The Optimizer Project:

WHERE The Statement Is Not The Clause

Paul D Sherman, San Jose, CA
Russell Lavery, Ardmore, PA

ABSTRACT

Improving clock time and running smaller are goals for any programmer. In the SAS® Data step, it is well known that certain filtering expressions placed in a `WHERE` statement often lead to better processing efficiency. These expressions are simple "static" equalities, those without arithmetic operations or function calls. The Data step engine processes these statements at a very low level – as observations are being read in from the input dataset – so that later and more complicated Data step operations have as few rows as possible to deal with.
Therefore a SAS programmer learns early on to do their easy work – first things, first, -- in the `WHERE` statement.

This is unfortunate. Folks in the RDBMS camp know that the `FROM` clause is the first point of contact to one's data model. Only after processing all specifications and expressions in the `FROM` clause will the database system go on to the `WHERE` clause. In SAS Proc SQL code, position of the data filters (e.g., `WHERE gndr eq "M";`) can affect how fast the query runs and how much disk space it takes while running. Proper coding of the data filter allows Proc SQL to "delegate" the filtering to a "lower level" internal SAS subroutine called the Data Engine.

We learn in this paper that

	IMPORTANT POINT
in **SAS**, the data step `WHERE` is a statement, and processed first but in **SQL**, the query `WHERE` is a clause, and processed last.	

This paper suggests, and justifies, that SAS SQL programmers can often improve performance by adopting a few simple rules and coding style. Many annotated demonstrations, and a few humerous examples, are included as well as on the companion web site.

INTRODUCTION/BACKGROUND

In SAS SQL, a programmer codes what they want done and not how to do it. There is a subroutine, called the SQL Optimizer, that interprets the SQL code, queries system conditions (RAM, file sizes, presence of an index etc.) and creates a "mini-program" to produce the results specified in the query. This "mini-program" can be a complex program involving sorting, merging and indexing of many "working files" on the way to creating the result specified in the SQL code.

The SAS SQL Optimizer queries system conditions and attempts to re-program your query before running it. It does an excellent job of writing code that runs both "fast" and "small". However, in some outer joins, a programmer can use the syntax described in this paper and make the SQL code run faster, especially if the programmer has access to information about the files (file sizes after filtering) that is not available to the SAS SQL Optimizer. The above rules do not disable the Optimizer, they simply help it to do what it is trying to do – make filtering happen early in the query process.

Optimizer activity is transparent to the user. To see what the Optimizer planned as the "mini-program" for creating the result specified in the SQL code, turn on `_method` and `_tree` by coding

```
Proc SQL _method _tree;
```

and look in the log for the output from these two options.

SAS Proc SQL "sits on top of" regular SAS, interacting with many parts of regular SAS and "delegating" tasks whenever possible. An example of "delegating a task" is ordering of data. Ordering is not done by Proc SQL. When a SQL query specifies ordering, instructions – and the file to be ordered – are passed to SAS Proc Sort for processing.

Another important delegation, and the heart of this paper, is data filtering. There is a low level SAS subroutine that handles data "reads", called the Data Engine. Proc SQL gets its data through the Data Engine and passes, to the Data Engine, simple tasks, like filtering out unneeded observations and variables. In a similar manner, the V6 discussions of using a "subsetting if" versus a "where as a data step option", is a discussion of reading the data into

the Program Data Vector (PDV) and filtering observations on program execution in the PDV, versus telling the Data Engine to filter the observations before they are brought into the PDV.

The above fact is a small introduction to the title of the paper. The `WHERE`, in Proc SQL, functions very differently from the where in a Data Step. Understanding the function of "`WHERE`" and "`ON`" in SQL, and how they can be used to help the Optimizer delegate data filtering to a low level and create small "SQL working files," is the main focus of this article.

In the short review of the data step processing presented below, we can see some effect of if versus `WHERE` processing in the SAS Data step. The `WHERE` statement in the Data Step can delegate filtering to the Data Engine.

The following datasets, and others, will be used through the paper.

```
data Stu_demog;
 infile Datalines
   missover;
 input @1  Stu_id 4.
       @7  Age    2.0
       @11 Class  $char2.
       @15 Gndr   $char1.
 ;
datalines;
0001  18  FR  M
0002  19  SO  F
0003  20  JR  M
0004  21  SR  F
;
 run;
```

```
data Billing;
 infile datalines
     missover;
 input @1  Stu_id 4.
       @6  Dept     $char4.
       @12 BYR_TERM $char6.
       @19 PAID_YN  $char1.
 ;
datalines;
0001 Dorm  2001_F Y
0001 Store 2001_F Y
0001 Dorm  2001_S
0001 PARK  2001_F N
0001 DORM  2000_F Y
0001 GYMN  2000_F N
0001 FINE  2000_F
0002 DORM  2000_F Y
;
```

```
data Housing;
 infile datalines missover;
 input @1 Stu_id    4.
       @6 HYr_term  $char6.
       @13 Dorm     $char10.
       @24 style    $char6.
 ;
datalines;
0001 2001_F Riley_Hall Single
0001 2001_S Riley_Hall Single
0002 2000_F Smith_Hall Double
0002 2000_S Smith_Hall Double
0002 2001_F Smith_Hall Double
0002 2001_S Riley_Hall Suite
0003 1999_S Riley_Hall Double
0003 1999_S Riley_Hall Double
0003 2000_S Riley_Hall Suite
0003 2000_S Riley_Hall Suite
;
```

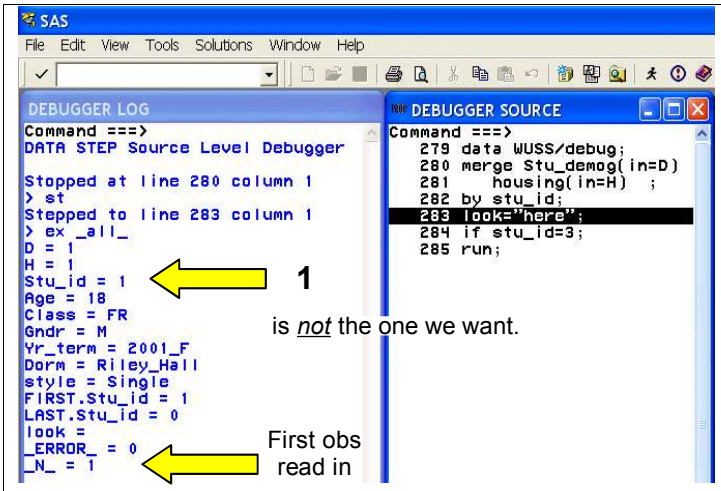
Figure 1

First, if the `WHERE` variable is not present in both files being merged, an error occurs as is show in Figure 2. Conversely, in SQL, a `WHERE` can be applied to a variable that is in only one of the data sets.

```
data WUSS/debug;
  merge Stu_demog(in=D)
        housing(in=H);
  by stu_id;
  look="here";
  where style="Suite";
  run;
```

```
256  data WUSS/debug;
257    merge Stu_demog(in=D)
258          housing(in=H);
259    by stu_id;
260    look="here";
261    where style="Suite";
ERROR: Variable style is not on file WORK.STU_DEMOG.
262    run;
```

Figure 2



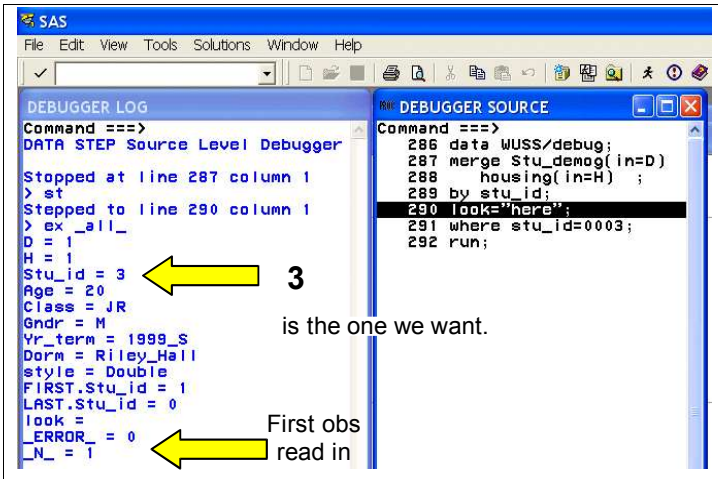
Secondly, if the programmer uses a subsetting if, observations are brought into the working "space" of the Data Step (here it is the Program Data Vector or PDV).

The data sets being merged are the stu_demog and housing data sets shown above. The program is in the right window and the PDV is in the left. The Data Step is about to execute the "nonsense" statement look="here";

As can be seen in Figure 3, _N_ is one and a student id Stu_id valued at 1 has been brought into the PDV. Time has been used to bring the observation into the PDV and have the data step determine if that observation should be "kept".

The Data step built-in loop must "spin" two times to get to Stu_id=3.

Figure 3



Thirdly, a subsetting where is more efficient than a subsetting if. Figure 4 shows the use of a subsetting where.

The data sets being merged are the Stu_demog and Housing data sets shown above.

As can be seen in Figure 4, _N_ is one and a student id valued at 3 has been brought into the PDV. The first observation brought into the PDV is the third observation in the source data set. This makes for a faster query.

The filtering of observations has been "passed down" to the Data Engine that feeds observations to the compiled data step code.

Remember, a data step can merge many files in just one pass through the data sets.

Figure 4

While Proc SQL can merge up to 32 files in one query, it has a very different merge process from the Data step (See "The De-normalize Transpose," in References, for a file-unlimited solution). In the Data step all files in the merge statement are opened at once and "currently active row pointers" created in/for each open file. The merge is processed in one top-to-bottom pass through the data sets. Alternatively, Proc SQL merges pairs of files (tables to SQL folks) and repeats the process until all files are merged. If one were to use SQL to merge files Able, Baker, Charlie and Delta, the process would be to merge Able and Baker into result_set_1. Then result_set_1 is merged with Charlie, giving result_set_2. Result_set_2 is merged with Delta, giving result_set_3, the desired output. The first file accessed by SQL is called the pivot table. SQL has several very different merging processes it can use to merge a pair of files and the Optimizer selects the process that it thinks will produce the fastest "time to solution."

The code below, run on the datasets shown above, illustrates the Data Step by "merge in one pass" process.

```
data WUSS;
 merge
   Stu_demog (in=D)
   billing (in=b)
   housing (in=H)
 ;
 by stu_id;
 look="here";
 run;
```

A row pointer in each file controls matching. For every "match" a line is output and the pointers are advanced, unless the pointer is on the last line of a by-group. Student 1 only had one line in Stu_demog and two observations in Housing. The one line in Stu_demog was matched to every line in the other files. The second line in housing was used many times (see inner red box).

Obs	Stu_id	Age	Class	Gnder	Dept	BY_TIME	PAID_YN	HYr_time	Dorm	Style	look
1	1	18	FR	M	Dorm	2001_F	Y	2001_F	Riley_Hall	Single	here
2	1	18	FR	M	Stor	2001_F	Y	2001_S	Riley_Hall	Single	here
3	1	18	FR	M	Dorm	2001_S		2001_S	Riley_Hall	Single	here
4	1	18	FR	M	PARK	2001_F	N	2001_S	Riley_Hall	Single	here
5	1	18	FR	M	DORM	2000_F	Y	2001_S	Riley_Hall	Single	here
6	1	18	FR	M	GYMN	2000_F	N	2001_S	Riley_Hall	Single	here
7	1	18	FR	M	FINE	2000_F		2001_S	Riley_Hall	Single	here
8	2	19	SO	F	DORM	2000_F	Y	2000_F	Smith_Hall	Double	here
9	2	19	SO	F	DORM	2000_F	Y	2000_S	Smith_Hall	Double	here
10	2	19	SO	F	DORM	2000_F	Y	2001_F	Smith_Hall	Double	here
11	2	19	SO	F	DORM	2000_F	Y	2001_S	Riley_Hall	Suite	here
12	3	20	JR	M				1999_S	Riley_Hall	Double	here
13	3	20	JR	M				1999_S	Riley_Hall	Double	here
14	3	20	JR	M				2000_S	Riley_Hall	Suite	here
15	3	20	JR	M				2000_S	Riley_Hall	Suite	here
16	4	21	SR	F							here

Figure 5

In comma separated inner joins, the implicit style `FROM` clause, the optimizer will re-write submitted code and "move" the `WHERE` (eg. `WHERE` gndr='M') so that it executes early in the query process (resulting in small working files). In outer joins, the optimizer is not able to "move" the execution of the `WHERE` clause and the filtering of data happens at (or close to) the place the `WHERE` was coded, often late in the query. Proper placement of the `WHERE`, and structure of the query, can help the optimizer. Note that the Data Engine can not do arithmetic, and a filter like `(height*12 LE 15)` can not be passed down to the Data Engine. This type of filter must be processed late in the query, in a position similar to that of the Filter in Example 1.

The understanding of the "`WHERE`" and "`ON`" in SQL, leads to five rules of writing SQL join syntax. These are rules that help the optimizer and are the major deliverable of the paper.

For outer joins the following rules are suggested:

1. Use explicit syntax for outer joins (use the `ON` clause style, not a `FROM` clause with files separated by commas)
2. Move filtering clauses from the "high level" `WHERE` statement to the `ON` clauses or the pivot table `WHERE` clause
3. Make the smallest table the pivot table (the first table listed in the `FROM`)
4. Sequence all other table joins in from smallest-to-largest size order
5. Don't use a `WHERE` clause in a multi-table SQL query except in the pivot table!!!
   Unles you're testing for non-existence or other Gestalt-like philosophy.
   See examples in the Appendix for some situations Where you really should use WHERE:
   Educational (life's a beach), Transportational (passing in the night), Clinical (no say, no pay)
   and Mathematical (off diagonal).

For a more in depth discussion of the optimizer and basic SQL processing, one might read the SUGI paper "The Optimizer Project" _method and _tree" mentioned in the references.

The Combinatoric Nature of the Problem Structure

This might be a good time to lay out the structure and complexity of the task faced when investigating SQL two-table join-type queries and applying the rules above. We can use the combination formula to understand the "size" of the problem.

There are inner joins, left outer joins, right outer joins and full joins. This paper will limit itself to inner joins and left outer joins. On the subject of inner joins, we just say the Optimizer lives up to its name and we have no suggestions for improving its functionality. (While we say that the Optimizer is effective on inner joins, we still suggest that a programmer apply the four rules of this paper (put smallest file in the pivot). Right outer joins are the mirror image of left outer joins and it is suggested that, while do-able, they are conceptually awkward and should be avoided. Full outer joins are not covered in this paper because of the explicit union involved in outer joins.

There are a variety of relation-expressions (a relation expression is an equation such as *variable* test-operation *value*) that can be used in Proc SQL. The test-operations are EQ, NE, LT, LE, GT, GE. Additionally, the value tested can be for a null value, against variable in the data set or against a constant value as in (`Age GE 45`).

The combinations of join-type, relation expression being tested, and kind of value being tested generates the possible "states of nature" that are shown in the table below. These "states of nature" define the size of the problem we face. All combinations or "states of nature" must be investigated, their underlying processes understood, the processes contrasted and finally rules abstracted.

Fortunately, we use some of our understanding of SQL to quickly collapse sections of the table and make the task smaller. Inner joins have been optimized so well, that we collapsed that whole section of the table and have no suggestions to make. When the value being checked is a constant or "static relation" (`Age=12`) or observation or "dynamic relation" (`test1.score GE grades.cutoff`) the processes are so similar as to allow us to discuss them as one process.

What `_tree` tells us about `WHERE` and `ON` processing is striking. Importantly, filtering occurs only at one of two places, early (low) in the process, by being passed to the Data Engine or later (higher) in the process when SQL applies the filter after joining tables.

Inner joins
The optimizer works so well, that we have not discovered anything to improve on the way it functions through coding**.** In all the two-table implicit "comma form" inner joins, the optimizer passed the filtering of observations down to the Data Engine, regardless of the use of `WHERE` or `ON`. The Optimizer is truly living up to its name.

For left outer joins, the result is similarly striking. When the four rules are applied (see examples 2, 4, 6, 8, 10, 12, 14 and 16 in the companion web site) the filtering is passed to the Date engine and applied early. An example of the improper coding style (example 1) and proper coding style (example 2 and 9) are shown immediately below.

Left Outer joins				
Logical Operator "Left Hand Side" (LHS) is a variable	Right Hand Side (RHS) Value being Checked	Test operator notation seen in _tree in the log *see appendix*	Implement Using a `WHERE` Clause NOT Recommended	**Implement Using an `ON` Clause** **Recommended**
Age **Is Null ;**	NULL	ISNL	Example 1	**Example 9**
Age **Is Not Null;**	Not NULL	NTNL	Example 2	**Example 10**
Age **=** 12 ; Age **=** T.age ;	Constant or var.	CEQ	Example 3	**Example 11**
Age **NE** 12; Age **<>** T.age;	Constant or var.	CNE	Example 4	**Example 12**
Age **GE** 12; Age **>=** T.age;	Constant or var.	CGE	Example 5	**Example 13**
Age **GT** 12; Age **>** T.age;	Constant or var.	CGT	Example 6	**Example 14**
Age **LE** 12; Age **<=** T.age;	Constant or var.	CLE	Example 7	**Example 15**
Age **LT** 12; Age **<** T.age ;	Constant or var.	CLT	Example 8	**Example 16**

Examples of proper and improper syntax and a Short introduction to output from the _tree option on the SQL statement on two table joins

The following six annotated pages illustrate how application of the five rules allows SQL to filter out observations early. Early filtering of observations makes for small working files and faster processing of the query. The first two examples show how the placement of the `WHERE` allows/prevents filtering of observations by the Data Engine for all possible relation expressions. The examples presented are examples 1 and 9 in the table above. These examples investigate two conditions/issues (`WHERE` placement and test operation condition in the `WHERE`) and all possible combinations of these conditions are presented and annotated in the appendix.

The next four annotated pages allow the reader to see how the rules generalize to three (and higher) table queries. The four pages show four ways of coding one query and how applying the rules given in this paper allow filtering to be passed to the Data Engine. Since these queries investigate only one issue (`WHERE` placement) that issue overlaps with the issues in the first 16 examples, the companion web site contains code to allow an interested reader to run each of the queries.

The dataset used in the 16 examples is shown below. It contains three variables and 15 observations.

```
data work.foo;
  infile datalines missover;
  input @1  pvar $char8.
        @10 xvar yymmdd8.
        @19 yvar 4.1
  ;
datalines;
aParm    20050510 14.1
aParm    20050511  6.1
aParm    20050512 13.1
aParm    20050513 11.1
aParm    20050514  9.1
bParm    20050515 12.1
bParm    20050516  8.1
bParm    20050517  7.1
bParm    20050518 15.1
bParm    20050519 16.1
cParm    20050520  4.1
cParm    20050521 11.1
cParm    20050522 10.1
cParm    20050523 20.1
cParm    20050524 15.1
;
  run;
```

The code and annotated output from representative left outer join examples 1 and 9 are shown on the following pages.

```
Example 1. ISNL/Where - IS NuLL - (Left) Outer Join
  SELECT 0 as nothing
  FROM foo as f
       left outer join foo as g ON f.pvar = g.pvar
                                   and f.xvar = g.xvar
  WHERE g.yvar is null;
Tree as planned.
                                  /-SYM-A-(nothing:1 flag=0031)
                        /-OBJ----|
          /-FIL----|
          |        |                        /-SYM-V-(g.yvar:3 flag=0001)
          |        |              /-OBJ----|
          |        |              |         \-SYM-A-(nothing:1 flag=0031)
          |        |--OTRJ---|
          |        |         |                          /-SYM-V-(g.pvar:1 flag=0001)
          |        |         |                /-OBJ----|
          |        |         |                |         |--SYM-V-(g.xvar:2 flag=0001)
          |        |         |                |         \-SYM-V-(g.yvar:3 flag=0001)
          |        |         |       /-SORT---|
          |        |         |       |        |                   /-SYM-V-(g.pvar:1 flag=0001)
          |        |         |       |        |         /-OBJ----|
          |        |         |       |        |--SRC----|         |--SYM-V-(g.xvar:2 flag=0001)
          |        |         |       |        |         \-TABL[WORK].foo opt=''
          |        |         |       |        |--empty-
          |        |         |       |        |                   /-SYM-V-(g.pvar:1)
          |        |         |       |        |         /-ASC----|
          |        |         |       |        |--ORDR---|         \-SYM-V-(g.xvar:2)
          |        |         |       |        |         \-ASC----|
          |        |         |       |--FROM---|
          |        |         |       |        |                   /-SYM-V-(f.pvar:1 flag=0001)
          |        |         |       |        |         /-OBJ----|
          |        |         |       |        |         |         \-SYM-V-(f.xvar:2 flag=0001)
          |        |         |       |        \-SORT---|
          |        |         |       |                  |                   /-SYM-V-(f.pvar:1 flag=0001)
          |        |         |       |                  |         /-OBJ----|
          |        |         |       |                  |         |         \-SYM-V-(f.xvar:2 flag=0001)
          |        |         |       |                  |--SRC----|
          |        |         |       |                  |         \-TABL[WORK].foo opt=''
          |        |         |       |                  |--empty-
          |        |         |       |                  |                   /-SYM-V-(f.pvar:1)
          |        |         |       |                  |         /-ASC----|
          |        |         |       |                  \-ORDR---|         \-SYM-V-(f.xvar:2)
          |        |         |       |                            \-ASC----|
          |        |         |       |--empty-
          |        |         |       |                            /-SYM-V-(g.pvar:1)
          |        |         |       |                  /-CEQ----|
          |        |         |       |                  |         \-SYM-V-(f.pvar:1)
          |        |         |       |--LAND---|
          |        |         |       |         |                  /-SYM-V-(g.xvar:2)
          |        |         |       |         \-CEQ----|
          |        |         |       |                  \-SYM-V-(f.xvar:2)
          |        |         |       |--JTAG(jds=1, tagfrom=1, flags=0)
          |        |         |       |--empty-
          |        |         |       |                            /-SYM-A-(nothing:1 flag=0031)
          |        |         |       |                  /-ASGN---|
          |        |         |       |                  |         \-LITN(0)
          |        |         |       \-OBJE---|
          |        |         |                 /-SYM-V-(g.yvar:3)
          |        |         \-ISNL---|
          \-SSEL---|
```

The characteristic to notice in this example is that Proc SQL is checking and filtering observations very late point in the process.

This means that all observations in the source data set have been read and processed up to the point of the filtering. SQL is filtering after the data after reading all obs, sorting all the obs and merging all obs. The working files are big and big files have been sorted.

BAD

7

Is Null Filtering of observations happens late, after the join

3

4

1

Outer join happens here

2

5

We coded 0 as Nothing to make the output smaller and fit on one page.

6

Above we can read the `_tree` output to understand the process. The process goes, in levels, from right to left, and from bottom to top, within a level. We see in ❶ that, we bring in `pvar`, `xvar` and `yvar`, from table foo, as a source of data. ❷ SQL can employ several join methods. The Optimizer decided that a sorted merge join was best and the information on how to sort is shown in 20. ❸ is a summary of the vars and their types that will be passed to the sort. ❹ shows the sort of the first table. The tree to the right of ❺ shows the same process was employed on the second table. ❻ summarizes the information to be passed to the subroutine that does the joining – joining will employ a logical AND of the two "dynamic relation" expressions . ❼ is a summary of the variables being passed to the join.

Finally the join happens. We can see that the variable gvar is passed through the join, because Proc SQL needs to FILter the observations on `yvar` – even though `yvar` is not required in the final query.

Lets compare this to the same query written using the ON clause.

```
9. ISNL/On - IS NuLl - (Left) Outer Join
  SELECT 0 as nothing
  FROM foo as f
       left outer join foo as g ON f.pvar = g.pvar
                                and f.xvar = g.xvar
                                and g.yvar is null;
Tree as planned.
                                /-SYM-A-(nothing:1 flag=0031)
                      /-OBJ----|
            /-OTRJ---|
            |         |                     /-SYM-V-(g.pvar:1 flag=0001)
            |         |           /-OBJ----|
            |         |           |         \-SYM-V-(g.xvar:2 flag=0001)
            |         |  /-SORT---|  3
            |         |  |        |                     /-SYM-V-(g.pvar:1 flag=0001)
            |         |  |        |           /-OBJ----|
            |         |  |        |           |         \-SYM-V-(g.xvar:2 flag=0001)
            |         |  |        |--SRC----|  1
            |         |  |        |           |--TABL[WORK].foo opt=''
            |         |  |        |           |          /-NAME--(yvar:3)
            |         |  |        |            \-ISNL---|
            |         |  |        |--empty-
            |         |  |        |                     /-SYM-V-(g.pvar:1)
            |         |  |        |           /-ASC----|
            |         |  |        \-ORDR-|  2
            |         |  |                    /-SYM-V-(g.xvar:2)
            |         |  |                    \-ASC----|
            |         |--FROM---|
            |         |  |                     /-SYM-V-(f.pvar:1 flag=0001)
            |         |  |           /-OBJ----|
            |         |  |           |         \-SYM-V-(f.xvar:2 flag=0001)
            |         |  \-SORT---|  4
            |         |  |        |                     /-SYM-V-(f.pvar:1 flag=0001)
            |         |  |        |           /-OBJ----|
            |         |  |        |           |         \-SYM-V-(f.xvar:2 flag=0001)
            |         |  |        |--SRC----|
            |         |  |        |           \-TABL[WORK].foo opt=''
            |         |  |        |--empty-
            |         |  |        |                     /-SYM-V-(f.pvar:1)
            |         |  |        |           /-ASC----|
            |         |  |        \-ORDR---|
            |         |  |                    /-SYM-V-(f.xvar:2)
            |         |  |                    \-ASC----|
            |         |--empty-
            |         |  |                     /-SYM-V-(g.pvar:1)
            |         |  |           /-CEQ----|
            |         |  |           |         \-SYM-V-(f.pvar:1)
            |         |--LAND-|  5
            |         |  |           |         /-SYM-V-(g.xvar:2)
            |         |  |           \-CEQ----|
            |         |  |                     \-SYM-V-(f.xvar:2)
            |         |--JTAG(jds=1, tagfrom=1, flags=0)
            |         |--empty-
            |         |  |                     /-SYM-A-(nothing:1 flag=0031)
            |         |  |           /-ASGN---|
            |         |  |           |         \-LITN(0)
            |          \-OBJE---|  6
  --SSEL---|
```

Using the ON increases the chance of the filter being passed down to the Data Engine. When the filtering is passed "down", working files are smaller and the query runs faster.

GOOD code

Outer join happens here

Is Null Filtering happens here

Pivot table

In example 9, the Optimizer has also selected a sorted merge as the most efficient process for this left outer join, so the tree will show sorting information and operations. ❶ shows the filtering operation passed to the Data Engine and observations with var NOT NULL are not read into the working files managed by SQL. Because of the early filtering of observations, there is minimal I/O associated with this file and it is likely to be very small. Then the file is ❸ sorted using information summarized in ❷. Sorting is a resource and time consuming process in SAS, but this file is as small as it can be, given the logic of this problem. All observations from the other copy of foo are brought in and sorted ❹. The Logical AND information for the join dynamic relation expressions is summarized in ❺. The information on how to create the variable called "nothing" is summarized in ❻. Finally the outer join happens. The key idea is that the joining is being done on a subset of foo and a "full copy" of foo. Smaller files are processed faster. In this case our first table accessed is very small. Filtering the pivot is discussed later in this paper.

If you use a “comma separated” implicit style inner join on a multiple (3 or more) file query, the Optimizer will check table sizes and will attempt to use the smallest file as the pivot table. Generally, using the smallest file as the pivot table results in small working tables. With a “comma separated” inner join, the order in which you specify the files in the FROM might not be the order in which the merge happens. The Optimzer will try to first merge the smallest table (pivot) with the second smallest file- and then that result with the largest file. Consider the from below.

```
FROM stu_demog_records, /* 1 record per student */
     billing,     /* 1 obs/student-"billing item" -> hundreds per student*/
     stu_housing  /*1 obs/student-term -> approx. 8 obs. in a students sr. year*/
```

Importantly, by specifying the phrase “Left Join” (or inner join) you put handcuffs on the optimizer. The tables will be joined in the order in which you specify them in the FROM. This means that the programmer can specify the pivot table and the order of table joining. This is useful if the programmer knows information that the Optimizer can not determine from its checking of file and system characteristics.

If there were no index on Stu_ID in the table Stu_demog_records *and I wanted information on only one student* (say stu_id=3), we would be in the position of knowing that the Stu_demog_records working file (the observations read into SQL from the file on disk) will be very small. In this case, we’d want to make the SQL use the filtered Stu_demog_records file (only one obs) as the pivot table. This can be done by coding the join as an explicit join (left join with an ON clause).

For a school with 10,000 students, some approximate file sizes for the above from in different join orders could be.
Stu_demog_records *joined to* billing *with result joined to* stu_housing
1 observation joined with 150 observations for that student → 150 observations in the working file:
joined with 8 observations for that student → 600 observations

if the merge were to go in reverse order, the working files might have the following sizes:
20,000 housing records joined with 150,000 billing records. → 3,000,000,000 records in the working file.
joined with 1 observation from Stu_demog_records → 600 observations

Join order can be important. Making the pivot table “small” can greatly improve “time to solution” and amount of disk space required. The Optimizer works to create minimum size working files without any programmer attention.

Here is a comparison of the SAS data step and SQL in general.

```
          --- BAD ---
data _null_;          SELECT 0 as nothing
  merge foo bar;      FROM foo
  by key;                  inner join bar ON foo.key = bar.key
  if b = 10;          WHERE bar.b = 10
  run;                                    Clause

          +++ GOOD +++
data _null_;          SELECT 0 as nothing
  merge foo bar;      FROM foo
  by key;                  inner join bar ON foo.key = bar.key
  where b = 10;                           and bar.b = 10
  run;
      Statement
```

When (statically) filtering on the pivot table the optimal situation is a little different. Since the pivot is special – it has no ON clause – we must use a WHERE clause for its filtering. However, we must not use the outer-most WHERE clause because this would happen *after* all of the joins. Instead, we make the pivot into a sub-query, telling the database we'd like to process the sub-query *first*. For added protection of processing order one should use GROUP BY in this sub-query, so that an over-active database optimizer doesn't rip apart the sub-query.

Below shows good Data Step coding and good SQL coding. Both are delegating data filtering to the Data Engine and cause filtering to happen as soon as possible. In the SQL query, the WHERE on the pivot table will be delegated to the Data Engine and the ON filters will happen as soon as possible. The SQL code below is an example of early filtering of observations in a complex query.

```
          +++ GOOD +++
data _null_;          SELECT 0 as nothing
  merge foo bar;      FROM  (
  by key;                     SELECT key
  where a = 10;               FROM foo
  run;                        WHERE a = 10  /* passed to Data Engine*/
                              GROUP BY key                        /* filtering */
                            ) as foo (key)                        /*  happens  */
                            left outer join bar ON foo.key = bar.key /* early    */
```

Aside from tightly binding a query fullselect statement, the `GROUP BY` clause provides another nice option. Usually, the first or pivot table object is a specification of primary keys driving one's data model. If any of these keys have non-unique (i.e., repeated) values between rows, incorrect output will result. By using a post-aggregate filter – the `HAVING` clause, as in `HAVING count(*)=1` – one can force uniqueness and exclude any uniqueness violating input data. Thus, the data model is kept extremely robust. Alternately, one can filter out all but the violating input in order to find and remedy the bad input data.

CONCLUSION

Between a statement (Data step) and a clause (SQL), `WHERE` is not the same. The Optimizer is very effective on inner joins. If one is coding an inner join, it is generally okay to use implicit comma form and let the optimizer decide how to process the code. If the programmer knows information about the files that the optimizer can not know, specifying an inner join and applying the rules laid out in the article can improve performance and clarify one's query model architecture in the source text.

For outer join queries, a programmer can often help the Optimizer by applying the rules below.

1. Always use SQL explicit join syntax (the `ON` clause style)
2. Move data step `WHERE` statements to SQL statement `ON` clauses
3. Make careful choice for the leading pivot table
4. Sequence all other table joins in smallest-to-largest size order
5. Don't use a `WHERE` clause in a multi-table SQL query!!!
   unless you're testing for non-existance or other Gestalt like philosophy

REFERENCES

Date, C. J. “The Database Relational Model : A Retrospective Review and Analysis.” Reading, MA: Addison-Wesley, 2001.

Delwiche, Laura and Susan Slaughter. "The Little SAS® Book: A Primer." Cary, NC: SAS Institute Inc., 2003.

Lavery, Russell. *The SQL Optimizer Project: _Method and _Tree in SAS9.1*, in *Proceedings of the Thirtieth Annual SAS User Group International Conference*. Cary, NC: SAS Institute Inc., 2005. Paper 101-30.

Lavery, Russell. *An Animated Guide: The Data Step Debugger* in *Proceedings of the Seventeenth Annual North East SAS User Group Conference*, Paper, 2004 Paper How 05

Sherman, Paul. *Creating Efficient SQL – Four Steps to a Quick Query*, in *Proceedings of the Twenty-Seventh Annual SAS User Group International Conference*. Cary, NC: SAS Institute Inc., 2002. Paper 073-27.

Sherman, Paul. *Creating Efficient SQL – Union Join without the UNION Clause*, in *Proceedings of the Twenty-Ninth Annual SAS User Group International Conference*. Cary, NC: SAS Institute Inc., 2004. Paper 064-29.

Sherman, Paul. *Creating Efficient SQL – The De-normalize Transpose*, in *Proceedings of the 2005 Regional Pharmaceutical SAS User Group Conference*. Cary, NC: SAS Institute Inc., 2004. Paper PO-42.

COMPANION WEB SITE – Appendicies and example SAS program code

http://www.idiom.com/~sherman/paul/pubs/dwdb_where-WU2005

CONTACT INFORMATION

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

Paul D Sherman
335 Elan Village Lane, Apt. 424
San Jose, CA 95134
Phone: (408) 383-0471
Email: sherman@idiom.com

Russ Lavery
9 Station Ave. #1
Ardmore, PA 19003
Phone: (610) 645-0735
Email: russ.lavery@verizon.net

CONTENT CONCRETE EXAMPLES – Situations Where you really should use WHERE

```
/* Educational (life's a beach)
   Your daughter is planning her high school class schedule for the spring term.
   She wants to find an easy elective which has no prerequisites nor is required
   by any other class. It's spring time, come on!

  (for any row in the class schedule there is no other row where CLASS matches
   PREREQ or PREREQ matches CLASS)
*/

data work.educdata;
  infile datalines missover;
  input @1  year   4.
        @6  term   $char8.
        @15 class  $char32.
        @48 prereq $char24.
  ;
datalines;
2005 Spring   Algebra 1                       .
2005 Spring   Algebra 2                       Algebra 1
2005 Spring   Analytic Geometry               Algebra 2
2005 Spring   Spanish 1                       .
2005 Spring   Spanish 2                       Spanish 1
2005 Spring   Politics & Power               Government
2005 Spring   Government                      History
2005 Spring   History                         European History
2005 Spring   English Literature              English
2005 Spring   English                         .
2005 Spring   Composition & Rhetoric          English Literature
2005 Spring   Drawing & Painting              Art History
2005 Spring   Art History                     History
2005 Spring   European History                .
2005 Spring   Underwater Basket Weaving       .
2005 Spring   Suntan Psychology               .
;
  run;

proc sql noprint _tree;
  create table work.educ as (
    SELECT s.year, s.term, s.class
    FROM work.educdata AS s
         LEFT OUTER JOIN work.educdata AS p ON s.year = p.year
                                            AND s.term = p.term
                                            AND  s.class = p.prereq
         LEFT OUTER JOIN work.educdata AS r ON s.year = r.year
                                            AND s.term = r.term
                                            AND s.prereq = r.class
    WHERE p.class is null AND r.class is null
  );
  quit;

proc print data=work.educ;
  title 'Educational Example';
  title2 'Easy electives which have no prerequisites';
  title3 'nor are required by any other class.';
  run;
```

Easy electives which have no prerequisites
nor are required by any other class.

Obs	year	term	class
1	2005	Spring	Underwater Basket Weaving
2	2005	Spring	Suntan Psychology

CONTENT CONCRETE EXAMPLES – Situations Where you really should use WHERE

```
  /* Clinical (no say, no pay)
     Find all of a patient's visits which have not yet been paid (their medical
     record file has not been processed) but ignore any visit for which there is
     no laboratory test result.

    (labdat has a valid ID/TS but the VALUE column is null for some PARMs)
  */
```

```
data work.patients;
  infile datalines missover;
  input @1 pid    6.
        @8 gender $char1.
  ;
datalines;
123456 m
234332 f
;
  run;
```

```
data work.visits;
  infile datalines missover;
  input @1 pid 6.
        @8 ts  datetime20.1
  ;
datalines;
123456 15JUL2005:14:37:12.1
123456 15JUL2005:14:37:12.1
234332 12JUL2005:08:25:00.1
;
  run;
```

```
data work.labdat;
  infile datalines missover;
  input @1  pid      6.
        @8  ts       datetime20.1
        @29 testname $char3.
        @33 value    6.3
  ;
datalines;
123456 15JUL2005:14:37:12.1 RBC 23.502
234332 12JUL2005:08:25:00.1 PT   1.323
234332 05MAY2005:10:46:23.4 PT   .
;
  run;
```

```
data work.payments;
  infile datalines missover;
  input @1  pid    6.
        @8  ts     datetime20.1
        @29 amount dollar6.2
        @37 pay_id 5.
  ;
datalines;
123456 15JUL2005:14:37:12.1 $142.25 33535
234332 12JUL2005:08:25:00.1  $25.00 13123
;
  run;
```

```
  proc sql noprint _tree;
    create table work.medi as (
      SELECT p.pid, v.ts, lab.testname, lab.value
      FROM work.patients AS p
           INNER JOIN work.visits AS v ON p.pid = v.pid
           LEFT OUTER JOIN work.labdat AS lab ON v.pid = lab.pid
                                             AND v.ts = lab.ts
                                             AND lab.value is not null
           LEFT OUTER JOIN work.payments AS pay ON v.pid = pay.pid
                                               AND v.ts = pay.pid
      WHERE pay.amount is null
    );
    quit;

  proc print data=work.medi;
    title 'Clinical Example';
    title2 'Patient visits which have laboratory test data';
    title3 'but have not been paid, or medical record file unprocessed';
    run;
```

CONTENT CONCRETE EXAMPLES – Situations Where you really should use WHERE

```
/* Transportational (passing in the night)
   The North-South train arrives every fifteen minutes. At its common destination,
   it sometimes meets the East-West train. Find the arrivals of the North-South
   train which do not meet the East-West connection.

   (timepoints of some trains do not line up to other trains at same stations)  */

  data work.nssched;
    infile datalines missover;
    input @1 trainid    3.  @5 stationid  $char16.  @22 arrtime   time8. ;
datalines;
101 Thong Lo          .
101 Asoke             .
101 Nana              .
101 Chid Lom          .
101 Siam              08:51:00
101 Natl Stadium      .
102 Thong Lo          .
102 Nana              .
102 Ploen Chit        .
102 Siam              09:06:00
;
datalines;
103 Prakanong         .
103 Siam              09:21:00
104 Onnuch            .
104 Ekka Mai          .
104 Prohm Pohng       .
104 Nana              .
104 Ploen Chit        .
104 Chidlom           .
104 Siam              09:36:00
104 Natl Stadium      .
;
  run;

  data work.ewsched;
    format trainid 3. stationid $16. arrtime time8.;
    input @1  trainid   3.  @5  stationid $char16.  @22 arrtime   time8. ;
datalines;
201 Siam              09:06:00
201 Racha Tevi        .
201 Annusavari Chai   .
201 Aree              .
201 Mo Chit           .
;
datalines;
204 Siam              09:21:00
204 Paya Tai          .
204 Sanam Pao         .
204 Sapan Kwai        .
204 Mo Chit           .
;
  run;
/* We first must derive a distinct list of East-West stations and use them to
   filter those stations of the North-South train which do not match. Hence the
   INNER JOIN step precedes our OUTER JOIN existence filter.

   Note the use of min() to avoid the following SAS Proc SQL behavior:
WARNING: A GROUP BY clause has been transformed into an ORDER BY clause because
         neither the SELECT clause nor the optional HAVING clause of the
         associated table-expression referenced a summary function.              */

  proc sql noprint _tree;
    create table work.tran as (
      SELECT ns.trainid, ns.stationid, ns.arrtime
      FROM work.nssched AS ns
           INNER JOIN (
             SELECT min(stationid) FROM work.ewsched GROUP BY stationid
           ) AS ewsta (stationid) ON ns.stationid = ewsta.stationid
           LEFT OUTER JOIN work.ewsched AS ew ON ns.stationid = ew.stationid
                                             AND ns.arrtime = ew.arrtime
      WHERE ew.arrtime is null
    );
    quit;
proc print data=work.tran; title 'Transportational Example';
title2 'North-South train arrival times';
title3 'which do not meet the East-West train';
run;
```

CONTENT CONCRETE EXAMPLES – Situations Where you really should use WHERE

```
/* Mathematical
   Find the set of off-diagonal elements of upper-triangular
   matrix (2-vector) L-U = foo * foo
*/

data work.mathdata;
  infile datalines missover;
  input @1  pvar $char8.
        @10 xvar yymmdd8.
        @19 yvar 4.1
  ;
datalines;
aParm    20050510 14.1
aParm    20050511  6.1
aParm    20050512 13.1
aParm    20050513 11.1
aParm    20050514  9.1
bParm    20050515 12.1
bParm    20050516  8.1
bParm    20050517  7.1
bParm    20050518 15.1
bParm    20050519 16.1
cParm    20050520  4.1
cParm    20050521 11.1
cParm    20050522 10.1
cParm    20050523 20.1
cParm    20050524 15.1
;
  run;

proc sql noprint _tree;
  create table work.mat as (
    SELECT f.xvar AS f,
           g.xvar AS g
    FROM work.mathdata AS f
         INNER JOIN work.mathdata AS g ON f.pvar = g.pvar
         LEFT OUTER JOIN work.mathdata AS gg ON f.pvar = gg.pvar
                                            AND f.xvar = gg.xvar
                                            AND g.xvar = gg.xvar
    WHERE gg.xvar is null
  );
  quit;

proc print data=work.mat;
  title 'Mathematical Example';
  title2 'off-diagonal elements of cartesian product mathdata * mathdata';
  format f g yymmdd8.;
  run;
```

APPENDIX – PROC SQL _METHOD DESCRIPTIONS (See companion web site for full source text)

```
************** (LEFT) OUTER JOIN ******************

1. ISNL/Where - IS NuLl - (Left) Outer Join
2. NTNL/Where - is NoT NuLl - (Left) Outer Join
3. CEQ/Where - Compare EQual - (Left) Outer Join
4. CNE/Where - Compare Not Equal - (Left) Outer Join
5. CGE/Where - Compare Greater than or Equal - (Left) Outer Join
6. CGT/Where - Compare Greater Than - (Left) Outer Join
7. CLE/Where - Compare Less than or Equal - (Left) Outer Join
8. CLT/Where - Compare Less Than - (Left) Outer Join
                        L7    L6      L5       L4       L3      L2   L1
 _tree path segment:--SSEL--FIL--+--OTRJ--+--FROM--+--SORT----SRC--TABL[]
                                 |        |        +--SORT----SRC--TABL[]  (pivot)
          test after the join    |        +--LAND-----CEQ--|
                                 +--{ISNL,NTNL,...}--|

 _method:       sqxslct  sqxfil  sqxjm  sqxsort  sqxsrc( WORK.FOO(alias = G) )
                                        sqxsort  sqxsrc( WORK.FOO(alias = F) )

9. ISNL/On - IS NuLl - (Left) Outer Join
10. NTNL/On - is NoT NuLl - (Left) Outer Join
11. CEQ/On - Compare EQual - (Left) Outer Join
12. CNE/On - Compare Not Equal - (Left) Outer Join
13. CGE/On - Compare Greater than or Equal - (Left) Outer Join
14. CGT/On - Compare Greater Than - (Left) Outer Join
15. CLE/On - Compare Less than or Equal - (Left) Outer Join
16. CLT/On - Compare Less Than - (Left) Outer Join
                                                          test at the source
                           L6    L5       L4       L3    L2       L1
 _tree path segment:      --SSEL--OTRJ--+--FROM--+--SORT--SRC--+--TABL[]
                                        |        |             +-{ISNL,NTNL,...}--|
                                        |        +--SORT--SRC--TABL[]  (pivot)
                                        +-LAND---CEQ

 _method:             sqxslct  sqxjm   sqxsort  sqxsrc( WORK.FOO(alias = G) )
                                       sqxsort  sqxsrc( WORK.FOO(alias = F) )

****************** INNER JOIN ********************

1-8.  ISNL,NTNL,CEQ,CNE,CGE,CGT,CLE,CLT/Where - Inner Join
9-16. ISNL,NTNL,CEQ,CNE,CGE,CGT,CLE,CLT/On - Inner Join
                               sqxslct  sqxjhsh  sqxsrc( WORK.FOO(alias = F) )
                                                 sqxsrc( WORK.FOO(alias = G) )
```

```
             SSEL - select
             ASGN - assignment
             LITN - LITeral Numeric
Proc SQL     SYM-A - SYMbolic Alphabetic (character) variable
             SYM-V - SYMbolic (numeric) Variable
 _Tree       TABL - physical base TABLe
             OBJ - abstract data type structure OBJect; defines column names
GLOSSARY     SRC - data SouRCe; contains row collection data
             FROM
             LAND - Logical AND
             JOIN - inner JOIN
             OTRJ - OuTeR Join
```