SESUG Paper BB-133-2017

Automatic Verification of Combined Datasets

Joe Matise, NORC at the University of Chicago; Marcus Maher, Ipsos Public Affairs

ABSTRACT

One common task for a SAS® programmer is to combine datasets from different sources, perhaps from multiple rounds of a study or from different panels in a survey. When combining these datasets, it is important to ensure that the final dataset(s) have the correct combination of rows, both that they have all of the expected rows from each source dataset and that the rows from the source datasets are not in conflict.

We present a solution for automatically verifying a combined dataset. We verify that every row in the source datasets has a matching row in the combined dataset, that every row in the combined dataset comes from a unique source, that the ID variable in the final dataset is unique, and provide several examples of possible logic checks that can be added to the application depending on needs, such as verifying that the study round matches the dataset source.

The intended audience for this presentation is an intermediate programmer. We use hash tables in this application, but present them in sufficient detail for a programmer with no background in hash tables to understand their purpose and use.

INTRODUCTION

When developing applications to manage datasets for projects with multiple rounds or waves, one common task that is necessary to perform is to append the current period's dataset with the dataset containing past periods' data. While the combination itself is typically a trivial task thanks to the numerous methods SAS® provides to accomplish this, verification that the process succeeded can be more challenging to accomplish in an efficient manner, particularly if the datasets are large and/or numerous.

In order to simplify this task, and ensure it is accomplished in a timely fashion, we will combine data-driven techniques with hash tables.

A few key prerequisites apply to our particular approach. First, the datasets must have a unique primary key; our presented solution assumes that key is a single variable, but it can easily be adapted to multiple variable keys. Second, the datasets must be small enough that their primary key(s) can fit into memory (in the hash tables). Finally, the way we present this approach assumes fairly simple combine options (either "stacked" or "merged"); datasets with more complex logic to their combination (including filtering some records from the combination, for example) will require more complex logic in the verification application.

Users of this application should be generally familiar with hash tables, although we do not use particularly complex hash table operations beyond adding and removing hash table elements. If the reader is not familiar with hash tables, the authors have added some recommended reading at the end of this paper.

DEFINING THE EXAMPLE DATASETS

We will work with three combined datasets in this paper that will serve as examples: `PRODUCT`, `STORE`, and `SALE`. `PRODUCT` and `STORE` are dimension tables, describing the stores in our fictitious study and the products those stores sell. `SALE` is a fact table, containing records of sales of products at stores. These three tables are updated monthly, with `SALE` having new records added to it, and `PRODUCT` and `STORE` may have old records updated or new records added.

OVERVIEW: WHY VERIFY?

The ideas behind this paper originated as the answer to a from business clients, while working with a process that appended current period data to past periods' datasets:

> *How do we know this combination step worked correctly?*

Now, this process wasn't particularly complicated; it was pretty close to this:

```
data combined.mydata;
  set prevcomb.mydata thisper.mydata;
run;
```

So when the question was asked, the initial suggestion was to simply verify the counts. After all, we generally trust SAS to do just _work_ when doing simple things like this; while it's common to check that the result has the correct number of records, verification of a simple combination beyond that seems unnecessary.

However, the business clients weren't satisfied with counts; and while plenty of other checks exist to verify the integrity of the data, they continued to ask specifically after the records themselves. Eventually, after some meetings to discuss the issue, their particular rationale for concern came out:

> *How do we know that the correct records came from the correct datasets?*

In this particular case, the datasets have a period variable for most datasets, which should be equal to the current period for the current period dataset, and to a value less than the current period for the past periods dataset. But beyond simply checking this variable, the clients were particularly concerned as well that the process that created the datasets was correctly set up – i.e., that it had the correct source libraries mapped. Hence the desire for a separate checking process, that produced output that the clients could look at to understand that the process was indeed successful.

REQUIREMENTS

The requirements for this process might be summarized as follows:

- Data-driven process that automatically runs for each source dataset
- For each table:
  - Verify that the destination dataset contains all of the rows from the two source datasets
  - Verify that all rows from the source datasets exist in the destination dataset
  - Verify that the Period variable values are appropriate for the particular source dataset
- Produce a report that indicates any errors found
- Produce datasets for each table that contain any records that fail any of the verification steps

DATA-DRIVEN PROCESS

One important aspect of this is that it is data driven: it should not require a separate program for each different application we might want to use it with. Instead, the process should be run entirely from the data provided as input.

To drive this process, there are two reasonable options. First, one could simply run the process on every dataset present in the destination library; that is appealing when all datasets are of similar nature (i.e., all have unique identifiers that are programmatically identifiable, all have similar period variables, etc.), because it is something that simply works without any additional information. The macros that will be used can simply be driven from a dictionary.tables query.

Second, if the above assumptions do not hold, this process could be driven from a master table list. The table list would be a table (database, excel, etc.) that contained a row for each table to process, and necessary information on it such as the name of the table ID, whether it has a period variable, and what that period identifier is. In many cases, this table may already exist; in the original case here for example, a master table list already existed with all of this information on it which existed for among other reasons to drive the original combine process.

For the example we explore in this paper, this might look like so:

```
data master_table_list;
input table $ table_id $ hasQuarter;
datalines;
product prodID 0
store storeID 0
sales saleID 1
;;;;
run;
```

If the period variable were to vary on the different datasets, then it might also be a column on this table. In this particular process, the period variable is `quarter` on all tables that have it, so instead we have a binary variable that defines whether it exists or not.

We then create the calls to the macro as follows:

```
proc  sql;
  select
      cats('%verifyDS(table=',table,
                   ',idvar=',table_id,
            case when hasQuarter = 1
            then    ',periodVar=quarter,period=5, unique=Y'
            else    ' '
            end,
                   ',finalLib=final, ',
                   'baseLib=base, ',
                   'appendLib=append);')
      into :verifyList separated by ' '
  from master_table_list
  ;
quit;

&verifyList.;
```

And we run it by submitting the macro variable itself to the system.

Of course, this macro still needs to be written…

HASHING THINGS OUT

The general approach for this process will be to load the IDs for the base and append datasets into hash tables, then cycle through the final combined dataset using the data step. Each final record will be removed from the hash tables where found, letting us identify which table it originates from (or if it originates from both, which may be an error). By the end of the data step, the expectation is that both hash tables should be empty – if they are not, then those records were not loaded into the final dataset (which should be reported as an error).

While cycling through the data, we will test for other errors (such as being in both tables when they shouldn't, having an incorrect period variable, and any other testing that is needed for the particular project).

Below, we step through the code of the core macro itself. We will present the code inside the macro, to make clear the context for the various parameters, and then present the macro statement at the end.

We start then with the initial code defining the output and input datasets:

```
data err_&table.(keep=&idvar. error_type &periodVar. periodVal);
  set &finalLib..&table. end=eof;
  length error_type $128;
  by &idvar;
```

The output dataset will contain any ID that has an error, and an error_type value. We set up the length of the error_type variable, and ask for the dataset to be sorted by the id variable; this enables duplicate checks, as follows in the first error identification:

```
  if not (first.&idvar) then do;
    error_type="Duplicate ID in Final";
      output err_&table.;
      if not (eof) then return;
  end;
```

We then set up the hash tables:

```
if 0 then set &baseLib..&table. &appendLib..&table.;

if _n_=1 then do;
    declare hash h_base(dataset:"&baselib..&table.",multidata:"yes");
    h_base.defineKey("&idvar");
    h_base.defineData("&idvar");
    h_base.defineDone();

    declare hash h_append(dataset:"&appendLib..&table.",multidata:"yes");
    h_append.defineKey("&idvar");
    h_append.defineData("&idvar");
    h_append.defineDone();
end;
```

The multidata=yes option is in case the source datasets have duplicates in them (as we'd like to know that). Otherwise, only the id variable is needed from these datasets.

Next, we search for the record in each hash table, test for duplicates, and remove the record(s) from the hash table(s) if they are found. Note that we only perform the code in this macro on the first record for each ID value; if there is a mistaken duplicate, we will report that out but do not want to repeat these tests for each duplicate record.

```
if first.&idvar then do;

      rc_base = h_base.find();
      rc_append = h_append.find();

      rc_dup_base = h_base.has_next(result:has_dup_base);
      rc_dup_append = h_append.has_next(result:has_dup_append);

      if rc_base=0 then h_base.remove();
      if rc_append=0 then h_append.remove();
```

We can then output errors for duplicates found; the parameter &unique will tell us whether it is permitted or not to have the record found in both base and append, or if it should only appear in one or the other.

```
      if has_dup_base or has_dup_append then do;
          error_type="Duplicate ID in "
                      ||ifc(has_dup_base,"Base","Append");
        output err_&table.;
      end;


      %if &unique. eq Y or &unique. eq 1 %then %do;
        if (rc_base=0 and rc_append=0)  then do;
          error_type="Found in both Base And append";
            output err_&table.;
        end;
      %end;

      if (rc_base ne 0 and rc_append ne 0) then do;
          error_type="Found in neither Base Nor Append";
          output err_&table.;
      end;
```

Of course, it is also an error if the record is in combined but in neither base nor append.

Then we follow up with the checks for valid period values. This is only performed if the period variable is specified in the parameter. If so, then we test to see if the period value for the base dataset is greater than the current period (&period), which would indicate a row was added in the future; we also can check the append dataset to make sure it only contains current period records, if appropriate, as specified by &checkappendperiod.

```
    %if not %sysevalf(%superq(periodVar)=,boolean) %then %do;
          if rc_base=0 then do;
              if &periodvar. ge &period. then do;
                    error_type="Failed period Check - Base";
                    output err_&table.;
              end;
          end;

          if rc_append=0 then do;
              if &periodVar. ne &period. then do;
                error_type="Failed period Check - Append";
                output err_&table.;
              end;
          end;
    %end;

    end;
```

The final end here ends the if first.&idvar from above, as we're now finished with our checks. If any additional logical checks were needed (checking the validity of the ID variable, for example), they would go here prior to the end.

Finally, we have the terminating section, which outputs the hopefully empty hash tables, and does some bookkeeping.

```
    if eof then do;
        h_base.output(dataset:"&table._base_only");
        h_append.output(Dataset:"&table._append_only");
    end;

    call missing(of _all_);
run;
```

That ends the macro. Here follows the macro statement, including parameters:

```
%macro verifyDS(
      table= ,
      idvar= ,
      periodVar ,
      period= ,
      finalLib= ,
      baseLib= ,
      appendLib= ,
      unique=N
);
```

&table defines the name of the table (in all three libraries), &idvar is the name of the ID variable, &periodVar and &period are used in checking the period, the three library names are specified, and &unique indicates whether it is an error if the row is in both base and append tables.

REPORTING OUR RESULTS

After iterating through the datasets, the results must be reported. Several different reports will be generated to accomplish this.

First, a simple "count" report can be generated. This report does not actually use the results of the macro, but instead simply uses the counts from dictionary.tables (assuming the datasets are not produced in a way that would cause this value to be errant). For tables with the hasQuarter=1, indicating that rows will be unique and thus an expected number can be calculated, it calculates the expected number of rows from (base+append) and compares to (final).

```
proc sql;
  select _base.memname label="SU_ID",
            nobs_base label="Number of Observations in Base",
            nobs_append label="Number of Observations in Append",
            case when M.hasQuarter eq 1 then nobs_base + nobs_append
            else . end
            as nobs_expected label="Expected Observations in Final",
                  nobs_final label="Number of Observations in Final",
            case when hasQuarter=1 then
            (calculated nobs_expected - nobs_final)/nobs_final
             else . end
            as nobs_failure label="Failure (%)" format=percent8.2
      from (
      select memname, nlobs as nobs_base
        from dictionary.tables T
        where libname="BASE"
  ) _base
      join
  (   select memname, nlobs as nobs_append
        from dictionary.tables T
      where libname="APPEND"
  ) _append
      on _base.memname = _append.memname
      join
  (
      select memname, nlobs as nobs_final
        from dictionary.tables T
        where libname="FINAL"
  ) _final
  on _base.memname=_final.memname
  inner join (
            select table, hasQuarter
            from  MASTER_TABLE_LIST
      ) M
      on _base.memname=upcase(M.table)
  ;
quit;
```

Then, we accumulate all of the error datasets produced, and produce a PROC FREQ to the report as well.

```
data all_Errs;
  length _ds table_name $32;
  set err_: indsname=_Ds;
  table_name = scan(_ds,3,'._');
run;

proc freq data=all_errs;
  tables table_name*quarter*error_type/list nocum nopercent missing;
run;
```

Finally, we gather the records that only appeared in one or the other original dataset and output them here.

```
proc sql noprint;
   select cats(table,"&quarter.Append_only")
            into :appendlist separated by ' '
      from MASTER_TABLE_LIST
      ;
      select cats(table,"&quarter.Base_only
            into :baselist separated by ' '
      from MASTER_TABLE_LIST
      ;

quit;

data all_Append;
  set &appendlist.;
run;

data all_Base;
  set &baselist.;
run;


proc print data=all_append;
run;

proc print data=all_base;
run;
```

That concludes the report. If the datasets being verified are very large, a reasonable maximum OBS to print should be set, or this could get tediously long.

CONCLUSION

Here we present a simple and powerful method for verifying datasets that have been assembled together. The presented application should be seen as a template, and can be easily expanded to include more specific logic or project-based details; for example, one additional method used in the original project was to verify the ID variable was properly constructed from its parts.

We hope this is a useful tool for your QC toolbox, and even if this specific problem is not one you face, hopefully the general approach here will be useful for sparking other ideas to enhance your quality checks, and help you assure your internal or external clients that your processes ran successfully and performed their tasks correctly.

REFERENCES

The authors include references to several papers on hash tables here for reference, both to assist those who are unfamiliar with hash tables as a concept, and refresh the memories of those who, like the authors, do not have every detail memorized to their satisfaction.

Dorfman, Paul. 2009. "The SAS Hash Object In Action." *NESUG 2009 Proceedings*, Burlington, VT. Available at http://www.lexjansen.com/nesug/nesug09/hw/HW04.pdf .

Dorfman, Paul, and Koen Vyverman. 2005. "Data Step Hash Objects as Programming Tools". *SUGI 30 Proceedings*, Philadelphia, PA: SAS Institute.
Available at http://www2.sas.com/proceedings/sugi30/236-30.pdf .

ACKNOWLEDGMENTS

J.M. would like to acknowledge the particular input of Caitlin Finan, Christopher Ward, and many others without whom this idea would not have come to be.

CONTACT INFORMATION

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

Joe Matise
NORC at the University of Chicago
Matise.Joe@gmail.com