Introduction

One of the most common data management tasks performed with SAS software is extracting a subset of observations from a larger file. The subset may be selected on some logical criterion, or may be a group of contiguous cases or a random sample of the original observations. Such subsetting operations are probably been discussed in one form or another in many past SUGI papers. This presentation will review the three important subsetting approaches to coding them in the SAS system.

We will confine our discussion to SAS Release 6.06 and Release 6.07. While most of the SAS pseudo-code presented here will work in earlier versions, some will not. Our examples and benchmarks were run on an IBM 3090/170J under MVS. Test master data sets were either raw data sets with fixed length records or simple SAS data sets. We made no attempt to extend our benchmarks to include SAS transport or compressed data sets or SAS views. While we expect that any improvements in efficiency realized by a coding technique presented here would generalize across hardware platforms, types of SAS data sets and versions of the SAS system, "your mileage may vary," and we urge you to compare approaches before applying them to your favorite 500 megabyte file!

Selecting a Subset Using a Logical Criterion

Perhaps the most common form of subset selection is that based on some logical criterion applied to the values of one or more variables in a master data set. If the logical criterion is met, the case is output to the new data set or used in a PROC. Otherwise it is excluded from use. When subsetting a raw data set, a DATA step using the subsetting IF is the only option available because the WHERE statement only acts on SAS data sets or views and filters out observations prior to their placement in the PDV. When subsetting a SAS data set not for use with a single PROC, employing the WHERE statement or WHERE data set option is the clear choice over preceding the PROC with a DATA step that subsets the data because using WHERE passes the data set only once. Dilorio and Dalberth (1991) found that use of the WHERE statement with a PROC consumed 50% to 75% less CPU time than coupling a PROC with a data step employing either a subsetting IF or WHERE statement. They also showed that this increased efficiency held across a range of subset sizes and was independent of whether the data set was indexed on the selection variable. While their tests were performed for Release 6.06 on a VAX, it seems safe to conclude that this finding should readily generalize across platforms.

If, however, one wishes to create a new SAS data set from a master SAS file, the situation becomes a bit murkier. With the advent of SAS Release 6.06 and 6.07, there are three basic methods for coding this type of selection process. The first and, prior to Release 6.06 on a VAX, it seems safe to conclude that this finding should readily generalize across platforms.

The subsetting IF with a WHERE statement or WHERE option on a SET statement, as

```
DATA new;
SET old (WHERE=logical_expression);
```

Logical expressions in WHERE statements may not contain most SAS functions or variables created in the DATA step. This method may only be used on existing values of variables or values that can be calculated in the WHERE statement with the basic arithmetic operators (+, -, /, *). The third possible method is to employ PROC SQL to create a new table using a WHERE clause to subset rows of the original data set as shown in the following.

```
PROC SQL;
CREATE TABLE new AS
SELECT *
FROM old
WHERE (logical_expression);
```

To complicate matters further, the master SAS data set may or may not be indexed on variables involved in the logical expressions used in WHERE statements.

How does one go about choosing between these alternatives? We explored this question empirically by constructing a test data set and then retrieving subsets of varying sizes from it using the three methods outlined above. The observations in the data set had a total length of 100 and were composed of 4 five character variables, 1 twenty character variable and 15 numeric variables of length 4. The test data set contained 100,000 observations and was approximately 10 megabytes in size. We used the RANUNI function to generate four binary selection variables that could be used to retrieve approximately 5, 20, 50 and 80 percent of the observations in each file and be uniformly distributed across the observations in a file. We only considered simple index variables that were uniformly distributed across a data set.

The CPU time reported by the FULLSTIMER option for each DATA step or PROC executed is presented in Figures 1 and 2 for retrievals using Release 6.06 and the beta test of Release 6.07. The first thing to note from these graphs is that 6.07 is significantly faster for all retrievals. CPU time for 6.07 range from about 60 to 67 percent of comparable 6.06 times. Second, use of indexed variables had mixed results. For Release 6.06 use of indexed variables greatly improved efficiency for subsets of 5 and 20 percent of total observations. This was not the case for 6.07, because the SAS system's index utilization algorithm chose to read the data sequentially rather than use the index variables. Also the WHERE statement proved slightly more efficient than PROC SQL. Third, the subsetting IF was clearly superior to all other techniques for retrieving larger subsets of the data.

We offer three conclusions based on these results. First, PROC SQL is not the preferred mechanism for just taking subsets of SAS data sets. This does not mean, however, that it is not useful for more complex retrievals or retrievals that also involve...
merges. Second, taking small subsets with DATA steps using WHERE statements on indexed variables may be more efficient than constructing comparable subsets using IF statements. However, one must understand the SAS system's criteria for using indexes with WHERE statements and be willing to incur the cost of creating and maintaining appropriate indices (Beatrous and Armstrong, 1991, Painter, 1991). If the master SAS data set is updated often or if the subsetting criteria are dynamic, the overhead of maintaining indexes may far outweigh any savings on subsetting tasks. Third, for smaller SAS data sets and for high percentage subsets of larger data sets, it is still hard to beat the venerable subsetting IF.

While using the subsetting IF is still the most efficient method for extracting a subset based on a logical criterion in many situations, there are methods of using it that are more CPU efficient than others and take little extra coding effort (DeFoor, 1991; Dilorio, 1984; and Howard, 1991). If the master file is a raw data set, the following technique will significantly improve efficiency.

```
DATA new;
FILENAME in 'filespec';
INFILE in;
READ criterion variable(s) and hold pointer */
INPUT check1 check2 @ ;
IF (logical_expression) THEN DO;
  INPUT v1 v2 v3 ;
  OUTPUT;
END;
ELSE RETURN;
```

Note that only the variables involved in the logical expression are read in the first INPUT statement and the remaining variables are read only if the logical expression is true. We compared these two approaches by retrieving 50,000 observations from a raw data form of our test data set. The degree of improvement is apparent from Figure 3. Note that the DATA step that read and tested the selection variable before reading the entire record (IF check) ran in roughly one third less time than the DATA step that read the entire record.

If the input file is a SAS data set, placing the SET statement inside a DO loop can result in some improvement. Thus code similar to the following can be employed.

```
DATA new;
DO UNTIL (eof) ;
  /* Use END= to set end of file flag */
  SET old END=eol;
  IF (logical_expression) THEN OUTPUT ;
  END;
```

Further efficiency can be realized if one also needs a subset of the variables. In this case add a KEEP option to the SET
statement to reduce the size of the program data vector. Remember to include logical expression variables in the KEEP list.

```
DATA new;
  DO UNTIL (eof);
    SET old (KEEP= check1 check2 v1 v2 v3 ... )
    END=eof;
    IF (logical_expression) THEN OUTPUT;
  END;
```

Figure 4 presents a comparison of these three approaches for selecting 50,000 observations from our 100,000 observation SAS file in 6.06 and 6.07. Using the looping code ( hoop) results in only marginal savings in 6.06, but almost a 10 percent improvement in 6.07. Adding a KEEP statement that kept nine of twenty variables significantly reduced execution time in both versions.

### Selecting Contiguous Subsets Using information on Order

In some situations one has information about the ordering of the observations of a data set that can improve the efficiency of a subsetting IF. Sometimes a programmer knows the ordering of cases on a variable that is the primary sorting field for the data but does not know the precise record numbers to retrieve. In this case using a STOP statement will avoid processing unnecessary records. The STOP may be implemented with IF statements or using SELECT and WHEN statements for both raw and SAS data sets. We present the IF implementation here.

```
DATA new;
  IF only KEEP Coding Method
  LOOP KEEP
  ELSE IF KEEP Coding Method
  STOP Coding Method
```

The increased efficiency of using a STOP with IF statements and using FIRSTOBS and OBS compared to a simple subsetting IF can be seen in Figure 5. We retrieved the middle 50,000 cases (25,001 to 75,000) from our test data set using a sequential case identification variable. For both releases of SAS the IF with a STOP code was about fifteen percent faster than the subsetting IF. The use of FIRSTOBS and OBS on the SET statement resulted in roughly 33 percent improvement.
Selecting a Random Sample

Many academic research and marketing questions may be answered as well and more economically by analyzing a random sample of observations from a large data set rather than the entire data set. In addition, some researchers wish to develop a statistical model on a randomly chosen subset of data and then test the fit of that model on the remaining cases. Both of these situations require selecting a random sample of a master data set. Generally these samples are of two types -- simple random and stratified random. A simple random sample is one in which the selection algorithm gives each observation in the data set an equal probability of being included in the sample. Thus, if the data set contains observations from different categories of a categorical variable, a simple random sample will usually include cases in those categories in about the same proportion as they occurred in the original data set. A stratified sample, on the other hand, allows the researcher to decide how many observations from each category to include in the sample data set. Stratified samples are used often in situations where one or more categories are rare and need to be "oversampled".

There are several general issues to consider when evaluating SAS code for sampling. The first is whether the code will yield a data set with the exact number of sample observations desired or merely a number close to the desired sample size. The second is whether or not the algorithm employed samples with or without replacement of each observation back into the pool of observations being sampled. Most statistical procedures assume sampling without replacement, but there are some situations (e.g., constructing bootstrap estimates) where sampling with replacement is necessary. Finally, some methods can be applied to any form of data set to be sampled (e.g., raw or SAS data set, SAS transport data set, compressed SAS data set or SAS view), while some cannot.

Simple Random Sampling

We will start with methods for selecting simple random samples. A search of SAS Institute documentation and the archives of SAS-L turned up many possible solutions. We present six general methods.

Method 1 -- Sort on Random Keys


```sas
DATA keys;
  * Or INFILE and INPUT *;
  SET master;
  * Assign random number to each obs. */
  ranumber = RANUNI(seed); 
  * sort by random number */
  PROC SORT DATA = keys;
  BY ranumber;
  * select first nn observations */
  DATA sample;
  * nn is desired sample size */
  SET keys (OBS=nn);

This method may be used to select an exact size sample from either a raw or SAS file. It guarantees sampling without replacement. In the first DATA step the file is read and a random number assigned to each observation. Using this random number, PROC SORT arranges the observations in a random order. Finally, in the third DATA step, the appropriate number of observations are stripped off the front of the reordered SAS data set and placed in the sample data set.
```

Method 2 -- Approximate Sample Size

This method is shown in the Guide, p. 227 and in SAS Language and Procedures Usage 2, Version 6, First Edition, p. 232. While this method seldom yields an exact size sample, it does guarantee nonreplacement, works with raw or SAS files and is intuitively appealing.

```sas
DATA sample;
  * Or INFILE and INPUT */
  SET master;
  * pp is sample proportion. */
  IF RANUNI(seed) < pp;

In the single DATA step the RANUNI function returns a random number in the interval 0 to 1 for each observation. If this random number is less than the specified sampling proportion, then the observation is included in the sample data set.
```

Method 3 -- Exact With Replacement, Random Access

This approach is found in the Guide, p. 235 and in Usage 2, p. 237. It guarantees an exact sample size and sampling with replacement. Since it uses direct access retrieval, it may be used only with simple SAS data sets. We present a revision of SAS Institute's code which dramatically improves performance by sorting the pointers prior to retrieving the sample observations.

```sas
DATA pointers;
  LENGTH obsnum 4;
  * nn is sample size */
  DO i=1 TO nn
    * Create random observation pointer */
    obsnum = INT (RANUNI(seed) • totalobs) + 1;
    OUTPUT;
  END;
  STOP;
  * Sort pointers for efficient retrieval */
  PROC SORT;
  BY obsnum;
  * Get sample using pointers */
  DATA sample;
  SET pointers;
  SET master POINT = obsnum;
```

This code generates a random pointer (obsnum) for each observation. If the number of observations in the master data set is known, it will retrieve the sample from the master data set.

Method 4 -- Out of the Hat, Sequential

This method simulates picking numbers out of a hat and was originally detailed in the Guide, p. 229. We present a modification presented on SAS-L by Nelson Pardee of Syracuse University that yields an exact sample size without replacement. While it requires that the number of observations in the master file be known, it will work for all types of data sets.

```sas
DATA sample;
  * nn is sample size */
  * mm is number of obs. */
  RETAIN sampsize nn totalobs mm;
  obsleft = totalobs;
  * Loop to execute SET */
```

224
DO i = 1 TO totalobs ;
* Or INFILE and INPUT */
SET master ;
IF RANUNI(seed) < sampsize/obsleft THEN DO;
* If observation selected */
OUTPUT ;
sampsize = sampsize - 1 ;
END ;
* Decrement population size */
obsleft = obsleft - 1 ;
END ;

In this technique, the RETAIN statement initializes the number of observations needed to complete the sample (sampsize) and the number of observations in the master data set (totalobs). Both must be known in advance. The RANUNI function generates a random number and if it is less than the current selection probability (sampsize/obsleft), the observation is output into the sample data set and the number needed to complete the sample is decreased by 1. The number of observations left to read (obsleft) is always decremented.

Method 5 -- Out of the Hat, Random Access

This method, originally outlined in the Guide, p. 228 and Usage 2, p. 235, also simulates drawing cases out of a hat. It produces an exact sample size without replacement. The master file size need not be known in advance. Because it uses direct access reads, it cannot be used for all types of data sets. Note that the use of the NOBS option on the SET statement could be incorporated into Method 4 as long as the input file is not a raw data set or a SAS view. This adaptation by Nelson Pardee greatly improves the efficiency of the original code.

DATA sample ;
* nn is sample size */
RETAIN sampsize nn totalobs ;
IF _N_ = 1 THEN obsleft = totalobs ;
* loop to insure SET execution */
DO i = 1 TO totalobs ;
IF RANUNI(seed) < sampsize/obsleft THEN DO;
SET master POINT = i NOBS=totalobs ;
OUTPUT ;
sampsize = sampsize - 1 ;
END ;
* Decrement population size */
obsleft = obsleft - 1 ;
IF obsleft = 0 OR sampsize = 0 THEN STOP ;
END ;

Figure 6 compares the CPU times consumed by using the above methods for selecting a twenty percent sample of our test data set. Method 1 uses an inordinate amount of CPU time because it has to pass the data once and sort it once. It should be avoided. Method 3 is quite efficient, and is the only one that we are aware of that gives exact size samples with replacement. Of the other three methods, two are the most general and may be applied to raw data files as well as SAS views and compressed data sets. If an approximate sample size will suffice, Method 2 is both quick and easy to understand. However, if an exact sample size is necessary, Method 4 accomplishes this efficiently. The less general Method 5 gets the job done fastest.

Stratified Random Sampling

As we noted earlier, a stratified sample selects observations from each level (i.e. stratum) of some categorical variable in the data set. In cases where such stratification is needed to lower the variance of statistical estimates or to oversample certain strata, the following techniques may be used.

Method 1 -- Equal-sized Samples, Two Pass

This method is shown in the Guide, p. 231 and in Usage 2, p. 240. It may be used to select exact sized strata from either a raw file or existing SAS data set and samples without replacement.

DATA sample ;
* Count observations in each category */
PROC FREQ DATA = master ;
TABLES category / OUT = bycount NOPRINT ;
* assumes data already sorted by BYGROUP */
DATA sample ;
RETAIN stratsiz ;
MERGE master bycount (RENAME=(COUNT=catleft)) ;
BY category ;
* nn is stratum sample size */
IF FIRST.category THEN stratsiz = nn ;
DO i = 1 to nn ;
* select obs */
IF RANUNI(seed) < stratsiz/catleft THEN DO ;
OUTPUT ;
stratsiz = stratsiz - 1 ;
END ;
* decrease number left in stratum */
catleft = catleft - 1 ;
END ;

The code above assumes that the data is already sorted by the stratification variable (category). The PROC FREQ creates a variable (catleft) that contains the number of observations in each BY group. In the DATA step the PROC FREQ output data set is merged with the original data set (master) on the stratification variable. For each new BY group (FIRST.category) the stratum sample size is initialized and then, using the "out of the hat" algorithm, observations are selected with probability of stratsiz/catleft.
Method 2 Equal-Sized Sample from Strata, One Pass

This method is shown only in the Guide, p. 232. It also, selects exact sized strata from a SAS data set and it guarantees sampling without replacement. It may only be used with simple SAS data sets.

/* Assumes data already sorted by BYGROUP */
DATA sample ;
RETAIN stratsiz obsleft;
/* Interleave data with itself */
SET master (in=in1) master ;
BY category ;
/* Count cases in stratum */
IF in1 THEN DO ;
catsize = catsize - 1 ;
END ;
catsize = catsize - 1 ;
END ;
/* Select cases from second group */
ELSE DO ;
IF RANUNI(seed) < stratsiz/catsize THEN DO ;
stratsiz = stratsiz - 1 ;
END ;
catsize = catsize - 1 ;
END ;

Assuming that the data is already sorted by the stratification variable (category), the single DATA step first interleaves the original data set with itself. This replicates a second set of the observations of each BY group immediately after the first set. The counter in the first DO group sums the number of observations in a stratum using the first set. After the first set of observations have been read, the sample is selected from the duplicate set by a stratum using the first set. After the second DO group using the "out of the hat" algorithm.

Method 3 -- Different Sized Samples From Each Strata

This method is shown in the Guide, p. 233 and in Usage 2, p. 241 and is designed for selecting a different number of observations from each category of a stratification variable with five categories. Of the two equal stratum sampling methods, Method 2 proved to be about 15 percent faster than Method 1. In general then, we recommend Method 2. However, if the master file is very large, interleaving the data set may exceed available disk space, and then Method 1 may be the only practical alternative.

Keeping the Subset

If the SAS data set containing a subset will be large, you need to pay some attention to storing it as efficiently as possible. We offer several basic suggestions. First, think about what information you really need and keep only what you need. Using KEEP or DROP data set options on a SET statement insures that you save only the variables that you will be using. Second, store categorical information that happens to be numerically coded in a one or two-byte character variable instead of an eight-byte numeric variable. If a variable is never going to be used in arithmetic operations, why incur the cost of storing it as numeric? Third, store integer valued variables using lengths of four or less. The amount of disk space needed may be reduced significantly without sacrificing accuracy. Large integers with up to seven significant digits can be accurately stored in half the space the SAS system uses by default. Remember, however, that one drawback to using the LENGTH statement is the potential damage done to numbers by truncation. Numeric variables whose values need to be exact for comparisons should not have their lengths reduced from the default.

None of these methods of minimizing storage space are unique to SAS, nor are they new. One new hint that we can offer now is to use Release 6.07 rather than 6.06. The newest version of SAS is more efficient than its immediate predecessor in the way that it creates and stores SAS data sets. A 6.07 data set occupies approximately 4 percent less disk space and can be created using 20-25 percent less CPU time. An additional technique for minimizing storage space available in SAS 6.06 and 6.07 is the COMPRESS option. One advantage of this method is its convenience. A programmer need only specify OPTIONS COMPRESS=YES; at the beginning of a SAS program to reap its benefits throughout the program.

What are these benefits likely to be? To investigate this question we constructed several test data sets. Tall data sets contained 100,000 observations; short data sets, only 10,000. Thin datasets had 20 variables -- 10 one or two-digit integer variables, 1 six-digit integer, 4 decimal variables, 4 five-byte character variables, and 1 twenty-byte character variable. Wide datasets contained 90 one or two-digit integers, plus all the other variables in the thin data set.

The disk space savings realized using compression were about the same in versions 6.06 and 6.07. We present the results for version 6.07 in Figure 7. Using just compression reduces the size of these datasets by 20 to 47 percent. The amount of reduction depends mostly on the number of variables in the dataset. Wider datasets provide more room for compression than do their thinner counterparts. When the data set is shortened by giving the 90 integer variables a length of 4 using a LENGTH statement but not compressing, we see disk space savings of 37...
to 65 percent. Combining a LENGTH statement with the COMPRESS option actually worsens disk space performance somewhat, most notably in the case of the thin datasets.

While it saves disk space, compression incurs significant CPU costs both when the data set is created and when the data set is used in subsequent DATA or PROC steps. Figure 8 shows the effect of compression on CPU usage for various DATA and PROC steps. When creating a data set, compression uses 80 to 100 percent more CPU time than the standard storage method. Using a compressed data set may require up to 600 percent more CPU time than a comparable uncompressed data set. A compressed, shortened dataset performs somewhat worse than an uncompressed dataset, but performs significantly better than a compressed, unshortened dataset. We recommend, therefore, using LENGTH statements wisely whenever possible and avoiding the SAS system's compression feature unless disk space is at a premium and CPU time abundant.

References


Acknowledgments

We are grateful to members of the Research Triangle SAS Users Group who listened to and offered helpful comments on a previous presentation of this material at their March, 1992 meeting. We would also like to thank Frank Dilorio for comments on an earlier draft of this paper and the SAS-L UICSTAT archive at the University of Illinois at Chicago for random sampling code examples.

SAS® is a trademark of SAS Institute, Inc., Cary, NC, USA.

1. SAS-L is an international discussion about using the SAS system that is accessible via both INTERNET and BITNET electronic mail.