STRATEGIES FOR ADVANCED MERGING APPLICATIONS
USING PROC SQL AND THE DATA STEP

JUDITH MOPSIK, Atlantic Research Corporation Information Systems Division
JEFFREY PHILLIPS, Atlantic Research Corporation Information Systems Division
MEG SELDIN, Atlantic Research Corporation Information Systems Division

Introduction

One of the hallmarks of SAS® software since its inception has been the powerful data manipulation tools it provides. The very first releases of SAS in the 1970’s provided users with a MERGE statement that could take the place of many lines of third-generation code used to do the same thing. At the same time however, SAS, through its DATA step programming language, has enabled users to take control of almost all aspects of file manipulation when such “manual override” is needed for the sake of efficiency or special problem solving. A merge application, for example, can be coded without using the MERGE or BY statements at all.

This combination of power and flexibility has given rise over the years to debate about what technique is best to use for many standard applications, especially those that involve combining lists to retrieve information. Merging, concatenating and joining data files are some of the most common data processing tasks. Until the arrival of Version 6.06 of the SAS System, the debate about file combination methods focused on the three DATA step techniques available: SET, MERGE, and UPDATE. Generally, the discussion centered on the machine efficiency of SET vs the ease of MERGE and UPDATE.

Version 6.06 of SAS has given programmers a new tool to use for manipulating multiple data sets: PROC SQL. PROC SQL contains an entire new language for data querying and manipulation; indeed, the whole vocabulary is new: “row” replaces “observation,” “table” replaces “data set,” “column” replaces “variable,” and so on. PROC SQL links SAS directly to the relational DBMS world and has powerful new capabilities such as the creation of data views and the simultaneous reading of SAS and non-SAS data via the SAS/ACCESS facility. At the basic level, however, PROC SQL is another tool on the programmer’s workbench. Like the DATA step, PROC SQL can be used to manipulate data in an almost infinite variety of ways. The question this tutorial considers, however, is whether PROC SQL is better than the DATA step for a task that both can perform.

Much of the current work at Atlantic Research Corporation’s Information Systems Division involves selecting the most efficient approach to a programming problem from among a variety of techniques. It is very rare that only one method is suitable for an application; the problem is to pick the best methods, weighing various measures of efficiency and effectiveness against each other for the benefit of a particular client. Among the factors to consider are:

- The machine efficiency of the code in terms of CPU time or I/O
- The human efficiency of the code in terms of development time

We have distilled our approach to this type of problem into a single example that will illustrate all of the factors described above, and will also show that “newer” is not always “better,” even in the world of data processing. As our example, we have selected a one-to-one match-merge application, producing reports of the matching and non-matching records. The results of this study compare three strategies using the DATA step with the MERGE, SET or UPDATE statements and two programming strategies using PROC SQL, to accomplish the same results. The efficiencies were compared by benchmarking the different approaches on the same hardware platform, as well as by benchmarking identical solutions on different hardware platforms. Programming and debugging time were also taken into consideration and are discussed along with the results. The following factors are evaluated:

- What hardware platform will be used
- What operating system will the SAS system be running under
- How large will the data set be
- What will the skill level of the programmer be

The following hardware and software configurations were used for benchmarking the elapsed CPU time:

- A MicroVax II running VMS and SAS 6.07
- A 30386/25 PC running OS/2 and SAS 6.06
- An IBM RS/6000 model 590 running AIX version 3.0 and SAS 6.07

In each case we used the most recently released version of the SAS system in order to take advantage of the enhancements that the SAS Institute has incorporated into the software. Performance efficiency was previously addressed on the IBM mainframe platform (reference 2). Similar results were achieved.

Programmer productivity was measured by the number of lines of code, the ease of code development, and the time to debug the solution. All of the programmers were SAS developers with at least ten years of SAS software development experience.
Statement of the Problem

Often we need data that are stored in separate data files or tables. In the example used for this paper we are presenting a banking problem that requires reports that link account information with current bank balance and activity data. Information from separate data files will be combined by matching key values, such as account number, that are available in both files. The final report will link information that was formerly available in each file separately.

We often do not have a choice of the hardware platform to be used to accomplish a task. However, the strategy for achieving the desired result is in the hands of the software developer. The approach has three major components. First, what result is to be achieved? Is the result a one-to-one match merge or a join resulting in all possible combinations of all matching observations? Second, what tasks are required to prepare the data for the merge? Must variables be renamed, must some subsetting of the data take place prior to the merge, or must data be sorted? Finally, what is the most practical approach in terms of programmer productivity and which approach will execute in the least amount of time (i.e. machine efficiency)?

Merging and Joining Data Sets

The process of bringing data from two sources together in order to present a more complete report or analysis of the data, is referred to as “merging” the data. While it is not required that the data have a common “key” identifier in order to match related data, the match-merge using one or more key variables is the most common application.

When every entry in one data source in a match merge has a match in the other data source, we refer to this as a “one-to-one” match merge. The one-to-one match is the simplest case of a match merge, and is relatively rare in the real world. When situations occur such as multiple entries for a key or missing entries for a key field, they must be managed in the computer program by either discarding the mismatches and/or duplicates, or by separating them out for review or further processing.

The merge has been traditionally performed in the SAS system using the DATA step with the SET, MERGE, or UPDATE statements. When using these features of the SAS language, the data sources must be sorted or indexed by the matching or key variables. The results of the merge can be presented by using the procedures such as PRINT, TABULATE, or REPORT as well as standard report writing techniques used in the DATA step.

With the release of SAS Version 6.06, the SQL procedure is also available for merging data. SQL can merge data in two basic ways: an SQL merge is performed when a SELECT statement references two or more input tables. The SQL merge, like the DATA step merge, is usually controlled by matching on one or more key variables. The mechanism for that control in SQL is the WHERE clause.

The second method of performing a merge in SQL only the last record for each account with the new, updated account balance. This solution is relatively straightforward since the MERGE statement automatically handles so much of the match processing.

The third solution (Figure 3) uses the UPDATE statement of the SAS DATA step. Except for the use of UPDATE, the solution appears identical to the MERGE solution in Figure 2. In fact, the special features of update (protection against overwriting nonmissing data in the master file and automatic application of multiple transactions to the same master record) are not used here, and UPDATE becomes merely a “synonym” for MERGE.

The next two solutions use the PROC SOL. In order to achieve the results of the standard merge described above, we used the LEFT JOIN operation. This gave us the same effect as the IN= on the MERGE or UPDATE statements. The computation for accumulating the final account balance was accomplished by using the SQL SUM function and Boolean expressions containing the transaction type. For a debit, the expression TYPE='D' would test true, resulting in a 1. The other expression, TYPE='C', would test false, resulting in a 0. Multiplying the Boolean results by the transaction amount results in:

\[ \text{BALANCE} + (0 \times \text{AMOUNT} - 1 \times \text{AMOUNT}) \]

And thus the debit would be subtracted from the balance. For a credit, the following would result:

\[ \text{BALANCE} + (1 \times \text{AMOUNT} - 0 \times \text{AMOUNT}) \]

So the credit would be added to the balance.

Working with the data in sort order was achieved with the GROUP BY statement. The ORDER BY statement orders the resulting output data sets WORK_ERRORS and WORK_ACCOUNT.

These solutions, while requiring very few programming statements, required a lot of research, since SOL was new to us. In particular, the interaction of the SUM function, the GROUP BY clause, and the DISTINCT operator required to achieve the correct results took a fair amount of experimentation to work out.

In the fourth solution (Figure 4), we used PROC SOL to accumulate the account balance totals as it was accessing the data set SUGI.TRANS and identifying the matching ACCT_NO. The first CREATE statement builds the SAS Data set that has all of the data for the master accounts with the resulting account balance. The second CREATE statement builds the SAS Data set that has the error transactions. It was necessary to perform both operations since PROC SOL can only output one data set (or View) in a CREATE statement.

The last solution (Figure 5) again uses PROC SOL, but has three CREATE statements. In an attempt to maximize the efficiency of SOL, we performed the computation for accumulating the final account balance in the CREATE statement. This had the effect of creating a temporary VIEW that had accumulated each account’s transactions. The remaining CREATE statements are identical to the previous solution. A substantial side benefit to the multi-statement process in Figure 5 is that the code is much more straightforward.
"Efficiency" vs "Effectiveness"

Evaluating code techniques is usually done in terms of the efficiency of the code. However, "efficiency" is a term that usually reflects hardware speed or performance measurements, and does not help evaluate a technique in functional terms. Efficiency as a concept also has the drawback of downplaying the human factor in code development and maintenance. In order to broaden our perspective, then, we have chosen the term effectiveness to describe how well a code technique "gets the job done".

Depending on the level of the "merge" that is required, there may be one or many different solutions to producing the desired results. The test application illustrates three DATA step solutions of merging, and two variations of PROC SOL. Each method accomplished the task. The question of "effectiveness," then, must be taken in more detail if we are not to throw up our hands and say "either way works". One criterion for evaluating the effectiveness of the code solutions in terms of number of code lines, on the theory that the less code, the less maintenance. The following table illustrates the number of source code lines for each method:

<table>
<thead>
<tr>
<th>CODE METHOD</th>
<th>APPROX. LINES</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATA step/SET</td>
<td>30</td>
</tr>
<tr>
<td>DATA step/MERGE</td>
<td>20</td>
</tr>
<tr>
<td>DATA step/UPDATE</td>
<td>20</td>
</tr>
<tr>
<td>SQL2 CREATE's</td>
<td>16</td>
</tr>
<tr>
<td>SQL3 CREATE's</td>
<td>22</td>
</tr>
</tbody>
</table>

Basing code effectiveness on the number of source lines is inaccurate at best, since the number of lines differs if coding conventions, terminal line size change, or other factors change. Using approximate lines as a guide, however, the SQL in Figure 4 is the winner.

Another measure of effectiveness is the amount of programmer time the code takes to write and debug. The following table illustrates our experience for this application:

<table>
<thead>
<tr>
<th>CODE METHOD</th>
<th>APPROX. TIME (Hrs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATA step/SET</td>
<td>2.0</td>
</tr>
<tr>
<td>DATA step/MERGE</td>
<td>0.5</td>
</tr>
<tr>
<td>DATA step/UPDATE</td>
<td>0.5</td>
</tr>
<tr>
<td>SQL2 CREATE's</td>
<td>6.0</td>
</tr>
<tr>
<td>SQL3 CREATE's</td>
<td>5.0</td>
</tr>
</tbody>
</table>

Using this measure, the MERGE was the clear choice. This measure varies, of course, depending on the background and experience of the programmer. Our programmers were experienced SAS DATA step programmers, but were relatively new to SQL. Thus, programming time is not so much a pure measure of code effectiveness as it is a measure of how effectively a given individual can use the particular code tool.

Programming and maintenance efficiency are important, if difficult, measures of code effectiveness. Another very important measure is machine efficiency. Machine efficiency, measured in CPU, time, memory used, or I/O time, is often the measure used to judge a code technique. While we do not agree that machine efficiency alone should determine whether a technique is good or bad, it is important, especially in production applications. As code is run more and more frequently, its machine costs continue to add up while its human costs increment only when maintenance is required.

Table 1 presents all of the timings for each step in the processes described above. The comparison of CPU times for just the DATA step vs. PROC SOL are presented graphically in Figure 6. The graph is especially dramatic: while there are differences between the individual DATA step and SQL techniques, it is obvious that any DATA step technique outperforms any SQL technique. The differences hold across platforms. Though other measures, such as I/O, should be taken into account, these results indicate that for an application such as ours, the DATA step may well be a better choice.

Conclusion

All of the approaches "got the job done". The PROC SQL techniques had fewer actual statements, but required significantly more programmer time. This was due to two factors. First, we were not SQL experts, and the structure of a complex SQL statement presented a challenge for us. Therefore, the "learning curve" was an important factor in that it took us approximately 5-6 hours to develop and debug the successful SQL solution. On the other hand, the effort required to develop a DATA step with the SET statements accomplishing the merge took about 2 hours to code, debug and test. The simplest solution for the experienced SAS programmer was accomplished using either the MERGE or UPDATE in the DATA step. Those solutions took only 30 minutes each from coding through testing.

The CPU time for each approach varied somewhat based on the platform, but the pattern of overall performance was the same. In every case the DATA step solutions required significantly less CPN time.

Our first conclusion, then, is that "newer" is not always "better". SQL has significant advantages over the DATA step in certain functional areas, and vice-versa. However, PROC SQL is apparently a very resource-intensive procedure in SAS, and that fact must be taken into account when deciding on a coding approach. Neither code methodology is the ideal solution for all applications.

References


Rabb, Merry G. "Using PROC SQL in Data Management Applications", NESUG '90 Proceedings; SAS Institute, Inc.

The authors can be reached at the following address:

Atlantic Research Corporation
Information Systems Division
1301 Piccard Drive, 2nd Floor
Rockville, Maryland 20850
(301)-258-5542
Figure 1

/**********************************************************/
/* Function: Illustrates Master file update */
/* using SET statement where there */
/* ARE account/trans mismatches */
/**********************************************************/

libname sugi '..............';
proc sort data = sugi.trans out = trans; by acct_no; run;

data account(drop=amount type trans_no old_trns)
  no acct(keep=trans_no amount type);
  retain trans_no /* Initialize transaction acct no. */
  old_trns '000-00-0000' /* and transaction type */
  type ' ';
  do until(lastacct);
    set sugi.account end = lastacct; /* read next main record */
    do while(trans_no <= acct_no and lasttran);
      if trans_no = acct_no then do;
        if type = 'C' then balance + amount;
        else balance + (-amount);
        end; /* apply transaction if account matches */
      else do; /* output trans-with-no-master */
        if old_trns ^= '000-00-0000' then output no acct;
        else do; /* output trans-with-no-master */
          set sugi.trans(rename=(acct_no=trans_no)) end = lasttran;
          old_trns = trans_no; /* store trans_no from previous SET */
          set trans(rename=(acct_no=trans_no)) end = lasttran;
        end;
      end; /* read transactions */
      output account;
    end; /* read and output any transaction records remaining after */
    /* ACCOUNT file is exhausted */
    do until(lasttran);
      if trans_no ^= acct_no then output no acct;
      set trans(rename=(acct_no=trans_no)) end = lasttran;
    end;
  stop;
/**********************************************************/
Figure 2

BEGIN PROLOGUE; Illustrates Master file update
/* Function: Illustrates Master file update */
/* using MERGE stmt where there are account/trans mismatches */
BEGIN PROLOGUE; Illustrates Master file update
END PROLOGUE;
libname sugi '...........';
proc sort data = sugi.trans out=trans;
by acct_no;
run;
data account(drop=amount type)
   no acct(keep=acct_no amount type);
merge sugi.account(in=mast)
   trans(in=trans);
   by acct_no;
if mast and not trans then output account;
else if trans and not mast then output no acct;
else do; /* records in both files, process balance */
if type = 'e' then balance + amount;
else balance + -amount;
if last.acct_no then output account;
end;
run;

Figure 3

BEGIN PROLOGUE; Illustrates Master file update
/* Function: Illustrates Master file update */
/* using UPDATE stmt where there are account/trans mismatches */
BEGIN PROLOGUE; Illustrates Master file update
END PROLOGUE;
libname sugi '...........';
proc sort data = sugi.trans out=trans;
by acct_no;
run;
data account(drop=amount type) by acct_no;
   no acct(keep=acct_no amount type);
update sugi.account(in=mast)
   trans(in=trans);
   by acct_no;
if mast and not trans then output account; /* no change to balance */
else if trans and not mast then output no acct; /* no master record matches */
else do; /* records in both files, process balance */
if type = 'c' then balance + amount;
else balance + -amount;
if last.acct_no then output account;
end;
/* Function: Illustrates Master file update */
/* using PROC SQL in one step */
*******************************************************************************/
options fullstop time msglevel=1;
TITLE 'SUGI 17 PAPER ON SET/MERGE VS SQL';
libname sugi '[house.sugi17.mjj]';
title2 'SQL Using Queries';
title3 'Join Account, Debits, and Credits';
title4 'Creating Final balance in Two Passes with One Query';
proc sql;
create table work.account as
  select distinct a.acct_no as acct_no, fname, lname, balance,
    balance + sum((b.type='C') * b.amount)-(b.type='D') * b.amount)
  as new_bal format=dollar10.2
from sugi.account as a left join sugi.trans as b
on a.acct_no = b.acct_no
  group by acct_no
  order by acct_no
/ * find error transactions */
create table work.errors as
  select b.acct_no as acct_no, amount, type
from sugi.trans as b
  where b.acct_no not in 
    (select a.acct_no
     from sugi.account as a)
; quit;

********************************************************************************
libname sugi ' ................ ';
title2 'SQL using Queries';
title3 'Join Account, Debits, and Credits';
title4 'Creating Final balance in Two Passes with One Query';
proc sql;
create view work.trantot as
  select distinct b.acct_no as acct_no,
    sum((b.type='C') * b.amount)-(b.type='D') * b.amount)
  tran_tot format=dollar10.2
from sugi.trans as b
  group by acct_no
create table work.account as
  select a.acct_no as acct_no, fname, lname, balance,
    balance + b.tran_tot
  as new_bal format=dollar10.2
from sugi.account as a left join work.trantot as b
  on a.acct_no = b.acct_no
  group by acct_no
  order by acct_no
/ * find error transactions */
create table work.errors as
  select b.acct_no as acct_no, amount, type
from sugi.trans as b
  where b.acct_no not in 
    (select a.acct_no
     from sugi.account as a)
; quit;

Figure 5
FIGURE 6: ONE-TO-MANY MERGE TECHNIQUES

Merge Step CPU Times

CPU SECONDS

0 100 200 300 400 500 600 700

SET  MRG.  UPDT.  SQL/2  SQL/3

OS/2: IBM PS/2 386
VMS: MicroVAX II
AIX: IBM RS/6000
Table 1: CPU Times for Application Code Solutions  
(Times measured in CPU seconds)

<table>
<thead>
<tr>
<th>PLATFORM</th>
<th>PROGRAM FUNCTION</th>
<th>SET</th>
<th>MERGE</th>
<th>UPDATE</th>
<th>PROC SQL 2 CREATE's</th>
<th>PROC SQL 3 CREATE's</th>
</tr>
</thead>
<tbody>
<tr>
<td>OS/2</td>
<td>SORT</td>
<td>43.95</td>
<td>43.47</td>
<td>44.63</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>&quot;Merge&quot; Step</td>
<td>38.09</td>
<td>51.3</td>
<td>53.29</td>
<td>655.1</td>
<td>495.51</td>
</tr>
<tr>
<td></td>
<td>PRINT 200</td>
<td>3.23</td>
<td>3.06</td>
<td>3.53</td>
<td>3.7</td>
<td>3.81</td>
</tr>
<tr>
<td></td>
<td>PRINT 2</td>
<td>1.26</td>
<td>0.66</td>
<td>0.64</td>
<td>0.58</td>
<td>0.75</td>
</tr>
<tr>
<td></td>
<td>TOTAL</td>
<td>86.53</td>
<td>98.49</td>
<td>102.09</td>
<td>659.38</td>
<td>500.07</td>
</tr>
<tr>
<td>RS/6000</td>
<td>SORT</td>
<td>2.26</td>
<td>2.16</td>
<td>2.1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>&quot;Merge&quot; Step</td>
<td>1.92</td>
<td>3.37</td>
<td>3.54</td>
<td>38.28</td>
<td>31.57</td>
</tr>
<tr>
<td></td>
<td>PRINT 200</td>
<td>0.26</td>
<td>0.23</td>
<td>0.22</td>
<td>0.28</td>
<td>0.3</td>
</tr>
<tr>
<td></td>
<td>PRINT 2</td>
<td>0.05</td>
<td>0.04</td>
<td>0.04</td>
<td>0.05</td>
<td>0.05</td>
</tr>
<tr>
<td></td>
<td>TOTAL</td>
<td>4.49</td>
<td>5.8</td>
<td>5.9</td>
<td>38.61</td>
<td>31.92</td>
</tr>
<tr>
<td>MICROVAX II</td>
<td>SORT</td>
<td>31.15</td>
<td>31.12</td>
<td>31.13</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>&quot;Merge&quot; Step</td>
<td>17.72</td>
<td>33.95</td>
<td>35.2</td>
<td>464.52</td>
<td>291.66</td>
</tr>
<tr>
<td></td>
<td>PRINT 200</td>
<td>2.81</td>
<td>2.56</td>
<td>2.65</td>
<td>3.47</td>
<td></td>
</tr>
<tr>
<td></td>
<td>PRINT 2</td>
<td>0.61</td>
<td>0.54</td>
<td>0.51</td>
<td>0.53</td>
<td></td>
</tr>
<tr>
<td></td>
<td>TOTAL</td>
<td>52.29</td>
<td>68.29</td>
<td>69.49</td>
<td>468.52</td>
<td>291.66</td>
</tr>
</tbody>
</table>