ABSTRACT

This tutorial illustrates the importance of understanding the internals of DATA step processing, particularly when you invoke functions such as LAG, code RETAIN statements, rely on WHERE constructs, and start combining various features of the DATA step language and performing conditional execution of certain functions. This anthology includes such spellbinding stories as:

- “LAG With a WHERE”
- “To LAG or to LEAD”
- “A DIFFerent LAG”
- “When RETAIN Doesn't Retain”
- “Don't Order My Variables Around”, and
- “The Case of the Missing Values”.

INTRODUCTION

The idea for this paper was conceived by a SUGI Section Chair who commissioned the author to write a paper addressing recurring issues from SAS-L. SAS-L is a peer email list disseminated by several list servers. It permits SAS users to participate in a discussion group (see REFERENCES below for information on subscribing to SAS-L). The list is an excellent source of advice, information, code, techniques, comments, opinions, and camaraderie, serving gurus and newbies alike. The chapters in this tutorial capture a few curious topics from the list.

CHAPTER 1: “LAG With a WHERE”

“LAG is a snag”. Ian Whitlock

A recurring theme: the LAG function does not work as advertised and its dysfunction is exacerbated during conditional execution. But when you investigate the way the LAG function works internally and simulate the storage queues for LAGged values, the behavior is more understandable. The syntax for the LAG function is as follows:

```
* Syntax:
LAG<n>(argument)

* n specifies number of lagged values
* argument is numeric or character
```

Important features of the LAG function to note:

- LAG functions return values from a queue
- A LAGn function stores a value in a queue and returns a value stored previously in that queue
- Each occurrence of a LAGn function generates its own queue
• \( n \) is the length of the queue
• the LAG function is executable
• storing values at the bottom of the queue and returning values from the top of the queue occurs **only when the function is executed**
• a LAG\( n \) function that is executed conditionally will store and return only values from the observations for which the condition is satisfied.

Consider the following basic DATA step code for a simple invocation of LAG:

```
data new;
  input x @@;
  lag1=lag1(x);
  lag2=lag2(x);
cards;
  1 2 3 4 5 6
;
```

lag1 and lag2 capture the two previous values of \( x \), and PROC PRINT of the resulting data set reveals the LAG queues. Note that the queues are initially populated with missing values for LAGged values that are not available.

<table>
<thead>
<tr>
<th>( x )</th>
<th>LAG1</th>
<th>LAG2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>.</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>4</td>
</tr>
</tbody>
</table>

When the LAG function is conditionally executed, as noted, values are stored and returned from the queue only when the LAG function is explicitly executed. The intent of the following code is to retrieve the LAGged value of \( a \) only when \( b=2 \).

```
data new;
  input a b @@;
  LAGa = LAG(a);
  if b=2 then LAGb=LAG(a);
cards;
  1 1 2 1 3 2 4 1 5 2 6 1
;
```

The statement:

\[
\text{LAGa} = \text{LAG(a)};
\]

is executed on each iteration of the DATA step. However, LAGb is set to LAG(a) only when \( b=2 \) and the assignment statement is executed.
The next example will attempt to capture “every other” LAGged value of x (the “even” values of x, where x is 2, 4, 6, or 8).

```
data new;
input x @@;
if mod(x,2)=0 then condLAG1 = lag(x);
LAGx=lag(x);  *<<<unconditional;
if mod(x,2)=0 then condLAG2 = LAGx;
cards;
1 2 3 4 5 6 7 8;
```

In order to store, and subsequently retrieve, the correct values in the queue, you must unconditionally capture the LAGged value and conditionally set it to equal to condLAG2.

<table>
<thead>
<tr>
<th>X</th>
<th>LAGx</th>
<th>condLAG1</th>
<th>condLAG2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>.</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>8</td>
<td>7</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>

Conditional execution can also be accomplished using the WHERE statement. The unique properties of WHERE add another dimension to DATA step processing:

- Selects observations before they are brought into the logical Program Data Vector (PDV)
- After data set options are applied
- Before any other data step statements executed, including SET, BY, etc.
- Functions differently with BY and first. and last.
- Only works with SAS data sets (not raw, non-SAS data)

The following DATA step code subsets observations, keeping only those where visit (date) is beyond January 1, 2003. However, we want to capture the LAGged value of weight, even if the previous observation was not in the date range:
Compare the difference between the example using the WHERE statement and the example using a subsetting IF:

Output from WHERE

<table>
<thead>
<tr>
<th>VISIT</th>
<th>WEIGHT</th>
<th>LAGWGT</th>
</tr>
</thead>
<tbody>
<tr>
<td>02JAN2003</td>
<td>22</td>
<td>.</td>
</tr>
<tr>
<td>03JAN2003</td>
<td>154</td>
<td>22</td>
</tr>
<tr>
<td>04JAN2003</td>
<td>21</td>
<td>154</td>
</tr>
<tr>
<td>05JAN2003</td>
<td>112</td>
<td>21</td>
</tr>
</tbody>
</table>

Output from subsetting IF

<table>
<thead>
<tr>
<th>VISIT</th>
<th>WEIGHT</th>
<th>LAGWGT</th>
</tr>
</thead>
<tbody>
<tr>
<td>02JAN2003</td>
<td>22</td>
<td>88</td>
</tr>
<tr>
<td>03JAN2003</td>
<td>154</td>
<td>22</td>
</tr>
<tr>
<td>04JAN2003</td>
<td>21</td>
<td>154</td>
</tr>
<tr>
<td>05JAN2003</td>
<td>112</td>
<td>21</td>
</tr>
</tbody>
</table>

Recall the features of the WHERE statement, and note that, in the first example, since WHERE is executed before any observations are read, there are no LAGged values for the first observation selected. In the second example, all observations are read, the subsetting IF is executed against each input observation, and the LAGged value of WEIGHT will be captured from the out-of-range visit.
CHAPTER 2: “To LAG or to LEAD”

Many users want to “look ahead” for variable values in the same way that the LAG function “looks back”. However, there is no “LEAD” function or “negative LAG” capability in SAS. The gurus of SAS-L provide several solutions.

One solution is to sort the data set in descending order (reverse) and then apply the LAG function. Another more elegant solution is to MERGE the SAS data set with itself, using a one-on-one MERGE, with no BY statement. The second data set on the MERGE statement is the same data set, controlled with a firstobs=2. Each iteration of DATA step execution looks at the current observation and the next observation simultaneously. Note, be sure to rename variables in common between the two data sets in order to compare the values.

```
data lagged ;
merge master ( keep = var ) master ( firstobs = 2, rename = (var =nextvar ) ) ;
**** no BY statement ;
run;
```

```
var nextvar
1 2
2 3
3 4
4 5
5 6
6 
```

CHAPTER 3: “A DIFferent LAG”

The ability to manage LAGged values is further enhanced with the DIF function. Its use is illustrated below. It should be noted that the DIF function stores and returns values from queues in the same manner as the LAG, and the same caveats with respect to conditional execution apply.

```
* DIF functions return the first difference between the argument and its nth lag.
* Defined as:

DIF(X) = X - LAGn(X) ;
```
"A DIFferent LAG"

- **DIF** function
- **Syntax:**

\[
\text{DIF}<n>(\text{argument})
\]

- \(n\) specifies number of lags
- \(\text{argument}\) is numeric

```
data new;
   input x @@;
   lagx = lag(x);
   difx = dif(x);
cards;
1 2 8 4 3 9 7
;```

PROC PRINT of data set new created above:

<table>
<thead>
<tr>
<th>x</th>
<th>lagx</th>
<th>difx</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>-4</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>-1</td>
</tr>
<tr>
<td>9</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>9</td>
<td>-2</td>
</tr>
</tbody>
</table>

**CHAPTER 4: “When RETAIN Doesn’t Retain”**

A RETAIN statement causes a variable whose value is assigned by an INPUT or assignment statement to retain its value from the current iteration of the DATA to the next iteration of the DATA step. It is not an executable statement; it provides information to the logical Program Data Vector (PDV), indicating which variables should not be initialized to missing at the beginning of each execution iteration. It should be noted that certain statements have an “implied” RETAIN. A variable whose value is determined in a SUM statement is also automatically retained, as are SAS automatic variables, like _N_. 
Retained Variables

- all SAS special variables, e.g.: _N_, first. and last.
- all vars in RETAIN statement
- all vars from SET or MERGE
- accumulator vars in SUM stmt

Any variables read by a SET, MERGE or UPDATE statement are automatically retained. These variables are reinitialized, however, when the data set changes or the value of the BY group variable changes.

Variables Not Retained

- Variables from INPUT statement
- User-defined variables/ vars created in DATA step

The following code concatenates data sets A and B, and, if site is missing (meaning, ostensibly, the observation being read on that iteration of the DATA step loop is coming from data set B created by the code in the second program), then site is determined by substrin ging the id variable:

```
data A;
  input id $ site $;
  cards:
    10212 00
    10213 00
  ;
data B;
  input id $ cards;
  cards;
    02001
    03005
    06000
  ;
data c;
  set A B;
  if missing(site) then site = substr(id,1,2);
run;
```

When looking at the output, it is at first difficult to understand why site has a value of 02 for all observations coming from data set B. But variables being read from a data set on a SET, MERGE or UPDATE statement are not initialized to missing at the beginning of each DATA step iteration. Consequently, site will never test as missing, and the SUBSTR will not be performed.
Why didn’t it work?

<table>
<thead>
<tr>
<th>id</th>
<th>site</th>
</tr>
</thead>
<tbody>
<tr>
<td>10212</td>
<td>00</td>
</tr>
<tr>
<td>10213</td>
<td>00</td>
</tr>
<tr>
<td>02001</td>
<td>02</td>
</tr>
<tr>
<td>03005</td>
<td>02</td>
</tr>
<tr>
<td>06900</td>
<td>02</td>
</tr>
</tbody>
</table>

If you change the condition under which the SUBSTR is performed, you will get the appropriate results. The IN= variable on data set B will test TRUE for any observation being read from data set B, so the assignment statement will be executed.

```
data C;
  set A B (in=inb);
  if inb then site = substr(id,1,2);
run;
```

test that the observation has come from B and only then extract the site value....

<table>
<thead>
<tr>
<th>id</th>
<th>site</th>
</tr>
</thead>
<tbody>
<tr>
<td>10212</td>
<td>00</td>
</tr>
<tr>
<td>10213</td>
<td>00</td>
</tr>
<tr>
<td>02001</td>
<td>02</td>
</tr>
<tr>
<td>03005</td>
<td>02</td>
</tr>
</tbody>
</table>
| 06900 | 06 !

CHAPTER 5: “Don’t Order My Variables Around”

“The variable order is not always declared where it seems to occur...” Ron Fehd

During DATA step compilation, variables are added to the logical Program Data Vector (PDV) in the order they are seen by the compiler. This becomes the order in which the variables are stored for output. PROC CONTENTS with the position option will show the logical PDV order. (There is a logical order (determined by the SAS compiler for use in DATA step processing) and a physical structure, determined for I/O optimization.)

The logical PDV order becomes the default order of the variables as they are used/displayed with PROCs. It is also the order in which variables will be captured using the double dash ( - - ) notation for referring shorthand to variable lists with a SAS data set.

Consider a simple DATA step creating a data set called “new” containing five variables:
data new;
input NAME $ SEX $ AGE ID RX_GRP;
cards;
John M 35 101 2
David M 53 206 1
Howard M 45 321 3
;
Proc contents position; run;

PROC CONTENTS with the position option yields the variable list in position order, in addition to the default alphabetic listing.

---Alphabetic List of Variables and Attributes---

<table>
<thead>
<tr>
<th>#</th>
<th>Variable</th>
<th>Type</th>
<th>Len</th>
<th>Pos</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>AGE</td>
<td>Num</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>ID</td>
<td>Num</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>1</td>
<td>NAME</td>
<td>Char</td>
<td>8</td>
<td>24</td>
</tr>
<tr>
<td>5</td>
<td>RX_GRP</td>
<td>Num</td>
<td>8</td>
<td>16</td>
</tr>
<tr>
<td>2</td>
<td>SEX</td>
<td>Char</td>
<td>8</td>
<td>32</td>
</tr>
</tbody>
</table>

---Variables Ordered by Position---

<table>
<thead>
<tr>
<th>#</th>
<th>Variable</th>
<th>Type</th>
<th>Len</th>
<th>Pos</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>NAME</td>
<td>Char</td>
<td>8</td>
<td>24</td>
</tr>
<tr>
<td>2</td>
<td>SEX</td>
<td>Char</td>
<td>8</td>
<td>32</td>
</tr>
<tr>
<td>3</td>
<td>AGE</td>
<td>Num</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>ID</td>
<td>Num</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>5</td>
<td>RX_GRP</td>
<td>Num</td>
<td>8</td>
<td>16</td>
</tr>
</tbody>
</table>

Users often want to force the order of variables in the logical PDV and subsequently in the output data set. The primary reason seems to have to do with a need for a reasonable, sensible or logical order to the data. Additionally, there is desire to manipulate the default order used by the PROCs. The most apparent reason for reordering the variables is related to the presentation layer.

**WHY?**
- exporting / export wizard
- SAS Viewer end users
- manipulate groups/lists of vars (age - - diag)
  using double-dash notation
- with PUT or ARRAY
- Presentation layer
- Default for PROC usage
To reorder the variables, a RETAIN statement should be coded before the SET or MERGE statement to establish the desired order of the variables in the logical PDV (and output data set). Use of statements like ATTRIB or LENGTH could accomplish the same thing, but the user risks inadvertently introducing errors to the attributes of the variables referenced.

A PROC CONTENTS of data set MASTER:

```
<table>
<thead>
<tr>
<th>#</th>
<th>Variable</th>
<th>Type</th>
<th>Len</th>
<th>Pos</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>NAME</td>
<td>Char</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>SEX</td>
<td>Char</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>AGE</td>
<td>Num</td>
<td>8</td>
<td>16</td>
</tr>
<tr>
<td>4</td>
<td>ID</td>
<td>Num</td>
<td>8</td>
<td>24</td>
</tr>
<tr>
<td>5</td>
<td>RX_GRP</td>
<td>Num</td>
<td>8</td>
<td>32</td>
</tr>
</tbody>
</table>
```

A PROC PRINT of data set MASTER:

```
NAME   SEX AGE ID RX_GRP
John   M   35 101  2
David  M   53 206  1
Howard M   45 321  3
```

To change the order, code a RETAIN statement, making sure the variables you want are the first reference to the compiler and are in the desired order. **With RETAIN the only reason for leaving out the value is to reorder so one has to do it with intent and not as a side affect.**  

Ian Whitlock

```
data new;
  retain id rx_grp name sex age;
  *** 1st reference to compiler;
  set master;
run;
```

A PROC CONTENTS of data set “new”, reordered:
CHAPTER 6: “The Case of the Missing Values”

Consider the following mystery posed to SAS-L:

“How do MISSINGs compare?”

**QUESTION:**

If TOT > SUM then <perform action> ;

If either A or B is missing, isn’t the statement just ignored?

Given the default numeric missing value, if TOT is missing and SUM is 10, then the expression is false and the action is not performed. If TOT is 20 and SUM is missing, then the expression resolves as true, and the action is performed. If both TOT and SUM are missing, the expression is false, and the action is not performed.

Depending on why your numeric data is missing, what data is missing, and the occurrence of any patterns, it is possible to distinguish between missing values. This is commonly used with survey data, where responses to questions might be “not applicable”, missing, “refused to answer”, “unknown”, etc. All the responses are missing, but represent a different kind of missing value. SAS provides:
Sample uses of these missing values:

```
if age = 99 then age = .A ;
else if age = 88 then age = .B ;
else if age = . then age = ._ ;

**** where...
.A represents “unknown”
.B represents “refused to answer”
_. represents “no data”  **************;
```

It should be noted that the 27 different missing values have different “values” and can be compared against each other, ._ having the lowest value and .Z the highest. They all test less than any negative number.

Consider the following SAS data set (OLD) and note the missing values for the NUMERIC variables TOT and SUM in the last two observations.

```
data set OLD
  NAME    TOT  SUM
  John    10   7
  David   A    _
  Howard  .    .
run;
```

.A compares greater than ._ and the condition resolves to true for the first two observations in OLD. The last observation will not be included in data set SUBSET (the default missings does not compare greater than one another).
Character missings are handled as follows:

Character missing values will test equal if they have the same value, even if the declared LENGTHs of the variables are different. In the following example, for the second and third observations, the values of KEY and ID are equal, even though KEY has a length of 4 and ID a length of 1.

Moral of this story: know your data.

CONCLUSION

deus ex machina n : unexpected, artificial, or improbable character, device, or event introduced suddenly in a work of fiction or drama to resolve a situation or untangle a plot.

In Greek and Roman drama, a god was lowered by stage machinery into the scene to resolve a plot or extricate the protagonist from a difficult situation.

So goes SAS-L. Take advantage of all the help facilities available to SAS users: documentation, SAS technical support, and on-line HELP. And there is no substitution for good trial and error, and robust testing of SAS code. But the SAS-L gurus have a wealth of SAS experience and remain a great sounding board for
issues, technical techniques, and problem solving of all sorts. Join the discussion with the giving and receiving of SAS wisdom.

REFERENCES

SAS-L is a peered email list disseminated by several listservers. To have the messages mailed to you as they are available, send e-mail to any one of the following mail servers. Be prepared for a lot of email!

- listserv@vm.marist.edu Marist University
- listserv@listserv.vt.edu Virginia Polytechnic University
- listserv@listserv.uga.edu University of Georgia
- listserv@AKH-WIEN.AC.AT University of Vienna

The subject line of the email is ignored and the body should contain the command:

```
subscribe SAS-L your name
```

e.g., subscribe SAS-L Tom Smith is how Tom Smith would subscribe.

Other SAS-L information can be found at the following:

- on-line SUGI proceedings at http://support.sas.com/usergroups/sugi/proceedings/index.html
- http://www.listserv.uga.edu/archives/sas-l.html
- Comp.soft-sys.sas - mirror site to SAS-L
- www.sas-l.com
- SAS-L BOF at SUGI in Montreal, May 2004
  http://support.sas.com/usergroups/sugi/sugi29/index.html
- From www.sconstig.com:
  - SAS-L On-Line
  - how to subscribe to SAS-L

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Neil Howard
i3 Data Services
131 Morristown Rd
Basking Ridge, NJ 07920
Work Phone: 973-348-1137
Email: neil.howard@i3data.com
www.i3data.com

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