

Selecting Records and Assigning Attributes

Jimmy DeFoor, Citi Card, Irving, Texas

Selecting Records, Part One

Two recurring tasks in programming are selecting the records (rows) and assigning values or attributes to the fields (columns) of a data set (table). Programmers often write code as follows:

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

This code allows only records for people from Fort Worth to be processed by the assignment statements that follow. It works, 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, one day 'Someone Has a Better Idea!' One member of the marketing group learns that the Sales data is sorted in ascending order by zip code so that the Sales management can get a monthly report of the number of customers in each zip code and

their total purchases for the month. The marketing analyst realizes that he can modify the above code to stop processing the Sales data after retrieving the last zip code he wants. He adds the following code before the delete statement in his program.

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

The code causes the data step to terminate when it finds a record with a zip of 76127 or greater, preventing the reading of all the records that follow. Since the zip codes in the data run well past 99000, doing this saves him significant processing time.

The smart coder tells his co-workers about his discovery. Soon, all are following the same approach. Resource usage for these types of programs drops 15%.

Despite the general improvement, however, however, several of the department workers get only a small reduction in their CPU time or clock time. They examine zip codes above 90000, so their code still reads almost all of the records before finding the records they need.

The company has an opening and hires a marketing analyst. The new analyst has attended several SAS user conferences and has learned of a timesaving method called indexing. He convinces everyone that if the file is indexed by zip code, a great deal of time will be saved. They agree and he writes and installs the indexing code.

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

Everyone runs to their computers and starts their programs and then, as they have done for many years, they go to lunch. When they return, their programs have finished successfully but they ran just as long as before. "What's Up!" they ask the new guy. Embarrassed, he returns to his cubicle and

opens his *Conference Proceedings*. Quickly, he discovers his error.

'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')));
```

Everyone adds the Where statements and then retests their code. Processing time drops to less than a minute for everyone. Total resource usage for these selection programs department drops another 15%.

The new analyst then has another thought. He wonders if he could accomplish the same result with a join in Proc SQL. He creates the SAS data set Zip with the zip codes he wants. Then he writes this code.

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

He doesn't use a Run statement 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 (file) to ensure that the index will be used for searching the table.

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

Assigning Attributes, Part One

Another marketing analyst, not wanting to be out-done by the new guy, decides that she, too, must attend a SAS user conference if she is to be trailblazer and not a follower. She

visits the SAS web site and discovers that a regional conference will be held in Dallas in two weeks. She quickly seeks approval to attend and gets it without complaint since the conference is so reasonable in cost. She finds the conference fun and rewarding. Particularly interesting to her were the papers on user-built formats. Shortly after returning from the conference, she begins experimenting with formats.

First, she builds a spreadsheet with these types of values:

Start	Last	Label
low	20	Drop-Ins
20	40	Occasionals
40	60	Small
(more rows)	-	-
300	400	Major
400	high	Top Flight

In total, she creates a spreadsheet with 20 rows. She does this because she often looks at sales for all zip codes in Fort Worth.

Currently, she assigns these groupings via a set of 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 300 and
       Sales le 400 then
  Group = 'Major';
Else if Sales gt 400 then
  Group = 'Top Flight';
```

At the conference, she 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 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` ¹ 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 also binary searched. 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 (dividing by 2) 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 60 were being sought, the binary search would begin around 180–220 (not shown) and then go down to the middle of the lower range, which is about 80–100. Finding that it was still high, the search would go the middle of 0 to 80, which would be around 40–60. 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 70 instead of 60, the search would probably have converted to sequential after not finding a match at 40–60. The maximum number of binary searches can be estimated by solving this formula for N: $2^{**}N = \text{number of values in format}$. For twenty values, the formula would be $2^{**}N = 20$, which in log form is $N = \log(20) / \log(2)$, or more than 4.

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 this `Retain` statement to set the other fields needed by the `Proc Format`.

```
Retain fmtname 'grpfmt'
      type 'n'
      sexcl 'y'
      eexcl 'n'
      ;
```

After re-executing the code, she uses the `Cntlin` option of `Proc Format` to read the data set into a user format.

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

`Proc Format` reads the data set `Grouping` and loads it into a user format. The variables in the data set specify the necessary elements of the format she is building. Notice that our analyst excluded the start of the range, but included the end of the range (see `Retain` statement above). This caused the ranges to be the equivalent of `GT Start` and `LE End`.

Variable	Format Function
<code>fmtname</code>	name of format
<code>type</code>	type of format
<code>start</code>	beginning of range
<code>end</code>	end of range
<code>sexcl</code>	exclude start of range
<code>eexcl</code>	exclude end of range
<code>Label</code>	values to be assigned

After loading the format, she copies her program and deletes her 'If Then Else' statements. She then adds one `Put` statement in their place.

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

After executing the modified program, she compares execution times. The new program runs only slightly faster than the old program.

At first, she is disappointed, but then she realizes that it makes sense. The old program would execute an average of 10 'If' statements for each record if the data were evenly distributed. This is because each record would be as likely to match an If statement in the top half of the statements as the bottom. The user format, on the other hand, would require a maximum of 5 searches to find a match.

However, the data is distributed toward the top of the 'If' statements. This means that most records will execute less than 10 'If Then Else' statements. This condition makes the two approaches close to being equally efficient.

Had there been more 'If Then Else' statements, however, the user format would be faster regardless of the distribution. Plus, had the data been uniformly distributed, the format would have been more efficient than 'If Then Else' statements for as few as 8 statements. Eight format lines would require three searches, while 8 'If Then Else'

Statements would require an average of 4 searches.

Not discouraged, the analyst explores further applications of user formats. The customer classification code seems to be a good choice.

Each customer is assigned a classification code 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. For example, one code could mean Travel Packages, By Magazine, and Young Family. Another could mean Dining Out, By Phone, and Single Female.

The analyst patiently builds a spreadsheet with the 1000 rating codes and their meanings. These are two of the entries

Rating	Offer	Media	Customer
012	Travel Packages	Magazine	Young Family
121	Dining Out	Phone	Single Female

After creating a CSV file and importing the data into a SAS data set, she recalls the code that created the data set and modifies it to create a new variable to contain all of the values being assigned. She also creates a Retain statement to set the values needed for the Cntlin option of Proc Format and a Rename statement to establish the correct variable names.

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

Since ranges aren't involved, only the absolutely essential variables are needed for the user format. Those are:

- Fmtname
- Type
- Label
- Start

Later, when the analyst uses the format to assign Alltext to a classification, she can decompose the text into its three variables with this code:

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

This approach works very well compared to an equal number of 'If Then Else' statements. The 1000 unique values in the format require a maximum of ten searches to find a match, while 1000 'If Then Else' statements would require an average of 500 searches. Her program flies, it is so fast!

But if the number of classifications had been much smaller, say 50, the format approach would have still much faster: 6 searches versus 25.

She shares this information with her co-workers and they follow her approach: implementing user-formats instead of 'If Then Else' statements for assigning values to their data. The processing efficiency of their programs improves significantly.

Assigning Attributes, Part Two

Yet, no one initially considers whether using an indexed data set would be even faster for assigning values – and more straightforward – than employing a user-built format. Eventually, though, the thought occurs to the new-hire who implemented indexing for data selection. 'Why wouldn't an index work well for assignments, also,' he thinks?

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 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 then modifies the import code so that it creates the SAS data set Text from the CSV file.

Rating	Offer	Media	Customer
012	Travel Packages	Magazine	Young Family
121	Dining Out	Phone	Single Female

Thereafter, he uses Proc Datasets to index the data set by Rating.

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

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

```
Proc SQL;  
  Create Table MyRatings as  
  Select * /* keep all variables */  
  From Sales B  
    Inner join  
    Text A  
  On A.Rating = B.Rating  
Quit;
```

Using an index instead of a user-built format is often faster because the index search doesn't return to the beginning of the index list unless the Unique option is specified.

Format searches, on the other hand, always return to the beginning of the list. Thus, if the value of the previous record required 13 searches to find a match, the next record would require the same 13 searches even if the value sought were identical. Using an index search under the same conditions would require zero (0) searches.

Indexes look for the location of a record using Binary Tree search. A Binary Tree is a group of linked lists, while a format is but one list. To see a description of a Binary Tree, review the example in the endnotes.²

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 used to take before indexes were used. What happened?

The answer lies in the problems and opportunities created by indexing. The indexed is sorted as it is created, but the data values being retrieved do not have to be sorted. Before, each zip code retrieval gathered chunks of records from the same area on the disk with one I/O operation. Now, retrieving all of the customers on a zip code requires that many I/O operations be executed.³

Increased I/O not only slows down each program, it slows down all programs because each has to wait longer to have access to the data it needs. Before, when program A was retrieving zip code 76108, it did not have to wait on program B as it was retrieving zip code 76126. Now it does, because some of 76108 customers reside in the same disk locations as the 76126 customers.

The analysts discover the solution, however. They find out that the sort has been dropped, so they sort the Sales data set and rebuild the index to see if that fixes the problem. It does, so they add the sort and the indexing to the monthly load of the data. All is again well in Cowtown.

Guidelines for Assigning Attributes and Selecting Records.

- Assigning attributes and selecting records both involve examining values to see if they match certain criteria. This matching can be done by If statements, formats, or indexes.
- Assigning attributes involves adding content to data, such as providing a description for a class code.
- Selecting records involves retrieving only the records that match certain criteria, such as being within a particular zip code.
- Not counting the overhead of creating and storing either, formats and indexes are more efficient than If statements if the number of If statements exceed more than a few: eight for uniformly distributed data.
- A format is just as efficient as an index for assigning attributes to ungrouped data if the number of lines in the format are less than a thousand, plus formats require less overhead to create.
- Formats are better suited for assigning attributes from a range of values than are indexes.
- Indexes should be used for assigning attributes to sorted data or for selecting records.
- Indexes can be better suited than formats for assigning multiple attributes.
- Indexes are to be used for retrieving subsets of data (no more than 30%), not for all of the data.
- Large subsets of data (20% to 30%) should be retrieved via an index only if the data is also sorted by that index. Retrieving all of the customers in a particular marketing area or group is an example.
- If only small amounts of data are being retrieved at one time by the index, the data need not be sorted by the index. An example would be retrieving data by individual customers.
- Verify these guidelines and adapt them to the unique qualities of your data processing environment.

References

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

SAS Institute Inc. SAS Language Reference Concepts, Version 8,
(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)

Author Information

Jimmy DeFoor
Citi Card, Irving, Texas
Email: jimmy_defoor@afcc.com

SAS is a registered trademark or trademark of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

Endnotes

¹ If NotSorted had been specified, the format would have been stored in the order it was read and would have been sequentially searched instead of binary searched. Had it been loaded in the same order as the 'If Then Else' statements, it would have executed with the same efficiency.

² Below is an example of how the index for the thousand values in the classification would be organized for a Binary Tree search. Notice that the root, branch, and leaf nodes are organized sequentially. This is necessary for all binary searching. Each root node is each linked to a separate branch node. Each branch node is linked to a separate leaf node. The leaf nodes point to the actual 'line' locations of the data, which are the italicized numbers. The data itself does not have to be sorted, which it isn't in this case.

This example shows the linkage from root node 96.

Root Node

0	48	96	143	191	239	286	334	382	429
477	525	572	620	668	715	763	811	858	954
999									

Branch Node

96	99	101	104	106	109	112	114	117	119
120	122	125	127	129	131	133	135	137	139
142									

Leaf Nodes

96	26	97	33	98	<i>104</i>
101	45	102	<i>12</i>	103	<i>88</i>
104	5	105	<i>21</i>		

Executing a binary search of a linked list can be faster than executing a binary search of a single list because more opportunities exist for finding the matching value more quickly. But those opportunities become more significant as the number of indexed values increases. Using an index for our 1000 values would likely not increase search speed. Here is an example:

A binary search for the number 105 in a format list would be executed as follows: 499, 250, 125, 63, 95, 110, 103, 106, 104, and 105. Ten searches would be required. Searching the Binary Tree would start with the root node, also beginning in the middle. The search would go to 477, then 191, then 96, then 143. It would then return to 96 since it was less than the target of 105. Then it would go to the 96-branch node and search in the same way: 120, 109, 101, and 104. Then, it would go to the beginning of the 104-leaf node and search sequentially: 104, 105. That's 4 + 4 + 2 searches or 10, same as the searching the format list.

As said in the text, the primary gain from using an index comes from *not using any search when the same value is being sought* on the current record as on the last. Also, however, indexes are the only way to

retrieve multiple records for a value, such as retrieving all the persons who live in a particular zip code. Multiple retrieves are not possible with formats because duplicates are not allowed.

3

When the underlying data is sorted by zip code, the leaf node of the index for zip code could look like this:

76108	101	102	103	76116	201	202	203	76126	301	302	303
-------	-----	-----	-----	-------	-----	-----	-----	-------	-----	-----	-----

If the data was not sorted by zip code, the leaf node could look like this:

76108	122	207	405	76116	23	343	515	76126	209	45	810
-------	-----	-----	-----	-------	----	-----	-----	-------	-----	----	-----

Clearly, for the unsorted leaf node, the data retrieval for the customers in any zip code would require more I/O operations than for the sorted leaf node. Plus, if one analyst wanted customers for 76108 at the same time as another analyst wanted customer data for 76126, they would contend for the disk area containing lines 207 and 209.