

Techniques to Improve Efficiency in SAS Programs

David Sommer, Optimal Systems, Inc.

This tutorial owes much to "SAS® Programming Tips: A Guide to Efficient SAS® Processing" (Copyright © 1990 by SAS Institute, Inc., Cary, NC, USA). I present what I have found to be some of the most useful ideas, suggest where some of the tips may not apply, and include some timing data.

Efficiency — what does it mean?

- Computer efficiency - a trade-off
- Human vs Computer

Computer efficiency: often involves substituting a resource which is abundantly available for one which is not. On PC's, CPU time is often relatively more abundant than I/O time. And if additional memory is available, this can save both CPU and I/O.

- CPU vs I/O: use of disk-caching on PC SAS; use of numeric variable lengths less than 8 bytes.
- Additional MEM can save CPU and I/O time in sorting. Use of Expanded MEM in PC saves CPU and I/O.
- Mainframe: very large memory requests shorten clock time (but may make you a bad citizen!)

Human vs computer:

- SAS is intended to save human time at the expense of computer time (as compared to FORTRAN, C, etc.)
- Some practices recommended in SAS programming tips use more programmer time to save computer time - *I avoid* these ideas in this talk.
- Among the tips discussed here, good programming practice *is no harder* than bad.

Minimize variable lengths

Or, what precision do I really need?

Length	Significant Digits	Largest Integer Represented Exactly On PC's	Largest Integer Represented Exactly On IBM Mainframes
2	2	(not allowed on PC)	255
3	4	8,192	65,535
4	6	2,097,152	16,777,215
5	9	536,870,912	4,294,967,295
6	11	137,428,953,472	1,099,511,627,775
7	13	35,184,372,088,832	281,474,946,710,655
8	16	9,007,199,254,740,992	72,057,594,037,927,935

Within a dataset, all numeric variables are 8 bytes in length. This means that variables coming from datasets that have shorter lengths are "padded" to 8 bytes. In SAS, *all numbers* are represented as floating point. Floating point numbers consist of three parts: a sign bit, an exponent, and the mantissa (which is always a binary number).

On IBM® mainframes, the sign bit and exponent occupy the entire first byte. The exponent is a *power of 16*. This means that in estimating precision you should recognize that up to three leading bits of the mantissa may be zero. Therefore, the number of significant bits in mantissa is at least: $8 \cdot (\text{bytes} - 1) - 3$, which may be simplified to $8 \cdot \text{bytes} - 11$. Example: a 4 byte number has at least $8 \cdot 4 - 11 = 21$ significant bits.

On PC's, $1\frac{1}{2}$ bytes are devoted to sign and exponent, giving the exponent 11 bits. Here, the exponent is a *power of 2*. Also, since a "normalized" floating point number would always have a leading 1 bit in the mantissa, the designers of the PC floating point format realized that they could actually *omit* the leading bit (and treat it as 1). Therefore, the number of bits in the mantissa of a PC floating point number is $8 \cdot \text{bytes} - 12 + 1$, or $8 \cdot \text{bytes} - 11$. Coincidentally, that is the same as the minimum significant bits for the IBM mainframe floating point number.

In estimating *decimal* significant digits, multiply the number of mantissa bits by .3. (For the mathematically inclined, .3 is $\log_{10} 2$.) We can substitute the expression for bits = $8 \cdot \text{bytes} - 11$, and get:

significant digits = 2.4·bytes - 3.3

Example: a 4-byte number will have $2.4 \cdot 4 - 3.3 = 6.3$, or a little better than 6 significant digits. The preceding table has a column "Significant Digits" showing for each numeric length the number of significant digits. Note that this is the same for PC's and IBM mainframes.

Now realize that the way a number is represented in a floating point format depends on the base. In fact, some fractional numbers can be represented exactly in one base but not in another. Example: $1/5$ is represented exactly in base 10 (0.2), but $1/3$ is not (0.333333...). In base 16 (IBM mainframe) or base 2 (PC), many common terminating decimal values are non-terminating. Examples: 0.5, 0.25, 0.75 are all terminating as binary fractions because they have only powers of 2 in their denominator ($1/2$, $1/4$, $3/4$ respectively). But 0.1, 0.2, 0.3 are all non-terminating, because their fractional representation involves a power of 5 in the denominator. Conclusion: *almost all* real numbers (or even rational numbers) are non-terminating in any particular base! The following example illustrates the obvious fact that when you compare the representation of the same number in two different lengths it usually won't match.

SAS Institute Cries Wolf!

In SAS Programming Tips, p. 78: "Caution do not shorten numeric variables containing fractions." This is actually an overstatement of the real facts. Here's an example showing effect of differing lengths. The comment /*None!*/ at line 7 expresses the fact that x and y are always identical *inside* this dataset, because both have the full 8 byte length. But at line 13, x and y will almost always be different because in the WORD.FRED dataset, x was saved with length 3, while y was saved with the default length of 8. In the dataset beginning at line 11, y has its full precision, but x is "padded" to bring its internal length up to 8.

```
1  data fred;
2    length x 3;
3    drop i;
4    do i = 1 to 9;
5      x = ranuni(0);
6      y = x;
7      if x ne y then put x = y = ; /*None!*/
8      output;
9    end;
```

```
10  run;
11  data _null_;
12    set fred;
13    if x ne y then put x = y = ; /*Most all!*/
14  run;
X=0.80444 Y=0.80451
X=0.72131 Y=0.72140
X=0.10884 Y=0.10885
X=0.51330 Y=0.51334
X=0.71118 Y=0.71126
X=0.57812 Y=0.57815
X=0.09953 Y=0.09954
X=0.70019 Y=0.70025
X=0.06288 Y=0.06288
```

Understanding how numbers are represented in floating point formats, we see why this happens. In general, making an EQ or NE comparison of fractional values can cause trouble, but in fact there is little practical need for such comparisons. If you do need to make such comparisons, it should be done within some tolerance, like this:

if abs(x-y) le tol then ...

(where "tol" is either a literal, like .001, a macrovariable, or a regular variable.

Test Results on Variable Length

My experiments were on a Gateway 2000® 4DX2-66V. Running SAS 6.08 under OS/2® 2.0, I created two datasets both consisting of 10,000 observations of 50 numerical variables. The difference between the two was that dataset FULL had full precision (8 bytes), while dataset NARROW had just 3 bytes. Not surprisingly, NARROW was approximately 3% the size of FULL (1,513,216 bytes versus 4,112,640).

In each dataset, one variable was generated as a random number. When I did a PROC SORT with this random number as the BY variable, NARROW sorted in 14.63 seconds, while FULL required 51.59 seconds, or 3.6 times longer.

This dramatic difference did not appear in a simple dataset:

```
data _null_; set FULL; run;
```

which took 3.25 seconds, while the corresponding dataset for NARROW took 2.27 seconds. Still, FULL did require 43% longer than NARROW.

Read only fields you need

Suppose only variables jday, sales, and invout are actually needed. Then here is an "acceptable" and a "more efficient" way to read the data:

Acceptable

```
data oct90;
    infile file-spec;
    input jday rcode $ sales exps salhrs exphrs
        invin invout;
run;
```

More efficient

```
data oct90;
    infile file-spec;
    input @1 jday @13 sales @42 invout;
run;
```

Read selection fields first

Acceptable

```
data temp;
    infile file-spec;
    input month [and lots more fields];
run;
data sales.aug90;
    set temp;
    if month eq 8;
run;
```

Better

```
data sales.aug90;
    infile file-spec;
    input month [and lots more fields];
    if month eq 8;
run;
```

Best

```
data sales.aug90;
    infile file-spec;
    input month @;
    if month eq 8 then input [lots more fields];
    else do;
        input; /* Release input line */
        return; /* Avoid output to dataset */
    end;
run;
```

Here, the Better method avoids an entire dataset as compared to Acceptable. Best does that and also avoids reading fields which are only needed if the record in question is one that is wanted (when month is equal to 8).

Store data in SAS datasets

Appears to be 2 to 5 times faster getting data from a SAS dataset than from a flat file. Actual improvements will depend on relative CPU and I/O performance on your computer. In any case, using this principle has no disadvantages except that you may need to maintain both the SAS dataset and the external form of the data, thus using more disk or tape storage.

Keep summaries of large SAS datasets

Summaries are usually only a fraction the size of the original dataset, so doesn't add much storage to do this, and avoids much computation.

Store only necessary variables

Acceptable

```
data good.stuff;
    infile filespec;
    input x y;
    x2 = x*x;
    y2 = y*y;
    xy = x*y;
run;
```

More efficient

```
data good.stuff;
    infile filespec;
    input x y;
run;
```

(Variables x2, y2, and xy can easily be calculated whenever they are needed.)

Process only variables needed

Acceptable

```
data stocks.gainers;
  set stocks.oct;
  keep tot numshrs company;
  if net gt 0 then
    ....
run;
```

More efficient

```
data stocks.gainers;
  set stocks.oct(keep= tot numshrs company
  net);
  drop net;
  if net gt 0 then ....
run;
```

Use of the dataset option `KEEP=` is very effective if there are many variables in `stocks.gainers`, since it reduces the work SAS must do to bring in extraneous variables from the dataset. It is also good programming practice because the dataset fully documents what variables are available, and can therefore avoid unpleasant surprises when alterations are made. I use `KEEP=` very extensively in all but throw-away code, both on input and output datasets.

Create all data subsets at one time

Acceptable

```
data dog;
  set master;
  if beast eq 'DOG';
run;
data cat;
  set master;
  if beast eq 'CAT';
run;
[etc.]
```

Much more efficient

```
data dog cat mouse ...;
  set master;
  select(beast);
  when('DOG') output dog;
  when('CAT') output cat;
  ....
end;
run;
```

Consider "parallel" dataset for repetitive information

Example: a master file has data by ZIP code. Another file has county data. Rather than merging the two files, *keep them separate!* Any time both are needed, a "merge master county; by ctyfips;" may be used rather than a "set bigfile;".

Edit character values with functions

SAS has an extensive set of character editing functions. Programmers used to other languages may not consider the full possibilities of these, and may use only a limited subset. Examples:

Suppose you want to check that variable `chars` has only the characters `0123456789+-`. Then do this:

```
if verify(chars,'0123456789+-') then [handling for wrong characters]
```

Suppose variable `fullname` has a person's complete name, which may be two, three, four, or more words long. To get the last name:

```
lastname =
  reverse(scan(reverse(fullname),1));
```

A telephone number may be written using `-`, `/`, spaces, or even commas as separators. To eliminate all separators preparatory to writing in a standard form:

```
telnum = compress(telnum,'-+ / ,');
```

Use _TEMPORARY_ Arrays

The `_TEMPORARY_` arrays introduced in version 6.06 are much more efficient in both storage and CPU time. Storage: about 30 bytes less storage *per array element*. CPU: with `_TEMPORARY_`, the reference is direct to the data:

array reference → data

For an array of variables:

array reference → table of pointers →
variable name → data

Use ABORT in Production Work

SAS is very convenient for researchers for many reasons, among which is the fact that it will "trudge on" even if data errors appear. However, in production work, you should check for abnormal conditions, post messages to the PRINT file (not LOG), and abort when abnormal conditions occur. Example: a transaction file contains inventory entries consisting of variables prodcod (product code) and quantity. Since a (fallible) human being entered this data, the following checks should be made:

- Is the prodcod valid? (Check against a product code dictionary.)
- Is the quantity a positive integer?
- Is the quantity "reasonable" considering the prodcod? (This would require a quantity bounds variable in the product code dictionary.)

Read "SAS Programming Tips", but be skeptical!

"SAS Programming Tips" has lots of good ideas, but not all work out very well in practice. Some of the comments below relate to experience with SAS 6.06, and may not be as relevant for 6.07 and 6.08 since those versions did correct certain performance problems of features introduced in version 6.06.

Tip 4.12: Create indexes when appropriate.

A SUGI paper found "appropriate" is rare. The Tip has a good list of the conditions under which use of an index may be appropriate.

Tip 5.21: Set the lower bound of arrays to 0.

For an array of variables, this is very confusing, unless you name your variables x0, x1, x2, ... rather than x1, x2, x3, ... For TEMPORARY arrays, this can be perfectly reasonable. However, I still prefer indexing that makes sense in terms of what you are trying to represent — sometimes that requires a base of 0, sometimes 1, sometimes something else. Actually, the SAS compiler should

be capable of handling this efficiently as all other compilers I know of do!

Tip 5.23: Use the stored program facility.

SUGI paper says time savings small, and it is very confusing for large programs, since *only* datasteps are compile-stored, and each data step must be stored separately.

Tip 7.8: Compress large SAS data sets.

SUGI paper found time increases *several times* when operating on compressed data sets.

Tip 7.10: Store numbers as 1-byte character values.

Saves disk storage. For one-digit numbers, the character representation is 1 byte rather than 2 or 3. The tip also shows how to store numbers up to 255 in 1 byte character.

But this is an ugly practice, and should be avoided unless there is a very good reason to use it.

Author information

David Sommer is the president of Optimal Systems, a consulting company specializing in SAS applications.

Optimal Systems, Inc.
10338 Colorado Road
Bloomington, MN 55438
voice: 612-831-5724
fax: 612-831-2765

Trademarks

SAS is a registered trademark of SAS Institute, Inc. in the USA and other countries. IBM and OS/2 are registered trademarks of the IBM Corporation in the USA and other countries. Gateway 2000 is a registered trademark of the Gateway 2000 corporation. ® indicates USA registration.