Efficient SAS Coding Techniques: Measuring the Effectiveness of Intuitive Approaches

Frank C. Dilorio
Group Management Information Services, Bank of New Zealand
Wellington, New Zealand

The subject of efficiency frequently underlies discussions of SAS usage. It is, however, seldom directly addressed and even less frequently with performance figures indicating the impact of suggested techniques.

This paper outlines the benefits and occasional drawbacks of implementing efficient coding and design practices in SAS programs. A brief review of methods for dealing with SAS datasets is presented, along with an analysis of a sample series of jobs which use both -good" and "bad" techniques.

The discussion is organised into three sections. The first defines efficiency in the context of this paper. It also identifies groups in the DP community for whom economical use of resources is an issue and the circumstances in which this consideration is most needed.

The second section reviews techniques noted in such sources as SAS manuals, SUGI proceedings and "oral history". A few key points and principles are highlighted.

The last section presents a small case study using manufactured data. It investigates the effectiveness of the rules noted in the previous section's review. The study's scope is necessarily limited; emphasis is placed on common data manipulation tasks such as copying datasets, performing calculations and presenting summary statistics. Once the discussion of the study is complete, we conclude with a summary of key points and note how the results may differ from what was expected.

BACKGROUND

An elaboration of the meaning of efficiency in the context of the current discussion should be made. The most readily apparent aspect is that which is the most easily measured: economical consumption of machine resources, particularly CPU time and disk space. An efficient solution is usually thought of as one which minimizes machine use in these categories.

However, the human resource aspect of efficiency must also be considered. None of us is a stranger to the situation which arises when a program written by a long-departed programmer must be modified. If schooled in practices which emphasize economy of machine use, it is not unlikely that the program will employ "clever code", code often written at the expense of adequate documentation. Hours of analysis may be required before alteration can be attempted; frequently the most expedient solution is a total rewrite.

In the above case and many like it, it is clear that efficient use of the machine takes precedence over that of the human resource: ultimately, the efficient solution is not always effective when all factors are considered, while this paper stresses techniques for efficiency, it notes cases where such solutions may create unforeseen inefficiencies in human tens. The assumption that the best solution is the one which conserves machine resources is not made.

WHO SHOULD BE CONCERNED, AND VHT?

Efficiency is a subject of burning concern in some quarters of the dp community and only passing interest in others. This section briefly outlines the reasons why three groups of computer resource consumers have a stake in efficient and effective use of software tools. Administrators, programmers and users are discussed in turn.

DP administration has perhaps the greatest concern and the least control over the computing resource. The administrator is charged with implementing policies which minimize the impact on machine resources Hade by programmers and, particularly, users. Training prerequisites, pricing policies and the like are Methods which can be employed to prolong the life of the resource- This concern with conservation must be balanced with providing users access to tools, while not forcing them to become programmers.

A second group of dp people concerned with efficient resource use is programmers. These are the personnel charged with writing production systems and utilities. Since their programs are executed many times, they have a keen appreciation of resource-intensive coding techniques. The result is often one which leans toward ‘clever coding.’ The immediate payback of decreased demand on CPO and disk resources tends very often to overshadow the costs incurred later in the software lifecycle - costs due to inadequate documentation and hard-to-unravel coding. Here again, what is efficient in the short run may not always be effective in the long ran.

Users comprise the last group for whoa efficiency is a salient issue. Their orientation of using the machine as a leans to an end has serious implications for resource consumption. Very often the balance between efficiency and effectiveness is swung heavily in favor of the latter, when the program "works" and the output appears correct, development is essentially complete. very rarely will a program be re-written to run raster or more economically.

This indifference to efficiency can be tempered by several factors. First, cost reduction can be achieved by using sole simple techniques. Second, Information Centre consultants and other resident savants can promote the idea that efficient code is usually •ore reliable and -correct" code: it implies more reliable results coming from a good knowledge of the task at hand (the structure of the data and how it is to be manipulated). In this sense, the primary beneficiaries of efficient techniques are the end-users: the "means" will produce a less expensive and more reliable "end".

The key aspect of raising user awareness of efficiency and program design is not turning them into programmers, but giving them an appreciation of what is and is not good programming practice. As will be shown below, a few simple rules go a surprisingly long way.

WHAT CIRCUMSTANCES?

Several types of applications are suitable for application of efficient, "tuned" coding practices. Among these are:

- Large datasets (many and possibly "long" records): inefficiencies which are not obvious in small applications compound themselves and become glaring in larger ones.
- Extensive data-handlings even with relatively small datasets, numerous PBOC and DATA steps can become costly. Usually, closer examination of the problem results in a "leaner" solution.
- Utilities/production programs: time spent tuning this code results in not a one time, hot a constantly repeated, saving.
- Prototyping: although small-scale, these programs will eventually toe run in production environments, often with large datasets.
- Complex logic: cases where calculations and other assignments are critical stand to benefit from table lookup and other techniques.

There are cases, however, where a desire for efficiency can be thwarted or unnecessary:

- PROCs: The user/programmer can exert a fair amount of control and self determination within the DATA step, but frequently falls short when dealing with procedures. Since the procedures are written by highly skilled programmers this lack of control is not too critical. Generally, one can take or avoid certain options in the PROCs to obtain better performance. PROCs often exhibit an overlap of function - knowing which PROC performs the required task most efficiently is another means of obtaining efficient and effective usage.
- "One-offs": if a program is truly a oncer and does mot handle large datasets, fine-tuning is usually not warranted. This is said tentatively, since such instances may encourage sloppy habits.
- Inappropriate applications: there are some types of applications (e.g., complex on-line systems) where no SAS product may be suitable. In these cases no amount of tuning will make the program easier to use and maintain.

TECHNIQUES

User group proceedings, SAS Communications and manuals, and word of mouth are all good sources of information about tuning SAS programs. The following is a summary of what these sources have to say about handling SAS datasets.

© CodeCrafters, Inc.

Mote that system options, MACROs, manipulation of raw data and PROC-specific topics are beyond the scope of this paper. The list of techniques does not pretend to be exhaustive, but does give a fairly well-rounded view of dataset usage and creation techniques.

Some general guidelines can be outlined:

- If a PROC can do the work for you, let it do so.
- Be aware of situations where a pre-processing DATA or PBOC step can be used.
- Know how to locate Information about "non-obvious" bat valid uses of SAS options and PROCs. Even if you are a "veteran", periodically review procedure options to refresh your memory of their capabilities.
- Minimize I/O. Avoid unnecessary sorting, and minimize the number of passes made through the data.
- Use DROP and KEEP statements to reduce the amount of data handled.
- Use an adequate size subset of the data for program testing.

Guidelines specific to the DATA step include:

- Use the LENGTH statement to reduce the amount of storage taken by a variable.
- Avoid coding conventions which generate "hidden" code: character to numeric conversions (a dubious practice in itself) and LINKs are two examples of code which are resource-intensive. Frequently, there are better ways to write the program.
- Avoid "clever code": unless well-documented, it will ultimately waste more human resources than it saves machine resources.
- Minimize the number of statements executed in the DATA step. This is easily done if structured techniques are used.
- Be able to simulate the Supervisor's maintenance of the Program Data Vector (PDV) when attempting more complex DATA steps.
- Reduce Supervisor overhead: minimize the number of executions of the DATA statement as well as keeping the PDV as uncluttered possibly.
- Use character rather than numeric data whenever possible.

Items unique to PROCs include:

- mow how to choose among PROCs with overlapping features. Some do the job more efficiently than others-
- Be aware of which options in a PBOC are inappropriate on certain types of data or for larger datasets. Such information is often provided in the manuals.

STUDY RESULTS

Given the guidelines of the previous section, we can run a sample stream of jobs on a "manufactured" dataset, testing the resource savings of the techniques suggested, we also note the techniques' ease of implementation and their unforeseen drawbacks. The scope of the study is outlined, followed by a brief outline of the system environments and discussion of the results.

SCOPE AND HETFAOD

The current study used only those techniques appropriate for handling SAS datasets. While efficient processing of -raw" data is important, it was felt that in a typical application the majority of one's time and effort would be spent "massaging" and reporting from SAS datasets.

The tasks attempted reflect a variety of typical activities: a dataset is read, new variables are computed, and a subset of the observations is kept. A printed report using the dataset is produced, followed by some data reduction and sampling.

The study is confined to manipulation of a single dataset at a time. Ho merges or updates are attempted, since the time-saving principles underlying the SET statement are much the same for multiple dataset applications.

Throughout the test runs, code is written in the style of a novice SAS programmer or user. This code is then fine-tuned using the guidelines noted in the previous section. CPO and disk storage figures are compared at each level of refinement. In the DATA step examples, the different tasks (classification, calculation, subsetting, tuning) are taken in turn, with the previous task's most efficient solution being used as the basis for the next layer of complexity.

It should be emphasized that in both the DATA and PBOC steps, the results of both the novice and fine-tuned solutions are identical. Emphasis is placed on discussing why the fine-tuned solution is faster and more efficient than that of the novice. The programs were run using a generated dataset of 27 variables, two of which were to be used as sort keys and classification fie" ds in later PROCs, the rest of which were assigned random, non-missing values. Default lengths were used for all variables (see **Figure 1** for data generation program).

Trials were run in both the CMS and MVS environments (see **Figure 1** for system particulars). Dataset sizes were 1,000, 2,000, 5,000, 10,000 and 25,000 observations. MVS also used 50,000 and 100,000 observation files. These size groupings were used to trace the effect of dataset size on processing time it was thought that once the number of observations passed a certain threshold level, economies of scale might be displayed.

RESULTS DISCUSSION: GENERAL NOTE

Unless otherwise noted, all results discussed are based on MVS processing of the 100,000 record dataset. Results compare total CPU time taken for the MVS SAS command or MVS job step (i.e., time taken by SAS initialization and related system overheads is included). MVS-CMS comparisons for different size datasets are discussed briefly at the end of this section. It was felt that an exhaustive discussion of the effects of varying the different systems and data- set sizes would obscure the main intent of the current study - discussing the most efficient techniques. Finally, note that the best "tuned" dataset produced by the DATA step is the one which will be used for the various PROC step reporting, data reduction and sampling applications.

DATA REDUCTION

A series of data reduction tasks was run. In each case the desired output was a dataset containing summary statistics. Display of the statistics was not required - when the PBOC used could have done so, options were used to suppress printed listings. Only PROCs were used, since they are faster and more flexible than hand-written DATA steps.

MEANS and SUMMARY were compared for processing groups of data (see **Figure 9**). SUMMARY was clearly superior to MEANS, running in little more than half the time. This was due to the I/O intensive sort required before the MEANS procedure could be run. SUMMARY has the minor drawback of being more difficult for beginners to understand and use.

The second (and perhaps fairer) example involved summarizing the dataset with no BY-group processing or CLASS statements (see **Figure 10**). Despite the lack of the CPU-hungry SORT, MEANS still proved less efficient, running half again as long as SUMMARY.

The results indicate the need to know, from one's own base of experience as well as that of others, which PROC to use when functionality overlaps. As with the TABULATE example discussed above, part of this responsibility lies with those charged with user training.

The third data reduction example (see **Figure 11**) illustrates once again the benefit of knowing when two steps are better than one. The task was to create a dichotomous variable and use it in a subsequent SUMMARY CLASS statement. The novice approach involve! two passes through the data: a DATA step to assign values and another pass by PBOC SUMMARY. The two-pass solution is made unnecessary by implementing (yet) another "non-obvious" feature of PROC FORMAT.

A format was written which would classify values in the same way as the novice's IF-THEN statements. The SUMMARY can then use the original variable in the CLASS statement, provided a FORMAT statement is added to the SUMMARY statements. The format does group assignments on the fly, letting the task run in less than three-quarters of the two-pass solution.

This example also illustrates how SAS assists people when thinking about solutions to problems. The two-pass solution resembles how people think about a problem. Creation of a new variable was addressed first, then the problem of summarizing. Each pass through the dataset represents a distinct, separate part of the problem.

However, while it resembled thought processes and was easily translated into correct code, the two-pass solution was not the most efficient. Further, the efficient solution was fairly obscure and not explicitly documented in or suggested by any SAS publications. Features such as this, for FORMAT and other PROCs, should be documented in the manuals if users with less experience are to have a chance at taking advantage of them.

© CodeCrafters, Inc.

The last technique uses PROC FORMAT-for table lookup. This requires an added step to the program, but is worth the effort. FORHAT employs efficient binary search techniques and, since the PUT function using the format creates a character variable, storage of the created variable takes less space than the preceding techniques, which created numerics. Additionally, character data requires less handling and decoding than numeric, thus making their use in subsequent steps more efficient.

The improvement over ordered IF-THEN-ELSE statements is marginal (3%). There are two reasons for this. First, the ordered IFs had the advantage of prior knowledge of the data, and could be more finely tuned than most typical applications. Second, the number of comparisons involved in the IF-THEN sequence was not especially large (nine). Had more distinct levels been required, the inefficiency would be highlighted (even with prior knowledge of their frequency). FORMAT would not have used appreciably more time to assign many more categories.

CALCULATIONS

The next DATA step task was performing some simple calculations (see **Figure 4**). The principle applied here is reduction of Supervisor overhead. The simple approach to handling calculations is to use arrays. This method is useful if the calculations are fairly straight-forward and don't require tricky manipulation of array indices. They also tend to be easy for most people to code and maintain.

They do, however, incur overhead, a consideration which becomes particularly acute as the number of arrays and/or variables involved grows. As was the case with the IF-THEN statements, once the application grows in size the diseconomies of a technique become more pronounced and obvious.

The alternative was hard-coding. Assignments are moved out of arrays and into separate statements. Array index maintenance and other overheads are eliminated. The price of the increased execution speed (29% faster than the code using arrays) is twofold: first, if many assignments are required ("many" being largely a matter of individual tolerance and endurance) the coding is tedious.

The second drawback is not as obvious. Embedding the SET statements within a DO loop reduces Supervisor overhead, but it also places maintenance of the PDV in the hands of the user (whether the user is aware of it or not). Note that the sample program will assign a value to each variable of each observation processed – a value missing in one observation will always be replaced by another value in the next observation.

Consider, however, the results when an IF is coded without an ELSE, or some other situation which results in not all of the variables being assigned values in an observation. The value of the variable will not be set to missing and will remain unchanged in the PDV once processing for the next observation begins. The user must ensure that all values; in the PDV are reset, either by explicit assignment at the top of the DO loop or, preferably, by ensuring logical closure in the branching and conditions following the SET.

This is clearly mot palatable for novices, and should be used with care by programmers. Perhaps the best compromise would be forsaking the DO UNTIL method when numerous calculations are involved. Use of the DATA/SET method ensures reliable maintenance of the PDV. This tradeoff is an example of sacrificing some machine efficiency in favor of harder to measure, but no less valid, human factors.

SUBSETTING

The sample tasks so tar have output all observations read. Attention is now given to two methods of subsetting these records (see **Figure 5**). The novice approach is illustrated via insertion of an IF statement which must be satisfied before the observation is output. The statement is placed at the bottom of the DO loop. This placement means that the calculations are made even if the output test fails.

Minimization of the number of statements executed would be achieved if the decision to continue with the observation were made as soon as the value of the tested variable was known. The efficient solution here is to place all required statements in a DO group immediately following the assignment of the classification variable. The group, which includes the step's only OUTPUT statement, would be executed only if the class variable fell into the range required for output.

The latter technique saves 6% of the CPU time taken by the former. It is easy for users to learn and implement, and avoids the tricky and error-prone techniques noted in the Calculations section.

TUNING

Attention now turns to refining the steps made above. Three techniques using LENGTH and DROP/KEEP statements are discussed (see **Figure 6**). Unless character variable lengths are involved, this fine-tuning is best left to the latter stages of program development. These methods, it should be noted, are the first which directly address the peripheral storage aspect of the discussion.

The first technique involves setting the lengths of all variables used via a LENGTH statement. In addition to forcing more thought about the nature of the data (how large, what type, etc.), the statement makes the physical ordering of the data explicit and obvious. This ordering can be useful later in the DATA or PROC steps. Use of the LENGTH statement doubled the number of observations stored per track, from 140 to 280. This savings is not only felt at the time of dataset creation but also in subsequent uses of the data, when I/0-related figures are reduced.

Next, a DROP= option on the SET statement was added. This produced an additional 35 increase in observations per track (to 378). It is important to note that the CPU savings made by adding the DROP option are not due to a decrease in I/O performed by the system. The number of EXCPs on the input device (i.e., the test dataset) is identical with or without the DROP (this is because a block of data must be moved from peripheral storage (disk/ tape) into a buffer before the Supervisor can move data into the PDV.

What is decreased is the work done by the Supervisor, since fewer variables need to be transferred. The savings are even greater if the looping SET technique is not used and the PDV contents must be set to missing: the fewer variables to be reset, the less work that must be done by SAS.

The third refinement was use of a DROP statement, eliminating variables not needed in the output dataset. In this example, variables used in the SET statement were used for calculations and then discarded - only classification and computed variables were kept.

The combined effect of all three techniques is dramatic. without any restrictions on length or presence of variables in the output dataset, 196 observations are stored per track. By using LENGTH statements to reduce the default 8-byte numerics to a more realistic 4 bytes and by restricting the dataset to only those variables which were to be used in later steps, the dataset contained 791 observations per track, more than a four-told increase. It is important to realize that I/0-related CPU time will be reduced every time the dataset is used, not just in this one step.

DATA STEP SUMMARY

A comparison of two DATA steps, using the extremes of efficiency and inefficiency was made (see **Figure 7**). The "best case" DATA took only 43K of the "worst case". Disk storage efficiency was increased dramatically as well. The comparison highlights the cumulative effects of the individual techniques.

REPORTING

Potentially a study in itself, we are limited here to a single instance of PBOC use to produce a report summarizing data and producing presentation-quality output (see **Figure 8**).

While not particularly "novice-friendly", TABULATE was used as the procedure to display and summarize data: it combines reduction and display functions, thus requiring only a single pass through the data.

The contrasting, efficient technique was use of PBOC SUMMARY prior to TABULATE to do the data reduction "grunt work", effectively using TABULATE to dress up and print the summary data. The CPU savings were substantial; 89% when processing 100,000 observations and only slightly less dramatic with smaller datasets.

The problem with this technique is that it is not obvious. Information Centre consultants and others involved with training need to provide information about these techniques. Two PROCs being more efficient than one is not intuitive and consequently requires extra effort to make known.

DATASET COPY

The first data manipulation task was a copy of the test dataset (sea **Figure 2**). Two variations of the DATA step and one PROC are compared. The first DATA step used the novice's approach: a SET statement immediately after the DATA. It is simple and quick to code, and ensures that the SET variables will be reset to missing each time DATA is executed. This resetting to missing incurs Supervisor

overhead and becomes increasingly significant as the number of variables in the SET dataset increases.

Be reduce Supervisor overhead and achieve a 13% savings in CPU by enclosing the SET within a DO loop. The DATA statement is executed only once, thus reducing the work done by the Supervisor. This method requires a little extra coding and has the disadvantage of being deceptively tricky if calculations are required (this point is discussed more fully in the "Calculations" section below).

Preference for PROCs to "roll your own" code was the reason for the inclusion of the PROC COPY example. The savings compared to the DATA steps are significant: 35 faster than the DATA/SET method. Selection of particular observations or variables is, however, not possible, so further DATA examples will build on the DO UNTIL method.

CLASSIFICATION

While calculations and classification are two variations of assignments, they are handled in separate sections. This is because the function served by classification variables is that of CLASS or SORT fields in PBOCs. They are not used in any arithmetic operations. This characteristic gives the programmer more latitude than normal when creating them.

The techniques used are three variations of IF-THEN statements and use of PBOC FORMAT for table lookup (see **Figure 3**). The novice approach is a simple sequence of IF-THEN statements. The rule of minimization of executable statements is violated here because every IF statement is executed in every case: even if the value is assigned by the third IF, the six remaining IFs would still be (needlessly) executed.

This wastage is reduced by the inclusion of ELSE statements. The result is a 12% reduction in CPU time, since once a class variable assignment is successfully made, subsequent comparisons are not attempted.

This technique is taken to its logical extreme by reordering the statements by their expected frequency, we may know (from prior experience or an educated guess) that values will occur most frequently close to 0. Placing statements assigning codes to these groupings at the beginning of the IF-THEN-ELSE sequence will result in the quickest possible "hit", thus saving CPO time. The chart shows execution in 81% of the time taken by straight IF-THENs (compared to the 88% achieved by unordered IF-THEN- ELSE).

SAMPLING

The final test addressed an I/o-bound operation, and highlights the effectiveness of an option not yet discussed here. The application involved taking a 20 uniform sample of the dataset (see **Figure 12**), The first method read a record and decided if it was eligible for output. Only one out of every five records was actually needed, so this method, while straightforward, wastes considerable amounts of resources on unnecessary I/o-related chores.

The alternative was looping through the dataset in increments of 5 (to yield a 20 selection), directly accessing the record number indicated by the DO loop index. In this way, only records that were to be in the output dataset were read from the input dataset. CPO consumption was reduced by 5% (a figure which would increase as the percent selected decreased).

CMS—MVS COMPARISON

The discussion has concentrated on MVS statistics, primarily on applications using the full 100,000 observations in the sample dataset. Attention now turns briefly to comparing MVS results with those in the CMS environment.

This is done for two reasons. First, CMS has a perhaps undeserved reputation for not being as "tuneable" as MVS. while we cannot fairly compare absolute performance for each task, we can look at relative performance: how do the two environments compare for each task's most efficient method? Do we have the same economies of scale and trends in improvement when we start using larger datasets?

The second reason for the comparison is for the benefit of users who have access to both environments. One might want to do program development under CHS and run production within MVS. Some reassurance that what is efficient in one environment will be so in another would be helpful.

The comparison of best methods reveals several interesting features. First, in most cases at 25,000 observations (the largest comparable size) the difference between the two is negligible. The largest gaps occurred in the best-worst DATA and report-writing tasks, which suggests that CMS may be more vulnerable to I/O problems than MVS. Otherwise the differences were minimal. Second, size comparisons leading up to the 25,000 observation level were often remarkably similar.

This similarity in patterns answers the second part of the questions posed above: uniformly, what is good practice in CMS will also be good practice in MVS (this is what you would expect, but it's always nice to see numbers to back you up).

SUMMARY/CONCLUSION

SUMMARY

Key points of the above discussion can now be summarized. First, it is clear that "traditional", well-documented approaches to efficient SAS usage are, in fact, borne out by the numbers in the current study. LENGTH, DROP/KEEP statements, using PROCs whomever possible, and so on do produce results which are significantly less resource-intensive.

Second, it is also clear that even more pronounced savings can be achieved with relatively little extra coding effort by "not obvious" (but legitimate) use of other techniques. Formats used for table lookup, pre-processing data with SUMMARY, knowing SUMMARY is more efficient than MEANS and the like are all resource savers, but are difficult to teach and not well-documented.

Third, the single best step toward reducing resource consumption is reducing I/O: unnecessary sorts and unneeded or excessively long variables are all easy targets for tuning, Fourth, this I/O reduction can be achieved not only via careful selection of PROCs, but in the DATA step as well.

Fifth, it was shown (in **Figure 13**) that most savings (on a CPU percent reduction basis) will be realized by 5,000 observations.

Finally, for most tasks, it was demonstrated that MVS and CMS are roughly comparable in CPU reduction, with MVS being somewhat •ore responsive to efficiency techniques than was CMS.

CONCLUSION

The effectiveness of both traditional and obscure techniques in saving machine resources points to the need for Information Center and other dp departments to educate the user community. If training is done properly, documentation and instruction for novice users in a few simple to learn and implement techniques can result in significant savings.

From the professional programmer's standpoint, it should be noted that the methods described here and elsewhere in the literature provide? often significant savings without the hidden costs associated with tricky, unmaintainable code- For both this group and users, the "Time" in "SAS Saves Time" should thought of in terms of both machine and human terms.

© CodeCrafters, Inc.

FIGURE 1: Dataset Creation/System Description

Data Creation

```
DATA ORIGINAL.DATASET;
ARRAY V V1-V25;
DO I = 1 TO 100000;
   NORM = ROUND(NORMAL(1234567)..2);
   DICHOT = MOD(I,2);
   DO OVER V;
      V = UNIFORM(0);
   END;
   OUTPUT;
END;
DROP I;
```

System Environment

Feature	MVS	CMS
Hardware	3081-G	3083-EX
Op. Sys.	XA	SP/HPO 3.2
Sort	SYNCSORT	SYNCSORT
WORK Disk	3380	3380
User disk	3380	3380
SAS Release	82.4	82.3
Dset. Alloc.	Contig. Cyl.	Contig. Cyl.

FIGURE 2: Copy Dataset

----- SET -----;

```
DATA ONE;
SET ORIGINAL.DATASET;
```

----- DO UNTIL -----;

```
DATA ONE;
EOD = 0 ;
DO UNTIL (EOD);
   SET ORIGINAL.DATASET END=EOD;
   OUTPUT;
END;
```

----- PROC COPY -----;

```
PROC COPY IN=INSUB OUT=WORK;
```

Comparison of Methods: MVS, 100k Observations

	CPU: % of Slowest Method
SET	100
DO UNTIL	87
PROC COPY	65

FIGURE 3: Classification

```
*----- IF-THEN -----*;
DATA ONE;
EOD = 0;
DO UNTIL (EOD);
   SET ORIGINAL.DATASET END=EOD;
   IF       NORM(=-2.2  THEN CLASSVAR=1;
   IF (-2.2(NORM(=-1.6) THEN CLASSVAR=2;
   IF (-1.6(NORM(=-1.0) THEN CLASSVAR=3;
   IF (-1.0(NORM(=-0.4) THEN CLASSVAR=4;
   IF (-0.4(NORM(= 0.2) THEN CLASSVAR=5;
   IF ( 0.2(NORM(= 0.8) THEN CLASSVAR=6;
   IF ( 0.8(NORM(= 1.4) THEN CLASSVAR=7;
   IF ( 1.4(NORM(= 2.0) THEN CLASSVAR=8;
   IF NORM ) 2.0        THEN CLASSVAR=9;
   OUTPUT;
END;

*----- IF-THEN-ELSE, expected Frequency -----*;
DATA ONE;
EOD = 0;
DO UNTIL (EOD);
   SET ORIGINAL.DATASET END=EOD;
   IF          (-0.4(NORM(= 0.2) THEN CLASSVAR=5;
      ELSE IF (-1.0(NORM(=-0.4) THEN CLASSVAR=4;
      ELSE IF ( 0.2(NORM(= 0.8) THEN CLASSVAR=6;
      ELSE IF (-1.6(NORM(=-1.0) THEN CLASSVAR=3;
      ELSE IF ( 0.8(NORM(= 1.4) THEN CLASSVAR=7;
      ELSE IF (-2.2(NORM(=-1.6) THEN CLASSVAR=2;
      ELSE IF ( 1.4(NORM(= 2.0) THEN CLASSVAR=8;
      ELSE IF NORM(=-2.2        THEN CLASSVAR=1;
      ELSE IF NORM ) 2.0        THEN CLASSVAR=9;
   OUTPUT;
END;
```

```
*----- IF-THEN-ELSE -----*;
DATA ONE;
EOD = 0;
DO UNTIL (EOD);
   SET ORIGINAL.DATASET END=EOD;
 IF                NORM(=-2.2  THEN CLASSVAR=1;
   ELSE IF (-2.2(NORM(=-1.6) THEN CLASSVAR=2;
   ELSE IF (-1.6(NORM(=-1.0) THEN CLASSVAR=3;
   ELSE IF (-1.0(NORM(=-0.4) THEN CLASSVAR=4;
   ELSE IF (-0.4(NORM(= 0.2) THEN CLASSVAR=5;
   ELSE IF ( 0.2(NORM(= 0.8) THEN CLASSVAR=6;
   ELSE IF ( 0.8(NORM(= 1.4) THEN CLASSVAR=7;
   ELSE IF ( 1.4(NORM(= 2.0) THEN CLASSVAR=8;
   ELSE IF NORM ) 20         THEN CLASSVAR=9;
     OUTPUT;
END;

*----- Table lookup -----*;
PROC FORMAT DDNAME=SASLIB;
VALUE NORMGRP
      LOW   - -2.2  = '1'  -2.2  - -1.6  = '2'
      -1.6  - -1.0  = '3'  -1.0  - -0.4  = '4'
      -0.4  -  0.2  = '5'   0.2  -  0.8  = '6'
       0.8  -  1.4  = '7'   1.4  -  2.0  = '8'
       2.0  - HIGH  = '9'
      ;

DATA ONE;
EOD = 0;
DO UNTIL (EOD);
   SET ORIGINAL.DATASET END=EOD;
   CLASSVAR = PUT(NORM.NORMGRP1.);
   OUTPUT;
END;
```

```
                                                    CPU: % of
            Comparison of Methods: MVS, 100k Observations   Slowest
                                                             Method
            |
IF-THEN     |**************************************************      1 0
IF...ELSE   |******************************************              88
IF.../freq  |*************************************                  81
Table lookup|*********************************                     78
            -----+----+----+----+----+----+----+----+----+----+
                 10   20   30   40   50   60   70   80   90  100
```

FIGURE 4: Calculation

```
*----- Arrays -----*;
DATA ONE;
EOD = 0;
DO UNTIL (EOD);
   SET ORIGINAL.DATASET END=EOD;
   CLASSVAR = PUT(NORM.NORMGRP1.);
   IN1   V1  - V5   ;
   IN2   V6  - V10  ;
   IN3   V11 - V15  ;
   CALC  CV1 - CV5  ;
   CALC2 CV6 - CV10 ;
   IF CLASSVAR(=5 THEN DO OVER CALC;
                          CALC = IN1 * .5;
                       END;
      ELSE DO OVER CALC;
             CALC = IN2 * .75;
           END;
   DO OVER CALC2;
      CALC2 = IN3 * 1.5;
   END;
END;
```

```
*----- Explicit -----*;
DATA ONE;
EOD=0;
DO UNTIL (EOD);
   SET ORIGINAL.DATASET END=EOD;
   CLASSVAR = PUT(NORM.NORMGRP1.);
   IF CLASSVAR (= '5' THEN DO;
      CV1 = V1 * .5 ;
      ...
      CV5 = V5 * .5 ;
      END;
      ELSE DO;
           CV1 = C6  * .75 ;
           ...
           CV5 = C10 * .75 ;
         END;
   CV6  = V11 * 1.5 ;
   ...
   CV10 = V15 * 1.5 ;
   OUTPUT;
END;
```

```
                                                    CPU: % of
            Comparison of Methods: MVS, 100k Observations   Slowest
                                                             Method
            |
Arrays      |**************************************************      100
Explicit    |**********************************                      71
            -----+----+----+----+----+----+----+----+----+----+
                 10   20   30   40   50   60   70   80   90  100
```

FIGURE 5: Subset

```
*----- Delayed -----*;
DATA ONE;
EOD=0;
DO UNTIL (EOD);
   SET ORIGINAL.DATASET END=EOD;
   CLASSVAR = PUT(NOR.NORMGRP1.);
   IF CLASSVAR(='5' THEN DO;
      CV1 = V1 * .5 ;
      ...
      CV5 = V5 * .5 ;
      END;
      ELSE DO;
           CV1 = C6  * .75 ;
           ...
           CV5 = C10 * .75 ;
           END;
   CV6  = V11 * 1.5 ;
   ...
   CV10 = V15 * 1.5 ;
   IF CLASSVAR(='5' THEN OUTPUT;
END;
```

```
*----- Immediate -----*;
DATA ONE;
EOD=0;
DO UNTIL (EOD);
   SET ORIGINAL.DATASET ENI=EOD;
   CLASSVAR = PUT(NORM.NORMGRP1.);
   IF CLASSVAR(='5' THEN DO;
      CV1  = V1 * .5 ;
      ...
      CV5  = V5 * .5 ;
      CV6  = V11 * 1.5;
      ...
      CV10 = V15 * 1.5;
      OUTPUT;
      END;
END;
```

Comparison of Methods: MVS, 100k Observations

	CPU: % of Slowest Method
Delayed	100
Immediate	94

10 20 30 40 50 60 70 80 90 100

FIGURE 6: Tuning

```
*----- LENGTH -----*;
DATA ONE;
LENGTH V1-V15 CV1-CV10 4 DICHOT 2 CLASSVAR $2 ;
EOD=0;
DO UNTIL (EOD);
   SET ORIGINAL.DATASET END=EOD;
   CLASSVAR = PUT(NORM.NORMGRP1.);
   IF CLASSVAR(='5' THEN DO;
      CV1  = V1 * .5 ;
      ...
      CV5  = V5 * .5 ;
      CV6  = V11 * 1.5;
      ...
      CV10 = V15 * 1.5;
      OUTPUT;
      END;
END;
```

```
*----- LENGTH + SET(DR  = ) -----*;
DATA ONE;
LENGTH V1-V15 CV1-CV10 4 DICHOT 2 CLASSVAR $2 ;
EOD=0;
DO UNTIL (EOD);
   SET ORIGINAL.DATASET(DROP=V6-V10 V16-V25) END=EOD;
   CLASSVAR = PUT(NORM.NORMGRP1.);
   IF CLASSVAR(='5' THEN DO;
      CV1  = V1 * .5 ;
      ...
      CV5  = V5 * .5 ;
      CV6  = V11 * 1.5;
      ...
      CV10 = V15 * 1.5;
      OUTPUT;
      END;
END;
```

```
*----- LENGTH + SET(DROP= ) + DROP stmt. -----*;
DATA ONE;
LENGTH V1-V15 CV1-CV10 4 DICHOT 2 CLASSVAR $2 ;
EOD=0;
DO UNTIL (EOD);
   SET ORIGINAL.DATASET(DROP=V6-V10 V16-V25) END=EOD;
   CLASSVAR = PUT(NORM.NORMGRP1.);
   IF CLASSVAR(='5' THEN DO;
      CV1  = V1 * .5 ;
      ...
      CV5  = V5 * .5 ;
      CV6  = V11 * 1.5;
      ...
      CV10 = V15 * 1.5;
      OUTPUT;
      END;
END;
DROP V1-V5 V11-V15;
```

Comparison of Methods: MVS, 100k Observations

	CPU: % of Fastest Subsetting (Figure 5)
LENGTH	92
+ SET(DROP=)	87
+ DROP stmt.	85

10 20 30 40 50 60 70 80 90 100

FIGURE 7: "Best"/"Worst" DATA

```
*----- Least Efficient DATA -----*;
DATA TUNED.DATASET;
SET ORIGINAL.DATASET;
ARRAY IN1    V1  - V5   ;
ARRAY IN2    V6  - V10  ;
ARRAY IN3    V11 - V15  ;
ARRAY CALC   CV1 - CV5  ;
ARRAY CALC2 CV6 - CV10 ;
IF NORM(=-2.2           THEN CLASSVAR = 1;
IF (-2.2(NORM(=-1.6) THEN CLASSVAR = 2;
IF (-1.6(NORM(=-1.0) THEN CLASSVAR = 3;
IF (-1.0(NORM(=-0.4) THEN CLASSVAR = 4;
IF (-0.4(NORM(= 0.2) THEN CLASSVAR = 5;
IF ( 0.2(NORM(= 0.8) THEN CLASSVAR = 6;
IF ( 0.8(NORM(= 1.4) THEN CLASSVAR = 7;
IF ( 1.4(NORM(= 2.0) THEN CLASSVAR = 8;
IF NORM ) 2.0            THEN CLASSVAR = 9;
IF CLASSVAR(=5 THEN DO OVER CALC;
                        CALC = IN1 * .5;
                     END;
   ELSE DO OVER CALC;
           CALC = IN2 * .75;
        END;
DO OVER CALC2;
   CALC2 = IN3 * 1.5;
END;
IF CLASSVAR(=5 THEN OUTPUT;
```

```
*----- Most Efficient DATA -----*;
PROC FORMAT  ;
VALUE NORMGRP (DEFAULT=1)
      LOW     - -2.2 = '1'
      -2.2    - -1.6 = '2'
      -1.6    - -1.0 = '3'
      -1.0    - -0.4 = '4'
      -0.4    -  0.2 = '5'
       0.2    -  0.8 = '6'
       0.8    -  1.4 = '7'
       1.4    -  2.0 = '8'
       2.0    - HIGH = '9'
       ;
RUN;

DATA TUNED.DATASET;
LENGTH V1-V15 CV1-CV10 4 DICHOT 2 CLASSVAR 4
EOD=0;
DO UNTIL (EOD);
   SET ORIGINAL.DATASET(DROP=V16-V25) END=EO
   CLASSVAR = PUT(NORM,NORMGRP1.);
   IF CLASSVAR (= '5' THEN DO;
      CV1  = V1   *    .5;
      ...
      CV5  = V5   *    .5;
      CV6  = V11  *   1.5;
      ...
      CV10 = V15  *   1.5;
      OUTPUT;
      END;
END;
DROP V1-V15;
```

```
            Comparison of Methods: MVS, 100k Observations        CPU: % of Slowest Method

Slowest     |**************************************************        100
Fastest     |**********                                                 43
            -----+----+----+----+----+----+----+----+----+----+
                 10   20   30   40   50   60   70   80   90  100
```

FIGURE 8: Tables

```
*----- PROC TABULATE -----*;
PROC TABULATE DATA=TUNED.DATASET;
CLASS DICHOT CLASSVAR;
VAR CV1-CV10;
TABLES CLASSVAR ALL,
       DICHOT*(CV1 CV2 CV3 CV4 CV5 CV6
               CV7 CV8 CV9 CV10)*(F=5.0)
       / RTS=10;
```

```
*----- SUMMARY/TABULATE -----*;
PROC SUMMARY DATA=TUNED.DATASET NWAY;
CLASS DICHOT CLASSVAR;
VAR CV1-CV10;
OUTPUT OUT=T(DROP= FREQ   TYPE ) SUM=;
RUN;

PROC TABULATE;
CLASS DICHOT CLASSVAR;
VAR CV1-CV10;
TABLES CLASSVAR ALL,
       DICHOT*(CV1 CV2 CV3 CV4 CV5 CV6
               CV7 CV8 CV9 CV10)*(F=5.0)
       / RTS=10;
```

```
            Comparison of Methods: MVS, 100k Observations        CPU: % of Slowest Method

TABULATE    |**************************************************        100
SUMM./TAB.  |*****                                                      11
            -----+----+----+----+----+----+----+----+----+----+
                 10   20   30   40   50   60   70   80   90  100
```

FIGURE 9: Reduction I - By Groups

```
*----- PROC MEANS -----*;
PROC SORT DATA=TUNED.DATASET OUT=T;
BY DICHOT CLASSVAR;

PROC MEANS NOPRINT;
VAR CV1-CV10;
OUTPUT SUM=SUMCV1-SUMCV10 N=NCV1-NCV10;
BY DICHOT CLASSVAR;
```

```
*----- PROC SUMMARY -----*;
PROC SUMMARY NWAY DATA=TUNED.DATASET;
VAR CV1-CV10;
CLASS DICHOT CLASSVAR;
OUTPUT SUM=SUMCV1-SUMCV10 N=NCV1-NCV10;
```

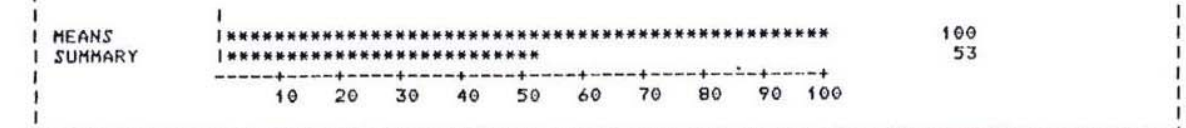


FIGURE 10: Reduction II - Entire File

```
*----- PROC MEANS -----*;

PROC MEANS NOPRINT DATA=TUNED.DATASET;
VAR CV1-CV10;
OUTPUT SUM=SUMCV1-SUMCV10 N=NCV1-NCV10;
```

```
*----- PROC SUMMARY -----*;

PROC SUMMARY NWAY DATA=TUNED.DATASET;
VAR CV1-CV10;
OUTPUT SUM=SUMCV1-SUMCV10 N=NCV1-NCV10;
```

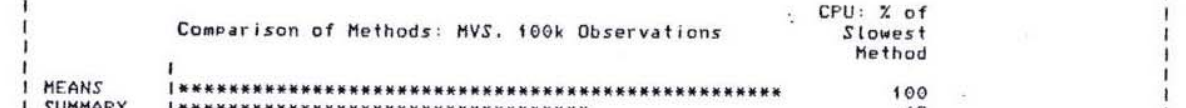


FIGURE 11: Reduction III - Entire File, Using Calculated Variable

```
*----- DATA/SUMMARY -----*;
DATA TEMP;
EOD=0;
DO UNTIL (EOD);
   SET TUNED.DATASET(KEEP=NORM CV1-CV10) END=EOD;
   IF NORM(=0 THEN ANVAR = 0;
      ELSE         ANVAR = 1;
   OUTPUT;
END;

PROC SUMMARY NWAY;
VAR CV1-CV10;
CLASS ANVAR;
OUTPUT SUM=;
```

```
*----- FORMAT/SUMMARY -----*;
PROC FORMAT DDNAME=SASLIB;
VALUE GROUP LOW - 0    = 1
            0  - HIGH = 2;
RUN;

PROC SUMMARY NWAY DATA=TUNED.DATASET;
VAR CV1-CV10;
CLASS NORM ;
OUTPUT SUM=;
FORMAT NORM  GROUP1.;
```

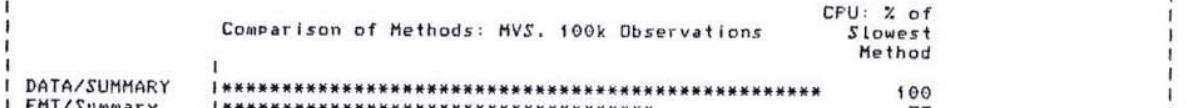


FIGURE 12: Uniform Sampling

```
*----- Read, Select -----*;
DATA TEMP;
EOD=0;
N  =0;
DO UNTIL (EOD);
   SET TUNED.DATASET END=EOD;
   N = N + 1;
   IF MOD(N,5) = 0 THEN OUTPUT;
END;
DROP N;
```

```
*----- Direct Access Using POINT= -----*;
DATA TEMP;
DO I = 1 TO 25000 BY 5;
   SET TUNED.DATASET POINT=I;
      IF _ERROR_ = 1 THEN STOP;
   OUTPUT;
END;
```

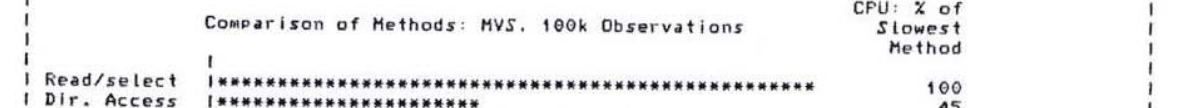
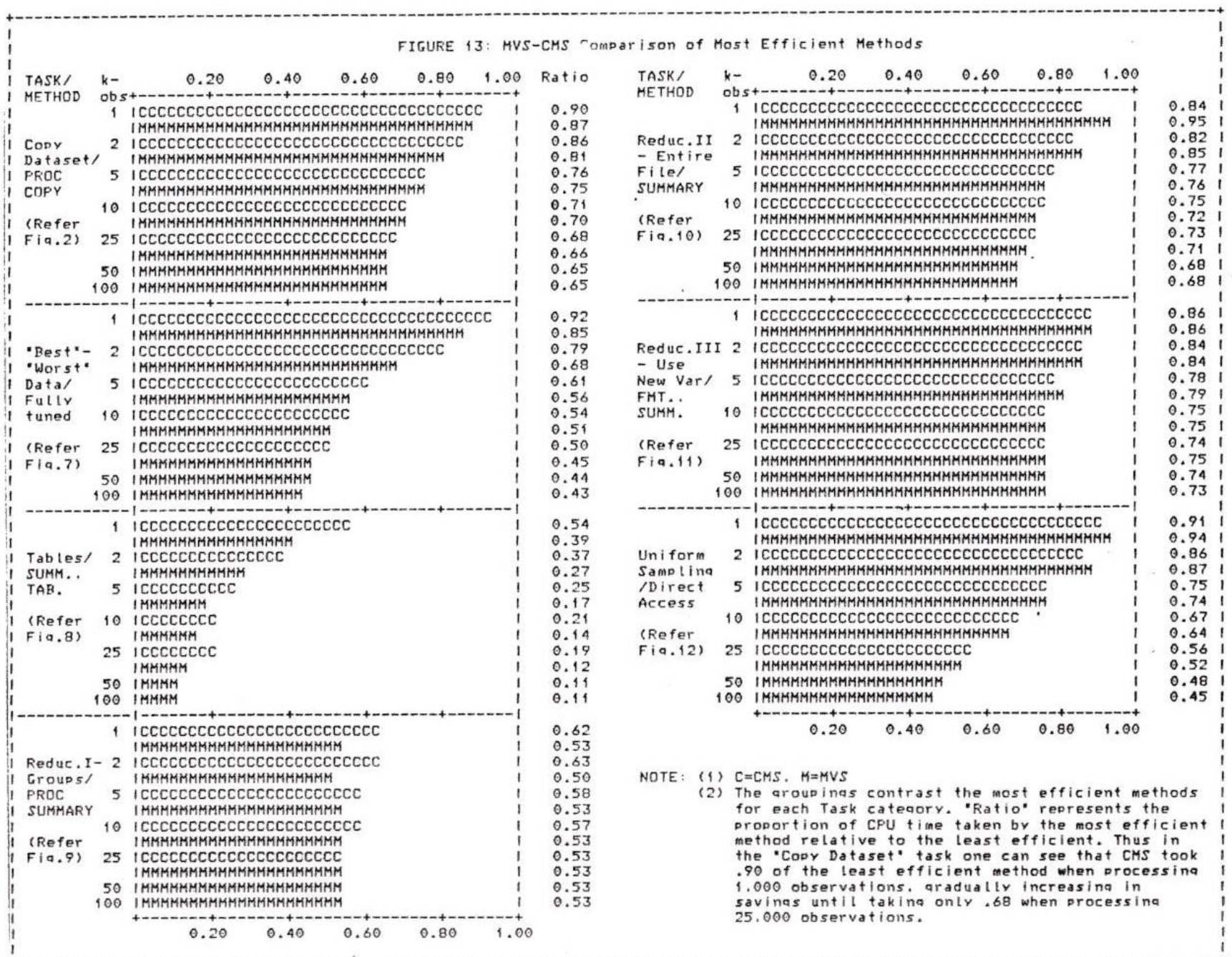


FIGURE 13: MVS-CMS Comparison of Most Efficient Methods



NOTE: (1) C=CMS, M=MVS

(2) The groupings contrast the most efficient methods for each Task category. 'Ratio' represents the proportion of CPU time taken by the most efficient method relative to the least efficient. Thus in the 'Copy Dataset' task one can see that CMS took .90 of the least efficient method when processing 1,000 observations, gradually increasing in savings until taking only .68 when processing 25,000 observations.

© CodeCrafters, Inc.