

Efficient Programming Revisited: A Review of Performance-Enhancing Features in Version 6.06

Frank DiIorio, Family Health International
Paul Dalberth, Family Health International

The introduction of Version 6.06 of the SAS System changed the rules for program tuning. New features were made available, some traditional techniques rendered ineffective, and the programmer were presented with a much richer and more complex environment in which to work. Simply put, Version 6.06 requires that approaches to program tuning must be reevaluated. This paper will discuss and quantify the impact of some of the new features of Version 6.06.

Background

As SAS software becomes more embedded in organizations' data management and decision-making processes its impact on system resources becomes greater. Methods for reducing this impact hardly constitutes a new SAS paper topic. Many SUGI papers have described methods for "hard coding" efficiencies into SAS programs. They typically discuss table lookup and low-level DATA step programming methods often borrowed from Third Generation languages. These techniques, when applied in the appropriate environment, are often highly effective. They improve performance in some applications and bring others into the realm of feasibility. They come, however, at the expense of a high degree of programmer intervention and sophistication.

Version 6.06 introduced indexing, dataset compression, and the stored program facility. These features offer an entirely new perspective on program tuning. They are the philosophical opposite of the methods used in earlier releases of SAS software. Rather than forcing the user to deal with program efficiency on a statement-by-statement basis, these features are global and transparent. That is, they are often decided upon when the application is designed, with the resulting efficiencies realized without further end-user or programmer intervention. This "passive tuning" of applications takes many decisions out of the hands of the dataset's users: the efficiencies and, unfortunately, diseconomies accruing to the methods are inherited by the end-user.

Organization

This paper presents resource usage measures for a variety of typical tasks involving dataset creation and use. Results should be considered in light of each system's configuration. Memory, disk, cpu and other constraints vary considerably from site to site. For example, a technique reducing disk storage requirements for a datasets at the cost of increased CPU usage would be inappropriate for a site with abundant storage and a high proportion of compute-intensive programs. What we describe here is how we selected the benchmark tasks, what resources were consumed, and why our results agree with or diverge from what was expected, given our reading of the literature.

Restrictions

This is, potentially, a very broad topic which could consume many hours of cpu and programmer time. We made the following restrictions (in some cases very reluctantly). First, only "generic"

tasks were tested. Products such as SAS/GRAPH¹ and SAS/AF¹ were not considered. Second, direct, "active" DATA step tuning was minimized. This topic has been covered in great detail elsewhere in the literature, and only diverts attention from the more global, underlying issues surrounding passive tuning.

A third restriction was avoidance of non-standard dataset engines. Fourth, access to datasets via SQL was considered important, so important that it warrants a separate paper. Finally, the tools investigated are system-independent: no operating system features (page size, blocking factors, etc.) are evaluated. The benchmarks are SAS-only, and reflect performance in the authors' workaday operating system, Digital Equipment Corporation's (DEC) VMS.

Method

The datasets used in our analysis were "synthetic." Rather than use idiosyncratic, hard-to-replicate production data we chose to create a test library of datasets which could be easily replicated and used for comparative tests at other sites. (Availability of the programs to create the datasets and run the tests is discussed at the end of the paper.)

A total of 81 datasets was created. Several features of the datasets varied. Among these were size, composition, compression, and indexing. Another feature, the concurrency engine, was of interest but not pursued since it was unique to the DEC VMS environment. Dataset features are briefly described below.

- **Size** Performance may be directly related to dataset size. Datasets of 100, 1,000, and 5,000 observations were created. These are sizes most frequently used by the authors in their routine data management tasks.
- **Composition** The number and data type of variables in a dataset may have an impact on I/O features such as dataset compression. The test datasets consisted of all numeric data with default (8-byte) lengths, all numeric data with user-specified lengths, and a mixture of character and numeric data of user-specified lengths.
- **Compression** Data were stored in compressed and uncompressed forms. There were two types of compressed datasets: one allowed reuse of deleted space (REUSE=YES dataset option), the other did not.
- **Indexing** Datasets were either unindexed or indexed on a single numeric variable. Both uniformly and non-uniformly distributed indexes were created.

What's New in Version 6.06

Many of the features of the SAS System introduced in Version 6.06 are, at first blush, the answer to a performance-hungry programmer's needs. The "no free lunch" axiom has seldom been more appropos than when using these features: the benefits of a technique are often accompanied by surprisingly high costs. This section briefly describes the techniques used in the tests. It is *not* intended to be a thorough review, but simply a quick overview of merits and drawbacks.

Dataset Compression

SAS programmers often fail to use disk storage efficiently. Default length numeric variables and overly-generous lengths for character data cause a large proportion of many datasets to be wasted space, high-tech "fluff." Dataset compression eliminates the extraneous space from a dataset. In a

¹SAS/GRAPH and SAS/AF are the registered trademarks of SAS Institute, Inc., Cary, NC.

compressed dataset, a variable using only three bytes of a 20 byte character variable will actually require less disk space to store than one using 17 of the 20 bytes. Accumulated savings in disk space can be significant.

The benefit of this technique is measured by disk space saved. The cost is seen in increased CPU usage: before an observation from a compressed dataset can be passed to a DATA step or procedure, it must undergo "decompression" not required with uncompressed data. Another, less quantifiable drawback of compression is that it may encourage sloppy dataset design. Rather than thinking through the variables' appropriate lengths a programmer may instead rely on compression to achieve efficient use of storage. Such reliance is ill-advised.

Stored Program Facility

The inherently slow compile-and-execute nature of the DATA step has been a long-standing source of frustration to SAS performance gurus. Version 6 saw the introduction of the Stored Program Facility (SPF). This technique allows the programmer to store a compiled version of DATA steps. Execution of this compiled code is almost always faster than its uncompiled equivalent, particularly when the step is not I/O bound. The compiled program cannot process macro variables or %INCLUDEd code, so some changes to programming style may be required.

The costs of the extra CPU time and disk space required for compilation can be amortized over the number of times the module is executed. There is another, more subtle cost attached to this apparent can't lose feature. The burden of source and executable concurrency is added to production environments. That is, the association between the source program and executable module must be known. Unlike compile-and-execute SAS, where the source code is executed directly, SPF requires that the programmer be certain that the executable module was generated by a particular source, or program, file. Protocols, automated and otherwise, must be developed to ensure source-executable correspondence.

Indexing

A traditional weakness of SAS software has been the sequential nature of its processing. DATA steps and procedures typically passed through the data observation by observation, even if only a small proportion of the dataset's observations were actually needed for processing. Version 6.06 introduced support for single and compound indexing of numeric and character variables. These features allow direct access to observations of interest based on values of significant, or index, variables. Indexes can bypass large amounts of resource-demanding input and output processing, since they allow SAS to move directly to observations of interest.

The cost of indexing is incurred several ways. First, it takes CPU time and disk space to build and maintain the index. The costs are magnified if a dataset has many indices. Second, the typical use of the dataset may not exploit the strengths of indexed access. For example, a large production file used mainly for periodic report writing would not be suitable for indexing if *ad hoc* queries were infrequent or involved non-index variables. Finally, selection and creation of indexes requires a somewhat higher level of programmer sophistication. The programmer must know, for example, that performance improvements for an excessively indexed dataset may be crippled by system overhead.

Benchmarks

The brief enumeration of strengths and weaknesses presented above prompts the question "how much?" How much disk space can be saved with these techniques? How much CPU time? A more

intriguing question is "which one?" Is the most effective method for a particular task the one which is suggested by the literature? To begin answering these questions we developed a set of dataset creation and usage benchmarks. They compared typical end-user activities using realistic size datasets. The results and discussion of each activity are presented in this section. Refer to "Production Environment" at the end of this paper for a detailed description of the system on which the programs were run and how the programs were executed.

Dataset Creation

A total of 81 test datasets was created (for each of three sizes, datasets differing by compression, indexing, and composition - see "Method" section, above). Measures for the uncompressed and compressed (REUSE=NO) group are presented in Table 1 (compress with REUSE turned on was similar to that with REUSE off).

Table 1: Dataset Creation								
Compress	Indexing	Dataset Size ▶	100		1,000		5,000	
		▼ Structure ¹	CPU	Disk	CPU	Disk	CPU	Disk
No	None	Default numeric	3.25	41	4.18	241	8.48	961
		Numeric (LENGTH)	3.71	89	9.64	441	35.12	2113
		Character-numeric	4.59	101	10.60	601	35.30	2951
	Uniform	Default numeric	1.56 1.10	61 33	4.10 3.78	241 129	8.67 13.04	961 417
		Numeric (LENGTH)	1.46 1.26	89 33	9.58 3.91	441 97	35.06 12.65	2113 321
		Character-numeric	1.68 1.26	101 33	10.36 4.02	601 97	35.48 13.15	2951 321
	Non-uniform	Default numeric	3.20 1.65	61 33	4.94 3.98	241 129	11.53 13.42	961 385
		Numeric (LENGTH)	4.18 1.62	89 33	10.69 3.75	441 97	41.02 13.49	2113 321
		Character-numeric	4.87 1.63	101 33	10.68 3.93	601 97	38.17 13.00	2951 321
Yes	None	Default numeric	3.10	17	4.44	129	11.13	705
		Numeric (LENGTH)	3.88	65	10.98	449	40.50	2177
		Character-numeric	4.58	129	11.98	577	41.60	2625
	Uniform	Default numeric	2.90 1.58	65 33	4.82 4.32	129 129	10.95 13.95	705 417
		Numeric (LENGTH)	3.72 1.75	65 33	11.07 4.43	449 97	40.36 16.87	2177 321
		Character-numeric	4.37 1.75	129 33	11.80 5.11	577 97	42.06 17.85	2625 321
	Non-uniform	Default numeric	3.25 1.63	65 33	5.38 4.06	129 129	13.58 14.21	705 385
		Numeric (LENGTH)	4.14 1.77	65 33	11.99 4.71	449 97	43.83 16.68	2177 321
		Character-numeric	4.82 1.82	129 33	12.21 4.80	577 97	44.42 18.18	2625 321

Table 1: Dataset Creation (cont.)

¹See "Production Environment" note at the end of the paper for details.

"disk" indicates the number of 512-bytes blocks used by the dataset or index.

In cells with two lines, the top line indicates measures for dataset creation. The bottom line indicates resources used for index creation.

The figures generally reflect what was expected: compressed datasets required more CPU than their uncompressed counterparts; disk and CPU usage increased linearly with dataset size. The compressed/uncompressed comparison is particularly noteworthy. Table 2 compares each dataset's resource requirements. Cells represent the ratio of compressed to uncompressed.

Table 2: Compressed-Uncompressed Dataset Creation

Indexing	Dataset Size ▶	100		1,000		5,000	
	▼ Structure	CPU	Disk	CPU	Disk	CPU	Disk
None	Default numeric	.95	.41	1.06	.53	1.31	.73
	Numeric (LENGTH)	1.04	.73	1.13	1.01	1.15	1.03
	Character-numeric	.99	1.27	1.13	.96	1.17	.88
Uniform	Default numeric	1.68	1.06	1.15	.53	1.14	.73
	Numeric (LENGTH)	2.01	.73	1.14	1.01	1.19	1.03
	Character-numeric	2.08	1.27	1.17	.96	1.23	.88
Non-uniform	Default numeric	1.00	1.06	1.05	.53	1.11	.73
	Numeric (LENGTH)	1.01	.73	1.15	1.01	1.11	1.03
	Character-numeric	1.02	1.27	1.16	.96	1.21	.88

Cells represent the ratio of compressed to uncompressed for the particular resource. Thus .95 at the top left indicates compression CPU time to create a 100 observation compressed dataset was somewhat less in the non-indexed, default numeric case than for a non-compressed dataset.

Shaded cells indicate anomalous results: CPU ratios less than 1, Disk ratios greater than 1.

Disk comparisons are for data only. Indexes were not taken into account since they were the same size for compressed and non-compressed datasets.

"disk" indicates the number of 512-bytes blocks used by the dataset or index.

The ratios emphasize the CPU penalty incurred by compression: 10% to 30% for 5,000 observation datasets. Particularly interesting is the *increase* in dataset size when compressing datasets with user-specified lengths. This pattern was consistent for all 1,000 and 5,000 observation datasets. It suggests that compression may be most appropriate in datasets with the slack guaranteed by the overly-generous default lengths used by SAS. User-specified lengths remove the slack and give SAS less wasted space to compress.

Using Compressed Datasets

The CPU cost of compression should, in theory, be incurred not only when the dataset is created but also when it is used. Table 3 presents comparisons of the PRINT and MEANS procedures on 100 and 5,000 observation datasets. All defaults were taken, so PRINT reflects mostly I/O, and MEANS a combination of I/O and CPU spent performing calculations.

Table 3: Compressed-Uncompressed Dataset Use									
Test ▶	100 Print			100 MEANS			5,000 MEANS		
▼ Structure	Cmp	Un	C/U	Cmp	Un	C/U	Cmp	Un	C/U
Default numeric	2.43	2.54	.95	.32	.31	1.03	7.72	6.48	1.19
Numeric (LENGTH)	6.38	6.47	.98	1.12	1.07	1.04	29.18	25.67	1.13
Character-numeric	5.86	5.81	1.01	.95	1.07	.88	26.74	21.0	1.27

The table shows modest CPU improvement for the "pure" I/O proc (PRINT). MEANS, on the other hand, generally revealed slight performance degradation at 100 observations and penalties ranging from 13% to 27% at 5,000 observations. Compression imposes more of a cost as dataset size increases for mixed I/O-calculating activities.

Stored Program Facility

Documentation for the Stored Program Facility (SPF) suggests that its most effective use is in cases where the DATA step has many statements and relatively little SAS dataset I/O activity. The SPF tests compared performance at 100 and 1,000 observations for compiled DATA steps of 20, 100, and 300 executable statements. The statements were simply a group of 20 IF-THEN-ELSE conditions used to create variables. The group was repeated as necessary to form the required DATA step size. Timings and measurements are presented in Table 4.

Table 4: Stored Program Facility									
DATA step lines▶	20 Disk: 33 CPU: 5.19 Stmt/sec: 3.85			100 Disk: 129 CPU: 20.04 Stmt/sec: 4.99			300 Disk: 257 CPU: 57.60 Stmt/sec: 5.20		
▼ Dataset Size	Cmp	Un	C/U	Cmp	Un	C/U	Cmp	Un	C/U
100	2.59	8.95	.28	6.51	27.73	.23	23.83	84.24	.28
1,000	6.85	11.8	.58	24.68	44.65	.55	123.2	183.9	.67

"CPU" in column heading refers to CPU time required to create the DATA step module. "Disk" is the number of blocks (1 block = 512 bytes) required to store the module. "Stmt/sec" is the number of statements compiled per second.

Theory in this case is borne out in practice. The performance gains of the SPF over uncompiled DATA steps are uniformly and substantially greater for the 100 observation row. The improvement does not seem to be related to the number of statements contained in the compiled modules. The cost of creating and storing the SPF code, while not high, should not be disregarded: nearly a minute of CPU time and 257 blocks to create the 300 statement module. There are slight economies of scale: the number of statements compiled per second increases with program size (3.85 for 20 statements, improving to 5.20 for 300).

Indexing and Observation Selection

One of the most common tasks performed by novices and seasoned SAS programmers alike is selecting a subset of observations for processing by a procedure. We compared three subsetting methods on indexed and non-indexed datasets. Four different selection rates were used. Tables 5 and 6

compare the impact of WHERE statements used directly in a PROC, WHERE used in a DATA step followed by a PROC, and an IF statement used in a DATA step followed by a PROC. Table 5 summarizes the timings, emphasizing which dataset performed best in each method-selection rate category.

Table 5: Selecting Observations - Index Impact							
Method	% sel.	Not Indexed		Uniform Index		Non-uniform Index	
		CPU	%	CPU	%	CPU	%
WHERE (PROC)	5	1.69	2.16	* .78	1.00	1.39	1.78
	20	2.18	1.52	* 1.43	1.00	1.93	1.34
	50	2.97	1.00	* 2.96	1.00	3.13	1.05
	100	4.80	1.01	4.86	1.02	4.75	1.00
WHERE (DATA), PROC	5	2.41	1.35	* 1.78	1.00	2.40	1.34
	20	3.06	1.03	* 2.96	1.00	3.29	1.11
	50	5.23	1.03	* 5.07	1.00	5.54	1.09
	100	8.38	1.00	8.52	1.01	8.62	1.02
IF-THEN, PROC	5	3.12	1.01	3.11	1.00	3.08	1.00
	20	3.87	1.00	4.01	1.03	3.92	1.01
	50	5.58	1.00	5.58	1.00	5.53	1.00
	100	8.31	1.00	8.46	1.01	8.48	1.02
Datasets were 1,000 observations, mixed character and numeric data. "" is ratio of a method's CPU time with others in same row. The 1.69 in the first row, for example, indicates that the Not Indexed dataset took 1.69 as long as the most efficient method, the Uniform Index. The most efficient method in each row is highlighted by a shaded cell. The symbol "" indicates a test run where an index was selected for WHERE clause optimization.							

The benefits of indexing are evident even when the key is nonuniformly distributed. When 50% or less of the observations are selected indexed datasets nearly always outperform the non-indexed. The smaller the percent selected, in general, the the greater the impact of the index, particularly if it is uniformly distributed. The uniform-nonuniform difference is especially pronounced when 20% or less of the dataset is selected.

When more than 50% of the observations are selected, the overhead required to refer to the index begins to take its toll and indexed datasets reach a rough parity with the non-indexed. Interestingly, only the uniform indexes were selected for WHERE clause optimization.

Table 6 looks at the data from a different perspective. It underscores the effectiveness of WHERE clauses as a means of bypassing unnecessary input data movement. In all but the 100% selection, WHERE clause processing directly in the PROC outperformed WHERE clauses in the DATA step. IF statements as a means of data selection were the worst performers in all but three of the 36 tests. Clearly, this pre-Version 6 method of observation selection has become dated. It should be used only when other, non-selecting work needs to be performed in the DATA step prior to the procedure (e.g., creating new variables, assigning formats and labels).

Table 6: Selecting Observations - Method Comparison							
% sel.	Method	Not Indexed		Uniform Index		Non-uniform Index	
		CPU	%	CPU	%	CPU	%
5	WHERE (PROC)	1.69	1.00	*.78	1.00	1.39	1.00
	WHERE (DATA), PROC	2.41	1.42	*1.78	2.28	2.40	1.72
	IF-THEN, PROC	3.12	1.84	3.11	3.98	3.08	2.21
20	WHERE (PROC)	2.18	1.00	*1.43	1.00	1.93	1.00
	WHERE (DATA), PROC	3.06	1.40	*2.96	2.06	3.29	1.70
	IF-THEN, PROC	3.87	1.77	4.01	2.80	3.92	2.03
50	WHERE (PROC)	2.97	1.00	*2.96	1.00	3.13	1.00
	WHERE (DATA), PROC	5.23	1.76	*5.07	1.71	5.54	1.76
	IF-THEN, PROC	5.58	1.87	8.52	1.88	5.53	1.76
100	WHERE (PROC)	4.80	1.00	4.86	1.00	4.75	1.00
	WHERE (DATA), PROC	8.38	1.74	8.52	1.75	8.62	1.81
	IF-THEN, PROC	8.31	1.73	8.46	1.74	8.48	1.78
Datasets were 1,000 observations, mixed character and numeric data. "%" is a ratio of a method's CPU time with others in the same group. The 1.42 at the beginning of the second row, for example, shows that when selecting 5% of a non-indexed dataset's observations, using a WHERE statement in a DATA step followed by a PROC takes 1.42 times the CPU as the most efficient method, using a WHERE statement directly in a PROC. The most efficient method for each dataset type-% selected is highlighted by a shaded cell. The symbol "*" indicates a test whose index was selected for WHERE clause optimization.							

Conclusion

This paper has demonstrated the effectiveness of Version 6.06's performance tools. Unfortunately, it has also presented enough anomalous results to suggest that simple rules for performance tuning are not possible. We saw, for example, instances of compressed datasets being larger than their uncompressed counterparts. In several cases, use of an index actually degraded performance. The Stored Program Facility's effectiveness tails off as dataset size increases. These unexpected findings, coupled with each system's unique mixture of CPU, memory, and disk resources makes hard and fast rules a dangerous exercise.

We *can*, however, risk some generalizations. Compression is most useful when you accept default variable lengths, have a stable (write once-read many) dataset, and have a system whose disk space is at a premium. The Stored Program Facility is most effective when the DATA step has many statements and when it processes small data volumes. Indexing is most effective when working with small subsets of the data and when the index variable is uniformly distributed. The timings and measurements for the 100 observation datasets were erratic. Even if they were stable and predictable, it is unlikely that even the most effective techniques would save anything more than a trivial *absolute* amount of resources.

Many other tuning features can be evaluated at a later time. Structured Query Language (SQL) can have a great impact on resource usage. Host-specific options and engines also have the potential for great savings. Finally, traditional, low-level tuning techniques can still be applied to Version 6.06 programs. This broad array of features characterizes the new programming environment in Version 6.06: the size, quality, and complexity of the programmer's tools has increased markedly. Finding the "right" combination will be a happy, ongoing dilemma for some time to come.

APPENDIX: Production Environment

All programs were run on a Digital Equipment Corporation (DEC) 6210 running VMS Version 5.3-1. Each program was run as a separate batch job. Relevant SAS options in effect for each program were: BUFNO=1, BUFSIZE=0, CATCACHE=0, ENGINE=V606, PROCLEAVE=16384, SYSLEAVE=16384, NOUNLOAD. Dataset options unique to the VMS V606 engine were *not* altered. These include ALQ, BUFSIZE, CACHESIZ, and DEQ. CPU timings were taken from output generated by the FULLTIMER option. Disk resource consumption was identified by the VMS DIR /SIZE command.

Test dataset structures: 10 numeric default length variables (LRECL 80), 50 length 4 numeric variables (LRECL 200), and 40 length 4 numeric, 5 length 5 and 5 length 20 character (LRECL 285). All datasets had 10% missing data, and had a uniform distribution of character variable lengths. Indexes were always created on numeric variables.

Further Information

Comments on methodology and results are welcome. Persons interested in reproducing the benchmarks presented here may obtain a diskette with the macros and related programs (please specify diskette size and density). The authors may be reached at:

Family Health International
PO Box 13950
Research Triangle Park, NC 27709

(919) 544-7040 (voice)
(919) 544-7261 (fax)