

Using the WHERE Statement

Bob Virgile
Robert Virgile Associates, Inc.

Overview

The SAS® software permits a WHERE statement in all version 6 releases. While the syntax is straightforward, it takes a little planning to figure out how to properly apply this tool. This paper examines WHERE applications at an intermediate to advanced level.

Some Basics

Subject to the limitations outlined below, WHERE can replace IF for subsetting in a DATA step. In theory, WHERE would be more efficient.

```
DATA FLORIDA;
SET USA;
WHERE /* vs. IF */ STATE='FL';
```

The theory is this. IF reads in every observation, and then deletes those not needed. WHERE examines the observation to see whether it is needed, and reads in the needed observations only. The next section will examine efficiency considerations in more detail.

These limitations apply to using WHERE, none of which apply to using IF:

1. WHERE works with SAS data sets only, never with raw data.
2. WHERE can refer to variables in the incoming data set, not to newly created variables.
3. Prior to release 6.07, WHERE cannot use functions.
4. A DATA step can use just one WHERE statement per SET, MERGE, or UPDATE statement.

Limitation #2 implies that WHERE cannot refer to N, IN=, END=, or BY variables. In addition, for DATA steps which read in more than one SAS data set, the WHERE statement variables must appear in every source. In some cases, using (WHERE=) as

a data set modifier can circumvent that last limitation. For example, the first program below can use (WHERE=), but the second must use IF.

```
DATA BOTH;
MERGE ONE (WHERE=(X=10))
      TWO (WHERE=(Y=20));
BY ID;
```

```
DATA BOTH;
MERGE ONE TWO;
BY ID;
IF X > Y;
```

Limitation #4 is misleading in that multiple WHERE statements generate no error messages. (They used to in 6.03/6.04.) Instead, the second WHERE statement replaces the first.

WHERE also supports new comparison operators which can simplify programs. The most useful operator, LIKE, uses wild card characters (_ and %) for selecting character values. For example, if SYMBOL had a length of 5, these two statements would select the same observations:

```
WHERE SYMBOL LIKE '_ABCD';
WHERE SUBSTR(SYMBOL,2,4)='ABCD';
```

The _ means any single character. Similarly, % means any combination of characters. So these two statements also select identical observations:

```
WHERE SYMBOL LIKE '%ABCD';
IF INDEX(SYMBOL,'ABCD') > 0;
```

Use single not double quotes when using %, or else %ABCD will be interpreted as calling the macro ABCD.

Other new WHERE operators primarily enhance interactive sessions.

=*

Selects values that "sound like" a specified string. May overcome spelling mistakes or variations.

SAME AND

When using RUN groups, adds new conditions to existing WHERE conditions, generating a smaller subset.

A few new operators do not really add to functionality:

? or CONTAINS

Selects observations containing a specified string. Could be replaced with an INDEX function.

IS NULL or IS MISSING

Selects missing values. Could be replaced with a comparison operator. For numerics, includes special missing values.

BETWEEN AND

Selects a range. Could be replaced with comparison operators.

Perhaps these new operators work faster than the alternatives. Check the manual for details on how to use them.

Efficiency

Despite the theory, test cases show that IF can be more efficient than WHERE. For example, the following test results compare WHERE vs. IF in a simple program:

```
DATA FLORIDA;
SET USA;
WHERE /* vs. IF */ STATE='FL';
```

These are not necessarily conclusive tests; they merely represent examples. The tests show the percentage of total CPU time saved by switching from IF to WHERE:

CPU % Saved by WHERE

		% Obs Selected		
		20	50	80
30 vars				
PC 6.08	27	8	-41	
PC 6.12	6	-1	-4	
MF 6.07	0	-11	-25	
MF 6.08	0	-16	-28	
100 vars				
PC 6.08	26	1	-12	
PC 6.12	7	-1	-4	
MF 6.07	15	3	-16	
MF 6.08	16	-2	-14	

While test results can vary widely from one release to the next and from one platform to the next, two factors suggest switching to WHERE:

1. The more variables being read in, and
2. The fewer the number of observations being accepted

This implies that careful programming technique should influence your choice of IF vs. WHERE. For example, compare these two programs:

```
DATA FLORIDA;
SET USA;
WHERE STATE='FL';
KEEP CITY POP;
```

```
DATA FLORIDA;
SET USA (KEEP=CITY STATE POP);
IF STATE='FL';
DROP STATE;
```

The data set USA might contain dozens of variables. In that case, the first DATA step carelessly reads in all variables. (The KEEP statement affects the output, not the input.) Under those conditions, WHERE would be more efficient than IF.

The second DATA step reads in only the required variables. (The KEEP= data set modifier on the SET statement affects the input, not the output.) IF would be more efficient than WHERE, since the number of variables read in is small.

WHERE can increase program efficiency by subsetting in a PROC step. Any procedure which reads a SAS data set can add a WHERE statement:

```
PROC MEANS DATA=USA;
VAR POP;
WHERE STATE='FL';
```

This feature can be a MAJOR advantage. However, it can also be misused. If you plan on running PROC MEANS for each of two states, don't process the larger data set twice. Instead, create subsets in a simple DATA step:

```
DATA FLORIDA VERMONT;
SET USA (KEEP=STATE POP);
IF STATE='FL' THEN
  OUTPUT FLORIDA;
ELSE IF STATE='VT'
  THEN OUTPUT VERMONT;
```

Feed each subset individually into PROC MEANS.

WHERE can utilize an existing index to retrieve data faster. Given that:

1. A suitable index exists, and
2. The software estimates that using the index will be faster,

the software automatically uses the index to retrieve the data. Notice that:

1. You have no control over whether or not an index gets used.
2. The order of the output observations changes (usually) if an index is used.

Creating an index is expensive. Don't create one just to be able to subset.

Check the manual for details on what makes an index "suitable" and when the index will be used.

Ease of Programming

When you work with logical variables such as BY and END= variables, WHERE should normally replace IF. By virtue of these features:

1. WHERE reads in only the needed observations.
2. The DATA step can look ahead through the data to find the next observation that will be read in.

WHERE will simplify programs which currently use IF. Let's take a typical example:

```
DATA _NULL_;
SET USA END=EOF;
WHERE /* vs. IF */ STATE='FL';
TOTAL + POP;
IF EOF THEN PUT
  'TOTAL FLORIDA POPULATION: '
  TOTAL;
```

Both IF and WHERE can perform the subsetting. Both are correct syntax. However, IF usually fails to generate a report. The key issues are the value of EOF, and the value of STATE on the last observation in USA. When STATE is FL on the last observation, both IF and WHERE work perfectly. But when STATE has any other value, WHERE works while IF fails.

IF reads every observation, deleting those that are not needed. Therefore, EOF remains 0 until the very last observation of USA is read. If that last observation has STATE=FL, the program executes the remaining programming statements: it computes the final value for TOTAL and prints the report. However, if the last observation has STATE=VT, the program deletes the observation and never executes the remaining programming statements. The DATA step will end without generating a report.

WHERE reads in just some observations, those having STATE=FL. WHERE can accurately detect end-of-file when reading the last FL observation; the DATA step has look-ahead capabilities. So WHERE sets EOF to 1 when reading the last FL observation, regardless of whether or not this is the last observation in USA. As a result, the report is always generated, as long as USA contains at least one FL observation.

In similar fashion, IF vs. WHERE generates different values for BY variables.

```
DATA BIGCITY;
SET USA;
BY STATE;
WHERE /* vs. IF */ POP > 50000;
```

Subsetting with IF, the selected observations could have FIRST.STATE and LAST.STATE always equal to 0. By reading in every observation, the DATA step assigns FIRST.STATE=1 to a state's first observation regardless of the value of POP. Subsetting with IF could subsequently delete that observation.

Subsetting with WHERE, the DATA step reads in just the desired observations. For the first of these (in each STATE), FIRST.STATE=1. For the last, LAST.STATE=1. In most programs, it is much more convenient when every selected STATE has an observation with BY variables equal to 1.

In the last two examples in this section, choosing WHERE vs. IF changes the value of _N_. The first example creates a macro variable for each numeric variable name in USA.

```
PROC CONTENTS DATA=USA NOPRINT
  OUT=VARS (KEEP=NAME TYPE);

DATA _NULL_;
SET VARS END=LAST;
WHERE /* vs. IF */ TYPE=1;
CALL SYMPUT
  ('VAR' || LEFT(_N_), NAME);
```

```
IF LAST THEN CALL SYMPUT  
('TOTVARS', _N_);
```

Key program features:

1. The output data set VARS contains one observation for each variable from the input data set USA.
2. NAME is a character variable in VARS, containing the name of the original variable from USA.
3. TYPE is 1 (numeric) or 2 (character) depending on the original variable's type.
4. Each execution of the SET statement reads in the next observation that satisfies the WHERE condition.

&VAR1 will be the name of the first numeric variable in USA, &VAR2 the name of the second, etc.

Using IF instead of WHERE would read in every observation, throwing off the value of _N_. It would be necessary to add a statement such as N+1; following a subsetting IF. Then, replace references to _N_ with the new variable N. In addition, the previous difficulty -- locating end-of-file with a subsetting IF -- arises once again. &TOTVARS might not get created.

WHERE can use _N_ for subsetting more easily than IF. This feature offsets another limitation on WHERE: it cannot be used with OBS= (other than MAX), FIRSTOBS= (other than 1), or POINT=. Both of these DATA steps attempt to read in a sample, for purposes of testing a longer program:

```
DATA FLORIDA;  
SET USA;  
WHERE STATE='FL';  
OUTPUT;  
IF _N_=10 THEN STOP;
```

compared to

```
OPTIONS OBS=100;  
  
DATA FLORIDA;  
SET USA;  
IF STATE='FL';
```

The first DATA step is guaranteed to produce a sample of 10 observations. Each iteration of the DATA step both increments _N_ by 1 and searches through the data set for the next desired observation.

The second DATA step, while able to use OBS=, generates a sample of unknown size. FLORIDA may contain anywhere from 0 to 100 observations, depending on how many times FL is found in the first 100 observations of USA.

WHERE vs. (WHERE=)

(WHERE=) can be used as a data set modifier. This gives the program flexibility when reading from more than one SAS data set. This last section examines applications that take advantage of the added flexibility.

The first example selects and prints all of a STATE's observations, as long as at least one POP is 300000 or more. Assuming the data are already sorted:

```
DATA LARGE;  
MERGE USA  
      (WHERE=(POP >= 300000)  
        IN=BIG KEEP=STATE POP)  
      USA;  
BY STATE;  
IF BIG;
```

(WHERE=) applies to just the first mention of USA, reading in the large POPs only. The second mention of USA reads in all observations. If two different data sets were being merged, the program would clearly illustrate selecting matches vs. mismatches. This program treats USA as if it were two separate incoming data sets. The Log could easily contain the message (harmless in this case):

MERGE STATEMENT HAS MORE THAN ONE DATA set WITH REPEATS OF BY VALUES.

Such a message signifies that at least one STATE has multiple observations satisfying the WHERE condition.

In this final example, let's be snobs. We're going to select a STATE to live in, based on the percentage of cities which contain:

1. a university, and/or
2. a four-star restaurant.

Assuming one observation per city, where the variables COLLEGE and _4_STAR can be YES or NO:

```
PROC SORT DATA=USA;
BY STATE;
```

```
PROC FREQ DATA=USA;
BY STATE;
TABLES COLLEGE /
      NOPRINT OUT=SKOOLING;
TABLES _4_STAR /
      NORPINT OUT=EATING;
```

PROC FREQ generates the variables COUNT and PERCENT, in each output data set. However, we are interested in the statewide percent, not the global percent. Therefore, the program uses sorted data and includes the BY statement with PROC FREQ. If we wanted COUNT but not PERCENT, we could have omitted the sort and produced two-way tables:

```
TABLES STATE*COLLEGE;
TABLES STATE*_4_STAR;
```

Current releases of the software also support including marginal percentages in an output data set. We finally proceed with WHERE:

```
DATA YUPPIES;
MERGE SKOOLING
      (WHERE=(COLLEGE='YES')
        RENAME=(PERCENT=SKOOLING))
      EATING
      (WHERE=( _4_STAR='YES')
        RENAME=(PERCENT=EATING));
BY STATE;
```

WHERE reads in just one (or none) observations per STATE, producing a one-to-one MERGE. Structure of the final data set YUPPIES:

- One observation per STATE.
- The variable SKOOLING is the percentage of cities in that STATE containing a university.
- The variable EATING is the percentage of cities in that STATE containing a four-star restaurant.
- SKOOLING and EATING could be missing, but could never be zero. We could add statements to change missing values to zero.

(WHERE =) allows the software to combine the two sources in a single DATA step.

If you have questions or comments about this paper, feel free to call (or write):

Bob Virgile
Robert Virgile Associates, Inc.
3 Rock Street
Woburn, MA 01801

(781) 938-0307

SAS is a registered trademark of SAS Institute Inc.