Merge-Purge of Very Large Datasets Using SAS/DQ – Cleanse: 
Algorithm for Multiple Match Group Criteria

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

Abstract

Merge/purge entails the identification of groups of records (“duplicate groups”) that according to user-defined fuzzy matching criteria are deemed to be the same person or entity. Possible matching criteria include the combination of name-address-zip, or name-phone. Often it is best to determine duplicate groups based on more than one set of criteria, e.g., to consider records that match on either name-address-zip or name-phone to be the same person or entity. In such situations, we consider two records that match on one set of criteria (say name-address-zip) and two records matching on the other set of criteria (name-phone) to all be the same person or entity if the pairs overlap.

SAS-DataQuality/Cleanse can be used to perform merge/purge on very large data sets. However, when we use more than one set of matching criteria we must combine overlapping pairs of duplicate groups. For example, if records 1 and 2 match on name-address-zip, and records 2 and 3 match on name-phone, then we need a way to connect records 1, 2 and 3 to avoid duplication, identify the same person or entity as accurately as possible, and include the maximum amount of information available about the person or entity from our list vendors.

The problem: SAS-DataQuality/Cleanse does not yet provide this functionality (combining duplicate groups based on alternate criteria) on its own. So, Palisades Research formulated an algorithm to perform this task using Base SAS.

The Data

The project involved merging millions of records from three types of sources. All three sources included names and addresses, but one of them did not include phone numbers. The sources did not necessarily represent either the name or address records in exactly the same way, necessitating fuzzy matching. One source might include a middle name or middle initial; another might have misspelled a name or part of the street address. A source may have updated an address but not a phone number, or vice-versa.

Matching was performed on two sets of fields:

<table>
<thead>
<tr>
<th>Name – Address – Zip Code</th>
<th>Name – Phone Number</th>
</tr>
</thead>
</table>

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. We used fuzzy matching for name and address, and only exact matching for phone number and zip code.

The criteria for choosing fields for fuzzy matching vs. exact matching, and determining what will and will not be considered a fuzzy match, are not the subject of this paper, and could probably be the subject of a short book. It is an iterative process, but it comes down to experience, common sense, experimentation, and a lot of grinding through the results, fine-tuning, and grinding again.

How SAS-DQ/Cleanse Performs Fuzzy Matching

SAS-DQ creates match codes for strings based on their characters, locale, and various user-selected settings. These settings include sensitivity, which influences match code complexity, i.e., the number of different match codes that will be created for a group of similar strings. 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.

Two strings that are not exact matches but receive exactly the same match code are considered the same. If the match codes are not exactly the same, the strings are not considered a match.
**How we used the Match Codes**

Once match codes were obtained for each name and address, we performed two separate concatenations for each record. We call the concatenated strings “indexes”. We concatenated the match codes with the actual phone number and 5-digit zip code, respectively:

```
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. Records in a dupe group are deemed to represent the same individual.

To complete the matching process, we had to combine overlapping pairs of dupe groups. “Overlapping” means that a dupe group based on `index1` had at least one record in common with a 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:

<table>
<thead>
<tr>
<th>Record</th>
<th><code>index1</code> dupe group</th>
<th><code>index2</code> dupe group</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>M</td>
<td>X</td>
</tr>
<tr>
<td>2</td>
<td>N</td>
<td>X</td>
</tr>
<tr>
<td>3</td>
<td>N</td>
<td>X</td>
</tr>
<tr>
<td>4</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>5</td>
<td>P</td>
<td>Y</td>
</tr>
</tbody>
</table>

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). Hence all five records are deemed to represent the same individual.

SAS-DQ does not provide functions or procedures to automatically combine the two sets of dupe groups (those based on each of our “index” variables). We formulated an algorithm to perform this task using Base SAS.

In creating `index` variables for the algorithm, the `index` with the greater number of dupe groups should be `index1`.

Important: Throughout this discussion, keep the following in mind: A dupe group is a dupe group precisely because every record in the group has (1) the same value for `index1`, or (2) the same value for `index2`, or (3) the same value for `index1` and the same value for `index2`. To avoid confusion with respect to the third type of dupe group, remember that given the nature of the indexes, the value of `index1` is never the same as the value of `index2`, no more than the value “automobile make” could be the same as “house model” – they are simply different animals.

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

Once we have created `index1`, `index2`, the dataset still contains all the many fields (demographic and other data related to each individual) from several list vendors. Call this dataset `source_data`. The goal of merge/purge is to find the records from each of the different sources, as well as from within a given source that contains duplicates, that are deemed to be the same individual. Each record in `source_data` has information from only one source. Remember, there were three sources, so the same person could be in one or more records from each of the three sources.

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 as a single character (they would normally be character strings of length 35 (`index1`) and 25 (`index2`)), and the variable `cluster_number` is the dupe group number. We begin by placing each record in a separate cluster (dupe group):
Step 2: Assign common cluster numbers to each By group on index1 and identify multi-record groups on index1

We start the analysis by finding duplicates based on the first index alone. Add by-grouping to dataset try on index1. Initially, each by-group will be a cluster, so we add the variable cluster_number1, which is given a unique value for each by-group (not each record). We distinguish between by-groups with only one record and by-groups with multiple records, using dupindex1:

```
proc sort data=try;
  by index1;
run;

data try;
  set try;
  by index1;
  retain cluster_number1;
  if first.index1 then cluster_number1=cluster_number;
  if first.index1.*last.index1=0 then dupindex1=1;
  else dupindex1=0;
run;
```

The value of cluster_number1 changes only when we start a new by-group. So cluster_number has the original cluster numbers (one for each record) and cluster_number1 has the new cluster numbers (one for each index1 by-group):

<table>
<thead>
<tr>
<th>Record</th>
<th>index1</th>
<th>index2</th>
<th>cluster_number</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>E</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>E</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>B</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>C</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>F</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>F</td>
<td>7</td>
<td>7</td>
</tr>
</tbody>
</table>

Note that cluster_number1=2 for both instances of index1=E and is 6 where index1=F, and dupindex1 shows which dupe groups based on index1 have multiple records at this stage.

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

Records which have a unique value for index1 and a unique value for index2 are not in any dupe groups and must be excluded from the search for overlapping dupe groups, so we remove them from further processing by the algorithm. We create a second dupindex (dupindex2) with the value 0 if index2 is unique and 1 if index2 is not unique. If a record has no duplicates on index1 (dupindex1=0) and no duplicates on index2 (dupindex2=0), it is totally unique and is excluded from the search for dupe groups. Also, this data step uses a new variable – cluster_number2 – representing new clusters based on index2 instead of index1, and assigns duplicates on index2 to the same cluster, through the use of the retain statement in the data step below.

```
proc sort data=try;
  by index2;
run;
```
data nodup(keep=index1 index2 cluster_number counter) dup;
set try;
by index2;
retain cluster_number2;
if first.index2 then cluster_number2=cluster_number1;
if first.index2*last.index2=0 then dupindex2=1; *>1 instance;
else dupindex2=0;
if (dupindex1+dupindex2)=0 then output nodup; *no dupes on either index 1 or 2;
else output dup;
drop dupindex2 dupindex1;
run;

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. For this record, dupindex1 + dupindex2 = 0, so it is output to dataset nodup.

Dataset nodup

<table>
<thead>
<tr>
<th>index1</th>
<th>index2</th>
<th>cluster_number</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>3</td>
<td>5</td>
</tr>
</tbody>
</table>

The sum dupindex1 + dupindex2 for all other records is 1, so they are output to dataset dup:

Dataset dup

<table>
<thead>
<tr>
<th>index1</th>
<th>index2</th>
<th>cluster_number</th>
<th>cluster_number1</th>
<th>cluster_number2</th>
<th>dupindex1</th>
<th>dupindex2</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>1</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>E</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>4</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>A</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>E</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>F</td>
<td>7</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>F</td>
<td>7</td>
<td>7</td>
<td>6</td>
<td>6</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Note that all of the records in this example have a duplication on index2 (unique values are 1, 2, and 7), so all have dupindex2 set to 1.

We continue processing the dataset dup.

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

In our example, we have a duplicate on the combination of index1, index2 for (F,7), so the last record in the immediately preceding table is an obvious duplicate and is removed from the process via the following code:

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

This leaves the following five records (in the real world, perhaps 50,000 or 5 million or 50 million) in the dataset temp:

Dataset temp

<table>
<thead>
<tr>
<th>index1</th>
<th>index2</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>1</td>
</tr>
<tr>
<td>E</td>
<td>1</td>
</tr>
<tr>
<td>A</td>
<td>2</td>
</tr>
<tr>
<td>E</td>
<td>2</td>
</tr>
<tr>
<td>F</td>
<td>7</td>
</tr>
</tbody>
</table>
Step 5: Identify non-unique values for index2 and obtain index1 values for such values of index2

If an index2 value is not unique, then all records with that same index2 value are deemed to be the same person, i.e., they are a dupe group. In the table above, we see that 1 and 2 are non-unique values of index2, so they form two distinct dupe groups, even though the index1 values within each such dupe group are different. We obtain the values of index1 for such index2 dupe groups by placing records with non-unique values for index2 in the dataset select:

```sql
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;
```

The net effect is to exclude the unique values of index2 (in our example, 7, even though it is part of exact index1-index2 duplication):

<table>
<thead>
<tr>
<th>index1</th>
<th>index2 (variable is not included in dataset)</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>1</td>
</tr>
<tr>
<td>E</td>
<td>1</td>
</tr>
<tr>
<td>A</td>
<td>2</td>
</tr>
<tr>
<td>E</td>
<td>2</td>
</tr>
</tbody>
</table>

In temp, the fifth record (F,7) is unique on index2, so it did not go into the dataset select. The values of index2 are shown above for clarity, but this variable is not included in the dataset select.

Next, sort select on index1 with the NODUPKEY option, to list the unique values of index1 in select:

<table>
<thead>
<tr>
<th>index1</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
</tr>
<tr>
<td>B</td>
</tr>
<tr>
<td>E</td>
</tr>
</tbody>
</table>

Now we have unique values of index1 that are paired with non-unique values of index2, excluding exact duplicates on both indexes (F,7, above).

Step 6: Separate the set of non-unique records into those that are members of overlapping dupe groups and those that are members of non-overlapping dupe groups.

If none of the records in a particular dupe group is a member of any other dupe group, we refer to that dupe group as “non-overlapping”. Conversely, if at least one record that is in the dupe group because of its value for index1 is also in another dupe group because for its value of index2, or if it is in the first dupe group because of its value for index2 and is in another dupe group because of its value for index1, both dupe groups are “overlapping” with each other (and perhaps other dupe groups, based on their other records).

Remember, every dupe group has only one value for index1, or only one value for index2, or only one value for each.

In our example, dataset dup, (see step 3), which has every record except those that are unique on both indexes, contains six records. For every record in dup either index1 or index2 or both is not unique. Each record in dup is described by only one of the following:
- a dupe group on index1 but no dupe group on index2
- a dupe group on index2 but no dupe group on index1
- a dupe group on both index1 and index2 (F,7)
- a dupe group on index1 and a different dupe group on index2
- a dupe group on index2 and a different dupe group on index1

We created the dataset select (list of non-unique values for index2) in step 5. We separate all records belonging to dupe groups (dataset dup) as follows:

1. dataset complex – records belong to overlapping dupe groups, i.e., records having values of index2 that are associated with more than one value of index1

2. dataset simple – non-overlapping dupe groups (all other records from dataset dup)

We merge the all-duplicates dataset (dup) with the list of values of index1 (including duplicates) with non-unique values of index2 (dataset select):

```plaintext
proc sort data=dup;
   by index1;
run;

data simple(keep=id index1 index2 cluster_number2 rename=(cluster_number2=cluster_number)) complex;
   merge dup(merge=a) select(merge=b);
   by index1;
   if a and b then output complex;  *a: dupes, b: non-unique index2;
   else output simple;
   drop cluster_number;
run;

The following records (plus thousands or millions more in real life) from dup go into the dataset complex:

<table>
<thead>
<tr>
<th>index1</th>
<th>index2</th>
<th>cluster_number</th>
<th>cluster_number1</th>
<th>cluster_number2</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>1</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>E</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>E</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

The other records from dup go into simple.

```

Dataset simple

<table>
<thead>
<tr>
<th>index1</th>
<th>index2</th>
<th>cluster_number</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>F</td>
<td>7</td>
<td>6</td>
</tr>
</tbody>
</table>

Step 7: The iterative process of collapsing overlapping dupe groups

Combine overlapping dupe groups by collapsing the dataset complex (represented by the macro variable &m_complex) until the number of dupe groups r(clusters) stays the same from one iteration to the next. The iterative process is controlled by a %do %while loop whose condition is met as long as the group count (number of dupe groups) changes from one iteration to the next. Each of the macro variables &d1, &d2 and &d3 is incremented by +1 in each iteration. The macro variable &m_cluster_field stands in for cluster_number.
The first iteration of the macro collapses dupe groups based on different values of index2 that have records with the same index1.

The macro variables &cluster_field.&d1, &cluster_field.&d2 and &cluster_field.&d3 resolve to cluster_number1, cluster_number2 and cluster_number3, respectively. For the first record in each by-group based on cluster_number1 (there are three), cluster_number3 takes the value of cluster_number2. For other members of cluster_number1 by-groups (here, only the second “2” for cluster_number1), cluster_number3 takes the value assigned to cluster_number3 for the preceding record. In other words, for each cluster_number1 by group, all values of cluster_number3 are the same (for the second and third record, cluster_number3 = 4) and are assigned the value of the first cluster_number2 for that by group (which is 4 in the second record). In essence we are saying that two dupe groups on index2 that overlap on index1 should have the same number for some new cluster field based on index2. And that is what the macro does in the first iteration:

<table>
<thead>
<tr>
<th>index1</th>
<th>index2</th>
<th>cluster_number1</th>
<th>cluster_number2</th>
<th>cluster_number3</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>E</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>E</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>B</td>
<td>1</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

When the loop ends, groupcount is 3, i.e., the number of unique values in cluster_number1 (1, 2, 4).

This exhausts the possible collapsing of groups based on index1 by itself. Intuitively, we see that we could complete the collapsing based on the presence of index1=E in dissimilar index2 dupe groups (groups 1 and 2), but if we merely changed the index2 value in the second row to 2 or the index2 value in the second row to 1, or did the same with cluster_number2, the game would be over because we would have no basis to bring the first record (index1 = A) out of the cold. So, instead, we use the new cluster_number3 to record the collapse of the two groups with E, but retain the “memory” of the connection between the first and third records through cluster_number2.

In the second iteration we add a new cluster_number variable (cluster_number4). The macro variables &cluster_field.&d1, &cluster_field.&d1 and &cluster_field.&d1 now resolve to cluster_number2,
cluster_number3 and cluster_number4, respectively. By-grouping is based on cluster_number2, instead of cluster_number1 (which is gone). One can see at a glance that this iteration will not be the last one:

<table>
<thead>
<tr>
<th>index1</th>
<th>index2</th>
<th>cluster_number2 (used for sort)</th>
<th>cluster_number3</th>
<th>cluster_number4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>E</td>
<td>2</td>
<td>1</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>E</td>
<td>1</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>B</td>
<td>1</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

There are 2 cluster numbers for the new field (cluster_number4), which is the same number as the last iteration created in cluster_number3, but the value of groupcount, which is the number of unique values in &clustenumberfield.&d1 (cluster_number2 in this iteration), goes down from 3 to 2. The next iteration will be allowed to proceed because the value of groupcount has changed.

In the third iteration, the cluster variables are cluster_number3, cluster_number4 and the newly minted cluster_number5. By-grouping for cluster_number3 now controls whether cluster_number5 takes (1) the value of cluster_number4 or (2) the value of cluster_number5 in the preceding record. The second, third and fourth records all belong to the same cluster_number3 by-group, so the value of cluster_number5 is the same for all of them. The value of cluster_number5 for the first record (A,2) comes from cluster_number4.

<table>
<thead>
<tr>
<th>index1</th>
<th>index2</th>
<th>cluster_number3 (used for sort – no change)</th>
<th>cluster_number4</th>
<th>cluster_number5</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>E</td>
<td>1</td>
<td>4</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>E</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>1</td>
<td>4</td>
<td>4</td>
<td>1</td>
</tr>
</tbody>
</table>

Obviously, when all values for cluster_number5 are the same (which happens only because our example has so few records) iteration should stop. But what really stops the iterations is the fact that groupcount = 2, which is the number of unique values of cluster_number3 is 2, which equals the previous groupcount. All dupe groups overlapping on index1 or index2 have now been identified.

What have we really done?

Iteration 1: Collapse cluster_number1 (3 \(\rightarrow\) 2 dupe groups) based on cluster_number2. This leaves us with an orphan group problem: If we view the second, third and fourth records as being one group with index2=1 and index1=E (the connecting values), then A,2 is left out even though we can see it belongs in the group. This condition is reflected in the values in cluster_number3. The groupcount, based on cluster_number1, is 3.

Iteration 2: We try to collapse cluster_number2 based on cluster_number3, but we can’t, since cluster_number3 reflects the orphan condition of A,2 as well as the connectedness of records 2, 3, and 4. But in this iteration we create cluster_number4, which reflects the state of affairs with respect to index2 and which provides a bridge via the second record (E,2) through which we will reach A,2 in the next round. The groupcount, based on cluster_number2, is 2.

Iteration 3: Collapse cluster_number3 based on cluster_number4, which provides the bridge needed to bring A,2 into the group with E,2, which already is in the same group with E,1 and B,1. This is reflected in cluster_number5. The groupcount, based on cluster_number3, is 2.

Iterations stop.

Step 8: Bring all the records (dupes and non-dupes) back together

1. Create the final &cluster_field (by renaming cluster_number5 as cluster_number).
2. Append the non-overlapping dupe group dataset simple to the final version of complex, (identified by the macro variable &complex).
3. Append the non-duplicate records (dataset nodup).

4. Merge the combined dupe group dataset (complex) back into the original source dataset (source_data) from which we extracted the fields we needed for dupe group analysis. Recall that we had stripped off fields not needed for this stage of processing to save on I/O time (I/O being very piggish). The cluster_number in each record in source_data is the link between records in source_data and each dupe group.

These four steps are shown below:

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

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

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

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

After resolving dupe groups: Combining the records in each dupe group into a single record for each dupe group

All that we’ve done up to this point is attached a cluster number to each record, tagging records that are part of the same (resolved) dupe group – that is, records that may not be identical but are deemed to represent the same person – with the same cluster number. Of course, we finally have to combine the information from each record within each dupe group, so the final number of records is equal to the number of dupe groups plus the number of records not included in any dupe group.

We will not describe the record combining 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. The priorities – that is, the order of “trust” in the form of each name, address, etc. appearing in the different vendor lists – is a critical determination which is implemented in the programming for this step. But such prioritizing may involve exceptions. We may assign records from vendor A higher priority than records from vendor B, except where vendor B’s record contains a certain item of information that tells us that the information from vendor A is suspect. So, depending on your business rules, the resulting data step may be very complicated, but it does not involve anything beyond a lot of detail and routine Base SAS techniques (or, as the Penguin said, a lot tape and little patience, or was that a little tape…?).

**Conclusion**

We have employed SAS-DQ and created an algorithm in Base SAS to combine overlapping duplicate groups based on two different sets of criteria. Being able to use two sets of criteria increases the accuracy of fuzzy matching. The SAS-DQ product would be enhanced by the addition of a procedure to join dupe groups.
Acknowledgement: SAS® and all other SAS Institute Inc. product or service names are registered trademarks of SAS Institute, Inc. in the USA and other countries. ©indicates USA registration.

REFERENCES

3 The use of the term “index” for the root of the variable names “index1” and “index2” should not be confused with the concept of indexing in managing databases.
4 There might be more than one record for an individual from one of the sources if: (1) there could be duplications, e.g., obsolete records that had been updated but not yet purged by the source vendor, or (2) one of the sources actually provided three separate lists from three different types of records, with overlap between two of the three almost assured. So, in fact we were dealing with five lists, not three.