Paper 1331-2017

Haven't I Seen You Before? An Application of DATA Step HASH for Efficient Complex Event Associations

John Schmitz, Luminare Data LLC

ABSTRACT

Data processing can sometimes require complex logic to match and rank record associations across events. This paper presents an efficient solution to generating these complex associations using the DATA step and data hash objects. The solution applies to multiple business needs including subsequent purchases, repayment of loan advance, or hospital readmits. The logic demonstrates how to construct a hash process to identify a qualifying initial event and append linking information with various rank and analysis factors, through the example of a specific use case of the process.

INTRODUCTION

This paper demonstrates the use of hash tables to quickly process complex record association logic. Record association logic can be difficult and slow to process using more traditional methods such as sorting and DATA steps or SQL joins. Requirements to capture multiple and conditional data characteristics and managing potential duplicate matches that may accompany such requests can dramatically increase the data processing challenge.

In this paper, sample data is generated as a baseline for the processing example. These generated data are representative of retail consumer purchase data and are used to process a study of consumer purchase activities. These methods can be easily adapted to similar processing needs in financial services, retail, heath care and other industries. Full code has been included with the paper so that readers may construct and test other variations of the approach.

THE STUDY SCENARIO

The study scenario considers an electronics store reviewing customer activity following a new computer purchase. The primary question regards expenditures for 30 days following that initial purchase. The extraction process is complicated by the question of how to handle customers who purchase a second or even third computer within that window, as well as other related questions that interests management. After considerable discussion, the following criteria was established for the data collection effort:

- Any transaction by the customer on the same day and in the same store would be treated as part of the original computer purchase. Dollar amounts of those transactions would be included with the original purchase amount, not with added purchases.
- Excluding same day, same store purchases, any transaction prior to the initial computer purchase can be ignored.
- If the customer made a same day purchase in another store, it would be counted as an added purchase, but only if it occurs after the computer purchase. This event should also trigger a flag as the customer may have been forced to a second store due to inventory issues.
- The analysis would include transactions for the customer over the next 30 days, but should be limited to no more than 5 additional transactions.
- Music and video purchases would only be included at 50% of transaction amount and would not count towards the 5 transaction limit.
- Finally, if the customer purchased an additional computer during the 30 day period, the search period should be extended to at least 15 days following the latter computer purchase. However, all activity should be attributed to the initial purchase, not the second computer purchase.

Transaction data for the study includes a master customer identifier (CUST_ID), a transaction identifier (TRANS_ID), a store indicator (STORE_ID), a purchased product code (PROD_ID), the transaction

timestamp (TRANS_DT), and the transaction amount (PURCHASE_AMT). TRANS_ID is sequentially assigned and is a reliable indicator of transaction order. PROD_ID are assigned values 1 through 5 and are described as 'hardware', 'software', 'accessories', 'other electronics', and 'music & video', respectively. Computer purchases are coded as PROD_ID=1.

As previously mentioned, data that meet these requirements have been generate for the effort. The focus here is on using hash tables and the associated logic to process these records. The discussion here will focus on the hash process elements only. Complete code that includes both data generation and the hash process is included in an Appendix.

CREATING THE HASH DATA

Hash data is extracted from the existing transaction data using a data view. That view appears as:

```
data lookup_vw / view=lookup_vw;

    ** LOGIC TO READ AND SELECT ALL HARDWARE PURCHASES;
    set transactions;
    where prod_id = 1;
    drop prod_id;

    ** RENAME FIELDS DUPLICATED IN HASH;
    rename  trans_id = hash_trans_id
            trans_date = hash_trans_date
            store_id = hash_store_id
            purchase_amt = hash_purchase_amt;

    ** GENERATE ADDITIONAL FIELDS TO RETAIN DATA IN HASH;
    hash_days = 30;
    hash_added_purchases = 0;
    hash_trans_count = 0;
    hash_inv_warning = 0;
run;
```

The view includes a filter for hardware (PROD_ID=1), renames fields to include a HASH_ prefix, and creates four new fields used and updated by the iterative hash search process. The prefix is needed to distinguish hash table fields from the corresponding fields read from TRANSACTIONS in the next DATA step. Only the hash key field, CUST_ID, is not renamed in the hash object.

In the following example, all accumulated data are managed and retained in the hash table. However, the HASH values are also retained in the main output dataset (SEARCH) of the hash step. Analysis can be conducted from this table as well. For brevity, that output is not discussed here.

The next step creates a DATA step with an output table SEARCH. The code to declare that hash is shown here:

```
declare hash lookup (dataset:'Lookup_vw',
    multidata:'Y', ordered: 'N');
rc = lookup.definekey('cust_id');
rc = lookup.definedata( 'cust_id', 'hash_trans_id',
   'hash_store_id', 'hash_trans_date', 'hash_days',
   'hash_purchase_amt', 'hash_added_purchases',
   'hash_trans_count','hash_inv_warning');
rc = lookup.definedone();
call missing (cust_id, hash_trans_id, hash_store_id,
   hash_trans_date, hash_days, hash_purchase_amt,
   hash_added_purchases,hash_trans_count,
   hash_inv_warning);
format hash_trans_date date9.;
```

Within SEARCH, the data from the previous view is loaded to memory as a hash object for the subsequent search effort. This code uses mostly common hash definition elements, including the DECLARE statement, calls to standard hash methods, including DEFINEKEY, DEFINEDATA, and DEFINEDONE methods and usage of CALL MISSING on all variables in the hash. Readers who are unfamiliar with these common hash methods should review SAS documentation or a more introductory discussion on hash tables.

Two elements in this definition are particularly relevant to the exercise at hand: the MULTIDATA and ORDERED tag settings on the DECLARE statement. The MULTIDATA tag indicates that the key values used, in this case CUST_ID, may not be unique within the hash. The ORDERED tag tells the hash that the key order is undefined. By default, both tags would be set to 'N' so the ordered tag setting would ultimately have no effect. It is included since the process assumes the hash is in time order, not in customer order. The MULTIDATA tag changes the default behavior and is required to allow proper analysis when a customer may have multiple computer purchases to review.

USING TRANSACTIONS TO UPDATE HASH VALUES

Now that data and hash structures have been defined, a DATA step using a hash object can be used to track transactions that match the previously defined criteria. To accomplish this, the data step will read each transaction and search the hash to determine if a preceding computer purchase exists and if all relevant criteria are met.

The initial effort is accomplished with a SET statement and hash FIND method:

```
set transactions end=last;
rc=lookup.find();
if rc ^= 0 then done = 1;
```

A SET statement is used to read all TRANSACTIONS data. The option END=LAST is added to generate a last record indicator, used below. The hash FIND method returns the first computer purchase record in hash for the CUST_ID read from TRANSACTIONS. If a record is found, RC will be 0 and the transaction will be checked against other criteria below. If none are found, RC will be non-zero, resulting in DONE being set to 1. This indicates no additional evaluation is required for this transaction and the process may proceed to the next TRANSACTIONS record. Note that the FIND method searches from the top of the hash. Since the hash is time ordered, the returned hash record will be for the earliest computer purchase recorded in hash for that customer.

For transactions where RC = 0, so not DONE, a loop structure is used that can review each record in the hash to determine if it meets all the criteria established in the scenario above. The loop structure itself, excluding the internal logic represented by …, is straight forward:

```
do i = 1 to 1000 while (^done);
    …
end; ** DO WHILE LOOP **;
```

A DO WHILE loop is used so that the previously assigned DONE value is evaluated prior to entering the the first pass of the loop. Hence, the loop will not execute when the customer has no computer purchases to evaluate. The loop counter (i=1 to 1000) is optional, but adds a forced exit to help insure that an error could not trigger an infinite loop.

Inside the loop, a series of checks are performed to determine which of the various possible items identified in the scenario may apply. These steps will be reviewed one section at a time. It is important to recognize this is an if-then-else structure inside the loop. Once an if statement is found where all the criteria set evaluate a true, no additional if statements within the if-then-else block are evaluated.

Check 1, the first IF statement, looks for purchases on same day and same store:

```
if trans_date = hash_trans_date and store_id = hash_store_id
    and trans_id ^= hash_trans_id then do;
```

```
    hash_purchase_amt = sum(hash_purchase_amt,purchase_amt);
    done = 1;
end;
```

This logic verifies same day, same store and necessarily excludes the matching transaction already loaded in hash. The PURCHASE_AMT of any matching records are added to HASH_PURCHASE_AMT) as requested. Once assigned, there is no need for additional evaluations on this transaction. DONE is assigned 1 and the loop may complete. The process will read the next record from TRANSACTIONS and continue the process. The updated value of HASH_PURCHASE_AMT currently exists in the transaction record only. A later step is used to trigger an update of hash data prior to completing the DO WHILE loop.

Check 2 evaluates time order based on TRANS_ID:

```
else if trans_id <= hash_trans_id then do;
    done = 1;
end;
```

Transaction that occurs before the hash record was generated are not considered, excluding same day, same store items addressed in Check 1. Since the search is time ordered, any subsequent purchases would also occur after this transaction as well. The search may safely conclude that no matches exist that satisfy the established criteria. It sets DONE=1 to exit the loop and proceed to the next transaction record.

Check 3 determines if transaction date occurs within the search days:

```
else if trans_date > hash_trans_date + hash_days then do;
    done = 0;
end;
```

If this evaluates as false, the transaction occurs after the search days have concluded. However, it is possible that a later computer purchase may provide a satisfactory match. Therefore, it assigns DONE=0. This will trigger logic below to read the next hash record and repeat the loop.

By Check 4, an acceptable match has been established based on timing and date requirements. However, the scenario definition limits to 5 matches. A check against the current match count for this hash record is performed. If 5 transactions have already been recorded in hash record, this one will be skipped and the loop set to terminate:

```
else if hash_trans_count >= 5 then do;
    done = 1;
end;
```

If none of the above checks have evaluated as true, there is a satisfactory match to all established criteria and the transaction must be recorded to the hash record. This section updates the four fields in hash that record subsequent purchase activity:

- HASH_TRANS_COUNT – counter for added transactions, excluding PROD_ID=5,
- HASH_ADDED_PURCHASES – subsequent purchases, with 50% of purchase for PROD_ID=5,
- HASH_INV_WARNING – A flag for same day, different store, following a computer purchase,
- HASH_DAYS – number of days to search, updated for additional computer purchases.

The logic here should be straight-forward to most readers. It determines if specific criteria for an adjustment are met and computes the new value for the hash record based on that assessment. Those checks and updates are:

```
else do;
    if prod_id ^= 5 then do;
          hash_added_purchases = sum(hash_added_purchases, purchase_amt);
          hash_trans_count = sum(hash_trans_count,1);
```

```
    end;
    else do;
          hash_added_purchases =
                sum(hash_added_purchases, 0.5 * purchase_amt);
    end;
    if trans_date = hash_trans_date then do;
          hash_inv_warning=1;
    end;
    if prod_id = 1 then do;
          hash_days = max(hash_days,trans_date - hash_trans_date + 15);
    end;
    done = 1;
end;
```

All required hash values have now been computed, but currently exist in the transaction record only. It is necessary to update the hash object with these new values. Also note that there is a potential adjustment made above that would alter HASH_PURCHASE_AMT. That too only exists currently on the transaction record, not in the hash record. The hash record update is accomplished through using the REPLACEDUP method:

```
rc = lookup.replacedup();
```

REPLACEDUP is required rather than the more common REPLACE method due to the MULTIDATA option in the hash. It is applied outside of the previous section so that it can capture changes made here or to the purchase amount made earlier in the loop. REPLACEDUP will replace the current record in the hash object, not all matching objects. Since this replaces the current record, it is essential that this statement run before a FIND or FIND_NEXT executes, or the desired information will be lost.

In the logic above, only CHECK 3 results in a requirement to iterate the loop to look for a potential subsequent computer purchase that may generate a suitable match. The next step is to trigger a search for that next record:

```
rc = lookup.replacedup();

if done=0 then do;
    rc=lookup.find_next();
    if rc ^= 0 then done=1;
end;
```

Earlier, the logic used a FIND method call to find the first computer purchase recorded in the hash. Inside the loop, that is replaced with a FIND_NEXT method call. The FIND_NEXT call is used in conjunction with the MULTIDATA tag. Rather than starting at the top of the hash as was done by FIND, FIND_NEXT begins with the current record and continues the search. Because the hash records are time ordered, FIND_NEXT will retrieve the next computer purchase recorded for the customer and repeat the loop. As with the FIND logic used initially, it is necessary to insure that a subsequent record is found. If so, RC will equal zero and the loop will iterate. Otherwise, DONE is assigned 1 and the loop will terminate, allowing the process to continue with the next transaction record.

Finally, upon concluding the review of all transactions, it is necessary to output the hash data. This is done with the output command:

```
if last then rc = lookup.output(dataset: 'trans_summary');
```

This commands executes on LAST, generated by the SET statement above. This command writes all contents of hash to a dataset ‘trans_summary’. This is a standard data set and can be further processed as any SAS dataset would be managed.

CONCLUSION

The process shown here provides a clean and efficient method to handle complex record association logic. The code was executed against SAS University Edition on a basic laptop with 1.5 gb RAM and 2 CPUs allocated to the SAS virtual machine. Using the code that is attached on that VM, a 1 million record transaction file was completed in less than 6 seconds. The hash is memory resident so performance will suffer notably if the hash size exceeds available system RAM.

The code used here has been tested against SAS 9.4. It has not been tested but should run correctly on SAS 9.3. I have not considered or reviewed code on older SAS versions.

CONTACT INFORMATION

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

John Schmitz
Luminare Data LLC
john.schmitz@luminaredata.com
www.luminaredata.com