Paper PR04
The Skinny on Processing Very Large Health Claims Data Sets
Gopal Rajagopal, Merck Research Laboratories, Blue Bell PA

ABSTRACT

Peering into the future of an average SAS professional processing health claims data, from health claim data vendors such as Ingenix, Medstat and GPRD to support Epidemiology, Health Economics and Pharmaco-vigilance oriented studies/analysis, it seems likely that many will be processing larger and larger data sets. The typical SAS program uses relatively small data sets and may not scale well.

The focus of this paper is to present general SOPs/tips for processing very large health claim data sets and evaluate some SAS techniques to mitigate the local and immediate programming burden of the occasionally onerous I/O intensive ‘DATA’ step. Incrementally, as needed, one could consider global options which include changing the SAS environment, buying more hardware and disk space etc. Moving from local to global options, especially in a multi user SAS resource constrained environment, can produce a more uncertain outcome that potentially results in diminished control and increased costs.

INTRODUCTION

At the outset, it may be useful to clarify key terms such as ‘very large’ data set, ‘processing’, and ‘efficiency’ which are at the heart of this paper. The term ‘very large’, as used in the title of the paper, refers to a data set which has too many rows or too many columns or both. However, this explanation is somewhat nebulous in that, what is ‘small’ for one setting may be deemed to be ‘very large’ for another setting. The term ‘processing’ is restricted to typical DATA step tasks such as sub-setting, searching, sorting and merging. The term ‘efficient’ implies getting the job done with minimal or at least ‘acceptable’ expenditure of valuable resources such as time, disk space, programmer sanity and programming ease etc. So any data set is ‘very large’ if you have ‘trouble’ processing it!

Trouble could mean anything from program taking ‘too long’ to run and/or running into constraints such as disk storage and memory. Even if one does not process huge volumes of data often, there may be merit in routinely using some of the ideas suggested in this paper. One may thus pay homage to the oft invoked mantra of ‘doing more with less’ and complying with good programming practices.

At the first sign of ‘trouble’ with a very large data set, the SAS programmer’s first urge should not be to scrounge around for ‘more’ disk space, memory, CPUs etc. Adopting suitable SOPs to deal with the unique challenge of processing large data sets and using an incremental approach to reduce and optimize I/O, a frequent and significant culprit, may be fruitful. Subsequently, if needed, one could look at external factors to the SAS program such as changing the SAS environment and attempt to intervene at the operating system/hardware level.

It is important to recognize that there are trade offs involved in using various approaches and techniques. The trade offs, if applicable, will be broached at relevant points in this discussion.

PROCESS MANAGEMENT SOP/TIPS FOR LARGE HEALTH CARE CLAIMS MANIPULATION/DATA WAREHOUSING

There is a big difference between making one cup and one million cups of coffee i.e. the average SAS program, which may process ‘small’ data set(s), probably does not scale well. This statement encapsulates the central message of this paper. When it comes to processing small volumes of data, the main task/context encompasses specifying and developing the program followed by QC and other relevant task(s). With big health claim data sets, the odds are that the processing footprint is heavy and demanding vis-à-vis the general programming environment and infrastructure. Consequently, there is much more to working with large data sets than just developing/testing individual programs even though it is a critical task by itself. This section will consist of SOP suggestions and practical tips for processing large volumes of data. These SOPs/ tips have been garnered over years of processing large claims datasets (using SAS) from data vendors such as Medstat, Ingenix, CMS (formerly HCFA) and loading them into an Oracle based Data Mart.
Moving an elephant even one inch can be a non trivial task i.e. processing large data sets, especially during the development phase, in the interests of sticking to the project time line and saving the programmer’s sanity, should be approached differently.

- Iterative development and validation of an algorithm churning through huge production volumes can be problematic and time consuming. If feasible, tune and validate the algorithm by first using a small and representative sample of your data. Consider using SAS functions such as RANUNI to create your sample. It may be possible to accommodate many sample based development/QC iterations in the time you may need to complete one full volume run.

‘Divide and rule’ when your environment and resources do not allow for the processing of all data in one program

- Split your data into logically consistent chunks and process it.
- To reduce code maintenance and maintain quality/consistency, you may want to use a parameter driven splitting/processing SAS algorithm for both sample and production. The SAS code could be designed in such as way that the full volume can be processed with minimal changes to the code processing the sample data.
- Consider using tapes (especially with SAS on the MVS operating system) or other auxiliary storage medium to store intermediate transformation(s)/version(s) of the data. Tapes, albeit slower than disk, are relatively inexpensive. If you have multiple processing steps, you can track, if needed, intermediate data transformations more efficiently. Storing intermediate transformation(s)/version(s) may also allow you to start a multi step process in the middle or some other convenient point rather than having to start from the beginning every time the process is interrupted for one reason or the other.
- You might want to look into serializing process when parallel runs are problematic especially when the work library, CPU, memory, tape drives etc. are shared across programs and users. Serializing processes could take more time. It may also be useful to look into a scheduler, with automation capabilities, so that non working hours may be harnessed for processing.
- Localize and focus on the I/O heavy hot spot(s). The SAS FULLSTIMER option’s statistics (i.e. OPTIONS FULLSTIMER STIMER STIMEFMT= Z) is useful in this context.

Keep tabs on work library usage. It is a critical aspect of processing large data sets as programs often choke in context.

- Reduce the size of the data sets being stored in temporary work space
- Minimize, to the extent possible, the number of ‘big’ data sets processed in the temporary work space.
- Notify users on a shared system before running particularly big jobs which can potentially grab big chunks out of a shared work library.
- PROC DATASETS is invaluable in getting a listing and the size of all work datasets at different break points of a ‘big’ SAS program, and also in deleting unneeded work data sets. If you implicitly replace data sets, instead of explicitly deleting using PROC DATASETS, SAS will keep a copy of the data set to be replaced until the data step is complete. Thus, explicit deletion, instead of implicit replacement, can reduce the work library overhead.
- Assign, if available, a different but less crowded, work library. For e.g. in UNIX you can make SAS use another directory, to which you can write to, as the work library. Use the ‘SAS –work /new directory name prog.sas’ syntax from the UNIX command line where the ‘/new directory name’ is the alternative directory. This should be possible, using appropriate syntax, with other operating systems too.
- Tapes may be used as a work library if disk space availability is an issue. We have used this method with the MVS operating system.
- If the program terminates abnormally, the SAS work directory must be cleared (especially in UNIX, PC) or the debris of a previous run(s) can interfere with the any future run(s).

Your data volume’s footprint may be heavy but you still want to tread softly ….

- Befriend and appraise the infrastructure personnel (such as UNIX administrators, Network and MVS support personnel) about upcoming big runs.
- It is better to be known as the big data set tamer rather than a system crasher! If you work on a PC, you may not be sharing a CPU, but you might be sharing network storage and bandwidth. In UNIX you can use utilities such as cron, and features such as nice to schedule and prioritize jobs. As for MVS, you can submit jobs in job queues with lower priority but allow for more resources.

If you need to search mountains of health care data frequently and efficiently, acquiring and processing big volumes of data ready for use is only half of the equation. Some options, external to BAS SAS, which facilitate quick retrievals, are the SAS SPD server or housing the data in a data warehouse using a DBMS such as Oracle.
Research with health care claims data often involves the sub setting of claims based on inclusion/exclusion criteria such as a specific set of drugs and/or ICD 9 and/or procedure codes. Creating a set of searchable tables, maybe in a DBMS such as Oracle or MS Access, to cull the specific set of codes associated with the research hypothesis can significantly add to efficient data access. For example, if the study needs all the drugs (i.e. NDC codes) associated with a disease such as depression, you may need to isolate thousands of drugs from a constantly evolving set of approximately one quarter million drug codes. Consequently, it might be optimal to acquire and use different drug classification schemas such as AHFS (American Health Formulary System), GPI, Therapeutic group classification hierarchy from Medstat.

PROGRAMMING TECHNIQUES/APPROACHES FOR EFFICIENT PROCESSING OF LARGE HEALTH CARE CLAIM DATA SETS

Sequential I/O, slowest amongst different kinds of I/O, is SAS’s default. The ‘slowness’ of sequential I/O can become noticeable as the data volume increases. Consequently, it is logical for a programmer to attempt to minimize sequential I/O as it often represents a ‘big’ chunk of the cost of processing large data sets.

Further, the programmer would want to maximize non sequential I/O, since it is faster. Additionally, given that memory based I/O is faster than disk based I/O, the ideal situation would be to maximize memory based non sequential I/O to the extent possible. Most systems have far less RAM than disk space and RAM, like disk, is shared amongst users, applications and the operating system. So it may not be practical to load ‘everything’ to memory. SAS, by design, based on its configuration, also uses buffers i.e. RAM based I/O. The memory based techniques discussed in this paper are in addition to what SAS does by itself.

A number of techniques have been profiled in this paper in the context of processing large data sets.

The baseline environment was …

- The hardware was HP Model 9000/800. The operating system was HP-UX B.11.11 with 250 GB of disk storage. The UNIX box had 4 CPUs and 8 GB RAM. SAS 8.2 (TS2M0) was used for most programs. SAS 9.1 (TS1M3) was used for profiling the HASH object.
- A simulated data set consisting of 300,000,000 rows (50 to 53 GB) was created. This data set had 12 numeric variables (each 8 bytes long) and 2 character variables (each 39 bytes long). Two numeric variables, a unique and a non unique column, were indexed.
- All the technique profiling programs have been run with the same number of records using the hardware/software described above. They have been ‘run’ in a comparable environment so that changes can be reasonably attributed to whatever technique is being profiled.
- All time related statistics are averages from multiple runs so that the UNIX server load variance is factored in to the extent possible. All the program runs were serialized so that each running profiling program, to the extent feasible, had the server for ‘itself’. Other programs/processes running on the UNIX server were not interrupted.

Simple, i.e. hardware/ SAS version independent, metrics such as ‘Real Time’ and disk space have been used in the analysis keeping in mind the heterogeneous population of SAS users who might read this paper. ‘Real Time’ represents elapsed time as provided by the SAS FULLSTIMER option and the data set size statistic comes from the output of PROC CONTENTS of the data set in question.

The panoply of techniques/approaches discussed in this paper is by no means comprehensive or complete. It is hoped that the paper inspires the reader to go beyond the ‘small data set’ mindset when it comes to processing ‘very large’ data sets and thus help him/her confront and successfully overcome scaling issues.

SMALL IS BEAUTIFUL (EVEN WHILE PROCESSING LARGE DATA SETS!)

Three approaches to reducing the data set size are:

- Horizontal methods i.e. minimize the number of columns
- Data set compression
- Vertical methods i.e. minimize the number of rows

Any of these techniques, in general, can be used in combination with the others. A good point starting point is to get a PROC CONTENTS and a PROC FREQ of ‘critical’ variables of your big data set if it already available as a SAS data set. The metadata and related documentation provided by the data vendor can also help irrespective of the ‘raw’ data format. Saving the data in a permanent SAS data set, in general, should
be considered when any size reducing techniques are used so that the DATA step I/O overhead to make the big data set smaller is not needlessly repeated.

Horizontal (i.e. column oriented methods) for reducing data set size.

Table 1 presents profiling data on some of the column oriented techniques mentioned below. Discussion for the techniques is available in the section following the table.

<table>
<thead>
<tr>
<th>Label</th>
<th>Technique Description</th>
<th>Data Set Size</th>
<th>Time for IF 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>H1</td>
<td>Use LENGTH to reduce 8 Numeric variables (8 bytes each) to 3 bytes each.</td>
<td>39 GB</td>
<td>23.2</td>
</tr>
<tr>
<td>H2</td>
<td>TRIM length of the 2 char variables from 39 to 8 characters in addition to numeric variable lengths</td>
<td>19 GB</td>
<td>15.4</td>
</tr>
<tr>
<td>H3</td>
<td>Convert the 3 integer variables into 3 (1 byte) character variable</td>
<td>48.1 GB</td>
<td>18.2</td>
</tr>
<tr>
<td>H4</td>
<td>Map 3 integer variables with binary value (0 and 1) to bits of a 3 byte integer using bit functions BOR and BAND</td>
<td>53.4 GB</td>
<td>21.8</td>
</tr>
<tr>
<td></td>
<td>Before ‘bitmapping’ 3 integer variables</td>
<td>48.1 GB</td>
<td>28.8</td>
</tr>
<tr>
<td></td>
<td>After ‘bitmapping’ 3 integer variables</td>
<td></td>
<td></td>
</tr>
<tr>
<td>H5</td>
<td>Technique : COMPRESS the data set with Binary option (13 Numeric variables (length 8) and 2 character variables (length 39)</td>
<td>50 GB</td>
<td>26.7</td>
</tr>
<tr>
<td></td>
<td>Before compression size</td>
<td>21.9 GB</td>
<td>28.6</td>
</tr>
<tr>
<td></td>
<td>After compression size.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>H6</td>
<td>DROP variables (2 char variables and 5 numeric variables) Note : Size &lt; 25% of original size and IF run time is down to &lt; 50%</td>
<td>6.4 GB</td>
<td>10.0</td>
</tr>
</tbody>
</table>

1 For technique H4, the baseline was slightly different even though the total number of rows was 300 million.

Initial baseline size: 53.4 GB (with 13 Numeric variables (length 8) and 2 char variables (39 Chars).

Initial baseline IF run (in minutes): 27.1 (returning 112,504,793 rows)

2 All time is in minutes. All the techniques used SAS 8.2

- Explicitly specifying LENGTH of numeric variables (Technique H1 of the Table 1)
  Integer variables native to the data set or calculated integer variables that do not require decimal precision can be considered. The default length for numeric variables is 8 bytes. The minimum size for a numeric variable in UNIX and PC SAS is 3 bytes. An 8 byte numeric variable, even if signed, can accommodate a big number. Consequently, there may be opportunities to reduce the size of numeric variables in most big data sets. In order to avoid data distortion, before reducing a numeric variable’s length, the present and future minimum and maximum value of the variable under consideration needs to be considered.

- Removing padded blanks from character variables. (Technique H2 of Table 1)
  The length of character variables can be limited by the LENGTH statement. Removing padded blanks from character variables using functions such as the TRIM, COMPRESS or SUBSTR may help remove padded blanks. If the data is unfamiliar to you, consider examining the frequency listing of the variable to avoid inadvertent loss of data by using the COMPRESS or SUBSTR functions or the LENGTH statement.

- Convert Numeric variables into Character variables (Technique H3 of the Table 1)
  Numeric variables which are meant to be flags (i.e. with values such as 0 or 1) can be good candidates for this method. If they are stored as numeric type, you can reduce them to 3 bytes. However, if they are not used for numeric calculations, it may be worthwhile to change the numeric variable to a 1 byte character variable to reduce the data set size. It took 32 minutes to convert 3 numeric variables (each 8 bytes long) into three 1 byte character variables in the profile experiment.

- ‘Bit mapping’ technique (Technique H4 of the Table 1)
  SAS has bit functions by which individual bits in a multi byte numeric variable can be set and retrieved.
  1. For example, to store 3 flag like numeric variables (with values 0,1), a minimum of 9 bytes per record is needed. If these 3 variables are stored as bits of a 3 byte integer variable using the SAS bit function BOR, you can save space when the original flag variables are dropped.
  2. To retrieve the flag value(s), the SAS bit function BAND can be used.
The DATA step to create the multiple bit flag numeric variable needed 119 minutes. Retrieving the specific bit is also ‘slower’ than simpler conversion method i.e. technique H3 of Table 1 (18 versus 27 minutes). Missing values of the original flag variable can be a complication.

- **SAS Data Compression (Technique H5 of the Table 1)**
  If disk storage availability is a significant issue, compression may be an option. The ‘OPTION = BINARY’ (version 8) option attempts to compress numeric variables too.
  The following issues, at a minimum, need to be considered:
  1. Every time a ‘big’ compressed data set is used in a PROC or DATA step, a resource (i.e. CPU and disk) intensive process is used to uncompress ‘on the fly’. Users have no control.
  2. Data sets without ‘long’ character variables are less suitable for compression. If your data set has only numeric variables, compression may increase your data set’s size. Peruse the SAS Log to avoid this mishap.

  It took 70 minutes to compress the data. If the output data set created with the IF clause was also compressed, it took 60 more minutes.

- **Minimize data set flab by using KEEP, DROP statement(s) (Technique H6 of Table 1)**
  A strategic and parsimonious approach to keeping and dropping the variables needed for subsequent processing/analysis will reduce the size and the time needed to successfully run programs.

- **Normalizing (i.e. data modeling) your data set(s).**
  Occasionally health claims data come in a denormalized form i.e. it has redundant data. Redundancy can be useful in minimizing data set merges but it usually makes the data set bigger. Balancing between normalization and denormalization, based on how the data is used, is potentially a critical success factor especially when claims data is processed and loaded in a DBMS based data warehouse.

For example with outpatient data, which is usually the bulkiest amongst health claims data sets, some vendors provide redundant demographic information. In one of the health claims databases we have processed, a total of 4 million patients can have 90 million outpatient claims per year.

If you do not consistently need demographic data while processing outpatient data, the demographic related columns can be taken out using KEEP, DROP etc. The equivalent of a unique patient identification field needs to be kept in both the slimmed outpatient and the separate but much smaller demographic data set.

PROC SQL (with appropriate indices) or a sort/merge can be used when you need to marry the outpatient data with the corresponding demographic information.

Surrogate keys can also be used to shorten ‘long’ variables. Assume, for example, you have the whole state name in a big data set. Consider introducing a 2 character surrogate key (i.e. ‘Pennsylvania’ = ‘PA’) and DROP the variable with the lengthy state name. Save the data set version with the surrogate key as a permanent data set. A format can be used when the whole state name is needed for reporting etc.

**Vertical (i.e. subset by row selection) methods**

<table>
<thead>
<tr>
<th>Label</th>
<th>Technique description</th>
<th>Time to Select$^3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>V1</td>
<td>Used 'WHERE var in (&amp;macrovar)' syntax with a macro variable storing all search values. Note : Reduced I/O selected rows being read in at outset and search values in memory</td>
<td>18.9</td>
</tr>
<tr>
<td>V2</td>
<td>Used Hash Object ( use only Version 9) Note : The search values are stored as a HASH object in memory. It behaves like an index in memory.</td>
<td>20.8</td>
</tr>
<tr>
<td>V3</td>
<td>Used 'PROC SQL' syntax (after sorting big data set ) Used 'PROC SQL' syntax without sorting the big data set</td>
<td>23.0 23.6</td>
</tr>
<tr>
<td>V4</td>
<td>Used 'WHERE var in (x,y,z)' syntax by listing search values Note : Reduced I/O. However, macro variable method seems faster ( i.e. Technique V1).</td>
<td>23.2</td>
</tr>
<tr>
<td>V5</td>
<td>Created a format to house search values. Used a WHERE clause syntax 'WHERE PUT(formvar,? formtmp.) EQ '999' ;' Note : WHERE clause’s reduced I/O. Also used PROC FORMAT’s memory based I/O.</td>
<td>31.2</td>
</tr>
</tbody>
</table>
Using WHERE instead of IF to subset data (Techniques V1 and V4 of Table 2)
This is a good technique especially when you expect to get back a fraction of the records (about 12.5% of the records came back in our profile experiment). It is faster since only qualifying rows are pulled in at the outset which reduces the I/O, albeit sequential. We did not create an index for our sub setting experiment but the WHERE clause can use an index. Index usage allows for 'fast' disk based non sequential I/O.

Using HASH objects (SAS V9 only) (Technique V2 of Table 2)
This technique loads the values to be searched in memory as HASH objects. Consequently, the performance gained may be partially attributed to memory based non sequential I/O. However, the HASH object has to be recreated each time the program is run.

Using Proc SQL (with and without sorting the 'big' data set) (Technique V3 of Table 2)
PROC SQL automatically uses an inbuilt feature called the optimizer. Before executing the SQL code, the optimizer chooses the minimal I/O cost method by utilizing indices (if available) and other methods. PROC SQL's 'plan', generated by the optimizer, can be acquired for any given SQL query without executing the query by prefixing the PROC SQL statement with 'PROC SQL _method nonexec' option. Occasionally, the optimizer may 'plan' to sort the big data set. Given that sorting a big data set can be a time consuming task, another non sorting method may be considered in such as situation. In our profiled queries, PROC SQL did not sort as per the execution plan. Furthermore, PROC SQL's optimizer does not seem to care whether the big data set is sorted or not. (23 versus 23.6 minutes).

Using a format to subset with a WHERE clause. (Technique V5 of Table 2)
1. The unique values to be searched are written to a format where all the values are set to a single known value such as '999'.
2. As illustrated in the syntax for Technique V5 of Table 2, the PUT function is used with the format and the search field (i.e. formvar) from the big data, to output those records whose format based value resolves to '999'. PROC FORMAT's memory based I/O makes this method faster than the IF method. Even though PROC FORMAT is slower than the WHERE clause method (18.9 versus 31.2 minutes), it has potential as a 'merge without a sort' technique as described in a later section of the paper.

Using SORT/MERGE (Technique V6 of Table 2)
Sorting a big data set is often slow and may require a lot of disk space. The MERGE of data sets after sorting is also another DATA step. Sorting cannot be avoided if there is a need for BY logic with a SET statement associated with a sorted data set. A few methods for optimizing sorting are discussed later in this paper.

A FEW WAYS TO SIMPLIFY PROGRAM OVERHEAD ...

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Time Taken</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline : 2 datasets (large=300 million, small=1 record) are being concatenated</td>
<td>27.2</td>
</tr>
<tr>
<td>Time needed for ‘SET LARGE SMALL’ syntax</td>
<td></td>
</tr>
<tr>
<td>Note: I/O needed for reading both data sets and creating a new data set.</td>
<td></td>
</tr>
<tr>
<td>The order of data sets in the SET statement does not matter</td>
<td></td>
</tr>
<tr>
<td>Time needed for ‘PROC APPEND BASE=BIG DATA=SMALL’ syntax</td>
<td>0 (!)</td>
</tr>
<tr>
<td>Time needed for ‘PROC APPEND BASE=SMALL DATA=BIG’ syntax</td>
<td>16.6</td>
</tr>
</tbody>
</table>

All time in minutes. These tests use SAS V8.2
• Use proc append to concatenate data sets instead of using ‘SET X Y’ syntax (where X,Y represent data sets to be appended).
Assume that you have 2 data sets called ‘large’ and ‘small’. The ‘large’ data set has 300 million rows. The ‘small’ data set, identical in structure to the ‘large’ data set, houses 1 row.

The SET statement concatenation method of these two data sets is far less efficient than PROC APPEND. Sequential I/O is the clear culprit for the SET statement’s inefficiency as it reads both the data sets and then writes the third data set.

With PROC APPEND, the data set specified with the ‘DATA = ’ component can improve performance significantly as only that data set is read. It takes a negligible amount of time, using ‘PROC APPEND BASE=BIG DATA=SMALL’ syntax as only the ‘small’ data is read and appended. However, when ‘PROC APPEND BASE=SMALL DATA=BIG’ is used, it takes 16 minutes to read in and append the ‘large’ data set. This is, however, still faster than the ‘SET LARGE SMALL’ method.

<table>
<thead>
<tr>
<th>Table 4</th>
<th>Reduce Division-by-zero and IF statement overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Scenario</strong></td>
<td><strong>Time Taken</strong></td>
</tr>
<tr>
<td>Time needed for with SAS checking divide-by-zero for 300 million records</td>
<td>36</td>
</tr>
<tr>
<td>Time needed for 300 million records for division of only those records with a non-zero denominator (using an IF statement)</td>
<td>28.3</td>
</tr>
<tr>
<td>Use Plain IF to compute another variable based on 7 values of single variable for 300 million records</td>
<td>28.5</td>
</tr>
<tr>
<td>IF statement with ELSE computation for 7 values of a single variable for 300 million records</td>
<td>26.2</td>
</tr>
</tbody>
</table>

6 All time in minutes. These tests use SAS 8.2.

• Reduce Divide-by-zero exception overhead and minimize IF–THEN-ELSE overhead
In a DATA step division operation, when the denominator is zero, SAS has to generate missing values for the computed variable and issues warnings. When processing millions of rows, dividing only when the denominator is greater than zero can reduce the processing overhead.

• Using IF-THEN-ELSE statement instead of plain IF-THEN statement in a DATA step
Use this technique when making complex calculations with many IF-THEN contingencies. Given the mutually exclusive nature of the IF-THEN-ELSE statement, SAS can avoid the program overhead of traversing every IF-THEN contingency for each row depending on the value being checked.

If the frequency distribution of the variable being evaluated in the IF-THEN-ELSE clause is lopsided i.e. one value is far more frequent that all others, placing the most frequent value on the top of IF-THEN-ELSE statement(s) helps improve performance as other infrequent IF-THEN-ELSE contingencies are bypassed.

• If possible, combine and avoid redundant data steps. Use DATA _NULL_ i.e. do not create un-needed data sets while creating reports.

SORT OUT SORTING HASSLES ....

1. Reduce data set size (horizontally and vertically)
DATA set size reduction has a positive effect in reducing the sorting overhead. Consider using the WHERE clause and KEEP/DROP directly with PROC SORT to subset rows and columns and thus avoid a DATA step and its I/O.

2. Consider using TAGSORT option (50.5 minutes versus 57.4 minutes for a ‘regular’ sort; SAS V8)
The TAGSORT option is handy if the length of the BY variables is small compared to the total row’s length. This technique first sorts key variable(s) and then pulls in the remaining variables of the data set. As a result, temporary disk space overhead is reduced but it can be slower than a regular sort. Many other SORT tuning options are available.
3. Consider SORTEDBY dataset option
If a 'big' data set being read in (as a text file) is already sorted, the SORTEDBY option allows for the creation of variables such as LAST.byvarname or FIRST.byvarname without actually sorting the data in SAS.

A SORT/MERGE can also be simulated. A few methods are …

1. PROC FORMAT can be useful when you need to merge a 'big' and a 'small' data set.
   a) The 'small' data set's key and associated non key values to be merged can be put in a format where all the unique key values are set to a specific text string. The text string is composed of the corresponding, concatenated non key values which need to be 'merged'.
   b) The format also needs to have an 'other' value set to a string such as 'nofind' to represent non-matching records.
   c) The PUT function is used, as specified in the syntax for technique V5 of Table 2, to locate matches i.e. where PUT function's output applying the format is not equal to 'nofind' value(s).
   d) After the format is applied, the string from the format for each matched/merge value record can then be parsed into its constituent parts.

2. PROC SQL
PROC SQL's cost based optimizer may use an index (to JOIN) on the 'to be' merged BY column(s) of the 'big' data set to speed up the process. You might want to first check out the optimizer’s plan before executing the query. Please refer to comments associated with Technique V3 of Table 2.

3. Use HASH objects (Version 9 only)
HASH object, in addition to facilitating searching by a key, can simulate a merge as it supports the specification of all non key values with the key in the hash object. Given that the HASH object lives in memory, please remember that RAM is not an unlimited resource.

SEARCHING FOR A NEEDLE IN A HAYSTACK

<table>
<thead>
<tr>
<th>Label</th>
<th>Scenario/Technique</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1</td>
<td>Look up 125 records from 300 million unique values&lt;br&gt;Baseline: Time taken using 'IF v_id in (x,y,z)' syntax&lt;br&gt;Time taken using 'WHERE v_id in (x,y,z)' syntax</td>
<td>43.1 4.1</td>
</tr>
<tr>
<td>L2</td>
<td>Return 72,000,001 rows using Non Unique field with 100 look up values&lt;br&gt;Time taken using 'IF formvar in (x,y,z)' syntax&lt;br&gt;Time taken using 'WHERE formvar in (x,y,z)' syntax</td>
<td>44.2 9.2</td>
</tr>
<tr>
<td>L3</td>
<td>Look up 100 records from 300 million unique values&lt;br&gt;Time taken using SET=KEY option</td>
<td>.2</td>
</tr>
<tr>
<td>L4</td>
<td>Using 'SET=KEY' option, look up non unique values from 300 million records for 100 distinct values (i.e. one-to-many)</td>
<td>NA</td>
</tr>
<tr>
<td>L5</td>
<td>Look up 100 unique values from 300 million unique value records&lt;br&gt;Time taken using PROC SQL syntax</td>
<td>3.2</td>
</tr>
<tr>
<td>L6</td>
<td>Return 297,000,001 records from 300 million records for 100 distinct values&lt;br&gt;Time taken using 'PROC SQL' syntax</td>
<td>43.5</td>
</tr>
</tbody>
</table>

7 all Time in minutes. All the programs were run using SAS 8.2

1. WHERE clause (Technique/Scenario L1 and L2 of Table 5)
The WHERE clause and PROC SQL (unlike the IF clause) are capable of using SAS indices. Creating a SAS index for a column which is used in a look up can reduce I/O cost through the disk based non sequential access method. Some relevant issues to be considered before creating an index are ….
   • Only big data sets that need to be searched efficiently should be indexed strategically and sparingly. However, indices have an I/O and disk storage ‘cost.’ For a 53 GB data set, with 2 indexed columns, the size of the index was 7.2 GB.
   • Consider only columns which are searched routinely. In a compound index with multiple columns, the left most column should be used in a WHERE clause or the index may not be used.
   • Track index usage by using the ‘OPTIONS MSGLEVEL=I’ option. PROC SQL’s optimizer’s plan also indicates the usage of an index. If an index is consistently underutilized, delete it and save space.
   • An index is updated when the associated data set is updated. An index, to justify its cost of maintenance, should to be used (i.e. read) often and updated minimally.
The type that indices that BASE SAS creates are more suitable for highly unique columns. Unfortunately, often, efficient look up/sub setting by non unique fields such as NDC codes are needed by researchers using health care claims. NDCs are non unique in that there may be thousands of unique NDCs dispersed amongst hundreds of millions prescription claims. DBMS’s such as Oracle boast of a type of index, suitable for non unique column searching, called the ‘bit mapped’ index. SAS’s SPD server, but not BASE SAS, also has this feature.

**SET=KEY** method (Technique/Scenario L3 and L4 of table 5)
It is a good method to reduce I/O to look up rows based on a unique value column which is indexed. The index on the searched column V_ID (in the simulated data set), used in the experiment, is specified as unique. However, for a ‘one-to-many’ relationship key, it retrieves only the first matched record.

2. **PROC SQL** (Technique/Scenario L5 and L6 of table 5)
PROC SQL’s I/O ‘cost based’ optimizer determines the use/non use of an index for an indexed column specified in a WHERE (i.e. JOIN) clause. To access the optimizer’s plan, please consult comments associated with Technique V3 of Table 2.

The index on column V_ID was used in scenario L5. Index usage implies non sequential disk based I/O and it made the query ‘fast’.

In scenario L6, the optimizer did not use the index for the column formvar even though an index was available for this column. This probably happened since the query was expected to return more than 95% of the records i.e. it is more efficient to execute a simple sequential search rather than reading the index, if it is used, in addition to the original data set.

3. **HASH** objects (Version 9 only)
HASH objects, which involve memory based I/O can also be used for look up. HASH objects, being memory resident, have the advantage that they do not, unlike disk based indices, use disk space. However, they are re-created every time the program is run. Further, since there is lot less RAM than disk space in a typical SAS installation, you might want to be cautious about loading millions of keys of a big data set into the memory as a HASH object.

**QC/VALIDATION FOR LARGE DATA SET PROCESSING**

The quality of health claims data can be an issue as it typically comes from external sources. Consequently, one never seems to be completely out of the development phase as fresh data keeps coming in.

A random and representative sample can be a good start for validating processing and intensive QC. Intensive QC with samples is probably more efficient compared to iterative development/QC churn with large volumes. Of course, even if the sample passes the QC/validation test(s), it still does not guarantee that the production level QC will be acceptable. A ‘representative’ sample is, thus, critical. Even if you have a ‘good’ sample, your production level QC, maybe a bit less intense and comprehensive than QC with your sample, is still needed.

The more you ‘touch’ the data, the greater is the potential QC burden. Splitting up production/sample data and considering numerous steps to process these data chunk(s) may be an appropriate programming strategy to deal with the volume. However, from a QC/validation perspective, you need to make sure that the data is split, processed and put together correctly.

A client representative’s active and routine participation (and maybe commiseration!) will have a salutary effect on the credibility of the QC plan and the quality of the data.

You might want to compare a ‘raw’ set of frequencies and record counts for critical variables with the corresponding ‘final’ numbers, factoring in algorithmic changes to the data. Comparing final and raw frequencies/ranges etc. for critical variables, across time to spot anomalous trends (if available) can be useful.

If you need to collapse records, for example, to get ‘adjudicated’ cost(s) based on specified summation criteria for each patient ID, direct ‘raw’ data to ‘final’ row count and frequency comparisons may not be appropriate. Therefore, consider reconciling data computations by following the transformation algorithm for a small (and random) selection of patients from the final ‘version’ back to the ‘raw’ data.
CONCLUSION AND SUMMARY

A few SOPs/tips and programming techniques for processing large data sets have been presented in this paper. The discussion, while by no means comprehensive or complete, hopefully helps the reader to look beyond the ‘small data set’ modus operandi when confronted with processing big claim data sets.

Tackling the I/O burden of the SAS program while adopting appropriate big-data-volume-processing footprint sensitive SOPs might be a good beginning. If need be, one can then consider potentially costlier options such as, changing the SAS version, ‘tune’ the operating system and acquire more disk space or more powerful hardware etc.

Reducing ‘slow’ sequential I/O and maximizing non sequential memory or disk based I/O is a recommended approach for successful and efficient SAS programming with big data sets.

The simplest way of reducing I/O is to reduce data set size. This can be done by SAS data set compression, reducing the size and the number of columns and minimizing the number of rows needed for a programming task at hand. The judicious use of the LENGTH statement along with appropriate KEEP/DROP of variables are some column based methods. One could limit the number of rows by using techniques such as the WHERE clause, PROC FORMAT, PROC SQL and the HASH object (version 9).

PROC FORMAT, PROC SQL (with an index if appropriate) and the HASH object, with some modification(s) in context, can be also be used in DATA step operations such as look up. Further, they can also be considered as alternatives to the oft onerous merge/sort operation when a ‘very big’ data set is involved.

TRADEMARK INFORMATION

SAS and SAS/MACRO software are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration. Other brand and product names are registered trademarks or trademarks of their respective companies.

REFERENCES

SAS Companion for the Microsoft Windows Environment, Version 8
SAS Companion for the UNIX Operating System Version 8
Andrew Wilcox, SUGI 25, Efficiency Techniques for Accessing Large Data Files, Indianapolis, IN, 1999
Tony Brown, A Practical Approach to Solving Performance Problems with the SAS System, SAS Institute Inc., Dallas, TX, 2001
Bart Heinsius, Querying Star and Snowflake Schemas in SAS, SUGI 26, Long Beach, CA, 2001
Paul M. Dorfman, Koen Vyverman, SUGI 30, Data Step Hash Objects as Programming Tools, Philadelphia, PA, 2005

ACKNOWLEDGMENTS

The author wishes to thank M. Coughlin, K. Harkins, D. Usavage, A. Golla, R. Ganesan, G. Havildar, L. Pyle and X. Zhu for their comments and feedback on this paper.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:
Gopal Rajagopal
Senior Programmer/Analyst, Scientific Programming
Merck Research Laboratories
P.O. Box 4, BL3-7
West Point, PA 19486-0004
Phone: (484) 344-3810
Email: gopal_rajagopal@merck.com