PharmaSUG 2022 - Paper AP-038

Fast hashes for complex joins

Bartosz Szymecki, Syneos Health

ABSTRACT

Even though SAS® hash objects have been around for a long time, their application within the industry is still rather limited. It is well known that they can successfully replace data step MERGE statements with improved efficiency, but what about more complex types of joins that statistical programmers can come across in their day-to-day work? PROC SQL would be a natural choice, especially for small datasets. However, as data volumes increase, as do the number of runs on that data, the advantage of using SAS hash objects over PROC SQL becomes clearly visible. If, in addition, a SAS hash approach is implemented within an easy-to-use macro, one may start to wonder why it shouldn't be used as the default for the majority of joins we do. This paper presents such macro in comparison to PROC SQL, using real-world examples that can arise whilst working with ADaM/SDTM standards.

INTRODUCTION

The main obstacle for statistical programmers to using hash objects is making the extra effort to learn the syntax and a few specific concepts that differ from what we are used to in base SAS. However, if we are able to apply them within a macro (%mergehash) and use a call as simple as in the example below (Table 1), then this issue is eliminated.

Example syntax - comparison	
Proc sql approach	**%mergehash approach**
`proc sql;`	`%mergehash(`
`  create table EX_SQL as`	`    inDat      = ADSL`
`    select a.*, b.EV_DT`	`  , keydat     = DIARY`
`    from ADSL a`	`  , outdat     = EX_HASH`
`      left join DIARY b on`	`  , join_type = left`
`        a.SUBJID=b.SUBJID`	`  , by_vars    = SUBJID`
`    order by a.SUBJID;`	`)`
`quit;`	

Table 1.

If DIARY dataset contains just SUBJID and EV_DT then both approaches above give the same results, i.e. EX_SQL and EX_HASH are identical. This simple example presents one-to-many type of join so it could also be easily performed with MERGE statement in the data step (although slower), but this paper focuses on more complicated operations such as:

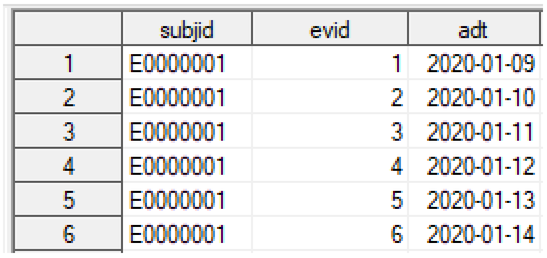
- Joins with custom conditions (e.g. a.VAR1<=b.VAR2<=a.VAR3)
- Many-to-many joins, and reshaping them to many-to-one, depending on needs
- Managing unexpected join results

The following sections will cover such scenarios, providing efficiency comparisons and detailed explanation on every parameter used in a given example. Later on, we will take a peek under the hood, providing clarification on the crucial parts of the macro. If you need a general overview of hash objects please see [1] in the references section. The book mentioned there is not only the ideal introduction to the topic, but also a tremendous source of knowledge, often exceeding the official documentation.

EXAMPLE 1: ASSIGN A VISIT WINDOW

Processing clinical data often involves dealing with dates. Data stored in BDS domains are usually associated with a single date, for example laboratory assessment performed within one day is represented by ADT variable. On the other hand, some occurrence data, like Adverse Events need two variables to store start and end date of an event, i.e. ASTDT and AENDT respectively. Quite often planned analysis requires implementing visit windows, that, similarly as occurrence data are associated with periods rather than single dates.

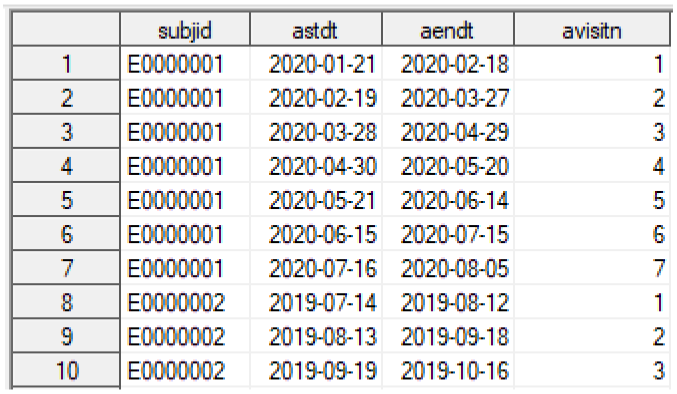
Let's think of a situation where we have a DIARY dataset with events reported in a patient diary, with the structure presented in Display 1.



	subjid	evid	adt
1	E0000001	1	2020-01-09
2	E0000001	2	2020-01-10
3	E0000001	3	2020-01-11
4	E0000001	4	2020-01-12
5	E0000001	5	2020-01-13
6	E0000001	6	2020-01-14

Display 1. DIARY dataset structure (first 3 variables)

And VISTW dataset with AVISITN information along ASTDT and AENDT, presented in Display 2.



	subjid	astdt	aendt	avisitn
1	E0000001	2020-01-21	2020-02-18	1
2	E0000001	2020-02-19	2020-03-27	2
3	E0000001	2020-03-28	2020-04-29	3
4	E0000001	2020-04-30	2020-05-20	4
5	E0000001	2020-05-21	2020-06-14	5
6	E0000001	2020-06-15	2020-07-15	6
7	E0000001	2020-07-16	2020-08-05	7
8	E0000002	2019-07-14	2019-08-12	1
9	E0000002	2019-08-13	2019-09-18	2
10	E0000002	2019-09-19	2019-10-16	3

Display 2. VISTW dataset structure

Our task is to assign a visit to every event. This can be achieved with the 2 following approaches (Table 2):

Example 1 – SAS code

Proc sql approach

```
proc sql;
  create table DIARYVISITS_SQL as
    select a.*, b.ASTDT, b.AENDT,
      b.AVISITN
    from DIARY a
      left join VISTW b
        on a.SUBJID=b.SUBJID and
           ASTDT <= ADT <= AENDT
    order by 1, 2, b.avisitn;
quit;
```

%mergehash approach

```
%mergehash(
    inDat          = DIARY
  , keydat         = VISTW
  , outdat         = DIARYVISITS_HASH
  , by_vars        = SUBJID
  , by_condition = ASTDT<=ADT<=AENDT
)
```

Table 2.

Time needed to complete the task (in seconds) is presented in Table 3. It has been calculated as an average out of 50 batch runs, following the approach presented in [2]. Four scenarios are covered, with a different number of observations in DIARY. There were 10 variables in DIARY and 4 variables in VISTW. Note that the 50 runs average approach is used in the subsequent examples as well.

Example 1 – efficiency comparison			
nobs in DIARY	**Time assessed**	**Proc sql approach**	**%mergehash approach**
200 000	Real time	1.37	0.53
	CPU time	1.52	0.36
500 000	Real time	8.85	1.65
	CPU time	6.68	1.51
1 000 000	Real time	42.55	7.10
	CPU time	13.92	2.06
2 000 000	Real time	100.87	21.88
	CPU time	25.83	5.28

Table 3.

If there's a need to perform inner join instead of left join this can be easily achieved by specifying "join_type = inner" in the %mergehash call. Note than when VISTW is created correctly it won't contain any overlapping periods, so there's a guarantee that maximum 1 observation from VISTW will be joined to a single record from DIARY. However, when we operate on data that hasn't been cleaned yet, there is a possibility VISTW contains some overlapping periods. In such case it's possible to use "multiwarn = Y" parameter in %mergehash call, which will result in a warning message (Output 1) if more than 1 record was joined from "&keyDat".

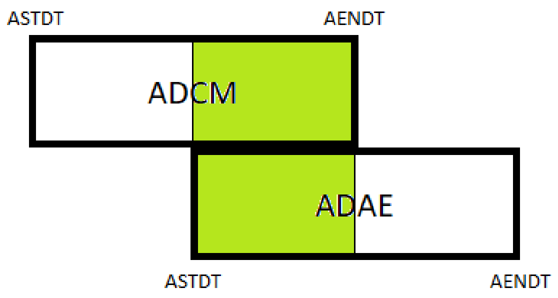
```
WARNING:: More than 1 obs available for joining. Specify
multitype=all/first/last depending on the needs. For example:
subjid=E0000001 astdt=2020-02-17 aendt=2020-03-25 avisitn=2
```

Output 1. Warning obtained from MULTIWARN=Y parameter

This is another advantage of %mergehash when comparing to proc sql approach – the check is done in the same data pass, hence it doesn't affect efficiency. The "multitype" parameter mentioned in the warning enables user to specify the way in which multiple joined records should be handled. This is discussed further in the next example.

EXAMPLE 2: JOIN OVERLAPPING PERIODS

Getting back to the occurrence data, where both start and end dates are available, let's imagine a situation in which a medical team is asking us to provide a list of all concomitant medications that were taken during specific Adverse events, separately for every patient. Such request requires joining overlapping periods rather than single days. We are interested in identifying intersections between AE and CM records for every patient. That is, CM records will be joined if they started on or before an AE ended, and if they ended on or after the AE started. Visually this can be imagined as 2 connected rectangles that can slide along longer side but can't be set apart (Display 3).



Display 3. overlapping periods for single records from ADCM/ADAE

In terms of ADAE and ADCM structure, they both contain SUBJID, ASTDT, AENDT. In addition, ADAE contains AESPID and 10 event related variables, and ADCM contains CMSPID and CMDECOD. ADAE is sorted by SUBJID and AESPID, while ADCM is sorted by SUBJID and CMSPID. We want to preserve this order for further processing.

The output dataset can be created in the following ways (Table 4):

Example 2 – SAS code

Proc sql approach

```
proc sql;
  create table ADAE_CM_SQL as
    select a.*, b.ASTDT as ASTDT_CM,
      b.AENDT as AENDT_CM, CMSPID,
      CMDECOD
    from ADAE a
      left join ADCM b
        on a.SUBJID=b.SUBJID and
          a.ASTDT<=b.AENDT and
          b.ASTDT<=a.AENDT
    order by a.SUBJID, a.AESPID,
      b.CMSPID;
quit;
```

%mergehash approach

```
%mergehash(
    indat         = ADAE
  , keydat        = ADCM (rename=
                     (ASTDT=ASTDT_CM
                      AENDT=AENDT_CM))
  , outdat        = ADAE_CM_HASH
  , by_vars       = SUBJID
  , by_condition  = ASTDT<=AENDT_CM and
                    ASTDT_CM<=AENDT
  , count_var     = Y
)
```

Table 4.

Time needed to complete the task (in seconds), depending on number of observations in both datasets, assuming that both ADAE and ADCM always contain 20 observations per subject (100 000 observations is equivalent to 5 000 subjects) - Table 5:

Example 2 – efficiency comparison				
nobs in ADAE	**nobs in ADCM**	**Time assessed**	**Proc sql approach**	**%mergehash approach**
100 000	100 000	Real time	10.42	1.37
		CPU time	4.01	0.69
200 000	200 000	Real time	22.19	5.54
		CPU time	7.74	1.46
500 000	500 000	Real time	62.70	22.11
		CPU time	19.13	4.88

Table 5.

If ADCM contained more variables, we could specify "join_vars = ASTDT_CM AENDT_CM CMSPID CMDECOD" as a parameter in %mergehash call, which would pick the ones of interest. Also, "count_var = Y" has been added as an extra parameter. It'll create __H_CNT variable in the output dataset, assigning a number for every observation joined from ADCM. The counter is reset for every record in ADAE. If no records are joined then __H_CNT=0 on such record, and if for example 2 records are joined then __H_CNT=1 on the first of them, and 2 on the next one. Creating such variable is not really needed in this example but it illustrates how easy it is to incorporate within the macro further functionalities which can be computed in a single data pass, hence further improving efficiency.

EXAMPLE 2B: MODIFIED REQUEST – PICK FIRST

Let us modify the discussed request so that only the first medication joined from ADCM should stay in the output dataset, for every ADAE record. One option for the continuation of the PROC SQL approach would be to utilize min() aggregate function along with built-in remerging functionality (see example code below). Another option would be to create the output dataset from the original example and add a data step with by statement along with “FIRST.” temporary variable that could be used to keep only the first record per single AE. The former may be more efficient since PROC SQL engine tries to optimize queries behind the scenes. Nevertheless, those 2 approaches will be significantly slower than %mergehash with “multitype = FIRST” parameter which will complete the same task faster than in the original request. This happens for 2 reasons – first of all, many output operations are omitted and second of all, hash object can stop iterating and move to the next observation once the first match is found. More details on this behavior are included in the next sections.

The codes used to complete the modified requests (Table 6):

Example 2B – SAS code

Proc sql approach

```
proc sql;
  create table ADAE_CM_FIRST_SQL as
    select a.*, b.ASTDT as ASTDT_CM,
      b.AENDT as AENDT_CM, CMSPID,
      CMDECOD
    from ADAE a
      left join ADCM b
        on a.SUBJID=b.SUBJID and
          a.ASTDT<=b.AENDT and
          b.ASTDT<=a.AENDT
    group by 1,2,3,4,5,6,7,8,9,
             10,11,12,13,14
    having min(CMSPID)=CMSPID
    order by a.SUBJID, a.AESPID,
      b.CMSPID;
quit;
```

%mergehash approach

```
%mergehash(
    indat          = ADAE
  , keydat         = ADCM (rename=
                      (ASTDT=ASTDT_CM
                       AENDT=AENDT_CM))
  , outdat         = ADAE_CM_FIRST_HASH
  , by_vars        = SUBJID
  , by_condition = ASTDT<=AENDT_CM and
                     ASTDT_CM<=AENDT
  , multitype      = FIRST
)
```

Table 6.

Time needed to complete the task (in seconds), depending on number of observations in both datasets, assuming that both ADAE and ADCM always contain 20 observations per subject (100 000 observations is equivalent to 5 000 subjects) - Table 7:

Example 2B – efficiency comparison				
nobs in ADAE	**nobs in ADCM**	**Time assessed**	**Proc sql approach**	**%mergehash approach**
100 000	100 000	Real time	7.47	0.20
		CPU time	6.08	0.19
200 000	200 000	Real time	23.72	0.37
		CPU time	12.08	0.36
500 000	500 000	Real time	80.89	3.08
		CPU time	31.74	1.14

Table 7.

Similarly big time savings would be obtained if the request involved picking last ADCM record per every AE instead of first. PROC SQL would be modified to pick max(CMSPID) instead of min(), and in %mergehash it would be sufficient to update “multitype = LAST”.

%MERGEHASH – OVERVIEW AND FURTHER FUNCTIONALITIES

The full list of parameters along with defaults is:

```
%mergehash(
          inDat                 =
        , keyDat                =
        , outDat                =
        , by_vars               =
        , by_condition          =
        , join_vars             = [ALL]
        , join_type             = LEFT
        , multitype             = ALL
        , multiwarn             = N
        , overwrite_vars        = N
        , overwrite_set_missing = N
        , count_var             = N
        )
```

There are few things worth mentioning, that haven't been covered in the previous examples:

- The way &by_vars and &by_condition work together can be understood as ([&by_vars. being equal]) AND (&by_condition.). It is possible to specify only one of the two parameters, but they can't be both blank.
- All variables specified in &by_condition need to appear in only one of the datasets. Rename dataset option can be used in &inDat and &keyDat if needed.
- The default of &join_vars is "[ALL]". Specific variables can be listed, separated by space. It's also possible to leave this parameter blank, and in such case the macro should be called with &join_type=inner, since this way a subset of &inDat can be created.
- "multiwarn=Y" works regardless of multitype. It prints only one warning with the first encountered issue, even if multiple present.
- "overwrite_vars=N" by default, which means that common variables won't be joined from &keyDat even when requested. This way no values will be overwritten. If, however, Y is specified, then the values from &keyDat will overwrite the common variables from &inDat.
- "overwrite_set_missing" can be used when "overwrite_vars=Y", in order to control what happens with common variables when no observation was joined from &keydat. If "overwrite_set_missing=N" then the original values from &inDat will be used, and if "overwrite_set_missing=Y" then those variables will be set to missing in such case.
- The macro can be successfully used for simple joins as well, usually resulting in substantial time savings

MACRO CODE – PEEK UNDER THE HOOD

In order to make the review easier, the full code is provided in the next section along with the line numbers. Lines 16-122 are responsible for basic data checks, pre-processing and handling some of the parameters, like &overwrite_vars or &join_vars.

The crucial part of the macro starts on line 124, where &outdat is created. &Indat is read with the set statement, that is, using the implicit data step loop, and an extra block is run within its first iteration (lines 130-148). Within this block the hash table h is created and instantiated with "dcl hash h()" statement. Two different methods for inserting items are used, depending on the presence of &by_vars:

- If &by_vars is specified then implicit insert operation is done with the DATASET argument tag, which essentially inserts the whole &keyDat into the hash table. &by_vars also become the key portion of the hash table since values of those variables will be later on searched by the internal hashing algorithm to match the PDV values coming from &inDat – this is performed by “h.find()” and “h.find_next()” methods later on (line 152). Those methods retrieve the data portion of a hash table into the PDV when the match is found, overwriting the existing values hence making the join possible.

- If &by_vars is not specified, then a different approach need to be taken, since now there are no specific values that could be looked up in a hash table. The key portion is created artificially with '_n_' variable since it's the most convenient approach – hash table can't exist without the key portion. Data portion of the hash table is read manually using set statement along with “h.add()” method (lines 139-143) . Since the hash table created in this way can't be used for searching the values, a hash iterator object is created on line 144 - “dcl hiter ih('h')”. This object will be used for iterating the hash table for every observation read from &inDat in order to check if &by_cond was met.

In both approaches, the MULTIDATA argument tag is used. It enables the hash table to contain more than 1 record with the same key values. Note that in the second approach it's also required because although _n_ was specified as a key variable, its value will always be 1 since the data portion of this hash table is read within the first iteration of the implicit data step loop. Both approaches also end with the obligatory “h.definedone()” method. Line 146 is necessary for the first approach – while DATASET argument tag is a convenient way to read the content for a hash table, it requires the presence of all variables in the PDV. Since the PDV structure is determined during the compilation phase, 'if 0' condition is used to fill this purpose.

Once the hash objects are set-up we can move to the actual processing. &join_vars are set to missing at the beginning of every implicit data step loop (line 149). This step is required, because as mentioned earlier, hash objects retrieval methods overwrite the existing PDV values. __H_CNT helper variable is reset to 0 as well. Now the actual joining can begin. In hash object world this is done by the retrieval operation methods. Depending on the value of &by_vars we use:

- “h.find()” and “h.find_next()” (line 152) when &by_vars are specified, meaning the hash table can be searched for the key variables. The “h.find()” method tries to find a record in the hash table with its key portion being equal to the values stored currently in the PDV. When the search is successful, the method returns 0 and what is more important, it retrieves the data portion of a hash table into the PDV.

- “ih.next()” (line 155) when no &by_vars are used. In this case we can't search the hash table for a specific value of key variables, so a different approach is required when we iterate through the whole hash table to see if non-key variables meet the &by_condition. In order to make this iteration possible we use hash iterator. The “ih.next()” method causes the hash iterator pointer to move to the next record in the hash table. At the beginning it's outside of the table, so the method points in to the first hash table record. Again, the important aspect of this method is that it retrieves the data portion of a hash table to the PDV. Only then we are able to determine whether &by_condition is met.

Depending on whether &by_condition is specified, “if (&by_condition.) then do;” (line 157/158) is added. Everything that's included between line 159 and 187 is executed only if the match is found. __H_CNT is increased, &multiwarn parameter is handled on lines 160-171. In the next block the &multitype parameter is handled. If FIRST is used, we want to output the observation only for the first match found, so we can leave the do block whenever __H_CNT is greater than 1, or even if it is 1 and we don't need to look for the next occurrence (“&multiwarn ne Y”). Note that the do block that we are leaving (line 176) is the one starting on line 152/155, regardless of the presence of the block starting on line 158. If “multitype=ALL”, the observation is simply being output. When “multitype=LAST” the hash table value from every match needs to be temporarily retrieved into __H_TMP variables on line 184. This is done so that __H_TMP vars contain the data portion of the hash table coming from the last matched record.

When the loop ends on line 188, all records in the hash table have already been searched. Now we need to handle the cases where no matches were found and left join was specified, and we also need to finally output the proper values for “multitype=LAST”, because this was not done before. Hence, on line 191 all &join_vars are reset to missing if no match was found for a left join. Note that this step is required, because &join_vars can be non-missing on this stage, for example if &by_vars was specified, and the key was found in a hash table, but &by_condition was not met. Finally, if the match was found for “multitype=

LAST", the values stored temporarily in __H_TMP are transferred to &join_vars, and only then the observation is output (line 204).

%MERGEHASH - MACRO CODE

```
1   %macro mergehash(
2            inDat                 =
3          , keyDat                =
4          , outDat                =
5          , by_vars               =
6          , by_condition          =
7          , join_vars             = [ALL]
8          , join_type             = LEFT
9          , multitype             = ALL
10         , multiwarn             = N
11         , overwrite_vars        = N
12         , overwrite_set_missing = N
13         , count_var             = N
14         );
15
16      %local keyvars datavars join_vars2 join_vars_ov rename
17        join_vars_rename;
18
19      %if &join_vars.= and %upcase(&join_type.)=LEFT %then %put
20        %str(WAR)NING:: no variables specified for a left join;
21      %if &join_vars.= and &by_vars.= %then %do;
22          %put %str(ER)ROR:: at least one of JOIN_VARS and KEY_VARS params
23            needs to be specified.;
24          %goto endm;
25      %end;
26      %if %upcase(&overwrite_vars.)=N and %upcase(&overwrite_set_missing.)
27        =Y %then %put %str(WAR)NING:: OVERWRITE_SET_MISSING matters only
28        when OVERWRITE_VARS=Y;
29      %if %index(&indat.,%str(%()) %then %do;
30          data __indatopts;
31              set &indat.;
32          run;
33          %let indat=__indatopts;
34      %end;
35      %if %index(&keydat.,%str(%()) %then %do;
36          data __keydatopts;
37              set &keydat.;
38          run;
39          %let keydat=__keydatopts;
40      %end;
41
42      %if %index(&indat.,.) %then %do;
43          %let indat_lib=%scan(&indat.,1,.);
44          %let indat_dat=%scan(&indat.,2,.);
45      %end;
46      %else %do;
47          %let indat_dat=&indat.;
48          %let indat_lib=%sysfunc(getoption(user));
49          %if &indat_lib.= %then %let indat_lib=work;;
50      %end;
51
52      %if %index(&keydat.,.) %then %do;
```

```
        %let keydat_lib=%scan(&keydat.,1,.);
        %let keydat_dat=%scan(&keydat.,2,.);
    %end;
    %else %do;
        %let keydat_dat=&keydat.;
        %let keydat_lib=%sysfunc(getoption(user));
        %if &keydat_lib.= %then %let keydat_lib=work;;
    %end;

    %let keyvars="%sysfunc(prxchange(s/\s+/%bquote(",")/,-1,&by_vars.))";
    %if %upcase(&join_vars.)=[ALL] %then %do;
        proc sql noprint;
          select upcase(name) into :join_vars separated by ' '
          from dictionary.columns
          where memname=upcase("&keyDat_dat.") and
            libname=upcase("&keyDat_lib.") and
            upcase(name) not in (%upcase(&keyvars.));
        quit;
    %end;
    %let datavars="%sysfunc(prxchange(s/\s+/%bquote(",")/,
      -1,&join_vars.))";
    %if %upcase(&overwrite_vars.)=N and &join_vars. ne  %then %do;
        proc sql noprint;
          select upcase(b.name) into :join_vars_ov separated by ' '
          from (select name from dictionary.columns where (memname =
             upcase("&inDat_dat.") and libname=upcase("&inDat_lib.") and
             upcase(name) not in (%upcase(&keyvars.)) and upcase(name) in
             (%upcase(&datavars.)))) a
            right join (select name from dictionary.columns where
             (memname=upcase("&keyDat_dat.") and libname=upcase(
             "&keyDat_lib.") and upcase(name) not in (%upcase(&keyvars.))
             and upcase(name) in (%upcase(&datavars.)))) b
            on upcase(a.name)=upcase(b.name)
          where a.name='';
        quit;

        %if %length(&join_vars_ov.) ne %length(&join_vars.) %then %do;
            %let join_vars=&join_vars_ov.;
            %let datavars="%sysfunc(prxchange(s/\s+/%bquote(",")/,
              -1,&join_vars.))";
            %put NOTE:: &=OVERWRITE_VARS. so JOIN_VARS updated to
              &join_vars_ov.;
        %end;
    %end;
    %else %if %upcase(&overwrite_vars.)=Y and &join_vars. ne %then %do;
        proc sql noprint;
          select upcase(b.name) into :join_vars_rename separated by ' '
          from (select name from dictionary.columns where
              (memname=upcase("&inDat_dat.") and libname=upcase(
              "&inDat_lib.") and upcase(name) not in (%upcase(&keyvars.))
              and upcase(name) in (%upcase(&datavars.)))) a
            inner join (select name from dictionary.columns where
             (memname= upcase("&keyDat_dat.") and libname=upcase(
             "&keyDat_lib.") and upcase(name) not in (%upcase(&keyvars.))
             and upcase(name) in (%upcase(&datavars.)))) b
            on upcase(a.name)=upcase(b.name);
        quit;
    %end;

    %let join_vars2=%sysfunc(prxchange(s/\s+/%str(, )/,-1,&join_vars.));

    %if &join_vars_rename. ne and %upcase(&overwrite_set_missing.)=N
    %then %do;
        %let rename=%str(rename=%( );
        %do i=1 %to %sysfunc(countw(&join_vars_rename.));
          %let rename=&rename. %scan(&join_vars_rename.,&i.)=__h_rnm_
              %scan(&join_vars_rename.,&i.);
        %end;
        %let rename=&rename. %str( %) );
    %end;

    data &outdat.;
        set &indat. %if &rename. ne %then (&rename.); %else %if
          %upcase(&overwrite_set_missing.)=Y and &join_vars_rename. ne
          %then (drop=&join_vars_rename.);;
        retain warnfn 0;
        rc=.;
        if _n_=1 then do;
            dcl hash h(%if &by_vars. ne %then dataset: "&keydat.
              (keep=&by_vars. &join_vars.)",; multidata:'Y');
            %if &by_vars. ne %then h.definekey(%unquote(&keyvars.));
            %else h.definekey('_n_');;
            %if &join_vars. ne %then h.definedata(%unquote(&datavars.));;
            h.definedone();

            %if &by_vars.= %then %do;
                do until(eof);
                    set &keydat.(keep=&join_vars. %if &join_vars.= %then
                      drop=_all_;) end=eof;
                    rc=h.add();
                end;
                dcl hiter ih('h');
            %end;
            %else %if &join_vars. ne %then if 0 then set &keydat.(keep=
              &join_vars.);;
        end;
        %if &join_vars. ne %then call missing(&join_vars2.);;
        __h_cnt=0;
        %if &by_vars. ne %then %do;
            if h.find()=0 then do until(h.find_next() ne 0);
        %end;
        %else %do;
            do while(ih.next()=0);
        %end;
            %if %length(&by_condition.)>0 %then if (&by_condition.)
              then do; ;
                __h_cnt=__h_cnt+1;
                %if %upcase(&multiwarn.)=Y %then %do;
                    if warnfn=0 and __h_cnt>1 then do;
                        put 'WAR' 'NING:: More than 1 obs available for
                          joining. Specify multitype=all/first/last
                          depending on the needs. For example: '
                            %do i=1 %to %sysfunc(countw(&by_vars.
                              &join_vars.));
```

```
                                %scan(&by_vars. &join_vars.,&i.)%str(= )
                            %end; ;
                        warnfn=1;
                    end;
                %end;

                %if %upcase(&multitype.)=FIRST %then %do;
                    if __h_cnt=1 then output;
                    if upcase("&multiwarn.") ne 'Y' or __h_cnt>1 then
                      leave;
                %end;
                %else %if %upcase(&multitype.)=ALL %then %do;
                    output;
                %end;
                %else %if %upcase(&multitype.)=LAST and &join_vars. ne
                  %then %do;
                    %do i=1 %to %sysfunc(countw(&join_vars.));
                        __h_tmp&i.=%scan(&join_vars.,&i.);
                    %end;
                %end;
                %if %length(&by_condition.)>0 %then end; ;
            end;

        if (upcase("&join_type.")="LEFT" and __h_cnt=0) then do;
            %if &join_vars. ne %then call missing(&join_vars2.);;
            %if &rename. ne %then %do i=1 %to
              %sysfunc(countw(&join_vars_rename.));
                %scan(&join_vars_rename.,&i.)=
                  __h_rnm_%scan(&join_vars_rename.,&i.);
            %end;
            output;
        end;
        else if (upcase("&multitype.")="LAST" and __h_cnt>=1) then do;
           %if %upcase(&multitype.)=LAST and &join_vars. ne %then %do i=1
             %to %sysfunc(countw(&join_vars.));
               %scan(&join_vars.,&i.)=__h_tmp&i.;
           %end;
           output;
        end;
      drop rc %if %upcase(&multitype.)=LAST and &join_vars. ne %then
        __h_tmp:; %if &rename. ne %then __h_rnm_:; %if %upcase(
        &count_var.) ne Y %then __h_cnt; warnfn;
    run;
    %endm:
%mend;
```

LIMITATIONS

Hash objects have certain limitations. The biggest one is the fact that the &keyDat dataset needs to be loaded into memory, meaning the volume of available operating memory is crucial. If it is less than the &keyDat size, then loading the dataset at once won't be possible. There are still certain ways to accommodate this issue. The first thing in such case, is making sure that only required variables are used for joining, both in &keyDat and &inDat. Another common technique is to utilize group-by processing, where the hash object size is determined by the largest by-group in the key dataset. Such hash object is cleared after a given by-group is processed, releasing the memory for the next group. An example implementation and general overview of options for better memory management can be found in [1]. One can also discover that sometimes using minimal number of variables for complex joins and remerging the rest of the variables in the next step (simple join), will result in a better overall performance. Nevertheless, using %mergehash in all such partial join steps, will often yield better results than other approach.

CONCLUSION

The presented macro is a universal tool that can be used for different types of joins, covering both simple and complex ones, providing a tempting alternative for standard approaches due to better performance and syntax simplicity. The final point is that the presented macro can be further improved, for example, the dictionary approach (lines 66, 77, 81, 100, 104) can be replaced by faster SAS File I/O functions (OPEN etc.) or PROC CONTENTS with OUTPUT statement. We could also work around the memory restrictions, by enabling group-by processing in an extra parameter. Another useful implementation would be fuzzy merging – matching the closest observation. Numerous other parameters could be added to cover any additional operations, depending on needs.

REFERENCES

1. Dorfman, Paul and Don Henderson. 2018. *Data Management Solutions Using SAS® Hash Table Operations: A Business Intelligence Case Study*. Cary, NC: SAS Institute Inc.

2. Pisegna, Tony. 2010. “A Benchmarking Technique to Identify Best Practices in SAS.” *Proceedings of the North East SAS Users Group Conference*, North Wales, PA. Available at https://www.lexjansen.com/nesug/nesug10/ma/ma09.pdf.

CONTACT INFORMATION

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

Bartosz Szymecki
Syneos Health
Warsaw, Poland
bartosz.szymecki@syneoshealth.com