

DOES PROGRAM EFFICIENCY REALLY MATTER ANYMORE?

SIGURD W. HERMANSEN, WESTAT, ROCKVILLE, MD

Faster computer platforms plus cheaper runtime and I/O charges have changed the trade-off between computer charges and programmer hours, but have not made good programming obsolete. Here we describe some traps in SAS® procedural and 4GL programs, and we illustrate some methods that will help programmers avoid escalating computer charges, maxing out disk space, and ugly abends. The presentation includes examples of seemingly benign programs that will, under the right conditions, cause very large systems to crash. As an antidote, it will also include examples of programs that will process millions of observations relatively efficiently.

I. WHOSE TIME MATTERS?

Most experienced programmers spend at least some time and effort trying to make programs more efficient. Talk to a programmer and he or she will tell you how after spending a few hours reworking a program it runs faster, performs fewer I/O's, uses less disk space, and generally improves the well-being of humanity. We have to ask "Whose time matters? Does it make sense to spend your valuable time to save program execution time and disk space?" No one really knows how to answer these questions, but knowing at bit more about program efficiency will help you deal with those who think they do.

Efficiency through Better Programming?

After considering a wide range of factors affecting the efficiency of application systems, we find that few individual attempts to improve the efficiency of programs produce much benefit, and that the benefits often fall short of losses due to labor cost overruns, delayed deliveries of programs or their products, or errors introduced during reprogramming. Computer systems do not in general suffer from heavy use. Real inefficiencies occur more often when programs

fail, or even worse overwhelm a system and interfere with other users.

Even when program execution time on a given platform or disk space requirements do begin to matter, upgrading the platform or shifting part of the workload to another platform may prove a better solution. At current incremental costs of around \$10.00 per MHz of CPU speed, \$40.00 per MB of RAM, and 30¢ per MB of disk space, it does not take much additional programmer time to exceed the cost of doubling the speed and capacity of a PC. Similar trade-offs hold true for larger systems. Real gains in efficiency occur when platforms and programming methods suit both the users' requirements for application systems and the skills of the system developers. For a simple analogy, imagine a groundskeeper deciding to buy one or more lawnmowers and hiring one or more persons to operate them. Finding the right size and power of mower and matching it to the skills of the person hired to operate it will do more to save time and money than individual operators' attempts to make better use of a mower of the wrong size or type for the job. To achieve true efficiency, system managers and application programmers need to find the right balance of platform, methods, and people.

Do Variable Computer Use Charges Promote Program Efficiency?

Organizations typically recover computer system and technical support costs by charging users for CPU cycles, I/O's, disk space, and other resources used per month. System managers fret over what must seem to them the programmers' insatiable hunger for faster and bigger systems; they use, we suspect, the threat of higher computer charges to make users think twice before demanding new equipment and other systems components. They set charge rates at levels that they expect will let them offset the costs of acquiring and

installing system components, plus pay overhead for existing facilities and technical support. The charges have little to do with promoting efficient use of computer resources after installation.

For computing resources already in place, use charges will promote efficient use of resources only when they discourage uses of the system that would impede other users from making better use of the system. This can only happen when the system is operating close to capacity. Charging in proportion to the levels of resources used makes no economic sense except when the combined demand for the resources would otherwise exceed capacity or significantly degrade performance.

Programmers who measure gains in program efficiency by decreases in billed computer charges may pay for these paper savings with higher labor charges and delayed deliveries. The mere threat of overrunning the computer budget may force programmers to spend many precious hours monitoring automated processes that might work just as well unattended, and to avoid methods that require much computer time and disk space to develop. When operating under typical rate schedules for computer use, programmers tend to worry about program efficiency at a level well past the point at which it matters. They fail to take full advantage of the computing capacity available to them.

The Effect of Eliminating Computer Use Charges on Program Efficiency

The outcome of a little natural experiment provides some insight into how efficiently, in the absence of use charges, a group of programmers and statistical analysts would use a fairly large system. After working for a few years on computer systems with CPU use, I/O, and disk space charges, our small group of programmers and analysts obtained concurrent access as LAN clients to a RISC

server with a performance and capacity rating close to that of the original system. We agreed to a fixed monthly charge for use of the server. Use charges would no longer constrain the programmers' and analysts' use of the system. The system provided no disk back-up, and minimal support, maintenance, or training services. Benchmarking and basic testing of the server ended around the beginning of 1995. The group started using the server shortly thereafter. By that time, many of the programmers already had a year or more of on-the-job training in effective methods for intensive analysis of large databases.

After looking at the Unix SAS benchmarks for the server and surveying the vast 3 GB expanse of empty disk space assigned to applications, we urged the other programmers and analysts to forget the old rules of programming efficiency and treat CPU usage and disk space as free resources. We even issued a challenge: try any program you think will get the job done sooner and don't worry about overwhelming the system.

Six months later we have heard virtually no complaints about system slowdowns and had only four crises that required system administrators to intervene. While we would have preferred to avoid altogether any need for intervention by system administrators, less than one instance of down time per month compares very favorably with other multi-user systems. These results tell us that revolving groups of seven or so active users competing for the same computing resources seldom push the system to its limits. Combined with usage statistics for the server, they suggest that even though we repeatedly skim through millions of observations and match them to other millions of observations, and we frequently run very complex statistical procedures on large sets of data, we are not yet making full use of a free resource.

A closer look at the four system crises requiring intervention tells us more. In each case the program that precipitated the crisis contained one or more of those obvious traps that can drag down a system of any capacity. The method used clearly did not fit the scale of the application. Heavy use of computer resources by many users did not overwhelm the system. It took only one program to push the system past its limits.

Our experience suggests that reasonably good programming techniques applied intensively to the processing of large databases rarely test the limits of a computer system. Instead, computer systems stall or get pushed beyond their limits when programmers inadvertently fall into obvious traps.

Whose time really matters? We see no reason to try to refine programs to reduce charges for CPU cycles, I/O's, or disk space. Except under special peak-load conditions, attempts to reduce computer use charges seem more likely to waste computer capacity than to conserve it for better purposes.

Not long ago, multi-user computer systems cost fifty times more than the annual salary of senior programmer. Today, that ratio has decreased by a factor of at least fifty; hence, each hour of programmer time devoted to making programs more efficient has to yield fifty times the savings it once did to prove cost effective. Conserving programmers' and clients' time certainly matters more than conserving computing time.

Program efficiency still matters, but only in the appropriate context. The time required to train programmers and analysts in methods that will help them avoid obvious and disastrous traps matters a great deal. The time required to research, develop, and test new methods matters as well. We have identified some programming traps to avoid and explored some methods that could improve

the productivity of systems and programmers. In the next sections we look at details and programs that may have some value to programmers and developers of application systems.

II. TRAPS

A few traps will drag a system down, no matter how large its capacity. We have a special concern when those traps lead to the need for system administrators to intervene, or they use up system capacity to the extent that impedes the work of other users.

Most of the traps fall naturally into three classes. We recognize them by their effects.

- **Exploding demands on file systems** cause excessive shuffling of data to and from disk, and may eventually exhaust the capacity of the file system.
- **Shells** trap automated processes in little whirlpools of wasted CPU cycles and I/O's.
- **Mixed signals** lead to confusing results and errors

Exploding Demands on File Systems

Data management programs based on fixed column file formats, such as SAS or commercial RDBMS's, make it particularly easy to trigger explosive growth of requirements for disk space. Simple arithmetic tells us that simultaneous doubling of the total width of columns (variables) and the number of rows (observations) will quadruple the dimensions of a data table [in more general terms, $m(\text{\#columns}) n(\text{\#rows}) = mn (\text{\#columns})(\text{\#rows})$]. The expansion factor mn in this basic example plays an important role in matching application requirements to methods.

In applications that do not involve table joins (merges), adding new columns presents the greater danger. Adding columns to a table that contains a large number of rows will progressively multiply the size of a file that will hold it. We can do just that with this short SAS program:

```

DISKCRASH sas test program: SWH 6/23/95 version
%macro (Otrap(iobs=1,i1st=1,inth=1,n_out=1);
data;
  x1=0;
  do j=1 to &iobs;
    output;
  end;
  drop j;
run;
%do %while(1);
  data;
    length i | x1-x&inth 8;
    set;
    array x(&i1st-&inth) x&i1st-x&inth;
    do i= &i1st to &inth;
      x[i]=0;
    end;
    do j=1 to &n_out;
      output;
    end;
    drop j;
  run;
  %let i1st=%eval(&inth + 1);
  %let inth=%eval(&inth + (2 * &inth));
%end;
%mend (Otrap;

%|Otrap(iobs=10000,i1st=2,inth=2,n_out=1)

```

If we set the number of rows in the initial work dataset (argument named `iobs=`) to a large enough number and then expand the number of columns, the program quickly fills up all available disk space and terminates with an "out of resources" error. It may lock up all other users of the file system as well. Note that the trap does not require macros and arrays of variables; they merely make it easier to blow up a file system!

Starting with a fairly small number of rows, the program will likely reach the system limit for the total length of a row in a SAS dataset, and either stop or continue running under `obs=0`, depending on the options in effect. It will probably not have much effect on any other users.

Other arguments of the program test the effects of adding both columns and multiple rows (`inth=,n_out=`). You can use this handy program to test your system. When it blows up, you know you have reached its capacity. Or you could sample the amount of available disk space remaining during periods of heavy use and select the minimum, use the expansion factor to calculate the space required, and determine whether the method will fit the requirements and the system.

In applications requiring joining (merging) of data tables, assessing disk space requirements becomes slightly more complicated. For example, a SQL join of two tables on key values produces a data table containing all or some of the columns found in the source tables, and a number of rows that depends on the number of matches among the key values. The data table produced by an unrestricted join will have at most $cols_1 + cols_2$ columns and $rows_1 * rows_2$ rows. An unrestricted join of two tables, called the Cartesian product after the number of points in a two-dimensional space defined by (x,y) coordinates, can easily overwhelm a system, even when the data tables have fairly small numbers of rows. For example, when key variables in each table happen to have the same constant and identical values, or the programmer omits the expected SQL WHERE statement, the resulting Cartesian product of two data tables with 10,000 rows and 100 eight-byte columns (totalling 8 MB each) could swell to the vicinity of 100 million rows and fill up 160GB of disk space!

One does not have to carry a scientific calculator on one's belt to identify these black holes of computing. Any program that adds a large number of columns to a data table that contains a large number of rows deserves special attention. The same rule holds for joins or merges of large data tables: if you do not know that one or the other of the two tables has unique values, watch out! Duplicated and matching key values in both tables can lead to errors. Use this SQL code to identify duplicates:

```
create table dups as
select *
from <dsn> as T1
group by T1.ID1,T1.ID2,...
having count(*) > 1
```

Relational database integrity rules protect users against some of the traps, provided the data model in effect establishes and enforces the rules.¹ Using a SQL join to test for referential integrity, for example, may reveal some errors and prevent others. This SAS SQL² program checks the ID and amount in transactions (T3) used to update T2 against the same values already in T1:

```
* Test for referential integrity. SWH 6/27/85
proc sql;
  insert into T2(refID,name,due)
  select T3.ID,T3.refID,T3.name,T3.due
  from T3
  where T3.refIDtrim(put(T3.due,B.2)) in
  (select T1.IDtrim(put(T1.owea,B.2)) from T1);

title "Referential Integrity Error";
select distinct T3.ID,T3.refID,T3.name,T3.due format=dollar6.2
  from T3,T2
  where T3.ID not in (select T2.ID from T2);
```

Some of the more dangerous traps arise under the cover of procedures. Summary procedures in general, and cross-frequencies in particular,

produce a data table that contains a cell for each possible combination of the values of two variables. The number of discrete states of each of the two variables determines the number of cells defined. The cross-frequency of two columns of unique ID's produces a data table equal to the product of the number of rows in each column; that of two columns of 10,000 real numbers can produce a print file containing as many as $10,000^2$ or 100 million cells.

Understanding these traps may, in addition to helping us avoid them, also help us identify methods that minimize or even reduce an application system's requirements for disk space. In that direction lies the path of true gains in program efficiency.

Shells

Shells have a vital role in interactive systems, but in other contexts we identify them as automated processes that retain program control longer than the application requires. A closed shell will put a program in an endless loop. Either the programmer or a system administrator has to intervene in the application and terminate it. Other forms of shells will terminate eventually without intervention, but their scope exceeds the need for them. As a result, the program containing this form of shell may interfere unnecessarily with the work of others.

In 4GL languages, and we should include SAS in that group, programmers rarely find it necessary to specify a sequential loop through the rows of a data table, iteratively or recursively, and risk constructing a shell. 4GL procedures control automatically the looping structures. While embedding loops through variable arrays may introduce a shell in a 4GL program, using fewer variables would solve that problem and perhaps others as well. Despite the automatic control of looping in 4GL systems, bothersome shells can occur in

macro language procedures that execute other automated processes iteratively or recursively, and in SQL in-line views. It often pays to compute the product of the estimated number of iterations of the outermost loop, the estimated number of iterations of the next outermost loop, etc. A programmer can avoid shell traps by testing loop counters at each nested level of the loop structure and terminating the loop when the loop counter reaches an extreme value, as in

```
* Macro loop guarded against shell trap.
* SWH 6/27/95.
* Macro loop guarded against shell trap.
* SWH 6/27/95.
%macro shell;
%let n=1;
%numobs(x);
%do %while((&n numobs > 0) and (&n < 100));
data x y;
set x;
by ID;
if first ID then output y;
else output x;
run;
data ID&n;
set y;
run;
%numobs(ID&n);
%let n=%eval(&n+1);
%and;
%mend shell;
%shell
```

Should an error occur, such as finding dataset x not ordered on ID, the guarded loop will not force an abnormal end to the process.. SQL in-line view programs tend to conceal logical traps. For example:

```
select *
from (select distinct T1.*
      from T1,T2
      where T1.ID ne T2.ID
      )
where T1.attr1='OK'
```

not only imposes an extreme I/O burden on a system, it also does not omit rows of T1 where the ID matches an ID in T2. The in-line view (in parentheses) either blows up the file system or returns the original table T1 if at least one ID in T2 differs from those in T1.

The program,

```
select *
from T1
where T1.attr1='OK'
and T1.ID not in (select distinct t2.ID
                  from T2)
```

uses fewer computing resources and correctly excludes rows from the result where the ID in T1 matches any element of the set of ID's in T2.

Mixed Signals

Nothing misleads programmers as quickly as the program that used to work. Say a program compares a variable of type real (decimal number) in one data table to the same type of variable in another table. If it works once, shouldn't it work again? A lot of wasted effort goes into checking other components of an application system for errors because the programmer chooses to ignore the actual source of the error because he or she tested it and it worked before.

Bluntly put, high-level programming languages allow a small but unavoidable margin of error to creep into the machine code they produce. In the case of real numbers, you do not necessarily get what you see on the screen or in print. Storing real numbers inevitably entails truncation or rounding errors. We know, for example, that something as basic as the fraction 1/3 has no exact representation as a real number. Comparisons of two variables of type real may evaluate as false even though the two numbers look identical in a display.

To make the application development process more efficient, programmers have to learn to recognize the conditions that may produce mixed signals. The greater the volume of data processed by an application system, the more the value of precise methods increases. This rule follows from the idea that a programmer can ignore some odd cases because they almost never happen. Does that mean that these odd cases occur once in ten thousand observations? Once in a million? Once in a billion? An application that joins hundreds of thousands of rows of one table to hundreds of thousands of rows in another could easily require a billion comparisons.

As the scale of our data tables has grown to millions of rows, we have tended away from using real numbers as the types of key variables, replacing them with character types. Unless we know the truncation or rounding process used to create them, we shy away from using them directly in Boolean expressions. The SAS PUT function or similar function converts a real number to a form more suitable for comparisons. SAS date comparisons tend to work OK, even though represented as type real, but probably deserve a closer look.

You should not think by this initial emphasis on real numbers that mixed signals occur only during comparisons of variables of type real. Real numbers merely serve as a typical example. In fact, we have found a number of mixed signals in programs that seem to work perfectly well with smaller data tables, but fail when fed larger data tables compiled by different processes on different platforms.

For example, a missing value of a or b in just one observation will trap this process in a closed shell:

```
data x;
  set y (keep= a b c d);
  do while(b-a ne 0);
    c=d;
    a=a+1;
    output;
  end;
run;
```

A list of some of the mixed signals that we have encountered recently appears below:

Mixed Signal Traps

- automatic type conversions;
- uninitialized variables;
- numeric expressions containing variables with missing values;
- direct comparison of real numbers;
- ambiguous string lengths.

Some of the gains in productivity realized by using declarative 4GL and other high-level languages come from the way these languages take over the task of deciding how to represent data elements. We have to realize that even those talented assembly and C language developers who make our life easier cannot always provide both convenience and precision. Those of us who use high-level languages have an obligation to recognize the situations likely to produce mixed signals and take necessary precautions.

III. EFFICIENT METHODS

Methods that optimize the dimensions of data tables or references across linked tables required for application systems have the best chance of producing true and substantial gains in program efficiency. Brooks said it best: "Beyond craftsmanship lies invention, and it is here that lean, spare, fast programs are born. Almost always those are the result of strategic breakthrough rather than tactical cleverness

from redoing the representation of the data or tables.⁷⁸

Database compression techniques, such as summaries and partitions, convert research databases into more compact forms while preserving the information required for applications. Views make it possible to divide programs into a series of simple queries without having to shuffle data around on disk.

Database Compression

Both database compression and file compression can convert a database to a form that makes it more compact, while still preserving the information it contains. Despite their common purpose, the methods differ in fundamental ways. File compression operates at the system implementation level. It converts the method of representing data from the operating system's default method to one that reduces data storage requirements. As a rule, file compression affects database access methods, in that it must as a first step decompress all or part of the database. Database compression operates at the logical level. It converts the data model for the database to an equivalent but more efficient view of the data.

Summaries

A simple and limited example illustrates a method of database compression. Say we have on tape a database consisting of records of one or more events per person, records of test results per event, and records of the attributes of persons. A client wants to determine by demographic and event categories the proportion of events with certain test results. To support this task, we can put the data on-line in a linked collection of SAS files or a RDBMS, and develop a set of summary queries to compute the proportions. Alternatively, a compressed form of the original database might support the summary queries just as

well, yet take much less disk space to store and far less time to access.

Unique identifiers of records and other variables with large numbers of possible values have no role in summary queries. Stripping these from an on-line database of many rows will leave a categorical database containing a large number of duplicate rows. Summarizing the categorical database gives us one row per unique combination of column values and a column containing the frequency of each row. The summary contains the same information as the detailed categorical database, but far fewer rows.

The SAS PROC SUMMARY (with the NWAY option) or PROC SQL summary query makes the summarizing of categorical databases almost trivial:

```
* SAS Summary Template
SWH 6/26/95;
proc summary data=<dsn> nway missing;
  class <variable list>;
  output out=<outsn> (drop= _type_);
run;

* SAS SQL Summary Query SWH 6/26/95;
* Precede w/ format definitions;
proc SQL;
  create view temp as
  select distinct sex, age as agecat format=$agecat,
    race, educatn1, country1,
    First_T, Type1, cus, transfus, HTLV,
    case HTLV when 'NF' then 1
              when 'PO' then 2
              when 'NE' then 3
              when 'IN' then 4
              else 5
    and as outcom,
    count(*) as _freq_
  from in.<dsn>
  group by sex, agecat1,
    race, educatn1, country1,
    First_T, Type1,
    cus, transfus, HTLV
  having count(*) > 0
  order by calculated outcom;
```

Both methods construct the same form of summary. Note that the SQL query illustrates conditional reassignments of values and formatting of variables prior to the summary.

Summarizing often achieves remarkable rates of compression. The product of the numbers of states of all categorical variables (the Cartesian product) sets the upper limit on the number of rows the summarized version of the categorical database will contain. One can estimate the upper limit from frequencies of samples taken from the database. These frequencies, listed in descending order of frequency, will also help one estimate the compression ratio. The series formed by the cumulative percentage of rows accounted for by successively increasing percentages of categories in the ordered frequencies will likely begin approaching a constant number of rows. In one summary involving moderately fine divisions of at least some variables, we observed compression factor of more than fifteen.

Summary tables have some useful properties. One can produce from them the same frequencies by subsets of the categorical data table that one could compute from the original database. (See the FREQ statement of SAS PROC FREQ for details.) The same goes for frequencies of selected rows, say for a single year or location.

With the WEIGHT statement of SAS PROC CATMOD, an analyst can specify a summary data table as the input dataset and produce correct logistic regression parameter and variance estimates. (The newer SAS PROC LOGISTIC requires a summary data table in events-trials format). Many other statistical procedures also accept summary tables as source data.

Partitions

A variety of partitions of a large database may under some circumstances reduce the disk space required to store the database and the computer resources required to access it. Most worthwhile partitions reduce the number of redundant data elements in the database. A data model for the database may replace redundant data with implicit links between identical values of key variables. Shifting repeating sets of variables in a large data table into a smaller data table, keyed back to the larger one, may save some space and offer other advantages as well.

Knowledge of the relations among data elements may help us do even better. In a database defined by a relational data model (in particular, one with unique primary keys and no significance in the ordering of rows), we can without losing information partition complete rows of any data table into two or more smaller data tables. If we know a way to use one or more of the columns to partition the rows in a way that leaves in one of the partitions a set of columns with the same values per column, we can define this pattern of column values as the default for those columns in that partition. For example, a data table (T) contains records of events (blood donations) that includes a set of eight screening test outcomes. Over 90% of the rows in the table have exactly the same pattern of test outcomes. If we select all of the rows with this dominant pattern of test outcomes from T, we can crop the test outcome columns from the resulting data table, T1. The rows in the original table that do not have the dominant pattern of test outcomes go into a separate data table, T2.

Except for the missing pattern of test outcomes in T1, the partitioned data tables contain exactly the same information as T. To prove that, we can reconstruct T as a virtual table, utemp, defined by a view program:

```

* Create view
proc sql;
  create view utemp as
  select *
  "NR" as v1,
    "NR" as v2,
    "NR" as v3,
    "NR" as v4,
    "NR" as v5,
    "NR" as v6,
    "NR" as v7,
    "NR" as v8
  from lib.dan1
  outer union corr
  select * from lib.dan2;

```

Defining the database in this form means that it takes 40MB (almost 20% in this case) less disk space to store records of more than 3 million blood donations. In important ways, it also improves access to the data.

Views

The view program listed above reconstructs a virtual table equivalent to T from partitions T1 and T2. The virtual table has many of the same properties as T. If the virtual table name, utemp, appears later in the same program following a SQL FROM clause or a SAS SET or MERGE statement, it will have the same effect as the name of an actual data table called T.

The view utemp differs from T in that the program does not actually read data from the source tables T1 and T2 until it has to commit data to a physical file. This means views can replace many of the work files that programmers use to partition data into subsets before combining the subsets into a more compact data table or report. Reducing in this manner the CPU cycles, I/O's, and disk space used to create, store, and reread work files will truly improve program efficiency.

IV. Conclusions

Program efficiency still matters. We have discovered that one grossly inefficient program can overwhelm a computer system. As the scale of an application increases, training programmers to avoid obvious traps and use better methods minimizes the strain on system administrators and the risk of some users interfering with the work of others. Further, combining better methods with the right balance of computing resources and programmer skills does lead to true efficiencies.

Acknowledgments

Ian Whitlock, Michael Rhoads, and Willard Graves at Westat contributed comments and suggestions that led to substantive improvements in content and presentation. Jerry Gerard improved the design and layout of text and examples. The author alone takes responsibility for remaining defects. The views presented do not necessarily represent those of Westat, Inc.

- ¹ Codd, E.F., *The Relational Model for Database Management*, Version 2. Reading, MA: Addison-Wesley, 1990, pp. 243-257.
- ² SAS Institute, Inc., *SAS Guide to the SQL Procedure*, Version 6, First Edition, Cary, NC: SAS Institute., 1989. (Other standard SAS manuals not referenced).
- ³ Brooks, Frederick P., Jr., *The Mythical Man-Month*, Reading, MA: Addison-Wesley, 1975., p. 102.