SAS® GLOBAL FORUM 2021

Paper 1089-2021

Making SAS Tables Smaller and Faster Without Data Loss

Glen Becker, USAA

ABSTRACT

This session explores how SAS physically stores and retrieves data from SAS data files. It describes SAS data compression and other space-saving techniques. It explores the performance storage implications of SAS data and SQL views, and demonstrates how views can save space and time. It also explores the performance implications of compression and the choice of variable length, including a surprising technique that can make a SAS table smaller by making some variables larger. Lastly, it introduces "Zipper", a SAS macro that zips SAS data, and builds a SAS data view that expands the zipped data on-the-fly.

INTRODUCTION

Analytic demands sometimes seem infinite, but an analyst has limited space and time. An unoptimized SAS application seems to run forever and fill disks. This paper explores the ways in which an analyst can make SAS tables smaller, and process faster than, but without data loss. These techniques apply to all SAS platforms, and range in complexity from simple changes available to beginners to very advanced techniques.

Overview of topics:

- Save space with short numerics
- SAS Compression
- OS-based Compression
- Save space and time with SAS views
- Performance Implications of Short Numerics, including a surprising technique that can make a SAS table smaller by making some variables larger.
- Zip, gzip, and Zipper

A PowerPoint presentation is also available included.

SAS code in this paper uses certain utility macros whose source code is included in the Appendix. These macros have been tested on the Linux and Windows version of SAS.

SAVE SPACE WITH SHORT NUMERICS

All numeric variables are 8-byte double-precision floating-point in memory. When SAS writes them to disk SAS allows them to be truncated to 3-7 bytes when written to disk. (Mainframe allows 2 bytes.)

Truncated numeric values allow the same magnitudes as a full-size 8-byte value, but with a loss of precision. The following table shows the number of significant bits of mantissa available for each length on distributed platforms:

Length	Significant Bits	Max Consecutive Integer	Approx Digits
3	13	8,192	3-4
4	21	2,097,152	6
5	29	536,870,912	8-9
6	37	137,438,953,472	11
7	45	35,184,372,088,832	13-14
8	53	9,007,199,254,740,992	16

Since most of us do not think in terms of significant bits, these formulas provide the largest consecutive integer that can be stored for each length:

Distributed: 2 ** (length*8 – 11)

Mainframe: 2 ** (length*8 – 8)   or 256 ** (length-1)

Nit: the SAS documentation says, "largest integer stored exactly", but this is not true. Some larger integers can be stored exactly but not all. For example, on distributed platforms the limit of a 3-byte value is 8192 (2**13). Slightly lrger values lose one bit of precision, so the odd number 8193 cannot be stored exactly, but the even value 8194 can be.

What does this mean in practice?   Generally small integers can be safely stored as length 3, SAS dates as length 4, and SAS datetime values accurate to one second as length 6.

SAS does NOT warn you about truncation. Be careful!

I was burned by this once, when I defined a value to store the low 4 digits of a sensitive number as length 3.   On Unix or Windows, the largest consecutive integer is 8192. In my testing, all seemed OK.   But much later, I got complaints about corrupted data.   Why? All odd values between 8193 and 9999 were being truncated to the next lower even number, but all even numbers were OK. I had to re-define the column as length 4 and re-load it from source data.

Lastly, please remember that this truncation only occurs when the data is written to disk or passed between views; it does NOT affect calculations within a DATA or SQL step.

Of course, a SAS character variable can be defined as any length, but like numeric variables, SAS does not warn you if you assign a long character value to a variable too short to hold it. But there is one nice exception: The (fairly) new CAT, CATT, CATS, and CATX functions do warn you. Example:

```
Length shorty $8 verbose $16;
<... code that populates verbose, verbosely ...>
shorty = verbose;         /* No warning, just truncates */
shorty = catt(verbose);   /* fails if truncates after truncation */
```

SAS COMPRESSION

SAS provides two built-in compression options: Compress=Yes, which literally means Compress=Char, and Compress=Binary.

COMPRESS=CHAR, AKA COMPRESS=YES

Compress=Char changes 4+ identical bytes to 3. In its simplest form, Compress=Yes is a poor-man's VarChar: it eliminates the penalty of storing the many trailing blanks commonly found at the end of long character variables.

This becomes very important for data extracted from databases. For example, if an Oracle database defines a column a VARCHAR(255), SAS renders it as LENGTH $255. If the column typically has less than 50 characters, then each row of data wastes more than 200 bytes of disk without SAS compression. This problem is so prevalent that many sites make Compress=Yes the default, because the alternative is massive, space-wasting table bloat.

Paradoxically, Compress=Char essentially nullifies the benefits of correctly sizing a character variable. Since 3 or more trailing blanks collapse to exactly 3 bytes, shortening a character variable provides space savings only on values that have two or fewer trailing blanks. If the contents of a character variable have a widely varying numbers of actual characters, then shorting a character variable from 64 bytes to 16 will provide almost no savings if most values have 13 or fewer actual character. But, shortening character variables helps a lot if most values have the same number of characters.

Compress=Char is only marginally helpful with numeric data. Specifically, if small numeric values are stored as length 8, they tend to have many consecutive zero bytes, and Compress=Char helps.

However, the number zero is 8 zero bytes, so many consecutive zero values compress to only 3 bytes. This can be very useful: (see the Appendix for the size macro)

```
/* Table with many zeros as Compress=Yes */
options compress=yes;
data lotsa_zeros;
   length zero1-zero100 8;
   retain zero1-zero100 0;
   do n=1 to 1E6'
      output;
   end;
run;
%size;
Results:   96% compression
Size of WORK.LOTSA_ZEROS with Compress=YES is 31,326,208.
```

COMPRESS=BINARY

Compress=Binary is for sparse or repetitive numeric data. For a very sparse table, which means one whose numeric values are mostly missing values, it provides 70% compression vs. only 34% with Compress=Yes. However, Compress=Binary requires more CPU:

```
/* Table with many nulls as Compress=Yes */
data lotsa_nulls;
   length null1-null100 8;
   retain null1-null100 .;
   do n=1 to 1E6;
      output;
   end;
run;
%size;
Results:   34% compression, 2.23 seconds CPU.
Size of WORK.LOTSA_NULLS with Compress=YES is 531,365,888.
```

```
/* Table with many nulls as Compress=Binary */
options compress=binary;
data lotsa_nulls2
 set lotsa_nulls;
run;
%size;
```

```
Results:   70% compression, 2.98 seconds CPU.
Size of WORK.LOTSA_NULLS2 with Compress=BINARY is 247,857,152.
```

RECOMMENDATION

In general, Compress=Yes, which really means Compress=Char, is best for general use. For applications with sparse data, one gets far better compression by changing nulls to zeros and using Compress=Yes than using Compress=Binary.

If your application has a sparce table, but must distinguish between zero and missing values, you can do a transformation whereby all missing values are stored as zero, but all zero values are stored as the SAS special missing value .Z. You can then create a simple DATA-step view that restores the data. Illustration:

```
/* Crude utility macro to repeat code 100 times, changing ? to number.
   Repeated code that directly names variables is faster
   than accessing the same variables via arrays. */
%macro repeat100(text);
   %local N;
   %do N=1 %to 100;
       %sysfunc(tranwrd(&text,?,&N))
   %end;
%mend repeat100;

/* Table with nulls changed to zeros as Compress=Char */
options compress=char;
data change_nulls_to_zeros;
 set lotsa_nulls;
 %repeat100( /**/ if null? eq . then null? = 0;
             else if null? eq 0 then null? = .Z; )
run;
%size;
Results:   96% compression, 2.08 seconds CPU.
Size of WORK.CHANGE_NULLS_TO_ZEROS with Compress=CHAR is 31,326,208.

/* Create view to restore nulls and zeros as they were. */
data lotsa_nulls_restored /
view=lotsa_nulls_restored;
  set change_nulls_to_zeros;
  /* Test more likely case first for best performance. */
  %repeat100( /**/ if null? eq  0 then null? = .;
              else if null? eq .Z then null? = 0; );
run;
```

```
/* Read view to verify it worked. */
data _null_;
  set lotsa_nulls_restored;
  if null99 ne . then abort;
run;
```

```
Results:   0.59 seconds CPU.
```

FILESYSTEM-BASED COMPRESSION

The Windows NTFS filesystem offers transparent compression. You simply turn on compression for a directory, and Windows compresses all files on that directory as they are written. My experience is that NTFS compression with SAS Compression works better than either alone, probably because their compression algorithms are very different.

This compression may not be available at your site: Transparent compression is not yet available for Linux, and many third-party Windows filesystems do not offer transparent compression, like those on network storage appliances. Ask your systems staff.

To turn on NTFS compression in File manager: Right-click, Properties, Advanced, check "Compress contents to save disk space". You can also use the Windows "Compact" command from a Windows command line.

SAVE SPACE AND TIME WITH SAS VIEWS

A view behaves like a table in that it is read the same way, but the view obtains its data from another source. SAS offers two kinds of views: an SQL view is a SELECT statement that implicitly runs whenever the view is read; the view reader receives the results of the SELECT statement as if they were stored in a table. A SAS Data-step view is a compiled program that runs as the view is read; the view reader receives the data rows as the view writes them.

A simple view can save space by not storing multiple copies of the same data. It can also save time by not writing and re-reading data to/from a disk, because a view can do derivations on-the-fly as its reader reads the data. Illustration:

```
/* Step 0: Create a large SAS dataset that any SAS user can readily
re-create: 100 copies of each row of SASHELP.ZIPCODE */
options compress=yes ls=150;
data ZipCodes;
  set sashelp.ZipCode;    /* Largest physical table shipped with SAS */
  do N=1 to 100;
     output;
  end;
  drop N;
run;
%Size;
```

```
Results: Size of WORK.ZIPCODES with Compress=YES is 657,457,152
```

Now, suppose I want a copy, but with each "Alternate City Name" in Alias_City and Alias_CityN to be a separate row, and I also want the length of the City name. Then, run PROC MEANS on the result:

```
/* Champion:  Create a derivative table */
data ZipCode_City;
  /* Capture logic in a macro var so can easily re-run it. */
  %let ZipCode_City_Statements = %str(
      set ZipCodes;
      label  City = "City name or alias";
      attrib City_name_len length=4
           label="Length of city name or alias";

      link output;   /* Write line for Primary City */

      /* Code note: naive parsing for illustration.
         Finding boundaries with FINDC() would be faster. */
      do N=1 by 1;   /* Nominally infinite loop to parse Alias_city */
         City = scan(Alias_city, N, '|');
         if City = ' ' then leave;
         link output;
      end;

      /* Same for Alias_CityN, which by observation is not a
         duplicate of Alias_City */
      do N=1 by 1;  /* Nominally infinite loop to parse Alias_cityN */
         City = scan(Alias_cityN, N, '|');
         if City = ' ' then leave;
         link output;
      end;
      return;  /* end of this input row */

      /* Link routine to determine length of city name and write
         output row */
      output: City_name_len = length(City);
              output;
              return;   /* to caller */

      /* drop transient var and now-irrelevant input vars */
      drop N Alias_City Alias_CityN  City2;
  );  /* end of %LET statement */

  &ZipCode_City_Statements;
run;
%Size;

proc means data=ZipCode_City;
run;

Results:
Size of WORK.ZIPCODE_CITY with Compress=YES is 1,030,619,136
DATA step  required 8.74 seconds elapsed, 8.39 seconds CPU.
PROC MEANS required 2.98 seconds elapsed, 3.87 seconds CPU.
Total is therefore 11.72 seconds elapsed 12.26 seconds CPU.
```
(CPU time can exceed real time because PROC MEANS is multi-threaded.)

Now, let's do the same thing, but make a SAS data view instead of a SAS data table.

```
/* Challenger:  Same as above, but as a view. */
data vZipCode_City / view=vZipCode_City;
  &ZipCode_City_Statements;
run;

/* This PROC MEANS runs the view code on-the-fly without storing its
results */
proc means data=vZipCode_City;
run;
```

```
Results:
DATA step  required 0.07 seconds elapsed, 0.01 seconds CPU. see below)
PROC MEANS required 6.94 seconds elapsed, 6.32 seconds CPU.
Total is therefore  7.01 seconds elapsed  6.33 seconds CPU.
```

Wow! The total effort required is just over half of the normal method. The reason is that in the naive "Champion" solution, the initial DATA step had to read 627 megabytes (MB) from ZIPCODES, then write 983 MB of data to ZIPCODE_CITY, as well as do all of the parsing logic. PROC MEANS had to read back that 983 MB of data from disk.

The "Challenger" solution's Data step took almost no time because it only compiled a program; it processed no data. The Data step view and PROC MEANS were running at the same time. They read that 627 MB from disk, performed the transformations, and passed each row from the Data step to PROC MEANS via fast memory-to-memory transfer; nothing was written to disk at all.   The "Challenger" also consumed almost no disk space, because a SAS data view is just a small image of a compiled program.

Think about this in terms of disk transfers:

Champion: Disk → DATA step → Disk, then Disk → PROC MEANS.   3 disk passes.
Challenger: Disk → DATA step → PROC MEANS.   1 disk pass.

Now, this test was run on a large SAS server with more than 300 GB of memory, so it is entirely possible that both large tables were in disk cache. On a smaller machine, the "Champion" may have had to write and re-read physical disk and would run much slower.

PERFORMANCE IMPLICATIONS OF SHORT NUMERICS

SAS physically arranges numeric variables on disk in a different order than their logical order.   It puts all 8-byte variables first, on double word alignment.   All 4-byte variables are next, on single word alignment.   Lengths 3, and 5 to 7 are last with no alignment.

CPUs process aligned storage faster than unaligned. Thus, if one has the choice of making a numeric variable length 3 vs 4, or 7 vs. 8, the performance benefits of the longer lengthmay outweigh the space savings of the shorter length.

These examples use views rather than physical data files to pass data, to measure only the effects of truncation, not disk input and output.

Proof that views truncate short numerics:

```
data v / view=v;
  length x 3;
  x=1234567890;
run;
data y;
  length x 8;
  set v;
  put x=;
run;
Results: x=1234436096. Clearly the value was truncated to 3 bytes.

/* Stress test: Length=3 unaligned storage vs Length=4 aligned. */
%let limit=1E7;
data vChampion / view=vChampion;
  length X1-X100 3;
  retain X1-X100 12345;
  do n=1 to &limit;
     output;
  end;
run;
data _null_;  set vChampion; run;

Results:  3.39 seconds CPU.

data vChallenger / view=vChallenger;
  length X1-X100 4;
  retain X1-X100 12345;
  do n=1 to &limit;
     output;
  end;
run;
data _null_;  set vChallenger; run;
Results:  2.75 seconds CPU, even though vChallenger passed more data.
```

ZERO-LOADING

Suppose an application has a block of variables that are more likely to be zero than the ones around them. Or supposed they are more likely to be missing, and one desires to use the missing-to-zero transformation discussed in the Compress=Char section above.

As discussed above, one could define those variables together, so that multiple occurrences of those zero values compress more tightly. However, some applications may require preserving a natural variable order, so that SAS variable lists like THIS--THAT have an application-specific meaning. It is still possible to enjoy the space savings of consecutive zero storage by defining the lengths of the variables so that the ones more likely to be zero are physically stored together.

Zero-loading is a technique that saves space by reserving one of the following three groups for variables most likely to be zero: 8-byte variables, 4-byte variables, or all other. Zero-loading works because all variables in the group are physically together. In the following example, there are 10 pairs of variables unimaginatively named ODD1, EVEN1, ODD2, EVEN2, etc. where the ODD variables have random values, but the EVEN variables are all zero:

```
options compress=yes;

data Naieve_var_lengths;
   /* Baseline:Define 20 vars as 10 ODD-EVEN pairs, all begin 0 */
   Array VARs[20] 3  ODD1 EVEN1 ODD2 EVEN2 ODD3 EVEN3 ODD4 EVEN4
                     ODD5 EVEN5 ODD6 EVEN6 ODD7 EVEN7 ODD8 EVEN8
                     ODD9 EVEN9 ODD10 EVEN10 (20*0);

   max_len3 = 2 ** (8*3 - 11);

   do row=1 to 1000000;
      /* Make every ODD var non-zero */
      do idx=1 to 19 by 2;
         vars[idx] = intZ(uniform(0) * max_len3);
      end;
      output;
    end;
    drop max_len3 idx row;

    /* Write one row of output as a demo */
    PUT _ALL_;
run;
%Size;
Result: Size of NAIEVE_VAR_LENGTHS with Compress=YES is 74,186,752

/* Cheater: re-order the vars to put the all-zero EVEN vars first,
then the non-zero ODD vars. */
data reordered;
  length odd1  - odd10  3;
  length even1 - even10 3;
  set ME.Naieve_var_lengths;
run;
%Size;
Size of WORK.REORDERED with Compress=YES is 57,409,536
```

As expected, this makes the table smaller, but "cheats" by re-ordering the variables.

```
/* Challenger: zero-loading makes the table smaller by making the
all-zero EVEN variables larger but preserves variable order. */
data Zero_loaded;
  set Naieve_var_lengths;
  length even:  4;
run;
%Size;

Size of WORK.ZERO_LOADED with Compress=YES is 57,409,536
```

(Pay no attention to percent compression, as nominal size is larger.)

Wow! Same physical size as the cheater, but without cheating. Lengthening the EVEN variables from 3 to 4 forces them to be physically stored together and saves space.

USE ZIP TECHNOLOGY TO COMPRESS BETTER

SAS Compression was designed to save CPU at the cost of far-from-optimal compression. The popular "zip" tools have the opposite goal: to compress as tightly as possible with less concern for CPU consumption. Thus, zip technology compresses better than SAS compression.

One can compress SAS tables with Unix or Windows "zip" or "gzip" tools. A SAS table in Unix, Windows, or a zOS HFS library is physically memname.sas7bdat, where memname is the SAS member name in lower case.

Problems with this:

1. If the table has an index, there is also memname.sas7bndx, and if the table has extended attributes, memname.sas7bxat. To properly preserve the table, you must zip and unzip these pieces together as a unit
2. To use the data, you must completely unzip it, even to do a simple PROC CONTENTS on it.
3. The original and zipped versions must be managed separately. This creates headaches for those unaccustomed to these tools, since zip and gzip behave very differently.
4. Once the data is zipped, it disappears from SAS/EG and Studio listings.
   Out of sight, Out of mind.

The Unix version of SAS offers a creative solution that reads zipped SAS data via a named pipe but requires awkward manual effort to start the unzip process in a separate physical process from the SAS session. SAS hangs indefinitely if it tries to read from a pipe to which no one is writing. Not recommended.

ZIPPER

The Zipper includes with this paper solves those problems. It zips SAS data and builds a SAS data view that reads it on the fly. It uses the new FILENAME ZIP technology, so does not require shell escape (Options XCMD) or any other software. A "zipper" view is a SAS data view that appears in a library listing as if it were any other SAS dataset.

But the best part is that anyone with access to the library can read a zipper view without knowing or caring that he is really reading a zipped copy of the data. Performance is slower than reading a normal SAS table, and one cannot use random access techniques like SET-with-POINT or use an index, but for normal SAS work, a zipper view behaves exactly like the original table.

Zipper has been tested on Unix, but should work on Windows, also. Zipper and sample testing code is in Appendix B, mostly to allow it to have wider margins for SAS code.

CONCLUSION

Ever-larger datasets put increasing strains on analysts and computer resources. The techniques presented in this paper can help save space and time, so that resource limits become less likely to interfere with our ***real*** purpose: to provide analytic insight.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Glen Becker
USAA
(210) 913-5193
Glen.Becker@USAA.com

APPENDIX A : UTILITY MACROS

The examples in this paper use the LibName, Table_is, and Size macros. These are intended for cut-and-paste to a SAS session. The wide margins and small font are a vain attempt to make them readable in this document.

```
/* Note, all these macros require names to be simple MEM or LIB.MEM, where both are SAS V7 names */

/* Utility macros to return member named and explicit or implied libname from a table name, no dataset
   options. */
%macro libname(table);
   %local libname;
   %let libname = %scan(&table, -2);  /* Return LIB of LIB.MEM, or null if none */
   /***/ %if %length(&libname)       %then &libname;  /* Caller provided explicit libname */
   %else %if %sysfunc(libref(USER)) %then WORK;      /* Non-zero means USER not assigned */
                                     %else USER;
%mend libname;

/* Return WHERE clause fragment to identify this table in the DICTIONARY views */
%macro table_is(table);
   %local libname memname;
   %let table=%upcase(&Table);  /* Because DICTIONARY tables have LIBNAME and MEMNAME uppercase */
   ( libname eq "%libname(&table)" and memname="%scan(&table,-1)" )
%mend;

/* Macro to write the physical size the caller's table or file to the SAS log.
If Table is not a physical file name, it is interpreted as SAS table, default is most recently created
table. */
%macro size(table);
  %local filepath fileref ID filesize;
  %if not %length(&table) %then %let table = &syslast;

  /* Get fully-qualified physical file name.  OK to use forward slash in Windows; SAS understands. */
  /***/ %if %sysfunc(fileexist(&table))  %then
             %let filepath=&table;  /* Was a physical filename */
  %else %if %sysfunc(exist(&table))      %then
             %let filepath=%sysfunc(pathname(%libname(&table)))/%lowcase(%scan(&table,-1)).sas7bdat;
  %else %if %sysfunc(exist(&table,view)) %then
             %let filepath=%sysfunc(pathname(%libname(&table)))/%lowcase(%scan(&table,-1)).sas7bvew;
  %else %do;
     %put ERROR: Size macro determined that &table does not exist.;
     %return;
  %end;

  /* FILENAME() 1st parm is the name of the macro var for the fileref, not the fileref itself.
     Since it is null, SAS assigns a fileref. */
  %if %sysfunc(filename(fileref, &filepath)) %then; /* No check return code */

  %let ID=%sysfunc(fopen(&fileref));   /* Open the file */
  %if &ID %then %do;
      /*  FINFO() is a character function, so must format the result with a separate %sysfunc(putN()).
          FINFO() 2nd argument is case-sensitive literal text.
      Assigning result to a macro variable does an implicit TRIM(). */
      %let filesize = %sysfunc(putN( %sysfunc(Finfo(&ID, File Size (bytes) )), comma20.));
      %let ID = %sysfunc(fclose(&ID));    /* Close file, invalidate now-stale handle */
      %put Size of &table with Compress=%sysfunc(getoption(Compress)) is &filesize..;
  %end;
  %else %put ERROR: Size macro could not open &filepath..;

  %if %sysfunc(filename(fileref, )) %then;   /* Release fileref, ignore return code */
%mend size;
```

APPENDIX B : ZIPPER

```
/* Macro to build a "zipper" view of the given dataset with the given output name.

   Usage:  %Zipper(Input_table, out=Output_table);

Both parameters are required:

Data    Names the input SAS dataset.  Must be a one- or two-level SAS V7 name, no dataset options, but it
        can have non-standard column names per OPTIONS VALIDVARNAME=ANY.  Can name any type of SAS dataset,
        even an extant "Zipper" view.

Out=    Names the output SAS data view.  May be a one- or two-level SAS V7 name, no dataset options.
        Output library must be a SAS V9 library, not a remote SAS/Connect or SAS/Access library.
        Out cannot name the same table as Data.

Zipper creates two physical output files in the output directory, both of which are required to read the data.

        whatever.sas7bvew   The SAS data view that users will read.
        whatever.gz         A gzip file that contains the zipped image of the data.

To delete a zipper view, you must delete both files.  Zipper does not provide a way to do so.

IMPORTANT: A "Zipper" view imbeds the physical path of the gz file into the view.  A zipper view does NOT
Tolerate being moved to another location by copying files.  To move a zipper view to another SAS library, you
must re-build it:

   %Zipper(OLDLIB.WHATEVER, OUT=NEWLIB.WHATEVER);

Limitations:
1. On zOS, the output library must be an HFS library.
2. The SAS Attrib statements that define the variables in the resulting view must fit into a single macro
   variable, with a maximum of 32,767 bytes.  The length of the variable labels thus limits the number of
   variables that Zipper can handle.  (Easy work-around with more sophisticated ATTRIB statement generation.)

Physical output format:  the whatever.gz file is a gzipped undelimited stream of record images, aka RECFM=F.
Each record has numeric data, followed by character.  Numeric data is stored as double-precision floating-
point, which SAS calls Format=RB8.  Character data is stored as-is, which SAS defines as Format $CHARnn.
*/
%macro zipper(data, out=);

%local Attribs   /* SAS Attrib statements to define variables in the table */
       Charlens  /* Blank-delimited list of character variable lengths */
       Charvars  /* Number of character variables */
       Num_lens  Char_lens  /* sum of numeric and character lengths */
       Lrecl     /* Number of bytes in a SAS row image: 8 bytes per num var, sum(lengths) for char vars */
       ZipFile   /* fully-qualified name of zip file */
       C         /* generic counter */
       ;

/* Get table metadata */
proc sql noprint;
  create table defs(compress=no) as
  select varnum
       , Nliteral(name) as name length=48     /* simple name or 'quoted Name-literal'N */
       , upcase(type)   as type length=1      /* N or C */
       , length
       , format
       , label
  from dictionary.columns where %Table_is(&data)
  order by varnum;

 /* Attrib statements to reproduce these definitions */
 select catx(' ', 'Attrib', name
               , catt('length=', translate(type,'$ ','CN'), length)
               , case when format=' ' then ' ' else 'format=' || format              end
               , case when label =' ' then ' ' else 'label='  || quote(trim(label)) end)
   into :attribs separated by '; ' from defs;

  /* Create blank-delimited list of character lengths */
  select length into :charlens separated by ' ' from defs where type='C';
  %let   charvars = &SQLobs;

  /* Num_lens, Char_lens, and Lrecl are the nominal lengths of SAS variables as they appear in the PDV */
  select sum((type='N') * 8), sum((type='C') * length)
   into  :num_lens          , :char_lens
   from defs;
  %let Lrecl = %eval(&Num_lens + &Char_lens);
quit;

/* Name the ZipFile the same name as the physical SAS table, but with a .gz suffix */
%let ZipFile = %sysfunc(pathname(%libname(&out)))/%lowcase(%memname(&out)).gz;

filename zipper zip "&ZipFile" gzip;

data _null_;
  set &data;
  array nums [*] _numeric_;     /* Array of all numeric vars */
  array chars[*] _character_;   /* Array of all char vars */

  file zipper recfm=F lrecl=&lrecl;   /* Output file is zipper, fixed-length records */
  if dim(nums) then put (nums[*]) (RB8.) @;    /* Write numeric vars as RB8. which a is bit-for-bit copy. */
  /* Each character var must be written with the correct $CHARnn output format */
  %do C=1 %to &Charvars;
      put chars[&C] $char%scan(&Charlens,&C). @;
  %end;
  put;  /* Close output line */
run;
filename zipper;   /* Free fileref */

/* Define output view */
data &out / view=&out;
   &Attribs;   /* Define original vars */
   /* Create array of original vars before we define any transient ones */
   array nums [*] _numeric_;
   array chars[*] _character_;

   /* Define temp vars */
   _begin_temps = 0;              /* so can DROP _BEGIN_TEMPS -- _END_TEMPS;  */
   %if &Num_lens %then %do;
      length _Num_image $ &num_lens;    /* Define a char var large enough for all numeric vars */
      length _Num_addrs $ 8;            /* Plase to save address of num vars */
      _Num_addrs = addrlong(nums[1]);   /* All num vars are in consecutive storage beginning with nums[1] */
   %end;

   /* Create fileref ZIPPER for &Zipfile with the ZIP Access Method, GZIP keyword requests the output be in
      gzip format.  */
   if filename('ZIPPER', "&ZipFile", 'ZIP', 'GZIP') then;   /* Ignore return code. */

   /* Open that fileref, S=Sequential, F=Fixed-length records */
   _id = fopen('ZIPPER', 'S', &lrecl, 'F');

   if not _ID then putlog "ERROR: Zipper could not open &ZipFile, cannot run without it.";
   else do while(~fread(_id));     /* Iterate until read fails */
      %if &Num_lens %then %do;
          if fget(_id,  _Num_image, &Num_lens) then;  /* Copy numeric var image to _num_image, ignore RC */
          call pokelong(_Num_image, _Num_addrs);      /* Copy _num_image to numeric vars */
      %end;
      %do C=1 %to &Charvars;
          if fget(_id, chars[&C], %scan(&Charlens,&C)) then;
      %end;
      output;
   end;

   if fclose(_id) then;         /* Close zipfile, ignore return code */
   if filename('ZIPPER') then;  /* Release the fileref, ignore return code */

   _end_temps = 0;
   drop _begin_temps -- _end_temps;
run;
%mend zipper;
```

```
/* Testing code: */
   data bigshoes(compress=yes);
      set sashelp.shoes;
      do n=1 to 100;  output; end;
   run;

   %zipper(bigshoes, out=zipshoes);

   proc compare base=bigshoes compare=zipshoes; run;
   %put Compare return code is &sysinfo;
   /* Explanation 5 = 1 (for Dataset label not reproduced),
                    + 4 (for Informats not reproduced) */
   %Size(bigshoes);
   %Size(zipshoes);
   %Size(%sysfunc(pathname(%libname(zipshoes)))/zipshoes.gz);
```