

FUZZY MERGES: Examples and Techniques

Malachy J. Foley

University of North Carolina at Chapel Hill, NC

ABSTRACT

Frequently SAS® programmers must merge files where the values of the key variables are only approximately the same. Merging on names with approximately the same spelling, or merging on times that are within three minutes of each other are examples of these kinds of merges. These merges are often called fuzzy merges. This tutorial examines the realm of fuzzy merges. It looks at relevant techniques and gives examples of actual programs that produce fuzzy merges.

INTRODUCTION

Fuzzy merges appear in the real world quite often. They come in all shapes, sizes and disguises.

This paper examines four such merges. First, it looks into a merge on approximate times. Then, it explores a merge on the most recent occurrence by date. Finally, it delves into phonetic merging and merging on names.

This article is for anyone who has at least one year of SAS BASE experience and is familiar with match merging. On finishing this paper, you will have seen many fuzzy-merge techniques and should have a basic understand of how fuzzy merges work.

MERGING

File merging, in computing, has been around for a long time. In fact, it was around in 1888 when Herman Hollerith invented computer cards (also know as IBM cards or Hollerith cards).

Merging is an operation performed on two or more files, sorted in a given order, to create a single file in the same order. There are many types of merges. There are merges involving computer cards and electronic files. There are one-to-one merges, match-merges, and fuzzy-merges. Furthermore, there are many kinds of fuzzy-merges. While merging often seems simple, in reality it is a large and complex topic.

Merging is too large a topic for just one paper. For instance, the author wrote two prior papers on match-merges alone. This paper, on the other hand, considers just a few examples of fuzzy merges.

Fuzzy merging is more demanding than match merging. Match-merging usually is easily performed with SAS's match-merge facility. However, there is no such facility for fuzzy merges. Fuzzy merges are more of an art than a science. They often draw on all of a

programmer's knowledge and the best of his or her analytical skills.

A WARNING ABOUT MERGING

Before looking at fuzzy merges, be warned that merges are tricky. In the paper "MATCH-MERGING: 20 Some Traps and How to Avoid Them" the author outlines 28 traps associated with match-merges. Many of these traps can go undetected and cause unexpected results.

Most of these 28 match-merging traps apply to fuzzy merges. For example, to do a fuzzy merge one must make sure that the key variables have the correct case (upper and lower), length and justification. Also, one must develop strategies for handling missing values in key variables.

In a similar fashion, most of the recommendations to avoid traps in match merging apply to fuzzy merges as well. For instance, one should keep the merge as simple as possible. Perhaps the most important recommendation is to avoid manipulating ANY input variable in ANY way.

It is beyond the scope of this paper to describe fuzzy merge traps. This paper merely looks at how to do fuzzy merges, while trying to employ good programming practices that avoid merging traps.

FUZZY TIME MERGE

To start the exploration of the world of fuzzy merges, consider the following two files:

Exhibit 1: Input Files to Fuzzy Merge

INPUT FILE ONE			INPUT FILE TWO		
TIME	NOTE	INT	TIME	NOTE	INT
9:32	101	2.3	9:34	201	2.2
9:56	102	1.4	9:36	202	0.1
10:53	103	0.7	9:59	203	1.5
11:59	104	1.8	11:04	204	0.1
13:00	105	0.1	12:02	205	1.8
14:02	106	2.3	14:01	206	2.4
16:00	107	2.9	14:59	207	0.1
16:12	108	4.2	15:59	208	2.9

In the exhibit, NOTE stands for note number and INT is a variable for intensity. There are other variables in each of the files, like DIRECTION, which for simplicity are not shown.

Each of the above files represents data collected at a different site. The data is on the same phenomena, for instance on earthquakes or forest fires. In the example,

notes 101 and 201 are about the very same quake or forest fire.

In a perfect world, notes 101 and 102 would have an ID number that would uniquely identify the quake or fire. At the very least, one would hope that the quake or fire would have exactly the same time. But for some reason the times are slightly different. (The times can be slightly different due to a difference in the clocks at the two sites. Or the times can be off because the forest fire was sited at different times by the viewers at the two sites, or the vibrations of the earthquake arrived at different times at the two sites.)

Because of the slight difference in times, the task is to match records from the two files where the time is within 5 minutes of each other. A further requirement is that each input record can only be outputted once. Thus, if three records are all within 5 minutes of each other, the first two records should be combined and the third deleted. The desired results are:

Exhibit 2: Desired Fuzzy Merge Results

TIME1	TIME2	NOTE1	NOTE2	INT1	INT2
9:32	9:34	101	201	2.3	2.2
9:56	9:59	102	203	1.4	1.5
11:59	12:02	104	205	1.8	1.8
14:02	14:01	106	206	2.3	2.4
16:00	15:59	107	208	2.9	2.9

Notice that the desired results show that a final requirement for the merge is to output all of the original values from both input files. Thus, the input variables need renaming.

Obviously, the desired results are unattainable with a match-merge. So, how can these files be combined using SAS BASE. One solution to combining these files comes from the past.

OLD-TIME MERGES

In ancient computing times, when files consisted of decks of computing cards, match merging was a multi-step process. The cards, of course were records, which today are also known as observations or rows. The process for merging the two decks or files was: (1) concatenate the files (physically stack one deck of cards behind the other); (2) use a machine to sort the cards on the key fields (after the sort, all the cards for one ID were physically next to each other); (3) feed the cards into the computer; and (4) merge the data from all the cards for a given ID into one record in the computer's memory. (The resulting merged data could then be written from the memory to magnetic tape or some other output media.)

For old-time merges there typically was a field on each card to identify the card type. In other words, there was a field to identify the input file (deck) from which the card came.

This same technique, with appropriate adjustments for electronic records and SAS, can serve to perform the fuzzy merge outlined in exhibits 1 and 2.

STEPPING THROUGH A FUZZY MERGE

One or two DATA Steps suffice to do the current fuzzy merge. However, the fuzzy merge is a fairly complex concept. It involves many elements. The next sections of the tutorial examine these elements from varying points of view. At the end, everything is put together, and all the elements will be combined into a single DATA Step.

To get started, remove the INT field from exhibit 1:

Exhibit 3: Simplified Version of Exhibit 1

INPUT FILE ONE		INPUT FILE TWO	
TIME	NOTE	TIME	NOTE
9:32	101	9:34	201
9:56	102	9:36	202
10:53	103	9:59	203
11:59	104	11:04	204
13:00	105	12:02	205
14:02	106	14:01	206
16:00	107	14:59	207
16:12	108	15:59	208

The INT field was removed from the example to simplify it and to make the exhibits that follow easier to view. Of course, files ONE and TWO can have any number of variables. No matter how many variables the input files have, the fuzzy merge technique remains the same.

Observe there is already a field in each file that identifies the file. Specifically, the first digit of NOTE field is a 1 or a 2 which corresponds to the file name.

Again, the old-time card merge is the model for the current fuzzy merge. As such, the next exhibit applies steps one and two of the old-time merge to the input files. These steps are to concatenate the files and then sort them on TIME.

Exhibit 4: Concatenate/Sort the files

```
DATA CONCAT;
  SET ONE TWO;
RUN;
PROC SORT DATA=CONCAT OUT=CAT_SORT;
  BY TIME;
RUN;
```

INPUT FILE ONE		CONCAT		CAT_SORT	
TIME	NOTE	TIME	NOTE	TIME	NOTE
9:32	101	9:32	101	9:32	101
9:56	102	9:56	102	9:34	201
10:53	103	10:53	103	9:36	202
11:59	104	11:59	104	9:56	102
13:00	105	13:00	105	9:59	203
14:02	106	14:02	106	10:53	103
16:00	107	16:00	107	11:04	204
16:12	108	16:12	108	11:59	104
		9:34	201	12:02	205
		9:36	202	13:00	105
		9:59	203	14:01	206
		11:04	204	14:02	106
		12:02	205	14:59	207
		14:01	206	15:59	208
9:34	201	14:59	207	16:00	107
9:36	202	15:59	208	16:12	108
...					

Remember the goal is to match records from the two input files where the TIME is within 5 minutes of each other. Notice how the times in the CAT_SORT file are in sort order. Thus, the records that require merging are now next to each other in the output file. We are now ready to read the records with another DATA Step and match the ones that are within 5 minutes of each other.

INTERLEAVING

However, before the merge is done, a simpler way to concatenate and sort the two files is examined. In the old days, concatenating and sorting was the only way to interleave cards. Now, with electronic files and SAS, the two processes can be combined into one. SAS calls the combined process interleaving. The SAS code for interleaving follows. Basically, the code consists of a SAS concatenate (SET ONE TWO;) with a BY statement. To do an interleave, SAS requires the two input files to be sorted on the BY variables. The following exhibit illustrates interleaving.

Exhibit 5: Interleaving files ONE & TWO

DATA INTER;					
SET ONE TWO;					
BY TIME;					
RUN;					

INPUT		INPUT		OUTPUT	
FILE ONE		FILE TWO		FILE INTER	
-----		-----		-----	
TIME	NOTE	TIME	NOTE	TIME	NOTE
-----		-----		-----	
9:32	101	9:34	201	9:32	101
9:56	102	9:36	202	9:34	201
10:53	103	9:59	203	9:36	202
11:59	104	11:04	204	9:56	102
13:00	105	12:02	205	9:59	203
14:02	106	14:01	206	10:53	103
16:00	107	14:59	207	11:04	204
16:12	108	15:59	208	11:59	104
				12:02	205
				13:00	105
				14:01	206
				14:02	106
				14:59	207
				15:59	208
				16:00	107
				16:12	108

Notice that the file INTER is exactly the same as the file CAT_SORT from the example given in exhibit 4.

FANCY INTERLEAVING

As hinted earlier, knowing what input file the current record is coming from is essential in the old-time merges. Similarly this knowledge is useful in the current fuzzy merge example. As such, the next refinement to the program includes the creation of a field called TYP. TYP refers to the current input record type. There are a variety of ways to calculate this field. For instance, observe the following code.

```
TYP=SUBSTR(NOTE, 1, 1);
```

However, in general, one does not have the record type included in one of the input variables. On the other hand, one can always use the IN= data set option to determine the record type.

Also, part of the merge specification was to rename our input variables. So when the NOTE and TIME variables come from file ONE, their names are NOTE1 and TIME1. Likewise, when NOTE and TIME come from file TWO, their names are NOTE2 and TIME2. One way to achieve this name change is via the IN= data set option.

The next example uses the same interleave technique described in the previous exhibit, but adds the new output variables.

Exhibit 6: Interleave Plus 5 Variables

DATA INTER2 ;							
SET ONE (IN=IN1) TWO ;							
BY TIME;							
TYP=2-IN1; *CREATE CURRENT RECORD TYPE;							
* GETTING OUTPUT VARIABLES;							
IF IN1 THEN TIME1=TIME; ELSE TIME2=TIME;							
IF IN1 THEN NOTE1=NOTE; ELSE NOTE2=NOTE;							
RUN;							

FILE INTER2							

TIME	NOTE	TYP	TIME1	TIME2	NOTE1	NOTE2	

9:32	101	1	9:32	.	101	.	
9:34	201	2	.	9:34	.	201	
9:36	202	2	.	9:36	.	202	
9:56	102	1	9:56	.	102	.	
9:59	203	2	.	9:59	.	203	
10:53	103	1	10:53	.	103	.	
11:04	204	2	.	11:04	.	204	
11:59	104	1	11:59	.	104	.	
12:02	205	2	.	12:02	.	205	
13:00	105	1	13:00	.	105	.	
14:01	206	2	.	14:01	.	206	
14:02	106	1	14:02	.	106	.	
14:59	207	2	.	14:59	.	207	
15:59	208	2	.	15:59	.	208	
16:00	107	1	16:00	.	107	.	
16:12	108	1	16:12	.	108	.	

Please that INTER2 is exactly the same as the INTER, except there are five more variables in the new file.

The prior code calculated the record type using (TYP=2-IN1;). This code produces a value of 1 when the input record is from file ONE and a value of 2 when the input record is from file TWO. A more intuitive way to determine the value of TYP is

```
IF IN1=1 THEN TYP=1; ELSE TYP=2;
```

Note that all the output variables are available in INTER2. Moreover, the records that need merging are next to each other in the file. For example, note number 101 is within 5 minutes of note number 102. As a result, these two records should be merged. But how do you get note 101 data together with note 102 data. There is an amazingly easy trick. You add a RETAIN statement to the previous example and presto:

Exhibit 7: Interleave, 5 Vars, & a RETAIN

```
DATA INTER3;
  SET ONE (IN=IN1) TWO ;
  BY TIME;
  RETAIN TIME1 TIME2 NOTE1 NOTE2;
  TYP=2-IN1; *CREATE CURRENT RECORD TYPE;
  * GETTING OUTPUT VARIABLES;
  IF IN1 THEN TIME1=TIME; ELSE TIME2=TIME;
  IF IN1 THEN NOTE1=NOTE; ELSE NOTE2=NOTE;
RUN;
```

FILE INTER3						
TIME	NOTE	TYP	TIME1	TIME2	NOTE1	NOTE2
9:32	101	1	9:32	.	101	.
9:34	201	2	9:32	9:34	101	201
9:36	202	2	9:32	9:36	101	202
9:56	102	1	9:56	9:36	102	202
9:59	203	2	9:56	9:59	102	203
10:53	103	1	10:53	9:59	103	203
11:04	204	2	10:53	11:04	103	204
11:59	104	1	11:59	11:04	104	204
12:02	205	2	11:59	12:02	104	205
13:00	105	1	13:00	12:02	105	205
14:01	206	2	13:00	14:01	105	206
14:02	106	1	14:02	14:01	106	206
14:59	207	2	14:02	14:59	106	207
15:59	208	2	14:02	15:59	106	208
16:00	107	1	16:00	15:59	107	208
16:12	108	1	16:12	15:59	108	208

One of the cardinal rules of avoiding traps in merging is to never manipulate or use an input variable in ANY way. This rule also applies to any file-combining process, such as an interleave. Please note that the RETAIN statement was applied to derived variables and not to input variables.

ONE-STEP FUZZY MERGE

Up until now, all that has been done is to interleave (concatenate and sort) the two input files. To the basic interleave, 5 derived variables and RETAIN statement were added. While this is a simple process, look at what has been achieved. Essentially, the fuzzy merge is complete.

As previously mentioned, the first two notes that require a merge are 101 and 102, because they are within 5 minutes of each other. These two notes are already merged in the second record above! The first record, of course, needs to be rejected. The fuzzy merge now comes down to which of the previous records we want to keep and which we want to reject or delete.

Remember the rules for matching (accepting or rejecting) records in this case. They are (1) to match records from different files (2) where time is within 5 minutes (or 300 seconds) of each other. (3) Each input record is to be outputted only once. Thus, when three records are all within 5 minutes of each other, combine the 1st two records and delete the third. (4) The output variables should be TIME1, NOTE1, TIME2 and NOTE2.

The fuzzy merge consists of taking the prior code and applying rules 1 to 3 to it.

Exhibit 8: One-Step Fuzzy Merge

```
DATA FUZZY (KEEP=TIME1 TIME2 NOTE1 NOTE2);
  SET ONE (IN=IN1) TWO ;
  BY TIME;
  RETAIN TIME1 TIME2 NOTE1 NOTE2;
  TYP=2-IN1;
  * GETTING OUTPUT VARIABLES;
  IF IN1 THEN TIME1=TIME; ELSE TIME2=TIME;
  IF IN1 THEN NOTE1=NOTE; ELSE NOTE2=NOTE;
  * RULES;
  IF (ABS(TYP-lastTYP)) =1 AND
     (ABS(lastTIME-TIME) <=5*60) AND
     lastFLAG ne 1 THEN FLAG=1;
  * REMEMBERING VALUES AND OUTPUTTING;
  RETAIN lastTIME lastTYP lastFLAG;
  lastTIME=TIME;
  lastTYP =TYP;
  lastFLAG=FLAG;
  IF FLAG THEN OUTPUT;
RUN;
```

INPUT FILE ONE		INPUT FILE TWO	
TIME	NOTE	TIME	NOTE
9:32	101	9:34	201
9:56	102	9:36	202
10:53	103	9:59	203
11:59	104	11:04	204
13:00	105	12:02	205
14:02	106	14:01	206
16:00	107	14:59	207
16:12	108	15:59	208

FUZZY			
TIME1	TIME2	NOTE1	NOTE2
9:32	9:34	101	201
9:56	9:59	102	203
11:59	12:02	104	205
14:02	14:01	106	206
16:00	15:59	107	208

Again, in the above code, the cardinal rule of avoiding traps is used. Namely, all the new code was applied only to derived variables and not input variables.

The new code merely implements the rules.

The FLAG variable indicates if a record is to be outputted. FLAG=1 means to output the record. If FLAG has a value other than one, the record is not outputted.

Whether a record is kept depends on the rules. For example, rule 1 is to match records from different files. "IF (ABS(TYP-lastTYP))=1" is the code for rule 1. It checks that current record type is exactly 1 type different from the previous record type.

It is tempting to use (TYP ne lastTYP) to perform the check for rule 1. This code works for most situations. Notwithstanding, it does not work for note 101. For note 101, (lastTYP=.) and (TYP=1). Since for these values (TYP ne lastTYP) is true the computer will output a record for note 101 without a corresponding merged record. So, it is better to check that the difference in record types equals exactly 1.

Rule 2 states that the time on the two records should be within 5 minutes (or 300 seconds) of each other. $(ABS(lastTIME-TIME) \leq 5*60)$ performs this task. The times in the example are SAS times. SAS times are measured in seconds. As such, the code checks that the difference in times is less than 5 times 60 or 300 seconds.

The code in exhibit 8 assumes that all the input values are non-missing. If there are missing values for time, the rules and code should change accordingly. For instance, if you did not want to merge records with missing values for time, the following code might be appropriate.

```
(0 <= ABS(lastTIME-TIME) <= 5*60)
```

Rule 3 is to output each input record only once. The code $(lastFLAG \neq 1)$ makes this happen. This code says that if the previous record was used in a merge, it can not be merged with the current record.

FUZZY-MERGE DESIGN CONSIDERATIONS

The previous example highlights the need to consider a series of design questions before even starting to program a fuzzy merge. The following table lists some such questions.

Exhibit 9: List of Design Considerations

- ```

- What are your key fields.
- What is the format of the data in the key
 fields.
- What are the criteria for matching 2 records
- How to insure you match records from
 different files.
- How to handle:
 - missing values
 - input Records that don't match
 - input records that match more than once
 - input record type
- What are your output variables.
- What does your output file look like.
- How to rename retained variables.

```

These design questions are implicit in almost every fuzzy merge. As such, exhibit 9 serves as a checklist of questions to consider before even attempting a fuzzy merge.

Aside from the questions in the above exhibit, it is good to observe how the fuzzy merge was accomplished. Namely, it had three components. First, it used an interleave. Second, it used retain values on non-input variables. Third, it used a set of rules to determine when input records were to be merged.

This same three-part technique and the design questions are used in the following example.

## ANOTHER FUZZY MERGE

In the previous fuzzy merge, two files were merged on a time variable where that variable was inexact or fuzzy. That was just one example of a fuzzy merge. There are

many other fuzzy problems that programmers confront. This section looks at another one of them.

This case involves merging by date order. More specifically, there are two input files. One file comes from the student registrar office. This file contains all the different registration statuses for each student. The second file contains the student's test results. Both files have date fields that are SAS dates. The task is to merge each test result with the MOST CURRENT registrar data. Here are the two input files.

Exhibit 10: Input Files

| REGISTRAR FILE |    |          | TEST FILE |      |          |
|----------------|----|----------|-----------|------|----------|
| ID             | GR | UP_DATE  | ID        | TEST | UP_DATE  |
| A01            | 3  | 08/12/96 | A02       | 96   | 09/12/96 |
| A02            | 4  | 09/01/96 | A02       | 87   | 10/06/96 |
| A02            | 4  | 05/12/97 | A03       | 76   | 10/17/96 |
| A04            | 3  | 08/16/96 | A04       | 79   | 03/29/97 |
| A04            | 4  | 08/16/97 | A04       | 83   | 10/14/97 |
| A04            | 4  | 12/11/97 |           |      |          |

While each student has an ID, the records from the two files can not be matched by ID alone. They must also be merged on the update date. However, these dates seldom match. The goal of this merge is to match each test record with the MOST RECENT registrar record.

To get a handle on what is required, the two files are interleaved on the ID and date.

Exhibit 11: Interleaving 2 Files

```

DATA INTERLV;
 SET REG TST;
 BY ID UP_DATE;
RUN;

```

| INTERLV |          |    |      |
|---------|----------|----|------|
| ID      | UP_DATE  | GR | TEST |
| A01     | 08/12/96 | 3  | .    |
| A02     | 09/01/96 | 4  | .    |
| A02     | 09/12/96 | .  | 96   |
| A02     | 10/06/96 | .  | 87   |
| A02     | 05/12/97 | 4  | .    |
| A03     | 10/17/96 | .  | 76   |
| A04     | 08/16/96 | 3  | .    |
| A04     | 03/29/97 | .  | 79   |
| A04     | 08/16/97 | 4  | .    |
| A04     | 10/14/97 | .  | 83   |
| A04     | 12/11/97 | 4  | .    |

From this listing, it is obvious that second record should be merged with the third. To do this, a RETAIN statement is in order. But remember we can not retain GR because that is an input variable. So to retain the grade value, another variable should be created. To this end, a variable called GRADE will be created.

Also this listing shows, via student A03, that it is possible to have a test record without a corresponding registrar record. Taking all of this in consideration, we can perform the fuzzy merge of adding the most recent registrar data to each test record. This fuzzy merge is demonstrated in the following Exhibit.

Exhibit 12: Merging the Most Recent Registrar Record to the Test Records.

| REGISTRAR FILE |    |          | TEST FILE |      |          |
|----------------|----|----------|-----------|------|----------|
| ID             | GR | UP_DATE  | ID        | TEST | UP_DATE  |
| A01            | 3  | 08/12/96 | A02       | 96   | 09/12/96 |
| A02            | 4  | 09/01/96 | A02       | 87   | 10/06/96 |
| A02            | 4  | 05/12/97 | A03       | 76   | 10/17/96 |
| A04            | 3  | 08/16/96 | A04       | 79   | 03/29/97 |
| A04            | 4  | 08/16/97 | A04       | 83   | 10/14/97 |
| A04            | 4  | 12/11/97 |           |      |          |

```

DATA MERGE (KEEP=ID UP_DATE GRADE TEST);
SET REG (IN=INR) TST (IN=INT);
BY ID UP_DATE;
* CREATE AND CHECK GRADE;
RETAIN GRADE;
IF FIRST.ID THEN GRADE=.;
IF FIRST.ID and INR=0 THEN
 PUT "*** REG MISSING FOR "ID=";
IF INR=1 THEN GRADE=GR;
IF INT=1 THEN OUTPUT;
RUN;

```

| MERGE FILE |          |       |      |
|------------|----------|-------|------|
| ID         | UP_DATE  | GRADE | TEST |
| A02        | 09/12/96 | 4     | 96   |
| A02        | 10/06/96 | 4     | 87   |
| A03        | 10/17/96 | .     | 76   |
| A04        | 03/29/97 | 3     | 79   |
| A04        | 10/14/97 | 4     | 83   |

In this case, it was possible to have a test record without a corresponding registrar record. To adjust for this possibility, the first.ID variable had to be used to initialize the GRADE variable to missing. In general, it is good practice to initialize all of your derived variables.

Also, it is good practice to report on missing information. In the prior exhibit code was added to this fuzzy merge to follow up on missing registrar records.

```

IF FIRST.ID and INR=0 THEN
 PUT "*** REG MISSING FOR "ID=";

```

This code was, of course, unnecessary to the merge.

There are some reoccurring themes in this illustration. For instance, a KEEP= data set option on the output file is used to subset the variables. Again, a SAS interleave was used to fuzzy merge the records. The SAS interleave was also used to get a handle on how to merge the records in the first place.

A RETAIN statement was used to physically bring the data from the previous interleave record so it can be merged with the data on the current record. A new variable was created in the DATA Step to hold the retained value. The IN= variable was utilized to assign values to be retained variable.

Rules and the OUTPUT statement were used to decide when to match and output records. Or more precisely rules were used to decide which interleaved records to output and which to delete.

The fuzzy merge that uses the interleave allow rules to vary, sometime widely, according to the circumstances. This variation, in turn, allows for a great flexibility in doing fuzzy merges.

Up until now, all the fuzzy merges presented were with input variables whose values never changed. Below, a totally different type of fuzzy merge is scrutinized. Here the fuzzy merge will be performed by recoding the input variables and match merging on the recoded values. One such merge is the phonetic merge.

## PHONETIC MERGING

Another type of fuzzy merge is matching two files on words that sound alike. Surely, the granddaddy of all phonetic merging is the "Soundex" algorithm. Margaret K. Odell and Robert C. Russell invented Soundex. They patented their algorithm in 1918 and 1922. They developed Soundex to find sir names that sound alike.

Soundex is typically used to find one person in a customer database. For example, if a customer calls about an existing airline reservation, the person answering the phone can ask for the customer's last name. Suppose that the airline person hears the customer say his name is "Johnson". "Johnson" is then typed into the computer. A Soundex search of the reservations would yield Johnson and phonetically similar last names (like Johnsen, Jonson, Jonsen, Johanson, etc.). In this example, the airline avoids asking how to spell the last name. Thus, the airline saves time and money, and the customer receives better service.

Soundex is not infallible. Occasionally, it misses similar names. For instance, Soundex does not match Rogers and Rodgers. On the other hand, it sometimes matches dissimilar last names like Conygham and Cangemi. But all in all, Soundex does a good job of matching sir names.

Soundex is described on pages 391-392 of Knuth and on page 388 of Hall (see References). Fundamentally, it is an algorithm that converts any string of letters into a code that consists of one letter and 3 digits. For example, "Euler" converts to "E460". This code is called the Soundex code. There are 26,000 possible Soundex codes. The codes are created as follows.

Exhibit 13: The Soundex Algorithm.

- ```

-----
- The 1st letter of the code=1st letter of name.
- Use the remaining letters to get the 3
  digits as follows:
  - Remove the letters A E I O U Y H & W.
  - Convert the letters B F P & V to 1.
  - Convert C G J K Q S X & Z to 2.
  - Convert D & T to 3; and L to 4
  - Convert M & N to 5; and R to 6.
- Keep three digits (pad right with zeroes)
-----

```

Soundex is implemented in SAS as the =* operator. See page 502 of *SAS Language: Reference* for a description of this operator. Below is a sample SAS

program that implements the airline reservation example presented earlier in this section.

Exhibit 14: Example of Using Soundex

```
-----
DATA SUB_TEST;
  SET TEST;
  WHERE NAME="JOHNSON";
RUN;
-----

-----
FILE TEST      FILE SUB_TEST
-----
NAME           NAME
-----
Jackson        Johnson
Jennings       Johnson Jr.
Johnson       JOHNSEN
Johnson Jr.   Jon son
Johnsen       Jon-son
JOHNSEN       JONSEN.
Jon son       johanson
Jon-son       Johanson
JONSEN.       Jonsson
JOHANSON      Jonzon
Johanson
Joyner
.Jonson
Jonsson
Jonzon
```

Notice how the SAS version of the algorithm is fairly robust. To start with, it is case insensitive. Furthermore, it ignores leading blanks, embedded blanks and embedded punctuation. However, it does not ignore leading punctuation (See . Jonson above).

In the example, Soundex ignores "Jr." by luck. In the particular case of "Johnson", it just happens that the algorithm ignores everything after the last name. This is not the case with shorter sir names. Moreover, Soundex requires left-justified names. (See "Johnsen" in exhibit 14.) In general, it is a good idea to clean and/or check your character data before attempting any fuzzy merge. A later section in this paper demonstrates some techniques for cleaning character data.

Page 40 of the August 19,1991 issue of *InformationWeek* details a modified version of Soundex. This version creates a code that consists of one letter followed by five digits.

In short, Soundex was designed a long time ago to find sir names that sound the same as pronounce in English. By its very design, it is limited in finding phonetic matches. A more modern and a more general phonetic matching algorithm is presented by Lawrence Philips (see References). Philips' technique is probably better than Soundex for generalized phonetic matching.

MATCHING ON NAMES

When discussing fuzzy merges, one topic inevitable comes up. It is how do you merge two files based on a person's name. This question arises frequently in the health, marketing, and insurance industries. The reason for this is that, at first glance, the name field may seem to be the only identifying field common to the files you need to merge.

However, matching records on names is almost an oxymoron because it is impossible to merge two files on names alone. Names by their very nature are ambiguous. For example, the author opened up his local small-town phone book at random. On that random page and for just one sir name, there were 3 Amy's, 2 Cynthia's, 9 Michael's, 2 Richard's, and 2 William's! Obviously, just knowing someone's name is often not enough to even find them in a small-town phone book.

In practice, the files that require merging usually cover much more than one small town. Often they cover a State or the entire country. So, to match people you need more information than just their name.

The first design question then is, aside from the name, what other variables do you have to match on. Do you have birth date, birthplace, mother's maiden name, current zip code, current address, current phone, etc. You are looking for anything that will distinguish one person from the next in you input files.

After you examine the fields on the input files, you may not even need the name field to match people. Or you may not need the both names or the whole name field. For example, if you are trying to match adult males and you have date of birth and current phone number in both input files, you are home free. An exact merge on those two fields alone will very nearly match all but adult twins living at the same address.

If you must use names to match your files, you need to know your data very well. All name fields are not alike. PROC FREQ's and PRINT's on a sorted subset of the data are helpful in becoming familiar with the data. The following table lists some questions that show how name fields can differ. It also lists some design questions, beyond those given in exhibit 9, that need consideration in name matching.

Exhibit 15: Questions Regarding Name Matching

- ```

- Are all names in upper case letters.
- Are all names left justified. Why not.
- Are last names separate from first names.
- If not separate, can you separate them.
- Are full names always given as first names
 or are nicknames used.
- Do you have middle initials (for everyone).
- Do you have hyphenated names.
- Do you have leaders like J. Patrick Foley.
- Do you have trailers like "Jr." & "III".
- Do you have unwarranted embedded blanks
 or punctuation.
- How similar are your different sources.
- How many false matches can you tolerate.
- How many misses can you tolerate.
- Do you need to label the input records
- Will you flag the output records as to how
 the match was made.
- Are you have phonetic error, source errors,
 keying errors and/or spelling errors.

```

This list illustrates how complicated and different name matching can be. Space limitations do not allow this paper to treat name merging thoroughly. However, a general discussion of the problems and techniques used is possible and ensues. At the outset one must say that most merges on names are different and require different combinations of techniques to match them.

Names in a computer database almost always initially come from a paper form. These forms are then usually keyed into a computer. The names on the forms are often written by the persons themselves, or by an interviewer who asks the person to spell their name. Occasionally, a mother or father will write the name for a child. In all these cases, the spelling of the name is probably correct. Seldom are names written phonetically.

While the spelling on the form is usually correct, names are prone to contain keying errors. For example, the author's first name, Malachy, is almost always correctly spelled on the mail he receives. The two exceptions are Malaclay and Malachi. The Malaclay variation came from some keyer who mistook the "h" in the name for a "la". The Malachi variation surely comes from a keyer who recognized "Malachi" as a valid name and assumed that the "y" was a mistake.

One method that is often touted as a way to match names is the Soundex algorithm. But as discussed in the previous section, Soundex was intended for phonetic matching of last names. Soundex was never intended to find keying errors or nicknames. So, it is not surprising that Soundex does not work well with matching names.

In matching names, the programmer is seldom up against spelling errors or phonetics. More often he is up against keying errors, nicknames, hyphenated names, trailers (suffixes), leaders (prefixes), maiden names, etc.

## STARTING THE NAME MERGE

So how does one merge on names. As with exact merges, one should start with making sure the name fields in the input files are all the same case and left justified. This can be easily attained with the UPCASE and LEFT functions. Also the field LENGTHS should usually be the same. Corresponding fields should be shorted to the smallest length using the LENGTH and assignment statement. The merge should be kept as simple as possible. All these and other techniques of an exact merge are described in the two papers by Foley.

Next, you may want to take stray punctuation out of the name fields. This can be accomplished with the COMPRESS function.

Exhibit 16: Eliminating Stray Punctuation.

```

NAME_C=COMPRESS(LASTNAME,".'*");
```

Here the variable NAME\_C will be the same as the variable NAME except the periods, apostrophes, and asterisks will be removed from the field and the field will be collapsed. Hyphens and blanks were purposely kept so that they can later be used to find different pieces of names in the name fields.

Next, if at all possible, the first name should be separated from the last name and middle initial. Hopefully, this is the way your data was collected. If the names are not separate, a good knowledge of your data

and a combination of the SCAN, SUBSTR and LENGTH functions can probably separate the fields. You may want to keep the periods in your fields during the separation process.

Once the fields are separated, you can then try to match on each of them. If the other fields you are matching on strongly distinguish people, you may not need the whole names to match on. Perhaps the first, middle and last initial will be enough. Notice that there are 17,576 different sets of 3 initials (almost as many different codes as Soundex provides). If you have many missing middle initials, you may want to match on just the first and last initials.

## LAST NAMES

Last names are fairly easy to match on. Last names do not contain nicknames, leaders, or initials. Last names do contain hyphenated names, maiden names, and trailers like Jr. and III. The last name of men almost never changes. Unfortunately, women will alternatively use their husband's last name, their maiden name or a hyphenated name.

To get rid of trailers and names following the hyphen, apply the SCAN function to the compressed field from the previous exhibit.

```
LAST=SCAN(NAME_C,1);
```

How you will handle the changing last name of women, depends on your data. You may already have a maiden name field. You may know that the last names of women in your input files have been collected consistently. You may want to ignore the problem. Or, you may want to try the INDEX function to find the last name of one file within the last name of the second file.

## FIRST NAMES

First names can sometimes be a problem because of nicknames. For example, Catherine can be Cathy, Cathie, Kay, Kattie, Katty, Kathy, Kitty, Kate, Kit, Kaye, etc. Although such variations are possible, they are unlikely on formal documents like birth certificates, death certificates, or bank accounts. For instance, the author knows one family where there is a Katty, Kitty, Catherine, and Kay. The different names are used to know whom you are talking about. Yet, all these people have "Catherine" on their birth certificate. So if you are trying to merge two official sources of data, nicknames probably are not a problem.

On the other hand, some unofficial sources are riddled with nicknames. In these cases, you can use a variety of techniques. The first technique is not to use the first names at all. Perhaps, with the other information you have, you only need a last name to distinguish people. Another possibility is to change all the nicknames to a

formal first name. You can use a can use someone else's list or make your own. For example, Yahoo uses something called "SmartName". If you decide to make your own, you can start by PROC FREQing the first names to see what you are up against.

## FINAL NAME TECHNIQUES

If you are going for match on full names rather than initials, at some point, you will want to try an exact merge. This merge, of course, will be on the name fields as well as the other fields you have to match on (for example date of birth). It is amazing how many matches are made this way.

After you try an exact merge, you need to do something more interesting. The author suggests that you apply a modified form of the Soundex algorithm on the separated and cleaned fields. This code tries to avoid keying errors. The modified algorithm can be called the "Spelldex" algorithm.

As shown above, the Soundex algorithm starts out by assuming that the first letter is correct. This is a reasonable assumption. The first letter is usually a capital letter. When written on a form, it has no letters to its left to interfere with its reading, etc.

Next, Soundex gets rid of vowels beyond the first letter. This is also a good idea for avoiding keying errors. Written vowels usually are lower-case letters that often look approximately the same. Furthermore, since vowel sounds for two different vowels or diphthongs are often similar, the keyer may inadvertently confuse these while keying. (This is what happened with Malachi.)

The final step in Spelldex algorithm is to simply take the first many consonants from the first and/or last name. The author suggests that you take only two letters from the first name and 3 letters from the last name. It is assumed that the first consonants are more often non-missing and non-corrupt than the consonants later on in the name. Since there is a greater variety in the problems with first names than with last, it is suggested that you can use more letters from the last name than the first.

Here is one way the algorithm can be coded for a last name.

```
Exhibit 17: "SPELLDEX"

DATA SPELL;
 SET TEST;
 LENGTH NAME_U NAME_P NAME_2 NAME_S $15;
 NAME_1 $1 SPELLDEX $4;
 NAME_U=UPCASE(LEFT(NAME));
 NAME_P=COMPRESS(NAME_U,"' - . ,");
 NAME_S=SCAN(NAME_P,1);
 NAME_1=SUBSTR(NAME_S,1,1);
 NAME_2=SUBSTR(NAME_S,2);
 NAME_C=COMPRESS(NAME_2,"AEIOUY");
 SPELLDEX=NAME_1|SUBSTR(NAME_C,1,3);
RUN;

```

Exhibit 17 (cont): "SPELLDEX"

| -----<br>SPELL<br>----- |          |          |        |              |
|-------------------------|----------|----------|--------|--------------|
| NAME                    | NAME_P   | NAME_S   | NAME_C | SPELL<br>DEX |
| ACKERS                  | ACKERS   | ACKERS   | CKRS   | ACKR         |
| BAAS JR.                | BAAS JR  | BAAS     | S      | BS           |
| Jackson                 | JACKSON  | JACKSON  | CKSN   | JCKS         |
| Jennings                | JENNINGS | JENNINGS | NNNGS  | JNNN         |
| Johnson                 | JOHNSON  | JOHNSON  | HNSN   | JHNS         |
| Joyner                  | JOYNER   | JOYNER   | NR     | JNR          |
| -----                   |          |          |        |              |

Notice that the Spelldex code consists of 4 letters. These 4 letters offer 208,000 possible codes. While Soundex offers only 26,000 possible codes. As such, Spelldex is more discriminating than Soundex and it gives fewer false matches.

Matching on names is a delicate balancing act. As exhibits 9 and 15 show, a myriad of factors are involved. This section touched on many of those factors. The name merging techniques mentioned in this paper are rather straightforward and easy to implement. If you are interested in other, sometimes more sophisticated, techniques, see Hall's paper given in the reference section of this paper. Hall cites additional works that are relevant to this topic.

## CONCLUSION

This paper investigated a variety of fuzzy merges.

First, it looked at two examples of fuzzy merges that matched records without changing the values of the input variables. It was found that by interleaving the input files and retaining copies of the key variables, one can perform these merges well (see exhibits 8 and 12). Exhibit 9 outlined some design considerations for this type of fuzzy merge.

Second, the paper reviewed phonetic matching and matching on names. These are two examples of fuzzy merges where the key variables were recoded. Then the input files were match-merged on the coded key variables. Exhibit 15 lists design consideration for matching on names.

The paper found that all fuzzy merges require that the programmer know his or her data very well. Also, the programmer needs an appropriate set of tools to carry out the fuzzy merges. This paper examined a variety of such tools and gave examples of how they work.

## REFERENCES

Foley, Malachy J. "Advanced Match-Merging: Techniques, Tricks and Traps" *Proceedings of the Twenty-Second Annual SAS Users Group International Conference* (1997) pp 199-206.

## REFERENCES (Cont.)

Foley, Malachy J. "Match-Merging: 20 Some Traps and How to Avoid Them" *Proceedings of the Twenty-Third Annual SAS Users Group International Conference* (1998) pp 277-286.

InformationWeek, "Soundex: Rules for Mismatches", *InformationWeek* (August 19, 1991) p 40.

Hall, Patrick "Approximate String Matching" *ACM Computing Surveys* (December, 1980) Vol. 12, Num 4, page 388

Knuth, Donald E., *The Art of Computer Programming* (Massachusetts: Addison-Wesley, 1973) Vol. 3, pp 391-312.

Lawrence Philips, "Hanging on the Metaphone", *Computer Language* V7 n12 (December 1990) pp39-43.

SAS Institute, Inc., *SAS Language: Reference*, Version 6, First Edition (Cary, NC: SAS Institute Inc., 1990)

## TRADEMARKS

SAS is a registered trademark or trademark of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

Other brand and product names are registered trademarks or trademarks of their respective companies.

## AUTHOR CONTACT

The author welcomes comments, questions, corrections and suggestions.

Malachy J. Foley  
2502 Foxwood Dr.  
Chapel Hill, NC 27514

Email: FOLEY@unc.edu