Paper 1277-2017

Leads and Lags: Static and Dynamic Queues in the SAS® DATA STEP, 2nd ed.

Mark Keintz, Wharton Research Data Services

ABSTRACT

From stock price histories to hospital stay records, analysis of time series data often requires use of lagged (and occasionally lead) values of one or more analysis variable. For the SAS® user, the central operational task is typically getting lagged (lead) values for each time point in the data set. While SAS has long provided a LAG function, it has no analogous "lead" function – an especially significant problem in the case of large data series. This paper will (1) review the lag function, in particular the powerful, but non-intuitive implications of its queue-oriented basis, (2) demonstrate efficient ways to generate leads with the same flexibility as the lag function, but without the common and expensive recourse to data re-sorting, and (3) show how to dynamically generate leads and lags through use of the hash object.

INTRODUCTION

Have you ever needed to report data from a series of monthly records comparing a given month's results to the same month in the prior year? Or perhaps you have a set of hospital admission and discharge records sorted by patient id within date and been asked to find length of stay for each patient visit (or even time to next admission for selected DRGs). These are just two examples of the frequent need SAS programmers have to find lag or lead values. All too often programmers find themselves in a multi-step process of sorting, joining, and resorting data just to retrieve data items from some prior or subsequent record in a data set. But it is possible to address all these tasks in single data step programs, as will be shown below.

SAS has a LAG function (and a related DIF function) intended to provide data values (or differences) from preceding records in a data set. This presentation will show advantages of the "queue-management" character of the LAG function in addressing both regular and irregular/interleaved times series, whether grouped or ungrouped. Since there is no analogous "lead" function, later sections show how to use multiple r SET or MERGE statements to find leads. In prior presentations on this topic I emphasized the use of SET statements for leads. As you'll see in this edition, I have come to prefer MERGE as simpler technique.

The final section will address cases in which you don't know in advance how many lag series or lead series will be needed – they are determined by the data. This problem of "dynamic" lags and leads can't be effectively handled through clever use of the lag function or multiple SET/MERGE statements. Instead straightforward application of hash objects provides the best solution.

THE LAG FUNCTION – RETRIEVING HISTORY

The term "lag function" suggests retrieval of data via "looking back" by some user-specified number of periods or observations, and that is what most simple applications of the LAG function seem to do. For instance, consider the task of producing 1-month and 3-month price "returns" for this monthly stock price file (created from the **sashelp.stocks** data – see appendix for creation of data set SAMPLE1):

Table 1
Five Rows From SAMPLE1
(re-sorted from sashelp.stocks)

Obs	DATE	STOCK	CLS
1	01AUG86	IBM	$138.75
2	02SEP86	IBM	$134.50
3	01OCT86	IBM	$123.62
4	03NOV86	IBM	$127.12
5	01DEC86	IBM	$120.00

The program below uses the LAG and LAG3 (3 record lag) functions to compare the current closing price (CLS) to its immediate and it "third prior" predecessors, needed to calculate one-month and three-month returns (RTN1 and RTN3):

Example 1: Simple Creation of Lagged Values

```
data simple_lags;
  set sample1;
  CLS1=lag(CLS);
  CLS3=lag3(CLS);
  if CLS1 ^=. then RETN1 = CLS/CLS1 - 1;
  if CLS3 ^=. then RETN3 = CLS/CLS3 - 1;
run;
```

Example 1 yields the following data in the first 5 rows:

Table 2
Example 1 Results from using LAG and LAG3

Obs	DATE	STOCK	CLS	CLS1	CLS3	RETN1	RETN3
1	01AUG86	IBM	$138.75	.	.	.	.
2	02SEP86	IBM	$134.50	138.75	.	-0.031	.
3	01OCT86	IBM	$123.62	134.50	.	-0.081	.
4	03NOV86	IBM	$127.12	123.62	138.75	0.028	-0.084
5	01DEC86	IBM	$120.00	127.12	134.50	-0.056	-0.108

LAGS ARE QUEUES – NOT "LOOK BACKS"

At this point LAG functions have all the appearance of simply looking back by one (or 3) observations, with the additional feature of imputing missing values when "looking back" beyond the beginning of the data set. But actually the lag function **instructs SAS to construct a FIFO (first-in/first-out) queue** with (1) as many entries as the specified length of the lag, and (2) the queue elements initialized to missing values. Every time the lag function is executed, the oldest entry is retrieved (and removed) from the queue and a new entry (i.e. the current value) is added. Why is this queue management vs lookback distinction significant? That becomes evident when the LAG function is executed conditionally, as in the treatment of BY groups below.

As an illustration, look at observations 232 through 237 generated by Example 1 program, and presented in Table 3. This shows the last two cases for IBM and the first four for Intel. For the first Intel observation (obs 234), the lagged value of the closing stock price (CLS1=82.20) is taken from the IBM series. Of course, it should be a missing value, as should all the shaded cells. Lag values are not completely correct again until the fourth Intel record (obs 237).

Table 3
The Problem of BY Groups for Lags

Obs	DATE	STOCK	CLS	CLS1	CLS3	RETN1	RETN3
232	01NOV05	IBM	$88.90	81.88	80.62	0.086	0.103
233	01DEC05	IBM	$82.20	88.90	80.22	-0.075	0.025
234	01AUG86	Intel	$23.00	*82.20*	*81.88*	*-0.720*	*-0.719*
235	02SEP86	Intel	$19.50	23.00	*88.90*	-0.152	*-0.781*
236	01OCT86	Intel	$20.25	19.50	*82.20*	0.038	*-0.754*
237	03NOV86	Intel	$23.00	20.25	23.00	0.136	0.000

The "natural" way to address this problem is to use a BY statement in SAS and (for the case of the single month return) avoid lag execution when the observation in hand is the first for a given stock. Example 2 is such a program (dealing with CLS1 only for illustration purposes), and its results are in Table 4.

Example 2: A "naïve" implementation of lags for BY groups

```
data naive_lags;
  set sample1;
  by stock;
  if not(first.stock) then CLS1=lag(CLS);
  else CLS1=.;
  if CLS1 ^=. then RETN1= CLS/CLS1 - 1;
  format ret: 6.3;
run;
```

Table 4
Results of
"if not(first.stock) then CLS1=lag(CLS)"

Obs	STOCK	DATE	CLS	CLS1	RETN1
232	IBM	01NOV05	$88.90	81.88	0.086
233	IBM	01DEC05	$82.20	88.90	-0.075
234	Intel	01AUG86	$23.00	.	.
235	Intel	02SEP86	$19.50	*82.20*	*-0.763*
236	Intel	01OCT86	$20.25	19.50	0.038
237	Intel	03NOV86	$23.00	20.25	0.136

This fixes the first Intel record, setting both CLS1 and RETN1 to missing values. But look at the second Intel record (Obs 235). CLS1 has a value of 82.20, taken not from the beginning of the Intel series, but rather from the end of the IBM series, generating an erroneous value for RETN1 as well. In other words, **CLS1 did not come from the prior record (lookback), but rather it came from the queue, whose contents were most recently updated in the last IBM record**.

LAGS FOR BY GROUPS (IFN AND IFC ALWAYS CALCULATE BOTH OUTCOMES)

The fix is easy: unconditionally execute a lag, and then reset the result when necessary. This is usually done in two statements, but Example 3 shows a more compact solution (described by Howard Schrier – see References). It uses the IFN function instead of an IF statement - because IFN executes the lag function embedded in its second argument regardless of the status of first.stock (the condition being tested). Of course, IFN returns the lagged value only when the tested condition is true. Accommodating BY groups for lags longer than one period simply requires comparing lagged values of the BY-variable to the current values ("lag3(stock)=stock"), as in the "CLS3=" statement below.

Example 3: A robust implementation of lags for BY groups

```
data bygroup_lags;
  set sample1;
  by stock;
  CLS1 = ifn(not(first.stock),lag(CLS),.);
  CLS3 = ifn(lag3(stock)=stock,lag3(CLS),.) ;
  if CLS1 ^=. then RETN1 = CLS/CLS1 - 1;
  if CLS3 ^=. then RETN3 = CLS/CLS3 - 1;
  format ret: 6.3;
run;
```

The data set BYGROUP_LAGS now has missing values for the appropriate records at the start of the Intel monthly records.

Table 5
Result of Robust Lag Implementation for BY Groups

Obs	STOCK	DATE	CLS	CLS1	CLS3	RETN1	RETN3
232	IBM	01NOV05	$88.90	81.88	80.62	0.086	0.103
233	IBM	01DEC05	$82.20	88.90	80.22	-0.075	0.025
234	Intel	01AUG86	$23.00	.	.	.	.
235	Intel	02SEP86	$19.50	23.00	.	-0.152	.
236	Intel	01OCT86	$20.25	19.50	.	0.038	.
237	Intel	03NOV86	$23.00	20.25	23.00	0.136	0.000

MULTIPLE LAGS MEANS MULTIPLE QUEUES – A WAY TO MANAGE IRREGULAR INTERLEAVED SERIES

While BY-groups benefit from *avoiding* conditional execution of lag functions in the solution above, sometimes conditional execution is the solution. In the data set SALES below (see Appendix for its generation) are sales sorted by month and product, so within-product series are interleaved with other products. Also a given product×month combination is only present when sales are reported. As a result each month has a varying number of records, depending on which product had reported sales. Table 6 shows 5 records for month 1, but only 4 records for month 2. Unlike the Sample 1 data, this time series is not regular, so comparing (say) sales of product B in January (observation 2) to February (obs 7) would imply a LAG5, while comparing March (obs 10) to February would need a LAG3.

Table 6
Data Set SALES (first 13 obs)

OBS	MONTH	PROD	SALES
1	1	A	4
2	1	B	20
3	1	C	11
4	1	D	7
5	1	X	20
6	2	A	16
7	2	B	12
8	2	D	2
9	2	X	8
10	3	B	14
11	3	C	4
12	3	D	7
13	3	X	6

The solution to this task is to use conditional lags, with one queue for each product. The program is surprisingly simple:

Example 4: LAGS for Irregular Interleaved Time Series[1]

```
data irregular_lags;
  set sales;
  select (product);
    when ('A') change_rate=(sales-lag(sales))/(month-lag(month));
    when ('B') change_rate=(sales-lag(sales))/(month-lag(month));
    when ('C') change_rate=(sales-lag(sales))/(month-lag(month));
    when ('D') change_rate=(sales-lag(sales))/(month-lag(month));
    otherwise;
  end;
run;
```

Example 4 generates four pairs of lag queues, one pair for each of the products A through D. Each invocation of a lag maintains a separate queue, resulting in eight distinct queues in Example 4. Because a given queue is updated only when the corresponding product is in hand (the "when" clauses), the output of the lag function must come from the most recent observation having the same product, no matter how far back it may be in the data set. Note that, for most functions, clean program design would collapse the four "when" conditions into one, such as

```
when('A','B','C','D') change_rate= function(sales)/function(month);
```

Succumbing to this temptation confounds the series of all the products, giving erroneous results.

LEADS – USING EXTRA SET (OR MERGE) STATEMENTS TO LOOK AHEAD

SAS does not offer a lead function. As a result many SAS programmers sort a data set in descending order and then apply lag functions to create lead values. Often the data are sorted a second time, back to original order, before any analysis is done. For large data sets, this is a costly three-step process.

There is a much simpler way, through the use a MERGE statement (or extra SET statements) in combination with the FIRSTOBS parameter. The following simple program generates both a one-month and three-month lead of the data from Sample1, by merging a dataset with itself.

Example 5: Simple generation of one-record and three-record leads

```
data simple_leads;
  merge sample1
        sample1 (firstobs=2 keep=CLS rename=(CLS=LEAD1))
        sample1 (firstobs=4 keep=CLS rename=(CLS=LEAD3));
run;
```

There are three mechanisms at work here:

[1] A more compact expression would have used DIF instead of LAG, as in "change_rate=dif(sales)/dif(month).

1. Self-MERGE: Usually a merge statement is used to get data from different data sets, but in this case, there are three parallel streams of data from one source. SAS treats them as if three unique data sets are being simultaneously read. In fact the log from Example 5 displays these notes reporting three incoming streams of data:

```
NOTE: There were 699 observations read from the data set WORK.SAMPLE1.
NOTE: There were 698 observations read from the data set WORK.SAMPLE1.
NOTE: There were 696 observations read from the data set WORK.SAMPLE1.
```

2. Data Set Name Parameter FIRSTOBS: Using the "FIRSTOBS=" parameter provides a way to "look ahead" in a data set. For instance the second reference to SAMPLE1 has "FIRSTOBS=2 (the third has "FIRSTOBS=4"), so that it starts reading from record 2 (and 4) while the first reference is reading from record 1. This provides a way to synchronize leads with any given "current" record.
3. Data Set Name Parameters RENAME, and KEEP: All three references to SAMPLE1 read in the same variable (CLS), yet a single variable can't contain more than one value at a time. In this case the original value would be overwritten by the value from the second, and subsequently by the third reference to SAMPLE1. To avoid this problem CLS is renamed in the additional SAMPLE1 entries, resulting in three variables: CLS (from the "current" record, LEAD1 (one period lead) and LEAD3 (three period lead). To avoid overwriting any other variables, only variables to be renamed should be in the KEEP= parameters.

The last 4 records in the resulting data set are below, with lead values as expected:

Table 7:
Simple Leads

Obs	STOCK	DATE	CLS	LEAD1	LEAD3
696	Microsoft	01SEP05	$25.73	$25.70	$26.15
697	Microsoft	03OCT05	$25.70	$27.68	.
698	Microsoft	01NOV05	$27.68	$26.15	.
699	Microsoft	01DEC05	$26.15	.	.

LEADS FOR BY GROUPS

Just as lags for by groups, generating leads for by groups requires a little extra code to avoid contaminating the last records of one group with leads from the early records of the next group. This time the merge statement keeps and renames not only the analysis variable (CLS) but also the group variable (STOCK). It compares the current and lead stock values, and sets corresponding LEAD1 and LEAD3 to missing when the stock values differ.

Example 6: Generating Leads for By Groups

```
data bygroup_leads;
  merge
      sample1
      sample1 (firstobs=2 keep=stock CLS rename=(stock=stock1 CLS=LEAD1))
      sample1 (firstobs=4 keep=stock CLS rename=(stock=stock3 CLS=LEAD3));
  if stock3^=stock then do;
```

```
    call missing(lead3);
    if stock1^=stock then call missing(lead1);
  end;
  drop stock1 stock3;
run;
```

The result for the last four IBM observations and the first three Intel observations are below, with LEAD1 set to missing for the final IBM, and LEAD3 for the last 3 IBM observations.

Table 8
Leads With By Groups

Obs	STOCK	DATE	CLS	LEAD1	LEAD3
230	IBM	01SEP05	$80.22	$81.88	$82.20
231	IBM	03OCT05	$81.88	$88.90	.
232	IBM	01NOV05	$88.90	$82.20	.
233	IBM	01DEC05	$82.20	.	.
234	Intel	01AUG86	$23.00	$19.50	$23.00
235	Intel	02SEP86	$19.50	$20.25	$21.00

GENERATING IRREGULAR INTERLEAVED LEAD "QUEUES" – PRECEDENCE OF WHERE OVER FIRSTOBS

Generating SALES data lags in the earlier section required multiple queues – via LAG functions conditional on product value. This resolved the problem of varying "distances" between successive records for a given product. Developing leads for such irregular time series requires the same approach – one "queue" (one data stream actually) for each product. Similar to the use of separate LAG functions to manage separate queues, generating leads for irregular interleaved times series requires a separate SET statements – each "filtered" by an appropriate WHERE condition. The relatively simple program below demonstrates:

Example 7: Generating Leads for Irregular Interleaved Time Series

```
data irregular_leads;
  set sales;

  select (product);
   when ('A') if eofa=0 then set sales (where=(product='A') firstobs=2
              keep=product sales rename=(sales=LEAD1)) end=eofa;

   when ('B') if eofb=0 then set sales (where=(product='B') firstobs=2
              keep=product sales rename=(sales=LEAD1)) end=eofb;

   when ('C') if eofc=0 then set sales (where=(product='C') firstobs=2
              keep=product sales rename=(sales=LEAD1)) end=eofc;

   when ('D') if eofd=0 then set sales (where=(product='D') firstobs=2
              keep=product sales rename=(sales=LEAD1)) end=eofd;
```

```
    otherwise call missing(lead1);
   end;
run;
```

The logic is straightforward. If the current product is 'A' [WHEN('A')] and the stream of PRODUCT "A" records is not exhausted (if eofa=0), then read the next product "A" record, renaming its SALES variable to LEAD1. The way to exclusively read product "A" records is to use the "WHERE=" data set name parameter. Most important to this technique is the fact that the ***WHERE parameter is honored prior to the FIRSTOBS parameter***. So "FIRSTOBS=2" starts with the second product "A" record. It does NOT mean to start with the second overall record and then find a product "A".

The first 8 product A and B records are as follows, with the LEAD1 value always the same as the SALES values for the next identical product.

Table 9
Lead produced by Independent Queues

Obs	MONTH	PRODUCT	SALES	LEAD1
1	1	A	4	16
2	1	B	20	12
6	2	A	16	15
7	2	B	12	14
10	3	B	14	15
14	4	A	15	3
15	4	B	15	16
19	5	A	3	20

TOO MANY LEADS AND SET STATEMENTS? – BACK TO MERGE

What if you want to generate three different leads (say 1 period, 2 periods, and 3 periods) for each product? You could use the above approach and enter twelve SET statements, three for each of four products. Or instead you can use one MERGE statement per product. A complete program is in Appendix 2. An abridged version follows:

Example 8: Merge for multiple irregular interleaved leads

```
data irregular_leads2 (label='1, 2, 3 period leads for prod A, B, C, D');
set sales;
call missing(lead1,lead2,lead3);

select (product);
 when ('A') merge
  sales (where=(product='A'))
  sales (where=(product='A') firstobs=2 keep=product sales rename=(sales=lead1))
  sales (where=(product='A') firstobs=3 keep=product sales rename=(sales=lead2))
```

```
  sales (where=(product='A') firstobs=4 keep=product sales rename=(sales=lead3))
  ;
 when ('B') merge   ... ;
 when ('C') merge   ... ;
 when ('D') merge   ... ;
 otherwise;
end;
run;
```

Aside from replacing multiple SETs with single MERGE statements, the other notable difference in example 8 is that each merge statement includes a SALES with no firstobs parameter. By including this there is no need to protect against the merge statement terminating prematurely, as was done via the ***if eofa=0 then set … end=eofa*** statements earlier.

DYNAMICALLY GENERATING LAGS AND LEADS

Entering a LAG function (or getting leads via SET or MERGE statements) for each product works fine as long as you know what values the PRODUCT variable takes. But what if you don't know in advance what products are in the data set? A static approach is no longer sufficient. You need a technique to dynamically establish a series for each product as it is discovered in the data set. Hash objects fit this need. Their contents, like values in lag queues, persist from record to record. And they can also be created, expanded, revised, and retrieved dynamically as well.

DYNAMIC LAGS – DROP THE LAG FUNCTION

In the case of lags, this problem is solved by use of two hash objects, one (CURRSEQ) for maintaining a running count of records for each product (adding items for CURRSEQ as new products are "discovered"), and one (LAGS) to hold product-specific data for later retrieval as lagged values. The program below, after declaring these two objects, has three parts:

Example 9: Dynamically Generating Lags

```
data dynamic_lags (drop=_:);
  set sales;
  _seq=0;

  if _n_=1 then do;
    /* Hash table CURRSEQ: track current count by product      */
    declare hash currseq();
      currseq.definekey('product');
      currseq.definedata('_seq');
      currseq.definedone();

    _sl=. ;  _ml=. ;        /* To store Lagged Sales and Month */
    /* hash table LAGS: vars _SL & _ML keyed by PRODUCT & _SEQ */
    declare hash lags();
      lags.definekey('product','_seq');
      lags.definedata('_sl','_ml');
      lags.definedone();
  end;

  /* Part 1: Update _SEQ for the current PRODUCT                */
  _rc=currseq.find();      /* Retrieve _SEQ for current PRODUCT */
  _seq=_seq+1;
```

```
  currseq.replace();       /* For current PRODUCT, update _SEQ  */

  /* Part 2: Add SALES and MONTH (as data elements _SL and _ML) */
  /* to the LAGS object, keyed on PRODUCT and _SEQ              */
  lags.add(key:product,key:_seq,data:sales,data:month);

  /* Part 3: Retrieve historical value and make the monthly rate*/
  _rc = lags.find(key:product,key:_seq-1);
  if _rc=0 then change_rate=(sales-_sl)/(month-_ml);
run;
```

Here the CURRSEQ hash object maintains only one item (i.e. one "row") per product, containing the variable _SEQ, which tracks the number of records read for each product. In part 1, the "currseq.find()" method retrieves the latest _SEQ for the current product, assuming the product is already in CURRSEQ. Otherwise _SEQ is zero. _SEQ is then incremented and replaced into CURRSEQ with every incoming record for the product. Knowing the _SEQ of the current record allows easy calculation of the _SEQ for any lags.

The LAGS object (keyed on the combination of PRODUCT and _SEQ), gets one item for each incoming record, and contains a running cumulative history of two variables: _SL (lagged sales) and _ML (lagged moth). In part 2, the new data is added to LAGS ("lags.add(…)"). The use of the two "data:.." arguments assigns the values of SALES and MONTH to _SL and _ML as they are stored in LAGS.

The third major element of this program simply checks whether the product in hand is beyond its first occurrence, and if so, retrieves lagged values from LAGS and generates the needed variable. Note the "key:_seq-1" argument tells lag.find() to retrieve the item (with variables _SL and _ML) corresponding to a single period lag. An n-period lag would require nothing more than the argument "key:_seq-n".

One problem with this approach is that the hash object LAGS will ultimately grow to contain every observation in SALES, an excessive consumption of memory for large data sets, likely resulting in slower performance. You can remedy this by storing only a fixed number of lagged items for each product in the hash object, effectively replicating the fixed queue size maintained by lag functions (i.e. lag3 is a queue of 3 values). For example, by using the MOD[2] function, simply changing

```
lags.add(key:product,key:_seq,data:sales,data:month);
```

to

```
lags.replace(key:product,key:mod(_seq,3),data:sales,data:month);
```

and replacing

```
_rc = lags.find(key:product,key:_seq-1);
```

with

```
_rc = lags.find(key:product,key:mod(_seq-1,3));
```

modifies the hash object LAGS to contain no more than three items per product: i.e. the current sales value and the two prior sales values. In essence the LAGS object simply recycles _SEQ values of 0, 1, and 2. The only requirement for this approach is to use a modulus at least as large as the biggest lag.

DYNAMIC LEADS – ORDERED HASH OBJECT PLUS HASH ITERATOR

In the case of leads, if you don't know how many PRODUCTs are expected, you can't use the ***where=*** parameter that works for static leads, because you don't know in advance what values to use in the where

[2] The MOD (for modulo) function divides the first argument by the second (the modulus) and returning the remainder. As the first argument is incremented without limit the second cycles and recycles from 0 to modulus-1.

condition. And while hash objects support maintaining rolling sets of lags as new PRODUCTs are encountered, that is not feasible for leads. Instead the object needs to have a complete set of lead values as soon as the first observation is process. That problem is solved in Example 10 below, a data step with two parts:

Example 10: Dynamically Generating Leads

```
data dynamic_leads (drop=_:) ;
  /* Part 1: Populate LEADS hash object, and assign an iterator */
  if _n_=1 then do;
    if 0 then set sales (keep=product sales month
              rename=(product=_PL sales=_SL month=_ML));
    declare hash leads (dataset:'SALES (keep=PRODUCT MONTH SALES
                  rename=(PRODUCT=_PL MONTH=_ML SALES=_SL)',ordered:'A');
      leads.definekey('_PL','_ML');
      leads.definedata('_PL','_ML','_SL');
      leads.definedone();
    declare hiter li ('leads');
  end;

  /* Part 2: Read a record from sales and retrieve leads */
  set sales ;
  /* Point hash iterator at hash item for current product/month */
  li.setcur(key:PRODUCT,key:MONTH);
  _rc=li.next();         /* Advance one item in the hash object */
  if _rc=0 and _PL=PRODUCT then changerate=(_SL-sales)/(_ML-month);
 run;
```

Before reading the data set record-by-record, the LEADS object is fully populated with all the PRODUCT, SALES and MONTH values (renamed to _PL, _SL and _ML), stored in ascending order by _PL and _ML. Because the hash object is declared as ordered, moving from one item to the next within the object is equivalent to traversing the series for a single PRODUCT. The hash iterator LI supports this process.

In part 2, a record is read, and the li.setcur method points the iterator at the data item in the object corresponding to the record. Moving the iterator forward by one item ("_rc=li.next()") retrieves the _PL, _ML, and _SL values for a one period lead.

CONCLUSION

While at first glance the queue management character of the LAG function may seem counterintuitive, this property offers robust techniques to deal with a variety of situations, including BY groups and irregularly spaced time series. The technique for accommodating those structures is relatively simple. In addition, the use of multiple MERGE or SET statements produces the equivalent capability in generating leads, all without the need for extra sorting of the data set.

The more complex problem is how to address irregular time series dynamically, i.e. how to handle situations in which the user does not know how many queues will be needed. For this situation, the construction of hashes used as tools to "look up" lagged (or lead) data for each queue provides a graceful solution.

REFERENCES

Matlapudi, Anjan, and J. Daniel Knapp (2010) “Please Don’t Lag Behind LAG”. In the Proceedings of the North East SAS® Users Group.

Schreier, Howard (undated) “Conditional Lags Don’t Have to be Treacherous”. URL as or 7/1/2013: http://www.howles.com/saspapers/CC33.pdf.

CONTACT INFORMATION

This is a work in progress. Your comments and questions are valued and encouraged. Please contact the author at:

Author Name: Mark Keintz
Email: mkeintz@wharton.upenn.edu

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.

Other brand and product names are trademarks of their respective companies.

Appendix 1: Creation of sample data sets from the sashelp library

```
/*Sample 1: Monthly Stock Data for IBM, Intel, Microsoft for Aug 1986 - Dec 2005
*/
proc sort data=sashelp.stocks (keep=stock date close)
  out=sample1 (rename=(close=cls)) ;
  by stock date;
run;


/*Sample 2: Irregular Series: Monthly Sales by Product,  */
data SALES;
  do MONTH=1 to 24;
    do PRODUCT='A','B','C','D','X';
      if ranuni(09481098)< 0.9 then do;
        SALES =ceil(20*ranuni(10598067));
        output;
      end;
    end;
  end;
run;
```

Appendix 2: example program of multi-period leads for irregular interleaved time series

```
/*To get leads for each product of interest (‘A’,’B’,’C’,’D’) for three
different periods (1, 2, and 3), you might issue 12 SET statements, as per the
section on leads for irregular series.  However, only 4 MERGE statements, with
matching WHERE parameters are needed, as below */


data irregular_leads2 (label='1, 2, 3 period leads for products A, B, C, D');
set sales;

call missing(lead1,lead2,lead3);

select (product);
 when ('A') merge  sales (where=(product='A'))
  sales (where=(product='A') firstobs=2 keep=product sales rename=(sales=lead1))
  sales (where=(product='A') firstobs=3 keep=product sales rename=(sales=lead2))
  sales (where=(product='A') firstobs=4 keep=product sales rename=(sales=lead3))
  ;
 when ('B') merge  sales (where=(product='B'))
  sales (where=(product='B') firstobs=2 keep=product sales rename=(sales=lead1))
  sales (where=(product='B') firstobs=3 keep=product sales rename=(sales=lead2))
  sales (where=(product='B') firstobs=4 keep=product sales rename=(sales=lead3))
  ;
 when ('C') merge  sales (where=(product='C'))
  sales (where=(product='C') firstobs=2 keep=product sales rename=(sales=lead1))
  sales (where=(product='C') firstobs=3 keep=product sales rename=(sales=lead2))
  sales (where=(product='C') firstobs=4 keep=product sales rename=(sales=lead3))
  ;
 when ('D') merge  sales (where=(product='D'))
  sales (where=(product='D') firstobs=2 keep=product sales rename=(sales=lead1))
  sales (where=(product='D') firstobs=3 keep=product sales rename=(sales=lead2))
  sales (where=(product='D') firstobs=4 keep=product sales rename=(sales=lead3))
  ;
 otherwise;
end;

run;
```