The Dirty Secret Behind the Box: Why Programming Efficiency Should Matter to All of Us
John J. Cohen

Abstract
In Data Mining we tend to focus on the sexy front-end tools, the built in procedures, and canned routines. These have certainly been optimized for us, and in any case, we will need to do whatever it takes to produce final results. Unfortunately the less visible part of the job - data preparation - can take up to 75% of the effort. Further, we sometimes have to create our own procedures. Because of this, writing efficient programs should be important to all of us. We will suggest here how to do that without losing sight of our main analysis.

Introduction – So Much Data, So Little Time
Data miners are busy people. When not interacting with colleagues, attending training sessions and conferences, preparing and making presentations, traveling between client sites, or shopping for business suits, we are likely engaged in recruiting and supervising the junior employees who will help us do our work. Sadly, even with this help, we are too often left with little time for what we do best, analyzing our data.

These difficulties are compounded by the typically too-tight deadlines, the very large size of our data sets, and the need for multiple iterations of compute-intensive procedures. Add to this our likely need to be at least somewhat involved in the data preparation phase. (After all, it was the father of data mining, Socrates who said, “The data will not speak to me unless I play with it first.”) Thus, even for those of us for whom resources are nearly unlimited (for hardware upgrades, software, and “people ware”), there is never enough time.

A good step towards reducing the amount of time required for each project is to have well-tested approaches in place for guiding each new endeavor. (For instance, see Parr Rud, Data Mining Cookbook.) Having good SOPs for handling many of your typical projects is enormously helpful to maintaining productivity in the typically busy shop.

How we get there likely will include making sure that our programs are as speedy and productive as possible. This is certainly important during the big number-crunching steps. In fact, it will be suggested here, it is equally important in initial data screening, preparation, and early descriptive efforts. While the hierarchy of importance (how large is the job, how often will it run, when will it run, etc.) still obtain, few of us have the luxury of re-visiting code unless required to do so just to get the program to run at all.

Reasonably accurate results are usually good enough to allow us to move on to the next task. Elegant code is something for computer scientists to ponder.

Unfortunately, if each little (or not-so-little) program takes even 10% or 20% longer than necessary (and we will show examples of even greater savings), the cumulative effect may be fewer programs --- and projects --- can be completed each day. If one makes a practice of writing generally efficient code, there should be little need to perform second-pass optimization except in those rare instances either when a “one off” project evolves into a production job or when the program absolutely will not run to completion without a significant engineering effort.

We will offer here and in the presentation a selection of efficiency tips worthy of incorporating into your programming tool kit. Making a regular practice of including these kinds of techniques should result in generally efficient programs with little if any additional effort. We will also offer solutions to some of those situations when your job just will not run.

The Things Which Matter
First and foremost, your time matters. The goal should be one pass programming, with code not so intricate that half-an-hour later, you can no longer understand what you did. Secondly, computer time is important. While we will mostly experience this as the elapsed time (or clock time), this likely is correlated with the CPU or computational time. Storage space may be important, depending on your platform, as may I/O (or data input and output).

We will not suggest a set of hard-and-fast rules to follow as programs seldom work quite so logically. (How often have we heard that the Data Merge doesn’t really work the way you think?) Rather, a collection of ideas will be presented relative to each of our goals and an admonition to test your programs in your environment. (What is a “big” data set depends on the size of your box. The threshold where efficiency benefits appear may be 1,000 observations, 1,000,000 observations, or for lucky ones some number too large to contemplate.)

We will describe here several tips and techniques. In the presentation we will also offer our benchmarks. Of course, past experience is not a guarantee of future performance, but these should give you a starting point for your own testing and evaluation.
Making Everything Small - Programs

The general goal is to minimize the amount of work performed, whether by keeping the size of your variables small, limiting the amount of data being manipulated, or by planning your programs to minimize processing. We will address these in reverse order.

Two processes are generally regarded as the most expensive ones to perform for SAS. The first we encounter when we read in data from (or write it back out to) external sources. This cannot be avoided the first time. But if you save this data set as a permanent SAS data set, you will only need to read it in again in testing your initial read or if the external data are updated.

The second expensive process is sorting. We almost always need to order our data at some point. Take advantage of procedures that can create output data sets that are ordered (such as Proc Freq or Proc Means – see example 1). Note whether your data are already sorted (either externally or in your permanent SAS data set). And try to plan your program so that you are not sorting it the same way more than once (see example 2).

Example 1 – Ordering Without Sorting

```
Proc Freq data=Perm.Stuff;
  Table var1 / out=sorted;
```

Example 2 – But It’s Already Sorted

```
Data One; Set Perm.Stuff;
  Newvar1 = Var1 * CPI;
Data Two; Set One;
  Newvar2 = Var2 * pop;
```

Finally, within data steps one can minimize the amount of work being performed. Imagine a data set with one million variables and observations. We will want to perform differential calculations depending on the value of Card (i.e., which brand of credit card our customers are carrying). A straightforward approach is shown in example 5.

Example 5 – Selective Processing With an IF

```
Data One; Set Perm.Stuff;
  If card = ’Platinum’ then . . . ;
  If card = “Gold” then. . . ;
  If card = “Silver” then. . . ;
  If card = ’Bronze’ then. . . ;
```

This is a fairly typical program for us. It is accurate in that the appropriate calculations and processing will only be performed on the desired observations. And it is not inordinately expensive. But each time we read in a record, all four If statements are evaluated, regardless of whether or not the value has already been found. (For instance, if observation 1 is Platinum, we will still also check to see if it is Gold, Silver, and Bronze in turn.)

We can save processing time by limiting the value comparisons to only those observations for whom we have not yet found a match by using an “if then else” construction. In example 6 once the match is encountered, we will not process that observation again, at least until the next set of statements.
Example 6 – If-Then-Else

We can further reduce processing if we have advance knowledge of the likely relative frequency of our population in each group and order these in our program from most likely to least likely. (If the bulk of our customers are carrying bronze cards, the first If should look for these, followed by the next most numerous group, etc.) This will minimize the number of observations being checked for their value of Card more than once.

Example 8 – Keep At Read In

The number of variables processed can be controlled even more powerfully by combining the Keep and Drop statements. This is especially useful when combining and/or creating multiple data sets, although it makes debugging and testing a more exacting exercise. In example 9 we are match merging Monthly data with State manager data by the key variable State. We are writing out a first half (H1) and a second half (H2) data set, keeping certain variables in common and specifically excluding selected variables from each of the output data sets (Q3 & Q4, Q1 & Q2, respectively).

Example 9 – Keep As Data Set Options

Finally, there may be instances where we need to read in a data set but not write it out at all, at least not as a SAS data set. Examples of this include the creation of flat files for sending to external clients, Data Step/Data _Null_ reporting, or, as in example 10, where our sole purpose for reading in the data is to capture and output information to a macro variable.
Make Your Data Sets Small – Limit Observations

Limiting the number of observations processed is the twin to limiting variables. This can occur when you read in a data set as well as through controlling subsequent processing. (We have seen examples of the latter using If and If-then-else logic in examples 5 and 6 above.) The sub setting If will limit, at the outset, which observations are processed past initial read in. In example 11 we will include in our analysis only those observations with certain values for the variable Segment.

Example 11 – Sub setting If

```sas
Data New_seg; set market;
  If segment = '1' or segment = '5' or segment = '7';
    If segment = '1' then Market = 'indirect competitor';
    Else if segment = '5' then Market = 'our product';
    Else if segment = '7' then Market = 'direct competitor';
```

This can be improved on by substituting a Where for the initial sub setting If. (An If operates after the entire observation is read in. In contrast, the Where will read in just the variables needed to evaluate the where clause, then either read in the rest of the variables or exclude them and go on to the next observation.) This can also be improved on both in programmer time and, fortuitously, computing time, by using an In operator to list the multiple values being compared. These are shown in combination in example 12.

Example 12 – Where Clause and In Operator

```sas
Data New_seg; set market;
  Where segment in ('1','5','7');
    If segment = '1' then Market = 'indirect competitor';
    Else if segment = '5' then Market = 'our product';
    Else if segment = '7' then Market = 'direct competitor';
```

Example 13 – Where Clause with a Proc

```sas
Proc Means data=market(keep=segment Sales State);
  Where segment = '7';
  Table State*Sales/list;
```

A final instance of limiting the number of observations uses the data set option Obs=xx as in example 14. This is used often in testing (and, unfortunately, cannot be combined with the Where clause). We are more likely to use a random sampling technique if our analysis can be performed on a numeric subset of the data. (What is the generalizability of the first xx observations to our entire population?) However, there are certain circumstances where this will be meaningful. Here we are ordering our customers from highest-to-lowest by total annual sales volumes. We then pick off the top 1,000 customers for a special promotion.

Example 14 – Obs = xx

```sas
Proc Sort data=market;
  By Descending Sales;
Data Special; set market(obs=1000);
  (additional processing)
```

Making Your Variables Small

Small variables have an obvious benefit in terms of storage space. But much as we can limit the volume of data being processed by limiting the number of variables, restricting the size of the variables included will also reduce the volume. Depending on the operating system, SAS will store numeric variables in a default of up to 8 bytes. For a variable such as the number of family members, for instance, likely 2 bytes will be sufficient. Character variables, especially those created in a data step with certain functions, may default to a length of 200 bytes (even if the actual data display in a length of as little as 1 character). Both can be controlled using a Length statement as in example 15.
Example 15 – Length Statement

```plaintext
Data Special; set market;
  Length family 2 segment $15;
  Segment = trim(left(market)) || State;
```

Setting a length on a numeric variable may be tricky - the length we set may be effective only for the duration of that data step. In the next use of the data set, SAS may re-expand to the default length. In instances where the real nature of the data in question is ordinal (or even categorical), one need not forsake numerals. One might instead consider converting 1 or 2-digit numeric variables into 1 or two-digit character variables. There may be instances where this results in a more accurate representation of the data, such as when leading zeros would otherwise be truncated (e.g., zip codes, social security numbers, and many id codes). In example 16 we are converting numeric zip (here-to-fore stored as an 8-byte numeric) into a 5-byte character variable. (We create a new character variable named New_zip, drop the old numeric Zip, then rename the new variable as Zip.)

Example 16 – Numeric to Character Conversion

```plaintext
Data Special; set market;
  New_Zip = put(zip,Z5.);
  Drop zip;
  Rename new_zip = zip;
```

A final technique for saving space involves the conversion of long character variables to codes for processing. At reporting time, the original values are then restored. Depending on the size of the values, the savings may be substantial. This is one solution for bypassing the 16-character limit in Proc Format. In example 17 we are converting long values to a one-character code (Data Step). We create a format $class to capture the relationship between the new code and the original values (Proc Format). At reporting time (Proc Print) we restore the original values using the format $class.

Example 17 - Numeric to Character Conversion

```plaintext
/*convert long values to 1-byte*/
Data New; set market;
  If class = 'Early Nester'
    then Code = '1';
  Else if class = 'Yuppie'
    then Code = '2';
  Else if class = 'Retiree'
    then Code = '3';
  Else if class = 'Outlandish Poseur'
    then Code = '4';
  Drop class;

/*code-long value relationship*/
Proc Format;
  Value $class
    '1' = 'Early Nester'
    '2' = 'Yuppie'
    '3' = 'Retiree'
    '4' = 'Outlandish Poseur';

/*restore original values*/
Proc Print Data = New;
  Id Code; Var sales;
  Format Code $class.
```

Special Cases – When Regular Programming Efficiencies Are Not Enough

Sometimes generally good programming technique is not enough (and we won’t mention Cartesian Products). No matter how good your program, your job will not finish successfully. The simplest step is to be sure that you have access to maximum available resources and that no program or system option is overriding and somehow limiting this. For instance, check to make sure that the Sortpgm option is set optimally for your installation.

The next step may be special engineering of your program, re-working your program to fit in the available space. While these failures may manifest themselves as lack of sort space, workspace, memory, or of time, they likely fall into a class of either failed sorts, or failed ordering/crunching (which are really sorts as well). Somehow, you are unable to acquire sufficient resource for the job. Approaches suggested here sometimes represent elegant solutions and other times more resemble brute force.

Running out of sort space can be bypassed by explicitly targeting the output destination for the ordered data set. Normally, SAS will, for the briefest of instances, store pre-sorted and post-sorted versions in the same space – for which you really may
NOT have enough room. In example 18 we assure ourselves of space to grow.

**Example 18 – Proc Sort Out=**

```
Libname temp 'D:\SAS8\Workspace';
Proc Sort Data=market
   Out=temp.market;
   By class;
Data market; set temp.market;
   (additional processing)
```

The presumption here is that your D: drive is NOT your normal SAS workspace. We instead are using it for auxiliary storage. On some platforms, it may be prudent to write these intermediate data sets out to tape medium. Generally the approach of saving intermediate results is a way to get past “timing out” on certain platforms --- presuming that storage charges and space are not even less appealing.

A second set of error messages will tell you something to the effect that you have too many levels of your class or by variable for the available memory (often Proc Means or Proc Freq). With Proc Means/Summary and two or more Class variables, when the goal is reliable summarization (and processing time/sorting are not the limiting factor, nor are summary statistics at that level of detail the goal), a work-around involves first sorting the data by the higher order Class variable. Next you re-run the Proc Means By the higher order class variable, leaving one less variable on the class statement. The output data set will look identical in result with the processing now fitting into memory. (See example 19.)

**Example 19 – Proc Means By/Class**

```
/*before - runs out of memory*/
Proc Means data=market;
   Class segment state;
   Var sales;
   Output out=summary sum=;
/*after - runs to completion*/
Proc Sort Data=market; By segment;
Proc Means data=market; By segment;
   Class state;
   Var sales;
   Output out=summary sum=;
```

Similar problems can happen with Proc Format, which does not lend itself as readily to the same solution. Example 20 below is using the very nice Cntlin feature of Proc Format to dynamically create a format storing the relationship between customer id code and state of residence. This is generated from an in-house database. This will be used to flag physicians from California as they are read in from a new external file without having to sort or combine either file (a “table lookup” technique). For normal processing, however, the input table (of id-to-state) is too large. The work-around solution (maximum of 65k unique values on this platform) is to break the data set into two roughly equal pieces, create two separate formats, and then to sequentially poll incoming observations using the two formats. As format names cannot contain numerals, these are called $stateo (for $state_one) and $statet (for $state_two). Proc Format is generally an extremely efficient tool for processing data in SAS.

**Example 20 – Proc Format Work-Around**

```
/*create split data sets*/
Data Customer1 Customer2;
   Set Perm.Physicians;
   label = state;
   start = ID
   end   = ID
   fmtname = 'STATEO';
   type = 'C' ;
   if _N_ <= 35000 then
      output Customer1;
   else do;
      fmtname = 'STATET';
      output Customer2;
   end;

/*create split formats*/
Proc Format CNTLIN = Customer1;
Proc Format CNTLIN = Customer2;

/*apply split formats*/
Data CA_Docs; infile nation;
   Input . . .;
/*subsetting If on California*/
   If put(ID,STATEO.) = 'CA' or
      put(ID,STATET.) = 'CA';
```

A more general work-around has been called an “elephant sort” by Neil Howard at NESUG '88. Variations on this theme exist (see pp.96, 97 in SAS® Programming Tips). A very large data set is broken up into smaller pieces using the highest-level variable values (or the only one if only one is desired on the By variable list). Each of the smaller pieces can now be successfully processed individually. The results are then concatenated (on a Set statement) in the
proper order. The final result should be a fully ordered, successfully processed data set. In example 21 we will only be performing a sort, but this approach can be extended to more complex problems.

Example 21— An “Elephant” Sort

```sas
/*break data into logical groups*/
Data A_G H_M N_Z;
Set Perm.Too_Big;
If Lastname < 'H' then output A_G;
Else if Lastname < 'N' then output H_M;
Else output N_Z;

/*sort each smaller group*/
Proc Sort Data=A_G; By lastname;
Proc Sort Data=H_M; By lastname;
Proc Sort Data=N_Z; By lastname;

/*concatenate in correct order*/
Data Sorted;
Set A_G H_M N_Z;
*** same result as if sorted ***;
*** Perm.Too_Big itself. ***;
```

Parallel Processing with SAS Multi-Process Connect

On other occasions your time is the limiting factor, tasks take hours to run (although they will complete), and updates and supplemental analyses are required with regularity. An alternative approach to trying to access a bigger box is to instead use multiple boxes as described here.

This section adapted from John Bentley’s paper “SAS Multi-Process Connect: What, When, Where, How, and Why” with the author’s permission.

Very large databases—data warehouses and data marts—containing tens of millions of records are relatively common nowadays, in the form of data warehouses and data marts. Because of their size and complexity, despite the best efforts of dedicated programmers data access is often not easy and processing time is measured in hours. Version 8 of SAS/Connect includes a product, however, that can sometimes dramatically reduce access and processing time.

The Multi-Process Connect facility (MP Connect) can exploit multi-processor capabilities of symmetrical multiprocessor (SMP) and massively parallel processor (MPP) systems “by allowing parallel processing of self-contained tasks and the coordination of all the results in the original SAS session”. (SAS/Connect User’s Guide, On-line version.)

Without MP Connect, SAS Software, including Version 8, executes sequentially one step or procedure at a time on a single processor. (SPDS is an exception.) Even when extracting from parallel relational database engines, the results of the SQL query must be returned to a single processor where the remaining SAS code executes sequentially.

Parallel Processing

The term parallel in the computing context used in this paper refers to simultaneous or concurrent execution—individual tasks being done at the same time. Without MP Connect, processing is restricted to sequential dependency, in which Task B can’t begin until Task A is finished.

Consider a simple SAS program that reads in a flat file to create a SAS data set, transforms the variables, sorts the data set, and then calculates grouped summary statistics. PROC SORT doesn’t begin until the DATA step ends, and PROC MEANS won’t run until the sort finishes. The total execution time for the job is the sum of the times for each of the separate steps.

On multiprocessor computers, MP Connect provides independent parallelism. This occurs when there are no dependencies between tasks and they can therefore be run concurrently. Suppose our example has two flat files that need to be read, manipulated, sorted, and then merged before calculating group summary statistics.

In our example, if we read in and manipulate both data sets at the same time and have the output data sets sorted at the same time, then we’ve implemented independent parallelism for this part of the processing. The processes of gathering the sorted data sets into a single data set and calculating the statistics are constrained by sequential dependency because all the data must be available at the same time. Execution time is calculated by adding the time required for the lengthiest series of parallel processes plus the time needed for the sequential processing.

MP Connect and Multiprocessing

MP Connect is part of the SAS/Connect product and is limited physically only by the number of CPUs available. But instead of running a SAS session on a remote host, it runs a SAS session on a different CPU on the same host as the local SAS session. It eliminates the need for executing a spawner program or a TELNET session on the remote host, and it also eliminates the need for a script file on the local host and the need to configure the remote access method.
Although MP Connect’s syntax is simple and straightforward, the programs must be “encapsulated” using RSUBMIT and ENDRSUBIT to create parallel and sequential code sections. Because the parallel sections must be self-contained, converting complex production jobs this may require a significant redesign and redevelopment effort. The ancient art of flowcharting can be a big help.

As with most thing, there is no silver bullet or free lunch. MP Connect reduces processing time at the cost of increased resource usage. If your job is using two processors instead of one, then the second processor is not available for another job. Although your job finishes faster, other users’ jobs may be delayed.

A Sample Parallel SAS Program

Code Sample 1 is a simple program written for MP Connect. It sorts two existing SAS data sets, merges them into one, then calculates descriptive statistics and prints the output. To supplement the SAS Institute documentation, Garner (1999) and Bentley (2000) are excellent references for the details of MP Connect syntax.

Code Sample 1. An MP Connect Program

```sas
options autosignon=yes sascmd='sas';
libname loans '/prod/data/home_loans';
libname cards '/prod/data/credit_cards';
libname common '/saswork';
rssubmit process=loans wait=no;
  libname loans '/prod/data/home_loans';
  libname common '/saswork';
  proc sort data=loans.may
     out=common.loan_data;
     by market district;
  run;
  endrsubmit loans;
  proc sort data=cards.may
     out=common.cards_data;
     by market district;
  run;
  waitfor loans;
  rget loans;
  data combined;
  merge common.loan_data
     common.cards_data;
  by market district;
  exposure=sum(homeLoanTotal,carLoanTotal,
              cardBalanceTotal);
  run;
  proc means data=combined;
```

Performance Tests

At a major east coast regional bank, three different production simulations were run on two different UNIX SMP systems, one IBM and the other HP. On each platform the default installation configuration of SAS Version 8.1 was used. Different data sets were run on each platform. All data sets were compressed and not indexed. The tests are reported in detail in “SAS Multi-Process Connect: What, When, Where, How, and Why” (Bentley, 2000). Table 2 shows the test results. _Time_ shows the total elapsed time—real time—in hh:mm:ss for the tests to execute.

**Table 3. Test Results**

<table>
<thead>
<tr>
<th>Platform</th>
<th>Task</th>
<th>Sequential Time</th>
<th>3-way Parallel Execution Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>RS/6000</td>
<td>Scoring, stats, and flat file creation</td>
<td>0:46:04</td>
<td>0:17:33</td>
</tr>
<tr>
<td>RS/6000</td>
<td>Sort, freqs, and stats</td>
<td>0:30:55</td>
<td>0:10:38</td>
</tr>
<tr>
<td>HP 9000</td>
<td>Report and data set creation</td>
<td>0:21:54</td>
<td>0:06:04</td>
</tr>
</tbody>
</table>

Writing SAS Programs for Parallel Processing

MP Connect syntax and implementation is clear and straightforward, but without a clear plan, a complex parallel program can turn into some of the worst spaghetti code ever seen. It’s temptingly easy to copy and paste existing code segments and then encapsulate them with RSUBMIT-ENDRSUBMIT, but a program of a few hundred lines with numerous DATA steps, LINKs, GOTOs, and PROCs can quickly balloon to a thousand lines of what looks like unmanageable repetitious code, depending on the number of remote sessions being used. Flowcharting makes parallel programming much easier to plan and implement. A visual representation of the process clearly identifies which sections must be sequential and which can be parallel.

Using macros and %INCLUDEs are important coding techniques for reducing the number of lines of code that must be written and maintained. Good programming style, resulting in improved readability, is also critical.

Keep in mind the trade-offs that MP Connect requires: More programmer time for faster execution; faster execution for increased resource usage; your job finishing faster while other jobs take longer. MP Connect is a great solution for some but not all problems. There are no silver bullets.

Conclusion

An argument has been made here that all of us can benefit from adopting a variety of efficiency techniques as part of our programming tool kit. The
goal is usually NOT to try to squeeze out every last superfluous CPU second or to save every byte of storage space. We are much too busy to spend that much time on every program. Rather, it is to regularly end up with generally pretty good programs, programs simple-enough and standard-enough in your shop to be readily maintainable and re-usable on the first try. Only for those very special cases will extra engineering be required.

To get there, in part, we are offering a collection of tips. Many may already be familiar to you or are readily adaptable. Your charge will be to test these techniques on your platform with your data and to develop the tool kit you will use.

Contact Information
John J. Cohen
Jcohen1265@aol.com

References


SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc in the USA and other countries. © indicates USA registration.

* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *