PROC SQL: From SELECT to Pass-Through SQL

Christopher W. Schacherer, Clinical Data Management Systems, LLC
Michelle A. Detry, University of Wisconsin-Madison School of Medicine and Public Health

ABSTRACT

Many data analysts and programmers begin working with SAS® as a data management and analytic tool after having developed reporting and data manipulation skills in relational database management systems such as Oracle®, SQL Server®, DB2®, or the like. Other SAS users have been utilizing DATA STEP programming and MERGE statements for years and may have only discovered PROC SQL more recently. For both of these groups (as well those new to SAS who do not have a background in SQL programming) the present work provides extensive examples of using PROC SQL to extract, join (merge), summarize, and update data for reporting and analysis. In addition to presenting information on performing equi-joins, left joins, and union joins, this paper provides in-depth explanations of the use of summary functions and "group by", "order by", and "having" clauses as well as information on inserting, updating, and deleting rows in existing SAS datasets. Finally, this paper provides examples of how to establish connections to remote databases using SAS/ACCESS and LIBNAME statements and explains how proprietary database languages such as Oracle's PL/SQL and Microsoft's T-SQL can be utilized in PROC SQL queries.

INTRODUCTION

PROC SQL is one of the most powerful data management tools available to SAS programmers. It enables SAS programs to create, update, or modify SAS datasets, generate printed output based on datasets or subsets, and aggregate summary statistics describing a dataset. Unlike other PROCs that perform a fixed set of related functions controlled by MODELs, BY groups, and OPTIONS, PROC SQL actually exposes an entire, highly flexible language (Structured Query Language – SQL) to the SAS programmer—acting as the SAS System's SQL engine and accepting any ANSI-standard SQL statement. PROC SQL has grown in popularity since its introduction in SAS 6.06 not only because of the wide range of functionality that if affords users, but also because of the growth of relational database management systems, the relatively intuitive form of the SQL language for operating against these databases, and, the ease with which SQL programmers can transition to working in the SAS environment by leveraging their existing expertise against new analytic challenges.

The popularity of PROC SQL has also been stimulated by several SUGI/SGF papers that helped experienced DATA STEP programmers compare and contrast BASE SAS PROCs and DATA STEP methods for combining datasets to their corollaries in PROC SQL (Dickstein & Pass, 2002; Foley, 2005; and Whitlock, 2001). As these examples began winning over some "DATA Step Die-hards" (Williams, 2008) PROC SQL gained legitimacy in the SAS community, and several user-group papers and books began to skip this comparison and simply provided strong introductory-to-advanced tutorials on PROC SQL techniques—see, for example, Lund (2005), Lafler (2003, 2004a, 2004b, 2005), and DeFoor (2006). PROC SQL had clearly arrived as an important tool for SAS data management and analysis.

The present paper continues the tradition of introducing new users to PROC SQL and attempts to add to the body of knowledge by covering the breadth and depth of PROC SQL techniques used by data management professionals.

BASIC SYNTAX

SELECT/FROM

Far and away the most common use of PROC SQL is to select data from a dataset—i.e., to "query" a dataset. In its simplest form, a PROC SQL query consists of only two required parts—the SELECT statement and the FROM clause.

```
PROC SQL;                                    PROC PRINT DATA=work.claim_detail;
    SELECT *                                 RUN;
      FROM work.claim_detail;
QUIT;
```

In the preceding example, the asterisk (*) is used as a wildcard to indicate that we want the PROC SQL query to return the values for all variables found in the dataset "claim_detail". The output generated by this query is equivalent to that produced by executing a PROC PRINT for the same dataset. In both cases, the dataset is written out to the current default output destination—by default the SAS Output Window. By adding a CREATE TABLE command, the destination of the query is changed from the default destination to a new dataset.

The result is equivalent to creating a copy of the dataset with a DATA STEP.

```
PROC SQL;                                      DATA work.gross_charges;
 CREATE TABLE work.gross_charges AS             SET work.claim_detail;
  SELECT *                                     RUN;
    FROM work.claim_detail;
QUIT;
```

Of course, the purpose (and power) of the PROC SQL SELECT statement is to enable the specification of which variables one wants to extract from the source dataset—and in which order. In the following examples we create the dataset "gross_charges" by selecting variables (e.g., billing_id and gross_charges) from the "claim_details" dataset. Note that in creating this new dataset we can also easily specify the order of the variables in the new dataset, which can facilitate the visual review of datasets with a large number of variables.

```
PROC SQL;                                      DATA work.gross_charges;
 CREATE TABLE work.gross_charges AS             SET work.claim_detail;
  SELECT billing_id, gross_charges             KEEP billing_id gross_charges
     FROM work.claim_detail;                   RUN;
QUIT;
```

We can even create new variables based on fixed, literal values or on calculations performed against the other variables in the dataset. In the following example, "billing_note" will contain the value "Pending Review" for every record in the dataset and "net_charges" will contain the sum of "gross_charges" and "adjustments".

```
PROC SQL;
 CREATE TABLE work.gross_charges AS
  SELECT billing_id, gross_charges,
                 'Pending Review' as billing_note,
                 gross_charges + adjustments AS net_charges
     FROM work.claim_detail;
QUIT;
```

WHERE

The SELECT statement gives us complete control over the variables included in the dataset (or "table" as we refer to these data objects in the SQL language) that is created by a PROC SQL select statement. In order to exercise control over which <u>records</u> are included in the target dataset, the WHERE clause is used.

```
PROC SQL;
 CREATE TABLE work.jul2010_charges AS
 SELECT *
   FROM work.claim_detail
  WHERE billing_date BETWEEN '01JUL2010'D AND '31JUL2010'D;
QUIT;
```

In the preceding example, we create a subset of the "billing_detail" dataset comprised of records generated in July, 2010. Likewise, we might want to subset the dataset to extract only those billing line-items over a specified gross charge amount.

```
PROC SQL;
 CREATE TABLE work.gross_charges AS
 SELECT *
   FROM work.claim_detail
  WHERE gross_charges > 10000;
QUIT;
```

Or, we might want to apply multiple criteria (e.g., those billing records generated within a specific date range that also exceed a specific charge value).

```
PROC SQL;
 CREATE TABLE work.gross_charges AS
 SELECT *
   FROM work.claim_detail
  WHERE gross_charges > 10000 AND
        billing_date BETWEEN '01JUL2010'D AND '31JUL2010'D;
QUIT;
```

Conversely, we could select records based on any one of a set of criteria being met. In the following example, the dataset "charge_range_check" will contain all records in the dataset "claim_detail" that EITHER have a negative value or exceed $100,000.

```
PROC SQL;
 CREATE TABLE work.charge_range_check AS
 SELECT *
   FROM work.claim_detail
  WHERE gross_charges < 0 or
        gross_charges > 100000;
QUIT;
```

Criteria specified in the WHERE clause can also be combined to evaluate more complex criteria. In the following example, we evaluate billing records for specific combinations of medical procedure codes and the charge value that triggers further review of the claim.

```
PROC SQL;
 CREATE TABLE work.july_charge_reviews AS
 SELECT *
   FROM work.claim_detail
  WHERE (cpt_code IN (43846, 55845, 29806, 22554) AND
        gross_charges > 10000) OR
        (cpt_code IN (33208, 44960, 47600, 64721) AND
        gross_charges > 5000);
QUIT;
```

This query would select a record with code 33208 and a charge of $6,152 but would not select records with code 43846 and a charge of $7,238. This Boolean logic can be extended to enforce increasingly sophisticated logic constraints on the rows of data to be retrieved—with the combination of AND and OR operations following the normal rule that the operations enclosed in the innermost set of parentheses are performed first, followed by the next level of parentheses, and so on.

ORDER BY

Having defined which variables (SELECT) and which rows (WHERE) are extracted to the target dataset, the ORDER BY clause can be used to sort the dataset. Perhaps the account representative that works with "Company XYZ" likes her data to be sorted by billing date (from the end of the month to the beginning), account number, claim id, and claim line item number. The following ORDER BY clause will impose that exact sort order on the target dataset.

```
PROC SQL;
 CREATE TABLE work.July_XYZ_Co_Claims AS
    SELECT bill_date, account_no, claim_id, claim_line_item, dx_code AS diagnosis,
           line_item_charge AS claim_line_item_amount
      FROM work.July_2010_Claims
     WHERE client_id = 'XYZ Co.'
  ORDER BY bill_date DESC, account_no, claim_id, claim_line_item;
QUIT;
```

The ORDER BY clause, by default, orders data in ascending order, so the DESC (descending) option is specified for the order of the "bill_date" and the other variables are allowed to use the default (ascending) order. For each billing date (sorted in descending order), the records will be sorted in ascending order of account number, within a given account number the records will be sorted in ascending order by claim id, and so on. You could also specify this ORDER BY clause using explicit sort orders for each variable as follows:

```
     ORDER BY bill_date DESC, account_no ASC, claim_id ASC, claim_line_item ASC;
```

Explicitly indicating the sort order of your variables is especially helpful when mixing ascending and descending sort orders in your code, and it helps clarify the intent of your sort order when you review your code at a later date.

```
     ORDER BY bill_date DESC, account_no ASC, claim_id DESC, claim_line_item ASC;
```

INSERT, UPDATE, DELETE, AND ALTER TABLE

Finally, in this introduction to the basic PROC SQL syntax, we feel it is important to give a few brief examples of other SQL commands that are often overlooked by users starting out with PROC SQL as a way to extract and transform data. Because so much of the focus of using PROC SQL is to create new datasets "from the ground up" by selecting the appropriate records and columns from a source dataset, we often overlook the fact that PROC SQL can be equally valuable to us as a data management tool due to its ability to insert, update, and delete records in a SAS dataset.

Suppose, for example, that you have a dataset of your customers, members, or patients that you want to update with a current "snapshot" of that population at the end of each month. This is commonly done by healthcare providers, employers, and health insurance companies to study the changes to a population over time. At the close of each month, you might extract this information from an operational system and insert it into an analytic dataset or relational database management system table as follows:

```
   PROC SQL
     INSERT INTO analytic.membership
        (lname,fname,memberno,gender,dob,plan,account,client_id,load_date)
     SELECT lname,fname,memberno,gender,dob,plan,account,client_id,'31JUL2010'D
       FROM ops.members
      WHERE status = 'Active';
   QUIT;
```

This INSERT statement takes all records in the "members" table with a "status" of 'Active' from our operational system and inserts those records (along with the date of the current data load) into our analytic archive "membership".

Alternatively, if you need to insert records containing data that does not exist in another dataset, you can manually insert records rather than SELECTing the records to INSERT. Suppose in our monthly snapshot we needed to manually insert a single record that was somehow missed by our extract from the operational system. This could be achieved in a manner similar to the following[1]:

```
   PROC SQL;
     INSERT INTO analytic.membership
        (lname,fname,memberno,gender,dob,plan,account,client_id,load_date)
     VALUES
        ('Jones','Mary',123661,'F','01MAY1963'D,'SMSD','AU123661','CD','31JUL2010'D);
   QUIT;
```

Of course, if you are inserting records into a dataset, sooner or later you may also need to delete records. Suppose we loaded the July 2010 membership data above using an INSERT statement that loaded the "lname" and "fname" variables into the wrong columns in our archive dataset:

[1] However, doing such "hard-coded" inserts in production code is not recommended, because this record may be availabile in the source data at the time of the next execution of the program—causing potential for duplicate records to be loaded.

```
PROC SQL;
  INSERT INTO analytic.membership
      (lname,fname,memberno,gender,dob,plan,account,client_id,load_date)
  SELECT fname,lname,memberno,gender,dob,plan,account,client_id,'31JUL2010'D
    FROM ops.members
   WHERE status = 'Active';
QUIT;
```

To remove these records before reloading the data, we would first delete the records that were loaded in error.

```
PROC SQL;
  DELETE FROM analytic.membership
   WHERE load_date = '31JUL2010'D;
QUIT;
```

Note here that the WHERE clause works in the same fashion with a DELETE statement as it does with a SELECT statement—by specifying the rows on which we want the PROC SQL statement to operate. In this example, however, rather than specifying which records to select, it is dictating which records will be deleted.

Finally, we can use PROC SQL to ADD new variables to an existing dataset and UPDATE them. In the following example, we add the 'flag' variable "dental_coverage" to our dataset and specify its data type (character) and length (1). We then UPDATE the records in the dataset—assigning the new variable a value of 'Y' (yes) or 'N' (no) based on the plan code associated with each record.

```
PROC SQL;
  ALTER TABLE analytic.membership ADD dental_coverage CHAR(1);
  UPDATE analytic.membership
      SET dental_coverage = 'Y'
   WHERE plan IN ('SMSD', 'FMFD', 'FMSD');
  UPDATE analytic.membership
      SET dental_coverage = 'N'
   WHERE dental_coverage = '' AND plan IS NOT NULL;
QUIT;
```

There are a couple of interesting things to note about this PROC SQL statement. First, it is actually a set of three SQL statements—one ALTER TABLE statement and two UPDATE statements—within the same call to PROC SQL. Secondly, because the UPDATE statements are separate commands within the PROC SQL statement, we can take advantage of this fact and assume (in the second UPDATE statement) that any records that do not yet have a value assigned to "dental_coverage" after the preceding UPDATE statement are, by definition, correctly specified as member records associated with plans that do not have dental coverage. However, in so doing, we also must be careful not to implicitly assign "dental_coverage" the value of 'N' simply because it did not have a value associated with a plan code that we know to indicate dental coverage. The record could also have a null value for "plan", so we must add the IS NOT NULL criterion to our second update statement.

BEYOND THE BASICS

With a solid understanding of SELECT, FROM, WHERE, and ORDER BY (as well as the less prevalent, but also important INSERT, UPDATE, and DELETE), analysts are empowered to begin performing fairly sophisticated data manipulations. Beyond these basic functions, however, are a host of additional techniques that allow analysts to leverage PROC SQL against even more sophisticated data management challenges.

SAS FUNCTIONS & FORMATS IN PROC SQL.

As discussed earlier, PROC SQL accepts all ANSI-standard SQL syntax; to the extent that a query conforms to this standard, in can be executed in SAS just as readily as in any other tool compliant with the ANSI standard. Beyond that syntax, however, PROC SQL also allows analysts to utilize SAS functions in the SELECT statement, WHERE clause, and ORDER BY clause. As an example of where this might be useful, consider the query example provided in the preceding section on the ORDER BY clause.

```
PROC SQL;
 CREATE TABLE work.July_XYZ_Co_Claims AS
    SELECT bill_date, account_no, claim_id, claim_line_item, dx_code AS diagnosis,
           line_item_charge AS claim_line_item_amount
      FROM work.July_2010_Claims
     WHERE client_id = 'XYZ Co.'
  ORDER BY bill_date DESC, account_no, claim_id, claim_line_item;
QUIT;
```

If we were working with date variables stored as strings (e.g., '01/02/2001'), sorting by "bill_date" could result in unexpected results whereby records might be sorted as follows:

```
01/04/2004
01/03/2007
01/02/2003
```

This makes sense because "01/04" comes before "01/03" when sorting these strings in descending order even though the sort does not result in the correct descending sort order of the associated dates. This query sorts the string contents of "bill_date", not the corresponding date values. However, if we use the SAS INPUT function, we can sort by the dates corresponding to those string representations.

```
PROC SQL;
 CREATE TABLE work.July_XYZ_Co_Claims AS
    SELECT bill_date, account_no, claim_id, claim_line_item, dx_code AS diagnosis,
           line_item_charge AS claim_line_item_amount
      FROM work.July_2010_Claims
     WHERE client_id = 'XYZ Co.'
  ORDER BY INPUT(bill_date,MMDDYY10.) DESC, account_no, claim_id, claim_line_item;
QUIT;
```

Having applied the SAS INPUT function to the "bill_date", the records are now sorted in their correct descending order based on the date associated with this string variable—even though the "bill_date" variable returned in the target dataset is still stored as a string.

```
01/03/2007
01/04/2004
01/02/2003
```

In order to return the "bill_date" as a numeric date variable in the target dataset, we could also put the INPUT statement in the SELECT statement.

```
PROC SQL;
 CREATE TABLE work.July_XYZ_Co_Claims AS
    SELECT INPUT(bill_date,MMDDYY10.) AS bill_date, account_no, claim_id,
           claim_line_item, dx_code AS diagnosis,
           line_item_charge AS claim_line_item_amount
      FROM work.July_2010_Claims
     WHERE client_id = 'XYZ Co.'
  ORDER BY INPUT(bill_date,MMDDYY10.) DESC, account_no, claim_id, claim_line_item;
QUIT;
```

Alternatively, suppose the source system for the creation of our July 2010 charges dataset stored "bill_date" as a datetime variable instead of a string or date variable. We could specify the inclusion criteria using datetime literals as in the following example, but this is a bit cumbersome.

```
PROC SQL;
 CREATE TABLE work.jul2010_charges AS
 SELECT *
   FROM work.claim_detail
  WHERE billing_date BETWEEN '01JUL2010:00:00:00'DT AND '31JUL2010:23:59:59'DT;
QUIT;
```

In order to simplify the specification of the inclusion dates (and make our code cleaner), we could use the DATEPART function to return a value for "billing_date" that converts the datetime value to a date value that can be validly compared to date literals.

```
PROC SQL;
 CREATE TABLE work.jul2010_charges AS
 SELECT *
   FROM work.claim_detail
  WHERE DATEPART(billing_date) BETWEEN '01JUL2010'D AND '31JUL2010'D;
QUIT;
```

These examples demonstrate a very important point in understanding how to use PROC SQL; despite the fact that any ANSI-standard SQL statement can be executed within this PROC, so to can SAS functions, formats, and labels be used within this syntax. That is, the SELECT statement is part of the standard SQL language, but the SAS implementation of SQL also allows the use of SAS functionality that is not part of the SQL Standard. For SAS programmers, it is the best of both worlds. The following PROC SQL statement demonstrates some of these SAS features.

```
PROC SQL;
 CREATE TABLE work.july_2010_claims AS
   SELECT acct_number,
          UPCASE(STRIP(member_lname))||', '||UPCASE(STRIP(member_fname)) AS member,
          claim_id, claim_line_item,PUT(dx_code, icd9_label.) AS diagnosis
          claim_amount LABEL = claim_line_item_amount
     FROM work.claim_detail
    WHERE INPUT(bill_date,mmddyy10.) BETWEEN '01JUL2010'd AND '31JUL2010'd
QUIT;
```

In this example, we use the UPCASE and STRIP functions to concatenate the insurance plan members' first and last names such that trailing blank spaces in the name variables will be removed and the resulting name will be in all caps. As a result of performing this concatenation, PROC SQL does not know what to call this variable so we assign the result a column alias of "member". Without this alias, the resulting column would be assigned the name "_TEMA001" and subsequent columns without explicitly defined column names would be named "_TEMA002", "_TEMA003", etc. Also in this example, we see that the PUT function is used to convert the diagnosis code (dx_code) to the label associated with it in the user-defined format "icd9_label" (see Schacherer & Westra, 2010; Cody, 2010 for information on creating user-defined formats in SAS). Finally, as is too often the case in healthcare claims data, the "bill_date" in this example is stored as a string variable instead of a numeric date field. Therefore, in order to make the comparison we wish to make with the date literals in our BETWEEN comparison, we use INPUT to assure that the numeric dates in our criteria are compared with a numeric date equivalent. Also of (minor) interest here is the fact that the variable "bill_date" is not in the resulting dataset. We can use it in the WHERE clause without having to write it to the resulting dataset. In an actual dataset of this type, though, we would likely include "bill_date" so that we could roll up claims by day of week, week of the billing period, etc.

Finally, just as one can use SAS formats or user-defined formats with the PUT and INPUT functions, one can also apply formats to variables in a dataset created by PROC SQL, while retaining the data type of the formatted variable. For example, in the following PROC SQL statement, the "admit_date" is stored as a SAS date (i.e., the integer representing the number of days since January 1, 1960), but we would prefer the admit date to be represented in the familiar MM/DD/YYYY format in our printed dataset and when browsing the dataset in the SAS VIEWTABLE window. To accomplish this, we specify that the format MMDDYY10. be applied to "admit_date" within the SELECT statement.

```
PROC SQL;
 CREATE TABLE work.july_2010_admissions AS
    SELECT acct_number, claim_id, admit_date FORMAT MMDDYY10., billed_charges
      FROM work.inpatient_admissions
     WHERE admit_date BETWEEN '01JUL2010'd AND '31JUL2010'd
QUIT;
```

In this way, the values of "admit_date" will be represented in the familiar date format, but will continue to be stored as integers representing the number of days since January 1, 1960.

CASE STATEMENTS

Despite the fact that many SAS functions and dataset operators can be used within the PROC SQL statement, however, experienced DATA STEP programmers new to PROC SQL are often perplexed by the conspicuous lack of IF/THEN-ELSE logic in PROC SQL queries. This seeming oversight reflects the nature of the SQL language in comparison to DATA STEP programming—with the latter being a procedural language performing operations sequentially on each record in a source dataset, and the former focusing on the definition of the resulting dataset.

That said, however, the SQL language does have a way to execute data manipulations based on conditional logic. In the following example, the CASE statement is used to assign values to a new variable (charge_level) based on the value of the "billed_charges".

```
PROC SQL;
 CREATE TABLE work.charge_levels AS
 SELECT billing_id,billed_charges,
        CASE
            WHEN 0 <= billed_charges < 1000 THEN 'LOW'
            WHEN 1000 <= billed_charges < 10000 THEN 'MEDIUM'
            WHEN 10000 <= billed_charges THEN 'HIGH'
        END AS charge_level
   FROM work.claim_detail;
QUIT;
```

When the preceding code is executed, the user will receive the following feedback reminding them that all possible conditions in their dataset have not been addressed with respect to the assignment of the "charge_level" variable:

```
NOTE: A CASE expression has no ELSE clause. Cases not accounted for by the WHEN clauses will
      result in a missing value for the CASE expression.
```

If we had previously confirmed that there are no negative "billed_charges" values in our dataset, this message can be safely ignored. Normally, however, if you expect all records in a dataset to evaluate to a value for the new variable, you should include an ELSE clause to account for unexpected values in your dataset.

```
PROC SQL;
 CREATE TABLE work.charge_levels AS
 SELECT billing_id,billed_charges,
        CASE
            WHEN 0 <= billed_charges < 1000 THEN 'LOW'
            WHEN 1000 <= billed_charges < 10000 THEN 'MEDIUM'
            WHEN 10000 <= billed_charges THEN 'HIGH'
            ELSE '**PLEASE REVIEW**'
        END AS charge_level
   FROM work.claim_detail;
QUIT;
```

In this case, the only values that are not assigned a value by a WHEN clause are those that have a "billed_charges" value less than 0; these records will be flagged as needing further review. As a corollary to the need for an ELSE clause in situations where you believe every record should be assigned a value, it should also be noted that the value of the variable being created is assigned by the first WHEN statement that evaluates to TRUE (Lund, 2005).

Therefore, if you allow two conditions to overlap in the same CASE statement (a situation you should avoid), the assignment of the computed value will be determined by which WHEN statement is encountered first.

In this statement, records with "billed_charges" values equal to 10,000 will be assigned the value of "MEDIUM",

```
PROC SQL;
 CREATE TABLE work.charge_levels AS
 SELECT billing_id,billed_charges,
        CASE
            WHEN 0 <= billed_charges < 1000 THEN 'LOW'
            WHEN 1000 <= billed_charges <= 10000 THEN 'MEDIUM'
            WHEN 10000 <= billed_charges THEN 'HIGH'
            ELSE '**PLEASE REVIEW**'
        END AS charge_level
   FROM work.claim_detail;
QUIT;
```

whereas using the following PROC SQL statement will produce a value of "HIGH" for those same records:

```
PROC SQL;
 CREATE TABLE work.charge_levels AS
 SELECT billing_id,billed_charges,
        CASE
            WHEN 10000 <= billed_charges THEN 'HIGH'
            WHEN 1000 <= billed_charges <= 10000 THEN 'MEDIUM'
            WHEN 0 <= billed_charges < 1000 THEN 'LOW'
            ELSE '**NEEDS REVIEW**'
        END AS charge_level
   FROM work.claim_detail;
QUIT;
```

Finally, without an AS operator following the END statement, SAS will (just as it did in producing the output of a function) assign its own name to the resulting variable (e.g., _TEMA001). In the preceding three examples, the result of the CASE statement will be the variable "charge_level".

AGGREGATE FUNCTIONS & THE GROUP BY CLAUSE

Creating subsets of data, creating new variables, and transforming existing variables are all very important features of PROC SQL, but people looking for answers to questions about organizational performance (be it hospital quality scores, geographic market penetration, or reimbursement rates and case mix) are rarely excited about raw data. They want to summarize the data and see what is happening at a higher level (by customer group, industry, payor type, etc.). This is where aggregate (or, summary) functions come into play. Summary functions in SQL are functions that aggregate data across a defined set of records to produce summary results describing that set of records. Some of the most frequently used functions are listed below:

Function	Description
MIN	Returns the minimum value of the summarized variable within the specified record set
MAX	Returns the maximum value of the summarized variable within the specified record set
AVG	Calculates the average of the summarized variable within the specified record set
STD	Calculates the standard deviation of the summarized variable within the specified record set
SUM	Sums the values of the summarized variable within the specified record set
COUNT	Counts the number of records in the specified record set that contain non-null values for the summarized variable

In the following example, suppose that every month after the claims data are cleaned, you have a few simple reports that you want to distribute to key individuals. Perhaps you need to prepare a quick high-level overview of claims paid for the previous 12-month period. You could perform the following query to produce the high-level financial activity summary requested:

```
PROC SQL;
 CREATE TABLE work.client_rolling_year_summary AS
  SELECT client_id,
         PUT(YEAR(bill_date),4.)||'-'||PUT(MONTH(bill_date),Z2.) AS fiscal_period ,
         SUM(billed_charges)     AS billed_charges,
         SUM(duplicate_charges)  AS duplicates,
         SUM(ineligible_charges) AS ineligible,
         SUM(billed_charges) - SUM (duplicate_charges) - SUM (ineligible_charges)
            AS net_charges
    FROM work.claim_detail
   WHERE bill_date BETWEEN '08/01/2009'd and '07/31/2010'd
GROUP BY client_id, fiscal_period
ORDER BY client_id, fiscal_period DESC;
QUIT;
```

The records on which the aggregation is performed are defined by the GROUP BY clause. In this case, the SUMs calculated are those associated with each unique combination of "client_id" and "fiscal_period".

client_id	fiscal_period	billed_charges	duplicates	ineligible	net_charges
XYZ Co.	2010-07	$588,051	$5,469	$2,663	$579,919
XYZ Co.	2010-06	$589,215	$5,480	$2,669	$581,067
XYZ Co.	2010-05	$590,379	$5,491	$2,674	$582,215
XYZ Co.	2010-04	$591,543	$5,501	$2,679	$583,363
XYZ Co.	2010-03	$547,633	$5,093	$2,480	$540,060
XYZ Co.	2010-02	$538,431	$5,007	$2,439	$530,985
XYZ Co.	2010-01	$558,210	$5,191	$2,528	$550,490
XYZ Co.	2009-12	$558,615	$5,195	$2,530	$550,890
XYZ Co.	2009-11	$597,363	$5,555	$2,706	$589,102
XYZ Co.	2009-10	$598,173	$5,563	$2,709	$589,901
XYZ Co.	2009-09	$600,400	$5,584	$2,719	$592,097
XYZ Co.	2009-08	$600,501	$5,585	$2,720	$592,197
ABC Co.	2010-07	$337,874	$3,142	$1,530	$333,202
ABC Co.	2010-06	$338,540	$3,148	$1,533	$333,859
ABC Co.	2010-05	$313,410	$2,915	$1,419	$309,076

In the following example, we roll up the records to summarize the data at the client level of specificity for the same time period:

```
PROC SQL;
 CREATE TABLE work.client_rolling_year_summary AS
  SELECT client_id,
         PUT(YEAR(bill_date),4.)||'-'||PUT(MONTH(bill_date),Z2.) AS fiscal_period ,
         SUM(billed_charges)     AS billed_charges,
         SUM(duplicate_charges)  AS duplicates,
         SUM(ineligible_charges) AS ineligible,
         SUM(billed_charges) - SUM (duplicate_charges) - SUM (ineligible_charges)
            AS net_charges
    FROM work.claim_detail
   WHERE bill_date BETWEEN '08/01/2009'd and '07/31/2010'd
GROUP BY client_id;
QUIT;
```

client_id	billed_charges	duplicates	ineligible	net_charges
XYZ Co.	$6,958,516	$64,714	$31,516	$6,862,286
ABC Co.	$3,695,721	$34,370	$16,738	$3,644,612

To further demonstrate the relationship between the data aggregation driven by the GROUP BY clause and the variables in the select clause, consider the following example, in which data are aggregated at the client_id level of specificity, but the fiscal period associated with each record in the source dataset is also included in the SELECT statement.

```
PROC SQL;
 CREATE TABLE work.all_client_financial_summary AS
  SELECT client_id, claimid,
         PUT(YEAR(bill_date),4.)||'-'||PUT(MONTH(bill_date),Z2.) AS fiscal_period ,
         SUM(billed_charges)     AS billed_charges,
         SUM(duplicate_charges)  AS duplicates,
         SUM(ineligible_charges) AS ineligible,
         SUM(billed_charges) - SUM (duplicate_charges) - SUM (ineligible_charges)
            AS net_charges
    FROM work.claim_detail
   WHERE bill_date BETWEEN '08/01/2009'd and '07/31/2010'd
GROUP BY client_id;
QUIT;
```

In this case, the summary data are still aggregated by "client_id", but because another variable that is not part of the GROUP BY specification is included in the SELECT statement, PROC SQL returns every record from the source dataset, but attaches the summary data computed at the level of the "client_id" to each record in the target dataset that is associated with the summarized "client_id"[2].

client_id	claimid	fiscal_period	billed_charges	duplicates	ineligible	net_charges
XYZ Co.	6578891	2010-07	$6,958,516	$64,714	$31,516	$6,862,286
XYZ Co.	6578892	2010-07	$6,958,516	$64,714	$31,516	$6,862,286
XYZ Co.	6578893	2010-07	$6,958,516	$64,714	$31,516	$6,862,286
XYZ Co.	6578894	2010-07	$6,958,516	$64,714	$31,516	$6,862,286
XYZ Co.	6578895	2010-07	$6,958,516	$64,714	$31,516	$6,862,286
XYZ Co.	6578896	2010-07	$6,958,516	$64,714	$31,516	$6,862,286
XYZ Co.	6578897	2010-07	$6,958,516	$64,714	$31,516	$6,862,286

In this case, SAS writes the following NOTE to the LOG:

```
NOTE: The query requires remerging summary statistics back with the original data.
```

This remerging of summary statistics is not necessarily an error, because it is possible that what you want to do is determine (for example) the proportion of the total "billed_charges" for each client that is represented by each line item in the source dataset.

```
PROC SQL;
 CREATE TABLE work.all_client_financial_summary AS
    SELECT client_id, claimid, billed_charges,
           (billed_charges/SUM(billed_charges))*100 AS pct_billed_charges,
           SUM(billed_charges) as total_12mo_charges
      FROM work.claim_detail
     WHERE bill_date BETWEEN '08/01/2009'd and '07/31/2010'd
  GROUP BY client_id;
QUIT;
```

client_id	claimid	billed_charges	pct_billed_charges	total_12mo_charges
XYZ Co.	6578891	$588.05	0.00845%	$6,958,516
XYZ Co.	6578892	$589.22	0.00847%	$6,958,516
XYZ Co.	6578893	$60,153.77	0.86446%	$6,958,516
XYZ Co.	6578894	$591.54	0.00850%	$6,958,516
XYZ Co.	6578895	$112,512.00	1.61690%	$6,958,516

[2] Please note here, as well, that we have included the "claimid" in the select statement as well to clarify the specificity of the records returned by the query.

However, if you are anticipating that the resulting dataset will contain a summary dataset at the level of specificity indicated in the GROUP BY clause, the "remerging" note should serve as a warning that you may have incorrectly specified either your SELECT statement or your GROUP BY clause.

HAVING

In addition to aggregating data using aggregate functions and GROUP BY clauses, we sometimes also want to conditionally limit the summary records returned to our target dataset. For example, suppose that an account manager wants a list of the health plan members who had claims over a certain threshold—say, $10,000 for "XYZ Co." In that case you would first want to SUM the charges—grouping by the member number (memberno)—and specify "XYZ Co." in the WHERE clause.

```
PROC SQL;
 CREATE TABLE work.xyz_co_high_cost_claimants AS
  SELECT memberno,
         PUT(YEAR(bill_date),4.)||'-'||PUT(MONTH(bill_date),Z2.) AS fiscal_period ,
         SUM(billed_charges) AS total_billed_charges,
    FROM work.claim_detail
   WHERE client_id = 'XYZ Co.'
GROUP BY member_no, fiscal_period
QUIT;
```

The preceding query gets us close to our goal by summing charges for each health plan member covered by XYZ Co.'s health plan, but we want a list of only those members whose claims exceed the $10,000 threshold. One common mistake when trying to perform a query like this is trying to put the selection criterion identifying the amount of "total_billed_charges" in the WHERE clause.

```
PROC SQL;
 CREATE TABLE work.xyz_co_high_cost_claimants AS
  SELECT memberno,
         PUT(YEAR(bill_date),4.)||'-'||PUT(MONTH(bill_date),Z2.) AS fiscal_period ,
         SUM(billed_charges) AS total_billed_charges,
    FROM work.claim_detail
   WHERE client_id = 'XYZ Co.' and
         total_billed_charges > 10000
GROUP BY member_no, fiscal_period;
QUIT;
```

Even though we have defined "total_billed_charges" in the SELECT statement, it does not exist on the individual rows of data and cannot, therefore, be used to evaluate records for inclusion in the dataset. But in executing the previous query we are given a hint that will help write the correct SQL statement.

```
ERROR: Summary functions are restricted to the SELECT and HAVING clauses only.
```

The HAVING clause is used to limit summary rows returned from a PROC SQL query to those summary records in which the value of the summary function meets a specified criterion. Adding a HAVING clause to a query that contains aggregated data allows you to evaluate the summarized data and only write to the target dataset those summary rows that "have" the summary criteria specified by the HAVING clause. In that sense, it is similar to a WHERE clause for aggregate values. In the following example of the "high_cost_claimants" query, we specify that the only records to be written to the dataset are those that have a summary value of "total_billed_charges" that is greater than or equal to $10,000. Also note that in the HAVING clause, the name of the summarized variable "total_billed_charges" is understood perfectly well. This is because by the time execution reaches the HAVING clause, summary variables have already been computed across the variables specified in the GROUP BY clause.

```
PROC SQL;
 CREATE TABLE work.xyz_co_high_cost_claimants AS
  SELECT memberno,
         PUT(YEAR(bill_date),4.)||'-'||PUT(MONTH(bill_date),Z2.) AS fiscal_period ,
         SUM(billed_charges) AS total_billed_charges,
    FROM work.claim_detail
   WHERE client_id = 'XYZ Co.'
GROUP BY member_no, fiscal_period
  HAVING total_billed_charges > 10000;
QUIT;
```

One final tip on using the HAVING clause is that it is a very useful way to check for duplicate records. Using the COUNT summary function, one can include in the query all of the columns that should uniquely identify a record in the dataset and determine if there are any combinations of those variables that result in a record count greater than one. If the query returns any records, your dataset contains duplicate records within that combination of variables. In the following example, we compute the number of records associated with each combination of "claimid", "line_item", and "record_type".

```
PROC SQL;
 CREATE TABLE work.unanticipated_duplicates AS
    SELECT claimid,line_item,record_type,COUNT(*) AS numrecs
      FROM work.claim_detail
  GROUP BY claimid,line_item,record_type
    HAVING numrecs > 1;
QUIT;
```

If the combination of these three variables constitutes a valid unique key in the dataset, you receive the following LOG message:

```
NOTE: No rows were selected.
```

If there are duplicates in your dataset, the dataset "unanticipated_duplicates" will contain one record for each combination of "claimid", "line_item", and "record_type" that occurs more than once in your dataset—along with the value of "numrecs", which indicates the number of duplicate records that share those specific values for the other three variables.

SQL JOINS

By far the greatest boon to SAS programming that has resulted from the inclusion of PROC SQL has been the highly intuitive ability to join datasets. As mentioned in the introduction, early examples of PROC SQL that were discussed among SAS practitioners were the comparison of PROC SQL to other BASE SAS PROCs and DATA STEP programming methods. The majority of these discussions focused intently on the comparison of SQL JOINs to DATA STEP MERGE statements, and die-hard DATA STEP advocates would go to great lengths to dispel the notion that PROC SQL was capable of performing the necessary joins produced by their more complex MERGE methodologies. We will not rehash those arguments here, but I think it suffices to say that (as Whitlock, 2001, p. 6 states) "My conclusion is that a good SAS programmer can no longer ignore PROC SQL and remain good".

The remainder of this section will focus on the EQUI-JOIN, LEFT (and RIGHT) OUTER JOIN, the FULL OUTER JOIN, and the UNION JOIN.

EQUI-JOIN

Joining two datasets using PROC SQL follows many of the same syntax rules described previously in our discussion of SELECT statements performed against a single dataset. We SELECT the variables we want to extract FROM the dataset(s) in which they are stored. Joining data from more than one dataset, however, requires that some additional information be specified. In the following example, we will perform an equi-join between the members dataset and an aggregated dataset (claim_summary) that represents the sum of the billed_charges for the last 12 months for each "member_id":

members					
member_id	fname	lname	dob	race	gender
0123345	Smith	Mary	5/13/1966	H	F
0123346	Jones	Steve	4/28/1971	B	M
0123347	Apple	Larry	5/23/1988	W	M
0123348	Steine	George	12/11/1972	W	M
0123349	Lane	Curtis	2/12/1973	W	M
0123350	Mills	Jordan	3/20/1945	W	M
0123351	Lee	Wayne	8/28/1959	A	M
0123352	Davis	Lisa	11/29/1982	B	F

claim_summary		
member_ id	billed_charges	net_charges
0123345	$188.05	$182.58
0123346	$5,687.98	$5,321.07
0123348	$60,677.52	$60,332.12
0123351	$538.43	$538.43
0123352	$558.21	$558.21

```
PROC SQL;
 CREATE TABLE work.member_claim_summary AS
       SELECT a.member_id, a.lname, a.fname , a.dob, a.race,
              a.gender, b.billed_charges, b.net_charges
         FROM work.members a,
              work.claim_summary b
        WHERE a.member_id = b.member_id
QUIT;
```

member_claim_summary							
member_id	fname	lname	dob	race	gender	billed_charges	net_charges
0123345	Smith	Mary	5/13/1966	H	F	$188.05	$182.58
0123346	Jones	Steve	4/28/1971	B	M	$5,687.98	$5,321.07
0123348	Steine	George	12/11/1972	W	M	$60,677.52	$60,332.12
0123351	Lee	Wayne	8/28/1959	A	M	$538.43	$538.43
0123352	Davis	Lisa	11/29/1982	B	F	$558.21	$558.21

First, in the above example, the SELECT statement needs to unequivocally identify the source of the variables being extracted. In this example we do this by assigning table aliases (a & b) to the datasets work.members and work.claim_summary, respectively. These aliases are assigned in the FROM clause and are used in the select statement to specify the dataset from which each variable is to be extracted. We could perform this select without using aliases—by specifying the columns as, for example, members.member_id, members.dob, etc.—but aliases allow us to keep our code more compact. Also, if each variable in the SELECT statement were unique to only one of the datasets, SAS could determine on its own which dataset the variable comes from, but explicitly indicating the source dataset with which a variable is associated is a good habit to develop.

Second, the WHERE clause (just as in the previous examples) specifies which records from the possible source data are to be returned as the target dataset. However, unlike selecting rows from a single dataset, the default result of selecting data from more than one dataset is a dataset that represents all possible combinations of records from all datasets listed in the FROM clause. This default result is known as the Cartesian product. For example, if there are 100,000 members and 75,000 claim summary records, the Cartesian product would contain 7,500,000,000 (7.5 billion) rows of data—one row for each possible combination of a row in the members dataset with a row in the claim summary dataset. If there is no WHERE clause in the query, the result returned from the query will include all 7.5 billion rows. What the WHERE clause in this query does is specify that from the Cartesian product, we wish to return only those records where the "member_id" in the "members" record is the same as the "member_id" in the "claim_summary". As a result, only those "members" that have at least one record in "claim_summary" will be represented in the target dataset. In other words, if a member has not generated any claims, there will be no rows in the new dataset that contain that member's "member_id". Likewise, if there is a "claim_summary" record that contains a "member_id" that is not present in the "members" dataset[3], those claim details will not be returned to the target dataset. This type of join is known as an "equi-join" because it returns only those rows that are "equal" (or, equivalent) in terms of the variables on which the datasets are joined.

This example demonstrates that the type of query result you are trying to achieve and the structure and relationship of the source data are critical to choosing the correct type of join. In this example, we wanted to add member information to the claim details. We know that in our billing system, claim detail records cannot be written without a valid member id, and therefore we know that an equi-join will result in the desired dataset—a dataset with all claim details having the desired member information. What this query will NOT do, however, is provide us with a dataset

[3] In reality this should not happen in this particular dataset because the operational system requires a valid member_id in order to enter a claim record.

that we can use to assess the total claim dollars (or utilization of healthcare services) for each member in our "members" dataset—because if a member does not have any claims, they will not be represented in the dataset produced by the equi-join. Instead, to answer these types of questions we would perform a LEFT JOIN.

LEFT (RIGHT) JOIN

A LEFT (or RIGHT) JOIN is performed when you want to return all of the rows from one dataset and only those rows from the other dataset that have a matching value on the variable(s) on which the rows are being matched. In the following example, we create a dataset that contains all of the records in "members" and all rows in "claim_summary" that are associated with those member records.

```
PROC SQL;
CREATE TABLE work.full_member_claim_summary AS
       SELECT a.member_id, a.lname, a.fname , a.dob, a.race,
              a.gender, b.billed_charges, b.net_charges
         FROM work.members a LEFT JOIN work.claim_summary b
           ON a.member_id = b.member_id
QUIT;
```

member_claim_summary							
member_id	fname	lname	dob	race	gender	billed_charges	net_charges
0123345	Smith	Mary	5/13/1966	H	F	$188.05	$182.58
0123346	Jones	Steve	4/28/1971	B	M	$5,687.98	$5,321.07
0123347	Apple	Larry	5/23/1988	W	M	.	.
0123348	Steine	George	12/11/1972	W	M	$60,677.52	$60,332.12
0123349	Lane	Curtis	2/12/1973	W	M	.	.
0123350	Mills	Jordan	3/20/1945	W	M	.	.
0123351	Lee	Wayne	8/28/1959	A	M	$538.43	$538.43
0123352	Davis	Lisa	11/29/1982	B	F	$558.21	$558.21

For those members with claims in "claim_summary", the values for "billed_charges", and "net_charges" will be returned to the new dataset. For those members with no claims, there will be a row that represents the data extracted from the "members" table, and null values will be returned for "billed_charges", and "net_charges" You will notice that the ON clause is used here to specify those rows from the Cartesian product that will be returned to "member_claim_summary". The specification of the join (as a LEFT or RIGHT JOIN) comes from the order of the dataset names in the FROM clause and the keyword LEFT or RIGHT. We would get the same result as the previous example if we specified the FROM clause as the follows:

```
FROM work.claim_summary b RIGHT JOIN work.members a
```

It should also be pointed out that despite the ON clause being used to specify the records to be returned from the two datasets (instead of a WHERE clause), one can also include a WHERE clause to further filter the results of a LEFT JOIN. For example, if we wanted to create a dataset of the claims associated with members from a specific client company, we might perform a query similar to the following:

```
PROC SQL;
CREATE TABLE work.XYZ_Co_member_claim_summary_XYZ AS
       SELECT a.client_id, a.member_id, a.lname, a.fname , a.dob, a.race,
              a.gender, b.billed_charges, b.net_charges
         FROM work.members a LEFT JOIN work.claim_summary b
           ON a.member_id = b.member_id
        WHERE a.client_id = 'XYZ Co.';
QUIT;
```

As in the previous example, all members, regardless of whether they have had claims will be represented in the result set, but in this example, the target dataset will contain records associated only with those members who are employed by client "XYZ Co."

member_claim_summary_XYZ								
client_id	member_id	fname	lname	dob	race	gender	billed_charges	net_charges
XYZ Co.	0123345	Smith	Mary	5/13/1966	H	F	$188.05	$182.58
XYZ Co.	0123346	Jones	Steve	4/28/1971	B	M	$5,687.98	$5,321.07
XYZ Co.	0123347	Apple	Larry	5/23/1988	W	M	.	.
XYZ Co.	0123350	Mills	Jordan	3/20/1945	W	M	.	.
XYZ Co.	0123351	Lee	Wayne	8/28/1959	A	M	$538.43	$538.43

Like the queries performed against a single datataset, SQL JOINs can also be used to summarize the JOINed data. In the following example, in addition to LEFT JOINing the "members" data to "claim_detail", we sum the "billed_charges" and "netcharges" across gender:

```
PROC SQL;
 CREATE TABLE work.gender_claim_summary_XYZ AS
    SELECT a.gender, SUM(b.billed_charges) AS total_billed_charges,
           SUM(b.net_charges) AS total_net_charges
      FROM work.members a LEFT JOIN work.claim_detail b
        ON a.member_id = b.member_id
     WHERE a.client_id = 'XYZ Co.'
  GROUP BY a.gender;
QUIT;
```

gender_claim_summary_XYZ		
gender	toal_billed_charges	total_net_charges
F	$188.05	$182.58
M	$6,226.41	$5,859.50

The key concept of a LEFT or RIGHT JOIN, however, is that all records from one dataset and only those from the other dataset that represent a match to the record in the first dataset will be represented in the resulting target dataset. The next logical extension to that idea is to represent all records in two datasets (matching and nonmatching) in a single dataset. This is the goal of the FULL (OUTER) JOIN.

FULL JOIN

The FULL JOIN is used when you want to create a dataset that contains all rows from either dataset—both those that match and those that do not. To demonstrate where this type of join might be appropriate, consider two datasets that contain membership data. One dataset might be missing records for some subset of members and another could be missing data from another subset of members.

membership1			
member_num	fname	lname	dob
0123347	Apple	Larry	5/23/1988
0123348	Steine		
0123349	Lane	Curtis	2/12/1973
0123350	Mills	Jordan	3/20/1945
0123351	Lee	Wayne	8/28/1959
0123352	Davis	Lisa	11/29/1982

membership2			
member_id	fname	lname	dob
0123345	Smith	Mary	5/13/1966
0123346	Jones	Steve	4/28/1971
0123347	Apple	Larry	
0123348	Steine		
0123349	Lane	Curtis	
0123350	Mills	Jordan	

Our goal in executing the following FULL JOIN is to create a membership dataset that identifies all members.

```
PROC SQL;
 CREATE TABLE work.all_members AS
    SELECT a.member_num, b.member_id
      FROM work.membership1 a FULL JOIN work.membership2 b
        ON a.member_id = b.member_num
QUIT;
```

all_members	
member_num	member_id
	0123345
	0123346
0123347	0123347
0123348	0123348
0123349	0123349
0123350	0123350
0123351	
0123352	

As you can see from the dataset above, we were successful in creating a dataset that has all of the members from the two datasets, but they are still in two different columns—with the pattern of missing data indicating the source of these member ids. Additionally, what we ultimately want to do is retrieve a complete set of data about these members from the two datasets (notice, for example, that some data elements are missing from "membership1" and "membership2") In order to create a full accounting of all possible membership data from these two datasets we will utilize the COALESCE function. The COALESCE function takes variables and literals as its input and returns the first non-missing value (SAS Institute, 2004c).

```
PROC SQL;
CREATE TABLE work.all_members AS
    SELECT COALESCE (a.member_id,b.member_num,'*******') AS member_id,
           COALESCE (a.fname,b.fname, 'Not Available') as fname,
           COALESCE (a.lname,b.lname, 'Not Available') as lname,
           COALESCE (a.dob,b.dob) as dob
      FROM work.membership1 a FULL JOIN work.membership2 b
        ON a.member_id = b.member_num
QUIT;
```

all_members			
member_id	fname	lname	dob
0123345	Smith	Mary	5/13/1966
0123346	Jones	Steve	4/28/1971
0123347	Apple	Larry	5/23/1988
0123348	Steine	Not Available	.
0123349	Lane	Curtis	2/12/1973
0123350	Mills	Jordan	3/20/1945
0123351	Lee	Wayne	8/28/1959
0123352	Davis	Lisa	11/29/1982

UNION JOIN (EXCEPT & INTERSECT)

Whereas the JOINs discussed so far (EQUI-JOIN, LEFT, RIGHT and FULL) create datasets by combining columns from rows that are matched on some key value(s), the last type of join creates datasets by combining rows from two or more datasets and aligning the columns. Suppose that you have two datasets that contain healthcare claims for two companies in the same industry--XYZ Co. and ABC Co. If we want to take a broader look at the claims in that industry, we may want to pool data from these two datasets.

claim_summary_xyz		
member_id	billed_charges	net_charges
0123345	$188.05	$182.58
0123346	$5,687.98	$5,321.07
0123347	$60,677.52	$60,332.12

claim_summary_abc		
member_id	billed_charges	net_charges
0024579	$188.05	$182.58
0024580	$672.33	$650.33
0224581	$6,537.52	$6,340.12

In order to create this larger analytic dataset, we use a UNION JOIN.

```
PROC SQL;
 CREATE TABLE work.manufacturing_claims as
   SELECT 'XYZ Co.' AS client, * FROM claim_summary_XYZ
   UNION
   SELECT 'ABC Co.' AS client, * FROM claim_summary_ABC;
QUIT;
```

manufacturing_claims			
client	member_id	billed_charges	net_charges
XYZ Co.	0123345	$188.05	$182.58
XYZ Co.	0123346	$5,687.98	$5,321.07
XYZ Co.	0123347	$60,677.52	$60,332.12
ABC Co.	0024579	$188.05	$182.58
ABC Co.	0024580	$672.33	$650.33
ABC Co.	0224581	$6,537.52	$6,340.12

The result of the UNION JOIN is the set of unique rows contributed from both datasets; by default identical rows in the result set are eliminated when using the UNION JOIN. This is sometimes a source of confusion when using the UNION JOIN for the first time. Suppose that "claim_summary_xyz" contains 50,000 records and "claim_summary_abc" contains 40,000 records; if the resulting "manufacturing_claims" dataset contains 89,999 records we might assume that the UNION JOIN has somehow failed because we are missing 1 record. It is most likely, in this case, that one of the datasets contained a duplicate record and it was eliminated by the UNION JOIN. However, when a UNION join results in a dataset with fewer records than the total number of records from the individual datasets, it is often the case that the query did not involve all of the columns necessary to uniquely identify the records across both datasets. Using the same summary datasets as the previous example, the following query fails to return both records with "billed_charges" equal to $188.05 because the columns that would make these two records unique are not in the SELECT statements. As a result, one record is dropped from the result-set.

```
PROC SQL;
 CREATE TABLE work.manufacturing_claims as
   SELECT billed_charges, net_charges FROM claim_summary_XYZ
   UNION
   SELECT billed_charges, net_charges FROM claim_summary_ABC;
QUIT;
```

manufacturing_claims	
billed_charges	net_charges
$188.05	$182.58
$5,687.98	$5,321.07
$60,677.52	$60,332.12
$672.33	$650.33
$6,537.52	$6,340.12

If one wishes to retain all records across both datasets, the UNION operator can be replaced with UNION ALL which retains all records—including duplicates:

```
PROC SQL;
 CREATE TABLE work.manufacturing_claims as
   SELECT billed_charges, net_charges FROM claim_summary_XYZ
   UNION ALL
   SELECT billed_charges, net_charges FROM claim_summary_ABC;
QUIT;
```

manufacturing_claims	
billed_charges	net_charges
$188.05	**$182.58**
$5,687.98	$5,321.07
$60,677.52	$60,332.12
$188.05	**$182.58**
$672.33	$650.33
$6,537.52	$6,340.12

Of course, in doing this, we have also created a dataset with duplicate records that cannot be differentiated from one another, and this situation should generally be avoided.

One should also note that when performing a UNION JOIN, the use of the asterisk (*) wildcard to indicate that all columns are to be selected from the source datasets can result in some unexpected results if one is not aware of how the UNION JOIN selects columns. By default the variables in the target dataset "manufacturing_claims" will be defined based on the column names and data types in the first source dataset in the UNION JOIN. So, for example, if the second column in dataset "claim_summary_xyz" is "billed_charges", the second column of "manufacturing_claims" will be "billed_charges". This works fine as long as the second column of "claim_summary_abc" is also "billed_charges", but if (as in the following example) the second variable in "claim_summary_abc" is "net_charges", the resulting dataset will have a column called "billed_charges" that is populated by the "billed_charges" from "claim_summary_xyz" and the "net_charges" from "claim_summary_abc".

claim_summary_xyz		
member_id	**billed_charges**	net_charges
0123345	$188.05	$182.58
0123346	$5,687.98	$5,321.07
0123347	$60,677.52	$60,332.12

claim_summary_abc		
member_id	**net_charges**	billed_charges
0024579	$182.58	$188.05
0024580	$650.33	$672.33
0224581	$6,340.12	$6,537.52

```
PROC SQL;
 CREATE TABLE work.manufacturing_claims as
   SELECT * FROM claim_summary_XYZ
   UNION
   SELECT * FROM claim_summary_ABC;
QUIT;
```

manufacturing_claims		
member_id	**billed_charges**	net_charges
0123345	*$188.05*	**$182.58**
0123346	*$5,687.98*	**$5,321.07**
0123347	*$60,677.52*	**$60,332.12**
0024579	**$182.58**	*$188.05*
0024580	**$650.33**	*$672.33*
0224581	**$6,340.12**	*$6,537.52*

The point here is to be sure you know the structure of the datasets with which you are working. SAS will not warn you that you are mixing "billed_charges" and "net_charges" because it has no way of knowing whether the two columns contain comparable data. However, SAS will raise an error when you try to UNION JOIN columns that are of different data types.

claim_summary_xyz		
member_id	billed_charges	net_charges
0123345	$188.05	$182.58
0123346	$5,687.98	$5,321.07
0123347	$60,677.52	$60,332.12

claim_summary_abc		
member_id	fname	net_charges
0024579	Mary	$188.05
0024580	Tom	$672.33
0224581	Joseph	$6,537.52

```
PROC SQL;
 CREATE TABLE work.manufacturing_claims as
   SELECT * FROM claim_summary_XYZ
   UNION
   SELECT * FROM claim_summary_ABC;
QUIT;

ERROR: Column 2 from the first contributor of UNION is not the same type as its
       counterpart from the second
```

For datasets that do have the same column names, the order of the columns can be removed as a source of errors by using the UNION CORRESPONDING operator. Applied to the previous example in which the datasets share the same column names (but have a different order within the dataset), this form of the UNION JOIN aligns the variables based on their names.

claim_summary_xyz		
member_id	billed_charges	net_charges
0123345	$188.05	$182.58
0123346	$5,687.98	$5,321.07
0123347	$60,677.52	$60,332.12

claim_summary_abc		
member_id	net_charges	billed_charges
0024579	$182.58	$188.05
0024580	$650.33	$672.33
0224581	$6,340.12	$6,537.52

```
PROC SQL;
 CREATE TABLE work.manufacturing_claims as
   SELECT * FROM claim_summary_XYZ
   UNION CORRESPONDING
   SELECT * FROM claim_summary_ABC;
QUIT;
```

manufacturing_claims		
member_id	**billed_charges**	net_charges
0123345	**$188.05**	*$182.58*
0123346	**$5,687.98**	*$5,321.07*
0123347	**$60,677.52**	*$60,332.12*
0024579	**$188.05**	*$182.58*
0024580	**$672.33**	*$650.33*
0224581	**$6,537.52**	*$6,340.12*

In addition to the UNION, UNION ALL, and UNION CORRESPONDING operators, PROC SQL also facilitates the selection of the intersection of two datasets and the exclusion of the intersecting records from one of a set of possibly intersecting datasets with the INTERSECT and EXCEPT operators, respectively. Specifically INTERSECT returns the rows that exist in both datasets.

analysis1		
member_id	billed_charges	net_charges
0123345	$188.05	$182.58
0123346	$5,687.98	$5,321.07
0123347	$60,677.52	$60,332.12

analysis2		
member_id	billed_charges	net_charges
0123346	$5,687.98	$5,321.07
0024580	$650.33	$672.33
0224581	$6,340.12	$6,537.52

```
PROC SQL;
 CREATE TABLE work.cases_in_common AS
    SELECT claim_id, billed_charges, net_charges FROM analysis1
    INTERSECT
    SELECT claim_id, billed_charges, net_charges FROM analysis2;
QUIT;
```

cases_in_common		
member_id	billed_charges	net_charges
0123346	$5,687.98	$5,321.07

Conversely, EXCEPT returns those records from the first dataset that are not in the second dataset. The EXCEPT operator starts with all rows in the first dataset and removes from that potential result-set those rows that are found in the subsequent dataset(s).

```
PROC SQL;
 CREATE TABLE work.cases_not_in_analysis2 AS
    SELECT claim_id, billed_charges, net_charges FROM analysis1
    EXCEPT
    SELECT claim_id, billed_charges, net_charges FROM analysis2;
QUIT;
```

cases_not_in_analysis2		
member_id	billed_charges	net_charges
0123345	$188.05	$182.58
0123347	$60,677.52	$60,332.12

Together, the EQUI-JOIN, LEFT, RIGHT, and FULL OUTER JOINs, and UNION joins provide a powerful way to combine datasets to create a wide variety of combinations of records from related datasets.

SUBQUERIES & CORRELATED SUB-QUERIES

With an understanding of how datasets are joined in PROC SQL queries, we now look at how the universe of possible target datasets can be further expanded using subqueries. Subqueries are queries that are nested within another SQL statement (usually a SELECT statement, but also UPDATE, DELETE, and INSERT statements) for the purpose of operating on the records in the "outer" query based on the results of another, nested, query—the subquery. In the following example, we want to select all of the medical claim records for diabetic patients. Our chronic disease database contains a table that identifies all diabetic patients by their member number (memberno). Therefore, we can use the following subquery to accurately select the claims data for diabetic health plan members:

```
PROC SQL;
 CREATE TABLE work.diabetic_financials AS
  SELECT *
    FROM work.claim_detail
   WHERE memberno IN (SELECT memberno
                        FROM chronic.diabetic_population);
QUIT;
```

The subquery in the previous SELECT statement (i.e., selecting "memberno" from "diabetic_population") is an example of a multi-value subquery; that is, a subquery that returns more than one record. The results of this subquery (i.e., the memberno values associated with each member in the diabetic population) will be used by the WHERE clause of the outer query to determine which records should be selected from the "claim_detail" dataset. The subquery will be executed first and the results cached; then the outer query is executed and the value of "memberno" in each of the "claim_detail" records is assessed for inclusion in the list of "memberno" values returned by the subquery. Those records from "claim_detail" that are found in the subquery result-set are output to the dataset "diabetic_financials".

Note that this query could also be performed by executing the following equi-join:

```
PROC SQL;
 CREATE TABLE work.diabetic_financials AS
  SELECT a.*
    FROM work.claim_detail a
         chronic.diabetic_population b
   WHERE a.memberno = b.memberno;
QUIT;
```

However, if what you wanted to do was update a "diabetic_flag" variable on the "claim_detail" dataset, you might use the following subquery:

```
PROC SQL;
 UPDATE TABLE work.claim_detail
  SET diabetic_flag = 'Y'
WHERE memberno IN (SELECT memberno
                     FROM chronic.diabetic_population);
UPDATE TABLE work.claim_detail
  SET diabetic_flag = 'N'
WHERE memberno NOT IN (SELECT memberno
                         FROM chronic.diabetic_population);
QUIT;
```

Like the previous SELECT statement that utilized a subquery to drive which records should be selected from "claim_detail", the UPDATE statement in the preceding example uses the subquery to determine which records should be updated with the values of "Y" and "N". Those records in "claim_detail" that have a member number found in the subquery of the diabetic population will be updated with the value of 'Y' assigned to "diabetic_flag"; those not in the result-set will be assigned the value 'N'.

Whereas the previous subquery examples result in a single column and multiple rows to specify the subset that the outer query uses to define the target dataset, there is another type of subquery that results in a single column and a single row to determine the outcome of the outer query. This type of subquery, called a single-value subquery, is used to evaluate records in the outer query in terms of the relationship between a specific value on each record and a single value returned by the subquery. In the following example, we evaluate the inpatient hospital admissions for health plan members admitted in July 2010 to identify those inpatient admissions with a "length of stay" (LOS) that is more than two standard deviations higher than the average length of stay across all inpatient admissions recorded in our claims system.

```
PROC SQL;
 CREATE TABLE work.admissions_for_review AS
 SELECT *
   FROM work.july_2010_admits
  WHERE los GTE (SELECT AVG(length_of_stay) + 2*STD(length_of_stay)
                   FROM claims.inpatient_admissions);
QUIT;
```

Whereas multi-value subqueries evaluate records for inclusion in a subset specified by the subquery, we see in the preceding example that the records in "july_2010_admits" are being evaluated with a "greater than or equal to" (GTE, or >=) operator. Because of the type of comparison being made, it is important that the subquery written evaluates to one and only one value (in this case the average length of stay plus two standard deviations). Alternatively, if we executed the following PROC SQL statement, SAS would generate an error:

```
PROC SQL;
 CREATE TABLE work.admissions_for_review AS
 SELECT *
   FROM work.july_2010_admits
  WHERE los GTE ( SELECT AVG(length_of_stay) + 2*STD(length_of_stay)
                    FROM claims.inpatient_admissions
                GROUP BY client_id);
QUIT;

ERROR: Subquery evaluated to more than one row.
```

The error occurs because you are asking SAS to evaluate the value of "los" against more than one value simultaneously—the average LOS plus two standard deviations, for each client company. Because a given record in the source dataset (july_2010_admits) may be above that value for one client company, but below that value for another, the evaluation cannot be performed.

When performing these types of data quality checks, it is often necessary to perform the range checks against data that are comparable along some other dimension. For example, in trying to flag length of stay values for hospital admissions to identify extreme outliers that may represent data errors, it makes sense to compare the inpatient admission records to those admissions associated with the same diagnoses. In order to make such comparisons, we will need to use a correlated subquery. A correlated subquery is a subquery in which data from the outer query are used to define the subquery so that the results of the subquery are appropriate for making the comparison to the record being evaluated in the outer query. In the following query, we again evaluate the inpatient admissions data looking for outliers in length of stay, but this time we correlate (or, make comparable) the result of the subquery with the outer join record being evaluated.

```
PROC SQL;
 CREATE TABLE work.admissions_for_review AS
 SELECT *
   FROM work.july_2010_admits a
  WHERE los GTE ( SELECT AVG(length_of_stay) + 2*STD(length_of_stay)
                    FROM claims.inpatient_admissions b
                   WHERE a.principle_diagnosis = b.principle_diagnosis);
QUIT;
```

In other words, this query compares the length of stay for each record in the "july_2010_admits" dataset to the average length of stay (plus two standard deviations) from all previous inpatient admissions that have the same principal diagnosis as the record being evaluated in the outer query. The average length of stay being calculated in the subquery is that associated with the principle diagnosis in the current outer query record.

Unlike non-correlated subqueries, which cache query results before executing the outer query, the results of correlated subqueries cannot be cached ahead of time because of their dependence on data from the outer query record being evaluated. Only as each new, unique value of the outer query variable is passed to the subquery can the correlated subquery compute the appropriate value for comparison. As a result, correlated subqueries can cause performance problems when they involve large datasets that must be scanned for a large number of possible responses. Despite this limitation, however, they can be a very powerful tool for manipulating datasets with PROC SQL.

CONNECTING TO RELATIONAL DATABASES

SAS/ACCESS LIBRARIES

In addition to the powerful data manipulation capabilities it provides, PROC SQL has become as popular as it has at least partly due to the fact that relational database management systems have come to dominate how we store, manage, and even think about operational data. As a result, it is increasingly rare that raw source data exists in the form of SAS datasets or ASCII files that must be read in with an INPUT statement. Instead, users connect to live production systems, operational data stores, or data warehouses that reside on remote database servers. In order to facilitate access to these systems, SAS allows users (via SAS/ACCESS) to connect directly to the databases to which they have access and read the associated database tables and views much as they would SAS datasets that reside on local and network drives.

In the following example, we define our SAS Library "hca" as a connection to the Microsoft SQL Server Database "UHHCA" maintained by "University Hospital's Health Care Analytics department. This library definition specifies that the database type is an Object Linking and Embedding (OLE) connection to the system interface SQLOLEDB.1. The library specification further defines the "initial catalog" as the database "UHHCA" that is running on the SQL Server named "HOSP-HCA". The bulkcopy processor option (BCP=Yes) is being utilized (which will allows loads to the database tables to proceed without committing every <x> rows) and the database schema to which the SAS library "hca" will attach is "DBO".

```
LIBNAME hca OLEDB PROVIDER=SQLOLEDB.1 REQUIRED=Yes
         USER = cschacherer
   DATASOURCE = "HOSP-HCA"
   PROPERTIES = ('initial catalog'=UHHCA 'Persist Security Info'=True)
          BCP = Yes
       SCHEMA = 'DBO';
```

In the next example, the library "billing" is defined as a connection to the "claims" schema on an Oracle database identified by its network shortcut (path) "financials".

```
LIBNAME billing ORACLE
        USER = 'cschacherer'
    DPPROMPT = YES
        PATH = "financials"
      SCHEMA = claims;
```

It should be noted that the type of database to which one is connecting will dictate the parameters of the library definition. For example, in the Oracle connection we specify the PATH whereas in the SQL Server connection we specify the DATASOURCE. However, in all SAS/ACCESS connections to database management systems, the type of database to which one is attempting a connection (e.g., OLEDB, ORACLE, DB2, etc.) is specified following the library name and is followed by the parameters required by the SAS/ACCESS engine for that particular database system. In addition, many database vendors have their own proprietary networking and client software that must be installed on the system running SAS in order to enable a connection to the database. For more detailed information on defining libraries as connections to databases, the reader is referred to Riley (2008) and SAS Institute, Inc. (2004a; 2010a) for discussions of SAS/ACCESS connections to Microsoft SQL Server and OLE DB Datasources, Rhodes (2007), Levin (2004), SAS Institute, Inc. (2004b), and Schacherer (2008) for SAS/ACCESS connectivity to

Oracle, and Gona & Van Dyk (1998) for an earlier, but still very relevant, description of SAS/ACCESS and the LIBNAME statement. For more advanced treatments of SAS/ACCESS programming efficiencies, see Levine (2001) and Helf (2002).

Once connected to the database, PROC SQL SELECT statements can be executed largely as though they were written against native SAS datasets. For example, in order to make a local copy of the dataset "claim_details" from our Oracle billing system, you could issue the following PROC SQL SELECT statement:

```
PROC SQL;
 CREATE TABLE work.claim_details AS
  SELECT *
     FROM billing.claim_detail;
QUIT;
```

Likewise, if you have the appropriate privileges, you can also insert, delete, and update data on these tables using PROC SQL as well. For this reason, we strongly recommend that you check with your database administrator regarding your privileges on any databases to which you have login privileges and make sure he or she is familiar with how you intend to access and use the database using SAS. We have worked with several analysts that assume that they do not have any database privileges that would allow them to "harm" the production environment only to find out later that they could accidentally delete all of the data in a production environment.

In fact, using SQL syntax that we have not discussed in the previous sections of this paper, one can (again, with the appropriate privileges) even completely drop a production database table.

```
PROC SQL;
 DROP TABLE billing.claim_detail;
QUIT;
```

So before connecting to your database with SAS, make sure you discuss your plans with your database administrator so that he or she is aware of what you are planning to do and can discuss with you any pertinent details of your user privileges in the database.

Unlike using PROC SQL to operate against SAS datasets, however, there are some differences one encounters when issuing SQL commands against a SAS/ACCESS library connecting to a relational database. For example, when creating SAS datsets, you can create a table with the same name twice and the second execution will simply overwrite the first.

```
PROC SQL;
 CREATE TABLE work.claim_summary AS
  SELECT *
     FROM billing.claim_detail;
QUIT;

PROC SQL;
 CREATE TABLE work.claim_summary AS
  SELECT billing_id, line_number, memberno
     FROM billing.claim_detail;
QUIT;
```

If we try to execute those same PROC SQL statements to create tables on the relational database, however, the second execution will generate an error, because a table with the name "claim_summary" will already exist on the Oracle "claims" database.

```
PROC SQL;
 CREATE TABLE billing.claim_summary AS
  SELECT *
     FROM billing.claim_detail;
QUIT;
```

```
PROC SQL;
 CREATE TABLE billing.claim_summary AS
  SELECT billing_id, line_number, memberno
     FROM billing.claim_detail;
QUIT;

ERROR: The ORACLE table claim_summary has been opened for OUTPUT. This table already exists,
       or there is a name conflict with an existing object. This table will not be replaced.
```

Similarly, there are database options adopted by some relational database vendors that can make data more difficult to access with SAS. For example, the following SELECT statement against the Microsoft SQL Server table 'this_is_a_very_very_long_table_name' cannot be executed as written because the SAS/ACCESS engine for OLEDB does not recognize names longer than 32 characters.

```
PROC SQL;
 CREATE TABLE work.local_longname AS
  SELECT * FROM hca.this_is_a_very_very_long_table_name;
QUIT;

ERROR 65-58: Name 'THIS_IS_A_VERY_VERY_LONG_TABLE_NAME' is too long for a SAS name
             in this context.
```

In this situation (under Microsoft SQL Server 2005 and beyond), you can work around this issue by creating a synonym for this table in the SQL Server environment that has a name that is 32 characters or less[4]. A synonym is simply a pointer to the longer table name. In the following example, we have created the database synonym "shorter" for the long table name in the previous example, and we can utilize the data in the database table by referencing the synonym instead of the actual table name:

```
PROC SQL;
 CREATE TABLE work.local_longname AS
  SELECT * FROM hca.shorter;
QUIT;
```

Despite these minor challenges, however, the SAS/ACCESS engines for relational databases make accessing data via SAS libraries very straight-forward.

PASS-THROUGH SQL

For users that are making the transition to SAS PROC SQL from a database vendor's proprietary implementation of the SQL language, however, one consistent point of frustration in accessing the database through a SAS/ACCESS library is the inability to use functions and stored procedures written in that language. As one simple example, the following query attempts to use the Oracle's "to_date" function to convert date literal strings to dates that can be used in a WHERE clause:

```
PROC SQL;
 CREATE TABLE WORK.july_2009_charge_summary AS
   SELECT a.member, a.gender, b.net_payment
     FROM claims.members a LEFT JOIN claims.claim_details b
       ON a.member = b.member
    WHERE bill_date BETWEEN TO_DATE('07/01/2009','MM/DD/YYYY')
                        AND TO_DATE('07/31/2009','MM/DD/YYYY');
QUIT;

ERROR: Function TO_DATE could not be located.
ERROR: Function TO_DATE could not be located.
```

[4] In earlier version of SQL Server (and in database systems that do not support aliases, you can also achieve this connection through a database view).

SAS cannot locate the "TO_DATE" function because it is an Oracle PL/SQL function—not a SAS function. SAS has no way to interpret what you are trying to do because it does not "speak" PL/SQL. Of course, you could just as easily resolve this problem by coding the date literals in a manner that SAS can understand, but the point here is that you cannot simply call database functions and stored procedures that are proprietary to the relational database management system's implementation of SQL simply by adding those function calls to your query. Whereas this is not a significant problem in cases where those function calls that can be easily replaced by equivalent SAS functions or syntax, it becomes a significant problem when you need to utilize functionality that is only available within the database. For example, as of the time of this writing, PROC SQL does not support the TRUNCATE command (which deletes all records from a database table without firing any database triggers as the rows are deleted). Moreover, your database development team may have written custom functions and stored procedures that can help you access the results of complex algorithms or special subsets of your data. By simply executing queries against the database through a SAS/ACCESS library, you will not be able to call these custom solutions because they are not commands that SAS recognizes.

The answer in both of these cases is to use PASS-THROUGH SQL. Pass-through SQL is a special type of PROC SQL command syntax that allows you to execute database commands in the native language of the database management system to which you are connected. In the following example, we will use pass-through SQL to execute the SQL query we attempted previously against our Oracle datasource:

```
PROC SQL;
 CONNECT to ORACLE
 (     USER = cschacherer
   DBPROMPT = yes
       PATH = "billing"
     SCHEMA = );
CREATE TABLE WORK.july_2009_charge_summary AS
 SELECT * FROM CONNECTION TO ORACLE
         (SELECT a.member, a.gender, b.net_charges
            FROM claims.members a LEFT JOIN claims.claim_details b
              ON a.member = b.member
           WHERE bill_date BETWEEN TO_DATE('07/01/2009','MM/DD/YYYY')
                               AND TO_DATE('07/31/2009','MM/DD/YYYY'));
DISCONNECT FROM ORACLE;
QUIT;
```

In this example, we are establishing a connection to Oracle using the same credentials we previously used for creating our SAS/ACCESS library, but instead of establishing a SAS library, we are simply establishing a temporary connection to Oracle. We then issue our PROC SQL CREATE TABLE command to specify the name and location of the local SAS dataset we want to create. At this point, instead of creating the dataset by directly querying the database through a SAS/ACCESS library, we specify that the new SAS dataset is to be created by selecting all results returned from our connection to Oracle. We then pass to this connection the PL/SQL syntax that we want to execute on the database—in this case the query selecting "member", "gender" and "net_charges" from the tables "members" and "claim_details". Because Oracle understands the "TO_DATE" function perfectly well, it executes the query without error. The result-set produced by this query is then used as the datasource for our "SELECT * FROM CONNECTION TO ORACLE", and the target dataset "WORK.july_2009_charge_summary" is created. We then disconnect our from the Oracle database.

Note that what is happening in pass-through SQL is that we are packaging Oracle (or SQL Server, or DB2) syntax, passing it to the database engine, and querying the result-set generated by that syntax. It is as if we are telling SAS, "don't try to interpret this, you might not understand some of the syntax; just send it to the database and let the database engine intrepret the syntax".

Similarly, you can use pass-through SQL to perform tasks that are part of more complex database maintenance tasks. For example, using SAS to perform extract, transform, and load processes for building and maintaining dimensional data models for business intelligence solutions, we use pass-through SQL to perform all of the SQL Server maintenance involved in loading new data to an analytic data mart (Schacherer, 2010). In the following example, prior to loading the SQL Server database tables with new, remodeled data, pass-through SQL is used to truncate the data in those tables:

```
PROC SQL;
 CONNECT TO OLEDB
        (provider=SQLOLEDB.1 required = Yes
         datasource="BI-PROD"
         properties=('initial catalog'=HCA
                     'Persist Security Info'=True
                     'Integrated Security' = SSPI)
         schema=DBO bcp=Yes);

 EXECUTE(truncate table dbo.patient_dim
         truncate table dbo.procedure_dim
         truncate table dbo.payor_dim
         truncate table dbo.diagnosis_dim
         truncate table dbo.provider_dim
         truncate table dbo.department_dim
         truncate table dbo.clinic_dim
         truncate table dbo.billing_fact)
      BY oledb;
DISCONNECT FROM oledb;
QUIT;
```

So, whether you are connecting to a relational database to simply extract data via a SAS/ACCESS library or need to interact with the database in a language that only the relational database management system engine can understand, SAS/ACCESS can be used to significantly extend the power of PROC SQL.

CONCLUSION

From its beginnings in SAS 6.06, the SQL PROCEDURE has come to be one of the most powerful data management tools available to SAS programmers. Whether using it in ways that mimic DATA STEP processing techniques or to summarize, join, or update datasets, we hope you are convinced of the flexibility of this extraordinary tool and will continue to discover creative new ways to use it in your work.

REFERENCES

Cody, R. (2010). Using Advanced Features of User-Defined Formats and Informats. Proceedings of the SAS Global Forum 2010. Cary, NC: SAS Institute, Inc.

DeFoor, J. (2006). Proc SQL - A Primer for SAS Programmers. Proceedings of the 31st Annual SAS Users Group International Meeting. Cary, NC: SAS Institute, Inc.

Dickstein, C. & Pass, R. (2004). DATA Step vs. PROC SQL: What's a neophyte to do? Proceedings of the 29th Annual SAS Users Group International Meeting. Cary, NC: SAS Institute, Inc.

Foley, M. (2005). MERGING vs. JOINING: Comparing the DATA Step with SQL. Proceedings of the 30th Annual SAS Users Group International Meeting. Cary, NC: SAS Institute, Inc.

Gona, V. & Van Wyk, J. (1998). Version 7 Enhancements to SAS/ACCESS Software. Proceedings of the 23rd Annual SAS Users Group International Meeting. Cary, NC: SAS Institute, Inc.

Helf, G.W. (2002). Can't Relate? A Primer on Using SAS With Your Relational Database. Proceedings of the 27th Annual SAS Users Group International Meeting. Cary, NC: SAS Institute, Inc.

Lafler, K.P. (2005). Manipulating Data with PROC SQL. Proceedings of the 30th Annual SAS Users Group International Meeting. Cary, NC: SAS Institute, Inc.

Lafler, K.P. (2004a). Efficiency Techniques for Beginning PROC SQL Users. Proceedings of the 29th Annual SAS Users Group International Meeting. Cary, NC: SAS Institute, Inc.

Lafler, K.P. (2004b). PROC SQL: Beyond the Basics Using SAS. Cary, NC: SAS Institute, Inc.

Lafler, K.P. (2003). Undocumented and Hard-to-find SQL Features. Proceedings of the 28th Annual SAS Users Group International Meeting. Cary, NC: SAS Institute, Inc.

Levin, L. (2004). Methods of Storing SAS Data into Oracle Tables. Proceedings of the 29th Annual SAS Users Group International Meeting. Cary, NC: SAS Institute, Inc.

Levine, F. (2001). Using SAS/ACCESS Libname Technology to Get Improvements in Performand and Optimizations in SAS/SQL Queries. Proceedings of the 26th Annual SAS Users Group International Meeting. Cary, NC: SAS Institute, Inc.

Lund, P. (2005). An Introduction to SQL in SAS. Proceedings of the 30th Annual SAS Users Group International Meeting. Cary, NC: SAS Institute, Inc.
Rhodes, D.L. (2007). Talking to Your RDBMS Using SAS/ACCESS. Proceedings of the SAS Global Forum 2007. Cary, NC: SAS Institute, Inc.
Riley, C. (2008). Getting SAS to Play Nice With Others: Connecting SAS to Microsoft SQL Server Using an ODBC Connection. Proceedings of the SAS Global Forum 2008. Cary, NC: SAS Institute, Inc.
SAS Institute, Inc. (2004a). SAS/ACCESS 9.1 Supplement for Microsoft SQL Server. Cary, NC: SAS Institute, Inc.
SAS Institute, Inc. (2004b). SAS/ACCESS 9.1 Supplement for Oracle. Cary, NC: SAS Institute, Inc.
SAS Institute, Inc. (2004c). SAS 9.1 SQL Procedure User's Guide. Cary, NC: SAS Institute, Inc.
SAS Institute, Inc. (2010a). Accessing a Microsoft SQL Server Database from SAS on Microsoft Windows. SAS Technical Support Document TS-765. Retrieved July 28, 2010 from: http://support.sas.com/techsup/technote/ts765.pdf..
Schacherer, C.W. (2010). Base SAS Methods for Building Dimensional Data Models. Proceedings of the Midwest SAS Users Group.
Schacherer, C.W. (2008). Utilizing SAS as an Integrated Component of the Clinical Research Information System. Proceedings of the SAS Global Forum 2008. Cary, NC: SAS Institute, Inc.
Schacherer, C.W. & Westra, B.D. (2010). A SAS Primer for Healthcare Data Analysts. Proceedings of the SouthCentral SAS Users Group.
Whitlock, I. (2001). PROC SQL – Is it a Required Tool for Good SAS Programming? Proceedings of the 26th Annual SAS Users Group International Meeting. Cary, NC: SAS Institute, Inc.
Williams, C.S. (2008). PROC SQL for DATA Step Die-hards. Proceedings of the SAS Global Forum 2008. Cary, NC: SAS Institute, Inc.

ACKNOWLEDGMENTS

The authors wish to thank Brent Westra of Mayo Clinic Health Solutions for lending his content expertise to several of the healthcare claims examples.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. You can contact the author at:

Christopher W. Schacherer, Ph.D.
Clinical Data Management Systems, LLC
6666 Odana Road #505
Madison, WI 53719
Phone: 608.478.0029
E-mail: CSchacherer@cdms-llc.com
Web: www.cdms-llc.com