Unix Large File Processing Secrets

Karsten Self
PM Squared Inc., San Francisco, CA

ABSTRACT

To most SAS programmers working in the UNIX environment, large file processing is synonymous with the historical maximum file size limit of 2 gigabytes. Though this is a significant concern, there are additional dimensions to managing and working with large quantities of data. Large file processing is any job large enough that you think twice before typing *submit*. The term is applied here to any situation where a task is constrained by physical resources, including: system file size limits, system storage or workspace limits, total processing time, CPU availability, and process contention with other users.

The weapons which can be applied to these are programmer time and skill. Ultimately, pushing the limits of the envelope is a question of balancing these factors. This paper identifies several techniques for large file processing on Unix, including:

- Recent SAS® and OS large file support enhancements.
- Efficiency techniques.
- Sorting data.
- Working to and from tape
- Advanced data coding methods

LARGE FILE SUPPORT

SAS and OS Enhancements

The Unix maximum file size limit exists because of the limited file addressing space allowed by a 32 bit integer. Methods for overcoming this limit are described in Tom Truscott's 1996 SUGI paper "Open Systems Solutions to Large File Requirements".

Providing system large file support requires changes to both the OS and SAS – if your current environment does not provide large dataset support, an upgrade to your operating system, your version of SAS, or both, will be required. Currently, support is native to 64 bit environments,[1] and through the **partitioned dataset** method in SAS 6.12.

If these methods are available at your site, by all means use them – through them you overcome one of the largest and most frustrating obstacles to large file processing. This paper does not discuss these developments.

Users lacking one of these extensions are left to tried-and-true techniques for working around the limit.

WORK SMARTER

Smart Human Tricks #1

This is a grab-bag of techniques learned, developed, or acquired through the years. An excellent starting point is the SAS Institute's own *SAS Programming Tips: A Guide to Efficient SAS Processing*.

General Programming Strategies

The secret to success is to get as much done as you can while doing as little as possible. This may not be the tip you want to take to the boss from SUGI, but isn't efficiency is all about increasing the product to inputs balance? Unfortunately, most of the world still focuses on measuring *inputs,* and not product. Think about your program and what you need out of it. Figure out, within reason, the best way to accomplish this. Remember that there are usually several equivalent methods available.

After a reasonable enhancement in program efficiency has been achieved, it's generally best to forego further improvements. Remember: efficiency improvements only go so far. Development time is itself a cost.

Eliminate Unnecessary Processing

Be especially wary of code which does nothing, or very little, but requires heavy resources. Notorious suspects are routines which empty or delete datasets, which modify labels on variables or datasets, or which duplicate data structures.

I once saw the following code written to produce an empty dataset with the same field structure as an existing one. The input file ranged from tens to hundreds of thousands of records:

```
data mydata;
    set mydata;
    delete;
    run;
```

This is an example of a lot of work to do nothing – the code reads every input value, only to discard it. A lazier (and preferable) method might be:

```
data mydata;
    set mydata;
    stop;
    run;
```

The STOP statement in SAS halts processing of the current data step, in this case, on the first observation. It could even be moved before the SET statement – output data structure is defined at SAS compile time, not at execution.

Eliminate Unnecessary Data

This applies to eliminating both fields *and* records from processing. You've heard it before, you'll hear it again: DROP, KEEP, and WHERE= can be your best friends. In general, subset or summarize data before applying wholesale processing or sorting it.

[1] Digital Electronic Corporation (DEC) is the only current provider of 64 bit UNIX architecture I am aware of.

Minimize Field Storage Requirements

A strength of SAS is the simplicity of its data structures – character and numeric. On the other hand, this doesn't make for particularly compact datasets. Still, there are alternatives available.

Field Widths: LENGTH or ATTRIB statements can be used to specify variable lengths. Remember that the default numeric field-width is 8. On UNIX platforms, the minimum numeric is three.[2] Character fields default to the width of the first value encountered.

Formats for Standard Values: Where other procedural languages offer enumerated data types, and RDBMSs have joins, SAS has formats. Instead of storing the days of the week as 9 byte character fields, you can use a 3 byte numeric and a format – or a one byte character for greater savings. Additional alternatives are discussed later in this paper. Formats can be used for anything from short lists, as above, to code values and definitions containing tens of thousands of values, limited by available memory.

You should consider substituting a format for any variable whose values derive from a known list. The disadvantage of custom formats is that the format must be defined prior to accessing data in order for the data to be properly interpreted.

Use indexes for subsetting or look-up tables

When a large dataset is used primarily to create subsets or as a lookup reference, indexing will speed processing tremendously. Multiple datasets can be indexed individually and searched as one through an SQL view.

Indexes can be utilized when using larger portions (over 10%) of the data, but generally result in reduced efficiency compared to sequential access to sorted data.

Automate data management with DATASETS

PROC DATASETS is used primarily for deleting unnecessary datasets, a useful practice to include in large file processing. Although this and the alternatives (SQL's **DROP TABLE**, and the undocumented but ever popular PROC DELETE) are useful, there are other tasks it can perform, including:

- Assigning or changing variable labels
- Renaming datasets
- Assigning or changing dataset labels
- Adding or removing formats and informats
- Adding or removing indexes

With the exception of indexing, these tasks are performed on data header information, not the data itself, and are far more efficient than achieving the same result through data step processing. This is overhead well avoided.

[2] Both the minimum field size and representable values vary across operating systems. The minimum numeric width 3 is defined by IEEE standards, and is common to all UNIX platforms. Page 196 of the *UNIX Companion* (Version 6, 1st Edition) is a table of maximum exact and exponential value representations for all numeric variable sizes.

Don't Overlook the Obvious (and Not-So-Obvious) Fundamentals

These are some general tips and principles for working with large datasets outside the programming realm.

Workspace: Ensure your WORK directory is big enough for your needs. Know the location and size of your workspace, negotiate for more space if necessary. The SAS session option WORK specifies the WORK library path. In SAS 6.12, partitioned WORK libraries may be specified.

FULLSTIMER: Full system resource utilization reporting allows you to identify and tune 'hot spots' in your programs. The STIMEFMT= Z simplifies automated log analysis by presenting time statistics uniformly.

Metrics: Measurements of programs performance characteristics provide an understanding of program needs, possible improvement, and confirm successful improvement techniques. Compiling (and reviewing) log statistics is a start. Creating automated tools tracking CPU, memory, and disk utilization is a useful extension, and enables benchmarking.

Plan, test, and time: Know the time and space requirements of the work you are doing. Large scale processing on an overextended system can take hours or days. Scale up to your full runs on test, sample, or subset data, increasing input data by powers of two or ten. Watch for processing rates which degrade non-linearly as data volume increases. Experiment with new methods.

Use /dev/null: UNIX's write-only "bit bucket" can be used as an output location when testing large jobs – so long as you don't need the data back.[3] A libref assigned to /dev/null is treated as a tape device.

Know your algorithms: Computationally intense programs can often be expedited by refining algorithms. Complex functions are generally more CPU intensive than simpler ones – exponentiation is slower than addition. Missing value and divide-by-zero operations can be spectacularly inefficient on some systems and should be captured by program logic.

System load: Plan your job for off-peak hours, generally 10 p.m. to 10 a.m. on interactive systems. Use job control utilities, the UNIX **at**, **cron**, and **nice** commands, to schedule and prioritize jobs. Be considerate. Tell users on a shared system when you will be running particularly large jobs.

Don't surprise your system administrator: She or he is working hard to please a lot of people. Provide advance notice of special requirements, data arrivals, and hardware and software requests.

SAS system tuning: The BUFNO, BUFSIZE, and MEMSIZE system options (or dataset counterparts) can have a significant impact on system performance.

UNIX system tuning: The operating system also presents a number of tunable options effecting processor, memory, and disk operations. You should explore these alternatives with your system administrator.

[3] /dev/null is the null device (or file) in UNIX. Output directed to /dev/null is not written to disk or saved anywhere, it simply disappears. When read, /dev/null returns end-of-file (EOF). It is similar in concept to the _NULL_ reserved dataset name in SAS. Like the Arabic zero, nothing can be incredibly useful.

Know when to say when

You may be asked to perform miracles of processing or analysis with resources inadequate to the task. Though such projects can be technically challenging, foster strong team identity, and create the impression of much sweat and labor, they are often only exercises in bravado or worse. Frequently, reductions in scope or additions to capacity will greatly improve the inputs-to-results ratio. Simple back-of-the-envelope calculations of process rate or resource requirements to availability are usually enough to reveal a serious mismatch.

OUT OF SORTS(pace)

Workarounds and Avoidance Techniques

If a job is going to break for want of space, it will break on a sort.

As one of the most common, and most intensive, processing activities involved in data analysis, sorting plays a crucial role. Methods for increasing the efficiency of a sort, for increasing uses of a particular collation sequence in code, decreasing the resource utilization, or eliminating the need for sorts altogether, are often critical to successful large file processing.

Solution Strategies

Sorting is an intensive activity. Disk I/O, CPU, real time, and memory are the coins of the realm – though you don't always have to pay. The general strategies of sorting large files are:

Find more space. Is there a larger filesystem available, or is it possible to clear more space on the filesystem you are using?

Purchase a more efficient algorithm. There are several efficient sort programs on the market which may be purchased. SYNCSORT, produced by the company of the same name, is popular. A SAS-specific PROC SYNCSORT was recently released; information is available from the vendor. I have no experience or affiliation with these products.

Increase or tune available resources. If processing is possible, but performance is inadequate, tuning the SAS and OS as mentioned above may provide improved performance of your process. Try benchmarks with different settings of SAS options.

Use an alternative collating method. This falls into several categories, described below.

Use an avoidance method. Also effective, described below.

Play the breakup game. Popular in the '70s, this is still common practice among SAS programmers in the UNIX environment.

Constraints of SAS Sorts

There are three primary limitations of sorts in SAS processing.

- SAS sorts require additional space of approximately 2.5 times the size of the input file. If you are constrained to using the same filesystem for data storage and sort space, as is the case for temporary files in UNIX, the sort-space penalty is added to file size.
- A given collation sequence applies to only one variable at a time. If your program requires that MYVAR be sorted for one procedure, and YOURVAR for the next, then two sorts are required.
- Much of SAS analysis relies on collated data. Bypassing this requirement eliminates the need, and resource requirements, of sorting.

Alternative Methods

If sorted output is required, there are several routes to the destination.

PROC SORT

With version 6.09, PROC SORT offers the TAGSORT option. Though this can take much more time, both real and CPU, the space requirement is much reduced.

The graph below compares disk utilization and process time for these operations on a 10,000,000 observation, 1 Gig dataset. A standard PROC SORT runs first, requires 2.4 times the input file size for sort space, and completes in 29 minutes. Tagsort begins about ten minutes later; the first three hours of a 29 hour aborted run are shown. Maximum space requirement is about 1 times input file.

On smaller datasets, TAGSORT typically requires five to ten times more execution time.

PROC SQL

PROC SQL can produce ordered output. It has its own sort and may or may not offer better performance.

SQL's real strength is in a number of other abilities, the most powerful being the ability to define ordered views of data from which sorted data may be obtained. Efficiencies on indexed data may approach the speed of a sort. Simple index storage overhead is typically about 10% of dataset size.

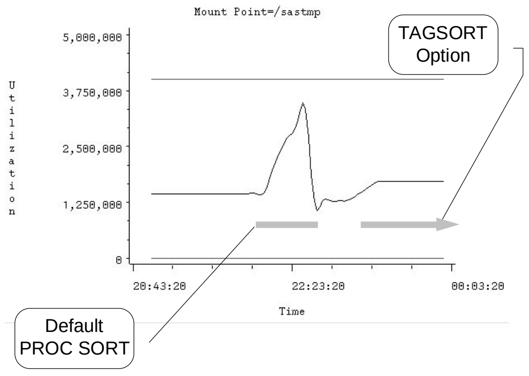
Because multiple indexes on data can be defined, and subsetting of data is especially amenable to SQL, this is a good solution to the problem of accessing small quantities of sorted data from a large dataset.

SET with BY

Data must be indexed, and this may not be particularly efficient – it is *very* inefficient on non-local or networked drives.

This method provides a 'one size fits all' solution when dealing with large numbers of datasets of varying sizes, some of which

Figure 1 -- SORT Disk Utilization



are known or suspected to exceed maximum sortable size. Case-by-case data management is eliminated.

Avoidance

Sometimes the best way to do get something done is to avoid it all together. This may apply either in whole or part.

Are the sorts necessary?

- Look at your program flow-of-control and identify areas where you either do not need to sort or can utilize a sort order several times before changing it.
- PROC SQL can access, process, and summarize unsorted data. Though this may necessitate an internal sort, it may still be more efficient than running a PROC SORT on the entire dataset. The undocumented _METHOD option to SQL may be used to identify SQLs internal processing.[4]

```
   proc sql _method;
```

- Summarization statistics can often be obtained by the use of a CLASS rather than a BY variable. A class statement can utilize unsorted inputs, though the number of distinct values is restricted by available memory.
- Using the NOTSORTED option with a BY statement eliminates the requirement that variables are sorted, though analysis results may differ with data order.
- The SORTEDBY= dataset option can be used to indicate data are already ordered, avoiding unnecessary sorts.
- Do the data come from a system with a different collate sequence? The SORTSEQ option can be used to specify an alternate to the default collate sequence of your system.
- Is the data from an RDBMS? Utilize SQL passthrough to specify the server sort your results.

Is all the data necessary?

- Can fields be dropped from the input data? Use KEEP or DROP statements or dataset options to specify the variables you need for processing.
- Can you subset your dataset prior to sorting it? Use a WHERE clause or dataset option to specify the observations you need for processing.
- Can you consolidate it prior to sorting it? This is another use for CLASS statements in a SUMMARY or MEANS procedure.

Manual Partitions

The last resort is to make little ones out of big ones – divide a large dataset into manageable smaller partitions. There are usually one or more logical orderings for the data, whether by geographic region, time, or other principle characteristics. In particular, there is no reason to build your partitions along your primary sort sequence. If your analysis is based on Social Security Number, but data arrives annually, partition the data by year, and sort each partition by SSN.

Sorted or indexed data can be read by specifying multiple datasets on a SET statement with BY processing:

```
data tape.bigfile;
   * small1 - small2 are sorted or indexed by
   *  sortvar;
   set
      small1
      small2
      small3
      small4
      small5
      ;
   by sortvar;
   run;
```

You can also create a single virtual dataset by combining the files in an SQL view. This allows a single point of access to all the datafiles, for simplified processing. Remember, however, that SAS views are read-only. It is not possible to update the underlying data through the view.

ROLL TAPE

Challenges and opportunities of sequential media

Because tape allows only sequential, and not random, access, arbitrary size limits do not apply. On the other hand, this means that:

- You *must* access data in a tape dataset sequentially – there is no way to shortcut to a particular record or past an arbitrary number of records.
- There is no way to know ahead of time the total number of records in a dataset. Tools such as the SET **NOBS=** option are not available.
- It is not possible to index tape datasets. You can, however, sort the data.
- Tape is *much* slower than disk. Sustained transfer rates are on the order of 0.25 - 1 MB/second.

Despite these limitations, there are times when working with tape is appropriate, in particular, when reading data from tape or when creating an archive to be maintained off-line.

The two most common UNIX tape formats, 4mm and 8mm cassette, allow for about 2 Gig and 5 Gig storage, respectively, without compression, and approximately double the amount compressed. Compression on UNIX is typically specified in the tape device-driver specification. This is discussed later.

As of release 6.09, SAS can read and write both raw and SAS data to tape. SAS datasets are accessed through the TAPE keyword in a libname statement:

```
libname mylib tape ”/dev/rmt/0m”
```

Tape Paradigms

When working with tape, there are four typical task configurations

Tape to multiple disk files: Done when data on tape needs to be subset in order to be stored on disk. If necessary, the partitioned disk files may be combined in a view as described above, in order to be analyzed as a single unit.

Tape subset to disk: Performed when an analytic or processing archive has been created and is maintained off-line. Data meeting specified criteria are extracted from tape and processed from disk.

Tape-to-tape: In some instances, a single level of processing is required of tape data, and the results may be stored to tape.

[4] Described in Paul Kent's 1995 SUGI paper "SQL Joins – The Long and The Short of It", cited in References. Interpretations for values returned by the _METHOD option are provided in the paper.

The most common situation would be simple data conversions, or conversion of raw data to SAS datasets.

Multiple disk files to tape: An alternative when partitioned disk data needs to be placed in an off-line archive. Multiple files may be read in a single data step. If the source data are sorted or indexed, the tape dataset may be written in sort order using BY processing.

The Last Reel – Processing Tips

Tape processing introduces its own challenges. Here are some miscellaneous suggestions and hints.

Compression, density, and device specification: Most tape devices support hardware compression of data. This is typically specified in the device call itself, e.g.:

```
/dev/rmt/Om.
```

Refer to your site documentation for specific options available.

SAS tape disposition: The **TAPECLOSE=** system and **FILECLOSE=** dataset option specifies what to do with the tape drive when processing is completed. Options are REREAD, REWIND, LEAVE, and DISP. The **mt** UNIX command can also be used for positioning, rewinding, or unmounting tapes.

Readiness testing: Because tape is not constantly in a 'ready' mode, your job needs to be either manually or automatically staged to execute when the appropriate tape drive and media are ready to accept input. On most UNIX systems, the **mt** command can be used to query a device for status.

This can be incorporated into SAS code or a macro which tests the return value, **%SYSRC**, of the command, and delays program execution until the drive is available. For example:

```
%sysexec mt -f &Drive status;
%if &sysrc ge 1 %then
    /* failure processing */
%else
    /* success processing */
```

IN THE RAW

Reading, Writing, and Getting Piped

One of the most flexible features of UNIX is the ability of programs to receive or send output to other programs through *stdin*, or standard input, and *stdout*, or standard output. This is known as *piped* output, *piping*, and is accomplished by a *pipe*, '|'. There is also a *stderr*. You might guess at what it means, it will appear later.

Pipes can be useful for simple tasks like paging through screen output so that you can read it:

```
ls -l | more
```

Multiple commands can be piped together, though this sometimes makes for difficult reading:[5]

```
descr=`ps -fp $job | grep $user | \
  sed -n "/.\{24\}\(.*$\)/s//\\1/p"`
```

Pipes can also be used directly by SAS for reading and writing output to the operating system. The SORT disk utilization chart presented earlier is produced from a program which reads input directly from the UNIX **bdf** disk utilization command, via a **FILENAME PIPE:**[6]

```
filename bdf pipe 'bdf';
```

This is read in a data step as with any other infile:

```
data   bdf;
    infile bdf;
    input /* etc. */
```

The fact that this is a pipe is acknowledged by SAS:

```
NOTE: The infile BDF is:
      Pipe command="bdf"

NOTE: 23 records were read from the infile
   BDF.
      The minimum record length was 50.
      The maximum record length was 67.
```

Pipes are most useful in large file processing in allowing direct access to and from compressed data.

Working With Compressed Raw Data

File compression is an attractive built-in utility to UNIX – compression rations of 80-90% are readily attainable. Although SAS cannot access datasets which have been compressed through the operating system, raw data may be piped through a compression or decompression program as it is written to or read from disk.

When reading large quantities of raw data, it is possible to perform decompression 'on the fly' by passing the file through a FILENAME PIPE with the **uncompress**, **zcat**, or **gunzip** commands:

```
filename rawcompr pipe "zcat /mypath/mydata";
```

Piped input or output may include error messages.[7] Typically, you avoid introducing these to your data stream by redirecting *stderr* to a null device, called **output redirection**. Implementation varies according to your UNIX shell and system; you are referred to system documentation for details.

Compressing SAS Data

Although SAS does not support direct access to datasets which have been compressed by UNIX utilities, it is possible to proxy this by creating special support for writing to and reading from a compressed flatfile. Alternately, SAS datasets can be scheduled for compression and decompression according to processing needs.

Accessing compressed data through SAS

This method has the benefit of not requiring storage for the entire decompressed contents of a file by accessing it via pipes.

Writing to the file might be through a macro of the form:

```
%macro writcmpr(
    indata= ,    /* input dataset */
    cmprfile= ,  /* compressed output
                 /* flatfile */
```

[5] For the curious, this expression extracts process description from the **ps** (process information) command, after filtering for the user id, and striping away the first 24 characters. It is used to identify processes which have terminated (in which case, no description exists), within a UNIX script.

[6] FILENAME PIPE is documented in *SAS Companion for UNIX Environments: Language, Version 6, First Edition*, p 176. See References.

[7] This includes the somewhat cryptic "mesg: cannot stat", along with more recognizable error output.

```
        );
        filename writcmpr pipe
          "compress > &cmprfile";
        data _null_;
            file writcmpr;
            set &indata.;
            put a best. +1
              b best. +1
              c best. +1
              x;
        run;
        %mend;
```

In this example, data cannot be appended to the file once it is written.[8] The program is specific to the structure of the file being compressed, though a utility such as the %FLATFILE macro could be extended to write generalized compressed output.[9]

Read access could be accomplished through a data step view for direct access to compressed data. This method may be used to treat any compressed flatfile as a SAS dataset when reading to a procedure or DATA step.

The drawbacks of this method are several:

- It is code intensive. Each dataset requires creation and extraction programs. The natural elegance of SAS not needing specific access programs for data is violated.
- Appending new records to compressed data is non-trivial.
- A significant processing overhead is incurred for compressing, decompressing, and converting data from raw to SAS form.
- Features available for use with SAS datasets (POINT= processing, indexing, etc.) are not available.

Nonetheless, if direct access to large quantities of data from disk are required, this is one solution.

Using compressed SAS data

A related method was employed at a site with limited disk storage and large quantities of data arriving on tape.

SAS datasets were compressed or uncompressed during processing as a macro loop cycled through a sequence of datasets. Input data were uncompressed in background one cycle prior to use, minimizing process interruption. Output was compressed in background immediately on completion. The nature of the task – repetitive cycling, large quantities of data, and several years of application, justified the development effort.

[8] Redirection ('>') is specified as overwrite only in the macro, attempting to add data to the file would replace it. More critically, simply appending output to an existing compressed file will corrupt the file.

Conceptually, the macro could be extended to move the original compressed data file to a temporary location, read from this file to the new output through FILENAME PIPEs, and append new data to the output file. However, my experiments with multiple FILENAME PIPEs in a single DATA step inevitably aborts the SAS session.

[9] The %FLATFILE macro produces text output from a SAS dataset without requiring specialized programming. It was described by M. Michelle Buchecker of the SAS Institute in the paper '%FLATFILE and Make Your Life Easier', SAS Institute, Inc., *Proceedings of the Twenty-First Annual SAS Users Group International Conference,* Cary, NC: SAS Institute Inc., 1996 pp178 - 180. It is available via anonymous ftp from ftp://ftp.sas.com/pub/sugi21 as "flatfile.sas".

Creating this system required:

- A process scheduling system. Sequentially numbered datasets were processed in a macro loop.
- A data uncompression interface. A macro launching the **gunzip** command, as a background process, through %SYSEXEC was used. Files are given temporary names during uncompression.
- File/data existence testing. Macros to test the existence of uncompressed files and datasets were required. These drove sleep/proceed loops in the main code. If data were not available when needed, the main program would 'sleep' up to a specified number of intervals until delivery (or operator intervention).

ADVANCED VALUE CODING

Smart Human Tricks #2

It is possible to improve greatly on the efficiency of SAS data storage, if you are willing to tolerate some fairly esoteric programming and data representation. The following are smart human tricks – though not necessarily good programming practice. You should think twice before introducing the complexity involved to your code in today's era of cheap storage.

Bitfields

The most common example of this is a bitfield. The default numeric variable contains 8 bytes, or 64 bits. The shortest UNIX numeric variable is 3 bytes, which is still 24 bits. A Boolean value requires only one bit for representation. In a bitfield, each bit of a variable is used to represent a single Boolean value.

It is possible to utilize the full informational capacity of a character field of length 1 to 25 bytes (or 200 bits – the maximum length of a SAS character string). For ease of interpretation, an SQL view of the bitfield can offer a more conventional view of the data. A format can be assigned to these values for straightforward interpretation.

This is illustrated in **Figure 2,** the equivalent of eight character fields of five bytes each are represented with a single byte of disk storage.

CharHex encoded character data

Character hexadecimal, or CharHex coding is a related technique. It encodes integer values on character fields based on the underlying hexadecimal code. As with bitfields, it makes available the full informational capacity of the data space.

SAS formats may be used to create presentable a data field by coding directly against the hexadecimal value of the field data values from CharHex fields.

This is a possible solution when presented with stable data structures consisting largely of categorical field with relatively small code schemas, and when storage is at a premium.

CharHex derives its storage efficiencies from the fact that integers can be far more efficiently represented in hex space than as floating decimals, and from the ability to define character variables as small as a single byte.

The number of representable values for fields of one to eight bytes (limits imposed by formats used in the method) are given in **Table 1**

The utility of this method is in smaller fields – it’s been a while since I needed a hundred million million of anything. Many categorical coding schemes can be coded into 256 or fewer values – gender, states of the Union, and so forth. Two to four bytes could handle applications such as medical billing codes to national registries – tens of thousands to billions of values.

Modeling a typical application, CharHex coding results in a dataset about 50% the size of conventional usage, the majority of this savings coming from encoding small schema sets.

Assuming that values are used to specify display formats, implementing this scheme requires:

- A method of generating sequential lists of character hex values. These become the FORMAT internal values.
  The can be done by translating a decimal value to the appropriate hex value through PUT and INPUT functions, as here:

```
x = input(put(y, binary2.),
   $binary2.);
```

  …or by incrementing an existing CharHex value, as here. Note that this macro is sensitive to the size of the storage field.

```
%macro CBitAdd( arg, incr= 1 );

   input(put(
     input(&arg, pib2. ) + &incr,
     binary16.), $binary16.)
   %mend;

data _null_;
   x= ’0001’x;
   x= %CBitAdd( x );
   put x $hex4.;
   run;

0001  0002
NOTE: DATA statement used:
      real time           0:00:00.11
      user cpu time       0:00:00.02
      system cpu time     0:00:00.02
      memory              41 K
```

- A method of translating the semantic value (display value) to the appropriate hex values. This can be done through a format which reverses the display assignment, or through an SQL join to an appropriate lookup table.
- A method of displaying the encoded data in the display format. Though this could be accomplished by defining the display formats on the underlying data, the preferred method is to define an SQL view displaying the interpreted (not raw storage) values. This allows searching or manipulating the data in terms of the display values in a more consistent manner for the user.

Table 1 – Character Hex Coding

Bytes	Values	Conventional SAS Numeric
1	256	N/A
2	65,536	N/A
3	16,777,216	8,192
4	4,294,967,296	2,097,152
5	1,099,511,627,776	536,870,912
6	281,474,976,710,656	137,438,953,472
7	72,057,594,037,927,900	35,184,372,088,832
8	18,446,744,073,709,600,000	9,007,119,254,740,992

Figure 2 -- Bitfield Example

```
proc format;
    value truth 0 = 'false' 1 = 'true' other= 'ERROR';
    run;

data bitty;
    attrib bitfield length= $1 format= $binary. label= 'Bitfield for sas--dog';
    string= '00000000';
    sas    = 1;
    love   = 1;
    gop    = 0;
    taxes  = 1;
    death  = 1;
    lottery= 0;
    cat    = 1;
    dog    = 0;
    array bits{*} sas -- dog;
    do i = 1 to dim( bits );
        if bits{i} then substr( string, i, 1) = '1';
        end;  * do i processing;
    bitfield = input( string, $binary8. );
    keep bitfield;
    run;

proc sql;
    create view bitview as
    select
      input( substr( put(bitfield, $binary8.), 1, 1), 1.) as sas     format= truth.,
      input( substr( put(bitfield, $binary8.), 2, 1), 1.) as love    format= truth.,
      input( substr( put(bitfield, $binary8.), 3, 1), 1.) as gop     format= truth.,
      input( substr( put(bitfield, $binary8.), 4, 1), 1.) as taxes   format= truth.,
      input( substr( put(bitfield, $binary8.), 5, 1), 1.) as death   format= truth.,
      input( substr( put(bitfield, $binary8.), 6, 1), 1.) as lottery format= truth.,
      input( substr( put(bitfield, $binary8.), 7, 1), 1.) as cat     format= truth.,
      input( substr( put(bitfield, $binary8.), 8, 1), 1.) as dog     format= truth.
    from bitty
    ;
    quit;
```

This creates the following output for BITTY and BITVIEW respectively.

.

```
OBS    BITFIELD

  1    11011010
OBS    SAS     LOVE     GOP     TAXES    DEATH    LOTTERY   CAT     DOG

  1   true     true   false      true     true      false   true   false
```

BEEN THERE, DONE THAT

Further Information

The following sources provide additional information on large file processing and general programming techniques, by persons with experience with the issues.

From SAS Institute, Inc., *Proceedings of the Twenty-First Annual SAS[®] Users Group International Conference*, Cary, NC: SAS Institute Inc., 1996.

- **Truscott, Tom, *Open Systems Solutions to Large File Requirements***
  In-depth discussion of UNIX large-file limitations and solutions developed or under development by the SAS Institute.

- **Raithel, Michael A., *Power Techniques for Processing Large Tape Data Sets Using the SAS System in the MVS Environment.***
  Specific to MVS, but strategies apply to any tape processing.
- **Lafler, Kirk Paul, *"Gaining efficiency with SAS software"* pp. 1577 - 1581**
  Discusses standard methods for improving SAS program efficiency.
- **Hardy, Ken, Sally Muller, and Arturo Barrios, "*You Want Me to Move How Many thousand Files from MVS to UNIX?"* pp. 1611 - 1619**
  Nuts and bolts issues involved in wholesale conversion from mainframe to UNIX systems by both direct connection and tape transfer.

The books listed below provide programming and computer-science insight to technical challenges.

- **Aster, Rick *Professional SAS Programming Secrets*, Windcrest/McGraw-Hill, Inc. New York, © 1991.**
  One of the few SAS books written from the perspective of a programmer, with focus on applying classic computer science techniques and algorithms to SAS. Aster provides one of the better examples of bitfield coding in the SAS literature.
- **McConnell, Steve *Code Complete: a practical handbook of software construction*, Microsoft Press, Redmond, WA, © 1993.**
  A wealth of information on all aspects of programming design and style, written for 3GL languages such as C++ and Pascal, but applicable to SAS. Includes useful tips on software tuning.

General UNIX references.

- **Gilly, Daniel et al, *UNIX in a Nutshell: a Desktop Quick Reference for System V & Solaris 2.0*, O'Reilly & Associates, Inc., Sebastopol, CA, © 1992.**
  General reference to UNIX commands and features.
- **Peek, Jerry, Tim O'Reilly, and Mike Loukides, *UNIX Power Tools*, O'Reilly & Associates, Inc., Sebastopol, CA © 1993.**
  A very humanly annotated *how to* guide for non-trivial UNIX tasks.

REFERENCES

Buchecker, Michelle "%FLATFILE and Make Your Life Easier", SAS Institute, Inc., *Proceedings of the Twentieth Annual SAS Users Group International Conference,* Cary, NC: SAS Institute Inc., 1995, p 178.

Kent, Paul "SQL Joins – The Long and The Short of IT", SAS Institute, Inc., *Proceedings of the Twentieth Annual SAS Users Group International Conference,* Cary, NC: SAS Institute Inc., 1995, p 206-215.

SAS Institute Inc., *Companion for UNIX Environments: Language, Version 6, First Edition*, Cary, NC: SAS Institute Inc., 1993

Truscott, Tom "Open Systems Solutions to Large File Requirements", SAS Institute, Inc., *Proceedings of the Twenty-First Annual SAS Users Group International Conference*, Cary, NC: SAS Institute Inc., 1996, pp. 1437-1440.

CONTACT INFORMATION

The author may be contacted at:

Karsten M. Self
PM Squared, Inc.
250 Montgomery St., Suite. 810
San Francisco, CA 94104
(415) 283-2437
kmself@pmsquared.com

NOTICES