

A Taste of SQL

Paul Kent
SAS Institute Inc.
<kent@sas.com>

Abstract

This paper provides an overview of Structured Query Language (SQL), and its implementation in Version 6 of the SAS System. I have tried to keep it focused on items where SQL is especially helpful to a programmer that already knows the "traditional SAS" way of doing things.

The History of SQL

The "Relational Data Model", proposed by Codd (CODD70) represents data in tables. The SAS data set concept blends very nicely with the concept of a table in the relational data model. Both have columns(variables) and rows(observations). SAS data sets are a little more liberal than true relational model tables -- they allow duplicate rows, and have an inherent ordering. Nevertheless, there are strong enough parallels between the two to make SQL a useful language for accessing SAS data sets. The terms database table and SAS data set are interchangeable in the context of this paper.

SQL is a language for accessing and manipulating data stored in tables. SQL is an acronym for "Structured Query Language" and is pronounced "ess-que-ell" or "sea-quel".

Structured Query Language

Implementations of SQL usually have two components for manipulating the data stored in the data base.

- A set-at-a-time non-procedural component, allowing a user to query and modify database tables. This paper is concerned with this component, and its implementation in the SAS System.
- A record-at-a-time procedural component, that is usually embedded into third generation programming languages. Support for Embedded SQL may be added in future versions of the SAS System.

Advantages of using SQL

SQL is a non-procedural language. An advantage of this is that the user does not have to concern himself with the details of actually processing the request. In short one gets to say **WHAT** they want, and allow the application program to resolve the itty-gritty details of **HOW** to get

the results. In addition SQL syntax is "English-like" promoting quick learning of its constructs.

SQL has been touted as providing data independence. This is not as big a selling point to SAS folks -- a SAS data set has always protected applications from changes to the underlying files and most carefully written programs are immune to new variables in the data sets they process.

SQL has been implemented by many people, on many hardware platforms. Many new database solutions offer a form of SQL, and existing vendors are retro fitting relational query capabilities to their products. Distributed database systems are becoming viable -- there are even products available that connect heterogeneous databases using SQL as the common thread. By the volume of SQL articles in the popular computing press, many people are working on products that support it.

Advantages of SQL for a SAS user

PROC SQL provides an alternative to existing SAS solutions to many data processing problems. The key advantage of SQL over traditional SAS solutions stem from the non procedural language -- SQL solutions do not require lots of procedural framework.

PROC SQL solutions are "self optimising". SQL will take advantage of indexes and inherent sort order in your data sets. If you [or more likely someone else after you have moved on to bigger challenges] add a useful index, or sort a key data set at a later point in time, all your SQL programs will take advantage of it. Most of your procedural programs would need re-writing to take advantage of this new optimisation opportunity -- something even the best intentioned programmer "forgets" to get around to.

PROC SQL will do the moral equivalent of DROP= and KEEP= programming at the earliest opportunity - something even the most accomplished SAS programmer forgets to do from time to time.

PROC SQL allows more direct communication to other SQL data bases via SAS/ACCESS Software. The SQL Passthru facility is often the most efficient way to extract information from an external DBMS.

PROC SQL may allow you to transfer the applications logic from some database application directly into SAS. This is useful if you were already using SAS for the reporting phases of the application.

SQL Statements

There are six main statements in the non-procedural component of SQL.

- **SELECT** to retrieve values from database tables.
- **INSERT** to insert rows into database tables.
- **DELETE** to delete rows from database tables.
- **UPDATE** to modify the values of rows in database tables.
- **CREATE** to create database tables, views and indexes.
- **DROP** to remove database tables, views and indexes.

It is my opinion that the bulk of SQL's value to the SAS programmer lies in the expressive power of the **SELECT** statement. The other statements are supported by PROC SQL for completeness, but **SELECT** is where the payback lurks!

The **SELECT** statement

The **SELECT** statement is used to query a table, or a collection of related tables.

The keyword **SELECT** introduces the object clause and lists the variables that you desire. The asterisk (*) is shorthand for requesting all the variables.

The keyword **FROM** introduces the table(s) that you are interested in.

The **WHERE** clause of the **SELECT** statement is used to specify which rows of a table that you want to process. If you are familiar with "Standard SQL", you will notice that we have added support for comfortable SAS-isms like the datetime constants and all the SAS data step functions.

```
select author, section, time
  from paper
 where time > '12:00't;
```

AUTHOR	SECTION	TIME
Marti	Graphics	14:30
Lewis	Info Sys	15:30
Jane	Graphics	16:15

In this example, SQL provides no added functionality over traditional SAS tools. However, SQL permits arbitrary expressions where variables might be specified. Suppose

that the conference convenors decide to delay the papers for 30 minutes, and wish to display the new paper times.

A single SQL statement achieves the same result that would have required three SAS steps. (A **DATA STEP** to create the new variable, then a **PROC SORT** and finally a **PROC PRINT**).

```
select author, section, title,
       time + '0:30't
       as newtime format=time5.
  from paper
 order by section, time;
```

AUTHOR	SECTION	TITLE	..	NEWTIME
Marti	Graphics	Multi-	..	15:00
Jane	Graphics	Making	..	16:45
Paul	Info Sys	Query	..	11:00
Lewis	Info Sys	Query	..	16:00
Tom	Testing	Automa	..	9:30
Jim	Users	Starti	..	11:45

You can use all the functions available to the **DATA STEP** in SQL expressions (with the exception of the **LAGn** family whose semantics become a little muddled when combined with those of SQL)

SQL features for Summary Statistics

SQL provides summary (or aggregation) operators, not unlike the services provided by **PROC MEANS** and **PROC SUMMARY**. You can request any or all of the traditional statistics for the entire table, or on a per group basis:

```
select max(rating) as maxr,
       min(rating) as minr
  from question;
```

MAXR	MINR
5	2

A traditional usage of summary functions would be in conjunction with sales data. "Who are the best salesmen in each region, and how do they stack up against our absolute best salesman???"

PROC SQL implements the ability to reference the elementary data items as well as the summary statistics in the same expression. This process of remerging the statistics back together with the data that generated them is useful for answering questions like "Who earned the most in each division?" The traditional SAS solution for a problem like this would involve creating a summary data set with the maximum for each department, then merging that data set with the original data looking for records with the calculated maxima.

SQL HAVING clauses can be considered **WHERE** clauses for each group of a query involving summary statistics, and may reference both elementary data items as well as

summary functions. This feature is not available in many SQL implementations -- whose work around is similar to the traditional SAS solution -- create a table with the maximum values for each group, and merge those values back with the original data.

```
create table winners as
select region, name, sales
  from sales
 group by region, name
having sales=max(sales);

select region, name, sales
      100*sales/max(sales)
  from winners
 order by sales desc;
```

SQL does support a block structured syntax, so if you are adventurous you can collapse this example into a single query:

```
select region, name, sales
      100*sales/max(sales)
  from (
    select region, name, sales
      from sales
    group by region, name
    having sales=max(sales)
  )
 order by sales desc;
```

Queries against many Tables

So far, PROC SQL with its non procedural SQL syntax has provided some improvements over traditional procedural solutions to problems. But there is more! SQL deals with multiple input tables in an intuitive fashion -- the user is free to concentrate on the WHAT, while the system concerns itself with the HOW.

Extracting data from more than one table in SQL is quite simple. You can join many tables by listing more than one on the FROM clause of a query. If you want to achieve some kind of matching between the rows of the various tables, you specify this in the where clause. These row matching conditions are often called join predicates.

Let us use these sample data sets for some examples. The tables record information about People and their Pets, as well as information on who feeds whom.

```
select * from people;
```

NAME	SEX	FEEDTIME
Paul	M	Morning
Denise	F	.
Kelsey	F	Evening

```
select * from pets;
```

PET	TYPE
Sable	Dog
Kudu	Dog
Sumo	Cat
Momma	Cat

```
MatSumo  Cat
Fred      Frog

select * from feeds;
```

NAME	PET
Paul	Sable
Paul	Kudu
Paul	MatSumo
Kelsey	Sumo
Kelsey	Momma
Kelsey	MatSumo

The default join in SQL selects on those rows that have a "match" in the table being joined with - this is not always the desired result, but many early SQL implementations could only perform this type of join. For Example, the answer to "Who Feeds Whom, and When" is:

```
select people.name, feedtime, feeds.pet
  from people, feeds
 where people.name = feeds.name;
```

NAME	FEEDTIME	PET
Paul	Morning	Sable
Paul	Morning	Kudu
Paul	Morning	MatSumo
Kelsey	Evening	Sumo
Kelsey	Evening	Momma
Kelsey	Evening	MatSumo

Depending on your point of view, you may be alarmed that we have lost some information in the result. It may be important that Denise exists, even though she does not currently feed any of the pets.

The Outer Join syntax of SQL is used to retain rows even if they have no match in the other tables being joined with. The keywords LEFT, RIGHT and FULL used with JOIN (instead of the simple comma) on the FROM clause give you control over whether you want non-matched records from either or both of the tables being joined. One can think of ON clauses as "MAYBE-WHERE" clauses - match if you can, otherwise provide missing values for the rows for which no match is found.

```
select people.name, feedtime, feeds.pet
  from people LEFT JOIN feeds
    ON people.name = feeds.name;
```

NAME	FEEDTIME	PET
Denise	.	.
Paul	Morning	Sable
Paul	Morning	Kudu
Paul	Morning	MatSumo
Kelsey	Evening	Sumo
Kelsey	Evening	Momma
Kelsey	Evening	MatSumo

Some folks get stumped when they need to perform more than one outer join -- perhaps we need better examples in our documentation. The block structured nature of SQL means that the outer joins can be cascaded together, so to include the pets that are not currently fed we would also

need to perform an outer join with the pets table. Don't forget that the ON clause attaches to its respective JOIN clause -- different than a WHERE clause that comes after all the items on the FROM clause have been listed.

```
select people.name, feedtime, feeds.pet,
       pets.type
from people LEFT JOIN feeds
  ON people.name = feeds.name
  FULL JOIN pets
  ON feeds.pet = pets.pet;
```

NAME	FEEDTIME	PET	TYPE
Denise			
Paul	Morning	Kudu	Dog
Kelsey	Evening	MatSumo	Cat
Paul	Morning	MatSumo	Cat
Kelsey	Evening	Momma	Cat
Paul	Morning	Sable	Dog
Kelsey	Evening	Sumo	Cat

This query is almost correct ☺. We still lose the name of the pet for records which have no matches in the FEEDS table -- this is because we have asked for the column feeds.pet in our query. What we really want to get is the value from which ever table contributes thevalue:

```
select people.name, feedtime,
       coalesce(feeds.pet,pets.pet) as pet,
       pets.type
from people LEFT JOIN feeds
  ON people.name = feeds.name
  FULL JOIN pets
  ON feeds.pet = pets.pet;
```

A sometimes useful feature of SQL joins is that the match condition need not necessarily be an "equals" match. A question often posed on SAS-L goes like this: "I need to locate records in one file whose date is between the start and stop dates of events stored in another file"

```
select *
from file1, file2
where file1.date between file2.start
                  and file2.end;
```

New Features in PROC SQL

Since the initial release of PROC SQL we have extended the SQL procedure in these areas:

- **DICTIONARY TABLES**
- **Expanded INTO support**
- **Support for External DBMS SQL**

These items are documented in the Changes and Enhancements Technical Reports for the different Releases of the SAS System, as well as in the help screens for PROC SQL. We have also made performance improvements in subsequent versions and maintenance releases, but these should be picked up by your existing

SQL programs without you needing to know (or code) any new syntax.

DICTIONARY Tables

Dictionary tables are "pseudo-tables" that PROC SQL materialises on demand. They contain information about the context of the SAS execution. An example usage that computes the number of observations that fit in a buffer for all tables in the SASHELP library whose name begins with 'A' would be:

```
select memname, obslen, bufsize, nob,
       floor(bufsize/obslen as bufobs
from dictionary.tables
where libname = 'SASHELP'
and memname like 'A%';
```

Member Name	Observation Length	Bufsi..
ADBEEX	80	40..
ADBELOC	119	40..
ADDON	72	40..
ADXPARM	236	40..

Recently added dictionary tables include DICTIONARY.MACROS, DICTIONARY.TITLES and DICTIONARY.OPTIONS.

The Into Clause

The INTO clause transfers values from the results of the query into SAS Macro Variables. This can be quite useful if you are in the habit of "writing programs that write programs". A recent query on the SAS-L electronic forum went something like:

I have a data set with many variables of whose name match the form Xnnn. They are stored in the data set in a haphazard fashion, and I want to process them in a ordered fashion.

The problem at hand is to build an array statement for all the numeric variables Xnnn in lexicographic order (I.E. X1 should precede X2, but X10 should be after X9). Here is some SQL that helps.

```
proc sql noprint;
select name
into :names separated by ' '
from dictionary.columns
where libname='MYLIB' and memname='MYTAB'
order by input(substr(name,1), 8.0);
```

```
data ...;
array x &names;
```

The original into clause would capture the first row of the result only. The new syntax allows you to string all the values for a column together, or to create a new macro

variable for each row of the result set. Suppose you wanted to run the UNIVARIATE procedure on any data set in the SALES library that had a variable called TRANSACT.

```
%macro do_uni;
proc sql noprint;
select memname
into :mem1 - :mem9999
from dictionary.columns
where libname='SALES'
and name='TRANSACT';

%do I = 1 to &sqlobs;
proc univariate
data=SALES.&&mem&i;
run;
%end;
%mend;
```

External DBMS SQL

Folks at SUGI and the regional users groups, and others who send me email from time to time began pondering if we could provide them with a more direct interface to the native SQL of the Database Software they had paid so much for. We developed the pass through feature of PROC SQL after listening to many ideas on how this should be done.

The SQL passed to the DBMS returns a relation table as its result set, so it made sense that we should incorporate this new syntax as an alternate in the FROM clause. Data from the DBMS could be treated in a consistent fashion as data that arrived in the form of SAS data sets.

The CONNECT Statement is DBMS specific so I'll have to refer you to the SAS/Access documentation for that DBMS, but the remainder of the example should translate.

```
proc sql;
connect to db2;

select *
from connection to db2
(
select creator,
count(*),
avg(npages)
from sysibm.systables
group by creator
)
as T1(creator, count, avgpage)
where substr(creator,1,3) ne 'SYS';
```

The entire bold section above is a single element to the FROM clause. It is as if the query were sent to the DBMS and the results stored in a SAS data set called T1 with three columns named creator, count and avgpage. The variables can be processed further, as demonstrated in the where clause where we exclude any creator whose ID begins with SYS -- a silly example as we would have preferred to use where creator like 'SYS%' in the SQL

we sent to DB2 - it makes sense to filter out the unwanted records as early as possible.

References

(ANSI86) X3.135 "Database Language SQL".

(ANSI89) X3.135-92 "Database Language SQL2".

(CODD70) Codd, E.F. "A relational model of data for large shared data banks", CACM 13 #6, June 1970.

(DATE81) Date, C.J. "An Introduction to Database Systems, Volume 1", Addison-Wesley, 1981. ISBN 0-201-51381-1

"Database Programming & Design", Miller Freeman Publications, ISSN 0895-4518, a monthly publication.

"DBMS", M&T Publishing, ISSN 1041-5173, a monthly publication.

"SAS Users Group International Conference Proceedings", 1988 through 1995 have papers that reference SQL.