Paper 43-28

Dynamically Building SQL Queries Using Metadata Tables and Macro Processing

Michael Molter, Excellus Blue Cross and Blue Shield, Rochester, NY
Scott Millard, Excellus Blue Cross and Blue Shield, Utica, NY
Steve Paciocco, Excellus Blue Cross and Blue Shield, Rochester, NY

ABSTRACT

The introduction of SAS/ACCESS® has allowed SAS® users to process information contained in a variety of database formats. The relative ease with which SAS/ACCESS makes this possible is often offset with poor performance caused by inefficient use of native database services. The addition of SQL pass-thru does allow users to process native databases efficiently but requires a solid understanding of SQL. The challenge of many data warehousing efforts today is one of balancing efficient database design (STAR schemas) and ease of use. Often times this challenge is met by purchasing additional software to administer complex join structures and generate the SQL needed. This paper addresses this challenge of generating efficient SQL easily from within the SAS environment. It makes use of Base SAS® and the macro facility as well as SAS Access. It is not limited to any operating system, and is intended for intermediate and above audiences.

INTRODUCTION

After five years with an Oracle star schema database and an SQL-generating query tool from a separate software vendor, our health care organization decided to re-assess its choice of query tools. Among the six different products under consideration was SAS's Enterprise Guide®. One of the leading factors in making this decision was how easy it is for a non-programmer to create and submit a query. The complexities of the database had to be completely hidden from the end user. Unfortunately, because it was the only one of the six that required users to define their own SQL joins from table to table, Enterprise Guide was immediately dismissed.

Fortunately, the power that is available through the Base SAS language and the macro processor allows a programmer to fill in holes that may exist in some of the pre-packaged products. For us, this became a necessity. Despite missing out on the sale of several hundred licenses for Enterprise Guide, SAS still maintains a presence of Base SAS and SAS ACCESS on twenty-five desktops within our analytical department. To facilitate the flow from data extract to data manipulation and analysis, we needed SAS solutions to our Oracle databases. Because of the complexities inherent within the database structure, and therefore, the complexities of the SQL code necessary to query from it, we were charged with developing what Enterprise Guide was missing – an SQL generator that produced joins between tables based on user input. It was to be written as a macro for people with at least a basic knowledge of SAS or Oracle syntax. This paper describes the development and use of metadata tables to derive the SELECT and FROM clauses as well as the JOIN conditions for such a tool using SAS's macro language.

THE APPROACH

The goal for this paper is to resolve the macro variables in the following code, found at the end of the macro.

```
SELECT &SLCT
FROM &FRM
WHERE &JN
```

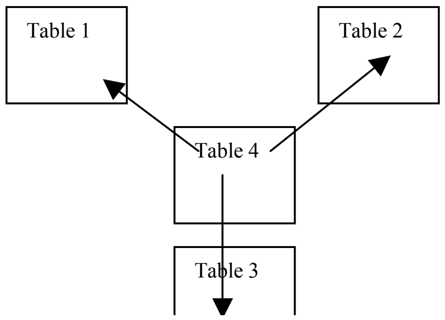
What should be clear, if not by the macro variable names, then by the location of their references within the query, is an idea of what these variables resolve to. &SLCT is a list of fields to be kept, separated by commas. &FRM is a list of tables separated by commas. &JN is the collection of join conditions separated by the word AND. Alternatively, these variables may be replaced by %DO loops that generate the lists. The mystery, and the theme of the paper, is in how these are derived. To get an idea of where the process would begin, consider any other SQL generator. How does it know which fields to pull? Which tables are needed? Which joins need to be created? As you point and click or drag and drop icons that represent database fields, the software interprets and translates your input and generates a SELECT clause. But a good query tool does not need to require you to know which tables contain the chosen fields, or how such tables are related. So how does the tool know which tables to list in the FROM clause, and which JOIN conditions to invoke in the WHERE clause? In what form does "the brain behind the software" exist, and how is it used? Somewhere, somehow, a storage facility must be in place that contains this information.

The macro method described here makes use of metadata tables to store such information. Of course, the macro's method of input is the macro parameter, in this case, SELECT=, in which you enter the fields you wish to see. Just as a point-and-click application does not need you to click on table names or primary keys to define JOIN conditions, a macro query tool does not need FROM and JOIN parameters. Instead, a separate database (or meta-database) is created. The macro enters the meta-database with SELECT= fields, and for each field, finds table, primary key, and foreign key information. The ability to create an SQL-generating SAS macro depends on the ability to set up these meta-databases

AN EXAMPLE

The layout of the metadata table(s) depends entirely on the structure of the database. With the right metadata, most, if not all of the SQL can be generated from macro code that uses this table. Ideally, little, if any SQL would be forced by macro code that was independent of the table (e.g. %LET statements). Consider an example in which a company keeps track of its sales in a basic star database with four tables. In such a database, each of three tables has a connection to the fourth, and none of the three is connected to each other (see figure 1). In other words, the fourth has among all of its fields, one foreign key for each of the other three tables. Often times, this fourth table is used to keep track of transactions, while the other three tables each provide more information about certain aspects of the fourth. For this example, suppose information about each sale is recorded in a table called TRANSACTIONS. A second table, EMPLOYEES, has information about each employee. A third table, CLIENTS, contains information about clients, and a fourth table, DATES, contains fields such as YEAR, MONTH, and QUARTER pertaining to any given date. TRANSACTIONS has fields for employee and client codes as well as a date field that each serve as foreign keys to their respective tables.

Figure 1: A basic four-table star schema



SETTING UP THE METADATA

Based on this structure, one metadata table (call it META) could be set up to generate the SQL for the query. META consists of four fields – FIELD, TABLE, PRIM_KEY, and FOR_KEY. The values of FIELD are all the field names that occur in the database. For any value of FIELD, the value of TABLE is the name of the table that contains that field. For rows where the value of TABLE is DATES, EMPLOYEES, or CLIENTS, the value of PRIM_KEY is the name of the primary key from that table, and the value of FOR_KEY is the value of the foreign key in TRANSACTIONS that links to PRIM_KEY. For rows where the value of TABLE is TRANSACTIONS, these two fields are blank. Assume in this example that no field name is found in more than one table so that table aliases are not necessary. See table 1.

Table 1: META

Field	Table	PRIM_KEY	FOR_KEY
CLIENTCODE	TRANSACTIONS		
EMPCODE	TRANSACTIONS		
DATE	TRANSACTIONS		
SALESAMT	TRANSACTIONS		
COMMISSION	TRANSACTIONS		
PPLAN	TRANSACTIONS		
DPLAN	TRANSACTIONS		
FNAME	EMPLOYEES	EMPKEY	EMPCODE
LNAME	EMPLOYEES	EMPKEY	EMPCODE
PHONE	EMPLOYEES	EMPKEY	EMPCODE
ADDRESS	EMPLOYEES	EMPKEY	EMPCODE
ZIP	EMPLOYEES	EMPKEY	EMPCODE
QUARTER	DATES	DATEKEY	DATE
MONTH	DATES	DATEKEY	DATE
YEAR	DATES	DATEKEY	DATE
YYYYMM	DATES	DATEKEY	DATE
TYPE	CLIENTS	CLIENTKEY	CLIENTCODE
CITY	CLIENTS	CLIENTKEY	CLIENTCODE
STATE	CLIENTS	CLIENTKEY	CLIENTCODE

ACCESSING META

The first step in generating any particular SELECT and FROM lists and the JOIN conditions is to create a data set (or table) that is a subset of META, containing only the observations where the value of FIELD is one of the fields specified in the SELECT= parameter. Call this table SUBSET. The values of FIELD in SUBSET will make up the SELECT list, the values of TABLE will make up the FROM clause, and the values of PRIM_KEY and FOR_KEY will help generate the JOIN conditions. SUBSET can be created with a simple DATA step or a SQL procedure. The IN list in the WHERE statement or clause would be made up of the fields specified in the SELECT= parameter, and can be read from the parameter in one of a number of ways, such as the following:

```
proc sql;
create table subset as
select *
from sugi28.meta
where field in (
        %let a=1 ;
        %let token=%scan(&select,1,%str( )) ;
        %do %until(&token=%str());
                %if &a=1 %then "&token" ;
                %else ,"&token" ;
                %let a=%eval(&a+1) ;
                %let token=%scan(&select,&a,%str( )) ;
        %end ;
)
order by table ;
quit;
```

Each iteration of the %DO loop, corresponds to a selected database field and inserts this field into the IN list. On all except the first iteration, the field name is preceded by a comma. Note the ORDER BY clause. This will allow you to use TABLE in a BY statement in the upcoming DATA step.

You now have a data set that contains the fields you need, and is ordered by TABLE. It also contains all of the JOIN relationships that are necessary. The question remains, what and how do you transfer the information in the data set to an SQL query against the database. The question of how is easy. The CALL SYMPUT routine, invoked inside a DATA step, allows you to remember values from a data set after the DATA step has completed, by assigning these values to macro variables. Therefore, the next step is to read SUBSET and assign its values to macro variables. To answer the question of what can be used from SUBSET, start with the SELECT clause. Every value of FIELD is needed in the SELECT clause, so assign each value a macro variable as follows:

```
select=compress('S' || put(_n_,3.));
call symput(select,trim(left(field)));
```

To generate the FROM clause, note that since any particular value of TABLE is likely to appear on multiple observations of SUBSET, not every value on every observation will need to be assigned a macro variable. Therefore, assign a macro variable only at LAST.TABLE.

```
if last.table then do;
        tbl_nbr+1;
        from=compress('F'||put(tbl_nbr,3.));
        call symput(from,trim(left(table)));
```

Finally, just as with the macro variables that build the FROM clause, macro variables to build JOIN conditions are also needed only once per table, and are not needed at all when that table is TRANSACTIONS. So within the same DO block stated above, include the following:

```
        if key1 ne ' ' then do; /* this is one way to identify when
        TRANSACTIONS is the table */
        jn_nbr+1;
        join=compress('J' || put(jn_nbr,3.));
        call symput(join,trim(left(key1)) || "=" || trim(left(key2)));
        end;
```

The entire data step is shown below:

```
data _null_;
set subset end=lastone nobs=sel_nbr;
by table;
length from $ 4 select $ 4 join $4 ;
retain jn_nbr 0 ;

select=compress('S' || put(_n_,3.));
call symput(select,trim(left(field)));

If last.table then do;
        tbl_nbr+1;
        from=compress('F' || put(tbl_nbr,3.));
        call symput(from,trim(left(table)));

        if key1 ne ' ' then do;
        jn_nbr+1;
```

```
            join=compress('J' || put(jn_nbr,3.));
            call symput(join,trim(left(key1)) || "=" || trim(left(key2)));
            end;
    end;

    if lastone then do;
            call symput("sel_nbr",sel_nbr);
            call symput("tbl_nbr",tbl_nbr);
            call symput("jn_nbr",jn_nbr);
    end;

    run;
```

Throughout the DATA step above, the variables SEL_NBR (created in the NOBS= option), TBL_NBR, and JN_NBR are used. TBL_NBR, and JN_NBR serve two purposes. One is to uniquely assign table names and join conditions to macro variables. _N_ serves the same purpose for assigning field names. The second purpose, served by all three variables, is to set an upper limit on %DO loops that will be used to generate the SQL. Once the last observation of SUBSET is read, CALL SYMPUT is used to assign these upper limits to macro variables.

GENERATING THE SQL
Returning now to the model query you started with,

```
  select &SLCT
  from &FRM
  where &JN
```

armed with unique macro variables that represent field names, table names, and join conditions, as well upper limits on how many of each of these are needed, you are now ready to replace &SLCT, &FRM, and &JN with %DO loops of the following form:

```
  %do i=1 %to &upperlimit ;
  %if &i^=&upperlimit %then &&X&I , ;
  %else &&X&I ;
  %end ;
```

X is either S, F, or J, and *upperlimit* is either sel_nbr, tbl_nbr, or jn_nbr, depending on which clause the loop is used in. In the SELECT clause, &&S&I resolves to the ith field name, and &sel_nbr is the number of fields. In the FROM clause, &&F&I is the ith table, and &tbl_nbr is the number of tables that contain the requested database fields. In the WHERE clause, &&J&I is the ith JOIN condition, and &jn_nbr is the number of JOIN conditions needed.

Two points are worthy of noting here. One, suppose your SELECT= parameter contains fields from more than one table, but none from TRANSACTIONS. The nature of the star database dictates that these two tables can only be joined through TRANSACTIONS. Unfortunately, SUBSET will not contain this table name as a value of TABLES. Therefore, logic must be used to force this table into the FROM clause if it is not there already.

```
  %let tran=N;
  %do i=1 %to &tbl_nbr;
    %if &i^=&tbl_nbr %then sugi28.&&F&i,;
    %else sugi28.&&F&i;
    %if &tran=N and &&F&i=TRANSACTIONS %then %let
tran=Y;
  %end;
  %if &tran=N and &tbl_nbr>1 %then ,sugi28.TRANSACTIONS ;
```

With this technique, a flag (&tran) is created and turned on once TRANSACTIONS is found to be a table. If the flag is never turned on and TRANSACTIONS is needed, then it is forced into the FROM clause.

Second, joins are not necessary if only one table is being used. Therefore, make the WHERE clause conditional on at least two tables being needed. &TBL_NBR can be used for this.

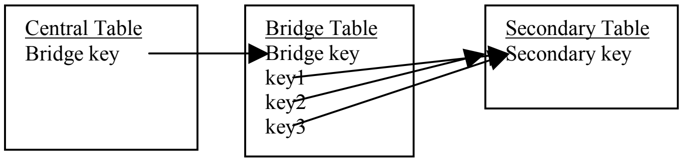
ADVANCED DATABASES

The above example demonstrated the use of metadata tables that were built based on a relatively simple database structure with only four tables. This led to relatively simple meta-code, or the code used against the metadata tables to generate an SQL query against the database. Though this may serve as a model for bigger databases, the nature of the data may require some twists in what otherwise may be a simple structure. This in turn may require some twists in the setup of the metadata tables and hence, some twists in the meta-code. Such is the case at our health care organization.

In a basic star schema, a central table has a connection to each of the other tables. However, in some of the stars, we have tables that don't link to the central table. I'll refer to these as secondary tables. Such a table can fall into one of two categories.

SECONDARY/TERTIARY CODES
In one category, the secondary table is linked to one other table referred to as a bridge, and the bridge is linked to the central table. Occasionally, the same set of codes is needed to populate multiple fields. Such is the case when a doctor makes a primary diagnosis and a secondary diagnosis, and maybe more, using diagnosis codes. As is the case with a basic star, descriptive information for these codes is kept in a separate table. At this point, you may think that each of the diagnoses (say three for the sake of discussion) will have its own field in the central table, and each can be used as a key to the descriptive table. However, as soon as any of these combinations is repeated on a separate record in the central table, space is wasted. Therefore, as an added efficiency, the combination of the three diagnoses can be represented in the central table with one field. A separate table is then created with a link to the central table by this field. The new table, otherwise called the bridge, also contains a field for each diagnosis, and each combination is unique. Each of these three diagnosis fields can then be joined to the secondary table to get descriptive information (See Figure 2).

Figure 2: A Bridge structure in a star schema



Note that BRIDGE KEY is a foreign key in the central table, and a primary key of the bridge. PRIM_KEY, FOR_KEY, and KEY3 are foreign keys of the bridge, linking to a primary key of the secondary table.
Think of the path from the central table to the secondary table through PRIM_KEY as a primary code, the path through FOR_KEY as a secondary code, etc.

ZIP CODES
The second type of secondary table is one that may link to multiple tables, each of which links to the central table. This is well illustrated with zip codes. Geographical information such as zip codes can pertain to anyone. Of course, your star structure may already have different tables for different kinds of people. For example, our database has a table for doctors and a separate table for members. Though much of what is in these tables is relevant only to a doctor or only to a member, zip code information may be relevant for both. Since both may be subject to the same zip codes, one zip code table can link to the member table and the doctor table.

RENAMING
In each of these cases and others, the presence of multiple fields that use the same variable name has forced us into using aliases for table names, and developing a re-naming convention for the

field names that are used for multiple fields. After all, if the user asks for zip code, then do they want the doctor's zip code or the member's. The combination of all these twists has led to the development of three metadata tables and meta-code that combines joins between all pairs of metadata tables. The first table maps a rename to its corresponding database name with a table alias. Now, &&S&I may resolve to *alias.database_field as rename.* The second table contains JOIN information between pairs of tables that don't include the central table (e.g. secondary table to a bridge). The third table contains JOIN information between the central table and those directly connected to it. With techniques similar to those mentioned above, SUBSET data sets are created and CALL SYMPUTS are used to create macro variables that are used to generate the SQL.

OTHER CONSIDERATIONS
This paper has focused on getting a FROM clause and JOIN conditions from your SELECT variables. However, an SQL-generator should allow you access to the other basic clauses of an SQL query.

- GROUP BY – the ability to summarize data rather than extract all of the detail
- summary functions in your SELECT that summarize by the GROUP BY variables
- ORDER BY – the ability to sort the data
- WHERE – the ability to filter criteria
- parameters that help further define the business focus to be added to the WHERE clause
- parameters that make use of efficiencies of the database (e.g. indexes) to be added to the WHERE clause

With efficient use of metadata and parameters that address database as well as business needs, SQL can be generated that runs efficiently and with relative ease to the user.

CONCLUSION

We come from a health care organization that maintains a small SAS presence. However, over 90% of health care organizations in this country are more significantly invested in SAS. As these companies grow and expand their customer base and their personnel, databases may grow larger and more complex. The programming and database knowledge Enterprise Guide requires of users may be more than what can be expected of people without programming experience. Investment in another vendor may be necessary at this point. In this paper, we have demonstrated that this may not be necessary. We have demonstrated that with a good set of metadata tables, relatively simple macro code can be written to automate the process of generating an SQL query. We have seen it with a hypothetical database in this paper. We have made it work for a complex star schema that features many twists in our health care data environment, and we believe it can be applied to other database structures as well. We also believe that in conjunction with techniques such as these, Enterprise Guide can compete better with other easy-to-use tools.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact authors at

Mike Molter
Blue Cross and Blue Shield of Rochester
165 Court Street
Rochester, NY 14647
(585) 238-4272
mike.molter@excellus.com

Scott Millard
Blue Cross and Blue Shield of Utica-Watertown
(315) 798-4314
scott.millard@excellus.com