Using the SQL Procedure in SAS® Programs

Susan E. Johnston, SAS Institute Inc., Cary, NC

ABSTRACT

The new SQL procedure combines the flexibility of the Structured Query Language (SQL) with the power of the SAS® System, Release 6.06. The SQL procedure uses SQL to create, modify, and retrieve data from SAS data sets and views derived from those data sets. You can also use the SQL procedure to join data sets and views with those from other database management systems through the SAS/ACCESS™ software interfaces. This paper describes some of the advantages and ways of using the SQL procedure in SAS programs. This session takes a task-oriented approach in introducing the SQL procedure to SAS users and in showing them how to use the procedure to write more efficient programs with the latest release of the SAS System.

OVERVIEW OF SQL

The SQL procedure implements the Structured Query Language for Version 6 of the SAS System. SQL is used by many of the database systems that the SAS System interfaces with through its SAS/ACCESS software, such as DB2™ and Rdb/VMS™ SQL is also used in relational database systems produced by IBM®, Oracle, Cullinet, Ingres, Computer Associates, and other database vendors. The SQL procedure will be fully described in a separate book when Release 6.06 of the SAS System is distributed.

The SAS System's SQL procedure gives you control over your data on three levels:

- You can retrieve data stored in tables and views using the SELECT statement. (The SAS data sets created and accessed by PROC SQL will be referred to as tables in this paper.) Or, you can use the VALIDATE statement to check the accuracy of your SELECT statement's syntax without actually executing it. You can also simply display a view definition, on your screen for example, using the DESCRIBE statement.
- You can create tables, views, and indexes on columns in tables using the CREATE statement; these tables and views can be stored permanently in SAS data libraries and referred to using librefs. Or, you can delete tables, views, and indexes using the DROP statement.
- You can add or modify the data values in a table's columns using the UPDATE statement or insert and delete rows with the INSERT and DELETE statements. You can also modify the table itself by adding, modifying, or dropping columns with the ALTER statement.

The PROC SQL statement also allows a number of options, which can be added, changed, or removed using the RESET statement.

Basic Concepts

An SQL table consists of rows and columns. Consider the sample Employee table (**Table 1**), which describes employees in a beach-supplies wholesale company. Each row of the table gives information pertaining to one employee. Each column of the table describes a category of information about an employee, such as an employee number, job title, or the employee number of that employee's manager.

Table 1 The Employee Table

EMPNUM	EMPNAME	EMPYEARS	EMPCITY	EMPTITLE	EMPBOSS
101	[illegible]	14	Ocean City	president	.
213	[illegible]	2	Virginia Beach	salesrep	201
214	[illegible]	1	Virginia Beach	salesrep	201
215	[illegible]	10	Ocean City	salesrep	201
216	[illegible]	6	Ocean City	salesrep	201
301	[illegible]	9	Wilmington	manager	101
314	[illegible]	5	Wilmington	salesrep	301
318	[illegible]	1	Myrtle Beach	salesrep	301
401	[illegible]	12	Charleston	manager	101
417	[illegible]	7	Charleston	salesrep	401

You will notice that the structure of a table is very similar to that of a SAS data set. A row is equivalent to a SAS observation and a column is equivalent to a SAS variable. Because an SQL table and a SAS data set are so nearly identical, *a table is considered a SAS data set in the SAS System.*

SQL can be used in the SAS System in two ways. As a SAS procedure, the SQL procedure processes SQL statements in a SAS program or interactive session. Other SAS procedures can read views or read and update tables created through the SQL procedure. Thus, SQL is fully integrated into the base SAS software.

Sample Tables

The examples in this paper are based on the Employee table, the Product table (**Table 2**) and the Invoice table (**Table 3**). These tables could be created as SAS data sets using the SAS DATA step or as SQL tables using the CREATE and INSERT statements. The sample tables are stored permanently in a SAS data library referred to by the libref "SUGI."

The Employee table is shown above. The Product table describes the products sold by the sample beach-supplies wholesale company, plus their per-item cost and list price.

Table 2 The Product Table

PRODNAME	PRODCOST	PRODLIST
flippers	$16	$20
jet ski	$2,150	$2,675
kayak	$190	$240
raft	$5	$7
snorkel	$12	$15
surfboard	$615	$750
windsurfer	$1,090	$1,325

The invoice table lists information on the company's sales: an invoice number, a customer name and store number, the employee number of the sales representative (salesrep) who made the sale, and the quantity and invoice price per item sold.

Table 3 The Invoice Table

INVNUM	CUSTNAME	CUSTNUM	EMPNUM	PRODNAME	INVQTY	INVPRICE
280	Beach Land	16	215	snorkel	20	$14
310	Coast Shop	3	318	windsurfer	2	$1,305
340	Coast Shop	5	318	flippers	15	$19
350	Coast Shop	5	318	raft	40	$6
360	Coast Shop	5	318	snorkel	10	$15
380	Coast Shop	14	417	windsurfer	1	$1,325
400	Del Mar	3	417	kayak	3	$230
410	Del Mar	8	417	raft	40	$6
450	New Waves	3	215	flippers	5	$20
480	New Waves	6	213	surfboard	4	$735
500	Surf Mart	101	417	snorkel	20	$14
510	Surf Mart	101	417	surfboard	2	$740
530	Surf Mart	118	318	flippers	15	$19
570	Surf Mart	127	318	surfboard	3	$740

SQL QUERIES AND THE DATA STEP

One of the strongest advantages of the Structured Query Language is its ability to "query" or retrieve data from tables with relatively few lines of code. The SQL procedure allows you, for example, to retrieve and manipulate data from multiple tables by just listing the tables to be joined and the conditions under which the joining should take place. The following section compares an SQL query with a SAS program. They both produce the same output table, but the SQL query is faster and shorter. The SQL query is explained and shown first, followed by the SAS program.

Queries

A PROC SQL *query* selects and displays columns and rows from one or more tables. A query retrieves the columns specified in the SELECT clause from table(s) specified in the FROM clause. Certain conditions (or predicates) for including rows in the output table can be specified using an optional WHERE clause. Predicates are composed of expressions, such as `custname='New Waves'` shown in the query below.

```
proc sql;
select custname, custnum
   from sugi.invoice
   where custname='New Waves';
```

```
CUSTNAME    CUSTNUM
-------------------
New Waves         3
New Waves         6
```

A query can also perform statistical summation, grouping, and sorting using the above clauses and others described later in this paper.

The SQL procedure executes a query without using the RUN statement. It also automatically displays the query's output on the screen (or sends it to a list file) without using the PRINT procedure, unless you specify the statement option NOPRINT. Because the SQL procedure is a resident procedure, you do not need to repeat the PROC SQL statement with each query.

In the following query, columns are selected from all three of the sample tables. A new column is created (TOTSALE) with an arithmetic expression that uses two existing columns as its operands; you can optionally specify the new column's name and format. All the columns except EMPNUM are selected from the Invoice table. Since the salesreps' names (EMPNAME) appear in the Employee table and you want to use their names in the output table instead of their employee numbers (EMPNUM), you specify the EMPNAME column in the SELECT clause and the Employee table in the FROM clause.

Finally, specifying the Product table in the FROM clause provides the cost information needed to satisfy the WHERE clause conditions. Notice that the PRODCOST column has not been selected for output. You can match columns from multiple tables or use columns in expressions in the WHERE clause based on any table listed in the FROM clause, regardless of whether the columns appear in the SELECT clause. Therefore, the example query below displays products that cost over $600 per item, the total sale per product, who sold them, and related sales information.

```
select invnum, custname, custnum, empname, invoice.prodname,
     invqty, invprice, (invqty*invprice) as totsale format=dollar.
   from sugi.invoice, sugi.employee, sugi.product
   where invoice.empnum=employee.empnum and
        invoice.prodname=product.prodname and prodcost>=600;
```

INVNUM	CUSTNAME	CUSTNUM	EMPNAME	PRODNAME	INVQTY	INVPRICE	TOTSALE
310	Coast Shop	3	Nick	windsurfer	2	$1,305	$2,610
380	Coast Shop	14	Sam	windsurfer	1	$1,325	$1,325
480	New Waves	6	Joe	surfboard	4	$735	$2,940
510	Surf Mart	101	Sam	surfboard	2	$740	$1,480
570	Surf Mart	127	Marvin	surfboard	3	$740	$2,220

When this query was run under the MVS/XA™ operating system on an IBM computer, it used 0.05 of a CPU second; the results will vary from system to system.

Join Queries

When more than one table is listed in the FROM clause of a query, as in the example above, the tables are joined to form one output table. These queries are called *join queries*, or more simply, *joins*.

Conceptually, a join query with a WHERE clause is evaluated in two phases. First, the FROM clause is processed. PROC SQL internally builds a temporary join table by combining each row from the first table with every row from the second table. In a three-table join, this two-way temporary table is then joined with the third table in the same way. This results in a Cartesian product of the three tables.

The WHERE expression is then applied to the Cartesian product. Rows are selected for the output table that satisfy the condition(s) listed in the WHERE clause. Omitting the WHERE clause can have a significant performance impact, so it is strongly recommended that you include it in a join query. The rest of the query is then evaluated, and the output table is displayed on your screen (during an interactive session) or sent to a list file (in batch mode).

While it is helpful conceptually to imagine that the SQL procedure builds an internal Cartesian product for every join, this is usually not the case. The SQL procedure employs a sophisticated optimizer that evaluates the join query and automatically processes it by the most efficient method.

Using a Join Query Instead of MERGE Operations

In just six lines of code, the SQL procedure can join the three tables so that only the products costing more than $600 are displayed. The SQL procedure does this by performing some of the same operations that the SAS DATA step and the SORT, MERGE, and PRINT procedures perform. A SAS program can also retrieve and output the same information, but it involves a DATA step to add the new variable TOTSALE, sorting the data sets four times by two variables, and merging the data sets twice, as shown below.

```
data sugi.invoice;
   set sugi.invoice;
   totsale=invqty*invprice;
   format totsale dollar.;
run;
proc sort data=sugi.invoice;
   by empnum;
run;
proc sort data=sugi.employee;
   by empnum;
run;
data newinv(keep=invnum custname custnum empname prodname
                 invqty invprice totsale);
   merge sugi.invoice sugi.employee;
   by empnum;
run;
proc sort data=newinv;
   by prodname;
run;
proc sort data=sugi.product;
   by prodname;
run;
data newinv(keep=invnum custname custnum empname prodname
                 invqty invprice totsale);
   merge newinv sugi.product;
   by prodname;
   if prodcost>=600;
   if invnum^=.;
run;
proc print data=newinv noobs;
   var invnum custname custnum empname prodname invqty invprice
       totsale;
run;
```

INVNUM	CUSTNAME	CUSTNUM	EMPNAME	PRODNAME	INVQTY	INVPRICE	TOTSALE
480	New Waves	6	Joe	surfboard	4	$735	$2,940
570	Surf Mart	127	Marvin	surfboard	3	$740	$2,220
510	Surf Mart	101	Sam	surfboard	2	$740	$1,480
310	Coast Shop	3	Nick	windsurfer	2	$1,305	$2,610
380	Coast Shop	14	Sam	windsurfer	1	$1,325	$1,325

When this SAS program was run under the MVS/XA operating system on an IBM computer, it used 0.52 of a CPU second compared with the 0.05 of a second that it took to run the comparable PROC SQL query. Thus, for faster data retrieval and faster coding, you can use the SQL procedure in your SAS programs.

PROC SQL VIEWS

Queries provide the basis for other SQL statements. The SQL procedure allows you to name and store a query so that its output can be used as input to another PROC SQL query, SAS procedure, or the DATA step. When a query is given a name and stored for later use, it is called a view. Views are described and shown here, and the advantages of using views in SAS programs are described in the following section.

While you can refer to a view in queries, SAS procedures, and the DATA step as if it were a table, a view is not the same as a table. A table is stored data while a view is a stored query. You can update data in a table but, in the current release, you cannot update data through a view because a view is just the definition of a virtual table. When a view is executed, it derives its data from one or more tables or views: the data it processes are often a subset or superset of the data in its underlying table(s) or view(s).

In this example, the join query used earlier is assigned the name "Highcost" and the libref "SUGI" to make it a permanently stored view. When this view is executed, it will be stored in the SAS data library pointed to by its libref. A message is displayed on the SAS log indicating that the view has been created.

```
proc sql;
create view sugi.highcost as
select invnum, custname, custnum, empname, invoice.prodname,
     invqty, invprice, (invqty*invprice) as totsale format=dollar.
   from sugi.invoice, sugi.employee, sugi.product
   where invoice.empnum=employee.empnum and
         invoice.prodname=product.prodname and prodcost>=600;

NOTE:  View SUGI.HIGHCOST has been defined.
```

The PROC SQL statement is repeated here because this CREATE VIEW statement follows a DATA step execution, which "turned off" the resident SQL procedure.

Once a view has been executed, you can use it in the DATA step or in any SAS procedure that accepts a SAS data set, as shown below.

```
proc print data=sugi.highcost;
run;
```

The Highcost view's output is the same as that shown in the previous section.

You can also use a view in other PROC SQL queries and in creating new tables and views. The following query joins the Highcost view with the Product table to determine the profit of each sale. This is done by computing the difference between a product's invoiced price and its cost and multiplying that value by the quantity sold. A new column, "Profit Over Cost," is added to the table that lists the profit per sale. It is specified using the DOLLAR. format and named using a label. H and P are assigned as aliases to the view and table names, respectively, to make coding the join query faster. The ORDER BY clause sorts the output in ascending order.

```
proc sql;
select invnum, empname as salesrep, h.prodname as product,
   invqty, invprice, (invprice-prodcost)*invqty
   label='Profit Over Cost' format=dollar.
   from sugi.highcost as h, sugi.product as p
   where h.prodname=p.prodname
   order by invnum;
```

INVNUM	SALESREP	PRODUCT	INVQTY	INVPRICE	Profit Over Cost
310	Nick	windsurfer	2	$1,305	$430
380	Sam	windsurfer	1	$1,325	$235
480	Joe	surfboard	4	$735	$980
510	Sam	surfboard	2	$740	$250
570	Marvin	surfboard	3	$740	$375

Advantages of Using Views

Using views in SAS procedures and the DATA step can increase the power and flexibility of your SAS programs:

- Instead of using multiple DATA steps to merge SAS data sets by common variables, a view can be constructed that performs a multi-table join.
- Disk space may be saved by storing a view definition, which stores only the PROC SQL query and not the actual data.
- Views can ensure that input data sets are always current, since data are derived from views at execution time.
- An Information Center can provide powerful views for use by the SAS user community, thereby avoiding the need for all users to learn SQL details.
- Views can reduce the impact of data design changes on users, by changing a query stored in a view without changing the characteristics of the view's query result.

For example, if data stored in one table are separated into two tables for design improvements, and a view on that single table is changed so that it now joins the two tables, the view's query results would remain the same.

- The SQL procedure can be used to join its own views or join views from other database systems to form one output table. You need to use the SAS/ACCESS interface to DB2, for example, to retrieve views from DB2 that the SAS System can read. See Paul Kent's SUGI14 paper, "Views–Your Window on Data," for more information on SAS/ACCESS views.
- Using SAS/SHARE® software, a view can join together SAS data sets that reside on different host computers, presenting the user with an integrated "view" of distributed company data.

EXTENDING SAS SYSTEM CAPABILITIES WITH PROC SQL

The SQL procedure builds on and extends some of the SAS System capabilities in the areas of merging data sets, set operations, and in the one-step remerging of statistical results with the data that generated them. Each of these areas is described and illustrated in the following sections.

Outer Joins

The DATA step allows you to merge SAS data sets in a number of ways, and you have seen in previous sections how the SQL procedure joins tables. A conventional or *inner join*, as shown in "Join Queries," returns an output table for all the rows in a table that have one or more matching rows in the other table(s), as specified by the condition(s) in the WHERE clause. This is the only kind of join that most SQL databases allow. However, the SQL procedure also provides *outer joins*, which further extend SQL's joining capabilities.

Outer joins are inner joins that have been augmented with rows that did not match with any row from the other table(s) in the join. Therefore, the output table includes rows that match and rows that do not match from the join's source tables. Outer joins can only be performed on two tables (or views) at a time. There are three kinds of outer joins: left, right, and full.

A *left outer join*, specified with the keywords LEFT JOIN and ON, lists matching rows and adds row(s) from the left-hand table that do not match with any row in the right-hand table of the join. The ON clause replaces the WHERE clause of an inner join, although it serves the same purpose of setting conditions for selecting rows for output. In the example below, a left outer join is used to list all the products (from the left-hand Product table) and sales (if any) of those products to the Beach Land store.

```
select p.prodname, i.invqty, i.invprice, p.prodlist,
       i.invqty*i.invprice format=dollar. label='Invoice Amount'
   from sugi.product p left join sugi.invoice i
   on p.prodname=i.prodname and i.custname='Beach Land';
```

This output table indicates that only one of the possible seven products was sold to the Beach Land store. An inner join could not be used here because it would list only the products sold to the Beach Land store, instead of all the products sold, including those to Beach Land.

PRODNAME	INVQTY	INVPRICE	PRODLIST	Invoice Amount
flippers	.	.	$20	.
jet ski	.	.	$2,675	.
kayak	.	.	$280	.
raft	.	.	$7	.
snorkel	20	$14	$15	$280
surfboard	.	.	$750	.
windsurfer	.	.	$1,325	.

A *right outer join*, specified with the keywords RIGHT JOIN and ON, lists matching rows and adds row(s) from the right-hand table that do not match with any row in the left-hand table of the join. A *full outer join*, specified with the keywords FULL JOIN and ON, has all the rows of the Cartesian product of the two tables for which the ON clause is true, plus rows from each table that do not match any row in the other table.

Summary Functions and the Remerge Capability

The SQL procedure can process almost all of the SAS System's functions, including its 16 statistical functions, such as AVG, SUM, MAX, and VAR. It also allows you to evaluate summary functions with non-summary expressions within a single query. This capability and summary functions are described in this section.

Summary functions produce a statistical summary of the entire table listed in a query's FROM clause or for each group specified in a GROUP BY clause. These functions reduce all the values in each row or column in a table to one "summarizing" or "aggregate" value. For example, the sum (one value) of a row results from adding all the values in the row; the values in each row of the table are added in this way. Or, the maximum value in a column is determined by considering all the values in that column and selecting the highest one; the values in each column of the table are evaluated likewise.

A summary function's value for a particular group is determined using the data associated with that group. Groups are usually defined by column names, such as EMPTITLE, and specified using a GROUP BY clause. If this clause is omitted, each row in the table is considered to be a single group.

A summary function can appear in the SELECT clause or HAVING clause of a query. A HAVING clause can be considered a WHERE clause for each group of a query involving summary statistics. When a summary function appears in either of these clauses along with one or more other arithmetic expressions or columns (that is, variables), *the SQL procedure may have to remerge the results from the summary function into the original data.*

It is necessary to remerge the data when a summary function's result value is involved in a calculation or is compared with columns that are not in the query's optional GROUP BY clause. Remerging involves two passes through the data. The first pass calculates the summary function's values for each row and the second one redistributes the result values across the rows (that is, original data) of the output table. The SQL procedure allows you to accomplish these two tasks in a single query.

The following example shows how the remerging process works. Here the output table lists employees, grouped by their job titles, who have worked more than the average number of years in their particular job title.

```
select empname, emptitle, empyears
   from sugi.employee
   group by emptitle
   having empyears>avg(empyears);
```

In the SQL procedure's first pass through the data, it groups the rows into three categories, according to each employee's EMPTITLE. It then calculates the average number of EMPYEARS per category. This internal arrangement and the average years per job title are shown below:

```
EMPNAME    EMPTITLE   EMPYEARS    avg(empyears)
-------------------------------
Sally      manager           9
Betty      manager           8
Chuck      manager          12          9.66

Herb       president        18         18

Fred       salesrep          6
Wanda      salesrep         10
Nick       salesrep          1
Marvin     salesrep          5
Jeff       salesrep          1
Sam        salesrep          7
Joe        salesrep          2          4.57
```

In the procedure's second pass through the data, the HAVING expression is evaluated and the data from the original table (EMPYEARS) are remerged with the results of the summary function. That is, the HAVING expression compares the EMPYEARS value for each row in the table (and EMPYEARS changes with each row) with the average number of years worked for each job title (a constant value). When an employee's number of years worked is greater than the average for the job title, the expression evaluates to true for that row.

```
EMPNAME    EMPTITLE   EMPYEARS    avg(empyears)    having
-------------------------------
Sally      manager           9          9.66        FALSE
Betty      manager           8          9.66        FALSE
Chuck      manager          12          9.66        TRUE

Herb       president        18         18           FALSE

Fred       salesrep          6          4.57        TRUE
Wanda      salesrep         10          4.57        TRUE
Nick       salesrep          1          4.57        FALSE
Marvin     salesrep          5          4.57        TRUE
Jeff       salesrep          1          4.57        FALSE
Sam        salesrep          7          4.57        TRUE
Joe        salesrep          2          4.57        FALSE
```

When the evaluations are completed, the rows for which the HAVING expression evaluates to true are displayed in the final output table. A message appears on the SAS log to make you aware that remerging has occurred.

```
NOTE: The query as specified involves remerging the summary
      statistics back with the data that created those statistics.
```

```
EMPNAME    EMPTITLE    EMPYEARS
-------------------------------
Chuck      manager           12
Fred       salesrep           6
Wanda      salesrep          10
Marvin     salesrep           5
Sam        salesrep           7
```

Evaluating summary functions with non-summary expressions in a single query is a SAS System extension to the ANSI Standard for SQL.*

Set Operations

The SET statement in the DATA step allows you to concatenate SAS data sets. The SQL procedure performs this task using the set operator OUTER UNION. In addition, the SQL procedure provides the traditional set operators from relational algebra: UNION, EXCEPT (difference), and INTERSECT. Using set operators gives you more flexibility in handling complex queries and in combining the results of queries.

When two or more queries are connected with set operators, they form a single *query-expression* and produce a single output table. The queries within that query-expression are then referred to as *table-expressions* for clarity.

A query-expression with set operators is evaluated as follows: each table-expression is evaluated to produce an (internal) intermediate result table. Each intermediate table then becomes an operand linked with a set operator to form an expression, for example, A UNION B. If the query-expression involves more than two table-expressions, the result from the first two becomes an operand for the next set operator and operand, for example, (A UNION B) EXCEPT C, ((A UNION B) EXCEPT C) INTERSECT D, and so on. Evaluating a query-expression produces a single output table that is displayed on your screen or sent to a list file.

The UNION operator produces an output table that contains all the unique rows produced by both table-expressions. That is, the output table contains rows that are the intermediate result from the first table-expression or the second table-expression or both. The names of the columns in the output table are the names of the first table-expression in the query. (In the first table-expression that follows, PRODNAME is temporarily renamed with the alias PRODUCT to emphasize this point.) The following example query displays products that have a proposed profit margin of at least 25% OR more than 75 of the products have been sold.**

```
title 'UNION Example';
select prodname as product
   from sugi.product
   where (prodlist-prodcost)/prodcost>.25
union all
select prodname
   from sugi.invoice
   group by prodname
   having sum(invqty)>75;
```

```
 UNION Example

 PRODUCT
 ----------
 kayak
 raft
 raft
```

Notice that two products are listed in the output because of their profit margins, while only "raft" satisfies the HAVING clause condition in the second table-expression. "Raft" is listed twice because the UNION ALL operator causes duplicate rows to remain in the output table.

The EXCEPT operator produces an output table that has rows resulting from the first table-expression that are not in the result of the second table-expression. EXCEPT is a set difference operator: if the first intermediate table has at least one occurrence of a row that is not in the second intermediate table, that row is included in the output table. The next query lists products having a proposed profit margin that is less than 25% AND fewer than 75 items of each product have been sold.

```
title 'EXCEPT Example';
select prodname, (prodlist-prodcost)/prodcost as margin
   from sugi.product
   where (prodlist-prodcost)/prodcost<.25
except
select prodname, sum(invqty)
   from sugi.invoice
   group by prodname
   having sum(invqty)>74;
```

EXCEPT Example

PRODNAME	MARGIN
jet ski	0.244186
surfboard	0.219512
windsurfer	0.215596

"Jet ski" is included in the output table because it satisfies both conditions of having a margin of less than 25% and having fewer than 75 sold (none of them sold). It also satisfies the difference operation because "jet ski" is a result row from the first table-expression only and does not appear in the result from the second table-expression.

The INTERSECT operator produces an output table having rows that belong to or are common to the intermediate results of both table-expressions.

The OUTER UNION operator is very similar to the SAS DATA step SET statement. The OUTER UNION concatenates the intermediate results from each table-expression, so that the final output table contains all the rows produced by the first table-expression followed by all the rows produced by the second table-expression. Columns with the same name are in separate columns in the output table. See the next section for more information on the OUTER UNION operator.

Optional Keywords for Set Operations

When you specify a set operator in a query-expression, the SQL procedure allows you to use two optional keywords, CORRESPONDING and ALL, to modify the results in your output tables.

The CORRESPONDING keyword causes the SQL procedure to match the columns by name, instead of ordinal position, which most SQL databases allow. Columns that do not match by name are excluded from the output table, except when the OUTER UNION operator is used.

When using an OUTER UNION CORRESPONDING operator, the columns with unique names are retained in the output table. For columns with the same name, if a value is missing from the result of the first table-expression, the value in that column from the second table-expression is inserted. For example, if an OUTER UNION produced the following output table on the left, an OUTER UNION CORRESPONDING would produce the output on the right, where the X columns from the two table-expressions would be concatenated. Therefore, you may want to use an OUTER UNION CORRESPONDING operation in some SAS programs rather than a DATA step SET statement.

OUTER UNION

X	Y	X	Z
1	one	.	
2	two	.	
4	four	.	
.		1	one
.		2	two
.		5	five

OUTER UNION CORR

X	Y	Z
1	one	
2	two	
4	four	
1		one
2		two
5		five

The set operators automatically eliminate duplicate rows from their output tables. The optional ALL keyword causes the duplicate rows to be left in, thereby reducing the processing by one step. Therefore, you can use the ALL keyword to improve the performance of your query-expressions.

SUMMARY

This paper has described some of the features of the new SQL procedure, which implements the Structured Query Language for Version 6 of the SAS System. It has shown some of the ways that you can use the SQL procedure to write more efficient SAS programs. It has covered the following topics:

- basic concepts and terms used in the SQL procedure
- PROC SQL queries and joins compared with a SAS program that does the same task
- views and their uses with SQL queries, SAS procedures, and the DATA step
- PROC SQL features that extend the SAS System's capabilities: outer joins, set operators, and remerging summary function results with the data that generated them.

For more information, watch for the SQL procedure book when Release 6.06 of the SAS System becomes available.

End Notes

* For more information, see Phil Shaw, Editor (November 1988), *ISO-ANSI Database Language SQL*, ANSIX3.135, International Standards Organization and American National Standards Institute.

** Special thanks to Henrietta Cummings of SAS Institute Inc. for providing these examples and technical assistance.