# MERGING vs. JOINING: Comparing the DATA Step with SQL

Malachy J. Foley, Chapel Hill, NC

## ABSTRACT

Which merges files better: the SAS ® DATA Step or SAS SQL? Traditionally, the only way to merge files in SAS was via the SAS DATA Step. Now SAS provides a Structured Query Language (SQL) facility which also merges files. This tutorial compares and contrasts these two merge facilities. It examines the pros and cons of each merge technique. It looks at DATA Step code to perform specific merges and then looks at the corresponding SQL code to perform the same merges.

## INTRODUCTION

This tutorial is a SQL primer for the SAS user with some experience with SAS DATA Steps and the MERGE statement, but little or no experience with SQL. It focuses on merging or joining two data sets in any combination with either the DATA Step or SQL. Upon completing the paper, the reader should have a good grasp of how SQL compares to match-merge.

## THE DEFAULT MATCH-MERGE

There are many definitions for merge. All of them talk about taking two or more sorted input files and combining them into one output file. Of course, sorted means that the input files have a common key and that the records in each file are ordered according to the key field(s). Consider the following two input files, as an example.

```
Exhibit 1: Two Input Files
------------------------------
 FILE ONE            FILE TWO
---------          -------------
ID   NAME          ID   AGE  SEX
---------          -------------
A01   SUE          A01   58   F
A02   TOM          A02   20   M
A05   KAY          A04   47   F
A10   JIM          A10   11   M
```

These two files have a common key field called ID. The records in both files are sorted by ID. A match-merge in SAS means that records from the one file will be matched up with the records of the second file that have the same ID. The information in the matched records is combined to form one output record. Here is how the above two input files are match-merged in SAS.

```
Exhibit 2: Default Match-Merge
------------------------------
DATA OUT;
    MERGE ONE TWO;
    BY ID;
RUN;
------------------------------

      FILE OUT
   -------------------
   ID   NAME AGE  SEX
   -------------------
   A01   SUE   58   F
   A02   TOM   20   M
   A04         47   F
   A05   KAY    .
```

```
                    A10   JIM   11   M
```

Note that in this merge every record from both input files exists in the output file.  Observe that the records A05 (file ONE) and A04 (file TWO) did not have a matching record, yet their information was included in the merged data set.


## ALL POSSIBLE (DATA) SETS

Exhibit 2 shows the default match-merge.  Now consider the following code which uses the same two input files as above (Exhibit 1), but has seven different output files.

```
Exhibit 3: All Match-Merge Sub-sets
-------------------------------------------
OPTIONS MERGENOBY=WARN MSLEVEL=I;
DATA ONEs TWOs inBOTH
         NOmatch1 NOmatch2 allRECS NOmatch;
 MERGE ONE(IN=In1) TWO(IN=In2);
 BY ID;
 IF In1=1 then output ONEs;
 IF In2=1 then output TWOs;
 IF (In1=1 and In2=1) then output inBOTH;
 IF (In1=0 and In2=1) then output NOmatch1;
 IF (In1=1 and In2=0) then output NOmatch2;
 IF (In1=1 OR In2=1)  then output allRECS;
 IF (In1+In2)=1       then output NOmatch;
RUN;
-------------------------------------------
```

```
  FILE ONE              FILE TWO
 ---------            -------------
ID   NAME         ID    AGE  SEX
---------            -------------
A01   SUE          A01   58    F
A02   TOM          A02   20    M
A05   KAY          A04   47    F
A10   JIM          A10   11    M


       (3a) FILE ONEs (In1=1)
       ------------------------
       ID      NAME    AGE    SEX
       ------------------------
       A01     SUE      58     F
       A02     TOM      20     M
       A05     KAY       .
       A10     JIM      11     M


       (3b) FILE TWOs (In2=1)
       ------------------------
       ID      NAME    AGE    SEX
       ------------------------
       A01     SUE      58     F
       A02     TOM      20     M
       A04              47     F
       A10     JIM      11     M
```

```
     (3c) inBOTH(In1=1 & In2=1)
-------------------------
ID     NAME     AGE    SEX
-------------------------
A01    SUE      58      F
A02    TOM      20      M
A10    JIM      11      M


     (3d) FILE NOmatch1
         (In1=0 and In2=1)
-------------------------
ID     NAME     AGE    SEX
-------------------------
A04             47      F


     (3e) FILE NOmatch2
         (In1=1 & In2=0)
-------------------------
ID     NAME     AGE    SEX
-------------------------
A05    KAY       .


(3f) FILE allRECS
         (In1=1 OR In2=1)
-------------------------
ID     NAME     AGE    SEX
-------------------------
A01    SUE      58      F
A02    TOM      20      M
A04             47      F
A05    KAY       .
A10    JIM      11      M


(3g) FILE NOmatch(In1+In2)
-------------------------
ID     NAME     AGE    SEX
-------------------------
A04             47      F
A05    KAY       .
```

   Exhibit 3 demonstrates that very little DATA Step code can create a lot of files.  Surprisingly, each of the seven output files is useful in real-world applications.

   The output file allRECS (3f in Exhibit 3) is the default match-merge that was obtained in Exhibit 2.  It contains all the information from both of the input files.

   All the other output files are subsets of allRECS.

   ONEs (3a) is a data set with all the information from file ONE along with any <u>corresponding</u> data from file TWO. Notice that ONEs does not have information on A04 since that ID is not in file ONE.

   Similarly, the TWOs (3b) data set contains all the data from file TWO with any corresponding data from file ONE.  TWOs does not include A05 since that ID does not exist in TWO.

   NOmatch1 (3d) is a file with data from file TWO that does not match the ID's from data set ONE. The Nomatch1 file contains the information that was not in  the input file called ONE.  Put another way, file ONEs (3a) and Nomatch1 together contain all the information from both input files.

    Analogously, Nomatch2 (3e) contains the information not found in file TWOs.

inBOTH (3c) contains only ID's which had records in both input files. Observe that inBOTH has neither A04 nor A05 in it.

Finally, NOmatch (3g) contains the records not found in inBOTH. In other words, NOmatch and inBOTH together have all the information from both files. Incidentally, NOmatch contains the information from both NOmatch1 and NOmatch2.

## VENN DIAGRAM OF DATA SETS

Another means of examining the match-merge given in Exhibit 3 is through a Venn diagram.

Exhibit 4: Venn Diagram of two files.
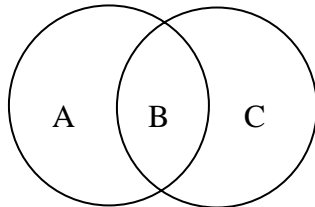


The circle on the left represents data set ONE. The circle on the right represents data set TWO. The contents of the circles are the different values of the ID's (key variables) in each data set. Each ID value falls into one of three areas: A, B, or C.

Area A contains the ID values that are unique to file ONE (i.e. values that exist in file ONE but do not exist in file TWO). Area B contains the ID values that exist in both files. Area C contains ID values that exist only in data set TWO.

The Venn diagram allows you to see all the possible sets and sub-sets that you can create via a match-merge of two data sets. Namely,

```
Exhibit 5: All possible (Data) Sets
----------------------------------
    SETS     FILES (in Exhibit 3)
----------------------------------
    A+B+C    allRECS
    A+B      ONEs
    B+C      TWOs
    A+C      NOmatch
    A        NOmatch2
    B        inBOTH
    C        NOmatch1
```

## OTHER CODES TO CREATE MATCH-MERGES

The code presented in Exhibit 3 is only one method of creating the seven possible sets with the IN= options. There are other methods. Here are some examples.

```
Exhibit 6: All Match-Merge Sub-sets
-------------------------------------------
DATA ONEs NOmatch1;
```

```
 MERGE ONE(IN=In1) TWO(IN=In2);
 BY ID;
 IF In1=1 then output ONEs;
    ELSE output NOmatch1;
RUN;
-----------------------------------------




Exhibit 7: All Match-Merge Sub-sets
-----------------------------------------
DATA inBOTH NOmatch;
 MERGE ONE(IN=In1) TWO(IN=In2);
 BY ID;
 IF (In1=1 and In2=1) then output inBOTH;
    ELSE output NOmatch;
RUN;

PROC PRINT NOOBS DATA=NOmatch;
  TITLE "Records Which Did Not Match";
RUN;
-----------------------------------------




Exhibit 8: All Match-Merge Sub-sets
-----------------------------------------
DATA ONEs inBOTH
       NOmatch1 allRECS NOmatch;
 MERGE ONE(IN=In1) TWO(IN=In2);
 BY ID;
 IF In1 then output ONEs;
 IF (In1 and In2)    then output inBOTH;
** IF (In1+In2)=2       then output inBOTH;
 IF (NOT In1 & In2)  then output NOmatch1;
 IF (In1=0 and In2=1) OR (In1=1 and In2=0)
    then output NOmatch;
RUN;
-----------------------------------------
```

Exhibits 6 and 7 demonstrate how the IF-THEN-ELSE statement can construct two mutually exclusive data sets.

Exhibit 8 illustrates various Boolean operations and alternative logical expressions to create different data sets.

It is left to the reader to contrast and compare Exhibits 6-8 with Exhibits 3-5.

Exhibit 7 shows how the two complementary output files would typically be used.  The inBOTH file is the merge you are trying to create.  The NOmatch file is a listing of all the records that did not match.  If you were expecting all the input records to match, you can print the NOmatch file as in Exhibit 7 and have a listing of errors.


**MATCH-MERGE VS. SQL**

Up until now this paper briefly toured the match-merge world. (More information about match-merge can be found in the first three references in the Bibliography.)

The remainder of this paper will explore SQL and compare it with match-merge.  Namely, the rest of the paper starts with an introduction to SQL.  After that, the paper delves into two questions: (1) what does the SQL world have to offer; and, (2) can SQL create all the seven sub-sets shown in Exhibits 3 and 5.

**WHAT IS SQL**

SQL stands for Structured Query Language.  SQL is pronounced either by saying each letter in the acronym S-Q-L, or as "sequel".

SQL is a computer language developed to create, maintain, and query relational databases. Both the American National Standards Institute (ANSI) and the International Organization for Standardization (ISO) have created standards that describe the implementation and syntax of the language.  As such, it is a well-defined language and fairly widely used. Often SQL is embedded within other programming languages and forms an integral part of those languages.

SAS data sets are not relational databases.  Yet, SAS appended SQL to its language via PROC SQL.  This way the SAS programmer can utilize SQL instructions to manipulate SAS data sets.

**SQL IS DIFFERENT**

SQL is different from regular SAS in several respects.  To start with, it employs separate terminology to describe a file.  The next Exhibit shows the equivalent words used in different settings.

```
Exhibit 9: Different Terminology
----------------------------------------
TRADITIONAL      ---SAS---         ---SQL---
----------------------------------------
   FILE          SAS DATA SET      TABLE
   RECORD        OBSERVATION       ROW
   FIELD         VARIABLE          COLUMN
```

Another difference between the DATA Step and SQL is syntax.  The following bullets outline the syntax differences.
- The PROC SQL statement turns on the SQL facility.  This facility remains on until it is turned off by a QUIT statement.  Thus, you can summit multiple SQL procedure statements (queries/tasks) after PROC SQL and before QUIT.
- Each SQL procedure statement is run automatically.  You do not need to submit a RUN statement.
- SQL procedure statements are often called SQL statements.
- Most SQL statements are longer than SAS statement.  Each SQL statement is comprised of a series of clauses.
- The order of the clauses in SQL is significant.
- Most clauses consist of different SQL components.
- There is no need to sort the input files. However, if the input is not sorted, the output will not be sorted unless you request it with the ORDER BY clause.  In other words, you do not get a free sort with SQL.
- In SQL, lists of programming elements (like variables or file names) are usually separated by commas rather than blanks.

**A DEFAULT QUERY**

SQL was designed to query (extract data and create a report from) a relational database.  Here is an example of such a query done via SAS's PROC SQL using files ONE and TWO (see Exhibit 1) as input.  Of course, SQL calls ONE and TWO tables.

```
Exhibit 10: A Default SQL Query
--------------------------------------
```

```
TITLE1 "*** A Default SQL Listing ***";
PROC SQL;
   SELECT *
      FROM ONE, TWO
   ;
QUIT;
--------------------------------------

    *** An SQL Default Listing ***

   ID     NAME    ID     AGE    SEX
   -------------------------------
   A01    SUE     A01    58     F
   A01    SUE     A02    20     M
   A01    SUE     A04    47     F
   A01    SUE     A10    11     M
   A02    TOM     A01    58     F
   A02    TOM     A02    20     M
   A02    TOM     A04    47     F
   A02    TOM     A10    11     M
   A05    KAY     A01    58     F
   A05    KAY     A02    20     M
   A05    KAY     A04    47     F
   A05    KAY     A10    11     M
   A10    JIM     A01    58     F
   A10    JIM     A02    20     M
   A10    JIM     A04    47     F
   A10    JIM     A10    11     M
```

The code in this Exhibit has three statements. PROC SQL starts an SQL session. The SELECT procedure statement is the actual query. The QUIT statement stops the SQL session.

Here the SELECT statement contains two clauses, the SELECT clause and the WHERE clause. The star (*) in the SELECT clause is a wildcard and means select all the variables. The FROM clause is part of the SELECT statement and means take the variables from tables (files) ONE and TWO. Note that a comma (not a blank) separates the files in the list.

The output from this code is a listing and not a table. The listing contains all the possible combinations of each row from table ONE with each row from table TWO. Notice that the listing has the column named ID twice. The ID on the left corresponds to the ID from table ONE. The ID on the right corresponds to the ID from table TWO.

**A DEFAULT JOIN**

In SQL the putting together the records from different input files is called a join. A match-merge also puts together records from different input files. However, the match-merge and the join use two entirely different techniques of matching the records from the input files. So, results from a match-merge and a join are often different.

As its name implies SQL is a query language. So, SQL creates tables via a query. The following code demonstrates how to create a join of tables ONE and TWO.

```
Exhibit 11: A Default Join of 2 Tables
--------------------------------------
PROC SQL;
  CREATE TABLE CART AS
     SELECT *
     FROM ONE, TWO
  ;
QUIT;
--------------------------------------
```

```
        TABLE/FILE CART
     ------------------------
     ID    NAME    AGE   SEX
     ------------------------
     A01    SUE     58    F
     A01    SUE     20    M
     A01    SUE     47    F
     A01    SUE     11    M
     A02    TOM     58    F
     A02    TOM     20    M
     A02    TOM     47    F
     A02    TOM     11    M
     A05    KAY     58    F
     A05    KAY     20    M
     A05    KAY     47    F
     A05    KAY     11    M
     A10    JIM     58    F
     A10    JIM     20    M
     A10    JIM     47    F
     A10    JIM     11    M
```

You can see the similarities between the codes in Exhibits 10 and 11. In fact, the CREATE TABLE statement in Exhibit 11 contains the same SELECT and FROM clauses from Exhibit 10.

The output in Exhibit 11 is a table and not a listing. CART is the name of the table. Of course, while SQL calls CART a table, it is still a SAS data set. Observe that the output in table CART is the same as the listing in Exhibit 10, except that it contains only one ID column. Since the ID from the first file/table is often different than the ID from the second file/table for the joined data values, the ID given in Exhibit 11 is often misleading.

Obviously, the output of this default join is quite different than the default match-merge (see Exhibit 3f).

CART is a join of tables ONE and TWO. This kind of join is called a Cartesian product. The number of rows in the table is the product of the number of rows in each of the input tables. Since there are 4 rows in table ONE and 4 rows in table TWO (see Exhibit 1), there are 16 (4x4) rows in CART.

From the previous paragraph, one sees that the "product" in Cartesian product refers to the multiplicative aspect of this type of join.


## THE CARTESIAN PRODUCT

One area where SQL shines is in its ability to create what is called a Cartesian product. To create a Cartesian product with the merge facility (see Appendix) is somewhat non-intuitive, whereas it is more obvious with SQL.

The Cartesian product in the previous example, by itself, is of little practical value. However, there are cases where the Cartesian product is exactly what you want. One such a case is where you want to create a list of all the possible screws that could be manufactured. You have several different characteristics for a screw. Some of the characteristics are head type, material, and size. You want a table with all the possible combinations of these characteristics. The following SQL code will do the trick.

```
EXHIBIT 12: Useful Cartesian Product
----------------------------------------
PROC SQL;
   CREATE TABLE SCREWS AS
     SELECT *
     FROM AA,BB,CC
   ;
 QUIT;
----------------------------------------
```

```
----      -----    ----     ----------------
  AA        BB       CC            SCREWS
----      -----    ----     ----------------
HEAD      MATL     SIZE     HEAD   MATL   SIZE
----      -----    ----     ----------------
Pan       Brass    # 6      Pan    Brass  # 6
Flat      Steel    # 8      Pan    Steel  # 6
Oval                        Pan    Brass  # 8
                            Pan    Steel  # 8
                            Flat   Brass  # 6
                            Flat   Steel  # 6
                            Flat   Brass  # 8
                            Flat   Steel  # 8
                            Oval   Brass  # 6
                            Oval   Steel  # 6
                            Oval   Brass  # 8
                            Oval   Steel  # 8
```

In a real-world application you would have more characteristics (profile, length, etc.) and more values for each characteristic than in this example. In this small illustration there were 3 kinds of screw heads, 2 kinds of materials and 2 sizes. Thus, this Cartesian product has 12 (3x2x2) rows in the output table.

Note that Exhibit 12 is Cartesian product of three input tables. You can cross up to 32 input tables.

The Cartesian product is such a powerful feature of SQL that you should add it to your toolbox even if you never use SQL for anything else.

In fact, the Cartesian product is so powerful that it is dangerous. The danger lies in the fact that you can easily create an output table that is too large for you computer to store. For example, if you cross two files with 10,000 rows each, you would obtain an output file with 100,000,000 (100 million) records! If a file gets too large, it may cause your computer to crash. So, before attempting a Cartesian product make sure you know that the resulting output table will not overwhelm your computer resources.

**A USEFUL JOIN**

The Cartesian product is some times very useful in itself as is in Exhibit 12. More frequently, the Cartesian product is a steppingstone to meaningful output. Namely, the product becomes useful when it is subsetted. The next illustrations show how the product generated in Exhibit 11 can be subsetted via a WHERE clause.

```
Exhibit 13: A Subsetted Table
----------------------------------------
PROC SQL;
  CREATE TABLE QinBOTH AS
    SELECT *
    FROM ONE, TWO
    WHERE ONE.ID=TWO.ID
  ;
QUIT;
----------------------------------------

        TABLE QinBOTH
   ------------------------
   ID     NAME   AGE   SEX
   ------------------------
   A01    SUE     58    F
   A02    TOM     20    M
```

```
        A10    JIM     11     M
```

The CREATE TABLE statement in Exhibit 13 has one additional clause: the WHERE clause.  This WHERE clause specifies that the records in the output file must have ONE.ID=TWO.ID.  Here ONE.ID refers to the ID variable in table ONE.  In a like manner, TWO.ID refers to the ID column from table TWO.

You will recognize that the above output is a subset of the output from Exhibit 11 per the WHERE clause.  This type of join is called an inner join.   Its output is often, but not always, identical to a match-merge where IN1=1 and IN2=1 (3c).  The inner join is discussed further in the next section.

## ANOTHER CODE FOR AN INNER JOIN

The next Exhibit demonstrates another way to code the inner join given in the previous section.  Some times this kind of join is called a table inner join.

```
Exhibit 14: An Inner Join
---------------------------------------
PROC SQL;
  CREATE TABLE QinBOTH AS
    SELECT *
    FROM ONE inner join TWO
    ON ONE.ID=TWO.ID
 ;
QUIT;
---------------------------------------


        FILE QinBOTH
    ------------------------
    ID     NAME    AGE    SEX
    ------------------------
    A01    SUE      58     F
    A02    TOM      20     M
    A10    JIM      11     M
```

Notice that the results in this example are identical to those in Exhibit 13.  Hence, here are two different SQL codes to construct the same inner join.   One code uses the FROM and WHERE keywords.  The other code uses the FROM, JOIN and ON keywords.

When you utilize the JOIN keyword it must be followed by the ON keyword.  Both the WHERE and ON keyword serve the same purpose.  They both subset the Cartesian product of the input files.  A JOIN-ON keyword pair can be followed by a WHERE clause if you need to do more subsetting.

An SQL inner join is often the same as a match-merge where IN1=1 and IN2=1.  They are different when there is a many-to-many merge involved.   This difference is described in the section entitled "Many-to-Many Merges".

## TYPES OF SQL JOINS

SQL has five types of joins.  They are the Cartesian product, the Inner Join, the Left Outer Join, the Full Outer Join, and the Right Outer Join.  Here is a list of the five joins and part of the SQL that produces these joins.

```
Exhibit 15:  Types of Joins & Their Code
-----------------------------------------
   TYPE OF
   JOIN      ---CODE---
-----------------------------------------
1) Carte-
    sian  ...FROM file1, file2 <,file3,..>

2a) INNER  ...FROM file1, file2 <,file3,..>
             WHERE key-field-condition

2b) INNER  ...FROM file1 INNER JOIN file2
             ON key-field-condition

2c) INNER  ...FROM file1 JOIN file2
             ON key-field-condition

3)  LEFT   ...FROM file1 LEFT JOIN file2
    OUTER     ON key-field-condition

4)  FULL   ...FROM file1 FULL JOIN file2
    OUTER     ON key-field-condition

5)  RIGHT  ...FROM file1 RIGHT JOIN file2
    OUTER     ON key-field-condition
-----------------------------------------
```

Exhibits 11 and 12 examined the Cartesian product. Exhibits 13 demonstrated an inner join with code 2a in the above table. Exhibit 14 presented an example of code 2b from the table. Code 2c in Exhibit 15 is the same as that given in Exhibit 14 except that the keyword INNER is not specified. The keyword INNER is optional. Hence, code 2b and 2c yield identical results.

The previous Exhibit reveals that the two forms of the inner joins and all of the outer joins employ the JOIN-ON keyword pair. This pair is known as "joined-table" component of a clause. This component only allows two input files per join.

**OUTER JOINS**

The last three joins in Exhibit 15 are outer joins. An outer join is an inner join with addition rows <u>from the original input files</u> (not from the Cartesian product). The left outer join contains all the rows from the inner join plus any rows from file1 (the left-hand table) that were not included in the inner join. The right outer join is an inner join plus any rows from file 2 (the right-hand table) that were not included in the inner join.

A left outer join contains the ID's in parts A and B of the Venn diagram given in Exhibit 4. A right outer join contains the ID's from B and C of the Venn diagram.

The full outer join is an inner join with rows from both original input tables that were not included in the inner join. A full outer join contains the ID's in parts A, B and C of the Venn diagram in Exhibit 4.

Often you will hear the "left outer join" referred to as just the "left join". This is because the code that creates the join uses only the keyword "left join". The terms, however, are synonymous. Likewise, "full join" means "full outer join", and "right join" means "right outer join".

The syntax of the outer joins (Exhibit 15) allows only two input tables.

The next many sections examine the various SQL outer joins.

## A LEFT OUTER JOIN

Here is how to perform a left outer join of tables ONE and TWO in SQL.  (The code in the following exhibit is similar to that given in Exhibit 14.)

```
Exhibit 16: An SQL Left Outer Join
-------------------------------------
PROC SQL;
  CREATE TABLE qONEs AS
    SELECT *
    FROM ONE left join TWO
    ON ONE.ID=TWO.ID
 ;
QUIT;
-------------------------------------


         FILE qONEs
   ------------------------
   ID      NAME    AGE    SEX
   ------------------------
   A01     SUE      58     F
   A02     TOM      20     M
   A05     KAY       .
   A10     JIM      11     M
```

Note that the file qONEs is the inner join of Exhibit 14 plus the row for ID=A05 from file ONE (Exhibit 1).  The row for ID=A05 was not in the inner join.

You will also notice that the file qONEs is the same as file ONEs in Exhibit 3a.  The left outer join is always the same as a match-merge with IN1=1, EXCEPT in the case of many-to-many merges (described later).

## ATTEMPTING A FULL OUTER JOIN

Here is how to perform a FULL outer join of tables ONE and TWO in SQL.

```
Exhibit 17: A Full Outer Join (Surprise)
-----------------------------------------
PROC SQL;
  CREATE TABLE ALL AS
    SELECT *
    FROM ONE full join TWO
    ON ONE.ID=TWO.ID
 ;
QUIT;
-----------------------------------------

         FILE  ALL
   ------------------------
   ID      NAME    AGE    SEX
   ------------------------
   A01     SUE      58     F
   A02     TOM      20     M
```

```
                    47      F
        A05    KAY      .
        A10    JIM     11      M
```

In this example, we expected table ALL to be the Inner Join from Exhibit 14 plus the row for ID=05 from file ONE (Exhibit 1) plus the row for ID=A04 from file TWO (Exhibit 1). Indeed this is the result except surprisingly the value "A04" for the row from TWO is missing while the rest of the values for that row are in the table.

This surprising result can also be appreciated by comparing the table ALL with the file allRECS from Exhibit 3f. Again the difference is that only the value A04 for the ID is missing in the SQL version of the join.

The reason why "A04" is missing can be found in the fact that SQL always has two ID's when it is joining. These ID's are ONE.ID and TWO.ID (see Exhibit 10 for example). When it comes time to create a table with only one ID value, SQL has to choose which value to put in the table. It chooses the left ID. That is why the ID value of A05 from the left-hand table is in the output. It is also why A04 from the right-hand table is not in the table ALL. SQL warns you of this situation with the message "WARNING: Variable ID already exists on file WORK.ALL."


## A FULL OUTER JOIN

One method to add the missing right-hand key values (i.e. A04) to a full join is to use the SQL COALESCE function. The COALESCE function returns the first non-missing argument. All of the function's arguments or parameters must be of the same of the same type (char or numeric). The next example illustrates how to do a full outer join with the COALESCE function.

```
Exhibit 18: A Full Outer Join (with A04)
----------------------------------------
PROC SQL;
 CREATE TABLE QallRECS(DROP=oldID)
   AS
   SELECT
      coalesce(ONE.oldID,TWO.oldID) AS ID,
      *
   FROM ONE (RENAME=(ID=oldID))
       full join
       TWO (RENAME=(ID=oldID))
   ON ONE.oldID=TWO.oldID
 ;
QUIT;
----------------------------------------

        FILE QallRECS
    ------------------------
    ID     NAME    AGE    SEX
    ------------------------
    A01    SUE     58      F
    A02    TOM     20      M
    A04            47      F
    A05    KAY      .
    A10    JIM     11      M
```

Exhibit 18 demonstrates how to do a full join without getting the "Variable already exists" warning message, while maintaining the order of the variable in the output table. Note that this souped-up version of a full join is similar to the default match-merge in Exhibits 2 and 3f. However, the SQL version is much more code intensive. The enhanced version of the full join is always the same as a default match-merge, EXCEPT in the case of many-to-many merges (described later).


## A RIGHT OUTER JOIN

For the sake of completeness, an example of a right outer join is given below. The default right join suffers the same difficulty as the default full join. Namely, the key variable values are lost from the right-hand input table.

Again, the COALESCE function can solve this difficulty.

```
Exhibit 19: An SQL Right Outer Join
----------------------------------------
PROC SQL;
 CREATE TABLE qTWOs(DROP=oldID)
   AS
   SELECT
       coalesce(ONE.oldID,TWO.oldID) AS ID,
       *
   FROM ONE (RENAME=(ID=oldID))
       right join
       TWO (RENAME=(ID=oldID))
   ON ONE.oldID=TWO.oldID
 ;
QUIT;
----------------------------------------

        FILE qTWOs
    ------------------------
    ID    NAME    AGE   SEX
    ------------------------
    A01   SUE      58    F
    A02   TOM      20    M
    A04            47    F
    A10   JIM      11    M
```

Note that the file qTWOs is the Inner Join of Exhibit 14 plus the row for ID=A04 from file TWO (Exhibit 1). You will also notice that the file qTWOs is the same as file TWOs in Exhibit 3b. The enhanced right outer join is always the same as a match-merge with IN1=2, EXCEPT in the case of many-to-many merges (described next).


## THE MANY-TO-MANY MERGE

The many-to-many merge refers to the situation where you have duplicate values for key variables in both input files simultaneously. This is the case for the value A25 in the files ALFA and BETA below. A25 appears in both files three times. Exhibit 20 shows that the match-merge in this situation gives you a one-to-one merge of the A25 values. Meanwhile, Exhibit 21 indicates that SQL gives you a Cartesian product of these values.

```
Exhibit 20: Match-Merge of Many-to-Many Data
------------------------------------------
    DATA ALFAbeta;
      MERGE ALFA(IN=IN1) BETA(IN=In2);
      BY ID;
      IF In1 and In2;
    RUN;
------------------------------------------


----------    ---------    -------------
FILE ALFA     FILE BETA    FILE ALFAbeta
----------    ---------    -------------
 ID   V1       ID   V4      ID   V1   V4
 --------      -------      -------------
 A21    8      A21   55     A21    8   55
 A25   24      A25    4     A25   24    4
 A25   22      A25   91     A25   22   91
 A25   76      A25   38     A25   76   38



Exhibit 21: Inner Join of Many-to-Many Data
------------------------------------------
    PROC SQL;
     CREATE TABLE ALfAbeta AS
```

```
        SELECT *
        FROM ALFA, BETA
        WHERE ALFA.ID=BETA.ID
      ;
     QUIT;
-------------------------------------------


----------      ---------      -------------
FILE ALFA       FILE BETA      FILE ALFAbeta
----------      ---------      -------------
 ID    V1        ID   V4        ID    V1   V4
 --------       -------        -------------
 A21    8        A21  55        A21    8   55
 A25   24        A25   4        A25   24    4
 A25   22        A25  91        A25   24   91
 A25   76        A25  38        A25   24   38
                                A25   22    4
                                A25   22   91
                                A25   22   38
                                A25   76    4
                                A25   76   91
                                A25   76   38
```

   Previously, the sections entitled "A Useful Join" and "Another Code for an Inner Join" showed that a match-merge with IN1=1 and IN2=1 is often the same as an inner join.  Indeed, this was the case with Exhibits 3c and 13&14.  It is even true for A21 in the above exhibits (Exh. 20 and 21).  However, as the two previous exhibits reveal, it is not the case for many-to-many merges.

   The match-merge situation illustrated in Exhibit 20 is almost always an error.  SAS signals that something is amiss by putting a note in the log.  The text reads "NOTE:  MERGE statement has more than one data set with repeats of BY values".  Usually, this note means that additional BY variables are required to make a proper match.  Sometimes, it means that the BY variables themselves are faulty.

   The SQL results in Exhibit 21 might be exactly what you want.  Sometimes you need a Cartesian product for each value of the key variables.

   However, if your goal is to replicate the match-merge in Exhibit 20, the results in Exhibit 21 are disappointing.  They prove that for many-to-many merges, the SAS DATA Step and SAS SQL do not give the same results for an inner join.  Since the inner join is a component of all the outer joins, the DATA Step and SQL do not give the same results on many-to-many outer joins either.


**FINAL NOTES ON JOINS**

   The preceding sections demonstrate that every join in SQL is either the Cartesian product itself, or a subset of the Cartesian product (inner join), or a subset of the Cartesian product with additional rows from the original input tables (outer join).

   The Cartesian product is always part of an SQL join.  As such, you needs to be careful to always specify a subsetting WHERE or ON clause when needed.  Otherwise you could inadvertently create a Cartesian product so large that is crashes your computer.

   One of the goals of this paper was to investigate if there is corresponding code for the match-merge and the join.  The DATA Step and SQL utilize completely different techniques to combine files.  Despite this fact, there are four cases where the two techniques can create the same output provided that the input files do not have many-to-many ID values.  The following table outlines where equivalent code exists.

```
    Exhibit 22: All possible Combinations
   ------------------------------------ --
    SETS      MERGE       JOIN    Example
    (Ex.4)    (Exh. 3)            Join code
```

```
    ----------------------------------------
    A+B+C   allRECS     Full    Exhibit 18
    A+B     ONEs        Left    Exhibit 16
    B+C     TWOs        Right   Exhibit 19
    A+C     NOmatch
    A       NOmatch2
    B       inBOTH      Inner   Exhs.13/14
    C       NOmatch1
```

Exhibits 3, 4 and 5 showed that there are seven possible ways to combine two sets of ID's. These seven combinations are repeated in the above table. SQL can reproduce the match-merge results in four cases with two provisos. First, the input data sets can not have more than one data set with repeats of BY values for a given BY group. Second, the full and right joins need to utilize the COALESCE function (Exhibits 18 & 19) or some other alternative code.

The table also mentions that there are three results where the match-merge cannot easily be reproduced. These are the NOmatch, NOmatch1 and NOmatch2 files that correspond to (3g), (3e), and (3d) respectively. These three files all involve data from only the input files without any information from the combination of the two files. Every SQL join has the Cartesian product combination as part of its output. As such, a simple join cannot create listings of only non-combined input records.

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 traps also apply to SQL joins. For example, to do a join 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.


**CONCLUSION**

When it comes to manipulating files, the DATA Step match-merge and the SQL joins are different. The match-merge was designed to manipulate sorted data sets. SQL was designed to create reports from relational (non-sorted) databases. Match-merge and SQL use different methodologies to create merge/joins. They have different syntaxes (see sec "SQL is Different"). They even utilize different terminology to describe files and file components (Exh. 9).

They put files together differently. The match-merge basically matches records one-to-one. It takes one record from the first input file and combines it with one record from the second input file provided they have the same ID. SQL on the other hand performs an all-to-all record match. It takes one record from the first file and puts it together with every record from the second file that have the same ID. Match-merge couples or pairs records. SQL multiplies or crosses records.

Due to the way SQL functions, SQL can easily create Cartesian products (Exh. 12) and BY-Group Cartesian products (Exh. 21). These joins are difficult to do with a DATA Step (see Appendix).

On the other hand, can effortlessly match and subset records (Exhs. 3&4). Some of these subsets are difficult to do in SQL.

One goal of this paper was to see if SQL could replicate the seven match-merges described in Exhibits 3, 4 and 5. Amazingly, SQL can replicate four of the seven match-merges (Exh 23) under two conditions. First, you must employ the COALESCE function on the full join and the right join. Second, the input files may not have duplicate BY values in both input files (no many-to-many situations).

Match merges allow you to combine many files at once. Cartesian joins and inner joins allow the same flexibility. However, outer joins allow only two input tables.

This paper started by asking "Which merges better". The answer is neither. The two facilities are different and cannot be readily compared. One is not better than the other. They are complementary. The DATA Step merges.

SQL joins. The two facilities perform significantly different functions. Both functions are valuable. The match-merge and the join are both different and valuable. "Vive la difference!"


## APPENDIX

The following code creates the join given in Exhibit 11 with a DATA Step rather than SQL.

```
Exhibit 23: The Default Join of 2 Tables
------------------------------------
DATA CART(rename=(ID1=ID) drop=ID);
  SET ONE (rename=(ID=ID1));
  DO i=1 TO NN;
     SET TWO point=i nobs=NN;
     OUTPUT;
  END;
RUN;
------------------------------------
```

Notice that in order to get the ID from the first file (ONE), you need to use Data Set Options on the ONE and CART files. Without these data set options, the ID would be taken from the second data set (TWO). The ID's in the output file are ambiguous in either case.


## BIBLIOGRAPHY

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.

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.

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

SAS Institute, Inc., *SAS Procedures Guide, Version 8*, First Edition (Carry, NC: SAS Institute Inc., 1999)

## TRADEMARKS

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks 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.


## ABOUT THE AUTHOR

Mal has been programming computers continuously since High School. His career includes being an international computing consultant, a university professor, and the CEO of his own computing company. Recently, Mal has taken a hiatus from his position of many years as senior SAS programmer/analyst with the Department of Biostatistics at the University of North Carolina, Chapel Hill. He actively participates in user groups and will chair the 2005 Pharmaceutical Industry SAS Users' Group. Currently, Mal is an independent SAS trainer and provides in-house SAS seminars around the country.

## AUTHOR CONTACT

Malachy J. Foley
2502 Foxwood Dr.
Chapel Hill, NC 27514
Email: FOLEY@unc.edu