Paper 261-29

Selecting Records and Assigning Attributes

Jimmy DeFoor, Citi Card, Irving, Texas

Abstract

This paper will discuss the primary techniques of selecting records and assigning attributes with SAS software and the relative strengths of each method. The vehicle for describing and relating these techniques will be the data processing problems faced by a group of marketing analysts in a fictional company in Cowtown, Texas.

Introduction

SAS users are very accustomed to using IF statements in the selection of records and assignment of attributes. Because IF statements are so flexible and easy-to-code, they usually handle the needs of either of those tasks, no matter how complex or extensive those tasks may be. As a result, the SAS user may have little motivation to venture into other techniques, no matter how easy those techniques may be to use. This paper will introduce the SAS coder to user-formats and indexing by showing the simplicity by which they can be implemented and the additional processing speed they can bring to the typical data processing task. It will also describe general guidelines the SAS user should remember when using these techniques.

Selecting Records, Part One

Two recurring tasks in data processing are selecting the records and assigning values or attributes to the variables of a data set. SAS users often write code as follows:

```
Data MySales;
  Set Sales;
  If City eq 'Fort Worth';     /* get sales from Fort Worth */
  If      Zip eq '76116' then  /* assign zip codes to area names */
    Area = 'Western Hills';
  Else if Zip eq '76108' then
    Area = 'White Settlement';
  Else if Zip eq '76126' then
    Area = 'Benbrook';
  Else                         /* delete other zip codes
    Delete;
  Run;
```

This code allows only records for people from Fort Worth to be processed by the assignment statements that follow. It works very effectively, but it requires every record to be read from the data set Sales. If that data set is quite large, processing can take many CPU minutes and even more clock time. That can create excessive resource usage and data contention if many workers are running the same type of programs against the same data sets.

In the marketing department of Cowtown Products, each worker runs these types of programs several times during the month. The execution times are quite lengthy, but everyone believes they are normal because they have nothing to measure against them. So they just plan their work accordingly: they submit their jobs before going home, just before lunch, or just before going to a meeting.

Then, 'Someone Has a Better Idea!' One day, when a member of the marketing group is asked to complete his monthly analysis by the next day, the pressure of having not run his job to pull Sales data earlier in the month forces him to brainstorm whether there is a way to get his program to finish sooner than usual. He remembers that the Sales data is sorted in ascending order by zip code so that the Sales management can get a monthly production report of the number of customers in each zip code and their total purchases for the month. The marketing analyst realizes that, if he adds a Stop statement to his code, his program will stop executing once it retrieves the Sales data for the last zip code he needs.

He quickly runs to his computer, opens his program, and then adds the following code before the delete statement in his program.

```
Else if Zip gt 76126 then
  stop;
```

Based upon the fact that the company has a relatively even volume of sales in every zip code, the marketing analyst predicts that the code will execute in 25% faster than it has before, saving him 30 minutes of clock time off of the usual run time of two hours. He then skips lunch and starts preparing PowerPoint presentations with the other information he already has available.

Sure enough, the job does finish 30 minutes faster than normal, so he is able to complete his analysis without delaying other work or working late.

Feeling cocky and proud, the analyst shares his discovery with his co-workers. Soon, all are following the same approach. Total resource usage for these types of programs drop 50%, with some analysts get their results very quickly and others getting their results with very little improvement in clock time since their jobs pull zip codes above 90000. This frustrates one analyst so much that he asks for and receives a transfer to another department.

The manager of the marketing analysis group decides that group would benefit from having a skilled SAS user as a member of the group and lists that skill as one of the requirements of the open position. Soon a new analyst is hired who has attended several SAS user conferences and has learned of a timesaving retrieval method called indexing. He explains to everyone that a great deal of processing time could be saved if the file was indexed by zip code. They agree and he writes and executes the following indexing code.

```
Proc Datasets Library = Sales;
  Modify Sales;
  Index Create Zip;
Run;
```

Excited, everyone runs to their computers, starts their programs and, then - as they have done for many years - they go to lunch. When they return, some of their programs are still running and all complete after the usual clock time. "What's Up!" they ask the new guy. Embarrassed, he returns to his cubicle and opens his Conference Proceedings. Quickly, he discovers his error.

'I forgot that IF statements don't use the index of a data sets,' he explains to his co-workers, 'we have to use Where statements instead. Add this code to the Set statements in your data steps.'

```
Data MySales;
  Set Sales(where=(Zip in ('76116','76108','76126'));
```

One of the analysts whose program retrieves zip codes in the 90000s decides to try the Where statement, believing that the change can't make his program process any slower. He is very pleased when the processing time for his program drops to 30 minutes from two hours. Because of his success, other analysts try the same change, but most don't get the same improvement. Those with the zip codes closer to 00000 get almost no improvement. The new analyst explains that indexes speed data retrieval by skipping records that aren't needed. The programs pulling zip codes close to 00000 don't process faster since those zip codes are near the top of the file.

The new analyst then has another thought. He wonders if he could accomplish the same result with a join in Proc SQL. First, he creates the SAS data set Zip with the zip codes he wants and then he executes the simple SQL code to the right below.

```
Data Zip;
  Format Zip $5.0;
  Input Zip;
    Cards;
      76116
      76108
      76126
      ;
Run;
```

```
Proc SQL;
 Create Table MySales as
  Select *  /* all variables */
   From Zip A
         Inner join
       Sales B
    On A.Zip = B.Zip;
  Quit;
```

He doesn't use a Run statement to end Proc SQL because Proc SQL submits the query when it encounters a semicolon. Any Run statement is ignored. He uses a Quit statement, however, because the SQL procedure continues to wait for another query unless a Quit statement is encountered or another Data Step or Procedure is initiated.

Reading his conference proceedings before he executes the code, he learns that the order of the tables in the From statement is important. He has to place the indexed file as the second table to ensure that the index will be used for searching the second table.

He then executes the code. The SQL code runs just as fast as the Where statements.

Assigning Attributes, Part One

Another marketing analyst, hearing that the new guy is very well paid, decides that she, too, must attend a SAS user conference. She visits the SAS web site and discovers that a regional one-day conference will be held In Fort Worth during the month of June. She quickly seeks approval to attend and gets it without complaint since the conference is so reasonable in cost and doesn't require travel. She finds the conference fun and rewarding. Particularly interesting to her were the papers on user-built formats and 'If Then Else' statements. Shortly after returning from the conference, she begins experimenting with formats.

First, she builds a spreadsheet with these ranges and labels:

Start	**Last**	**Label**
low	20	Drop-Ins
20	40	Occasionals
40	60	Small Potatoes
(*more rows*)	-	-
180	220	Average Buyers
(*more rows*)	-	-
300	400	Major Buyers
400	high	Choice Customers

In total, she creates a spreadsheet with 20 rows so that she can categorize sales for all zip codes in Fort Worth. Currently, she assigns these groupings with twenty 'If Then Else' statements.

```
If      Sales le 20 then
   Group = 'Drop-Ins';
Else if Sales gt 20 and Sales le 40 then
   Group = 'Occasionals';
      (more code)
Else if Sales gt 140 and Sales le 160 then
   Group = 'Average Buyers';
      (more code)
Else If Sales gt 300 and Sales le 400 then
   Group = 'Major Buyers';
Else if Sales gt 400 then
   Group = 'Choice Customers';
```

At the conference, she also learned that the order of the 'If Then Else' statements determines their efficiency. If they are ordered by the frequency of the occurrence of the values of the data, they can be very efficient. Though she didn't intend it when she wrote the code, she recognizes that the distribution of her data matches the order of her IF statements. There are far more customers at the lower end of the distribution than in the middle or at the top of monthly purchases.

But, she also learned that, if the number of assignment categories is more than a few, user-built formats are always faster in assigning values than 'If Then Else' statements. They are faster because they use binary search instead of sequential search to find the matching values. She realizes that the values of her data is not uniformly distributed, but thinks her number of assignment categories (20) could still make the format approach more efficient.

Unless the option Notsorted is specified, user formats are sorted in ascending order when built and are loaded into memory the first time a program calls them. They are then searched by halves. That is, the search goes to the middle of the format list, compares that value to the value being sought, and then goes to the middle of the remaining area of the format to be searched, which depends upon whether the value being sought is more or less than the value found in the previous search.

In this case, if the value 300 were being sought, the binary search would begin around 140–160 and then go up to the middle of the upper range, which would be about 220-260 (not shown). Finding that it was still low, the search would go the middle of the upper remaining range, which would be around 300-400. Finding a match, it would stop.

Formats stop binary searching when they get close to the value. They then start a sequential search. If the value of the data had been 280 instead of 300, the search would probably have converted to sequential after not finding a match at 220-260. The maximum number of binary searches can be estimated by solving this formula for B and rounding up to a whole digit.

```
B = log(number of attribute ranges) / log (2)
```

For 20 attribute ranges, the answer would be no more than 5.

Returning to our analyst, we next see her exporting the spreadsheet to a CSV file and then importing it via a SAS interactive session, which builds the code to read the data in into the SAS data set Grouping. She then recalls the code and modifies so that it will have the characteristics of the format she is going to build. She adds the Retain statement to set the other fields needed by the Proc Format. The modified code looks something like this:

```
Data  Grouping;
 Infile CSV delimiter = ',' missover dsd;
 Format Start 8.0  End  8.0  Label $15;
 Input  Start             /* Beginning of range */
        End               /* End of range */
        Label;            /* Text to be assigned */
 Retain fmtname 'grpfrmt' /* Name of format */
        type    'n'       /* Type of format: numeric or character */
        sexcl   'y'       /* Exclude starting boundary: yes or no */
        eexcl   'n';      /* Exclude ending boundary: yes or no */
```

After executing this code to read the CSV file into the Grouping data set, she uses the Cntlin option of Proc Format to transfer the data set into a user format.

```
Proc Format Cntlin = Grouping;
Run;
```

After loading the format, she copies her program and deletes her 'If Then Else' statements. She then adds one Put statement in their place that will perform exactly the same function..

```
Group = put(Sales,grpfmt.0);
```

After executing the modified program, the analyst compares execution times. The new program does runs faster than the old program, but – more importantly – if she ever has to make the same assignments to different data sets in the same program, she can do so by just repeating the same one line of code in each data step.

Of course, the analyst didn't have to build the user format using the Cntlin= option. She could have built it using the typical method:

```
Proc Format;
   Value grpfrmt
       low –  20  = 'Drop-Ins'
        20 -  40  = 'Occasionals'
        40 -  60  = 'Small Potatoes'
       (More format values.)
       300 – 400  = 'Major Buyers'
       400 – high = 'Choice Customers'
       ;
Run;
```

Encouraged, the analyst explores further applications of user formats. The customer classification code would be a good choice, but there are possible problems because it assigns three values not just one. Below is an example:

Classification	Offer	Media	Status
012	Travel Packages	Magazine	Young Family
121	Dining Out	Phone	Single Female

Each customer is assigned a classification of three numbers and there are 1000 such numbers. The codes define the type of offer, the media by which the offer was made, and a summary of the family status of the customer.

Fortunately, the analyst doesn't have to build a spreadsheet with the 1000 classification codes and their meanings. They are already located in an Access database controlled by the person who assigns the ratings and sends them in CSV format to the marketing company that provides reports on the performance of each classification. The analyst only has to ask for her own copy of the CSV file.

After creating a CSV file and importing the data into a SAS data set, she again recalls the code that created the data set and modifies it to create the variables needed for a Proc Format. She smartly eliminates the need to create three formats by concatenating the three variables into one new variable, which she conveniently names Label. Below is the code she adds to the program built by the SAS import facility. She doesn't need to enter the range control values of 'Sexcl' and 'Eexcl', because her format assigns discrete values, not ranges. She does have to rename Rating to Start, however, so that Proc Format knows which field contains the values to be used to assign the Labels.

```
Retain  Fmtname 'text'
        Type    'c' ;
Length  Label   $50;
Rename  Rating  = Start;
Label = Offer||'/'||Media|'/'||Customer;
```

When ranges aren't involved, the only variables are needed for user formats are:

- Fmtname
- Type
- Label
- Start

When the analyst uses the format to assign Alltext to a classification, she follows the Put statement that executes the format with three Scan functions that decompose the retrieved text into its three variables:

```
Alltext  = put(rating,$text.0);
Offer    = scan(alltext,1,'/');
Media    = scan(alltext,2,'/');
Customer = scan(alltext,3,'/');
```

The Scan function retrieves text between specified delimiters. In this case, the first Scan function retrieves the text before the first slash ('/'); the second, from the first slash to the second slash; and the third, from the second slash to the third. There isn't a third slash, of course, so the Scan function stops when it reaches the end of the word.

User-built formats are much faster than 'If Then Else' statements when assigning large numbers of attributes. Using our formula of b = log (number of attributes) / log (2), we see that each of the 1000 unique values in the format is assigned by no more than ten searches, while 1000 'If Then Else' statements would require an average of 500 searches to assign one value.

But if the number of attributes to be assigned had been only 50, the format approach would have still much faster: 6 searches versus an average of 25.

Assigning Attributes, Part Two

The female analyst is, understandably, quite excited with her work and shares with the other analysts in the group, including the new hire who introduced the use of indexed SAS data sets. He expresses true appreciation for her work, but wonders whether using an indexed data set would be even faster for assigning values – and more useful for him – than employing a user-built format.

He asks his female co-worker for access to her CSV file and the SAS code which imports it into a SAS data set. She gives it, but asks what he is planning. "I want to try assigning the fields with an index instead of a format," he replies. "I don't know if it will make any difference in processing speed, but it will enable me to use Proc SQL instead of a SAS Data Step".

The male co-worker imports the CSV file into the SAS data set Text and then uses Proc Datasets to index the data set by Classification.

```
Data Sales.Text;
  Infile CSV delimiter = ','
         dsd missover;
  Format Classification $3.0
         Offer  $20.0
         Media  $15.0
         Status $15.0;
  Input  Classification $
         Offer  $
         Media  $
         Status $;
Run;
```

```
Proc Datasets Library = Sales;
  Modify Text;
  Index Create Classification;
Run;
```

Then, he tests the index with the following SQL code:

```
Proc SQL;
 Create Table MyRatings as
   Select *    /* Keep all variables */
    From Sales B
    /* Keep all rows from Sales and those that match from Text */
         Left Outer Join
        Text A
      On A.Rating = B.Rating
Quit;
```

The SQL code executes quite quickly, but based upon the performance reported by the female analyst, it executes the text assignment no more quickly than did the user-built format. This motivates the new hire to read more about the performance of SAS indexes. He learns that they use a search technique similar to that employed in user-formats, so a faster speed in assigning text shouldn't normally be expected – with one exception. An index search doesn't automatically return to the beginning of the index list (unless the option Unique is specified). That means that, if the next classification being assigned is the same value as the previous classification, the index will assign the same text without executing a search.

Format searches, on the other hand, always return to the beginning of the list. Thus, if the value of the previous record required 10 searches to find a match, the next record would require the same 10 searches even if the value being sought were identical. The index would execute (0) searches. This means that index searches are especially suited for data sorted on the field that is indexed.

With all this in mind, the new hire decides to use the index approach because it lets him use SQL code and does not necessitate building a concatenated text variable that must he must later scan into three variables.

Selecting Records, Part Two

Life is good at Cowtown Marketing for quite a while. Jobs run quickly, leaving more time for analysis. There is less pressure on resources, so the system engineer spends less time balancing resources and filling out justifications for more space and processing power. Cowtown's owner doesn't have to spend as much money on new servers and disk drives. Everyone is quite content.

Then, one day, Marketing decides to cancel the monthly Sales report by zip code since the new process makes the report unnecessary. The programmer responsible for the report takes it out of the scheduling system. That job sorted the Sales data by zip code after it was updated with the latest information.

Immediately, everyone's queries by zip code begin running much slower than before. In fact, they take almost as much time as they did before the index was used. What happened, they all ask?

The answer lies in the problems and opportunities created by indexing. Before, when the data was sorted by zip code, the use of the index to get all of the sales in a zip code was accomplished by gathering contiguous records from the same area on the disk with one I/O (input/output) operation. Now, because the data is not sorted by zip code, the use of the index to retrieve all of the customers in a zip code requires that many I/O operations be executed.

Increased I/O not only slows down a single program that uses the data that is indexed, however. It slows down every program that uses that data because they all must wait until the disk area containing the data they need is available. This means that an index can actually slow down access to a data set if many programs are accessing the same data at the same time and they are all accessing more than a small percentage of the total records in the data set. If that is the case, the data must be sorted by the indexed field to reduce the I/O contention.

Again, it is the new hire that sees the probable solution. He calls the programmer in charge of the monthly update and asks if the sort was dropped. Finding that it was, he asks that be re-implemented. After the sort is executed, the index is rebuilt and the problem disappears. It does, so all is again well in Cowtown.

Conclusion

Assigning attributes and selecting records both involve examining values to see if they match certain criteria. The only difference in the two tasks is what happens after the match is found. In one case, the matching is followed by the adding of additional content. In the other case, it is followed by the action of retrieving or writing the matching record. In general, either IF statements, formats, or indexes can be used in assigning attributes or selecting records. Each technique has its strengths and pitfalls. These are some general guidelines for their usage.

Guidelines for Using If Statements, Indexes and User-Formats.

- Not counting the overhead of creating and storing either, formats and indexes are more efficient than IF statements for assigning attributes when the number of IF statements exceed more than a few: eight (8) for uniformly distributed data.
- Formats are as efficient as an index for assigning attributes to ungrouped data, plus formats require less overhead to create.
- Formats are better suited for assigning attributes from a range of values than are indexes.
- Indexes are more efficient than indexes for assigning single attributes to sorted data.
- Indexes assign multiple attributes without the extra coding effort required by formats.
- Indexes can be used for retrieving data, formats cannot.
- Indexes should be used for retrieving subsets of data (no more than 30%), not for retrieving all of the data. Sequential reads should be used for reading all of the data.
- Small subsets (1% to 5%) of data can be retrieved via index without the data being sorted by the indexed variable.
- Large subsets of data (20% to 30%) should be retrieved via an index only if the data is also sorted by the indexed variable.

References

SAS Institute Inc., Advanced SAS Programming Techniques and Efficiencies, Course Notes, 1999.

SAS Institute Inc. SAS Language Reference, Concepts, Version 8.2 *(see http://www.sas.com/service/library/onlinedoc/)*

Moorman, Denise J. and Deana Warner, SAS Institute, Inc., Cary, NC, “Indexing in the SAS System, Version 6”, SAS Online Documentation *(see http://www.sas.com/service/library/periodicals/obs/obswww08/index.html)*

Diane Olson, SAS Institute Inc., Cary, NC, “Power Indexing: A Guide to Using Indexes Effectively in Nashville Releases”, SAS Online Documentation *(see http://www.sas.com/service/techtips/ts_qa/guidetoindex.html)*

Contact Information

Jimmy DeFoor
Citi Card, Irving, Texas
Email: jimmy.a.defoor@citigroup.co