Dataset Self-Joins for Aggregation

Steve Fleming - Early Warning Services

Abstract

Have you ever needed to aggregate data within a dataset up to a certain record? For example, you might count how many transactions an account has had prior to the current transaction. Using a PROC SQL self-join can accomplish this task. This approach will be compared to Data Step Hash code, and advantages and disadvantages of each approach will be discussed.

Introduction

As part of variable generation we often need to calculate some aggregation function on all observations for a particular subject that precede the current observation by a certain time span. Let's say we have a dataset of transactions with variables:

- Transaction ID
- Transaction date
- Account

If we need to count how many transactions the account on the current transaction has had in the last 30 days, a first attempt at this task might be to create a new dataset by merging the current transaction to all records with the same account that took place in the past 30 days. Then, for each transaction ID, a simple count aggregation of records is performed. Once you get the hang of this, you can add other aggregations like summing the amount of transactions in a time period.

Sample data

A sample dataset for this demonstration can be generated with the following SAS code. There are 10 accounts observed over a time period of 3 months. Each account has a 10% chance of making a transaction each day.

```
data dat.example;
      call streaminit(4330906);  /* Initialize the random number stream */
                                 /* so results are repeatable */
      length tran_id $10;        /* The transaction ID */
      format day date9.;         /* The transaction date */

      do day = '01Dec2015'd to '29Feb2016'd;   /* Looping over 3 months */
            do account = 1 to 10;              /* For 10 accounts */
                  if rand('UNIFORM') < 0.1 then do;
                        /* 10% chance of account transacting each day */
                        tran_id = cats(put(day, yymmdd8.),
                                       put(account, z2.));
                        output;
                  end;
            end;
      end;
run;
```

Some sample records show that account 2 had a transaction on December 1 and 2. We will want our results to show that on the 2nd date, account 2 will have had 1 transaction in the past 30 days.

Obs	tran_id	day	account
1	15-12-0102	01DEC2015	2
2	15-12-0108	01DEC2015	8
3	15-12-0110	01DEC2015	10
4	15-12-0202	02DEC2015	2
5	15-12-0206	02DEC2015	6

PROC SQL Approach

In this approach, we first associate each transaction with every transaction for the same account that occurred in the past 30 days. Then we count the number of matching prior transactions and add in the zeros for when there are no prior transactions in the past 30 days.

```
proc sql;
create table procSQLmulti as
select a.tran_id,
       a.day,
       a.account,
       b.tran_id as prior_tran_id
                              /* 1. These are the prior transactions for */
                              /*    this account that occurred in the    */
                              /*    past 30 days.                        */
from dat.example a            /* 2. Since you are reading from the same */
join dat.example b            /*    dataset twice, you have to use      */
                              /*    dataset aliases                     */
on a.account = b.account      /* 3. Accounts match */
where a.day - 30 <= b.day < a.day
                              /* 4. Prior Transaction occurerd in past */
                              /*    30 days.  */
;
create table dat.procSQLmulti_agg as
select b.tran_id,
       b.day,
       b.account,
       max(0, count(a.prior_tran_id)) as num_prior_trans
                              /* 5. Count the prior transactions     */
from procSQLmulti a
right join dat.example b      /* 6. Use right join to add back the   */
                              /*    transactions with no prior       */
                              /*    transactions in past 30 days     */
on a.tran_id = b.tran_id
group by b.tran_id,           /* 7. Group by the current transaction */
       b.day,
       b.account
;
quit;
```

A print out of the intermediate dataset shows how the transactions get linked to prior transactions. For example, transaction 15-12-0202 gets linked to prior transaction 15-12-0102 because they have the same account and the prior one occurred only 1 day prior.

Obs	tran_id	day	account	prior_tran_id
1	15-12-0202	02DEC2015	2	15-12-0102
2	15-12-0510	05DEC2015	10	15-12-0110
3	15-12-0603	06DEC2015	3	15-12-0403
4	15-12-0703	07DEC2015	3	15-12-0403
5	15-12-0703	07DEC2015	3	15-12-0603

A print out of the aggregated dataset shows that all of the December 1 transactions have no prior transactions as expected.

Obs	tran_id	day	account	num_prior_trans
1	15-12-0102	01DEC2015	2	0
2	15-12-0108	01DEC2015	8	0
3	15-12-0110	01DEC2015	10	0
4	15-12-0202	02DEC2015	2	1
5	15-12-0206	02DEC2015	6	0

Data Step Approach

After mulling around with different Data Step approaches, we settled on a hash object comparison as the most straightforward way to accomplish the Cartesian join that is inherent in PROC SQL. We still end up passing through the data twice, but the filling of zeros occurs more naturally as part of the process. A disadvantage is the full dataset must fit into memory for the hash object to load. I have left in some PUT Statements that helped me debug this step as hashing is still relatively new to me.

```
data prior;
      set dat.example                      /* 1. Make a copy of the data. */
            (drop = tran_id                /* 2. Not used in hash so more */
                                           /*    efficient to drop.       */
             rename=(day     = prior_day   /* 3. Avoid naming conflicts */
                     account = prior_account))  ;
run;

data dat.data_step_agg ;
      length prior_day 8 prior_account 8 ; /* 4. Have to initialize data */
                                           /*    brought from the hash   */
      format prior_day date9.;

  if _N_ =1  then do;               /* 5. First time through,load all of  */
                                    /*    the hash data into memory.      */
      dcl hash hmatch;              /* 6. Create the hash */
      retain hmatch;                /* 7. Retain it for subsequent */
                                    /*    observations.            */
      hmatch= _new_ hash(dataset:"prior",
                                    /* 8. Dataset to load to hash. It must */
                                    /*    fit into memory.                 */
                  hashexp:20,       /* 9. How many spaces for hash match */
                                    /*    keys i.e. 2^20.                */
                  multidata:'Y');   /* 11. Can more than 1 observation */
                                    /*     have the same hash key?     */
        dcl hiter hi;               /* 12. Create a hash iterator */
        hi= _new_ hiter('hmatch');  /* 13. Define hash to iterate over */

        hmatch.DefineKey("prior_account");  /* 14. Match key */
        hmatch.DefineData(ALL:"YES");       /* 15. Transfer all        */
                                            /*     variables from      */
                                            /*     hash dataset.       */
    hmatch.DefineDone();            /* 16. Done defining hash */
  end;

      do while (not lastds ) ;
            num_prior_trans = 0;    /* 17. Default the transaction count */
                                    /*     to zero.                      */
        set dat.example end=lastds ;
**      put '***** New Example *****' ;
**      put account= day= tran_id=;
        if hmatch.find(key:account)=0  then do;

                                    /* 18. Is there an account match? */
**           put 'found key match: ' prior_account= prior_day= ;
             if day - 30 <= prior_day < day then do;
                  num_prior_trans = num_prior_trans + 1;
**                put 'it is a match on day range so add 1: '
**                    num_prior_trans= ;
             end;

             hmatch.has_next(result:d);  /* 19. Check for another match */
                                         /*     by account. Mimics the  */
                                         /*     Cartesian join embedded */
                                         /*     within PROC SQL.        */
             do while (d  ne 0);
                  rc=hmatch.find_next();
**                put 'has another match: ' prior_account= prior_day= ;
                  if day - 30 <= prior_day < day then do;
                        num_prior_trans = num_prior_trans + 1;
**                      put 'it is a match on day range so add 1: '
**                          num_prior_trans= ;
                  end;

                  hmatch.has_next(result:d);
             end;
       end;

       keep tran_id day account num_prior_trans ;
      output;
    end;
**    %put "finished reading"  ;
```

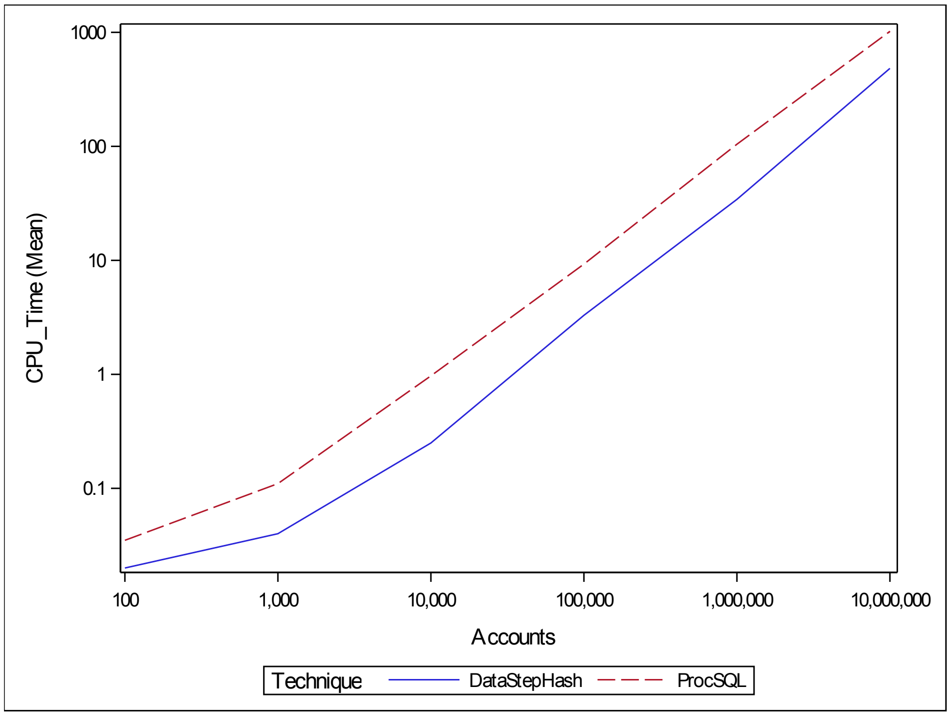
```
run;
```

Speed Benchmarks

Are any of the proposed solutions more efficient for processing? We ran benchmark speed tests on each of them using a range of data sizes. The tested values differed by the number of accounts from 100 to 10,000,000 in power of 10 increments. While testing the data step hashing method, we discovered that varying the value of hashexp such that we only have enough spaces for the current number of accounts was an important factor in processing speed.

Results show that the hash in the data step is faster for processing. The data step hash runs in less than half of the time of PROC SQL.

			Technique	
			DataStepHash	**ProcSQL**
Accounts				
100	**CPU_Time**	**Mean**	0.02	0.04
1,000	**CPU_Time**	**Mean**	0.04	0.11
10,000	**CPU_Time**	**Mean**	0.25	0.97
100,000	**CPU_Time**	**Mean**	3.30	9.27
1,000,000	**CPU_Time**	**Mean**	34.28	104.39
10,000,000	**CPU_Time**	**Mean**	482.96	1021.02



One drawback of the data step hash would be its use of memory. The default memsize on our system is 512 MB. I had to set it to 1 GB to run the 1,000,000 account version and to 8 GB to run the 10,000,000 account version. That might not be sustainable for another 10-fold increase in accounts.

Conclusion

We have shown how to use PROC SQL and a Data Step Hash Object to aggregate records contained within the same data set. In terms of code readability, using PROC SQL steps seems easier to read than a Data Step Hash. However, processing speed favors the Data Step Hash up to the point that the entire dataset would not fit into memory.