They SQL Here They SQL There

Ian Amaranayake & David Shannon
Amadeus Software Ltd.

Abstract

The SQL procedure has come to play a significant role in many of the SAS®9 solutions. Not only is it still a language within a language for the Base SAS programmer, but it is a key procedure in tools such as SAS Data Integration Studio, Enterprise Guide and behind the scenes of Information Maps.

A brief history of the SQL origins will be presented to understand why it is used and understood across several software packages and languages.

Common uses of SQL and how it is set apart from the data step will be shown, including consideration to the new joins and set notations added in the last major release of SAS. A presentation of some SQL utilities will be given that are invaluable to a SAS programmer aiming to make frequent reuse of their code. Finally the internal workings of SQL will be discussed including some little known options that visualise what processing Proc SQL does internally, hence helping programmers to write more efficient code.

This paper will be of benefit to any programmer of Base SAS who wishes to gain exposure to SAS SQL uses and techniques.

Introduction

Proc SQL is nothing new to SAS. It was introduced to Base SAS several major releases ago and has been a key tool to many SAS programmers since that time. Similarly, there many SAS programmers who rarely venture into coding with Proc SQL, preferring to use the data step and standard procedures.

So why did we write this paper now? Well the honest answer is we were asked to! However, we did suggest this topic and with good reason.

Since the release of SAS 9, all its code generating interfaces such as SAS Data Integration Studio (formally ETL Studio) and Enterprise Guide make a large use of Proc SQL. Additionally Proc SQL now has the ability to *tune* itself, with the ability to select the type of lookup algorithm is deems most appropriate to the volume and structure of data queried.

It is time for a closer look at SQL.

A Wider View of SQL and a Little History

Structured Query Language (pronounced as either the initials S.Q.L. or *sequel* with identical meaning) is not specific to SAS. The language is actually an ANSI standard adopted and extended by many RDBMS vendors.

The Structured Query Language was developed by Chamberlin and Ray Boyce in 1973, but this was because Ted Codd (an Oxford University graduate who moved to

the USA to work for IBM) published his paper on the theory of relational tables and developed System R.

Relational databases from virtually any vendor can trace its roots back to this system, with their adoption of SQL; for example an Oracle DBA uses PL-SQL, Microsoft SQL Server developers write T-SQL and SAS users can use Proc SQL.

Why is SQL Commonly the Generated Code Language?

Understanding that SQL is used by several software vendors to drive their databases and their implementations are all derived from the same standard. The enables users of one software system to quickly learn the skills needed to exploit another.

SQL is also a well language of few statements and key words when compared to the SAS data step. What SQL looses in flexibility (although expert SQL programmers may dispute that statement) it makes up for in its ability to conform to code structure standards. It is considerably more robust to use such languages in tools such as Enterprise Guide when automatically generating code.

SAS have adopted SQL across their many code generating interfaces to good effect, promoting consistency and manageability across their solutions and exposing the power of SAS software to a much wider audience of users who will be instantly comfortable and familiar with the language and syntax used.

Typical Uses of SQL

SQL has a wide variety of applications that are worthy of mentioning. Even the most devout of Base SAS programmers who stick rigidly to the confines of the 'Data step' and 'Proc step' will surely admit to occasionally straying into the territory of writing 'Queries'. Queries are fundamental to SQL language, providing the capability to extract data from one or more source tables and derive new information and influence the structure of the results returned all within a single instruction.

Perhaps the main frustration that many of those SAS Programmers will share is the seemingly back to front nature of writing SQL syntax; there is something not quite right about defining the columns required prior to even mentioning the tables which source them. The benefits to be gained once the initial hurdle with syntax has been overcome, however, are considerable.

A simple query, for example, which is used to extract columns from a table can precisely control the order in which columns are created in the output table, write the same column with both formatted and unformatted values and order the results, all within a single step.

```
proc sql;
  create table demog as
  select studyid,
         siteid,
         siteid as site_name format=site.,
         subjid,
         age,
         sex,
         weight
  from fmts.demography
  order by studyid, siteid, subjid;
```

One of SQL's key strengths is the flexibility afforded by numerous types of outer and inner joins.

Consider the use of the Natural join, for example, which dynamically identifies the columns in common between the tables being joined, and providing they are of the same type, will perform the selected join.

```
proc sql;
  create table haem as
  select h.*, l.normalrangelower, l.normalrangeupper
  from lab_h as h natural left join lab_ranges as l;
quit;
```

Cross joins, which along with Natural joins are new in SAS 9, provide the ability to perform a Cartesian product on both of the tables specified in the join. This can prove particularly useful when creating a template structure of a data set which is then used in an Outer join.

Consider the following example which builds up a template data set for quality of life score data comprising one record per question per subject in our trial. Joining the actual quality of life data from our clinical trial afterwards, using an Outer join, allows us to retain the original structure defined by the template.

```
proc sql;
  create table allscores as
  select d.subjid, q.*
  from fmts.demography as d cross join fmts.qol_template as q;

  create table actualscores as
  select *
  from allscores as a left join fmts.actual_qol_data as q
  where a.subjid=q.subjid and a.question=q.question;
quit;
```

One-to-one, one-to-many and many-to-many joins are all supported within SQL as is the ability to join tables on different columns names and using different operators.

Consider the following code which joins only the appropriate lab normal ranges to our Haematology data, as compared to its 'Data step' counterpart.

```
proc sql;
  create table haem as
  select h.*, l.normalrangelower, l.normalrangeupper
  from lab_h as h left join lab_ranges as l
  on h.labtest=l.labtest;
quit;
```

```
data haem;
  merge  lab_h(IN=h)
         lab_ranges(IN=l keep=labtest normalrangelower normalrangeupper);
  by labtest;
  if h;
run;
```

From an efficiency perspective one should note the SQL version takes fewer lines of code, is performed in a single instruction, and when working with reasonably large data will out perform the traditional Base SAS technique, particularly if the data sources being joined are not indexed or ordered and require sorting. One could also argue that the code in the SQL version is, if anything, more verbose than its 'Data step' counterpart.

This technique of using Outer joins can also be used to detect which records in one data set do not exist in another. Consider the next example which uses a Left join to identify which subjects did not receive treatment during a study.

```
proc sql;
  create table notreat as
  select d.*
  from fmts.demography as d left join fmts.treat as t
  on d.subjid=t.subjid
  where t.subjid = .;
quit;
```

One of the most common uses of SQL is to interact directly with a DBMS using a technique known as SQL pass-through. This provides the user with the advantage of querying a native database directly, using either SAS or DBMS specific SQL, without having to leave the current SAS session.

```
proc sql ;
 connect to ORACLE (user=&userid orapw=&pwd path=&path);
 create table accounts as
  select
     subjid   'Subject ID',
     labtest  'Lab Test',
     labresul 'Lab Result',
     labunit  'Lab Test Unit',
     labnrlu  'Lab Normal Range Upper',
     labnrll  'Lab Normal Range Lower'
  from connection to oracle
   (select subjid,
           labtest,
           labresul,
           labunit,
           labnrlu,
           labnrll
    from labs
    order by subjid, labtest);
 disconnect from oracle ;
quit ;
```

Inner queries in this instance can exploit the full power of the DBMS specific SQL implementation whilst also making use of SAS specific SQL to create a table in the outer query.

The SAS implementation of SQL also provides seamless integration with different elements of SAS language. Queries can make use of SAS functions and operators, as well as SQL specific operators, when constructing expressions. Global statements

such as titles, footnotes and options are all honoured when creating reports. Equally when creating tables or views, columns attributes such as lengths, formats and labels can also be specified.

```
proc sql;
  create table demog as
  select substr(usubjid,4) as subject length=4 'Subject ID',
         age,
         gender format=$gender.,
         height                    'Height in Metres',
         weight                    'Weight in Kilograms',
         weight/(height**2) as bmi  'Body Mass Index'
   from demography;
quit;
```

Proc SQL also interfaces with SAS Macro language, providing the capability to create macro variables as a result of a query. Consider the following typical example of SQL syntax which creates a macro variable containing the result of a COUNT DISTINCT function.

```
proc sql noprint;
  select count(distinct subjid) into: nobs
  from fmts.bloodpressure;
quit;
%put Number of Subjects: &nobs;
```

```
4    proc sql noprint;
5      select count(distinct subjid) into: nobs
6      from fmts.bloodpressure;
7    quit;
NOTE: PROCEDURE SQL used (Total process time):
      real time           0.01 seconds
      cpu time            0.01 seconds

8    %put Number of Subjects: &nobs;
Number of Subjects: 60
```

This technique has the advantage of being both more concise and more efficient than an equivalent technique which may be implemented using typical Base SAS code.

It is also possible to create a macro variable which contains a series of values resulting from a query, with a delimiter to separate each individual value.

```
proc sql;
  select distinct subjid into: subject_list separated by ' '
  from fmts.bloodpressure;
quit;
```

```
4    proc sql noprint;
5      select distinct subjid into: subject_list separated by ' '
6      from fmts.bloodpressure;
7    quit;
NOTE: PROCEDURE SQL used (Total process time):
      real time           0.03 seconds
      cpu time            0.01 seconds

8    %put &subject_list;
001 002 003 004 005 006 007 008 009 010 011 012 013 014 015 016 017 018
019 020 021 022 023 024 025 026 027 028 029 030 031 032 033 034 035 036
037 038 039 040 041 042 043 044 045 046 047 048 049 050 051 052 053 054
```

Further Applications of Proc SQL

There are a number of additional uses of SQL that should also be considered which go beyond the more common applications that have been discussed thus far.

When considering the macro interface to SAS, for example, it is possible to create a range of macro variables which adopt the same naming convention based on the result of a query.

Again, when it comes to considering efficiency and ease of coding, one should consider that the following code only requires two complete statements to determine the number of distinct subjects and then use that value to create the precise number of macro variables required. To code the equivalent process in Base SAS would yet again require additional steps, if only for the fact that the first macro variable would not be available for resolution in the same data step that attempts to use it.

```
proc sql noprint;
  select count(distinct subjid) into: nobs
  from fmts.bloodpressure;

  select avg(supsys) into :SUBJ1 - :SUBJ%cmpres(&nobs)
  from fmts.bloodpressure
  group by subjid;
quit;
```

One very powerful feature of SQL is the capability to reuse the value of a column several observations ago when deriving a new column. In the following example, the average supine systolic blood pressure at baseline is subtracted from the average calculated for each visit date to determine the change from baseline value.

```
proc sql;
  create table change_baseline as
  select day,
         avg(supsys) as mean_supsys,
         calculated mean_supsys - (select avg(supsys)
                                   from fmts.bloodpressure
                                   where day=1) as change_baseline
  from fmts.bloodpressure
  group by day;
quit;
```

In some situations, Proc SQL will actually re-merge the results of an aggregate function back onto the original data. In the following program, the average baseline measurement is merged back onto each subjects record in order to calculate a new change from baseline value.

```
proc sql;
  create table change_baseline as
  select a.subjid,day,
         supsys,
         (select supsys
          from fmts.bloodpressure as b
          where day=1 and a.subjid=b.subjid) as baseline,
          supsys - calculated baseline as change_baseline,
          avg(calculated baseline) as average_baseline,
          supsys - calculated average_baseline as change_baseline_average
  from fmts.bloodpressure as a
  order by subjid, day;
quit;
```

SQL also provides access to a series of read only views called 'DICTIONARY tables' which contain information about the current SAS session. These dictionary tables can only be accessed using Proc SQL and can be used to efficiently automate key processes.

The full list of available dictionary tables along with descriptions on what they actually contain is shown in the table below.

Dictionary Table	Table Contents	Available from SAS Version
CATALOGS	Information on SAS catalogs available to the current SAS session	V8
CHECK_CONSTRAINTS	Information on check constraints defined on all SAS tables	V9
COLUMNS	Information on all columns from all tables in all libraries in the current SAS session	V8
CONSTRAINT_COLUMN_USAGE	Information on all column constraints defined	V9
CONSTRAINT_TABLE_USAGE	Information on all table constraints defined	V9
DICTIONARIES	Information on all dictionaries and columns within SQL dictionary tables	V9
ENGINES	Information on all SAS library engines available	V9
EXTFILES	Information on all external file references	V8
FORMATS	Information on all available formats to the current session, including SAS supplied formats	V9
GOPTIONS	Information on all SAS graphic options and their values	V9
INDEXES	Information on all SAS Indexes defined	V8

Dictionary Table	Table Contents	Available from SAS Version
LIBNAMES	Information on all SAS libraries defined	V9
MACROS	Information on all SAS macros variables and their values	V8
MEMBERS	Information on all catalogs, tables and views	V8
OPTIONS	Information on SAS global system options and their values	V8
REFERENTIAL_CONSTRAINTS	Information on referential constraints	V9
REMEMBER	Remembered (cached) information	V9
STYLES	Information on all available ODS Styles	V9
TABLES	Table information from all libraries	V8
TABLE_CONSTRAINTS	Table constraints	V9
TITLES	The active titles and footnotes	V8
VIEWS	List of available VIEWS in the current session.	V8

One of the key benefits of using dictionary tables is that they are always dynamically generated at the point where the table is actually queried. The following code demonstrates how, for example, we can retrieve all of the titles and footnotes currently in effect in our SAS session and return the results in a table. If titles or footnotes are subsequently changed and the Proc SQL step re-submitted, the results returned will reflect the new titles and footnotes that have been defined.

```
proc sql;
  create table titlesineffect as
  select *
  from dictionary.titles;
```

The ‘Columns’ dictionary table can be particularly useful when attempting to retrieve all of the columns which have a specific naming convention and can be used across any or all of the tables and libraries that are visible to the current SAS session.

```
proc sql;
  create table allVALs as
  select *
  from dictionary.columns
  where upcase(name) like "VAL%";
quit;
```

Standard programs, macros and applications can be developed which make use of the ability to dynamically determine which libraries are in existence, or the values of specific options and goptions.

Consider the following example which lists all of the available libraries and then determines the physical location of the WORK library. This information could then be

used to create additional files (other than SAS library members) which are only required to exist for the duration of the current SAS session.

```
proc sql;
  select distinct libname
  from dictionary.libnames;

  select path into:work_path
  from dictionary.libnames
  where libname='WORK';
quit;
%put &work_path;
```

The ‘Tables’ dictionary table can also be used to good effect, to dynamically process specific tables that have been created. Consider the following program which creates a macro variable containing a list of tables that adhere to a specific naming convention and then uses this information to append the tables together.

```
proc sql noprint;
  select distinct cat(compress(libname),'.',compress(memname))
  into:table_list separated by ' '
  from dictionary.tables
  where libname in ('WORK','DATA_IN') and
        upcase(scan(memname,1,'_'))='LAB';
quit;
%put &table_list;

data all;
  set &table_list;
run;
```

It should also be noted that most of the dictionary tables that exist can also be viewed via a series of SAS views which can be found in the SASHELP library. For example, the VMACRO SAS view allows us to dynamically access any of the variables currently held in a global or local symbol table in the same manner as the ‘Macros’ dictionary table. The advantage of using one of the available SAS views is that they can be accessed within both Data steps and Proc steps as well as Proc SQL.

However, there are efficiencies involved in using the dictionary tables which relate to the way in which Proc SQL dynamically retrieves the information about the current state of your SAS session. Whenever a dictionary table is queried, Proc SQL optimises the query prior to any selection being performed. This makes the use of dictionary tables faster and more efficient than performing the equivalent operation within a data step or proc step using a SASHELP view. Consider the following code for example which aims to create a table containing all of the columns in all of the tables in our FMTS library.

```
proc sql noprint;
  create table allcols1 as
  select distinct memname, name
  from dictionary.columns
  where libname='FMTS'
  order by memname;
quit;

proc sort data=sashelp.vcolumn(keep=libname memname name
                               where=(libname='FMTS'))
          out=allcols2(drop=libname);
  by memname;
run;
```

Lookup used	Real Time (seconds)	CPU Time (seconds)
Dictionary Table	0.01	0.01
SAS View	0.25	0.24

Comparing the performance times that appear in the log we find that the Proc SQL step takes less time than the Base SAS procedure, even when processing a reasonably small volume of data.

A Look under the Covers of SQL

When submitted the SQL procedure attempts to optimise the written code. Whilst this is an extremely powerful feature, it is not a reason for disregarding good coding practices as the user can very much influence the efficiency of the query by the logic written. Good training and studying of materials are essential to achieve this.

To understand which execution algorithm Proc SQL is using, append the _method option onto the proc statement and examine the log after submission.

```
proc sql _method;
  select *
  from sashelp.class;
quit;
```

The log produced is shown below:

```
1    proc sql _method;
2      select *
3      from sashelp.class;

NOTE: SQL execution methods chosen are:

      sqxslct
          sqxsrc( SASHELP.CLASS )
4    quit;
NOTE: PROCEDURE SQL used (Total process time):
      real time           1.20 seconds
      cpu time            0.03 seconds
```

This shows that rows are selected (sqxsrc) from the select clause (sqxslct). The full list of abbreviations and meanings are shown in the table below:

Execution Algorithm	Description

Execution Algorithm	Description
sqxcrta	Create table as select
sqxslct	Select
sqxjsl	Step loop join (Cartesian)
sqxjm	Merge Join
sqxindx	Index Join
sqxhash	Hash Join
sqxsort	Sort
sqxsrc	Source rows from table
sqxfil	Filter rows
sqxsumg	Summary stats with group by
sqxsumm	Summary stats with NO group by

A further, perhaps more advanced method of understanding how proc SQL is executing a query, is the _tree option. Again, append this to the proc SQL statement and view the log after submission.

There is not enough room to present a detailed discussion of the output produced within this paper. For readers interested in optimising the SQL code we refer the reader to the paper “The SQL Optimizer Project: _Method and _Tree in SAS®9.1” by Russ Lavery.

Conclusions

Along side the standard 4GL programming language that SAS gives us is Proc SQL.

To the glancing eye this procedure consists of a different style of syntax that can possibly be achieved in the data step. Look at little closer and there is a well formed, universally understood and efficient language for handling data.

SAS Institute are using SQL behind the business face of many SAS 9 data transforms, enabling a much wider audience to understand the code generated.

References

“SAS SQL”, Elena Muriel, Amadeus Software Ltd.

“Inside PROC SQL's Query Optimizer”, Paul Kent, SAS Institute.

“The SQL Optimizer Project: _Method and _Tree in SAS®9.1”, Russ Lavery, SUGI 30.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Author Name:	Ian Amaranayake & David Shannon
Company:	Amadeus Software Limited
Address:	The Old School Hall, 11 Wesley Walk, Witney, Oxon OX28 6ZJ
Work Phone:	+44 (0) 1993 848010
Email:	ian.amaranayake@amadeus.co.uk or david.shannon@amadeus.co.uk
Web:	www.amadeus.co.uk

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.