Paper 39

So You're Running Out of Sort Work Space . . . Reducing the Resources Used by PROC SORT

Bob Virgile
Robert Virgile Associates, Inc.

Overview

If you're concerned about the storage space used by PROC SORT, there may be hope. The SAS software contains tools and strategies that reduce or trade off the resources needed to sort data. This tutorial explains these tools, with an emphasis on reducing the amount of sort work space. Therefore most of this paper applies specifically to the MVS environment.

The Basics of Sort Work Space

When the SAS® software runs PROC SORT, the operating environment reserves storage space to perform the sorting. Unlike under other operating systems, under MVS this space remains separate from the WORK area. If you run out of sort work space, increasing the WORK area (or deleting data sets which are no longer needed) will not help. Sort work space also has physical characteristics which differ from the WORK area. This would be a typical DD statement used by the SAS procedure to define a sort work area:

```
//SORTWK01 DD UNIT=SYSDA,
//         SPACE=(CYL,(&SORT),,CONTIG)
```

This JCL points out two key features of each sort work area:

- it occupies a contiguous block of space, and
- its size (&SORT) is a parameter that the programmer can pass to the operating system when submitting a job

SAS programmers can adjust two numbers to expand the total amount of sort work space:

- the size of each sort work area, and
- the maximum number of sort work areas created by the program.

Specify the &SORT parameter to adjust the size of each sort work area:

```
// EXEC SAS,SORT=8
```

Using this EXEC statement, the job would now reserve 8 cylinders for each sort work area.

Limit the number of sort work areas in the SAS code, not the JCL, either as a global option or as a PROC SORT option:

```
options SORTWKNO=3;

proc sort data=sales SORTWKNO=5;
```

This option determines the maximum number of sort work areas that PROC SORT is allowed to use.

These steps are effective as long as the operating system can provide enough sort work space. But what if it can't? How can you reduce the amount of sort work space your program requires? Alternatively, how can you locate and use additional sort work space which the operating system would not normally find?

Going Beyond the Basics

A number of measures, over and above SORT= and SORTWKNO=, can either increase available sort work space or decrease the need for sort work space. Here are some ideas.

Idea #1: Take a Systems Programmer to Lunch

Just for the record, I am not, nor have I ever been, a systems programmer. Systems programmers often know of disk packs with a lot of free space. Such disk packs might be reserved for another use. But if all the conditions are right (the pack is not being used, you need the space for one day only, and the systems programmer is inclined to locate such a disk pack for you), you
might be able to code your JCL to utilize a specific disk pack for sort work space:

```
//SORTWK01 DD UNIT=PACK35,
//         SPACE=(CYL,800,,CONTIG)
```

For each disk pack, you will need to find out how

much space is available, and code a SORTWK DD statement. Number the DD statements consecutively. The next one might be:

```
//SORTWK02 DD UNIT=PACK38,
//         SPACE=(CYL,545,,CONTIG)
```

Once again, the key steps are locating (partially) empty disk packs, obtaining permission to use them, determining how much contiguous free space they hold, and designating them in your JCL. Remember that any permission you receive to use a disk pack automatically comes with (approximately) a ten-hour time limit. If you attempt to use the disk pack the next day, but you haven't obtained permission for that day, it can cost many lunches to repair the damage.

Idea #2: Wait Until Monday

No, the computer is not in a better mood after a relaxing weekend. Normal activities during the week can include deleting data sets and releasing unused space from the end of data sets. As a result, by Friday, disk packs contain unused space interspersed between existing data sets. Since sort work space consists of contiguous space, these interspersed blocks of storage are not useful as sort work space.

On a regular basis (typically over the weekend), systems programmers run jobs to collect and combine unused disk space. Actually, the jobs copy all data sets off disk packs, then recopy them back to the beginning of the disk pack. (Additional maintenance may take place over the weekend as well, such as deleting uncatalogued data sets or archiving data sets which have not been used recently.) In so doing, these jobs combine smaller amounts of unused space, which on Friday appeared in between data sets, into one large block of unused space at the end of the disk pack. The net result: the system has a lot more contiguous free space available on a Monday compared to a Friday. Thus the possible values for the SORT= parameter are higher on Monday.

The type and frequency of maintenance varies from one disk pool to another. For example, sort work space may use disk packs from which all data sets are deleted overnight. In this case, instead of waiting until Monday, consider two related approaches:

- submit your job first thing in the morning
- submit your job to run overnight, but restrict it so that it doesn't begin until a nightly disk cleaning program completes.

Talk with a systems programmer to discover the disk pool used for sort work space, the timing of disk maintenance programs on various disk pools, and the JCL needed to release your job once another job completes.

Idea #3: Subset the Variables

The smaller the data set, the less sort work space is needed. By analyzing the program, you may discover that you can drop some of the variables. In that case, one possibility is to add a DATA step to subset the variables:

```
data narrow;
set wide (keep=just three vars);

proc sort data=narrow;
by vars;
```

However, this DATA step now adds to the CPU time for the program. Perhaps, that same DATA step could perform necessary data manipulation, eliminating the need for a DATA step following PROC SORT. However, in many cases that DATA step represents extra, unrecoverable CPU time. Instead, another possibility would be to ask PROC SORT to remove extra variables. Here are two attempts:

```
proc sort data=wide
         (keep=just three vars)
          out=narrow;
by vars;

proc sort data=wide
          out=narrow
         (keep=just three vars);
by vars;
```

In theory, placement of the `keep=` data set option should make a big difference. The first PROC SORT should be sorting three variables, and outputting the results. The second PROC SORT should be sorting ALL the variables, but outputting only three of them. In practice, under Version 6 of the SAS software, it makes no difference where you place the `keep=` data set

option. Both PROC SORTs sort all the variables, and subset upon outputting to the new data set. One reason for this (whether sufficient justification or not) is that the NODUPLICATES option would generate different results if it operated on three variables as opposed to all the variables. Under Version 7, you have the option of using the Version 6 behavior (sort all variables, subset upon output only) or subsetting the variables entering PROC SORT. If you are still working under Version 6, a simple workaround lets you minimize the CPU time for an extra DATA step. Use a view instead of a data set to subset the variables:

```
data temp / view=temp;
set wide (keep=just three vars);

proc sort data=temp out=narrow;
by vars;
```

Now PROC SORT uses the instructions in the view to retrieve observations from WIDE (retrieving only the three desired variables), sort the observations, and output them to NARROW. The DATA step runs very quickly, since it stores instructions only, never actually retrieving data from WIDE.

Another method to sort fewer variables in PROC SORT is the TAGSORT option:

```
proc sort data=wide TAGSORT;
by vars;
```

TAGSORT does not sort all the variables. Instead, it creates an identifier for each observation, and then sorts just the BY variable(s) and the observation identifier. Finally, after sorting, it uses the observation identifier to retrieve the remaining variables.

Unfortunately, TAGSORT can also use a lot of CPU time. In one set of test runs, PROC SORT with TAGSORT took 3.8 times the CPU time (compared to PROC SORT without TAGSORT). This observed increase took place under the MVS operating system, using a data set with 100 variables and 100,000 observations. Under other operating systems, TAGSORT can actually DECREASE the necessary CPU time. You MUST perform these types of tests on your own hardware/operating system!

Instead of relying on canned software, you can program your own version of TAGSORT. For example:

```
data temp;
set wide (keep=vars);
recno=_n_;
proc sort data=temp;
by vars;

data wide;
set temp (keep=recno);
set wide point=recno;
drop recno;
```

The first DATA step keeps just the sort variables plus a pointer to the current observation number. PROC SORT sorts that information only, as TAGSORT would have done. The final DATA step uses the observation numbers in the new order to retrieve all variables from the original data set in sorted order.

The net results: under MVS, this workaround performed considerably faster than TAGSORT. Instead of requiring 3.8 times the CPU time, it required 1.3 times the CPU time. Again, since TAGSORT reduced CPU time under other operating systems, this workaround was not efficient outside of the MVS world.

Idea #4: Break Up the Data

Breaking up the data into subsets uses more CPU time and storage space. However, it uses less sort work space, since the program can sort smaller subsets. When sort work space is the bottleneck, the extra CPU time may transform a failing program into a long but successful program. For example, consider this original program which sorts one large data set:

```
proc sort data=huge;
by id;
```

A replacement program could create three subsets, each holding one third of the observations. Here is one possibility, assuming that the original data set contains 300,000 observations:

```
data subset1 subset2 subset3;
set huge;
if _n_ <= 100000 then output subset1;
else if _n_ <= 200000 then output
subset2;
else output subset3;

proc sort data=subset1;
by id;

proc sort data=subset2;
by id;

proc sort data=subset3;
by id;

data huge;
set subset1 subset2 subset3;
```

```
by id;
```

The program sorts each subset, then interleaves the sorted subsets. In fact, that is the method that PROC SORT uses when it must use multiple sort work areas. PROC SORT uses each sort work area to sort a subset of the observations, and then interleaves the results.

This example could eliminate the first DATA step. Instead, each of three PROC SORTs could have selected 100,000 observations:

```
proc sort data=huge (FIRSTOBS=1
          OBS=100000) OUT=SUBSET1;
by id;

proc sort data=huge (FIRSTOBS=100001
          OBS=200000) OUT=SUBSET2;
by id;

proc sort data=huge (FIRSTOBS=200001
          OBS=300000) OUT=SUBSET3;
by id;
```

This approach requires prior knowledge of the number of observations in the incoming data. If your data are stored on disk, you can easily discover the number of observations in a SAS data set:

```
data _null_;
if 0 then set huge nobs=N;
put 'Total number of obs:  ' N;
stop;
```

The condition `if 0` is always false. Therefore, the SET statement never reads any observations. Despite those facts, the nobs= option on the SET statement still retrieves the number of observations in the data set. The DATA step uses very little CPU time, since it reads none of the observations. Instead, the software examines the descriptor portion of the SAS data set, which already stores the number of observations in the data set.

This technique readily lends itself to macro language, where the parameters are the data set name and the number of subsets to use when splitting up the data set. For the sake of simplicity, this macro hard codes one BY variable named ID:

```
%macro split (dsn=, sets=);

  %local first  /* first observation */
         last   /* last observation  */
         n      /* number of obs     */
         subset /* numbers subsets   */
         perblock /* obs per subset  */
   ;
  %let first=1;
  %let subset=1;

  data _null_;
  if 0 then set &DSN nobs=nobs;
  call symput('N', put(nobs, 9.));
  call symput('perblock',
              put(ceil(nobs/&SETS), 9.));
  stop;
  run;

  %if &N > 500 and &SETS > 1 %then
  %do %until (&LAST >= &N);

      %let last = %eval(&FIRST +
                        &PERBLOCK - 1);

      proc sort data=&DSN
            (firstobs=&FIRST obs=&LAST)
            out=subset&SUBSET;
      by id;
      run;

      %let first = %eval(&last + 1);
      %let subset = %eval(&subset + 1);

  %end;

  %else %do;

      proc sort data=&dsn;
      by id;
      run;

  %end;

%mend split;
```

This macro executes PROC SORT as many times as necessary, each time sorting a different subset of observations. The key macro variables and their roles are:

`&dsn`
The name of the incoming data set being split into subsets (with each subset being individually sorted).

`&sets`
The number of subsets to create when splitting the data.

`&n`
The number of observations in the incoming data set.

`&perblock`
The number of observations per subset. (The final

subset may contain fewer observations due to rounding.)

For each PROC SORT, three additional macro variables come into play:

`&first`
The first observation to select, when creating the current subset.

`&last`
The last observation to select, when creating the current subset.

`&subset`
Numbers the subsets (1, 2, 3, etc.) For example, when `&subset` is 3, PROC SORT creates an output data set named `subset3`.

Many variations exist on these techniques. Here are some possibilities to consider:

- For data sets stored on tape, a DATA step could count the number of observations.
- Instead of breaking up the data into &sets subsets, create as many subsets as needed. Each subset would hold the next 100,000 observations.
- Instead of basing a subset's size on the number of observations, also factor in the size of an observation. Remember, PROC CONTENTS can output a data set which includes the length of each variable.

Let's examine a special situation that permits an interesting split of the data. Here is the original program that fails because of a lack of sort work space:

```
proc sort data=caribbean;
by state city;
```

Notice two key conditions. First, the program sorts by multiple variables. Second, the primary sort key (STATE) takes on a limited set of values, and thus represents a good mechanism for splitting the data.

A replacement program would begin by splitting the data according to the primary sort key:

```
data AL FL LA MS TX;
set caribbean;
select (state);
   when ('AL') output AL;
   when ('FL') output FL;
   when ('LA') output LA;
   when ('MS') output MS;
   when ('TX') output TX;
end;
```

After splitting the data, the program can sort each of the smaller subsets:

```
proc sort data=AL;
by city;

proc sort data=FL;
by city;

proc sort data=LA;
by city;

proc sort data=MS;
by city;

proc sort data=TX;
by city;
```

Notice that the BY statement now uses just one variable for sorting. Because the primary key (STATE) was used to split the data, it no longer needs to appear in the BY statement, thus reducing some of the sorting requirements.

Finally, assemble the pieces. The lazy way is to use a DATA step:

```
data caribbean;
set AL FL LA MS TX;
```

A shorter method (in terms of CPU time) would append the data sets:

```
proc append data=FL base=AL;
proc append data=LA base=AL;
proc append data=MS base=AL;
proc append data=TX base=AL;
```

Of course, now the name of the final data set is AL instead of CARIBBEAN. However, PROC DATASETS can easily change (or exchange) the names of data sets.

Idea #5: Let the SAS Software Sort

The SAS software includes a sorting routine which it uses to sort small data sets. Typically, the operating system also provides a sorting routine which is more efficient for sorting larger data sets. When your program invokes PROC SORT, the SAS software evaluates the SAS data set as being

"large" or "small" and chooses the sorting routine it expects to be more efficient. (Actually, the software looks at the number of observations in the SAS data set and compares that number to the setting for the global option SORTCUTP. An OPTIONS statement can control the value of SORTCUTP, although such a step is rarely taken.)

Your program can control whether the program uses the SAS sorting routine for a given PROC SORT. Just modify the global option sortpgm:

```
options sortpgm=SAS; /* host, best */
```

The SAS software sorting routine uses memory, rather than sort work space, for sorting. In test runs (again, the test data set contained 100 variables and 100,000 observations), it took 1.5 times the CPU time. Beware! The CPU time may increase exponentially using the SAS sorting routine. Some sorting routines require CPU time proportional (in part) to the square of the number of observations. However, if sort work space (not CPU time) is the bottleneck, the SAS sorting routine becomes a viable option.

Some Final Notes

This paper raises many technical issues which stretch the scope of my knowledge. It might take a panel of experts rather than one individual to fully answer related questions. Here are some questions that I was still working on when I hit the deadline for submitting a final draft:

- With multiple sort work areas, does PROC SORT (as I have claimed) sort a portion of the data set in each, then interleave the results?
- Is PROC SORTT any different from the global option sortpgm=SAS?
- How often does sort work space default to using the same disk pool used for permanent data sets? What routine maintenance normally gets performed on different varieties of disk pools? What other functions can use various disk pools?
- What is the mathematical relationship between the number of observations in a SAS data set and the amount of CPU time used by various sorting routines?
- Sort work space gets allocated as contiguous with no secondary allocation. Is this a requirement, or is it related to speed or other considerations?
- What is the name and usage of the Version 7 option that lets PROC SORT subset variables from the incoming data set?

Finally, there may be analogous issues that apply under other operating systems. For example, must other operating systems use contiguous space? Can other operating systems swap space between the Work area and sort work space? One known "feature" is that other operating systems don't always release storage space that was temporarily needed by earlier PROC SORTs. In practice, you might gain space for PROC SORT by permanently saving your data, closing down your current session, and opening a new SAS session.

Remember, saving your results is easy if you plan ahead. First, define a permanent library:

```
libname perm 'path to directory';
```

To save your SAS data sets permanently, you don't have to change all the one-level data set names to two-level names. Instead, add one statement at the right point in the program:

```
options user=perm;
```

From that point in the program, all SAS data sets with one-level names get saved in the designated library.

I may have answers for these questions by the time the conference begins. Any information I collect will be presented at SUGI, and may also appear in a subsequent publication of this paper. All comments, questions, and suggestions are always welcome. Feel free to call or write:

Bob Virgile
Robert Virgile Associates, Inc.
3 Rock Street
Woburn, MA 01801
(781) 938-0307