

Joining Data with SQL

Kim L. Kolbe Ritzow of Systems Seminar Consultants, Kalamazoo, MI

Abstract

This paper will discuss the various types of joins available in PROC SQL and how they are performed. It will also compare these methods with traditional SAS code using the MERGE statement.

It will illustrate the differences between joining and subquerying data, provide some tips for more efficient joins using PROC SQL, and illustrate some of the more complex joins that can be used in PROC SQL.

This paper does assume some prior knowledge of PROC SQL's syntax.

Background

PROC SQL can be a very efficient way to join tables together, especially if they are large tables that are not sorted. In general, PROC SQL will require less CPU to join data than the traditional MERGE statement, but will use slightly higher I/O. In most cases the traditional MERGE statement is still better at merging smaller data sets together than SQL.

Types of Joins In SQL

- ☐ A Cartesian Join
- ☐ An Inner Join
- ☐ A Left Join
- ☐ A Right Join
- ☐ A Full Join

An inner, left, right and full joins are types of equi-joins. An equi-join is a join that has an equality comparison on the WHERE clause. Joins without a WHERE clause are referred to as cartesian joins.

An advantage of joining data in SQL is the data does not have to be sorted prior to bringing the data sets together. Although explicitly sorting the

data beforehand may improve performance in some cases (see Performance Issues later in this paper for details).

Be careful, the order of the data in the resulting table may vary from one version to another based on changes made to the SQL optimizer. The only way the order of the resulting data can be controlled is through an ORDER BY clause.

A Cartesian Join

It is very easy to do a cartesian join in SQL, because it is the result when a WHERE clause is not used when joining data together. It is difficult to perform a cartesian join with the traditional MERGE statement.

Table DATA1

VAR1	VAR2
ABC	10
GHI	15
MNO	20

Table DATA2

VAR1	VAR3
DEF	20
JKL	25
PQR	30

```
PROC SQL;  
SELECT *  
FROM DATA1, DATA2;
```

The resulting output:

VAR1	VAR2	VAR1	VAR3
ABC	10	DEF	20
ABC	10	JKL	25
ABC	10	PQR	30
GHI	15	DEF	20
GHI	15	JKL	25
GHI	15	PQR	30
MNO	20	DEF	20
MNO	20	JKL	25
MNO	20	PQR	30

A cartesian join is a **product** of all the data sets specified. That is, a cartesian join will take each of the observations from the first data set specified and join them with each of the observations from the second data set specified. If each of the data sets had ten observations, the result of a cartesian join would be a data set with one hundred observations.

An Inner Join

There are two styles of inner joins, one with the **INNER JOIN** specified, the other without. In either case, an inner join will give us an observation only if it is in both data sets:

Table DATA1

VAR1	VAR2
ABC	10
GHI	15
MNO	20

Table DATA3

VAR1	VAR3
ABC	20
JKL	25
PQR	30

```
PROC SQL;
SELECT *
FROM DATA1, DATA3
WHERE DATA1.VAR1=DATA3.VAR1;
```

This is the same as:

```
PROC SQL;
SELECT *
FROM DATA1 A INNER JOIN DATA3 B
ON A.VAR1=B.VAR1;
```

The result is the same for either piece of logic:

VAR1	VAR2	VAR1	VAR3
ABC	10	ABC	20

In either case, by default, VAR1 will appear twice in the result of an SQL join, but not in a MERGE. We can control for this by specifying on the SELECT statement the variable we would like to see regardless if we are explicitly specifying an **INNER JOIN** or not:

```
PROC SQL;
SELECT A.VAR1, VAR2, VAR3
FROM DATA1 A, DATA3 B
WHERE A.VAR1=B.VAR1;
```

The resulting output:

VAR1	VAR2	VAR3
ABC	10	20

The advantage of not specifying the **INNER JOIN** is that there is no limit to the number of data sets that can be joined together, whereas the explicit **INNER JOIN** requires two and only two data sets to be specified.

A one-to-one match merge using the MERGE statement that checks to make sure an observation is on both of the data sets will give us the same result:

```
DATA MERGEIT;
MERGE DATA1(IN=ONA) DATA3(IN=ONB);
BY VAR1;
IF ONA AND ONB;
```

Notice that with an inner join on PROC SQL we do not have to check to see if it is on both data sets, it is implied.

A One-to-Many Inner Join

Our previous example contained only one of each value in both of the data sets, but if there were multiple occurrences of a value in one of the data sets, we would call this a one-to-many join. A one-to-many inner join is performed the same way as a one-to-one inner join:

Table DATA1

VAR1	VAR2
ABC	10
GHI	15
MNO	20

Table DATA4

VAR1	VAR3
ABC	20
ABC	25
PQR	30

```
PROC SQL;
SELECT A.VAR1, VAR2, VAR3
FROM DATA1 A, DATA4 B
WHERE A.VAR1=B.VAR1;
```

The resulting output:

VAR1	VAR2	VAR3
ABC	10	20
ABC	10	25
MNO	20	30

Similarly, the MERGE statement's syntax would be the same regardless if a one-to-one versus a one-to-many match merge is being performed.

A Many-to-Many Inner Join

When there are multiple occurrences of a value in both of the data sets we still do nothing different when joining the data with SQL:

Table DATA5

VAR1	VAR2
ABC	10
ABC	15
MNO	20

Table DATA6

VAR1	VAR3
ABC	20
ABC	25
PQR	30

```
PROC SQL;
SELECT A.VAR1, VAR2, VAR3
FROM DATA5 A, DATA6 B
WHERE A.VAR1=B.VAR1;
```

The resulting output:

VAR1	VAR2	VAR1	VAR3
ABC	10	ABC	20
ABC	10	ABC	25
ABC	15	ABC	20
ABC	15	ABC	25

The result of a many-to-many SQL join is a product of matches.

The MERGE statement's syntax would not change when performing a many-to-many merge, but a message will appear in the LOG stating duplicate BY values exist and it will produce undesirable results in your output SAS data set.

Outer Joins in SQL

Left, right, and full joins are types of outer joins. Outer joins can only be performed on two tables at a time and an outer join can contain rows that both do and do not match from the join's source tables.

A Left Join

A left join will join data if they are on the data set to the left, regardless if a match is found or not:

```
PROC SQL;
SELECT A.VAR1, VAR2, VAR3
FROM DATA1 A LEFT JOIN DATA3 B
ON A.VAR1=B.VAR1;
```

The resulting output:

VAR1	VAR2	VAR3
ABC	10	20
GHI	15	.
MNO	20	.

The word **ON** is used instead of the **WHERE** for matching the data sets.

The result is similar to checking if observations are on one of the data sets specified with the MERGE statement:

```
DATA MERGEIT;
MERGE DATA1(IN=ONA) DATA3(IN=ONB);
BY VAR1;
IF ONA;
```

A Right Join

A right join will join data if they are on the data set to the right, regardless if a match is found or not:

```
PROC SQL;
SELECT B.VAR1, VAR2, VAR3
FROM DATA1 A RIGHT JOIN DATA3 B
ON A.VAR1=B.VAR1;
```

The resulting output:

VAR1	VAR2	VAR3
ABC	10	20
JKL	.	25
PQR	.	30

The result is similar to checking if observations are on one of the data set specified with the MERGE statement:

```
DATA MERGEIT;
MERGE DATA1(IN=ONA) DATA3(IN=ONB);
BY VAR1;
IF ONB;
```

A Full Join

A full join will join data if they are both of the data sets:

```
PROC SQL;
SELECT *
FROM DATA1 A FULL JOIN DATA3 B
ON A.VAR1=B.VAR1;
```

The resulting output:

VAR1	VAR2	VAR1	VAR3
ABC	10	ABC	20
GHI	15	.	.
MNO	20	JKL	25
.	.	PQR	30

The COALESCE function is used to overlay columns:

```
PROC SQL;
  SELECT COALESCE(A.VAR1, B.VAR1) LABEL='VAR1',
         VAR2, VAR3
  FROM DATA1 A FULL JOIN DATA3 B
  ON A.VAR1=B.VAR1;
```

The resulting output:

VAR1	VAR2	VAR3
ABC	10	20
GHI	15	.
JKL	.	25
MNO	20	.
PQR	.	30

This produces a result that is similar to a match-merge that does no checking to see if an observation is on one data set or the other:

```
DATA MERGEIT;
  MERGE DATA1 DATA3;
  BY VAR1;
```

Joining More Than Two Tables

An associative, or three-way join, is much more efficient to do with PROC SQL than with traditional SAS code because it typically can be done in one step versus several steps that the traditional MERGE statement requires:

Table BENEFITS

FNAME	LNAME	CLAIMS
ANN	BECKER	2003
CHRIS	DOBSON	100
ALLEN	PARK	10392
BETTY	JOHNSON	3832

Table EMPLOYEE

FNAME	LNAME	STORENO
ANN	BECKER	33281
CHRIS	DOBSON	33281
EARL	FISHER	33281
ALLEN	PARK	31373
BETTY	JOHNSON	31373
KAREN	ADAMS	31373

Table FEBSALES

STATE	SALES	STORENO
WI	10103.23	32320
MI	31209.33	33281
MI	15132.43	33312
IL	8332.11	31212
IL	30083.22	31381
IL	26223.12	31373

```
PROC SQL;
  SELECT B.FNAME, B.LNAME, CLAIMS, E.STORENO, STATE
  FROM BENEFIT B, EMPLOYEE E, FEBSALES F
  WHERE B.FNAME=E.FNAME AND B.LNAME=E.LNAME
  AND E.STORENO=F.STORENO AND CLAIMS > 1000;
```

The resulting output:

FNAME	LNAME	CLAIMS	STORENO	STATE
ANN	BECKER	2003	33281	MI
ALLEN	PARK	10392	31373	IL
BETTY	JOHNSON	3832	31373	IL

The WHERE statement is not only important in helping us "link" the files together, but it is also used to subset the data.

An associative join in traditional SAS code would require several steps to accomplish the same results:

```
PROC SORT DATA=BENEFIT;
  BY FNAME LNAME;
RUN;
```

```
PROC SORT DATA=EMPLOYEE;
  BY FNAME LNAME;
RUN;
```

```
DATA TEST1;
  MERGE BENEFIT(IN=ONA) EMPLOYEE(IN=ONB);
  BY FNAME LNAME;
  IF ONA AND ONB;
  IF CLAIMS > 1000;
RUN;
```

```
PROC SORT DATA=TEST1;
  BY STORENO;
RUN;
```

```
PROC SORT DATA=FEBSALES;
  BY STORENO;
RUN;
```

```
DATA TEST2;
  MERGE TEST1(IN=ONA) FEBSALES(IN=ONB);
  BY STORENO;
  IF ONA AND ONB;
RUN;
```

```
PROC PRINT DATA=TEST2;
RUN;
```

Subqueries Versus Merging

Many queries can be written either as a subquery or as a join. Which one is used should be based on the following guidelines:

- ❑ If data are needed from more than one table, then a join must be performed
- ❑ when more than one query is required to subset the data then a subquery is used
- ❑ sometimes subqueries will be changed to a join by the SQL optimizer
- ❑ generally, a join is more efficient because the subquery is unable to directly apply the WHERE condition

Subquerying Against the Same Table

Subqueries, also known as inner queries, are evaluated before the outer query. To find out what states had average sales that were greater than the company's average sales, a subquery would be used to first find the company's average sales and compare that with the individual state's average sales:

```
PROC SQL;
SELECT STATE, AVG(SALES) AS AVGSALES
FROM USSALES
GROUP BY STATE
HAVING AVG(SALES) >
  (SELECT AVG(SALES)
   FROM USSALES);
```

The resulting output:

STATE	AVGSALES
IL	21244.14
MI	26670.83

Subqueries Against Different Tables

If we wanted to know what are the store names and the state they are located in that had over \$20,000 sales in February we could use a subquery:

```
PROC SQL;
SELECT STATE, STORENAM, STORENO
FROM USSALES
WHERE STORENO IN
  (SELECT STORENO
   FROM FEBSALES
   WHERE SALES > 20000);
```

The resulting output:

STATE	STORENAM	STORENO
MI	WOODBIDGE GROCERS	33281
IL	OAKBRIDGE GROCERY STORE	31983
IL	VICTOR'S FOOD CENTRE	31381
IL	SAVE U MONEY	31373

Unfortunately, a subquery can only contain a single variable on the SELECT statement. If more than one variable is needed to make a match then a unique key has to be created to select upon:

```
PROC SQL;
SELECT *
FROM EMPLOYEE
WHERE COMPRESS(LNAME || FNAME) IN
  (SELECT COMPRESS(LNAME || FNAME)
   FROM BENEFITS
   WHERE CLAIMS > 1000);
```

The resulting output:

FNAME	LNAME	STORENO
ANN	BECKER	33281
ALLEN	PARK	31373
BETTY	JOHNSON	31373

COMPRESSING the value and converting a numeric to character may be important in creating a unique key to match upon.

Similarly the NOT IN condition can be used to check if some condition has not been met. If we wanted to know what are the employee names and the stores they work for that have not had an insurance claim, we could use a subquery to find the result:

```
PROC SQL;
SELECT FNAME, LNAME, STORENO
FROM EMPLOYEE
WHERE COMPRESS(FNAME || LNAME) NOT IN
  (SELECT COMPRESS(FNAME || LNAME)
   FROM BENEFITS);
```

The resulting output:

FNAME	LNAME	STORENO
EARL	FISHER	33281
GARY	HOWE	33281
JACK	KELLER	33312
LARRY	MOORE	33312
NANCY	PAUL	33312
RICK	TENNY	33312
VIC	WATSON	31983
ARNIE	CARLSON	31983
DAVID	GELDER	31983
HARRY	JACKSON	31983
KELLY	LARSON	31983
MARY	NELSON	31381
PAULA	RILEY	31381
STEVE	WALDON	31381
KAREN	ADAMS	31373

Performance Issues

If possible, join the data rather than subquerying, since subqueries cannot directly apply the where. In some cases, a subquery will be changed to a join for you by the SQL optimizer.

Performance can sometimes be improved in an SQL join by indexing the tables beforehand, particularly if at least one of the tables is very large. Indexing can decrease efficiency when approximately 15 percent or more of the observations are being selected from the total number of observations. In this case, it is more efficient for SAS to read the file sequentially than to access the data via the index.

If you are going to make more than one query against a large table, performance can be improved by explicitly sorted the data set beforehand. This saves SQL from having to implicitly sort the data prior to joining.

Smaller table joins can be improved within SQL by increasing the buffersize on the PROC SQL statement. Before doing this, consult the paper by Paul Kent that is listed in the bibliography at the end of this paper.

In Summary

PROC SQL can be a better alternative than a traditional SAS merge particularly when merging large tables together. In most cases the traditional SAS MERGE is still more efficient when merging smaller data sets together than SQL.

It is always a good idea to do some benchmark testing of the various joining methods to see which is best for your particular situation, especially if you are developing code that will be used on a regular basis. Keep in mind that the joining method you choose today may become less efficient over time due to changes in the SQL optimizer from one version to the next and due to the underlying data structure changing over time (the data set may get wider, longer, shorter, etc.). It is not a bad idea to occasionally re-visit your benchmarking tests to determine if your current method is still the best choice.

Trademark Notice

SAS is a registered trademark of the SAS Institute Inc., Cary, NC, USA and other countries.

Bibliography

Paul Kent, "SQL Joins- The Long and The Short of It", presented at SUGI 20 in Orlando, FL and soon to be published in the *SUGI20 Proceedings*.

Howard Levine, "A New Approach to Outer Joins with More than Two Tables", presented at SUGI 20 in Orlando, FL and soon to be published in the *SUGI20 Proceedings*.

Jack Shoemaker, "Fuzzy Joins Using SQL", presented at SUGI 20 in Orlando, FL and soon to be published in the *SUGI20 Proceedings*.

SAS Institute Inc. (1989), *SAS® Guide to the SQL Procedure, Usage and Reference, Version 6, First Edition*, Cary, NC: SAS Institute Inc.

Questions or Comments

Any questions or comments regarding the paper may be directed to the author:

Kim L. Kolbe Ritzow
Systems Seminar Consultants
Kalamazoo Office
927 Lakeway Drive
Kalamazoo, MI 49001
Phone: (616) 345-6636
Fax: (616) 345-5793