel[7]). Here the focus is on the relative benefits of compression and numeric length reduction for decreasing file size, with reduced I/O times an added benefit.

COMPARING COMPRESSION AND LENGTH REDUCTION

In the section on specifying numeric length I argued for a conservative approach, i.e. taking the SAS default length of 8 bytes as the default for your own project data sets. In the previous section I recommended the use of the COMPRESS=BINARY option unless it is specifically counter-indicated. In this section we'll compare more directly these two approaches to reducing the storage space required for numeric variables.

A great macro for reducing the length of variables on a data set to the minimum required is the %SQUEEZE macro written by Bettinger.[8] Although the %SQUEEZE macro operates on both character and numeric variables, we'll continue to focus on numeric variables. The macro uses the TRUNC function in a series of conditionals to successively test the minimum length specification consistent with the accurate representation of the maximum value of numeric variables on a data set. Note that a NOCOMPRESS option functions to exclude listed variables from length reduction. This option allows you to think outside the data set and, for example, exclude the dental expenditure variable discussed above. That is, it allows you to take into account permissible values that don't happen to be represented on the data set being input to the macro.

Because %SQUEEZE tests successively smaller length specifications for equality with the maximum value of the original input variable, it is smart enough to recognize, for example, that 16,384 can be accurately represented with the minimum numbers of bytes allowed (3 on a Windows system). As discussed in the INTEGERS section, this is better than simply checking values against a table which lists largest integer values for various byte length specifications (see the discussion in that section of the DATA step in (7)).

Using the %SQUEEZE macro is simple:

(19) %SQUEEZE (DSNIN, DSNOUT, <*options*>)

For this comparison we'll use one of the data sets discussed in the last section, the HC-097 MEPS file. As noted there, using the COMPRESS=BINARY option reduces file size from 269 MB (17,216 data set pages) to 69 MB (4,382 data set pages). The %SQUEEZE macro resulted in a file-size reduction to 179 MB (11,481 data set pages). In particular, %SQUEEZE reduced the length specification of 537 of the 554 numeric variables of length 8 to length 3 or 4.

As shown in the last row in the table below, there is also some small advantage to be gained by using both compression and length reduction on a data set. For all data sets listed here, data set page size was identical (16,384 bytes).

Table 2 Comparison of File Size Reduction with SAS COMPRESS=BINARY and %SQUEEZE Macro

OPTIONS	MEPS HC097 DATA SET SIZE (MB)	MEPS HC097 DATA SET, # Data Set Pages
UNCOMPRESSED	266	16,995
SQUEEZED	177	11,334
COMPRESS=BINARY	69	4,410
SQUEEZED + COMPRESS=BINARY	68	4,372

We can see why, in general, compression is more efficient than length reduction by taking a look at the 3-byte representation of the numbers 8,192 and 16,384. Imagine this situation repeated a few million times in a data set.

(20) 8,192

01000000 11000000 00000000

(21) 16,384

01000000 11010000 00000000

It's clear that, even when numeric length is reduced as recommended, a substantial number of repeating sequences remain that can be captured by the compression encoding. In part the advantage of compression over length reduction is due to the fact that the unit of length reduction is the byte, whereas compression can operate over any sequence of repeating bits. For the integers 8,192 and 16,384 on a Windows system, the minimum length specification is 3 bytes. The 3-byte sequences shown in (20) and (21) contain a substantial number of repeating zeroes—just what any compression algorithm loves.

CONCLUSION

Knowing the basics of how SAS stores numbers leads to more informed decisions regarding LENGTH specifications for numeric variables. The default specification in SAS is also the maximum specification, and this is exactly right. The costs and benefits of reduced numeric length should be carefully considered.

One guideline for storing numbers in SAS data sets is to only reduce length if the set of *permissible* values can be accurately represented in the number of bytes specified. Recall that determining *permissible*, as opposed to *accidental*, values of a variable requires that you consider the general context within which your data to being used. Data sets do not exist in a vacuum and their properties must be evaluated within the context of their use.

It is important to remember that reducing numeric length does not affect DATA or PROC steps, so the only reason for doing so is to save disk space. For many SAS data sets, considerable saving of disk space can be achieved with the COMPRESS=BINARY option.

Note that, even with a full 8-byte representation, not all numbers can be accurately stored. The ROUND function may be used to correct for this either by rounding the stored values of numeric variables, or by rounding for the specific purpose of a comparison within a DATA step. In general it is preferable to store the full values of numeric variables and to round for specific comparison and display purposes.

The general guidelines for the use of length reduction and compression with numeric variables are:

- Distinguish between *permissible* and *actual* values for a variable in a data set (Think outside the data set!)
- Use the default LENGTH specification of 8 bytes unless all the permissible values for a variable can be accurately represented in fewer bytes
- If there are no specific counter-indications, use COMPRESS=BINARY whenever it reduces data set size

Counter-indications include situations where the data set is being used as input to a conversion utility such as Stat/Transfer, or situations where the costs of additional CPU time outweigh the benefits of reduced file size or I/O time.

These are guidelines, not rules, and (as always) you need to take into account any specific properties of your data set or project that would argue for a different approach.

In addition, going beyond the basics, and the intuitively limited domain of individual decision making, allows for more informed decisions that are consistent with both particular and general project goals. The principles of the case study presented here are applicable across a wide range of programming and data-storage decisions faced by programmers and project mangers.

REFERENCES

[1] For more detailed discussion, see "Numeric Precision 101" (available at http://ftp.sas.com/techsup/download/technote/ts654.pdf and "Dealing with Numeric Representation Error in SAS Applications" (available at http://ftp.sas.com/techsup/download/technote/ts230.html.

[2] Thacher, Clarke. Why .1 + .1 Might Not Equal .2 and Other Pitfalls of Floating-Point Arithmetic. Proceedings of SAS Global Forum 2011 (275-2011).

[3] Go, Imelda C. Rounding in SAS: Preventing Numeric Representation Problems. Proceedings of SESUG 2008 (PO-082). Available at http://analytics.ncsu.edu/sesug/2008/PO-082.pdf .

[4] The 2005 MEPS Prescribed Medicines file is available at http://www.meps.ahrq.gov/mepsweb/data_stats/download_data_files_detail.jsp?cboPufNumber=HC-094A).

[5] All output reported in this paper was generated by SAS 9.1.3 (SP4) on Windows XP_PRO, with BUFNO=1 and BUFSIZE=0. For all data sets, data set page size = 16,384 bytes. BUFSIZE=0 is an instruction to use the minimum optimal page size. Output has been replicated using SAS 9.2.

[6] The 2005 MEPS Full-Year Consolidated data file is available at http://www.meps.ahrq.gov/mepsweb/data_stats/download_data_files_detail.jsp?cboPufNumber=HC-097

[7] Raithel, M. *The Complete Guide to SAS® Indexes*. Cary, NC: SAS Institute Inc.

[8] Sample 267: %SQUEEZE-ing Before Compressing Data, Redux (http://support.sas.com/kb/24/804.html).

ACKNOWLEDGMENTS

I would like to thank audiences at the Washington DC SAS® Users Group (DCSUG) and the NorthEast SAS® Users Group (NESUG), as well as the editors at *Pharmaceutical Programming*, where earlier versions of some of this material were presented. Their comments are greatly appreciated.

CONTACT INFORMATION

Please send any comments or questions to:

Paul Gorrell
Director of Analytic Programming
IMPAQ International, LLC
10420 Little Patuxent Parkway
Suite 300
Columbia, MD 21044
pgorrell@impaqint.com