Paper 014-29

SAS® Data Quality – Cleanse: Techniques for Merge/Purge on Very Large Datasets

By Michael Krumenaker – Sr. Project Manager, Palisades Research, Inc.
George Bukhbinder – President, Palisades Research, Inc.
Xiaoyan Yang - Analyst, Palisades Research, Inc.

Abstract

The SAS System can perform merge/purge – the process of combining lists and removing duplicates – with the functionality provided by SAS Data Quality-Cleanse ("SAS-DQ"). In developing SAS programs for monthly merge/purge on hundreds of millions of records, we encountered various challenges, including (1) how to significantly decrease processing time, and (2) how to combine partially overlapping duplicate groups based on various criteria.

Introduction: Performing Merge/Purge

In merge/purge, much attention goes to a software package's ability to perform "fuzzy matches", i.e., to recognize strings that match inexactly but really represent the same person, address, etc., or two very different first names that are really a name and nickname. If data is standardized, most matching involves simple exact matches. Standardization will not correct typos, but will unify the different forms of a first name. Still, fuzzy matching remains very important. Without well-conceived fuzzy match algorithms, the result is likely to be unsatisfactory, with too many false positives and/or false negatives.

SAS-DQ creates match codes[1] for strings based on their characters, locale, and various user-selected settings. These settings include sensitivity – a measure of match code complexity[2]. The higher the sensitivity, the more exact the match must be to obtain the same code. Users can also refine the process by manipulating string parsing algorithms and the rules for applying match code sensitivities through the companion product df-Customize® from DataFlux,® which is owned by SAS.

Parameters in Merge/Purge

The merge/purge process requires several choices:

1. The fields for matching. The process can include matching on more than one set of fields, in which case records which match on either set of fields are included in the same duplicate group. For instance, suppose match codes are created for the name and address and are combined in the following alternate match criteria with the actual zip code and phone number:

   Name – Address – Zip Code Name – Phone Number

   If records 1 and 2 match on Name – Address – Zip Code, and records 2 and 3 match on Name – Phone Number, then records 1, 2, and 3 should be a single duplicate group. Combining the two initial groups to form a group of records [1,2,3] is discussed later in this paper.

2. Selecting tokens from each field used for matching. "Tokens" are discreet, commonly understood portions of each type of field in the match criteria. Names may include first, middle and last names, a prefix (Mr., Ms.,…), a suffix (Jr., Sr.,…) and/or a professional designation (MD., CPA, etc.). Addresses may consist of pre-direction (East, West,…), house number, street name, street type (ST, AVE,…) and post-direction (East, West,…). Optimizing the matching process may include comparing match results using different combinations of tokens. Some token requirements are obvious (first name, last name, street number, street name), but testing will reveal which others will be helpful or harmful.

3. Selecting sensitivity, and, through df-Customize, adjusting the way changes in sensitivity affect match codes. A detailed discussion is beyond the scope of this paper. Sensitivity selection is a combination of choosing a numerical level of sensitivity and determining at each such level how many characters in each token SAS-DQ uses to create the match code.[3]

Challenge #1: Reducing Processing Time

Creating match codes using SAS-DQ is processor-intensive and slow, so we create algorithms in Base SAS to reduce the number of match codes SAS-DQ has to create. We can take advantage of the enormous amount of exact duplication in values for a given token, and then use simple sorting, by-grouping, and the **retain** statement to dramatically reduce the number of match codes we must create. This may require pre-parsing the whole name or address using base SAS, involving one or more sorts and perhaps one or more additional data steps. Although SAS-DQ can parse name, address, and other kinds of strings into the tokens described in item (2), above, that process is also slow and would defeat the purpose of speeding up match coding. There may be only a handful of repetitions (exact matches) of "11 MAIN ST" in the list, but consider how many times the street name "MAIN" appears and how many times the house number "11" appears and you will understand the value of the parsing operation.

The order of tokens in a string is often pre-determined. For instance, we may know that the names from a particular data source always appear in the order last-first-middle-suffix-profession. So, we can use the most elementary SAS parsing functions (scan(), indexw(), etc.), SAS regular expressions, or, in Version 9, Perl regular expression functions in SAS, to extract each token.

We parsed the addresses and retained the house number and street name tokens for match coding. We sorted the datasets by house_number and applied the **dqMatch()** function to the first record in each such By group, applying the **retain** statement to the variable **house_number**. We then sorted the dataset by street_name, and created match codes on street name in the same manner as house number. The strategy reduced the number of match codes being created on street addresses from over 2 million to just a few tens of thousands. Except for an extra **proc sort** (which ran quickly enough on our system), the enormous time savings was almost cost free – the Perl parsing functions we created to recognize common street address patterns (11 MAIN STREET, 67 E 33 ST) and render the tokens added very little processing time. The net time savings was enormous and very necessary to the success of the project.

The Perl regular expressions we used in SAS to analyze street addresses included the following:

```
*Format as in 11 MAIN ST;
prxAddress1Street = prxparse("/^([0-9][0-9a-zA-Z]*)(\s)([a-zA-Z0-9][a-zA-Z0-
9]+)(\s)(STREET|ST|AVENUE|AVE|LANE|LN|PARKWAY|PKWY|WAY|ROAD|RD
|DRIVE|DR|PLACE|PL|CIRCLE|CIR|COURT|CT|PIKE|TERRACE|TER|TRAIL|TRL|TL
|BOULEVARD|BLVD)$/");

*Format as in 11 E MAIN ST;
prxNumPreStreet = prxparse("/^([0-9][0-9a-zA-Z]*)(\s*)
(E|EAST|W|WEST|S|SOUTH|N| NORTH)(\s*)([a-zA-Z0-9][a-zA-Z0-9]+)
(\s*)(STREET|ST|AVENUE|AVE|LANE|LN|PARKWAY|PKWY|WAY|ROAD|RD|DRIVE
|DR|PLACE|PL|CIRCLE|CIR|COURT|CT|PIKE|TERRACE|TER|TRAIL|TRL|TL
|BOULEVARD|BLVD)$/");

*Format as in 35 ROCK CREEK RD;
prxAddress2Street = prxparse("/^([0-9][0-9a-zA-Z]*)(\s)([a-zA-Z0-9][a-zA-Z0-
9]+)(\s) ([a-zA-Z][a-zA-Z]+)(\s)(STREET|ST|AVENUE|AVE|LANE|LN
|PARKWAY|PKWY|WAY|ROAD|RD|DRIVE|DR|PLACE|PL|CIRCLE|CIR|COURT|CT|PIKE
|TERRACE|TER|TRAIL|TRL|TL |BOULEVARD|BLVD)$/");
```

Important: In the data step described above, the **prxparse()** functions should only be run on the first record in the dataset, and their values retained thereafter (using a **retain** statement). The function **prxparse()** runs slowly and creates a new parse definition value each time it executes, but does not need to be executed more than once per definition if such definitions are to be held constant for the entire input dataset (which is usually the case).

"Address" (the whole street address) is a data type in df-Customize. Our strategy requires the use of df-Customize to create two new data types for matching. We created separate data types for house number and street name, each with only a single token. Had we used the standard Address data type for each of these separate strings ("11" and "MAIN"), SAS-DQ would not always have properly evaluated whether the separate token being analyzed was a house number or street name (e.g., 34th is a street name). A detailed discussion of creating SAS-DQ data types using df-Customize is beyond the scope of this paper, but basically we created two copies of the standard Address data type and altered the parsing rules of each copy to create the two new data types.

Match coding on names did not require tokenization, although it might have been worth the additional sort time. It was enough to sort by the full name, since most matches were on exact name strings, and use the retain statement and by-grouping as previously described to obtain the match code on the entire name.

Challenge #2: Combining Duplicate Groups Created Using Two Different Sets of Criteria

Once match codes were obtain for each name and address, two separate indexes were created for each record by concatenating these match codes with the actual phone number and 5-digit zip code:

```
index1 = name_match_code||address_match_code||zip5;
index2 = name_match_code||phone;
```

Each group of records with the same **index1** constituted a duplicate group ("dupe group"). Similarly, each group of records with the same **index2** constituted a dupe group. Such dupe groups generally had one, two or three records.

To complete the matching process, we had to combine overlapping pairs of dupe groups. "Overlapping" means that one dupe group based on **index1** had at least one record in common with one dupe group based on **index2**. In fact, the pattern could get more complicated. For example, using X, Y, M, N and P to identify five dupe groups:

Record	**index1** dupe group	**index2** dupe group
1	M	X
2	N	X
3	N	X
4	N	X
5	P	Y

Trace the overlaps, and you see that all five of these records belong in a single dupe group (through the combinations X with M and N, Y with N, and Y with P).

SAS-DQ does not provide functions or procedures to automatically combine the two sets of dupe groups (those based on each "index"). We formulated an algorithm to perform this task using Base SAS. In creating **index** variables for the algorithm, the **index** (1 or 2) having the greater number of dupe groups should be **index1**.

Step 1: For efficient processing, reduce the byte size of each record

We now have a dataset containing **index1**, **index2**, and all the many fields (demographic and other data related to each individual) from several vendors. Call this dataset **&source_data**. The goal of the merge/purge is to find the records from each of the different sources - as well as from within a given source, if there are such duplications - that relate to the same individual. Each record in **&source_data** has information from only one source.

To minimize I/O time, if there is no simple unique key for each record, assign unique counter numbers (using **_N_**) to each record in **&source_data** and then create a "skinny" dataset (call it **try**) containing only four variables: **index1**, **index2**, the **counter**, and a new variable called **cluster_number** that is initialized with the same values as **counter** (_N_) for each record. Ultimately, each final dupe group will have a unique **cluster_number**. To prevent certain errors, if **index1** is blank then re-assign **index1** the value of **_N_**, and do the same for **index2**.

Using a (grossly) oversimplified example, dataset **try** has just the following seven records (as opposed to millions or tens of millions), where **index1** and **index2** are also grossly simplified (they would normally be character strings of length 35 (**index1**) and 25 (**index2**)), and the variable **cluster_number** is represented by the macro variable **&clusterfield**:

index1	index2	&clusterfield
A	2	1
E	2	2
E	1	3
B	1	4
C	3	5
F	7	6
F	7	7

Step 2: Assign common cluster numbers to each By group on **index1** and identify multi-record groups

Manipulate **try** to add by-grouping on **index1**, and two new variables: **&cluster_number.1** (with a unique value for each By group) and **dupindex1** (to identify By groups with multiple values):

```
proc sort data=try; by index1; run;
%let cluster_field=cluster_number;
data try;
       set try;
       by index1;
       retain &cluster_field.1;
       if first.index1 then &cluster_field.1=&cluster_field.;
       if first.index1.*last.index1=0 then dupindex1=1;
       else dupindex1.=0;
run;
```

Result:

index1	index2	&clusterfield	&clusterfield.1	dupindex1
A	2	1	1	0
B	1	4	4	0
C	3	5	5	0
E	2	2	2	1
E	1	3	2	1
F	7	6	6	1
F	7	7	6	1

Note how **&clusterfield.1**=2 for both instances of **index1**=E and is 6 where **index1**=F, and **dupindex1** shows which combined dupe groups based on **index1** have multiple records at this stage.

Step 3: Remove records which are unique on both index1 and index2

Records which have a unique value for **index1** and a unique value for **index2** are not part of the search for overlapping dupe groups, so we remove them from processing by the algorithm:

```
Proc sort data=try; by index2; run;
data nodup(keep=index1 index2 cluster_number counter) dup;
       set try;
       by index2;
       retain index2;
       if first.index2 then do index2= index1;
       if first.index2*last.index2=0 then dupindex2=1;
       else dupindex2=0;
       if (dupindex1+dupindex2)=0 then output nodup;
       else output dup;
       drop dupindex2 dupindex1;
run;
```

Result:

index1	index2	&clusterfield	&clusterfield.1	&clusterfield.2	to Dataset:
B	1	4	4	4	**dup**
E	1	3	2	4	**dup**
A	2	1	1	1	**dup**
E	2	2	2	1	**dup**
C	3	5	5	5	**nodup**
F	7	6	6	6	**dup**
F	7	7	6	6	**dup**

In our example, only the fifth record has **index1**=C, and only the fifth record has **index2**=5, so that this record is unique on both indexes and goes into dataset **nodup**. We continue processing the dataset **dup**.

Step 4: De-dup the list of all combinations of **index1** and **index2**

In our example, the second record with **index1** = F and **index2**=7 (the seventh record in the previous list) goes away:

```
proc summary data=dup;
       by index2 index1;
       output out=temp (keep=index2 index1);
run;
```

Result:

index1	index2
B	1
E	1
A	2
E	2
F	7

Step 5: Identify non-unique values for **index2** and obtain **index1** values for such values of **index2**

We obtain the values of **index1** for records for which there are dupes on the combination **index1**, **index2**.

```
proc sort data=temp; by index2 index1; run;
data select(keep=index1) ;
       set temp;
       by index2;
       if first.index2*last.index2=0 then output;    *index2 not unique;
run;
```

Result:

index1	index2
B	1
E	1
A	2
E	2

The fifth record in the previous list was unique on both indexes, so it did not go into the dataset **select**.

Next, sort **select** on **index1** with the **NODUPKEY** option, to find the unique values of index1 in **select**.

Result:

index1	index2
A	2
B	1
E	1

Step 6: Identify records whose **index1** value has unique pairing with **index2** and those that with non-unique pairing to **index2.**

Dataset **dup**, (see step 3) contains the six records for which either **index1** or **index2** or both is not unique. We created the dataset **select** (list of non-unique values for **index2**) in step 5. Separate **dup** into (1) records having values of **index2** that match more than one value of **index1** (dataset **complex**) and (2) all other records from dataset **dups** (dataset **simple**):

```
proc sort data=dup; by index1; run;
data simple(keep=id index1 index2 &cluster_field.2
 rename=(&cluster_field.2=&cluster_field.)) complex;
       merge dup(in=a) select(in=b );
       by index1;
       if a and b then output complex;
       else output simple;
       drop &cluster_field.;
run;
```

These records from **dup** go into **complex**:

Result:

index1	index2	&clusterfield	&clusterfield.1	&clusterfield.2
A	2	1	1	1
B	1	4	4	4
E	1	3	2	4
E	2	2	2	1

The other records from **dups** go into simple.

Step 7: The iterative process of collapsing dupe groups

Combine overlapping dupe groups by collapsing **complex** (represented by the macro variable **&complex**) until the number of dupe groups does not change from one iteration to the next. The iterative process is controlled by the **%do %while** loop. Each of the macro variables **&d1**, **&d2** and **&d3** is incremented by +1 in each iteration:

```
%macro dup_iterate(complex);
       %let d1=0;
       %let d2=1;
       %let d3=2;
       %let groupcnt1=1;
       %let groupcnt0=0;
       %do %while(&&groupcnt&d2 ne &&groupcnt&d1);
              %let d1=%eval(&d1+1);
              %let d2=%eval(&d2+1);
              %let d3=%eval(&d3+1);
              proc sort data=&complex; by &cluster_field.&d1; run;
              data &complex;
                     set &complex end=EOF;
                     by &cluster_field.&d1;
                     retain &cluster_field.&D3;
                     if first.&cluster_field.&d1 then do;
                            &cluster_field.&D3=&cluster_field.&D2;
                            groupcnt+1;
                     end;
                     if EOF then call symput("groupcnt&d2",
                      left(trim(groupcnt)));
                     drop groupcnt ;
              run;
       %end;
```

The fist iteration of the data step produces the following results for **complex**:

index1	index2	&clusterfield.1	&clusterfield.2	&clusterfield.3
A	2	1	1	1
E	1	2	4	4
E	2	2	1	4
B	1	4	4	4

groupcount = 3, i.e., the number of unique values in **&clusterfield.1** is 3.

Iteration 2:

index1	index2	&clusterfield.2	&clusterfield.3	&clusterfield4
A	2	1	1	1
E	1	1	4	1
E	2	4	4	4
B	1	4	4	4

groupcount = 2, i.e., the number of unique values in **&clusterfield.2** is 2.

Iteration 3:

index1	index2	&clusterfield.3	&clusterfield.4	&clusterfield.5
A	2	1	1	1
E	1	4	1	1
E	2	4	4	1
B	1	4	4	1

groupcount = 2, i.e., the number of unique values in **&clusterfield.3** is 2, which is the same as the previous **groupcount**, so iteration stops. All dupe groups overlapping on **index1** or **index2** have been identified. We then gather together all the records by creating the final **&clusterfield**, appending the dataset **simple** to **complex**, (identified by the macro variable **&complex**) and then appending the dataset **nodup**. Finally, we merge the combined dataset back into the original source dataset (**&source_data**). All of these steps are shown below:

```
        *Create final clusterfield variable;
        data &complex;
               set &complex(keep=index1 index2 id &cluster_field.&d3
                rename=(&cluster_field.&d3.=&cluster_field.));
        run;

        *Append simple to complex;
        proc datasets nolist library=work;
               append base=&complex data=simple(keep=&key_field1.
                Index2 &cluster_field. id);
        ;run;

        *Append nodup to complex;
        proc datasets nolist library=work;
               append base=&complex data=nodup;
        ;run;

        %*** Re-assemble dataset with all details (input set) ***;
        proc sort data=&complex; by id; run;
        data mylib.out_data;
               merge &source_data &complex;
        run;

%mend;
%dup_iterate(complex);
```

The cluster number (**&clustfield**) in each record in **&source_data** is the link between records in **&source_data** in each respective dupe group.

Step 8: Merging the records in each dupe group into a single record for each dupe group

For each dupe group in **&source_data**, subsequent processing creates a single record from the separate records for each dupe group. We do not describe the process here, except to say that we (1) sort **&source_data** by the cluster number and some other variable that we created in preprocessing that assigns priorities to each of the respective sources, and (2) use a data step with By grouping based on the cluster number to draw data from the various sources in the dupe group together - using retain statements - in such a way that that if data from sources A, B and C exist in the same dupe group, we do not overwrite data from a higher priority source with data from a lower priority source, but we do allow the lower priority source to fill in fields in the combined record not already populated by a higher priority source within that By group. Depending on your business rules, the resulting data step may be complicated, but it does not involve anything beyond a lot of detail and routine Base SAS techniques.

Conclusion

Merge/purge requires optimal selection of matching criteria and sensitivity parameters, and testing various combinations thereof. We have employed SAS-DQ using df-Customize and algorithms in Base SAS to reduce the time for match code creation and increase the accuracy of fuzzy matching. The SAS-DQ product would be enhanced by the addition of a procedure to join dupe groups, but we created an algorithm for this purpose.

CONTACT INFORMATION

George Bukhbinder President Palisades Research, Inc. 75 Claremont Road, Suite 312 Bernardsville, NJ 07924 908-953-8081 mkrumenaker@palisadesresearch.com	Michael Krumenaker Sr. Project Manager Palisades Research, Inc. 75 Claremont Road, Suite 312 Bernardsville, NJ 07924 908-953-8081 mkrumenaker@palisadesresearch.com

REFERENCES

[1] See, generally, SAS Institute, "Identifying Match Code Definitions Using dfPower Studio", support.sas.clm/rnd/warehousing/quality/stra_match.html.

[2] SAS Institute (2002), SAS Data Quality 9-Cleanse: Reference, 16.

[3] See DataFlux, "dfPower Customize® 6.0 User's Guide, 57-59.