Paper 75-27

How to Link Records Matching on at Least *m* Out of *n* Identifying Keys

Xinyu Ji, Fallon Clinic, Worcester, MA

ABSTRACT

This paper discusses how to link records matching on at least *m* out of *n* identifying keys. That can be easily done by first using the SQL procedure to retrieve all the pairs that match on at least *m* identifying keys. However the processing time of Cartesian Product joins by PROC SQL may be prohibitively long for a sizable medical treatment data set. An alternative approach is introduced, which requires more programming efforts, but less processing time.

INTRODUCTION

Client tracking by record linkage is often a crucial step in longitudinal data analysis. Many different linkage methods exist, ranging from the simplest record linking by the exact match on a single identifying key to a variety of more advanced fuzzy record linkage methods. Fuzzy linkage methods, which are usually able to effectively decrease the false negative error ratio, also tend to dramatically increase the computer resources needed to accomplish the task. Sometimes it is so resource-intensive that executing the program is only feasible in theory. In my work to link records matching on at least *m* out of *n* identifying keys, I encountered a case where I had to trade intensive programming efforts for shorter CPU time.

THE PROBLEM

Let us consider the simplest example where we need to link records matching on at least two out of three identifying keys. Suppose we have a data set with 10 observations containing medical treatment information. The variables are ADNUM (Admission Number), SSN (Social Security Number), Name (The first and the third letter of first name and last name), and DOB (Date of Birth in YYMMDD8. Formats). Here is what our raw data set looks like[1]:

ADNUM	SSN	NAME	DOB
101	856116534	ABCD	18760323
187	856176354	ABCD	18760323
233	856176354	ABCD	18670323
456	850176534	ABCD	18760323
490	856176534	ABEF	18670323
535	856176534	ABGF	18760323
601	856176534	ABEF	18760323
632	856176534	ABEE	18760323
879	856116534	ABEF	18760323
911	856176534	ABGF	18760323

Our record linkage criterion is: every two admissions are thought to belong to the same individual, if the two admissions match on at least two of the three identifying keys, namely, SSN, NAME, and DOB. Notice that there is an implicit statement behind this criterion -- for any two records not linked together based on the explicit criterion, if each of them is linked to a third common record based on our criterion, then all the three should belong to the same individual. This linkage approach requires "exactly matching" on at least two of the three identifying keys, but does not require matching at all for the third key, therefore this approach is fuzzy to some degree. Because of this fuzzy nature, it requires us to look at each pair of admissions in the data set.

[1] The data shown here is not the real-life data. It is fabricated for the purpose of this study. In the current SSN system, 9-digit number whose first three digits are between 800 to 999 are not valid Social Security Numbers.

PROC SQL APPROACH

One way to retrieve all the pairs that match on at least two identifying keys is using PROC SQL like the following:

```
PROC SQL;
   CREATE TABLE PAIRS AS
      SELECT X.ADNUM AS ADNUM1,
             Y.ADNUM AS ADNUM2
      FROM   RAWDATA X, RAWDATA Y
      WHERE (   (X.SSN EQ Y.SSN)
             + (X.NAME EQ Y.NAME)
             + (X.DOB EQ Y.DOB)   ) >= 2
        AND (X.ADNUM < Y.ADNUM)
      ORDER BY ADNUM1, ADNUM2;
QUIT;
```

Although it requires very little programming effort using PROC SQL, it may cost prohibitively huge computer resources to accomplish the task because we are using Cartesian Product joins. Suppose we have a data set containing 1,000,000, instead of 10, observations, then PROC SQL will have to look at (1,000,000*999,999)/2 potential pairs of admissions, which costs intensive processing time. It would be more efficient to first select out observations with distinct SSN||NAME||DOB values, and only execute PROC SQL on the selected sample. However, for a data set with 1,000,000 records, the final number of potential pairs that PROC SQL has to look at may still be very huge.

AN ALTERNATIVE APPROACH

Now let us consider the alternative way, which costs much less processing time, but much more programming efforts.

First we generate three packed identifying keys, called NSID, NDID, and SDID, by concatenating NAME and SSN, NAME and DOB, and SSN and DOB, respectively:

```
DATA ALLDATA1(KEEP = ADNUM NSID SDID NDID);
   SET RAWDATA;
   LENGTH NSID $ 13 SDID $ 17 NDID $ 12
          DOBC $ 8;
   DOBC = TRIM(LEFT(COMPRESS(PUT(DOB,
          YYMMDD10.),'-')));
   NSID = TRIM(NAME)||TRIM(SSN);
   NDID = TRIM(NAME)||DOBC;
   SDID = TRIM(SSN)||DOBC;
RUN;
```

Then we sort the data three times, each time by one of the packed keys, and assigning an index number to each packed key group. We could accomplish this by defining the following MACRO called GENKEY:

```
%MACRO GENKEY(INDATA_  = , /* INPUT DATA */
              OUTDATA_ = , /* OUTPUT DATA */
              KEYID_   =   /* BY VAR. */);
   %LOCAL INDATA_ OUTDATA_ KEYID_;
   PROC SORT DATA = &INDATA_;
      BY &KEYID_.ID;
   RUN;

   DATA &OUTDATA_(DROP = &KEYID_.ID);
      SET &INDATA_;
      BY &KEYID_.ID;
      IF FIRST.&KEYID_.ID THEN &KEYID_.KEY+1;
   RUN;
%MEND GENKEY;
```

and calling the MACRO three times:

```
%GENKEY(INDATA_  = ALLDATA1,
        OUTDATA_ = ALLDATA2,
        KEYID_   = NS)
%GENKEY(INDATA_  = ALLDATA2,
        OUTDATA_ = ALLDATA3,
        KEYID_   = ND)
%GENKEY(INDATA_  = ALLDATA3,
        OUTDATA_ = ALLDATA4,
        KEYID_   = SD)
```

Now our data set ALLDATA4 looks like:

```
ADNUM    NSKEY    NDKEY    SDKEY
 456       1        2        1
 101       2        2        2
 879       5        5        2
 233       3        1        3
 187       3        2        4
 490       6        4        5
 632       4        3        6
 601       6        5        6
 535       7        6        6
 911       7        6        6
```

Let us look at SDKEY for the moment. Any two admissions with the same value of SDKEY match on both SSN and DOB, and, by our criterion, belong to the same individual. Therefore the admission No. 101 and the admission No. 879 belong to the same person. Theoretically, if No. 101 and NO. 879 do belong to the same individual, then NDKEY and NSKEY should also link them together. But this is not the case with our data because the values for the NAME variable are different between No. 101 and No. 879. There may be many reasons why this happens. Perhaps there is data entry error for NAME variable, or the two admissions belong to a female who got married and changed her last name between the two admissions, or even the patient simply changed his/her first name. Regardless of the reason for the different names, SDKEY, NDKEY, and NSKEY need to be processed so that within each **KEY group, there should be no variation for the values of the other two **KEYs.

There are many ways to accomplish the above task. One is to create a format using CNTLIN= option in the FORMAT procedure from a data set in which the LABEL variable contains the smallest[2] KEY2 value within each KEY1 group, while the START variable contains other KEY2 values that are different from the smallest one within each KEY1 group. After the format is defined, we could "translate" values of KEY2 using PUT(KEY2, the-format-just-defined) function.

The MACRO GENFMTS is defined in the following to generate the format for translating:

```
%MACRO GENFMTS(INDATA_ = , /* INPUT DATA */
               FMTS_   = , /* FORMATS NAME */
               KEY1_   = , /* WITHIN */
               KEY2_   =   /* TRANSLATE */);
   %LOCAL INDATA_ FMTS_ KEY1_ KEY2_;
   PROC SORT DATA = &INDATA_;❶
      BY &KEY1_.KEY &KEY2_.KEY;
   RUN;

   DATA &FMTS_.0(KEEP = FMTNAME START LABEL
                        KEY);
      SET &INDATA_;
      BY &KEY1_.KEY;
      RETAIN FMTNAME "&FMTS_" &KEY2_.KEYF;
      IF FIRST.&KEY1_.KEY
         THEN &KEY2_.KEYF = &KEY2_.KEY;
      IF &KEY2_.KEY NE &KEY2_.KEYF THEN
         DO;
            START = &KEY2_.KEY;
            LABEL = &KEY2_.KEYF;
            KEY + 1;
            OUTPUT;
         END;
   RUN;

   PROC SORT DATA = &FMTS_.0 NODUPKEY;❷
      BY START LABEL;
   RUN;
%MEND GENFMTS;
```

We then call the MACRO GENFMTS. Notice that it is not important which one of the three **KEYs is processed first.

```
%GENFMTS(INDATA_ = ALLDATA4,
         FMTS_   = TRANSND,
         KEY1_   = SD,
         KEY2_   = ND)
```

Some explanation here:

❶ PROC SORT here insures that within each &KEY1_.KEY group &KEY2_.KEY will be in ascending order, so that later LABEL variable contains the smallest &KEY2_.KEY value within each &KEY1_.KEY group.

❷ The Data Step in the MACRO generates a data set TRANSND0 like the following:

```
FMTNAME    START    LABEL    KEY
TRANSND      5        2       1
TRANSND      5        3       2
TRANSND      6        3       3
TRANSND      6        3       4
```

But the third and the fourth observation actually do the same thing: translate NDKEY with the value 6 to the value 3. We keep only one record by NODUPKEY option in PROC SORT. Finally TRANSND0 looks like:

```
FMTNAME    START    LABEL    KEY
TRANSND      5        2       1
TRANSND      5        3       2
TRANSND      6        3       3
```

But we are far away from done! Careful readers may have noticed that we would encounter an error message like the following if trying to define the format from the data set TRANSND0: "This range is repeated, or values overlap: 5-5." In other words, it is not a valid format, because both 2 and 3 have been assigned as the formatted value for 5 at the same time. This is one of the two cases that makes format-building more complicated, and let us call it the "repeat case".

To solve the problem of the repeat case, we need to first understand why it may happen. Look back at the data set ALLDATA4: for NDKEY, 5 needs to be translated to 2 because the admission No. 101 and the admission No. 879 belong to the same person according to SDID. At the same time 5 also needs to be translated to 3 because the admission No. 632 and the admission No. 601 also belong to the same person, again according to SDID. Now our implicit criterion takes effect! NDID links the admission No. 879 and the admission No. 601 together, and none of the three packed identifying keys would link No. 101

[2] Alternatively, the LABEL variable could contain the largest KEY2 value within each KEY1 group. The important thing is to make sure that it is always translating larger numbers to smaller numbers or it is always translating smaller numbers to larger numbers. The reason why this is important will be discussed later.

and No. 632 together. However, since now No. 101 has been linked to No. 879, and No. 632 to No. 601, all the four admissions belong to the same person! The solution then is quite simple -- to be consistent with our design of translating larger numbers to smaller numbers, we need to update the data set to the following:

FMTNAME	**START**	**LABEL**	**KEY**
TRANSND	5	2	1
TRANSND	3	2	2
TRANSND	6	3	3

We will get a valid format from the above data set, but the format is still problematic. After the PUT function has been executed, there still will be at least one 3 which has not been translated into 2, and that 3 is the one which has just been translated from 6. It is another case that complicates format building, and we will call it the "link case". The solution still comes from the implicit criterion. Admissions with NDKEY equal to 6 and admissions with NDKEY equal to 2 should be linked together through a common link of admissions with NDKEY equal to 3. Therefore our data set is updated to:

FMTNAME	**START**	**LABEL**	**KEY**
TRANSND	5	2	1
TRANSND	3	2	2
TRANSND	6	2	3

In the simple example here, we have already reached a data set, which can translate NDKEY values successfully and completely. However, in reality, we may need to iterate many times before we eliminate all repeat cases and link cases. The following MACRO performs the task. Codes marked with numbers will be further explained in the next paragraph:

```
   %MACRO TRANS(INDATA_  = ,
                OUTDATA_ = ,
                INIFMTS_ = ,
                FINFMTS_ = ,
                TRANS_   = );
    %LOCAL
     INDATA_   /* INPUT DATA SET */
     OUTDATA_  /* OUTPUT DATA SET */
     INIFMTS_  /* INITIAL FORMATS */
     FINFMTS_  /* FINAL FORMATS */
     TRANS_    /* **KEY TO BE TRANSLATED */
     NEWOBS    /* OBS. IN INITIAL FORMATS */
     OLDOBS    /* OBS. IN FINAL FORMATS */
     LKOBS     /* NUM OF LINK CASES TO BE
                  UPDATED IN EACH ITERATION */
     REOBS     /* NUM OF REPEAT CASES TO BE
                  UPDATED IN EACH ITERATION */
     ITERS     /* NUM OF ITERATIONS */
     UP        /* SERIES NUMBER FOR THE
                  TEMPORARY UPDATE DATA SET */
     LK        /* SERIES NUMBER FOR THE
                  TEMPORARY LINK DATA SET */
     RE        /* SERIES NUMBER FOR THE
                  TEMPORARY REPEAT DATA SET */
    ;
      %LET UP = 1;
      %LET LK = 1;
      %LET RE = 1;
      %LET ITERS = 0;

      DATA _NULL_;
         CALL SYMPUT('OLDOBS',
                   TRIM(LEFT(PUT(OLDOBS,10.))));
         SET &INIFMTS_ NOBS = OLDOBS;
         STOP;
      RUN;
      %PUT %STR( );
      %PUT NOTE: DATA SET &INIFMTS_ HAS &OLDOBS
OBSERVATION PAIRS.;

   PROC SQL;❶
      CREATE TABLE LINK&LK AS
         SELECT Y.KEY AS KEY,
                Y.START AS START,
                X.LABEL AS LABEL
         FROM &INIFMTS_ X, &INIFMTS_ Y
         WHERE X.START EQ Y.LABEL
         ORDER BY KEY, START, LABEL;
   QUIT;

   PROC SQL;❷
      CREATE TABLE REPEAT&RE AS
         SELECT Y.KEY AS KEY,
              Y.LABEL AS START,
              X.LABEL AS LABEL
         FROM &INIFMTS_ X, &INIFMTS_ Y
         WHERE (X.START EQ Y.START)
           AND (Y.LABEL > X.LABEL)
         ORDER BY KEY, START, LABEL;
   QUIT;

   DATA _NULL_;
      CALL SYMPUT('LKOBS',
                  TRIM(LEFT(PUT(LINK,10.))));
      SET LINK&LK NOBS = LINK;
      STOP;
   RUN;
   %PUT %STR( );
   %PUT NOTE: DATA SET LINK&LK HAS &LKOBS
OBSERVATION PAIRS.;

   DATA _NULL_;
      CALL SYMPUT('REOBS',
                TRIM(LEFT(PUT(REPEAT,10.))));
      SET REPEAT&RE NOBS = REPEAT;
      STOP;
   RUN;
   %PUT %STR( );
   %PUT NOTE: DATA SET REPEAT&RE HAS &REOBS
OBSERVATION PAIRS.;

   %DO %WHILE ((&LKOBS NE 0)
            OR (&REOBS NE 0));❸
      PROC SORT DATA = &INIFMTS_;
       BY KEY;
      RUN;

      DATA UPDT&UP;
         UPDATE &INIFMTS_ LINK&LK;
         BY KEY;
      RUN;

      DATA UPDT%EVAL(&UP+1);
         UPDATE UPDT&UP REPEAT&RE;
         BY KEY;
      RUN;

      PROC SORT DATA = UPDT%EVAL(&UP+1)
                NODUPKEY;
         BY START LABEL;
      RUN;

      PROC SQL;
         CREATE TABLE LINK%EVAL(&LK+1) AS
            SELECT Y.KEY AS KEY,
                   Y.START AS START,
                   X.LABEL AS LABEL
            FROM UPDT%EVAL(&UP+1) X,
                 UPDT%EVAL(&UP+1) Y
            WHERE X.START EQ Y.LABEL
            ORDER BY KEY, START, LABEL;
      QUIT;

      PROC SQL;
         CREATE TABLE REPEAT%EVAL(&RE+1) AS
            SELECT Y.KEY AS KEY,
                   Y.LABEL AS START,
                   X.LABEL AS LABEL
            FROM UPDT%EVAL(&UP+1) X,
                 UPDT%EVAL(&UP+1) Y
            WHERE (X.START EQ Y.START)
              AND (Y.LABEL > X.LABEL)
            ORDER BY KEY, START, LABEL;
      QUIT;

      DATA _NULL_;
         CALL SYMPUT('LKOBS',
                TRIM(LEFT(PUT(LINK,10.))));
         SET LINK%EVAL(&LK+1) NOBS=LINK;
         STOP;
      RUN;
      %PUT %STR( );
      %PUT NOTE: DATA SET LINK%EVAL(&LK+1)
HAS &LKOBS OBSERVATION PAIRS.;

      DATA _NULL_;
         CALL SYMPUT('REOBS',
              TRIM(LEFT(PUT(REPEAT,10.))));
         SET REPEAT%EVAL(&RE+1) NOBS=REPEAT;
         STOP;
      RUN;
      %PUT %STR( );
      %PUT NOTE: DATA SET REPEAT%EVAL(&RE+1)
HAS &REOBS OBSERVATION PAIRS.;

      %LET ITERS = %EVAL(&ITERS+1);
      %LET INIFMTS_ = UPDT%EVAL(&UP+1);
      %LET UP = %EVAL(&UP+2);
      %LET LK = %EVAL(&LK+1);
      %LET RE = %EVAL(&RE+1);
   %END; /* END OF %DO %WHILE LOOP */

   DATA _NULL_;
      CALL SYMPUT('NEWOBS',
              TRIM(LEFT(PUT(NEWOBS,10.))));
      SET &INIFMTS_ NOBS=NEWOBS;
      STOP;
   RUN;

   %PUT NOTE: TRANS MADE &ITERS ITERATIONS.;
   %PUT NOTE: %EVAL(&OLDOBS - &NEWOBS)
DUPLICATE MAPPINGS WERE REMOVED.;

   OPTIONS NOTES;
   %PUT NOTE: SORTING &NEWOBS OBSERVATIONS IN
THE UPDATES DATA INTO &FINFMTS_..;

   PROC SORT DATA = &INIFMTS_ OUT = &FINFMTS_
             NODUPKEY;
      BY START LABEL;
   RUN;

   %IF &NEWOBS NE 0 %THEN❹
      %DO;
         PROC FORMAT CNTLIN = &FINFMTS_;
         DATA &OUTDATA_;
            SET &INDATA_;
            &TRANS_=PUT(&TRANS_,&FINFMTS_..);
         RUN;
      %END;
      %ELSE❺
      %DO;
         PROC DATA SETS;
            CHANGE &INDATA_ = &OUTDATA;
         RUN;
      %END;
   %MEND TRANS;
```

Some explanation for the MACRO TRANS:

❶ A link case happens when a value appears in both the START column and the LABEL column. This PROC SQL generates a table that will be used to update link cases. For each observation in the initial formats, the value of the START column is greater than the value of the LABEL column. Therefore, for those we have selected where X.START equals Y.LABEL, we have Y.START > Y.LABEL = X.START > X.LABEL. Since we select Y.START as START, and X.LABEL as LABEL, the value of START column is also greater than the value of LABEL column for each observation in the newly generated table “LINK&LK”.

❷ A repeat case happens when a value appears more than once in START column. This PROC SQL generates a table to be used to update repeat cases. It selects rows where X.START equals Y.START, and at the same time, Y.LABEL is greater than X.LABEL. For those selected, X.START = Y.START > Y.LABEL > X.LABEL. Since we select Y.LABEL as START, and X.LABEL as LABEL, the value for START is also greater than that for LABEL for each observation in the newly generated table “REPEAT&RE”.

❸ Stop the DO loop only when there are no link cases or repeat cases that need to be updated.

❹ Format building and value translation do not need to be done when the data set from which PROC FORMAT builds formats has no observations. Although this might never happen for huge medical treatment data sets in reality, codes that do not execute format building and value translation conditionally contain a hidden bug. If you are so lucky that no value for **KEY needs to be translated, then the data set “&FINFMTS_” has no observation, and PROC FORMAT will not output any formats. SAS will issue an error message “The format was not found or could not be loaded”. Execution of the PUT function will be stopped due to this error.

❺ When data set “&FINFMTS_” has no observations, we only need to change input data set name to output data set name.

It is not crucial whether NSKEY, NDKEY, or SDKEY is processed first. However If in the first step, variable B is processed based on each group of variable A, then in the second step, variable C should be processed based on each group of variable B. In the third step, variable D should be processed within each group of variable C. And so on, until in the last step it will process variable A based on each group of the variable which has just been processed in the previous step. For example, if we process NDKEY based on each SDKEY group first, by calling:

```
   %TRANS(INDATA_   = ALLDATA4,
          OUTDATA_  = ALLDATA5,
          INIFMTS_  = TRANSND0,
          FINFMTS_  = TRANSND,
          TRANS_    = NDKEY)
```

then we could call the following to process NSKEY based on each NDKEY group:

```
   %GENFMTS(INDATA_  = ALLDATA5,
            FMTS_    = TRANSNS,
            KEY1_    = ND,
            KEY2_    = NS)
   %TRANS(INDATA_   = ALLDATA5,
          OUTDATA_  = ALLDATA6,
          INIFMTS_  = TRANSNS0,
          FINFMTS_  = TRANSNS,
          TRANS_    = NSKEY)
```

And in the last step, we process SDKEY based on each NSKEY group by calling:

```
%GENFMTS(INDATA_ = ALLDATA6,
         FMTS_   = TRANSSD,
         KEY1_   = NS,
         KEY2_   = SD)
%TRANS(INDATA_  = ALLDATA6,
       OUTDATA_ = ALLDATA7,
       INIFMTS_ = TRANSSD0,
       FINFMTS_ = TRANSSD,
       TRANS_   = SDKEY)
```

Data set ALLDATA7 looks like:

```
ADNUM    NSKEY    NDKEY    SDKEY
 456       1        2        1
 101       1        2        1
 879       1        2        1
 233       1        1        1
 187       1        2        1
 490       1        4        1
 632       1        2        1
 601       1        2        1
 535       1        2        1
 911       1        2        1
```

Finally, we generate a variable called CLIENTID as the client identifier using the following codes. A unique CLIENTID value is assigned to each SDKEY group, because SDKEY is the last one processed.

```
PROC SORT DATA = ALLDATA7;
   BY SDKEY;
RUN;

DATA ALLDATA8(KEEP = ADNUM CLIENTID);
   SET ALLDATA7;
   BY SDKEY;
   IF FIRST.SDKEY THEN CLIENTID + 1;
RUN;
```

The last thing we need to discuss is why it is important to be consistent in translating – that is, either always translating larger numbers to smaller numbers, or always translating smaller numbers to larger numbers. If we do not impose this restriction, it is possible that after several iterations of the DO loop in MACRO TRANS, we may have the following in our data set:

```
START    LABEL
  1        2
  2        1
```

These are cases of link, and will be updated to the following:

```
START    LABEL
  1        1
```

Obviously, it is not what we want to see. However if we impose the restriction so that numbers are translated consistently from larger ones to smaller ones, or from smaller ones to larger ones, then the error above would be eliminated.

COMPARISON OF PROCESSING TIME

Processing time of the two approaches on samples with different sizes is compared in the following table. It was done on a PC running Windows NT and SAS Version 8. The time recorded in the table is the actual real time to complete the task. Notice that for the first approach, the time is only the processing time of PROC SQL, which retrieves all the pairs that match on at least two identifying keys. Some more steps (including steps dealing with link cases and repeat cases similar to those discussed above) are needed to accomplish the task of record linkage after PROC SQL in the first approach.

Time of Processing in Seconds

Sample Size	PROC SQL Part of Approach I	Approach II
100	0.12	2.37
500	0.59	2.43
1,000	2.17	2.48
5,000	51.40	2.90
10,000	198.27	3.53
50,000	4,950.48	11.78

It is obvious from the above table that as the sample size grows, processing time of PROC SQL part of the approach I increases rapidly. In my real work, I was dealing with a data set with 623,829 observations. The PROC SQL part of the first approach requires such a long processing time that the approach is virtually not feasible in practice. Meanwhile it took around three minutes and forty seconds to complete the task for the second approach.

CONCLUSION

The simple example above illustrates how to link records matching on at least two out of three identifying keys. It can be readily adapted to link records matching on at least *m* out of any given *n* keys. It also demonstrates a case when programming efforts have to be traded for shorter processing time in complicated record linkage.

CONTACT INFORMATION

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

Xinyu Ji
Fallon Clinic
100 Front Street
Worcester, MA 01608
Email: Xinyu.Ji@fallon-clinic.com