Paper CT13

Programming Solutions when Developing a Database Compare Macro

Michael S Rimler, FMD K&L Inc., Fort Washington, Pennsylvania, US

ABSTRACT

This paper discusses selected coding techniques implemented during the development of a database compare utility macro. Having the ability to efficiently assess how a database has changed from a previous transfer (or previous execution of database programming) can significantly reduce the amount of resources needed to maintain the programming infrastructure when the new data is put into production. Implementation of the desired macro functionality poses several challenges to ensuring that the utility is easy to use, easy to adopt, and difficult to break by the user. These challenges are outlined, along with proposed solutions for the macro developer. When illustrative, sample code is provided as reference.

MOTIVATION

Throughout the course of an ongoing study, programming groups may receive regular data transfers of their input data (e.g., raw data, SDTM data, or ADaM data). Often, the updated data is received when much of the programming infrastructure for the study is already in place. Ideally, the programming group would install the updated database and run all developed programs without error. However, successful execution relies on a stable design of the input database and, in many cases, near-clean data (i.e., no data issues). Early in the study, new data will be added on a regular basis (new subjects, new assessments, etc.). As the study nears completion, perhaps between a soft-lock and hard-lock, minimal changes to the database will be expected between transfers.

A robust database compare utility macro can improve efficiency of programming groups by quickly identifying changes in database structure from one transfer to another, highlighting where programming modifications are required to account for these database design changes. In addition, when expected changes are small (e.g., near database lock), the utility macro will isolate these few differences, allowing the programming group to assess where, if any, changes to programs are needed. The group can also compare its output database from the new transfer, relative to the previous version, verifying that only expected changes have occurred to the mapped database following a rerun of all programming.

To achieve this objective, a utility macro can be developed to compare two SAS ® databases beyond a simple PROC COMPARE. The generic macro design discussed in this paper is intended to be easy to execute and difficult to break, generating meaningful reports of database discrepancies. Reports include lists of datasets in only one database, new or removed variables, changes in metadata, records that are new (or removed), and discrepant data values for records that are common to both databases. Additional desired features of the macro include:

- Validation of macro parameters specified by the user to ensure that the macro will function properly
- Allowing simple specification of the datasets to be included in the comparison
- Executing an endogenous search for a unique sort key on which to compare two datasets

Development of a macro with these characteristics poses several challenges, which are discussed throughout this paper.

MAIN DESIGN

The main objective of the database compare macro is to report differences of similarly named datasets from two separate databases. Although the analyses proposed are akin to those performed by PROC COMPARE, the functionality presented is designed to be more robust and informative for the user. Solely leveraging the SAS compare procedure may produce limited results and generate uninformative comparisons without a more manual engagement of the process.

The parameters for the macro (as defined in the **%macro** statement) are presented in **Table 1**. This macro has a relatively simple call, with only 2 of the 9 parameters required for execution, namely the locations of the two databases. If the remaining parameters are unspecified, the macro will simply compare all datasets and produce all reports. In addition, the requirements for the parameters are intended to be easy to understand by the user. Ultimately, the macro design discussed is intended to be easy to use, difficult to break, and considered easy to adopt.

The global locations of the two databases to be compared are specified in **base_path** (BASE) and **comp_path** (COMP). Within the databases to be compared, the list of datasets to be compared are specified as a space-delimited list in **dset_list**. If this parameter is set to **_all_**, then the databases are fully compared. In this case, the user may selectively exclude datasets via a space-delimited list specified in **excl_dset**.

A sort key for each domain (to facilitate comparisons) is derived automatically within the macro. The user may globally prioritize variables in the search algorithm using a space-delimited list in **sortvars2incl**. The user may also globally deprioritize variables using a space-delimited list in **sortvars2excl**. As an added feature, if a variable begins or ends with an asterisk (*), it is treated as a wildcard character. Hence, if the user specifies **sortvars2excl=*seq**, then any variable names containing the string ‘seq’ will be deprioritized and relegated to the end of the search algorithm.

Table 1: Sample macro parameter definition of the database compare macro

```
%db_compare(base_path     =       /* Full path of base data for comparison           */
           ,comp_path     =       /* Full path of compare data for comparison        */
           ,dset_list     = _all_ /* Dataset list (if =_all_, compare full library)  */
           ,excl_dset     =       /* Datasets to be excluded, only works with _all_  */
           ,sortvars2incl =       /* Variable list to consider first                 */
           ,sortvars2excl =       /* Variable list to consider last                  */
           ,permdsetyn    = N     /* if = Y, then output permanent datasets          */
           ,perm_path     =       /* Full path of permanent output data              */
           ,majupdyn      =       /* Set =Y if major update, suppresses reports >=6  */
            );
```

If the user desires permanent datasets to be created, then **permdsetyn** should be set to **Y** and the (global) destination path (**perm_path**) must be specified. The user may desire this option if executing in batch mode or saving results for another individual to review.

Finally, by setting **majupdyn** to **Y**, the user may suppress observation level comparisons such as records contained in only one source dataset or value-level discrepancies for comparable observations. This feature may be of value, for example, when data cuts occur on 01-Jul-2017 and 01-Oct-2017, but database lock is planned for 31-Dec-2018. In this situation, the database may be expected to contain significant differences between cuts, and it may be sufficient to simply review differences at the dataset and variable level (e.g., common datasets, metadata, and common variables).

CHALLENGES

The purpose of this paper is to present several challenges facing a developer of a database compare macro, as well as proposed programming solutions. These challenges include:

1. Macro parameter validation
2. Dataset specification
3. Robust dataset comparisons – deriving a unique sort key
4. User control of sort key search algorithm
5. Addressing domains with fully replicated records
6. The output generated by the macro for the user

Regardless of programming language, any macro or subroutine grants the user control over processing through the parameters defined in the macro call. The developer has the ability, and arguably the responsibility, to balance control to the user (more flexibility) and rigid processing (less flexibility) through these parameters. Typically, more flexibility requires increased complexity of the call, either through additional parameters or increased syntax requirements of the call. In either case, increased complexity requires greater familiarity with the macro, possibly reducing the likelihood of adoption by new users. The solution presented here implements code which verifies the suitability of macro parameter values passed through a simple call with a limited number of parameters. In the case that the user specifies something incorrectly, the macro will cease processing and provide informative feedback to the user via the log, advising how to correct the issue.

Fundamental to the database compare utility macro is identifying which datasets are to be compared, and from which source databases. A simple solution would require explicit user specification of datasets and their location, however this can be onerous if the database contains a large number of datasets. Therefore, the solution presented here uses a combination of 4 macro parameters which identify the datasets to be compared. Two of these parameters specify the locations of the two databases (**base_path** and **comp_path**) and are required. If no other information is provided, the complete database of each location will be compared. The other two parameters are optional and provide added control over the datasets to compare. Datasets can be excluded through **excl_dset** or explicitly included through **dset_list**. Each dataset identified by the macro is then indexed to facilitate looped processing over each dataset and tracking of dataset properties useful in the analysis.

Simply identifying the datasets to be compared is not sufficient for successful processing. It is also necessary to construct a method for comparison. In many cases, it is not feasible to simply PROC COMPARE two datasets and interpret the results. For example, if a dataset with 22,000 records is updated so that it now contains 22,001 records, where the 100th record is the ‘new’ record, the PROC COMPARE will be meaningless, even if the datasets are pre-sorted similarly. The solution is to search for a sort key that is unique and common between the two datasets being compared, sort each dataset, and then compare.

Although the search for a unique sort key can be completely data driven, with no additional input from the user, there is no guarantee that the result will be meaningful when interpreting the results. For example, if the SDTM domain LB is being compared, we know that USUBJID and LBSEQ will generate a unique sort key in most cases (at least when compliant with the SDTM IG). However, from one database version to another, LBSEQ may be redefined as more data is captured in the database. Therefore, comparing two databases on USUBJID and LBSEQ is likely not a meaningful exercise. The solution is to provide limited control over the search algorithm through two parameters. First, allow the user to prioritize certain variables for primary consideration by the search algorithm (**sorvars2incl**). Second, allow the user to deprioritize certain variables, relegating them to be considered last by the search algorithm (**sorvars2excl**). In this way, the user can reduce the likelihood that LBSEQ is used for sorting purposes, while ensuring that USUBJID is used when available in the dataset.

Leveraging a unique sort key on a generic database poses an additional challenge, particularly in an ongoing study, in that a dataset may contain records that are the same on all variable values (fully replicated records). To deal with this, the proposed solution is to identify the fully replicated records, report them to the user, and remove them from subsequent analyses. It is also suggested to modify impacted reports to indicate that fully replicated records have been removed.

After completion of all analysis, the results must be reported out to the user. It is the developer's discretion on what to report, how to report it, and how much control to grant the user with respect to the reports. This may include designing readable reports printed to the output destination (e.g., output window or LST file), datasets that will allow the user closer inspection of the findings, and/or messages displayed in the log which inform the user on the status of the macro processing. In this paper, we suggest the creation of reports, each of which is printed to the output window (LST in batch mode) in conjunction with outputting the underlying report dataset to the WORK library. As an added feature, if the user specifies, via **permdsetyn** and **perm_path**, permanent versions of each of the WORK datasets will be generated. Finally, when database differences are expected to be substantial, the user can specify **majupdyn**=Y, suppressing observation level comparisons (e.g., reporting of observations found in only one dataset or value differences for observations common to both datasets).

SOLUTIONS

The remainder of this paper details proposed solutions to each of these challenges, providing pseudocode and/or sample SAS code for each solution.

MACRO PARAMETER VALIDATION

For the macro to function properly, without processing error, the macro must incorporate validation checks on the values passed through the macro parameters. To accomplish this, the macro employs a common mechanism with the **%goto** statement. At the end of the macro call, the label **%EndMac:** is placed just above the **%mend** statement. The developer can then write conditional processing code on any parameter value to ensure that the value submitted by the user is suitable for processing. Any such checks should be placed toward the beginning of the macro.

For example, to enforce that the **base_path** is specified, the developer can specify the following:

```
%if &base_path eq %then %do;
    %put %str(E)RROR: DB_COMPARE: The parameter BASE_PATH is required.;
    %goto EndMac;
%end;
```

If the user fails to provide a value in **base_path**, execution will stop and a message will be written to the log. One can also verify that the specified paths along **base_path** and **comp_path** actually exist. Using **%sysfunc(fexist())**, we can specify the following code:

```
%local rc fileref;
%let rc=%sysfunc(filename(fileref,&base_path));
%if %sysfunc(fexist(&fileref)) %then %do;
   libname BaseData "&base_path.";
%end;
%else %do;
   %put %str(E)RROR: DB_COMPARE (base_path): The directory "&base_path" does not exist.;
   %put %str(E)RROR: DB_COMPARE (base_path): Processing will be stopped.;
   %goto EndMac;
%end;
```

If the path exists, then the library **BaseData** is defined using **base_path**, else processing stops. For remaining parameters, it is the macro developer's discretion to implement similar validation code enforcing desired restrictions on parameters. For example, in this macro, if **permdsetyn** is set to **Y**, it is required that **perm_path** is specified and exists, similar to **base_path**.

Another mechanism, which enhances robustness of the macro, removes case sensitivity of macro parameter values. For file paths, the parameter values may be case sensitive. However, the remaining parameters (dataset lists, variable lists, and yes/no toggles) do not need to be case sensitive. If the user specifies **permdsetyn=YeS**, we would like the macro to behave appropriately. One can remove case sensitivity by appropriately up-casing or low-casing both the macro parameters and the values it is compared against. Sample code to remove case sensitivity is presented in the following two examples:

```
/* Example 1 */ if %upcase(&permdsetyn) eq Y or %upcase(&permdsetyn) eq YES %then %do;
/* Example 2 */ %let l_dset_list=%lowcase(&dset_list.);
```

DATASET SPECIFICATION

The intent of the macro is to facilitate comparisons of entire SAS databases with a simple call. In this macro, we want to make the dataset specification as simple as possible, but also allow the user explicit control over which datasets are compared. For example, if the SDTM.LB dataset is quite large, perhaps the user would prefer to compare 'all but LB', analyzing LB through a separate process. The parameters **dset_list** and **excl_dset** are designed for this functionality.

The user may explicitly provide a space-delimited list of datasets through **dset_list**. If the user specifies **dset_list=ADSL ADAE ADLB**, then only these three datasets will be compared by the macro. However, if the user leaves **dset_list** null or specifies **dset_list=_all_** (default), then the macro will compare all datasets found in one of the two specified paths.

```
%if &l_dset_list eq %str(_all_) or &l_dset_list eq %then %do;
   /* Insert code for PROC CONTENTS on both libraries   */
   /* Set together and retain all unique MEMNAME values */
   /* Exclude values specified in &excl_dset            */
   /* Populate resulting list in &full_dset_list        */
%end;
%else %do;
    %let full_dset_list=&dset_list;
%end;
```

Note that when the **dset_list** is null or **_all_**, the value of **excl_dset** is used to remove selected datasets from comparison. The following sample call will compare all datasets *except* LB and PC

```
%db_compare(base_path      = %str(Z:\user\mr\basedata\)
           ,comp_path      = %str(Z:\user\mr\compdata\)
           ,dset_list      = _all_
           ,excl_dset      = LB PC
            );
```

The syntactic requirements of **dset_list** and **excl_dset** are minimal. Each of these are treated as space-delimited lists. The values are separated into individual components (using the space as a delimiter) and a temporary dataset is created with each component as a separate record. The resulting dataset is then merged into the result of PROC CONTENTS, flagging the records matching the values in **dset_list** (or **excl_dset**).

For example, suppose the user specifies **dset_list=AE LB DM**. A temporary dataset is created with the variable MEMNAME, containing three records with MEMNAME taking values of AE, LB, and DM. When this is merged with PROC CONTENTS on the **base_path** library, any instance of MEMNAME=AE, LB, or DM is flagged as a dataset to be analyzed. If the user errantly specifies datasets such as **dset_list=AF LB DM ADSL** when comparing SDTM databases, the macro will not encounter an execution error. The result is that only LB and DM will be analyzed. Hence, the macro does not need to check for existence of each specified dataset and will not produce an error due to non-existence. This method of incorporating user-specified datasets provides a unique degree of robustness in the macro call.

DERIVING UNIQUE SORT KEY

Due to the simplicity of the designed call, very little information about the datasets is available to the macro. In order to generate a meaningful compare, the datasets need to be sorted in a similar way. This is accomplished through a simple search algorithm which converges to a common set of variables that comprise a unique sort key. Although the resulting sort key is not necessarily the most efficient for sorting, nor is it necessarily the most useful for reviewing the data (e.g., by subject, parameter, and date/time of assessment), it does achieve the fundamental objective. It provides an effective sort key for comparing two datasets.

For each dataset being compared by the macro, the algorithm begins by identifying variables that are common to both BASE and COMP datasets. These variables are then grouped into 4 tiers as follows:

- Tier 1: Any variables specified in **sortvars2incl**.
- Tier 2: Any variables in neither Tier 1 nor Tier 4 and identified by PROC DATASETS in SORTEDBY, i.e. used in the sort order of the permanent dataset. This enhances the efficiency of search algorithm when the information is available.

- Tier 3: Any remaining variables (neither in Tiers 1, 2, nor 4), in order of VARNUM, from PROC DATASETS. In effect, once the user's preferences for priority and dataset metadata are accounted for, the algorithm searches left to right in the dataset (skipping any variables in Tier 4)
- Tier 4: Any variables not in Tier 1 and specified in **sortvars2excl**. Note that these variables are not really excluded; they are simply deprioritized. For example, SEQ type variables are not very meaningful for use in comparison because database updates can often redefine the SEQ values. However, as a last resort for uniqueness, variables in **sortvars2excl** are retained and relegated to lowest priority for the search algorithm.

As with any algorithm, various search parameters must be initialized. This macro uses 4 such parameters:

- **StopFlag**: Sets the stopping condition [Initial condition = 0]
- **NumVars**: The number of variables in the sort key [Initial condition = 1]
- **VarList**: Current state of the sort key [Initial condition = null]
- **PrevNumObs**: The number of duplicate observations from previous sort key
  [Initial condition = Total number of observations in both datasets + 1]

Starting with the dataset in **comp_path** and prioritizing its variables by Tier, SORTEDBY, and VARNUM, the first variable in the list (Tier 1) is selected and a **DO UNTIL** loop is initiated:

```
%do %until (&StopFlag=1);
```

Within the loop, a PROC SORT NODUPKEY is executed and the number of duplicate observations deleted (**NumObs**) is compared to **PrevNumObs**. If the **NumObs** is equal to zero, **StopFlag** is set to 1 and the loop will terminate. Otherwise, the next variable in the list is selected and a subsequent NODUPKEY sort is executed.

```
*** If number of dups = 0, stop ***;
%if %eval(&NumObs.)=0 %then %do;
   %let StopFlag=1;
%end;
```

If the number of duplicate observations deleted is less than **PrevNumObs**, then the variable is added to the **VarList** and **PrevNumObs** is updated.

```
*** If number of dups decreased, keep variable in list ***;
%if %eval(&NumObs.) < %eval(&Prev_NumObs.) %then %do;
   %let varlist = &varlist. &AddVar.;
   %let Prev_NumObs = &NumObs.;
%end;
```

If the number of duplicate observations is unchanged (**NumObs = PrevNumObs**), the test variable is skipped and the next variable is tested. This iterative search continues until the number of duplicate observations is equal to 0, at which point, the algorithm has converged and identified a unique sort key on the **comp_path** version of the dataset.

The algorithm then uses the **base_path** version of the dataset, starting with the current unique sort key for the **comp_path** version, and initiating a second **DO UNTIL** loop. If number of duplicates is also 0, then the algorithm stops. If it is greater than zero, it continues the search on the **base_path** data.

Note that it is possible that fully replicated records exist in one or both datasets. An additional stopping condition is implemented to exit the loop when all variables have been tested.

```
*** If tested last variable, stop ***;
%if %eval(&NumVars.)>%eval(&MaxNumVars.) %then %do;
   %let StopFlag=1;
%end;
```

When this condition is satisfied, we know that the dataset contains fully replicated records and a series of additional processing steps are executed to handle the replicated records. This is discussed in further detail in a subsequent section on fully replicated records.

The result of the search algorithm is a unique sort key comprised of common variables used to compare the two datasets.

USER CONTROL OVER SORT KEY

There are many methods available when designing the search algorithm for a unique sort key. Two simple methods would be a very crude left to right search using VARNUM from PROC CONTENTS and searching alphabetically by variable name. However, these methodologies do not leverage any information to improve either efficiency of the sort or effectiveness of the dataset comparisons. For example, STUDYID is often the first variable in an SDTM dataset and rarely adds to the uniqueness of a sort key. Even when the SORTEDBY metadata is populated, STUDYID is often the first variable in the sort key.

Therefore, the macro is designed to allow the user some influence on the sort key algorithm. Through two macro parameters (**sortvars2incl** and **sortvars2excl**), variables can be specified to prioritize, and deprioritize, by the search algorithm. Variables specified in **sortvars2incl** are assigned to Tier 1 and variables in **sortvars2excl** are assigned to Tier 4. Recall that the parameters **sortvars2incl** and **sortvars2excl** are assumed to be a space-delimited list. The macro processes these parameters in a similar manner to **dset_list** and **excl_dset**, placing them into separate macro variables and assigning them to the appropriate tier. As with the parameters specifying dataset lists, no macro execution error will occur if a specified variable does not exist. For example, if the user is comparing SDTM.EG and specifies **sortvars2incl=USUBJID LBCAT LBTESTCD LBDTC**, only USUBJID will be placed in Tier 1. Example code for assigning tiers is provided below:

```
data outdata;
   set indata;
   %do ii = 1 %to %eval(&neqi.);
      %if &ii > 1 %then else;
      if upcase(NAME) = "&&veqi&ii." then TIER=1;
   %end;
   %do ii = 1 %to %eval(&neqe.);
      else if upcase(NAME) = "&&veqe&ii." then TIER=4;
   %end;
   else if B_SORTEDBY^=. | C_SORTEDBY^=. then TIER=2;
   else TIER=3;
run;
```

In this code, the macro variables **neqi** and **neqe** are the number of variables specified in **sortvars2incl** and **sortvars2excl**, respectively. The other two nested macro variables (**&&veqi&ii** and **&&veqe&ii**) represent the individual variable names, indexed by **&ii** and converted to upper case to remove case-sensitivity.

The developer may include an additional feature in the macro such as the asterisk wildcard character '*'. One method of implementation allows the user to specify a variable name substring with a leading or trailing asterisk. Any variable name containing the substring would be assigned to the appropriate tier. For example, suppose the user is comparing SDTM databases and specifies **sortvars2incl=USUBJID *CAT *TESTCD *DTC** and **sortvars2excl=*SEQ**. For SDTM.LB, USUBJID will be prioritized into Tier 1 for the search algorithm, as will LBCAT, LBSCAT, LBTESTCD, and LBDTC, if they exist in the dataset. Similarly, for SDTM.EG, Tier 1 variables will include USUBJID, EGCAT, EGSCAT, EGTESTCD, and EGDTC. The following code can be inserted in the previous example code to search for the specified substring within the variable name:

```
%do ii = 1 %to %eval(&ncoi.);
   else if find(upcase(NAME),"&&vcti&ii.") > 0 then TIER=1;
%end;
%do ii = 1 %to %eval(&ncoe.);
   else if find(upcase(NAME),"&&vcte&ii.") > 0 then TIER=4;
%end;
```

In this additional code, the macro variables **ncoi** and **ncoe** are the number of variables with a leading or trailing wildcard character, and **neqi** and **neqe** now represent the number of remaining variables specified in **sortvars2incl** and **sortvars2excl**, respectively. As with the previous code, the nested macro variables (**&&vcti&ii** and **&&vcte&ii**) represent the individual variable strings to check, indexed by **&ii** and converted to upper case to remove case-sensitivity.

FULLY REPLICATE RECORDS

During the search algorithm for the unique sort key, if all variables are tested and the number of duplicate records from a PROC SORT NODUPKEY remains non-zero, then the dataset contains records with all values equal for variables common to both datasets. For datasets with fully replicated records, comparison becomes more difficult because there is no straightforward way to address comparison of these records.

In this situation, the macro separates the replicated records from the non-replicated records using the resulting sort key. The sort key will be unique for the dataset subset which contains the non-replicated records, rendering record level comparisons possible using the subset. Messages should be reported to the user indicating that comparisons are performed after removal of replicated records. The replicated records should also be explicitly reported to the user for review. Sample code is as follows:

```
data fulldups&jj.;
   set base_dups&jj.(in=b)
       comp_dups&jj.(in=a);
   by &&sortkey&jj.;
   if a then dupsource="CompData Library";
   if b then dupsource="BaseData Library";
   MEMNAME="&&dset&jj.";
run;
```

Here, the code is executed within the loop over datasets, indexed by **&jj**. A PROC SORT NODUPKEY by the unique sort key **&&sortkey&jj** is executed. Deleted records from the **dupout=** option are used to identify all replicated records, which are placed into **base_dups&jj.** and **comp_dups&jj.** The resulting dataset (**fulldups&jj**) contains all instances of **&&sortkey&jj** values with multiple records in one of the two databases.

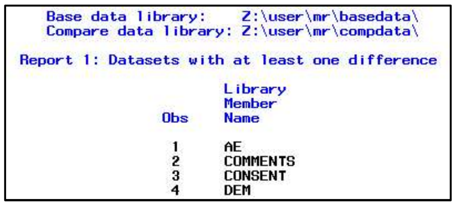
REPORTS TO THE USER

After completion of all analyses in the macro, the developer must determine what information will be reported, and through what mechanism. Providing readable reports that are meaningful to the user increases the likelihood of adoption. In the version of the macro presented here, three types of outputs are generated for each report.

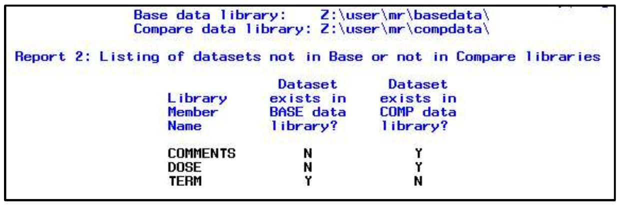
The first type of output are reports generated via PROC PRINT statements. If the macro is executed in interactive mode, the report is displayed in the output window. If executed in batch mode, the report will be printed to the SAS output file (e.g., .LST file). The second type of output are WORK library datasets used to generate each report. This allows the user to review the report findings more closely, particularly when the findings are more extensive. Third, if specified via **permdsetyn**, permanent versions of the WORK datasets are written to the library defined by **perm_path**. This may be useful if running in batch mode or if the final reviewer is not the user themselves (e.g., lead programmer executes, but other team member reviews).

DATABASE LEVEL REPORTS

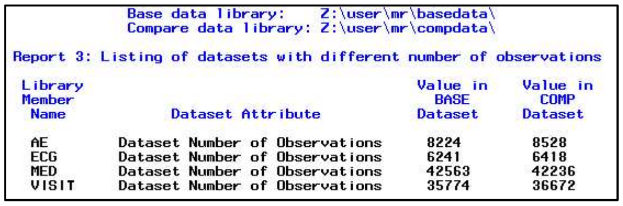
The macro generates three reports at the database level. The purpose of these reports is to provide a high-level overview of the degree to which the databases are discrepant. Report 1 is a listing of datasets with at least one difference identified in the macro (only the first four datasets are displayed in the example). Although this is the first report output by the macro, it cannot be generated until all analyses are completed. Report 2 lists datasets that are not found in one of the libraries (**base_path** or **comp_path**). Report 3 lists the datasets that have differing numbers of observations. Screenshots of these reports are below.



```
      Base data library:    Z:\user\mr\basedata\
      Compare data library: Z:\user\mr\compdata\

  Report 1: Datasets with at least one difference

                         Library
                         Member
                  Obs    Name

                   1     AE
                   2     COMMENTS
                   3     CONSENT
                   4     DEM
```



```
               Base data library:    Z:\user\mr\basedata\
               Compare data library: Z:\user\mr\compdata\

Report 2: Listing of datasets not in Base or not in Compare libraries

                             Dataset      Dataset
                 Library     exists in    exists in
                 Member      BASE data    COMP data
                 Name        library?     library?

                 COMMENTS       N            Y
                 DOSE           N            Y
                 TERM           Y            N
```



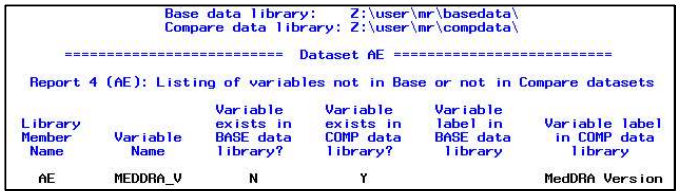
```
               Base data library:    Z:\user\mr\basedata\
               Compare data library: Z:\user\mr\compdata\

Report 3: Listing of datasets with different number of observations

 Library
 Member                                         Value in    Value in
  Name              Dataset Attribute             BASE        COMP
                                                Dataset     Dataset

  AE        Dataset Number of Observations       8224        8528
  ECG       Dataset Number of Observations       6241        6418
  MED       Dataset Number of Observations       42563       42236
  VISIT     Dataset Number of Observations       35774       36672
```

In the examples, there are multiple datasets with at least one difference identified by the macro (Report 1). Three of these datasets (**COMMENTS**, **DOSE**, and **TERM**) are due to existence in only one library (Report 2). **COMMENTS** and **DOSE** only exist in the **comp_path** library, whereas **TERM** is found only in the **base_path** library. Report 3 indicates that there are 4 datasets with different numbers of observations (**AE**, **ECG**, **MED**, and **VISIT**).

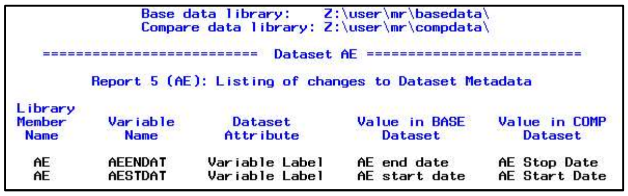
DATASET LEVEL REPORTS

The remaining reports are only generated if at least one discrepancy is identified for the dataset analyzed. If the datasets fully compare, then these reports are suppressed by the macro for ease of review by the user.

Two of these reports are generated at the dataset level. The first dataset level report (Report 4) lists variables that are found in only one dataset. For example, if the two databases are the result of a monthly raw data transfer from Data Management, this report will inform the user if any new variables were added or any variables were removed. For large databases, this can be a valuable mechanism for identifying unexpected changes to the database design. Report 5 lists any differences in dataset metadata (based on PROC CONTENTS) such as variable labels, lengths, formats, and types. As with Report 4, unexpected findings, such as changes in variable type, can significantly impact downstream programming if not identified swiftly. Screenshots of Reports 4 and 5 are below.



```
                 Base data library:    Z:\user\mr\basedata\
                 Compare data library: Z:\user\mr\compdata\

      ========================== Dataset AE ==========================

  Report 4 (AE): Listing of variables not in Base or not in Compare datasets

                        Variable    Variable    Variable
 Library                exists in   exists in   label in    Variable label
 Member     Variable    BASE data   COMP data   BASE data    in COMP data
  Name       Name       library?    library?    library        library

   AE       MEDDRA_V       N           Y                     MedDRA Version
```

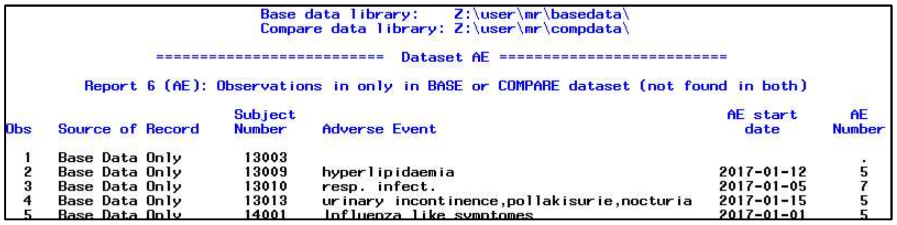


```
                 Base data library:    Z:\user\mr\basedata\
                 Compare data library: Z:\user\mr\compdata\

      ========================== Dataset AE ==========================

           Report 5 (AE): Listing of changes to Dataset Metadata

Library
Member     Variable       Dataset       Value in BASE    Value in COMP
 Name       Name         Attribute         Dataset          Dataset

  AE       AEENDAT     Variable Label   AE end date      AE Stop Date
  AE       AESTDAT     Variable Label   AE start date    AE Start Date
```

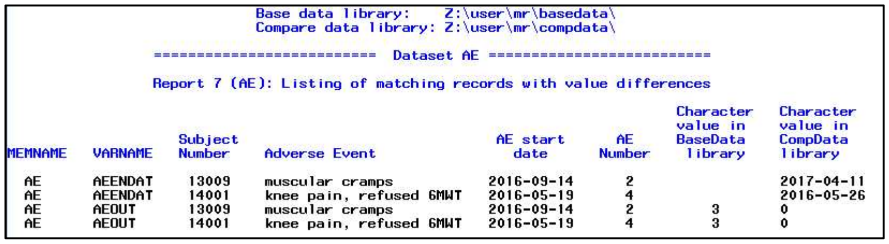
In the Report 4, we see that the source data has added the MedDRA Version Number to the database in the **comp_path** version of **AE**. In Report 5, we see that the labels on two variables (AESTDAT and AEENDAT) have been modified in the updated **comp_path** data.

RECORD LEVEL REPORTS

Reports 6 and 7 identify differences at the record level. Report 6 presents observations that are found in one dataset and not the other. Report 7 presents value level differences for each common variable on matching records. Identification of matching observations, versus those existing in only one dataset, is achieved through the unique sort key associated with the datasets being compared. For this reason, any fully replicated records existing in the original source datasets are removed prior to generating these reports. In this situation, an additional report is produced which lists all replicated records flagged with their source library (**base_path** or **comp_path**). As with all reports, the replicated records are also output to a dataset for further review. Screenshots of Reports 6 and 7 are below.



```
                          Base data library:    Z:\user\mr\basedata\
                          Compare data library: Z:\user\mr\compdata\

                 =========================  Dataset AE =========================

        Report 6 (AE): Observations in only in BASE or COMPARE dataset (not found in both)

                             Subject                                                AE start      AE
Obs   Source of Record       Number     Adverse Event                                 date      Number

 1    Base Data Only          13003                                                                  .
 2    Base Data Only          13009     hyperlipidaemia                            2017-01-12        5
 3    Base Data Only          13010     resp. infect.                              2017-01-05        7
 4    Base Data Only          13013     urinary incontinence,pollakisurie,nocturia 2017-01-15        5
 5    Base Data Only          14001     Influenza like symptomes                   2017-01-01        5
```



```
                          Base data library:    Z:\user\mr\basedata\
                          Compare data library: Z:\user\mr\compdata\

                 =========================  Dataset AE =========================

                 Report 7 (AE): Listing of matching records with value differences

                                                                          Character     Character
                                                                          value in      value in
                      Subject                           AE start    AE    BaseData      CompData
MEMNAME   VARNAME     Number     Adverse Event            date    Number  library       library

  AE      AEENDAT      13009     muscular cramps       2016-09-14    2                  2017-04-11
  AE      AEENDAT      14001     knee pain, refused 6MWT 2016-05-19  4                  2016-05-26
  AE      AEOUT        13009     muscular cramps       2016-09-14    2        3         0
  AE      AEOUT        14001     knee pain, refused 6MWT 2016-05-19  4        3         0
```

The information in Reports 6 and 7 can be produced by PROC REPORT. However, if the findings are extensive, details of some results may be suppressed by the PROC REPORT output listing. In this macro, complete findings are reported to the user via PROC PRINT and the output dataset. However, the macro also produces an additional report (Report 8) through the execution of PROC COMPARE using the unique sort key in the ID statement. For users that are more comfortable with the output generated by this procedure, they may find it more meaningful to review.

Finally, in addition to outputting the datasets for each report, the **base_path** and **comp_path** datasets are completely outputted to the WORK library via PROC SORT using the unique sort key, which may aid in the review of findings. As with other reports, if specified via **permdsetyn**, permanent versions will be created in **perm_path**.

USER SUPPRESSION OF RECORD LEVEL REPORTS

When database differences are expected to be substantial (e.g., during an ongoing study with active recruitment, as data continues to be entered into the database), the user can specify **majupdyn=Y**. Setting this value for the parameter will suppress record level reports (Reports 6-8), as well as the generation of the sorted **base_path** and **comp_path** datasets.

CONCLUSION

This paper discussed several challenges, and proposed solutions, when developing a robust database compare macro. Such a macro may prove efficiency enhancing to programming groups which receive regular data transfers of their input data after much of the programming infrastructure for the study is developed. The macro allows the user to assess the degree to which the database has changed from the previous transfer, potentially significantly reducing the amount of resource needed to maintain the programming infrastructure when the updated transfer is put into production. The macro can also be applied to the output of database programming, verifying that only expected changes have occurred to the mapped database following a rerun of all programming.

CONTACT INFORMATION

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

Michael S Rimler
FMD K&L Inc.
1300 Virginia Dr., Ste 408
Fort Washington, Pennsylvania, US, 19034
Email: michael.rimler@klserv.com