Paper SE05

Sublime Secrets of the SAS® SQLheads

Sigurd W. Hermansen, Westat, Rockville, Maryland

ABSTRACT

A clever application of pure science with a hint of magic, SAS SQL reduces solutions to many complex database programming problems to a few well-chosen phrases. In this look behind the scenes we go beyond the usual demonstrations of how SELECT and JOIN queries work and reveal the sublime secrets of SAS SQLheads. Did you know that true SQLheads organize datasets ahead of time to make SQL queries look simpler and more effective than SAS data step programs? Or that, under the right circumstances, SQL solutions rely on nothing more than grade school math? Shocking, yes, but it does not end there. SQLheads communicate with many different database systems using the ANSI SQL language. Find out why SAS data step experts are defecting to the SQLheads at an alarmingly high rate. Examples of SAS SQL in action show methods for identifying duplicate records, data filtering, linking SAS datasets on partial and inexact keys, data summaries, querying SAS metadata, and testing efficiency of queries.

INTRODUCTION

The SAS Institute introduced the PROC SQL implementation of the ANSI Structured Query Language (SQL) in 1989. Veteran SAS System DATA step programmers viewed this new take on SAS database programming with palpable apathy and distrust. A lunatic fringe, the 'SAS SQLheads' began campaigning for an immediate overthrow of the DATAsteppian rulers of the SAS user community. Aided by SAS Institute insider Paul Kent, the driving force behind PROC SQL, and others in SI with eyes on the burgeoning market for interfaces to relational database systems, the SAS SQLheads have evolved from a small lunatic fringe to a much more imposing lunatic fringe. While DATAsteppians continue to find new ways to program solutions to old problems, the SQLheads are reaching out to the larger community of SAS programmers. This brief introduction to the sublime secrets of the SAS SQLheads gives the reader an inside view of the aesthetics and discipline of this mysterious cult.

FIRST, DO NO HARM.

A SAS SQL select query of the form SELECT * FROM <SAS dataset1>,<SAS dataset2> ….; does not alter, rearrange, delete, or change attributes of SAS datasets or other sources of data. What harm can it do?

It may send huge volumes of data to display devices, disk, or memory. It may squander a system's CPU, storage, or network resources. Novice SQL programmers can (and often have) made themselves very unpopular with systems administrators by submitting a simple SQL program that brings an entire system to its knees. A few simple precautions will keep you off the system administrators' short list of troublemakers:

- *Always check to make sure that SQL queries have all appropriate conditions in place:*

SELECT * FROM <SAS dataset> WHERE <condition> does just as it reads. It selects only those rows (obs) that meet the condition specified in the WHERE clause. If, for example, a dataset has 100 rows, and each row has a distinct number between 1 and 100 (though not necessarily in any order), the SQL statement,

```
   SELECT n FROM <dataset> WHERE n BETWEEN 1 AND 25;
```

will select the column n in 25 rows where n ranges from 1 to 25. The Boolean condition n<=25 would produce the same results in this particular case. WHERE conditions limit the number of rows a query inputs from a table. WHERE conditions and related ON conditions also control links among datasets, as explained under **NORMALIZE DATABASES.**

- *Test the concept of a query with simple examples before pulling the trigger:*

The number of rows and columns of data does not matter that much. Whether operating on six rows or millions of rows of data, a SELECT query acts as if follows a few simple rules. More on this under **THINK BRUTE FORCE …**

- *Estimate the size of the yield of a query:*

A SAS SQL query selects data from one or more datasets and yields at most a single tabular data object. This makes it possible to estimate the burden it imposes on storage media or on display devices from the expected dimensions of the yield of the query. See **SIZE UP YOUR QUERIES** for more ideas.

KIS BNTS

An old military saying translates in polite language to 'Keep it simple' (KIS). An important qualification: 'But Not Too Simple' (BNTS). Typically programmers follow a divide and conquer strategy that splits a solution to a problem into simpler sub-problems. The idea of splitting a process into many simpler paths has an intuitive appeal, but the strategy begins to turn on itself when paths proliferate to a degree that the programmer begins to loose track of some of them. The old problem of loop control gives us a perfect example of how the divide and conquer strategy can go wrong. In a procedural language such as C, we might assign an initial value to a variable prior to a loop and increase the value of the variable within the loop. A condition for continuing the loop may appear at the beginning or end of the loop. To keep things simple for now, say we name the variable x, and we intend to have the loop continue through a specific range of the value of x:

```
Do; WHILE <condition>(?)> continue;
<statements>; x=x+abs(y); <statements>
UNTIL<condition>(?) repeat;
```

Now, if we decide to do something repeatedly with the value of x, the series of values of x now depends on

- Whether the loop control condition appears at the beginning or end of the loop. For example, if (not x>1099) apppears at the beginning of the loop, then x will not attain the value 1100. If the same statement appears at the end of the loop, it will;
- Whether the condition requires > or >=, < or <=, etc;
- Whether a statement preceding x=x+abs(y) or following changes the value of y.

The list could go on and on. Unexpected results of looping account for many of the more costly and frustrating programming errors

A SAS Data step simplifies loop control. Since the SET and MERGE statements loop by default through each row of a SAS dataset, conditions and assignments apply in turn to each row of data. The physical order of records in the file that contains the dataset determines the order of processing of rows of data. The Data step offers a different level of simplicity. The SAS compiler operates on columns of data, not individual values. This is simplicity of a second kind, a matter of discipline. All of the items in a column belong to the same data type and get treated the same. The programmer gives up item by item control for programming simplicity.

SQL takes programming discipline one step ahead in the direction of programming simplicity. No loops. No ordering of columns. No physical order of rows of data. Just select columns or expressions from tables where a condition holds. SQLheads revel in the abstract clarity and expressive power of a SQL query. Limiting data structures to tabular data objects, as illustrated in the next section, encourages good database design practices and optimizes SQL for database programming.

NORMALIZE DATABASES

Organizing data up front pays dividends. SQL derives from and works best in the context of relational database design. A relational database does not require Oracle or MS SQL Server or any other commercial relational database management system. Better logical organization of data does more for a database than moving it from one database system to another.

A few guidelines do wonders for database programmers:

- *Embed as much information as possible within data tables;*

A convincing example contrasts a typical flatfile design,

PersonsContacts

```
personID  sex birthDate          contactDate1 purpose1 outcome1
                                 contactDate2 purpose2,outcome2
                                 …..
```

with a standard relational design,

Persons	**Contacts**
personID sex birthDate	contactDate purpose outcome personID

A few repeating groups of column variables look great in a spreadsheet, but any large number off repeating groups of variables seriously tilts the ratio of column space to data. The maximum number of repeating groups of variables determines the number of columns required. Not only may many columns contain high proportions of missing data, the burden of managing many variable labels falls on the database programmer. Doubling the number of visits in this example doubles the number of variables in **PersonsContacts**.

- *Different datasets for different dimensions.*

Mixing dimensions leads to problems obvious to anyone who has collected multiple observations of sex and birthDate values. This structure of data,

Person_Contact

personID sex birthDate contactDate purpose outcome

leaves the database programmer liable to inconsistent reporting. The values of sex and birthDate for a given personID depend on which of the rows of data the programmer references.

So what if the database does have repeating groups of column variables or mixed dimensions? What's a database programmer to do?

- *Create appropriate views of data.*

A simple macroprogram creates a normalized view of **PersonsContacts** with __n repeating groups of variables:

```
%macro normalize(__n);          PROC SQL;
 CREATE VIEW PersonsVW AS SELECT personID,sex,birthdate
 FROM lib.PersonsContacts;
 CREATE VIEW ContactsVW AS
    %do __i=1 %to &__n.;
     SELECT personID,contactDate&__i. AS contactDate,
           purpose&__i. as purpose,outcome&__i. AS outcome
     FROM lib.PersonsContacts
                %if (&__i.<&__n.) %then %do; OUTER UNION CORR
%end;
    %end;
 ; QUIT;
%mend normalize;
```

Whenever required the two tables combine naturally into a single consistent relation among persons and contacts, The SELECT … ON … query,

```
CREATE TABLE personContact AS
SELECT t1.*,t2.sex,t2.birthDate
FROM ContactsVW AS t1 LEFT JOIN PersonsVW AS t2
ON t1.personID=t2.personID
WHERE <condition>;
```

Links the rows of data on the key value in each row. No special row ordering or pointers required!

Ordinary data values in the personID columns serve as 'keys' that link **Persons** and **Contacts**. In SQL, linking keys do not have to have the same name. Although they have to have the same data type, functions of column variables can also serve as keys, so one may convert values from, say, numeric to character type in an ON condition (for example, … ON put(t1.personID,z8.) = t2.personID to right set and fill with leading zeroes). A composite of many column variables spread throughout a row in a dataset may also serve as a linking key.

A database programmer can use the views ContactsVW and PersonVW as a virtual, normalized database. The views contain all of the information in the original flatfile database. The original flatfile database remains intact. Better organization of data and simple programs replaces convoluted programming around the defects of poorly designed databases.

THINK BRUTE FORCE

A SQL compiler optimizes queries. Though it does not always succeed in finding the most efficient solution for each specific query, the SAS SQL query does look at attributes of datasets, such as the SORTEDBY property, and at indexes. All this activity occurs in the background. The database programmer has little direct control over the query plan that the SAS SQL compiler devises. While this lack of control tends to frustrate DATAsteppians, it frees SAS SQLheads to focus on the solution to a programming problem and let the SQL compiler find the path to the solution.

Just as a Zen archer aligns bow and arrow to target, a database programmer arranges a SQL statement to fit a tabular solution to a query. Logic programmers call that 'declarative programming'. SQLheads know the forms of queries needed to produce specific solutions.

How do SQLheads know the forms of queries needed to declare a solution? They start with a small example and think *brute force*. The SQL compiler may improve on the brute force (least intelligent) method for executing queries, but in each and every case the improved method produces exactly the same solution as the painfully tedious brute force method. That means that the database programmer only has to understand the primitive brute force method, and that the programmer can verify the solution by systematically applying the brute force method to a minimally complete example of a database .

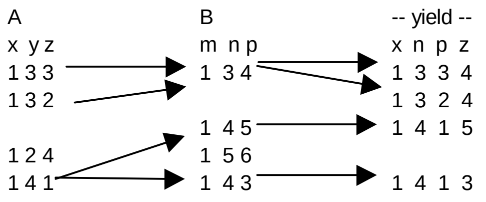
For instance, a SQL JOIN query that includes … A INNER JOIN B ON A.x=B.m AND B.n=A.y links

A	to	B
x y z		m n p
1 3 3		1 3 4
1 3 2		1 4 5
1 2 4		1 5 6
1 4 1		1 4 3

on the condition that the value in column x in table A (A.x) equals the value in column m in the table B (B.m) and the value in column y in A (A.y) equals the value in column n in B (B.n). The brute force method by default follows this algorithm (note the SQL convention that qualifies variable names with table names (<tablename>.<variablename>):

```
1) compare row in A to each row in B
   if the condition holds, SELECT column(s) from rows A,B
2) repeat above through last row in A.
```

Preceding the … JOIN … ON … clause in the query with SELECT A.x as x,B.n as n,A.z as p,B.p as z, the query yields:



A simple SELECT … FROM …<dataset> WHERE … query yields whatever list of column attributes the programmer specifies between the SELECT and FROM keywords.

The SAS System makes it easy to create minimally complete instances of data sources that a programmer needs to test SAS SQL queries. Neophytes work through brute force solutions as an aid to learning how to use SQL. True SQLheads use brute force methods of solution to prove the concept of a query and check to make sure that the SAS SQL query execution plan matches what the programmer expects.

ALL I NEED TO KNOW I LEARNED IN 6TH GRADE

At first SQL seems too limited to have much value to a real programmer. What good does it do to read columns from a file and subset rows? Any number of toy programming systems can do that.

The tieing together of SQL to the logical objects called 'relations', of linkage keys in databases to sets, and of conditions to Boolean logic means that a database programmer can represent any information as relations, and represent all operations that select, subset or combine information as set operators on key values.
Those ties connect SQL to a couple of thousand years of success with the first-order logic and set operations that digital computers handle so well. Over the years basic logic and set theory have infiltrated the 'new math' that students usually begin trying to master in the 6th grade. In the context of appropriate database design, SQL provides a good programming environment for most of what database programmers do.

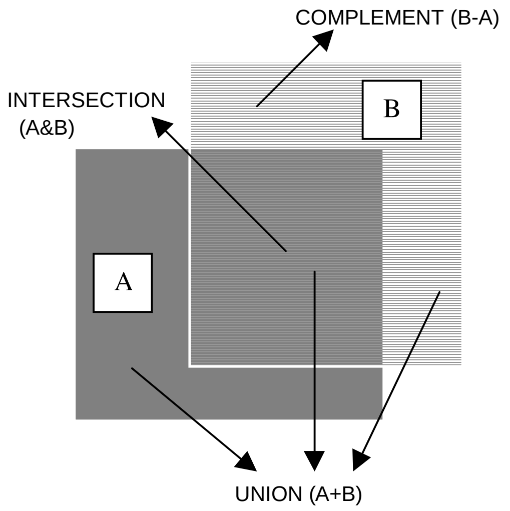
WHERE conditions handle very large numbers of simple Boolean True or False expressions of one or two values (say, x>=1) The NOT, AND, and OR operators make it possible to qualify simple expressions. SQL has all of the tools needed to filter and subset. SAS SQL gives the database programmer a rich assortment of operators, functions, and formats to use in WHERE conditions. Logical conditions may include, for example, functions that return 0 or another value. In SAS the 0 represents False and the other values True. A logical expression 'x=1 AND INDEX(txt,c) AND z>-3 would fail if the value of the variable txt does not contain the value of the variable c, since the INDEX() function returns 0 if the string value of the second argument does not appear in the string value of the first argument.

CASE … WHEN … clauses have much the same role as conditional assignments of values to variables (IF … THEN … ELSE assignments of a different value depending on the outcome of a condition in, say, a C, VB, or SAS DATA step program). A CASE clause substitutes for a column name in a SELECT list, so it must yield a value that will fit in a row-column cell of a table (or else generate a run-time error). Though the logic often looks backwards, it works the same as an IF <condition holds true> THEN … assignment:

```
… <select list>, CASE WHEN x=5 and y=8 THEN b
                                                ELSE 0
               END as z, <select list>
```

CASE WHEN clauses nest in much the way IF … THEN assignments nest.

SAS SQL lets us distinguish row subsetting logic (WHERE conditions) from alternative assignments to column variables (CASE WHEN … END), and both from conditions that control JOIN … ON's and UNION's of tabular data objects such as SAS datasets. If we constrain values in key columns of data in a tabular data object to a different and distinct value in each row, we make the key values a set. (When a key spans several columns of a table, we may think of the key as a distinct superstring that combines several column values.) For any sets A and B, the Venn diagrams that follow remind us how the key values in two tables relate to each other. The different shades represent the sets that result from the intersection, union, and complement operators (&, +, and -):



An intersection of two sets of key values produces a set that contains all of the values that appear in both sets. The union contains all of the distinct keys in A or B. The complement contains all of the keys in one set (B in this example) but not in the other (A).

Three set operations may not seem like much to work with, but intersection, union, and complement separately and combined give us all of the logical operations that we need to combine data tables representing relations, including MERGES, APPENDS, and table look-ups. For example,

Result wanted	Operation(s)	SQL clause
-Link rows IDs=	intersection	A INNER JOIN B ON …
-All rows of A,	complement,	
link to B Ids=	intersect,union	A LEFT JOIN B ON ….
- All rows A+B,	complement,	
link A&B Ids=	intersect,union	A FULL JOIN B ON ….
-Append A+B	union	A OUTER UNION B …

So far we have said nothing about the other column variables in the two tables. Since being on the same row relates the other values to the key, linkage on the key also links the other column values on rows related to the key:

other columns [a]←→ key[a] = key[b] ←→other columns[b]

In other words, the other variables in a row follow the keys.

NEVER HAVE TO SAY 'PROC SORT'

Putting data in a specific physical order introduces two major inefficiencies. First, sorting requires a lot of moving around of data within physical files. Second, putting data in one sort order for one purpose often arranges data in the wrong order for another purpose.

Sorting of linkage keys often turns out to be the most efficient way to limit the range of searches for matching key values. Nonetheless, the traditional DATAsteppian practice of sorting datasets on BY variables prior to MERGE'ing them often introduces gross inefficiencies into database programming. Consider this standard MERGE sequence:

```
PROC SORT DATA=large1 OUT=large1S;
  By timekey; ….
DATA joinedDS;
  MERGE large1S (in=in1) large2S (rename=(i=ii) in=in2);
    BY timekey;
      IF in1 AND in2 AND i<=10 AND ii>=999990; ….
```

If only a fraction of the rows in test have I<=10 and ii>=999990, SORT'ing the dataset test before subsetting it adds a substantial burden to the program. A close to equivalent SQL solution gives the SQL compiler latitude to subset the dataset test prior to indexing or rearranging its physical order. The subset sorts faster than the full dataset (usually by a factor of n(log(n)) for n rows of data). A contrived example illustrates a dramatic saving of processing time and effort due to optimizing a query that joins small subsets of rows in two datasets containing a million records each. Merely sorting one of the datasets takes about 11 seconds. The SQL query,

```
CREATE TABLE joined1 AS
SELECT  t1.*,t2.member_ID AS m,t2.time AS t,
t2.timekey AS tk, t2.type AS typ
FROM large1 AS t1 INNER JOIN large2 AS t2
ON t1.timekey=t2.timekey
WHERE t1.i between 1 and 10
     AND t2.i between 999990 and 1000000;
```

finds the solution in less than 2/3's of a second. It 'pushes down' the WHERE conditions and subsets the datasets prior to sorting.

To be fair to our DATAsteppian friends, applying WHERE conditions as PROC SORT streams data from the data sources reduces dramatically the time required to order small subsets prior to a MERGE, as in,

```
PROC SORT DATA=large1 (WHERE=(i<=100)) out=large1S;
  by timekey;
run;
```

(except that the MERGE will always produce results that vary with the physical order of variables not in the BY group where both of the datasets being MERGE'd contain the same BY value in more than one observation).

Also, caveat seems appropriate for the case of a 'disjunctive' (OR) in a WHERE condition. SAS SQL optimizes 'conjuctive' (AND) conditions at the expense of disjunctive conditions. As a result, it pays to subset data sources in in-line queries prior to joining them, as in:

```
…FROM (SELECT * FROM large1 WHERE i BETWEEN 1 AND
1000 OR i BETWEEN 999000 AND 1000000) AS t1
       INNER JOIN
     (SELECT * FROM large2 WHERE i BETWEEN 1 AND 1000
OR i BETWEEN 999000 AND 1000000) AS t2
  ON t1.timekey=t2.timekey; …
```

The sub-queries may improve query performance dramatically in this case. The sub-query filters cut down substantially on the perverse effects of multiples of key values occurring in both data sources.

SQL generally eliminates the problem of deciding when and how to subset and order data tables before combining them. SAS SQLheads prefer envisioning solutions to directing traffic from dataset to sorted dataset to MERGE BY.

STREAM WITH A VIEW

As illustrated in the last section, a programmer can substitute a SQL 'in-line view' for a dataset name in a FROM … clause. A SQL in-line view consists of an ordinary SELECT …FROM … statement enclosed in parentheses. One can think of a *view* as a program that produces the same result as reading data sequentially from a data source. In that sense a view produces a stream of data. It acts as a virtual dataset reference in a SQL query. Using views to stream data into a query, as opposed to writing results of queries to WORK datasets, reduces the burden on data storage devices such as disks. Hermansen (2002) describes how to nest queries in SAS SQL and how to determine whether or not a nested sub-query will yield the type of relation (table, column, or cell) required by its context in a SQL query.

BYE BY GROUP, HELLO GROUP BY

SQLheads don't attract many groupies. That may explain why the GROUP BY … HAVING … clause seems something of an afterthought. The GROUP BY …. HAVING clause in a SQL statement always follows the FROM clause in a SELECT query, and, if the query includes one, the WHERE condition. SQL programmers usually learn to use the GROUP clause effectively long after they gain a reasonably good understanding of the other components of a SELECT query. In fact, a programmer could avoid grouping data altogether by partitioning data on values of column variables before applying the same process to each partition. The GROUP BY … clause simply makes partitioning and processing each partition individually so convenient that database programmers find eventually that they have to learn how to take advantage of it.

One secret to understanding the SQL GROUP BY … clause: *it occurs near the end of a SELECT statement for a reason.* At that stage the data have begun streaming from data sources through, if included, a WHERE condition filter. The elements of the SELECT list have been evaluated. If the SELECT list includes an summary function, then and only then the SQL compiler needs to know how to partition a tabular data object into groups of rows. The SQL solution requires a summary value for each group (or to compute summary values for a HAVING clause). We might use the partial query,

```
… SELECT age,race,sex,city,COUNT(*) …
GROUP BY age,race,sex,city HAVING COUNT(*)>6 …
```

in an attempt to 'disclosure-proof' a class of attribute values by limiting reporting of classes containing fewer than seven subjects. In typical applications of GROUP BY clauses, the columns included in the SELECT list, other than summary values, match the GROUP BY list. In atypical applications, the SELECT list contains attributes not included in the GROUP BY list. The SAS SQL compiler allows this, but as a precaution it issues a warning message. The summary values correspond to the GROUP BY partitions, but the number of rows in the yield of the query may exceed the number of distinct classes of the grouping variables. SELECT'ing age, race, sex, and city but grouping by age, sex, and city only may yield a summary table that appears to overcount subjects. For these data,

age	race	sex	city
43	B	F	Duluth
34	W	M	Boise
43	W	F	Duluth
34	A	M	Boise

SAS SQL produces

age	race	sex	city	count
43	B	F	Duluth	2
34	W	M	Boise	2
43	W	F	Duluth	2
34	A	M	Boise	2

The sum of the counts obviously exceeds the number of subjects. The HAVING condition may then come into play. It could, for example, constrain the solution to the minimum value of race:

```
….GROUP BY age,sex,city HAVING race=min(race)….
```

The HAVING condition selects only the row with the minimum value of race in each group:

age	race	sex	city	count
43	B	F	Duluth	2
34	A	M	Boise	2

Of course that adjustment only works where only one row in each group has the extreme value. The undocumented MONOTONIC() function in SAS SQL adds a distinct sequence number to each row of data. The SAS-L archives contain a number of postings on the methods for using and problems associated with MONOTONIC().

FROM METADATA TO MACROVARIABLES

Relational databases do not differentiate queries of metadata (data about data) tables from queries of ordinary data (data just being data). SQL queries work equally well for either. SQLheads take advantage of this feature. Say one is looking for duplicate rows of data. The GROUP BY …HAVING count(*)>1 clause will select duplicate rows, but requires a comma delimited list of column variable names between BY and HAVING. What to do? SQLheads prefer to stay on the abstract side of typing lists. Fortunately, SAS SQL provides views of metadata. The first query,

```
SELECT name INTO :v SEPARATED BY ","
FROM DICTIONARY.columns
WHERE libname="WORK" AND memname="RECRODS";
```

constructs a list of column variable name in the dataset recrods. Substituting the list into the SELECT list and GROUP BY lists of the next query,

```
  CREATE TABLE dupsID AS
  SELECT &v,COUNT(*) AS n   FROM recrods
  GROUP BY &v   HAVING COUNT(*)>1  ;
```

to produce a table of rows that have the identical values in corresponding column variables.

SAS Version 9 has expanded substantially the number of dictionary views. Under V9 SQLheads have access to dictionary.vindex, and dictionary.vrefcon. The former lists <table>.<column> pairs indexed and distinguishes single from composite indexes. The latter lists referential integrity constraints. In SAS Explorer, these new V9 metadata views appear in the SASHELP library.

EMPOWER THE SAS SYSTEM

SAS SQLheads burrow through boundaries of operating and database systems. Changes in platforms, database systems, and technologies do not change the logical designs of databases. Once a database programmer has access to the metadata of another database system, the game is over. The SAS System provides all the tools SQLheads typically need to extract data from external data and prepare data inserts and updates. While the leading commercial RDBMS vendors encourage their contacts within organizations to buy their proprietary and expensive front-end and back-end development tools, SAS SQLheads quietly develop network connections, configure 'middleware', and build views of data sources throughout the home organization and beyond. In the struggle for survival in the world of information technology, SAS SQLheads have to empower the SAS System in their organizations to survive.

The SAS/ACCESS middleware for generic access to data sources (mainly Open Database Connectivity (ODBC) and OLE-DB) and so-called 'native drivers' for DB2, Oracle, and Sybase support connections to a very large proportion of data currently being stored in database systems. Local network and Internet connections to file systems give database programmers access through ftp and sockets to a perhaps even greater volume of data.

The SAS System currently has a critical role in moving data into, within, and from database and file systems. The old database system front-ends for data entry are being supplanted increasingly by Web forms, gimmickware, and bulk loading from external data sources. SAS preprocessing of data checks for data quality, including the key and referential integrity required by RDBMS's. SAS analysis and reporting tools make best use of valuable data resources. SAS also gives systems programmers and clients access to and control of data in otherwise closed database systems.

To burrow successfully through system boundaries, the database programmer must understand two concepts: 'connection strings' and engines. In this example, the connection strings appear in quotes:

```
LIBNAME SqlLib ODBC NOPROMPT="DRIVER=SQL
Server;Database=NCIDiet;Server=TestWEB1" schema='dbo';
```

The quotes suggest that SAS will send the strings as a message (consisting in this example of keywords and parameter values) across a system boundary to a connection object. The connection object, a block of computer memory, has to be installed and configured to read appropriate connection strings. The strings in this example do not include USER or PASSWORD information, so the connection object has to obtain them from the user's environment. When a correct string streams to the connection object, it interprets the string and establishes a connection through the LIBNAME reference (SqlLib) to the SAS session. To make the connection intuitive, SAS has kindly arranged to have the connection to an external data source look much like a connection to a SAS library on disk. From the viewpoint of the programmer, it has almost the same effect.

How does SAS know that it is reading from an external database and not from a disk file? The ODBC engine name in the example tells SAS to parse a data stream from the external database using the ODBC protocol for data transfers. The SQL query,

```
CREATE VIEW SQLSerTblMDVW AS
  SELECT * FROM dictionary.tables
  WHERE UPCASE(libname)='SQLLIB';
```

requests data from a database dictionary or catalog. In this case, the LIBNAME reference 'SQLLIB' in the WHERE condition specifies a SQL SERVER database. (See the LIBNAME above). The SAS System translates the request into SQL Server SQL and forwards it to the SQL Server database for processing. SAS uses the ODBC protocol to convert the yield of the query to a SAS view.

If we find the name of a table or view on the catalog of the SQL Server database, we can use it instead of a SAS dictionary reference to capture the tabular metadata object:

```
CREATE VIEW SQLSerIdxKeyVW AS
SELECT * FROM SQLLib.sysindexkeys;
```

That gives us access to external database system metadata even if SAS does not have a dictionary reference corresponding to it.

Once we have access to the metadata of an external database system, we have the keys to the castle (not counting hassles over access rights and battles with DBA's dedicated to the principle that all access to data in their databases should go through them). As an alternative to sending a connection string via a LIBNAME statement, we can 'pass thru' queries to external database systems in a connect statement. Oracle, Sybase, and DB2 examples follow:

```
proc sql;
connect to oracle (user=ora_id orapw=ora_pwd path="@alias")
    select * from connection to oracle  ( select * from tablename);
disconnect from oracle;
connect to sybase (user='sybase_username'
sybpw='sybase_passwd' server=SERVER);
     select * from connection to sybase ( select * from tablename);
disconnect from sybase;
connect to db2 (database=database);
    select * from connection to db2 ( select * from tablename);
disconnect from db2;
```

The queries differ only in the connection strings, engine names, and the syntax of the queries. In each case the innermost query that passes thru to the database system has to be in the syntax of the SQL flavor of the database system.

SI recommends the LIBNAME (native driver) method for database system access in typical situations. Exceptions include queries that use keywords and statements specific to the host database system and not to SAS, and grouping operations that only the host database system can optimize.

The close correspondence of SAS SQL queries to native queries in database systems empowers SAS to extract metadata and data from external database systems. In any organization that has different database systems operating concurrently, SAS SQLheads have a natural advantage.

MANAGE MISSINGS, NULLS, FUZZY LINKS

No database systems eliminates all the problems inherent in missing and null values. SAS SQL has no magic formula for handling these special values. ANSI SQL specifies a three-value logic for evaluating expressions that include missing or NULL values, whereas SAS DATA steps use a two-value logic. SAS SQL makes uneasy compromises between the two logics. In grouping operations, for example, SAS SQL treats missing values in grouping variables as matches to one another, while in summaries SAS SQL does not include missing values in calculations.

SQL does have a method for recognizing a missing or NULL value. The unary operator 'IS MISSING' combined with a column variable name evaluates to True whenever the variable has a missing value ('.' or special missing values if numeric or blank if character) or NULL where, as in a LEFT JOIN or UNION, values of variables in unmatched data items remain undefined. SAS SQLheads use this method to detect missings and NULL's and program around the difficulties they create. For example, SAS SQL evaluates a missing value as smaller than a non-missing value. The WHERE condition

(x<0) resolves to True when the value of the variable x is missing. The correct specification for the WHERE condition depends on context. Generally it makes sense to exclude undefined values. The condition (x<0 AND NOT x is missing) does that. In special cases it may make better sense to convert undefined values to zeroes. SQL provides the COALESCE function for that purpose:

(COALESCE(x,0)<0)

Functions such as COALESCE(), SOUNDEX(), and SPEDIS() yield 'fuzzy' values that work perfectly well in WHERE and ON conditions, but tend to confound SAS SQL's query optimization planning. In-line views presented earlier put the fuzzy values in a data stream that the SAS SQL compiler can index or order.
See Hermansen(2001) for a more on fuzzy linkage.

SIZE UP YOUR QUERIES

SQL query execution plans take action to reduce the volume of data at the earliest possible stage of processing. As a rule the SAS SQL query optimizer does a reasonably good job of restricting selections of column variables to those required for subsequent stages of processing and subsetting rows to those required for the solution. SQLheads combine JOIN's and UNION's in the same query, using in-line views if necessary, so the SQL compiler can optimize the query. While a good execution plan helps, it does not always prevent overuse of data storage resources. A crucial factor in determining data storage requirements for a query lies hidden in data: the number of multiples of key values in each of the tables being JOIN'ed. A good database design offers the first line of defense against multiples of key values. SQLheads also use SQL to count repeating key values in tables prior to joining them, and they control duplicates prior to joining the tables.

ACKNOWLEDGEMENTS

Westat colleagues and SAS-L contacts contributed substantially to the content of this article. Special thanks to Peter Crawford for information about dictionary tables in SAS V9. The author takes full responsibility for errors or omissions that remain.

REFERENCES

Hermansen, S. "Ten Good Reasons to Learn SAS Software's PROC SQL", Proceedings SUGI 1997, San Diego, 1997.
Hermansen, S.,'Structured Query Language: Logic,Structure, and Syntax', Proceedings of SESUG 2002, Savannah, 2002.
Hermansen, Sigurd, 'Fuzzy Key Linkage', Proceedings of SSU 2001, New Orleans, 2001.
Kent, Paul 'Inside PROC SQL's Query Optimizer', TS-320, http://ftp.sas.com/techsup/download/technote/ts320.html

CONTACT INFORMATION

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

Sigurd W. Hermansen
Westat, An Employee-Owned Research Company
1650 Research Blvd.
Rockville, MD 20850
(301) 251-4268
hermans1@westat.com