MWSUG 2018 - Paper SB-060

From Stocks to Flows: Using SAS® Hash Objects for FIFO, LIFO, and other FO's

Mark Keintz, Wharton Research Data Services, Philadelphia, PA

ABSTRACT

Tracking gains or losses from the purchase and sale of diverse equity holdings depends in part on whether stocks sold are assumed to be from earliest lots acquired (a FIFO queue) or the latest lots acquired (LIFO). Other inventory tracking applications have a similar need for application of either FIFO or LIFO rules.

This presentation shows how a collection of simple ordered hash objects, in combination with a hash-of-hashes is a made-to-order technique for easy data-step implementation of FIFO, LIFO, and other less-likely rules, like HIFO (highest price first out) and LOFO (lowest price).

INTRODUCTION: A SIMPLE FIFO TASK

Determining the net income or loss from a table of stock holdings depends not only on calculating the number of stocks bought or sold, but in the case of stock sales, also on determining the price of the stocks when they were originally acquired. For instance the table of IBM stock holdings in Table 1 below shows aggregate holdings on each date in which a transaction occurred, as well as the price of the stock on that date. Since holdings increased in each instance from 1/30/2009 through 5/29/2009, they indicate purchases (assuming no date has multiple transactions). Given that price is available for each of those dates, the total cost of purchase is easily computed (e.g. on 31MAR2009, 400 shares were purchased at $96.89, for a total cost of $38,756).

Table 1: IBM Holdings 1/1/2009-1/11/2010

DATE	TIC	SHRS_HELD	PRICE
1/1/2009	IBM	0	
1/30/2009	IBM	100	91.65
2/27/2009	IBM	200	92.03
3/31/2009	IBM	600	96.89
4/30/2009	IBM	700	103.21
5/29/2009	IBM	800	106.28
11/16/2009	IBM	650	128.21
12/31/2009	IBM	460	130.9
1/11/2010	IBM	200	129.48

However, on 11/16/2009, when 150 shares were sold at $128.21, determining the original cost of those shares depends on which of the prior "lots" were sold, i.e. which parts of the inventory are "first out." If the commonly used First In/First Out (FIFO) rule were applied to all the sales in Table 1, the flows of IBM stock would generate the net income column in Table 2. For example, in the 11/16 sale, the FIFO rule would assign all 100 shares from the 1/30/2009 lot and 50 from the 2/27/2009 lot. The net income would be $5,465 (150*$128.21 - 100*$91.65 - 50*$92.03).

Table 2: Table of IBM Flows Using FIFO Rules

DATE	TIC	SHRS	NET_INC
1/30/2009	IBM	100	-9,165
2/27/2009	IBM	100	-9,203
3/31/2009	IBM	400	-38,756

4/30/2009	IBM	100	-10,321
5/29/2009	IBM	100	-10,628
11/16/2009	IBM	-150	5,465
12/31/2009	IBM	-190	6,705
1/11/2010	IBM	-260	8,473

1. IMPLEMENTING A SINGLE FIFO QUEUE WITH AN ORDERED HASH OBJECT

The SAS programming objective is to process the sequence of holdings records, keeping track of the changing status of each stock lot, in which the oldest lot is always sold first. This requires a programming object that (1) "retains" a data table of currently held lots over every iteration of the DATA step, (2) supports access to any "row" (especially the oldest) in the table, and (3) can be modified based on inferred sales or purchases. The hash object possesses all these attributes. Using it, as in Example 1 below, solves the central task of implementing a FIFO (or any other) queue - namely it allows dynamic management of the inventory of stock holdings. New inventory can easily be added to the hash table when stocks are purchased, and old inventory can be reduced or removed in to fulfill a sale.

Example 1: A hash table and iterator to accommodate a FIFO queue of stock holdings

```
declare hash lots (ordered:'a');
  lots.definekey('date');
  lots.definedata('date','lot_size','PRICE');
  lots.definedone();
declare hiter IL ('lots');
```

The hash table LOTS above is indexed and ordered by the variable DATE, and contains two other variables – LOT_SIZE and PRICE. The variable LOT_SIZE records the number of shares in the corresponding lot, and is adjusted downward when stocks are sold. The hash iterator IL makes it possible to apply the FIFO rule, by allowing access to the first/oldest hash item (the "IL.first" method in the program below), and iterating through subsequent items ("IL.next" method). The rogram below uses these hash objects to produce the data in Table 2 from that in Table 1.

Example 2: Generating FIFO-based Net Income for a single stock

```
data IBM_FIFO (keep=tic date shrs net_inc);
  set holdings;
  where tic='IBM';

  retain prior_shrs_held 0;

  ** Establish an ordered hash table of stock lots **;
  if _n_=1 then do;
    declare hash lots (ordered:'a');
      lots.definekey('date');
      lots.definedata('date','lot_size','PRICE');
      lots.definedone();
    declare hiter IL ('lots');
  end;

  ** SHRS>0 means share purchased, SHRS<0 means shares sold **;
  shrs = shrs_held - prior_shrs_held;
```

```
  if shrs = 0 then delete;         ** Neither purchase or sale **;
  else if shrs > 0 then do;        ** A purchase ... **;
      net_inc = -1 * shrs * PRICE;** Negative cash flow **;
      lot_size=shrs;
      rc=lots.add();               ** Add a new lot indexed by DATE **;
  end;

  else if shrs < 0 then do;        ** A sale ... **;
    shares_to_sell= abs(shrs);
    net_inc=shares_to_sell*price; ** Initialize to Sales Value **;
    sell_date=date;


    ** ********************************************************** **;
    ** This sequence steps through the hash table, reducing prior  **;
    ** LOT_SIZEs, and subtracting their original cost from NETINC  **;
    ** until total SHARES_TO_SELL volume has been fulfilled        **;
    ** ********************************************************** **;

    do rc=IL.first() by 0 while (shares_to_sell>0);
      from_this_lot=min(lot_size,shares_to_sell);
      net_inc = net_inc - from_this_lot*PRICE;
      shares_to_sell=shares_to_sell - from_this_lot;
      lot_size = lot_size - from_this_lot;
      rc=lots.replace();
      rc=IL.next();
    end;

    /*  Want to monitor changes in LOTS?    Use the output method   */
    /* rc=lots.output(dataset:cats('LOTS_',_n_));                    */

    date=sell_date;
  end;
  prior_shrs_held = shrs_held;
  net_inc=round(net_inc,.01);
run;
```

The logic of the program above is straightforward, with the following major components:

1. When the HOLDINGS data implies a stock purchase (SHRS>0), then:
    a. The new lot is added to the hash table. But note that the size of the lot is copied from SHRS to LOT_SIZE. This is done so that the lot size value can be manipulated without impacting the SHRS variable as new records are read from HOLDINGS.
    b. The cost of the stock purchase, calculated as a negative cash flow, is recorded in NET_INC
2. When the HOLDINGS data implies a stock sale (SHRS<0), earlier lots will be processed chronologically to find the first available lots that can completely or partially satisfy the sale. Specifically it
    a. Calculates the initial number of SHARES_TO_SELL.
    b. Retrieves the oldest lot ("IL.FIRST()" method) and if needed, subsequently retrieves more lots in order (the "IL.NEXT()" method).
    c. For each lot, see if there are available stocks to sell. If yes, then reduce both the corresponding LOT_SIZE and SHARES_TO_SELL by the same amount

(FROM_THIS_LOT). Also subtract the original cost of those shares (FROM_THIS_LOT*PRICE) from NET_INC.

3. While in the middle of developing a program like the above, you'll want to track changes in the LOTS object as each new holdings record is processed. The often-used PUT and PUTLOG statements are not well-suited to this task, but hash OUTPUT method is built-to-order. It converts each row of the object into a single observation in a SAS data set. If you de-comment the output method in example 2, it will generate one data set per holdings observation. The first, named LOTS_1 will have one observation. The tenth, LOTS_10 will have 10. Unfortunately the output method can't add records to a data set – it just overwrites the entire data set.

Because the program above treats only one stock, only one ordered hash table is required. Two hash tables (and two hash iterators) would be needed to deal with two stocks. The general problem, of course, is how to handle an unknown number of stocks, given that one hash table and iterator is needed for each stock.

2. USING HASH-OF-HASHES TO MANAGE MULTIPLE STOCK HOLDINGS

The solution to this problem is to create a new hash table only when a new stock is encountered, rather than declaring a fixed number of hash tables in advance. Of course the related problem is to switch to the appropriate hash table as incoming data records switch between stock tickers. The technique for the latter is to track multiple hash objects the same way as individual hash objects track multiple stock lots – that is create a hash of hashes.

The hash of hashes ("hoh" below) can be constructed as here:

Example 3: A hash of hashes for managing multiple stocks

```
declare hash hoh ();
  hoh.definekey('tic');
  hoh.definedata('tic','LOTS','IL','prior_shrs_held');
  hoh.definedone();
```

At this point, SAS knows that HOH is a hash object, but the names LOTS and IL could just as well be variables like TIC and PRIOR_SHRS_HELD. So you need to issue a pair of unaccompanied DECLARE statements to let SAS know that LOT and IL are not variables, but hash object and iterator, respectively.

```
** DECLARE, in preparation for multiple instantiations **;
  declare hash LOTS;
  declare hiter IL;
```

What's different about the HOH table? The definedata method lists not only ordinary SAS variables (TIC and PRIOR_SHRS_HELD), but also the hash object LOTS and the hash iterator IL. In other words, if an instance of LOTS and IL has been established, then they could be valid elements of an item in the HOH table. You might want to think of them as ***pointers*** to the corresponding data collections. Notice also that LOTS and IL are only declared as hash objects in the above script, but they have not been instantiated at this point. That's only done later when each new stock is encountered.

So the general logic of the full program (in Example 4) is now this:

1. After reading a HOLDINGS record, see if the stock ticker is already in HOH (the "hoh.find()" method).
    a. If not then create new instances of the LOTS and IL objects, initialize PRIOR_SHRS_HELD and add a corresponding "row" to the HOH object. LOTS and IL will now be managing data

for the new ticker.

b. If yes, then the hoh.find method will not only retrieve the current value of PRIOR_SHRS_HELD for the stock, but also make the name LOTS refer ("point") to the correct collection of data (i.e. data for a given ticker) and name IL refer to the corresponding iterator.

2. At this point, apply the logic of the prior example for purchases and sales of stocks. After the appropriate LOTS object is updated, replace the revised value of PRIOR_SHRS_HELD in HOH.

Example 4: Complete hash of hashes DATA step for managing multiple stocks

```
***** Multiple Equities - Hash of Hashes *****;

data flows_fifo (keep=tic date shrs net_inc);
  set holdings;

  if _n_=1 then do;
    declare hash hoh ();
      hoh.definekey('tic');
      hoh.definedata('tic','lots','IL','prior_shrs_held');
      hoh.definedone();

    declare hash lots;      ** ready for multiple instantiations **;
    declare hiter IL;       ** ... same ... **;
  end;

  ** See if this ticker is in HOH already.  If not make a new   **;
  ** instance of LOTS and IL                                    **;
  rc=hoh.find();
  if rc ^= 0 then do;                  ** If this TIC is new ... **;
    lots = _new_ hash (ordered:'a');
    lots.definekey('date');
    lots.definedata('date','lot_size','price');
    lots.definedone();

    IL = _new_ hiter('lots');

    prior_shrs_held=0;
    rc=hoh.add();                   ** Add new ticker to HOH ... **;
  end;

  ** Shares transacted (positive means buy, negative means sell) **;
  shrs = shrs_held - prior_shrs_held;

  if shrs = 0 then delete;
  else if shrs > 0 then do;      ** If a BUY, add a new lot **;
    net_inc = -1 * shrs * price;
    lot_size=shrs;
    rc=lots.add();
  end;

  else if shrs < 0 then do;     ** If SELL, reduce old lot(s) **;
    shares_to_sell= abs(shrs);
    net_inc=shares_to_sell*price;
```

```
    sell_date=date;

    do rc=IL.first() by 0 while (shares_to_sell>0);
      from_this_lot=min(lot_size,shares_to_sell);
      net_inc = net_inc - from_this_lot*price;
      shares_to_sell=shares_to_sell - from_this_lot;
      lot_size = lot_size - from_this_lot;
      rc=lots.replace();
      rc=IL.next();
    end;
    date=sell_date;
  end;

  prior_shrs_held = shrs_held;
  rc=hoh.replace();
  net_inc=round(net_inc,.01);
run;
```

What the above depends upon is that any hash object can have multiple INSTANCES, in this case generated by the "LOTS = _NEW_" and "IL = _NEW_" statements. Basically, LOTS and IL are just pointers to data collections, one for each ticker. However, the instantiations must occur before the corresponding hoh.add method. Otherwise the LOTS object for the most recently used ticker would be added to hoh instead.

3. LIFO AND OTHER FO'S – ADJUSTING TO OTHER RULES

3.1 LIFO

Another common queue management scheme is Last In/First Out (LIFO). This process is so like the FIFO queue that the above program can accommodate it by changing one letter, namely converting the statement

```
    lots = _new_ hash (ordered:'a');
```

to

```
    lots = _new_ hash (ordered:'d');
```

This converts the order of items in LOTS from ascending to descending. The first "row" in the LOTS table is now the youngest, successive rows are progressively older, and the program above will implement a LIFO queue.[1]

3.2 HIFO and LOFO (look out for tied keys)

Instead of selling from the oldest or youngest lots, you might want to compute net income using the most costly (HIFO) or least costly (LOFO) inventory. This suggests that you could simply define PRICE as the hash key, by changing

```
    lots.definekey('DATE');
```

to

[1] Another alternative is to keep the ascending order, and replace IL.FIRST() with IL.LAST()_, and IL.NEXT() with IL.PREV().

```
    lots.definekey('PRICE');
```

And this indeed would be all that's needed … **if** there were only one stock lot for each price, just as the prior examples had one lot per date. But prices can recur over time for any given stock and multiple lots at a given price are highly likely. The problem is that the default hash object only accommodates one item ("row") per key value.

There are a number of ways to deal with tied key values. If you have SAS 9.2 or later, you can utilize the MULTIDATA:'Y' hash option to support items with duplicate keys, which would also require use of additional methods specific to MULTIDATA hash objects. A complete example using this technique is shown in appendix 1.

Or you could effectively "de-duplicate" by defining a multi-component key that is unique. I.e. just change

```
  lots.definekey('DATE');
```

to

```
    lots.definekey('PRICE','DATE');
```

and run the program as above to implement LOFO (or HIFO) queues.

The sample below takes a third approach. It collapses multiple purchases at a given price into a single hash item, tracking total shares purchased at that price. For each new purchase, it looks for pre-existing records at the same PRICE, and then adds to the corresponding LOT_SIZE. That is, the program below manages purchases similar to how sales were treated above.

Example 5: LOFO for multiple stocks (managing duplicate prices)

```
***** Introduce LOFO (Lowest Price In, First Out *****;

data flows_lofo_dell (keep=tic date shrs net_inc);
  set holdings;

  if _n_=1 then do;
    declare hash hoh ();
      hoh.definekey('tic');
      hoh.definedata('tic','lots','IL','prior_shrs_held');
      hoh.definedone();

      ** DECLARE, in preparation for multiple instanstiations **;
    declare hash lots;
    declare hiter IL;
  end;

  rc=hoh.find();
  if rc ^= 0 then do;                  ** If this TIC is new ... **;
    lots = _new_ hash (ordered:'a');   ** Instantiate corresponding hash **;
    lots.definekey('PRICE');
    lots.definedata('PRICE','lot_size'); ** Note no DATE variable **;
    lots.definedone();

    IL = _new_ hiter('lots');          ** Instantiate corresponding hiter **;

    prior_shrs_held=0;
    rc=hoh.replace();                  ** And add item to HOH ... **;
  end;

  ** Shares transacted (positive means buy, negative means sell **;
```

```
  shrs = shrs_held - prior_shrs_held;

  if shrs = 0 then delete;
  else if shrs > 0 then do;      ** If BUY, add new lot, or add to old lot **;
    if lots.find() ^=0 then lot_size=shrs;
    else                    lot_size=lot_size+shrs;
    net_inc = -1 * lot_size * PRICE;
    rc=lots.replace();
  end;

  else if shrs < 0 then do;     ** If SELL, reduce old lot(s) **;
    shares_to_sell= abs(shrs);;
    net_inc=shares_to_sell*price;
    sell_date=date;

    do rc=IL.first() by 0 while (shares_to_sell>0);
      from_this_lot=min(lot_size,shares_to_sell);
      net_inc = net_inc - from_this_lot*PRICE;
      shares_to_sell=shares_to_sell - from_this_lot;
      lot_size = lot_size - from_this_lot;
      rc=lots.replace();
      rc=IL.next();
    end;
    date=sell_date;
  end;

  prior_shrs_held = shrs_held;
  rc=hoh.replace();
  net_inc=round(net_inc,.01);
run;
```

While the HIFO and LOFO rules are far less frequently applied for generating net income values than FIFO and LIFO, they do have a number of other more prominent uses. In particular estimating the National Best Bid and Offer at any given time by assessing (frequently tied) quotes from multiple stock exchanges covering thousands of stocks is likely a very direct use of this concept. Brokers are obligated to execute trades at the best available bid and offer values. The hash object is a very likely tool for modelling this process in SAS.

4. HASH OBJECTS REQUIRE MEMORY

Hash tables and iterators are Data Step Component Objects, and are memory-resident. Each newly instantiated object increases memory demand, and an object's memory requirements can grow as items are added. With a large enough HOLDINGS file, it's possible that memory needs will impact performance. In the programs above, items whose LOT_SIZE has reached zero are no longer needed but remain in memory. Appendix 2 shows a program which removes items with LOT_SIZE=0, and deletes entire hash tables from memory when holdings reach zero.

CONCLUSIONS

The ordered hash object is probably the most suitable tool in the SAS environment for managing queues, making the implementation of FIFO, LIFO, and related rules into an essentially trivial programming task. Because hash objects can contain other hash objects, the ability to simultaneously manage multiple queues is achievable with very little programming effort.

ACKNOWLEDGMENTS

I have benefitted greatly from discussions on the use of hash tables by Paul Dorfman on SAS-L, as well as from his related presentations at multiple SAS User Group meetings. Of course, the first demonstration that I saw of managing hash-of-hashes in SAS was provided by Richard DeVenezia

CONTACT INFORMATION

Author:	Mark Keintz
Address:	Wharton Research Data Services University of Pennsylvania
Email	mkeintz@wharton.upenn.edu

APPENDIX 1: LOFO PROCESSING VIA USE OF MULTIPLE ITEMS PER PRICE KEY (MULTIDATA:"Y")

Note: In addition to the "multidata:'Y'" option, the program below also uses methods specific to hash objects with duplicate keys, namely find_next() and replacedup().

```
*** Introduce LOFO (Lowest Price In, First Out, Using "MULTIDATA:'Y' hash
object ***;

data flows_lofo (keep=tic date shrs net_inc);
  set holdings;

  if _n_=1 then do;    ** Make a hash to track stock identities **;
    declare hash hoh (ordered:'a');
      hoh.definekey('tic');
      hoh.definedata('tic','lots','IL','prior_shrs_held');
      hoh.definedone();

    **DECLARE allows for later instantiations of LOTS hashes (1 per stock)**;
    declare hash lots;
    declare hiter IL;
  end;

  rc=hoh.find();              ** Find this stock in HOH **;
  if rc ^= 0 then do;         ** If this TIC is new (not yet in HOH) ... **;
    ** Instantiate new LOTS table, supporting multiple items per PRICE**;
    lots = _new_ hash (ordered:'a',multidata:'Y');
    lots.definekey('PRICE');
    lots.definedata('PRICE','DATE','lot_size');
    lots.definedone();

    IL = _new_ hiter('lots'); ** Instantiate corresponding hash iterator **;

    prior_shrs_held=0;
    rc=hoh.add();             ** And add item to HOH ... **;
  end;

  ** Shares transacted (positive means buy, negative means sell **;
  shrs = shrs_held - prior_shrs_held;

  if shrs = 0 then delete;    ** If no flow, then no output record **;
  else if shrs > 0 then do;   ** If BUY, add a new lot item**;
    lot_size=shrs;
    net_inc = -1 * lot_size * PRICE;
    rc=lots.add();
  end;

  else if shrs < 0 then do;   ** If SELL, reduce old lot(s) **;
    shares_to_sell= abs(shrs);
    net_inc=shares_to_sell*price;
    sell_date=date;

    rc=IL.first();  ** Go to beginning of LOT (item with Lowest Price) **;
    do rc=lots.find() by 0 while (shares_to_sell>0);
      if lot_size>0 then do;
        from_this_lot=min(lot_size,shares_to_sell);
```

```
        net_inc = net_inc - from_this_lot*PRICE;
        shares_to_sell=shares_to_sell - from_this_lot;
        lot_size = lot_size - from_this_lot;
        rc=lots.replacedup();
      end;
      rc=lots.find_next();  ** Get next item at this price **;
      if rc^=0 then do;     ** If no more items at this price then ... **;
         rc=il.next();      ** ... Tell iterator to go to next price **;
         rc=lots.find();  ** ... Get first entry at this price ... **;
      end;
    end;
    date=sell_date;
  end;

  prior_shrs_held = shrs_held;  ** Update Prior Shares Held ... **;
  rc=hoh.replace();             ** and replace it in HOH **;
  net_inc=round(net_inc,.01);
run;
```

APPENDIX 2: CONSERVING MEMORY BY REMOVING EMPTY LOTS

Note the program below is Example 4 with additional statements (in *italics*) used to remove cases in which LOT_SIZE=0.

```
data flows_fifo (keep=tic date shrs net_inc);
  set holdings;

  if _n_=1 then do;
    declare hash hoh ();
      hoh.definekey('tic');
      hoh.definedata('tic','lots','IL','prior_shrs_held');
      hoh.definedone();

      ** DECLARE, in preparation for multiple instantiations **;
      declare hash lots;
      declare hiter IL;

    ** This hash is to track which lots are empty ("elots") **;
    declare hash elots();
      elots.definekey('date');
      elots.definedata('date');
      elots.definedone();
    declare hiter IEL ('elots');
  end;

  ** See if this ticker is in HOH already.  If not make a new  **;
  ** intance of LOTS and IL                                     **;
  rc=hoh.find();
  if rc ^= 0 then do;                  ** If this TIC is new ... **;
    ** Now instantiate the hash declared above **;
    lots = _new_ hash (ordered:'a');
    lots.definekey('date');
    lots.definedata('date','lot_size','price');
    lots.definedone();

    ** Instantiate corresponding hiter **;
    IL = _new_ hiter('lots');

    prior_shrs_held=0;
    rc=hoh.replace();                  ** And add item to HOH ... **;
  end;

  ** Shares transacted (positive means buy, negative means sell) **;
  shrs = shrs_held - prior_shrs_held;

  if shrs = 0 then delete;
  else if shrs > 0 then do;      ** If a BUY, add a new lot **;
    net_inc = -1 * shrs * price;
    lot_size=shrs;
    rc=lots.add();
  end;

  else if shrs < 0 then do;     ** If SELL, reduce old lot(s) **;
    shares_to_sell= abs(shrs);;
    net_inc=shares_to_sell*price;
    sell_date=date;
```

```
      do rc=IL.first() by 0 while (shares_to_sell>0);
        from_this_lot=min(lot_size,shares_to_sell);
        net_inc = net_inc - from_this_lot*price;
        shares_to_sell=shares_to_sell - from_this_lot;
        lot_size = lot_size - from_this_lot;
        rc=lots.replace();

        if lot_size=0 then rc=elots.add();
        rc=IL.next();
      end;

    ** Clear away empty lots, if any**;
    if elots.num_items > 0 then do;
      do rc=IEL.first() by 0 while (rc=0);
        rc=lots.remove();
        rc=IEL.next();
      end;
      rc=elots.clear();
    end;

    date=sell_date;
  end;

  prior_shrs_held = shrs_held;
  rc=hoh.replace();
  net_inc=round(net_inc,.01);

  ** If stock is completely depleted, delete its hash objects from memory**;
  ** and remove the corresponding item from HOH **;
  if shrs_held=0 then do;
    rc=il.delete();
    rc=lots.delete();
    rc=hoh.remove();
  end;
run;
```