Joining Tables of Data in the SQL Procedure

Kirk Paul Lafler, Software Intelligence Corporation

ABSTRACT

Joining two or more tables of data is a powerful feature found in the relational model and the SQL procedure. Often, the need to analyze or process data that resides in two or more tables is necessary. When this is the case, a common or connecting column is used from each table to form the basis for the join. Consequently, it is possible to discover new and exciting relationships between data.

This paper illustrates what a join is, why joins are necessary in data analysis, the concept of joins and the Cartesian product in relational theory, and the numerous types of joins that can be performed. In particular, participants will learn how to perform basic join operations on two or more tables (inner joins), specify join conditions, and perform left, right, and full joins (outer joins) using SAS System data sets (tables).

INTRODUCTION

The SQL procedure is a simple and flexible tool for joining two or more tables of data. A maximum of sixteen tables can be joined together when adhering to the syntax associated with conventional inner joins (no more than two tables at a time when performing outer joins).

This paper presents useful techniques with many examples to illustrate the advantages associated with joining tables of data. Discussions include the differences between merging and joining, (inner) joins and the Cartesian product, creating and using table aliases, joining three or more tables of data, and outer (left, right, and full) joins. Certainly, many of these techniques can be accomplished using other methods, but the simplicity and flexibility found in the SQL procedure makes it especially interesting, if not indispensable, as a tool for the practitioner.

TABLE DESCRIPTIONS

This paper presents a Dive Location Information System (DLIS) with three primary tables as the source of data. The tables from the DLIS are used to illustrate the information gathering process via the DATA step merge and SQL procedure. Duplicate matching values exist for tables SITE and ENVIRON, while the LOCATION table consists of no duplicate matching values.

TABLE - LOCATION

Variable	Type	Length
LOC_CODE	Char	3
LOCATION	Char	20
CITY	Char	13
STATE	Char	2

Figure 1.

TABLE - SITE

Variable	Type	Length
LOC_CODE	Char	3
WRECKS	Char	1
DEPTH1	Num	3
DEPTH2	Num	3
DESC1	Char	40
DEPTH2	Char	40

Figure 2.

TABLE - ENVIRON

Variable	Type	Length
LOC_CODE	Char	3
TEMP1	Num	3
TEMP1	Num	3
VIS1	Num	3
VIS2	Num	3
COND1	Char	40
COND2	Char	40

Figure 3.

WHY JOIN ANYWAY?

As relational data base systems take-off in popularity, the need to access normalized data that has been stored in separate tables becomes increasingly more important. By relating matching values in key columns in one table with key columns in two or more tables, information can be retrieved as if the data were stored in one huge file. Consequently, the process of joining data from two or more tables can provide new and exciting insights between data relationships.

MERGE AND JOIN DIFFERENCES

The outcome of a DATA step merge versus an SQL join depends on the make-up of the data itself. When at least one table has nonduplicate matching (key) values, the outcome of both are similar. But, in the absence of at least one table with nonduplicate key values, the merge and join produce very different results. The following examples present each method (DATA step merge and SQL join) for inspection.

Figure 4 illustrates the code used in the DATA step merge for nonduplicate matching values.

Merge - NonDuplicate Matching Values

```
proc sort data=perm.site;
  by loc_code;
run;
proc sort data=perm.location;
  by loc_code;
run;
data merge;
  merge perm.site
        perm.location;
    by loc_code;
run;
proc print data=merge noobs;
  var loc_code location wrecks;
  title1 'Merge Example';
  title2 '1 Duplicate / 1 NonDuplicate Table';
run;
```

Figure 4.

Figure 5 depicts the result from the merge operation presented previously in Figure 4. All rows satisfying the BY statement are selected and printed.

Merge - NonDuplicates - Output

Merge Example
1 Duplicate / 1 NonDuplicate Table

LOC_CODE	LOCATION	WRECKS
C01	Catalina Island	Y
C01	Catalina Island	N
C02	Coronado Island	Y
F01	Key West	Y
F02	Turkey Point	N

Figure 5.

Figure 6 illustrates the code used in the SQL step for joining tables with nonduplicate matching values.

SQL Join - NonDuplicate Matching Values

```
proc sql;
  title1 'SQL Join Example';
  title2 '1 Duplicate / 1 NonDuplicate Table';
  select l.loc_code,
         l.location,
         s.wrecks
    from perm.site as s,
         perm.location as l
   where s.loc_code = l.loc_code;
run;
```

Figure 6.

Figure 7 depicts the result from the SQL join operation presented previously in Figure 6. All rows satisfying the WHERE clause are selected and printed. There are very little, if any, differences between the results generated using a DATA step merge or SQL join when at least one table contains nonduplicate matching values.

SQL Join - NonDuplicates - Output

SQL Join Example
1 Duplicate / 1 NonDuplicate Table

LOC_CODE	LOCATION	WRECKS
C01	Catalina Island	Y
C01	Catalina Island	N
C02	Coronado Island	Y
F01	Key West	Y
F02	Turkey Point	N

Figure 7.

Figure 8 illustrates the code used in the DATA step merge for one or more tables containing duplicate matching values.

Merge - Duplicate Matching Values

```
proc sort data=perm.site;
  by loc_code;
run;
proc sort data=perm.environ;
  by loc_code;
run;
data merge;
  merge perm.site
        perm.environ;
    by loc_code;
run;
```

Figure 8.

```
proc print data=merge noobs;
  var loc-code wrecks coral kelp rocks;
  title1 'Merge Example';
  title2 'Duplicate Matching Values';
run;
```

Figure 8 (Continued).

Figure 9 depicts the result from the merge operation presented previously in Figure 8. All rows satisfying the BY statement are selected and printed.

Merge - Duplicates - Output

Merge Example
Duplicate Matching Values

LOC_ CODE	WRECKS	CORAL	KELP	ROCKS
C01	Y	N	Y	Y
C01	N	N	Y	Y
C02	Y	N	Y	Y
F01	Y	Y	N	N
F02	N	Y	N	Y

Figure 9.

Figure 10 illustrates the code used in the SQL step for joining tables with duplicate matching values.

SQL Join - Duplicate Matching Values

```
proc sql;
  title1 'SQL Join Example';
  title2 'Duplicate Matching Values';
  select s.loc_code,
         s.wrecks,
         e.coral,
         e.kelp,
         e.rocks
    from perm.site as s,
         perm.environ as e
   where s.loc_code = e.loc_code;
run;
```

Figure 10.

Figure 11 depicts the result from the SQL join present-ed previously in Figure 10. All rows satisfying the WHERE clause are selected and printed. There are several differences between the results generated using a DATA step merge or SQL join when duplicate matching values are found in the joining tables.

SQL Join - Duplicates - Output

SQL Join Example
Duplicate Matching Values

LOC_ CODE	WRECKS	CORAL	KELP	ROCKS
C01	Y	N	Y	Y
C01	N	N	Y	Y
C01	Y	N	Y	Y
C01	N	N	Y	Y
C02	Y	N	Y	Y
F01	Y	Y	N	N
F02	N	Y	N	Y

Figure 11.

THE CARTESIAN PRODUCT

Joins are processed in two distinct phases. The first phase determines whether a FROM clause contains two or more tables. When it does, a ***virtual*** table, known as the Cartesian product, is created resulting in each row in the first table being combined with each row in the second table, and so forth. The Cartesian product contains every combination of rows from each table that is joined together. Consequently, it can be extremely large in size.

The second phase processes the condition(s) specified in the WHERE clause. This is where the number of rows are reduced to a more manageable size.

TABLE ALIASES

Table aliases provide a "short-cut" way to reference one or more tables within SQL. They are often specified when performing joins, so columns can be selected with a minimal number of keystrokes.

Figure 12 illustrates the creation and use of table aliases.

Table Aliases - Creation and Use

```
proc sql;
  title1 'Table Aliases Example';
  select s.loc_code,
         s.wrecks,
         e.coral
    from perm.site as s,    /* ALIAS IS S */
         perm.environ as e /* ALIAS IS E */
   where s.loc_code = e.loc_code;
run;
```

Figure 12.

JOINING THREE OR MORE TABLES

A maximum of sixteen tables can be joined with the SQL procedure. Figure 13 shows the syntax requirements for joining three or more tables.

Joining 3 or More Tables - Syntax

```
proc sql;
  title1 'Joining 3 or More Tables - Syntax';
  select column-1,
         column-2,
            ...
            ...
         column-n
    from table-1,
         table-2,
            ...
            ...
         table-n
     where table-1.column-1 = table-2.column-2
       and table-1.column-1 = table-3.column-3
                  ...              ...
                  ...              ...
       and table-1.column-1 = table-n.column-n;
quit;
```

Figure 13.

Figure 14 illustrates the SQL code for joining the three tables SITE, ENVIRON, and LOCATION on the common key column LOC_CODE.

Joining 3 Tables - Code

```
proc sql;
  title1 'Joining 3 Tables';
  select s.loc_code,
         l.location,
         s.wrecks,
         e.coral,
         e.kelp,
         e.rocks
    from perm.site as s,
         perm.location as l,
         perm.environ as e
     where s.loc_code = l.loc_code and
           s.loc_code = e.loc_code;
quit;
```

Figure 14.

OUTER JOINS

Outer joins are significantly different from inner joins, since they show not only the rows that match the WHERE condition, but rows that do not match from one or more of the tables as well. The keywords for each type of outer join follows.

Left	LEFT JOIN and ON keywords.
Right	RIGHT JOIN and ON keywords
Full	FULL JOIN and ON keywords

The syntax requirement for a ***left*** outer join follows, (Figure 15).

Left Outer Join - Syntax

```
proc sql;
  title1 'Left Outer Join Syntax';
  select *
    from libref.left
         left join
         libref.right
      on left.key = right.key;
quit;
```

Figure 15.

Figure 16 illustrates the coded SQL step used to identify and select sites with a maximum depth of no more than 65 feet as well as rows in the left table (LOCATION) that do not match rows in the right table (SITE).

Left Outer Join - Code

```
proc sql;
  title1 'Left Outer Join';
  select location.location,
         location.state,
         site.depth
    from perm.location      /* LEFT TABLE */
         left join
         perm.site          /* RIGHT TABLE */
      on location.loc_code = site.loc_code and
         site.depth2 LE 65;
quit;
```

Figure 16.

The resulting output depicted in Figure 17 displays rows that satisfy the SQL ON expression plus any rows from the left table (LOCATION) that do not match any rows in the right table (SITE). Notice that the values for DEPTH2 are missing for the first two rows.

Left Outer Join - Output

Left Outer Join

LOCATION	STATE	DEPTH2
Catalina Island	CA	.
Coronado Island	CA	.
Key West	FL	65
Turkey Point	FL	55

Figure 17.

The syntax requirement for a ***right*** outer join follows, (Figure 18).

Right Outer Join - Syntax

```
proc sql;
  title1 'Right Outer Join Syntax';
  select *
    from libref.left
         right join
         libref.right
      on left.key = right.key;
quit;
```

Figure 18.

Figure 19 illustrates the coded SQL step used to identify and select sites with a maximum depth of no more than 65 feet as well as rows in the right table (SITE) that do not match rows in the left table (LOCATION).

Right Outer Join - Code

```
proc sql;
  title1 'Right Outer Join';
  select location.location,
         location.state,
         site.depth
    from perm.location      /* LEFT TABLE */
         right join
         perm.site          /* RIGHT TABLE */
      on location.loc_code = site.loc_code and
         site.depth2 LE 65;
quit;
```

Figure 19.

The resulting output depicted in Figure 20 displays rows that satisfy the SQL ON expression plus any rows from the right table (SITE) that do not match any rows in the left table (LOCATION). Notice that the values for LOCATION and STATE are missing for the first two rows.

Right Outer Join - Output

Right Outer Join

LOCATION	STATE	DEPTH2
		80
		125
Key West	FL	65
Turkey Point	FL	55

Figure 20.

Figure 21 illustrates the syntax for a ***full*** outer join.

Full Outer Join - Syntax

```
proc sql;
  title1 'Full Outer Join Syntax';
  select *
    from libref.left
         full join
         libref.right
      on left.key = right.key;
quit;
```

Figure 21.

Figure 22 illustrates the coded SQL step used to identify and select sites with a maximum depth of no more than 65 feet as well as rows in the left table (LOCATION) and rows in the right table (SITE) that do not have matching rows.

Full Outer Join - Code

```
proc sql;
  title1 'Full Outer Join';
  select location.location,
         location.state,
         site.depth
    from perm.location      /* LEFT TABLE */
         full join
         perm.site          /* RIGHT TABLE */
      on location.loc_code = site.loc_code and
         site.depth2 LE 65;
quit;
```

Figure 22.

The resulting output depicted in Figure 23 displays rows that satisfy the SQL ON expression plus any rows from the left table (LOCATION) that do not match any rows in the right table (SITE) plus any rows from the right table (SITE) that do not match any rows in the left table (LOCATION). Notice that the values for LOCATION and STATE are missing for the first and third rows, and the values for DEPTH2 are missing for the second and fourth rows.

Full Outer Join - Output

Full Outer Join

LOCATION	STATE	DEPTH2
		80
Catalina Island	CA	.
		125
Coronado Island	CA	.
Key West	FL	65
Turkey Point	FL	55

Figure 23.

CONCLUSION

The SQL procedure makes joining two or more tables simple and fun. As data is joined, it is possible to discover new and exciting relationships between data. As long as "connecting" columns exist, the possibilities for information enlightenment are enhanced.

Since a maximum of sixteen tables can be joined together when dealing with conventional inner joins, the possibilities for exploring data relationships are numerous. Outer joins (left, right, and full) permit no more than two tables at one time to be joined. Joining tables of data can certainly be classified as a technique worth further research and exploration.

ACKNOWLEDGMENTS

My sincere thanks are extended to Marian Oshiro, Co-Conference Chair, Sandy Thurman, Co-Conference Chair, Karen Spritzer, Publications, and Steve Schulz, Tutorials, for their continued support during the preparation of this paper and presentation.

AUTHOR CONTACT INFORMATION

Kirk Paul Lafler
Software Intelligence Corporation
P.O. Box 1390
Spring Valley, California 91979-1390
Telephone: (619) 670-SOFT or (619) 670-7638

TRADEMARKS

SAS is the registered trademark of SAS Institute Inc., Cary, NC, USA.