Paper TU07

An Animated Guide: Knowing SQL Internal Processes makes SQL Easy, makes SQL easy

Russ Lavery – Numeric Resources Contractor, Ardmore, PA

ABSTRACT

This paper shows a flowchart of Proc SQL internal processes that makes Proc SQL easier to learn. Knowing the flowchart allows creation of simple rules that can be used to explain how to code Proc SQL queries. Without the flowchart, Proc SQL is a black box. Teaching “black box Proc SQL” has usually been a process of showing examples until the student can say, “I know what will happen when I code like this, because I’ve done this query before”. Mastery only happens as students recognize situations encountered. This paper says "here is the Proc SQL process and here are rules for its behavior". The “process and rule” based learning is faster and a more precise learning method. The major deliverable of this paper is that it offers a graphical representation of the SQL process and some rules for describing/predicting the SQL process.

The internal process described below is not in any documentation and likely will remain so. While this process has been judged to be a close approximation of the current Proc SQL process by knowledgeable SAS employees, it is not guaranteed to be carried into future versions of Proc SQL.

INTRODUCTION

George Box, an industrial statistician, is credited with the saying, “All models are wrong. Some are useful.” This first part of this statement surely applies to the model described in this paper. Hopefully the second statement is also true.

Dr. G. Box says

“All models are wrong. Some are useful.....

I especially worry about any model from that comes from Russ.”

Actually, two SAS employees have informally judged these processes and rules to be *close approximations* of the processes of Proc SQL. However; these processes have been left undocumented so that SAS Institute can be free to improve SQL without notice to the SAS community.

This paper lays out processes for a Proc SQL Query and also lays out rules that will be useful in explaining Proc SQL summary functions and sub-queries to beginning Proc SQL students. Without an explanation of the internal process, it impossible to create rules predicting behavior. Without rules of behavior, skill growth comes from repeating examples until situations are remembered. This “repeat and repeat” process is poor pedagogy.

THE DATA SET USED

The data set most commonly used in the following examples is the one shown to the right. It looks much like SAShelp.class, but is smaller so that it can more easily fit into small boxes on PowerPoint slides.

The data set MySchool

Name	Sex	Age	Height	Weight
Joy	F	11	51.3	50.5
Jane	F	12	59.8	84.5
Jim	M	12	57.3	83.0
Alice	F	13	56.5	84.0
Jeff	M	13	62.5	84.0
Bob	M	14	64.2	90.0
Philip	M	16	72.0	150.0

THE OPTIMIZER AND THE DATA ENGINE

SQL has no “magic techniques” and writes programs in a manner very similar to that of a base SAS programmer. The “program writing process” is managed by the SQL Optimzer, a powerful subroutine that writes programs (complete with merges, index use and hash table creation) to produce the results you request via SQL syntax.

As an illustration, of how SQL works like you do. While it is often said that SQL can merge unsorted data, that is not quite true. The SQL Optimzer has the same merging techniques at its disposal as you do, and some techniques do not require sorting (like a hash table merge or an IORC merge). If SQL can not use one of these techniques, it often sorts files before merging. The Optimizer does not ask for permission to sort nor does it notify the programmer that it did sort. The Optimizers independence of action has contributed to the belief that SQL merges without sorting.

The SQL Optimizer applies good programming practice to your queries. The Optimizer knows that small files process faster than large files and is *always* trying to reduce file size (rows and columns). The SQL Optimizer only reads in *required* rows and variables. It also attempts to reduce rows and columns in SQL working files *as soon as it can*. The Optimizer will make several attempts to remove rows and variables as it processes a single SQL query.

The Optimizer has a helper; subroutine called the data engine and we need to know a bit about that subroutine.

Figure 1 shows the familiar screens we see (our look and feel) when we see any SAS installation.

SAS has created this common "look and feel" by making the code for the windows (and SAS calculations) common to all implementations and coding "data engines" to allow the common user interface and statistical code to communicate with different hardware and operating systems.

SAS calls this design, Multi-Vendor Architecture.

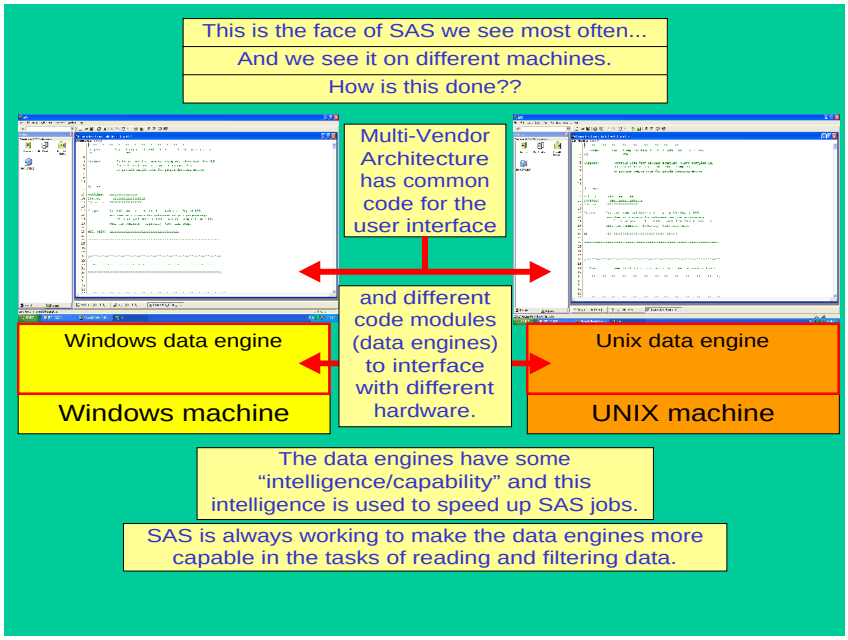

FIGURE 1

Data engines have some intelligence and SQL delegates as many tasks as it can to the data engine. It takes time to move observations (electrons) from the hard drive to SQL. SQL tries to delegate observation filtering and index management to the data engine, close to the hard drive, so fewer electrons are moved and the query runs faster.

THE INTERNAL PROCESS OF SQL- DETAILS OF FIGURE 2

Figure 2 describes the details of the SQL internal process for the code shown immediately below

```
Proc SQL;
select Name, count(*), Height, avg(height), weight/2.2 as Wt_KG
 from MyClass
 where sex="M" and calculated Wt_KG gt 38
 group by sex
 having substr(name,1, 2) NE "Bo"  and Height/avg(height) Gt .95;;
```

The SQL code is shown again in the upper right hand corner of Figure 2 and the query result (one line, for Phillip) is shown in the bottom right hand corner. The final result has one line; but that on that one line the variable "count" has a value of three and shows an average of 66.23. A major task of this paper is to explain how these numbers were calculated.

While Figure 2 is complete, it is admittedly complicated. The reader might plan to skim figure 2 and learn the material by reading the simpler examples in Figures 3, 4 and 5. One could then return to Figure 2 as a review of the material.

In Figure 2, the first SQL task is to find, and read, the data. These tasks are delegated to the date engine. The data engine can handle simple filtering of the data and the keeping of observations with sex ="M" (filtering out other values) was delegated to the data engine. The processing of observations with sex ="F" is stopped very early in the data reading process, in order to save time.

Observations for the four males are read one at a time, and influence/change two SQL working files. Only three variables (Name, sex and age) out of the five variables in the data set are read from the source file (height & weight are not read). The Optimizer knows that small files process faster and implements good programming practices to keep files small by dropping rows and columns of data.

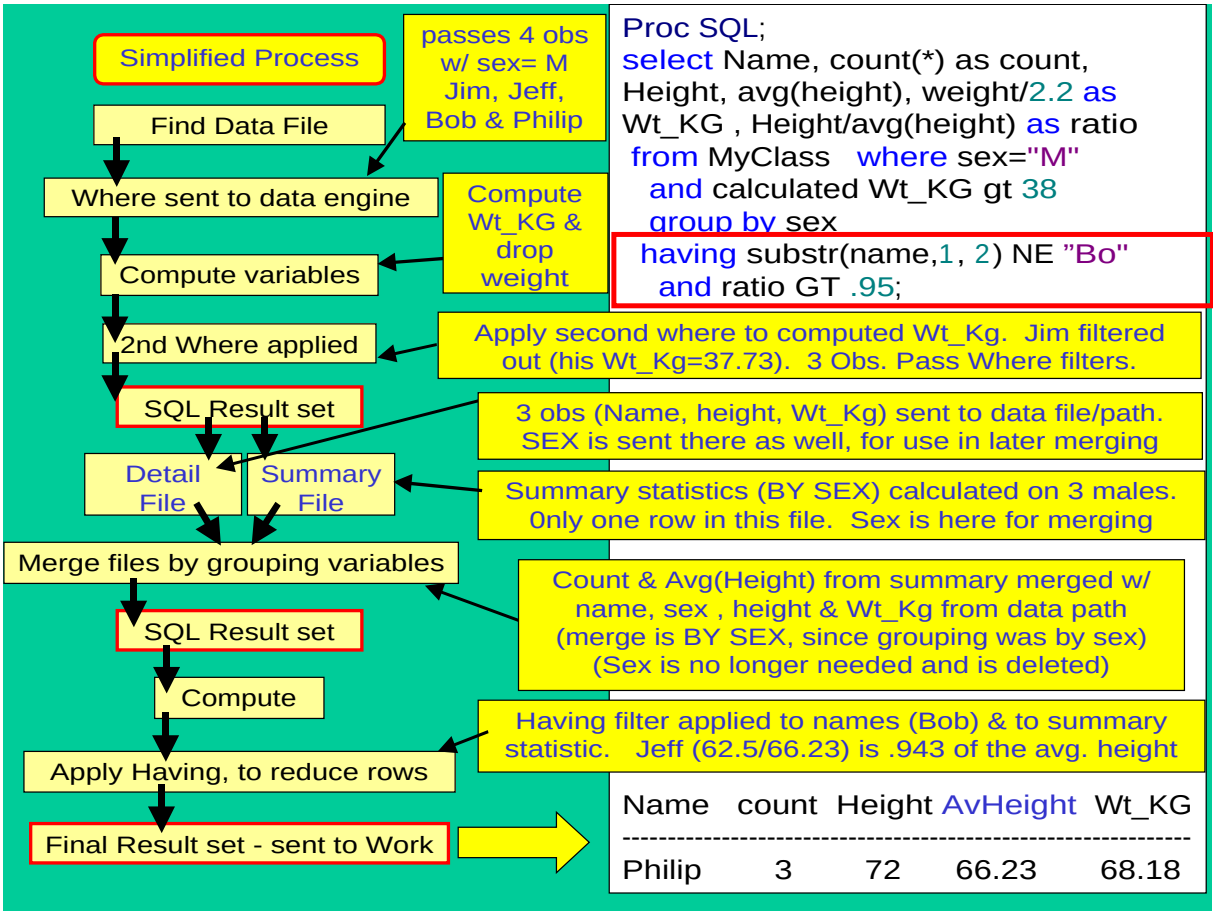

FIGURE 2

The data engine does not have the ability to multiply and so the second part of the "Where filter" (calculated Wt_kg GT 38) can NOT be delegated to the data engine. To apply the Wt_Kg filter, observations are read into SQL, and the Wt_Kg variable is computed by SQL. The "Where filter" logic is then applied. This 2nd "Where filtering" removes one observation, because Jim does not meet the Wt_Kg GT 38 criteria. Remember, since Wt_Kg is not in the source file, Wt_Kg filtering can NOT be delegated to the data engine. Where filtering happens in two places/steps. Note that "Ratio" can not be calculated at this time, because we do not have a value for average(height).

The next SQL step is to calculate any summary statistics. We say that summary statistics are calculated using all observations that have passed through the "Where clause" filtering. This next step, in SQL processing, involves creating two working files, one to hold "detail data" and one to calculate statistics.

In this query we want "detail information" (defined as data read from the source file e.g. the variables name and height) merged in with summary statistics (defined as numbers that we calculate across rows e.g. count and mean). If you were to program this query using data step programming, you might save the detail data in a work file, compute summary statistics into a "summary file" and then merge the two files. SQL has no magic processes or capabilities and produces results using a program very similar to what you might write. Remember, every observation passing the "Where filtering" has the ability to affect the two SQL working files (the detail file and summary file) that are shown in Figure 2.

When detail data (non-summary variables) are requested via the Select clause, the detail file is "activated/created" to hold data. After the "Where filtering", detail data (observations that are needed in final output or to process the rest of the clause, or to merge) are held in the detail file.

When summary statistics are requested in either the Select clause, or the Having clause, the summary file is activated/created. No matter if the summary statistic is coded in a Select, or Having, clause SQL uses all observations that pass the two "Where filters" to create the summary statistics. Note that the count value, shown in Figure 2, has a value of 3, even though there is only one observation in the final output. Three observations (Jeff, Bob & Philip) passed the two "Where filters" and were counted as they "entered" the summary path (two of these observations were filtered out later). SQL often brings variables not mentioned in the Select clause into the detail and

summary files, so that it can properly update rows in the summary files and merge the two files.
In the example above, sex is required in the summary file so that the statistics can be calculated and so that the merge of the data and summary files can be performed. An important rule of Proc SQL is that the summary file will have one row for every level of the grouping variable(s).

The next step is a merge of the data and summary files by the grouping variable(s). This merge produces a SQL result set that contains the detail data *and* the summary statistics. Importantly, statistics were calculated for each level of the grouping variables. Accordingly, the detail and summary files are merged by the grouping variables. Since a "merge by sex" is to be performed, the summary file contains three variables, count, avg(age) *and sex*.

Next SQL computes variables, like ratio, requiring summary statistics as part of the calculation. These calculations can only be done after the merging of the detail and summary result files. SQL operates as if the detail and summary files were merged and then observations pass, one at a time, through something like a "Having P.D.V".

The next task is to apply the "Having filter" to remove observations. Often, *but not always*, "Having filtering" involves summary statistics. Typically, books say "Having is applied to summary information" but the process in Figure 2 (filtering on name- which is detail data) suggests a need to change this phrasing. A more accurate explanation is: A "Having filter" removes observations late in the SQL process- when summary statistics are available.

Part of the "Having filter" in Figure 2 (the `substr(name,1,2) NE "Bo"`) is applied to *detail information* (the name variable) and not to summary information. The "Having filter" can be applied to summary statistics **and/or** detail data and it is for this reason why a new explanation of the Having filtering is suggested in this paper.

That this paper suggests that we use the rule "Having filters out observations (much like a Where clause does) but removes observations *late* in the SQL process – after summary statistics have been calculated and are available". Having and Where both filter out observations, and are very similar, except for their position in the SQL process. Their position in the SQL process allows them to access different data elements (Having can filter on summary statistics, where can not).

The final step in the SQL process is to sort the data (if required) and change the file from an SQL internal file to a file in the work library.

The example in Figure 2 is accurate, and good for people who understand the SQL process, but it is considered excessively complex to help a beginner in learning SQL. Additionally it is hard to make graphics for this process.

Since the process above has so many steps that it is difficult to show in a graphic and difficult to learn, the process will be collapsed to an eight-step process that is easy to represent on a slide.

The steps are: (and are used in the figures below,)

1) **From**: get the data from the source file – Where filtering of observations may be passed to the data engine.
2) Calculate and select: calculate new variables – remove variables not needed
3) **Where** filtering of observations to reduce the size of the working files
4) Create detail and summary files – summary statistics are calculated on all observations that pass through the Where filter. There is a row in the summary file for every level of the grouping variable(s)
5) Merge the files by the grouping variables – remove variables no longer needed
6) Calculate variables having summary statistics in their formula. Apply the having filter to remove observations.
7) **Order** (sort) observations, if required
8) Send the Proc SQL internal file to a file in a permanent or work library

The major deliverable of this paper is that it offers graphical representations of the SQL process and some rules for describing/predicting the SQL process. The rules are shown in boxes in the lower right hand corner of Figures 3-3 and 4-3.

Frankly, a simple reading of the rules is often confusing. It seems that the rules make little sense unless the reader considers them as a description of what happens in the graphic model presented later in this paper. The simplified graphic, shown in Figures 3, 4 and 5 is complete enough to be used in understanding the rules.

The paper starts with simple examples, illustrating only a few of the rules, and proceeds to more complex examples illustrating more rules.

In the three slides of Figure 3 we can see how SQL uses internal files to create a report with detail data

Figure 3-1 shows the first observation being processed.

The optimizer drops variables as soon as it can (keeping only name and sex) and also drops observations as soon as it can.

Observations passing the Where filter can affect both the detail and summary paths. There is no Where filter in this example.

This code does not call for the creation of a summary path.

It is useful to think of the boxes labeled “Calculate and Select” and “Where” as if SQL had program data vectors in which operations you code are applied to one observation from the data set. One at a time, observations pass through the imaginary “SQL PDVs”.

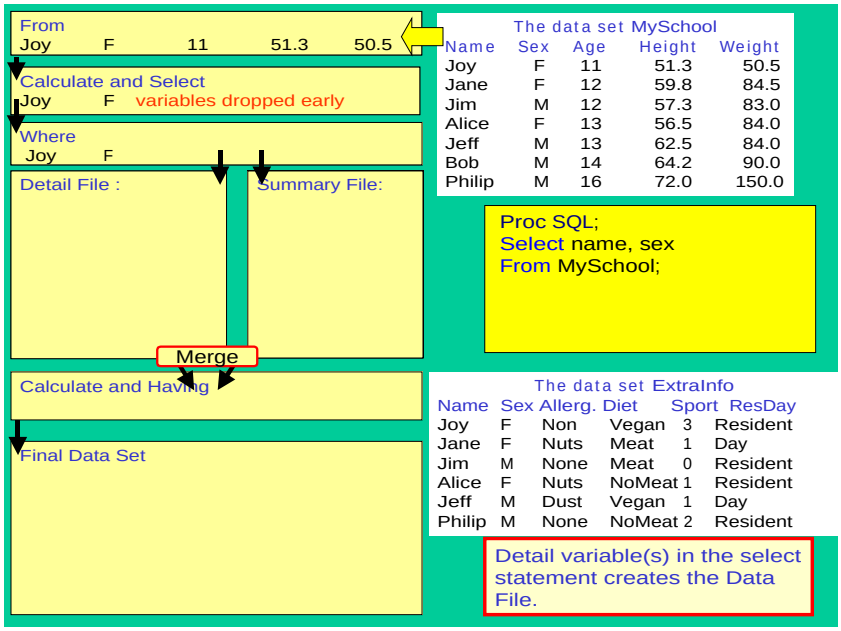

FIGURE 3-1

Figure 3-2 shows the SQL working files after several observations have been processed.

Please read the figure as if it were a strobe picture of the process. It shows several steps in the processing of the observation for Alice. Think of the observation coming into a “From” PDV and then moving to a “Calculate and Select PDV” and then to a “Where” PDV. If the observation is not filtered out by the Where, it can affect both the detail and summary files (if they are created)

One observation at a time, the source file is read into the detail file. All observations are processed into the detail file before any are evaluated by te “Calculate and Having” step

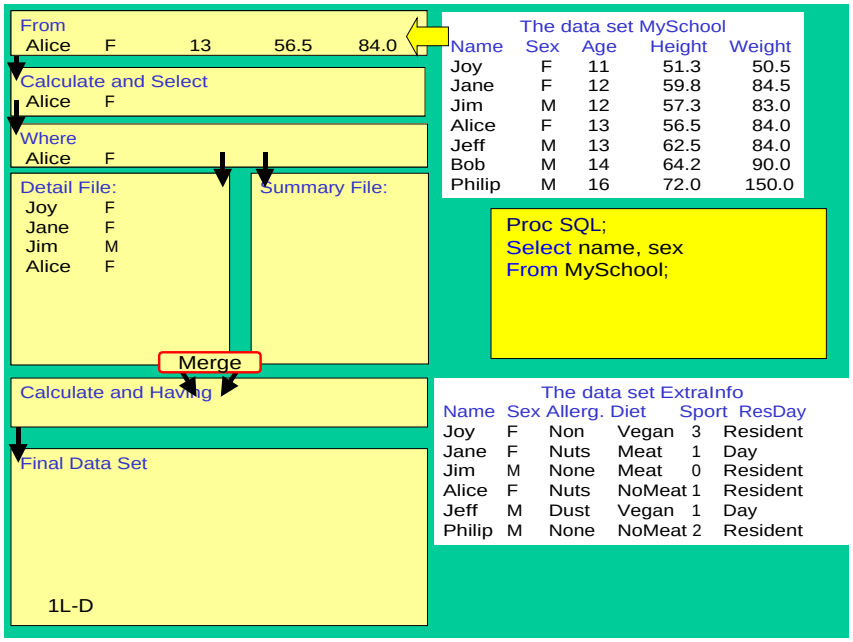

FIGURE 3-2

Eventually, the detail file will contain name and sex information for every observation in the table MySchool. At this point, observations will flow from the Detail file through the “Calculate and Having” (and any sorting) to the Final data set.

Figure 3-3 skips forward a few steps and shows the final result of the query.

All observations from MySchool were, at one time, stored in the detail file. After all the observations from MySchool had been stored in the detail file, observations from the detail file were sent through the "Calculate new variables and Having filter" step.

Observations getting through the Having filter are sorted (if the query calls for sorting) and sent to the output data set. There are no Having filter or sorting instructions, in this code.

The "suggested" SQL rules are shown in the box in the lower right hand corner. Later examples will illustrate more of these rules.

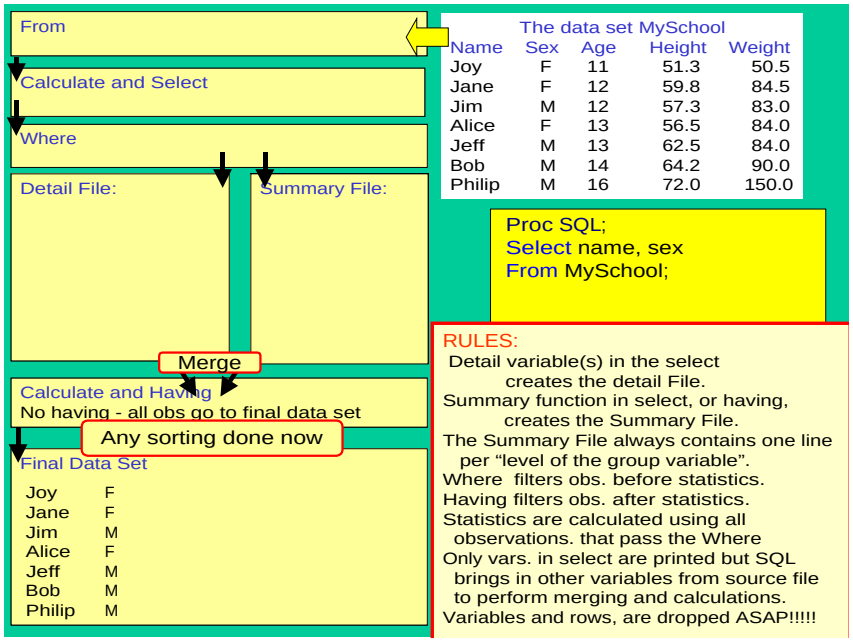

FIGURE 3-3

In the three slides of Figure 4 we can see how SQL uses internal files to create a report with summary data.

Figure 4-1 shows the first observation being processed through several steps and affecting statistics in the summary file.

One at a time, observations pass through the "SQL PDVs".

SQL drops variables as soon as it can and quickly filters out observations in the Where step (no Where in this example).

Observations passing the Where filter can affect both the detail and summary files.

This code does not call for the creation of a detail file. All observations passing the Where filter will affect the statistics being calculated in the summary file.

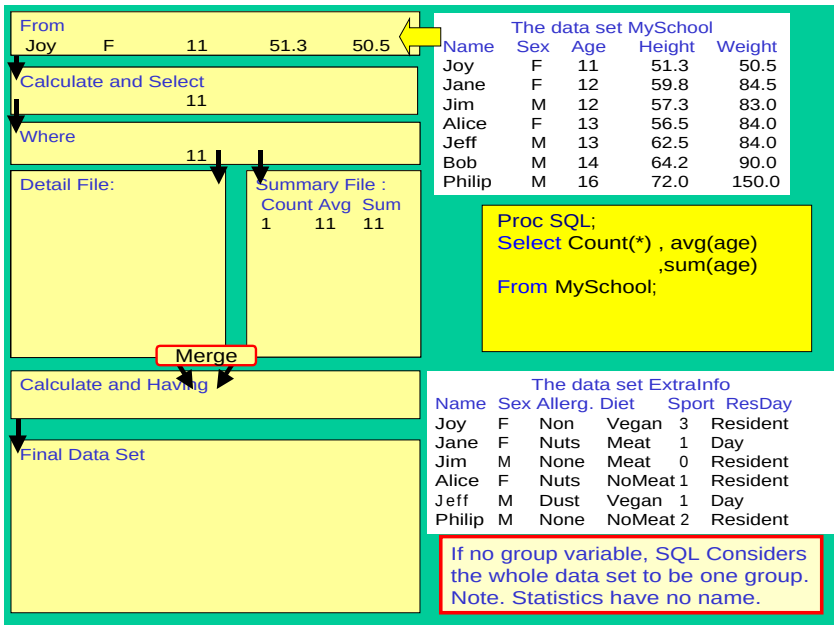

FIGURE 4-1

Figure 4-2 shows the SQL working files after several observations have been processed.

The source file is read into the summary file.

Statistics are calculated on all observations that get through the Where filter.

Again, as if this were a strobe picture, Bob's observation is shown in several steps in the process. Bob is the sixth observation in the data set MySchool and has already affected the summary file.

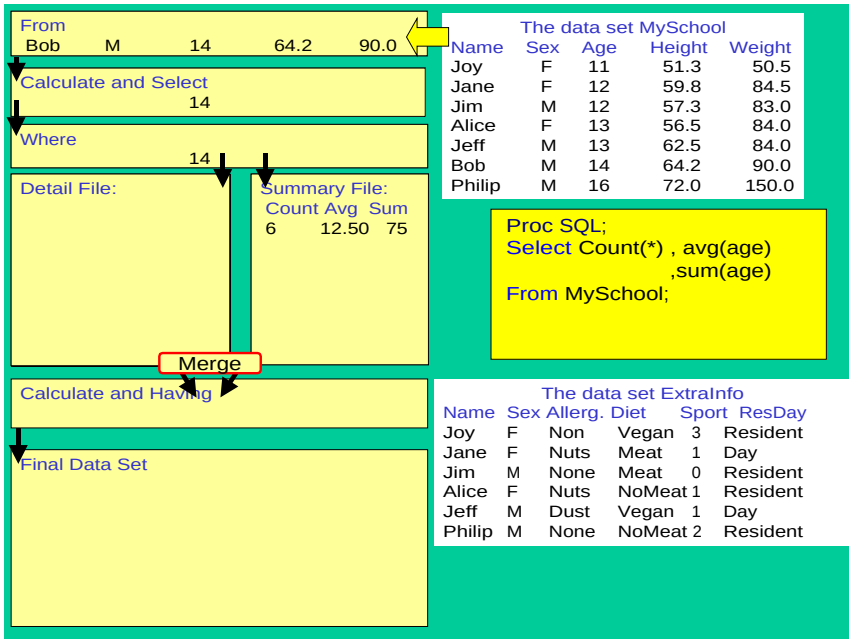

FIGURE 4-2

In figure 4-3 we see the final result of the query.

Since there was no Where filter, all observations from MySchool affected the totals in the summary file.

When all observations had been processed, the summary file was sent through the "Calculate new variables" and "Having filter" steps.

Observations getting through the Having filter are sent to the output data set.

The "suggested" SQL rules are shown in the box in the lower right hand corner.

Later examples will illustrate more of these rules.

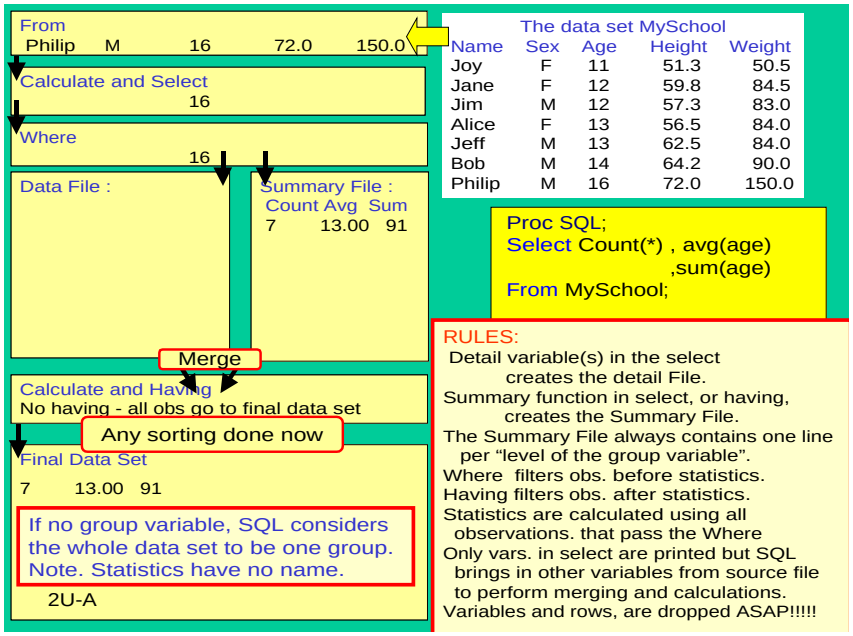

FIGURE 4-3

The five slides that make up figure 5 show a query that activates both the detail and summary files and uses both the Where and Having filters to filter out observations. It is helpful to think of the boxes labeled “Calculate and Select” and “Where” as if SQL had a program data vector, or several PDVs.

Figure 5 –1 shows an observation not getting through the Where filter. Joy is only 11 and gets filtered out by the Where. She does not affect the detail file or the statistics in the summary file.

This illustrates that the Where filters observations before summary statistics are calculated. In this example, counts will total less than the number of observations in the source data set.

The proposed rule is: Statistics are calculated using all observations that pass the “Where filtering”.

This rule can be restated as: only observations getting through the Where filter affect the summary statistics.

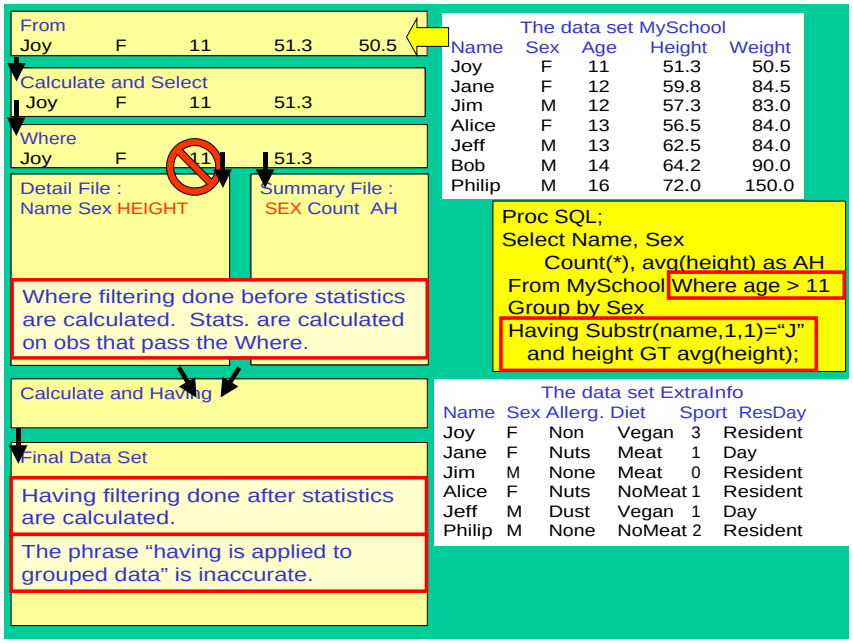

FIGURE 5 -1

Figure 5-2 shows the system after processing most of the file MySchool. Note that weight was dropped ASAP and that we see a stroboscopic view of the processing of Bob’s observation. The “Bob observation” is shown in the From, the Calculate and Having PDVs and has caused changes in the two internal files.

ALL the observations that got through the Where filter have been stored in the detail file.

ALL observations that got thorough the Where filter have been used to calculate the statistics in the summary file.

Our rule said: The summary file always contains one line per “level of the group variable”. We have a line for each level of sex.

Sex is in the summary file so that incoming observations can update the proper summary line (M or F) and for the merging that follows.

Note that height is not needed in the final output but is needed later for the Having filtering. The Optimizer, on it’s own, pulls height into the detail file.

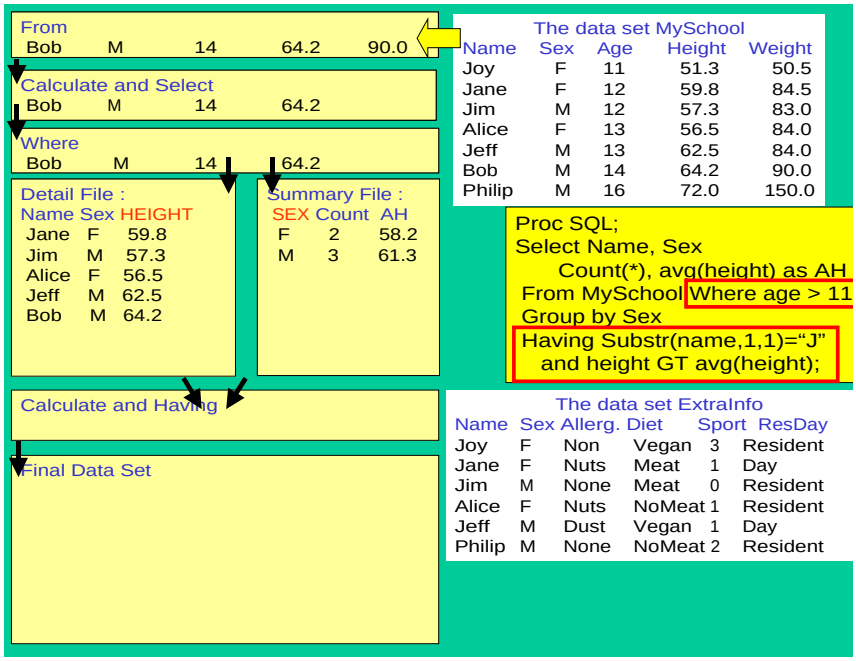

FIGURE 5 –2

Figure 5-3 shows several steps on one slide.

Figure 5-3 shows the merging of the detail and summary files. The merge is done by sex

SQL calculates statistics for each level of the grouping variables and then merges the detail and summary files by the grouping variables.

In this example SQL will perform a many to one merge.

This graphic also shows that Jane passes the two "Having filter criteria" and is sent to the final data set.

The Having filter removes observations but does so after summary statistics are calculated and the detail and summary files are merged.

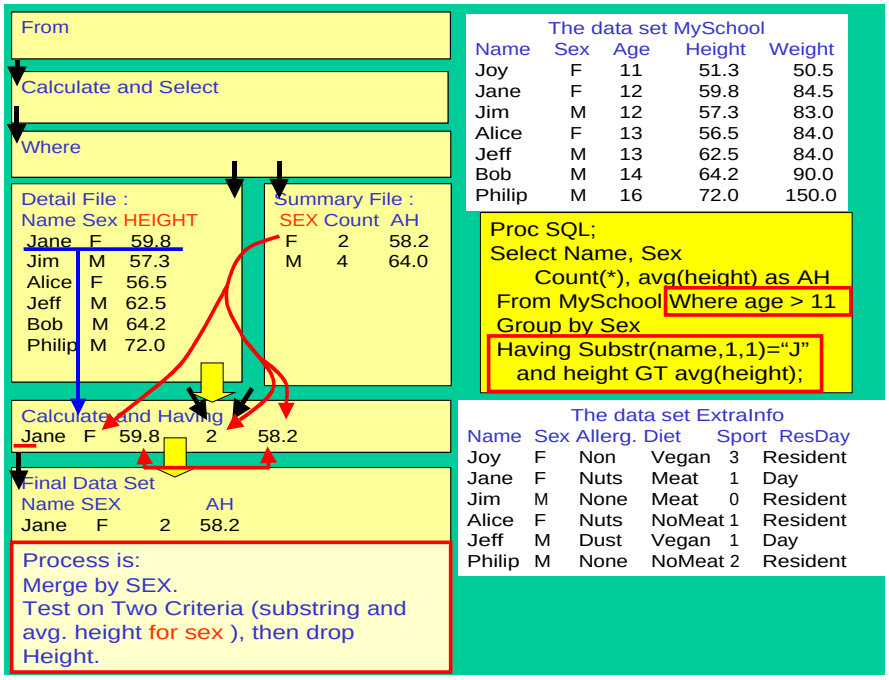

FIGURE 5 -3

Figure 5-4 shows several steps on one slide.

Figure 5-4 shows the merging of the detail and summary files, by sex, and shows Jim **not** passing the Having filtering.

For an observation to be sent to the final data set it must the meet the two criteria specified in the Having clause.

We see that a Having, like a Where, filters out observations, but Having filters observations AFTER statistics are calculated and the working files are merged.

The Having filtering removes observations but does NOT change the values of summary statistics.

The summary file has one row for every level of the grouping variables (every level of sex).

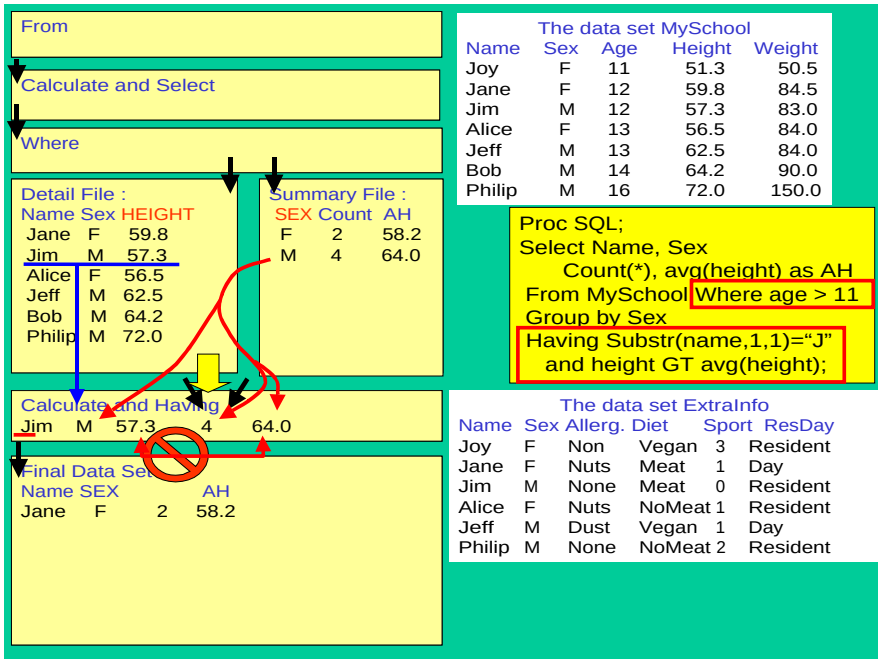

FIGURE 5 – 4

Figure 5-5 shows several steps on one slide. Figure 5-5 shows the merging of the data and summary files, by sex, and shows Philip **not** passing the Having filtering.

Figure 5-5 also shows the result of the query. The output file has just one line of data, with a count of 2.

Hopefully, this example shows how SQL can produce an output file with *one* observation and a count on that line that is different from *one*.

There is enough information on this file to understand all the rules. Enough material has been presented so that the complicated graphic in Figure 2 can be reviewed, and understood.

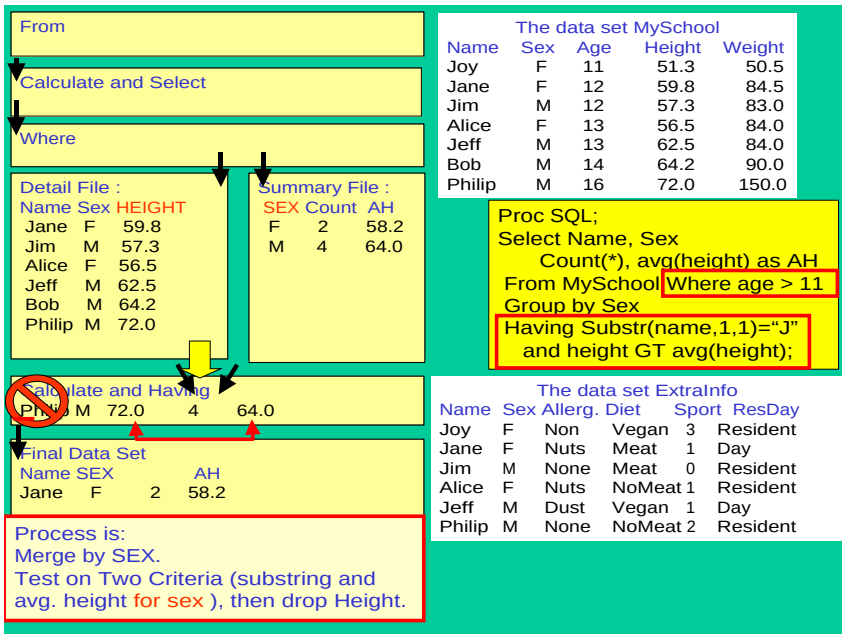

FIGURE 5 –5

RULES THAT CAN BE ABSTRACTED FROM THE ABOVE PROCESSES

We have now seen enough of this model of SQL internals to create some rules for SQL coders.

1) Classify variables as data (e.g. select *name, age, sex*) or summary (e.g. select *count(*), sum(age), max(age)*) .

2) Mentioning a detail variable in the Select clause triggers the creation of the detail working file. Mentioning a summary statistic (count, sum, mean, min, max etc.) in the Select, or Having clause, triggers the creation of the summary working file. Summary functions are restricted to the Select and Having clauses.

3) The summary file has one line of data for every level of the grouping variable(s). SQL **ALWAYS** groups the data when it calculates a summary statistic. Importantly, if there is no "group by" clause, SQL considers the whole file one group and creates summary statistics on all observations that get through the Where filter.

It is helpful to think of both the Having and the Where as "observation filters".

Where filtering can happen at more than one place, but always happens before any summary statistics are calculated. **Where filtering removes observations early in the process and affects summary statistics**. Having filters observations after the detail and the summary files have been merged.

A Having removes observations late in the SQL process and does not change summary statistics.

PROC SQL AND THE ANSI STANDARD

It should be noted that that the "split path" process described in this paper is an ***extra feature*** in SAS Proc SQL and is not required by the standards body for SQL. Below we compare the code for SAS Proc SQL and an ANSI compliant syntax. Note that using the typical SAS Proc SQL code causes the production of the well know "remerging summary statistics" note as the data and summary files are merged.

Typical SAS Proc SQL code	ANSI standard SQL code
Proc SQL number; select name, age, avg(age) from MyClass Group by sex;	**Proc SQL** number; Select L.name, l.sex,l.age ,AvAge From myclass as L Left join (select sex, avg(age) as AvAge from Myclass group by sex) as r on L.sex = r.sex;
7 Proc SQL number; 8 select name, age, avg(age) 9 from MyClass 10 Group by sex; NOTE: The query requires remerging summary statistics back with the original data. 11 quit; NOTE: PROCEDURE SQL used (Total process time): real time x.xx seconds cpu time t.tt seconds	12 Proc SQL number; 13 select L.name, l.sex,l.age ,AvAge 14 from myclass as L 15 left join 16 (select sex, avg(age) as AvAge 17 from Myclass group by sex) as r 18 on L.sex = r.sex; 19 NOTE: PROCEDURE SQL used (Total process time): real time y.yy seconds cpu time t.tt seconds

Figure 6

CORRELATED AND UNCORELATED SUB-QUERIES

Figure 7-1 shows an un-correlated query.

The sub-query runs once, before the outer query and returns the sex of the oldest person in the class

When SQL starts to process the sub-query it makes one full pass through the data to find the sex of the oldest person and stores that person's sex in a small result set.

When processing the "main query Where logic", SQL refers back to the stored result set that was returned by the sub-query.

Running a sub-query takes time and is to be avoided.

Storing the result of a sub-query can allow SQL to not re-run a sub-query.

(Apologies are offered for the use of data dependent queries. This query works because there were no ties for oldest person. While bad practice, this was outweighed by the desire for a small/familiar data set and a simple query).

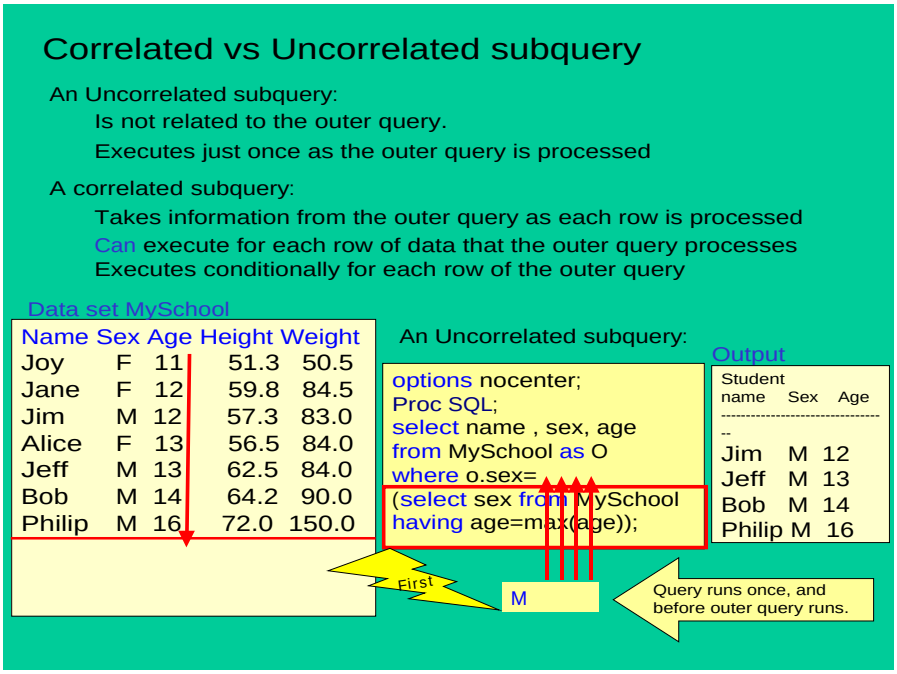

FIGURE 7-1

An important point is that un-correlated queries only run once and, thus, are efficient. The results of a sub-query is stored in a SQL internal, or working file.

Figure 7-2 shows a very abbreviated graphical depiction of a correlated query. The process and the graphic used in Figures 2 through 5 are still appropriate, but the many steps involved are difficult to represent on paper.

When SQL processes Joy, it has Joy's information in the outer SQL "PDV". Joy is Female and SQL has the sub-query make a full pass through the data set to find the height of the oldest female(59.8) SQL stores the answer in a result set that is both ordered and indexed (overkill for just a 1 row x1 column result). Full passes through the data set are expensive and are to be avoided. SQL will save this result set and use it to avoid future "full scans", if logic allows.

SQL then makes the Where comparison between Joy's age and the age in the result set.

When processing Jane, SQL checks to see if it can avoid a full scan. SQL checks to see if it can perform the Where filter comparison using the temporary ordered, indexed result set.

For Jane (sex=F), SQL can perform the Where filtering using the information stored in the result set. It does so, and does not make another full pass through the data set.

When SQL processes Jim (sex=M), the outer query tries to make the Where comparison using information in the temporary ordered, indexed result set.

It can NOT and is forced to call the sub-query, pass it the value of sex ="M" and make another pass through the data set.

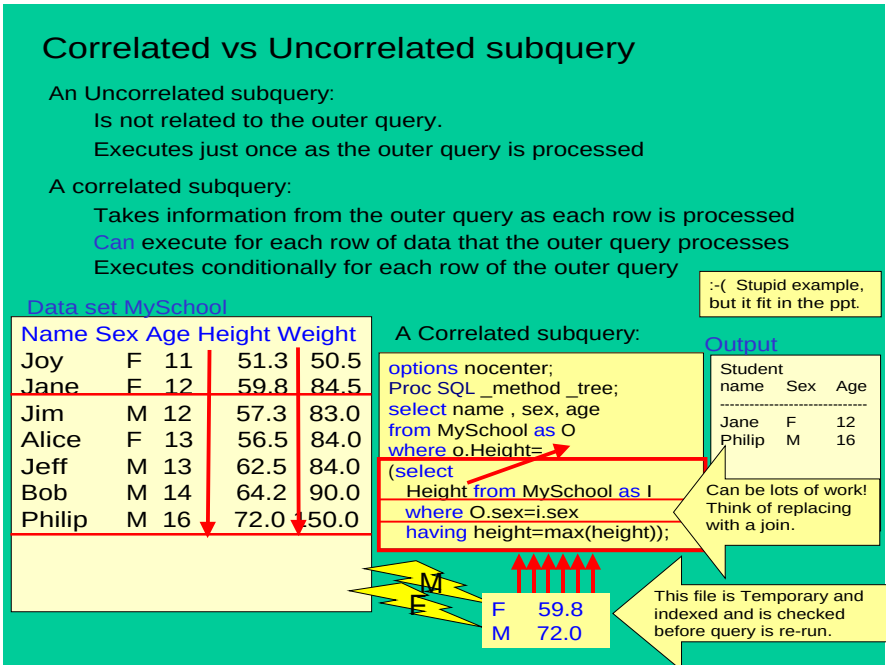

FIGURE 7-2

The information from the second run (sex=M) of the sub-query is added to the temporary, ordered, indexed result set. Correlated sub-queries can run many times and this can make for long run times. In this example, the sub-query runs once for each value of sex. Many books suggest re-coding a correlated sub-query into a join as a technique for speeding up queries.

Sub-queries are a confusing process and a second example might be helpful. Figure 7-3 uses the graphic developed in Figures 3 to 5 to illustrate a way of thinking about the details of correlated sub-queries. The query in Figure 7-3 accesses two files and the code is shown immediately below:

```
Proc SQL;
select name , sex, age
from MySchool as O
where exists
  (select * from ExtraInfo as I
    having  I.age   =   O.age
    and       I.sex  NE O.Sex)  ;
```

The sub-query is in the Where clause and an explanation of the graphic in Figure 7-3 follows.

Joy is read into the SQL outer query "PDV" and Proc SQL needs to determine if Joy's observation should be deleted by the Where clause of be passed on to the data file. The values of 11 as o.age and F as O.sex are passed from the outer query PDV to the inner query.

The inner query now runs, with a Having clause that evaluates to Having I.age = 11 and I.sex="F".

The Inner query performs a full pass through the data set ExtraInfo to find the answer for eleven year old females.

The relationship between the values in the outer query PDV and the result of the query is "Exists". Exists is an odd form of relationship and will be fully discussed later.

The Exists in this Where clause means that Joy will not be filtered out if the sub-query returns anything. The observation will not be deleted if a non-null return from the SQL sib-query *exists*.

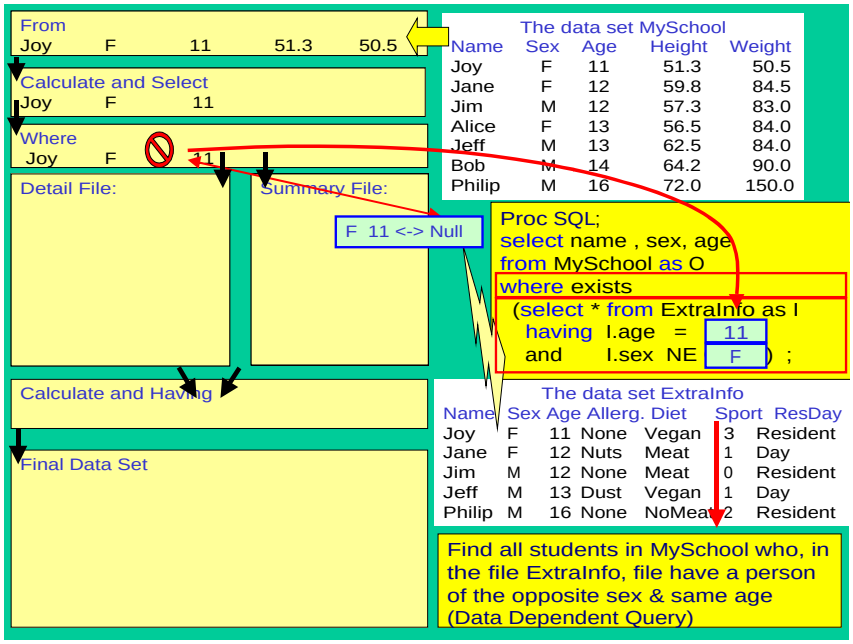

FIGURE 7-3

SQL stores the answer to the sub-query in a temporary, ordered and indexed working file. The Where clause first checks the temporary, sorted and indexed file to see if the observation should be deleted.

If the data set MySchool contains another eleven year old girl, SQL will be able to evaluate the Where clause by checking the sorted and indexed working file, instead of making a pass through the data set ExtraInfo.

An important point is that correlated queries can run many times and can be inefficient. The results of a sub-query is stored in a SQL internal, or working file and this file is checked by the Where clause to automatically avoid unnecessary re-running of the correlated sub-query. Consider replacing a correlated wub query with a query and a join.

SUB-QUERIES IN DIFFERENT PARTS OF THE SQL QUERY

Above, we showed an un-correlated and correlated sub-queries filtering observations in the Where clause. Both types of sub-queries can be used in other places in SQL code. We will examine this SQL feature in this section.

The issue to be discussed is the interrelationships among four conditions:

1) Which of the four clauses (From, Select, Where or Having) includes the sub-query
2) The type of the sub-query (correlated or uncorrelated)
2) The shape (row and column) of the result that the sub-query returns
3) For sub-queries in the Where and Having, the relationship (EQ, LE , GT etc.) specified between the value in the main query result "SQL Program data vector" and the result from the sub-query.

Issue three is fairly subtle and examples will be helpful. The next few slides show how sub-queries can, and can not, be placed in different places in the outer SQL query.

Remember that the SQL code looks like:

Proc SQL;
Create **T**able **A**s
Select …..<-*we can code sub-queries here and will examine the process in sections below*
From……<-*we can code sub-queries here and must examine the process in sections below*
Where ….<-*we can code sub-queries here and must examine the process in sections below*
Group by
Having….<-*we can code sub-queries here and must examine the process in sections below*
Order by….

(A way to remember the coding sequence is **C**ontinental **T**ourists **A**lways **S**ay **F**rench **W**omen **G**ive **H**appy **O**rders).

SUB-QUERIES IN THE FROM

From establishes a link to the data source for the SQL query.

SQL accepts files in any shape and a sub-query in the From is allowed to pass any data shape (return a result of any shape (1x1, 1xC , Rx1, RxC)– to SQL without SQL "complaining".

Correlated sub-queries imply a relationship between an "outer" result set and an "inner" result set.

When the From executes, there exists only one data set, and no outer query (no "PDV"), so correlated sub-queries are not allowed in the From.

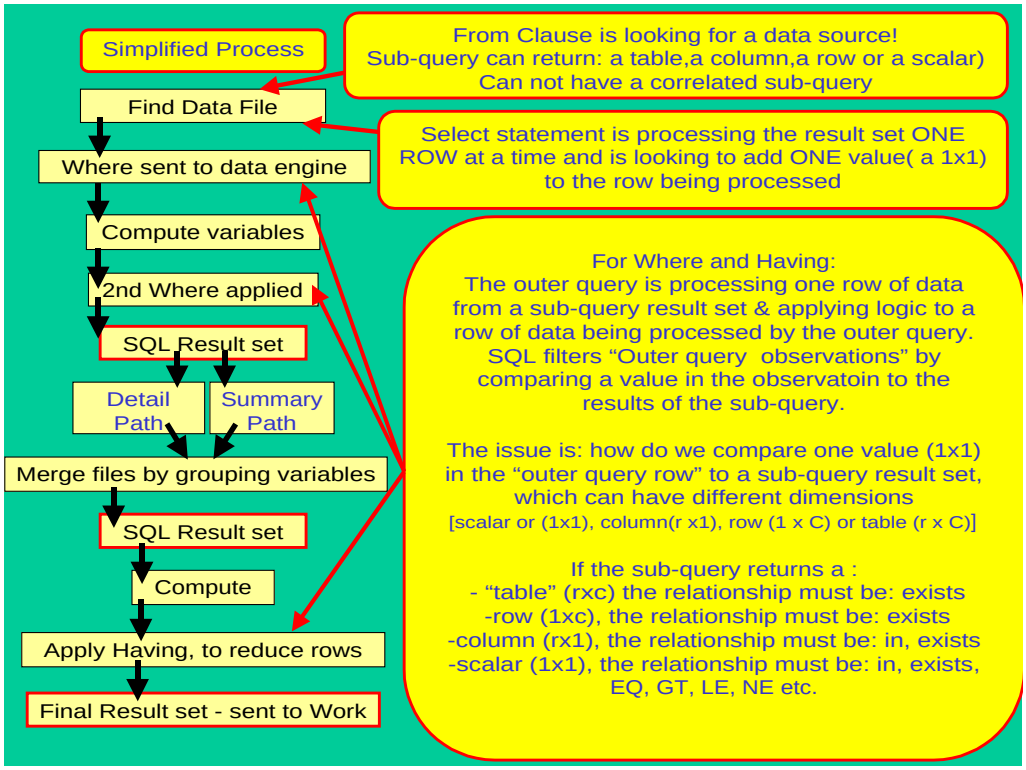

FIGURE 8

SUB-QUERIES IN THE SELECT

Imagine that the Select processes one row of data at a time and uses something like a Program Data Vector. This idea will help, in the future, when thinking of the data shapes that sub-queries are allowed to return to the Select.

An example of "a SQL PDV" is also shown below. Imagine that four sub-queries were written to fill in the blue missing calculated value (SbQryRslt) in the observation below.

Imagine the four sub-queries returned results of different dimensionality, only a 1x1 will "fit" into the observation.

It is useful to think that the result of the query must fit into one variable in the PDV (or into one cell in Excel, if you imagine the PDV as a one line Excel Spreadsheet).

If the sub-query is in the Select clause and the sub-query does not return a 1 by 1 the "returned data structure" simply does not "fit" into the space that the outer query has reserved for it .

Variable name -→	Name	Sex	Sub Query as SbQryRslt	Age
Variable Value -→	Lahong	F	Sub-query result goes here	26

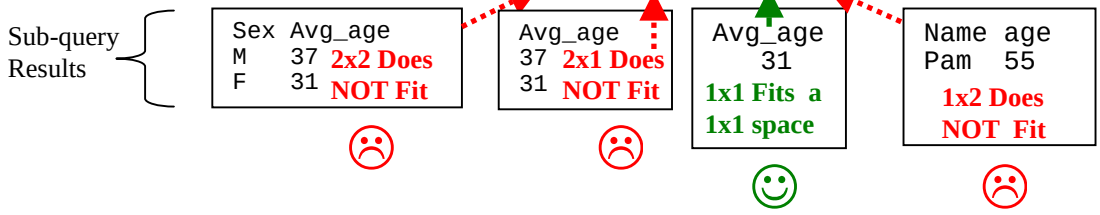

FIGURE 9

Studying Figure 9 helps understand why sub-queries in a Select must return a 1x1.

As the query below shows, Select will accept multiple sub-queries and will accept both correlated and un-correlated sub-queries. Since sub-queries in the Select are simply returning a value to (what I'd really like to call) the SQL PDV, no logical comparisons (no EQ, GE, LE etc.) are needed.

```
Proc SQL; /*pretty silly query, but it illustrates a point and uses MySchool*/
Select O.name , O.age, O.sex  , (select Max(age) from MySchool) as MA
, O.age  -(select avg(age)  from MySchool as I  where O.sex=I.sex )
                              as Diff_from_Sex_avg_age        /*correlated*/

, (select avg(age) from MySchool as I where O.sex=I.sex )
                              as Sex_avg_age                  /*correlated*/

,         (select avg(age) from MySchool as I)
                              as Ovrl_Avgvg_age              /*UN correlated*/
from MySchool as O;
```

Name	age	sex	MA	Diff_from_ Sex_avg_age	Ovrl_Avgvg_ Sex_avg_age	age
Joy	11	F	16	-1	12	13
Jane	12	F	16	0	12	13
Jim	12	M	16	-1.75	13.75	13
Alice	13	F	16	1	12	13
Jeff	13	M	16	-0.75	13.75	13
Bob	14	M	16	0.25	13.75	13
Philip	16	M	16	2.25	13.75	13

FIGURE 10

SUB-QUERIES IN THE WHERE AND THE HAVING CLAUSES

Where and Having both remove observations based on some logical evaluation. They execute as the SQL outer query *is processing a single observation* from a result set (See Figure 5 and Figure 8). Where and Having execute as the SQL outer query processes one observation and is deciding to remove the observation, or not, by ***applying comparison logic between a value in the "outer query PDV" and the results of the sub-query.*** The sub-query can return result sets of different dimensionality (shapes) and the logical form of the comparison is important as the returned shapes differ.

The sub-query can return result sets of different dimensionality and the issue is: how do we compare one value (1x1) in the "outer query PDV" to a ***result set*** from the sub-query, which can have different dimensions and variable types [scalar or (1x1), column(R x1), row (1 x C) or table (R x C) are all allowed].

If the sub-query, in a Where or Having, returns a:

- -Scalar (a 1x1), the relationship can be: =, GT, LE, etc. *as well as* "In" or "Exists"
- -Column (a R X 1), the relationship must be one of two kinds: "In" or "exists"
- -Table (a R x C), the relationship must be: "Exists"
  - Note that the syntax and logic for an Exists comparison is very counterintuitive.
  - Example below:

```
Where exists (select * from InnerDSN as Inner where Inner.age=Outer.age)
```

  - The "Select *" doesn't mean select all variables.
  - The exist comparison does not care what variables are returned.
  - It's a shorthand for "Does the query return anything?"
  - Keep the obs. In the outer query if the sub-query returns anything.
- -Row (a 1 x C) the relationship must be: "Exists".

The above rules are difficult to explain without a graphic and examples, both of which are in the figures to come. These upcoming figures use a modification of the graphic used in Figures 3 to 5. In Figures 11-1 to 11-3 we show several queries in boxes on the right hand side of the figure. Each of these sub-queries returns a different data "shape" (scalar, column, row or table) to the outer query.

The top box contains a query having a sub-query that returns a 1 x 1 shaped data element.
The second from the top box contains a query having a sub-query that returns a column shaped data element.
The third from the top box contains a query having a sub-query that returns a row shaped data element.
The bottom box contains a query having a sub-query that returns a table shaped data element.

SUB-QUERIES IN THE FROM:
The right hand side of this graphic shows four sub-queries in yellow boxes– all the sub-queries are located in the From clause.

In the top box, the sub-query returns a 1x1 to the From (see the white box on the same level as the query to see a representation of what the query returns).

In the second from the top box, the sub-query returns a 4x1 to the From (see the column of white boxes on the same level as the query to see a representation of what the query returns).

The sub-query in the next lower box returns a 1x3 row of data to the outer query. The sub-query in the lowest box returns a 3x5 table of data to the outer query.

These queries run without error.

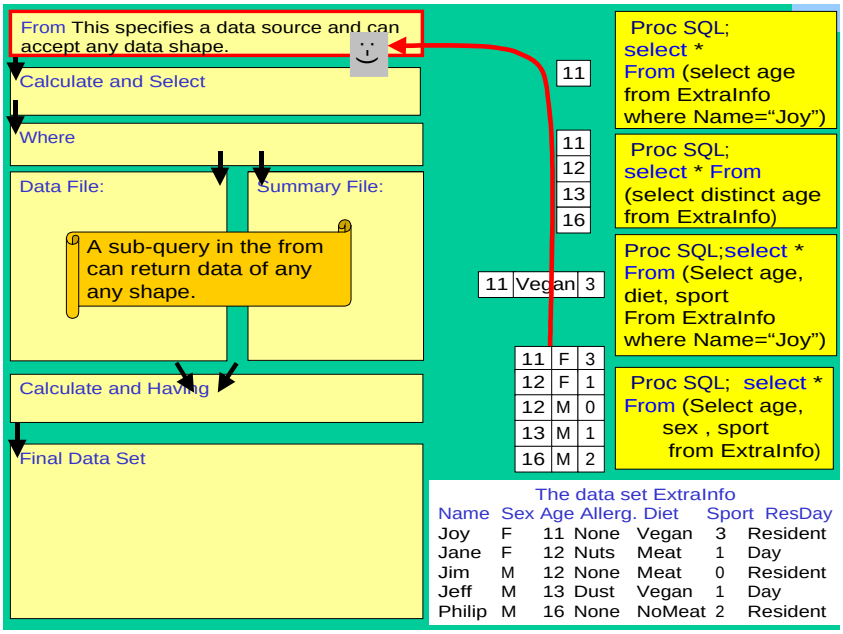

FIGURE 11-1

SUB-QUERIES IN THE SELECT:
Figure 11-2 is a graphic that covers, for a second time, the material presented in Figure 9.

The queries, in boxes on the right hand side, contain sub-queries that return four different data shapes to the Select clause.

Only the query in the top box will run. It has a sub-query that returns a 1x1 "data shape".

When the sub-query is in a Select statement, it must return a data shape that can fit into one cell in the SQL PDV, a 1 x 1 data shape.

As the little smiley face indicates, only one of the queries in the boxes in the right will run, the query that returns a 1x1.

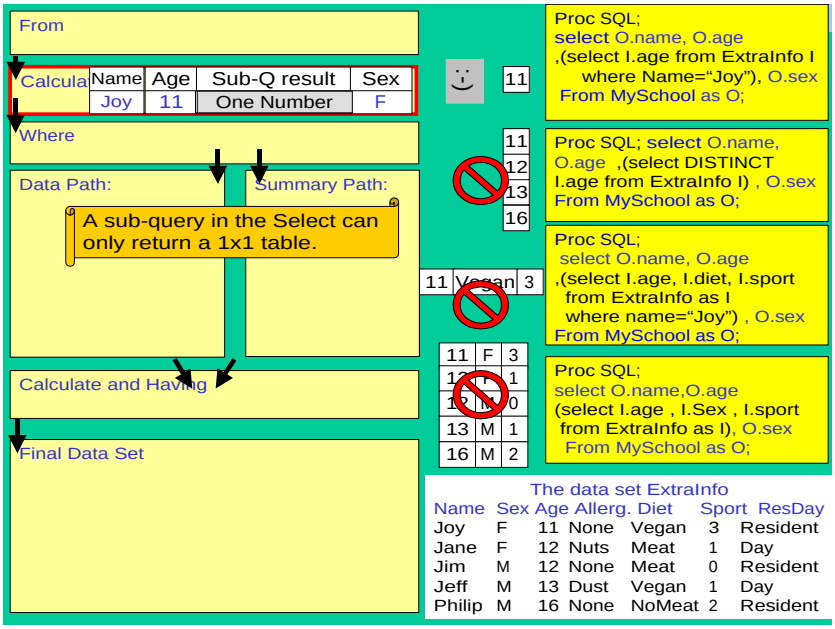

FIGURE 11-2

SUB-QUERIES IN THE WHERE/HAVING:

This figure illustrates some of the complexity of using a sub-query in the Where or Having. These two statements are used to filter out observations based on logical comparisons between the values in the "outer query PDV" and the results of the sub-query.

There are four sub-queries on the right hand side of the figure and we will examine the four in order – this time from bottom to top.

The outer query passes a value, (or values) from the "outer query PDV" to the sub-query. The sub-query returns a data shape to be used in the Where or Having filtering.

This passing of information from the outer query PDV to the sub-query is illustrated by the F in the bottom gold box.

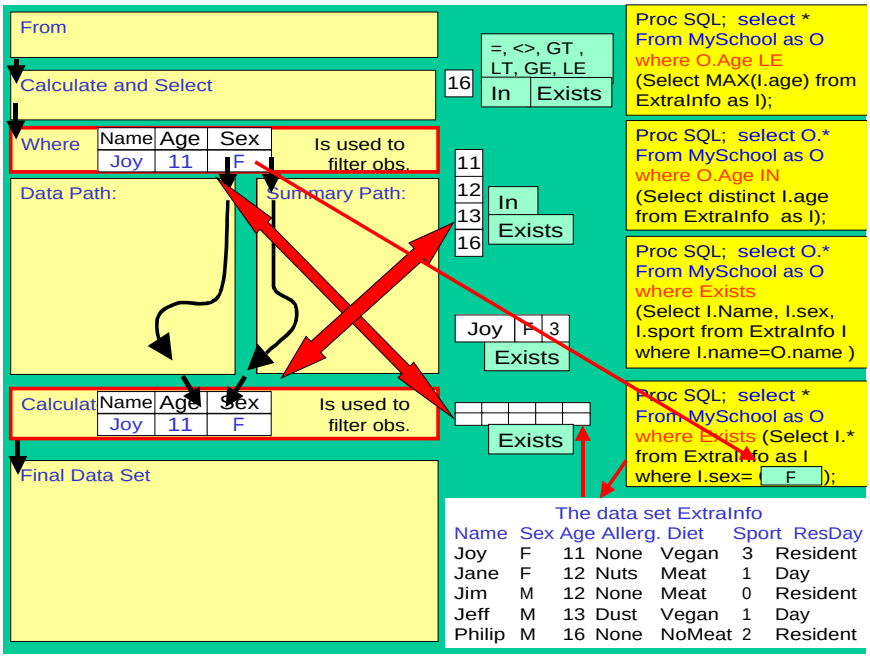

FIGURE 11-3

The query in the bottom gold box has a sub-query that takes values of Sex from the outer query PDV and returns a table shaped data element that the Where uses in a logical comparison. In the table that is returned from the sub-query , there are both numeric and character variables.

Consider, for a moment, how one can logically compare an element in the outer query PDV to a whole table. The answer is that *no real "comparison"* can be made. The only logical relationship that can be tested is if the data that was passed from the outer query PDV, to the sub-query, allowed the sub-query to return ***anything at all***- to return a non-null. The logical relationship that can be tested here is "Exists" or "Does the sub-query return a table at all?"

The query in the second to the bottom golden box returns a row shaped data element. In the row that is returned, there are both numeric and character variables. Consider, for a moment, how one could logically compare one element in the "outer query PDV" to a whole row of data. No real "comparison" can be made. The only logical relationship that can be tested is if the data that was passed from the outer query PDV, to the sub-query, allowed the sub-query to return ***anything at all***- to return a non-null. The logical relationship that can be tested here is "Exists".

The query in the third from the bottom golden box returns a column shaped data element. This data shape will always be of one data type (char or numeric). Consider, for a moment, how one could logically compare one data element in the "outer query PDV" to a column of data. Logically, one could ask if the value of the variable in the outer query PDV is one of the values in the column (is **in** the column). When the sub-query returns a column, an "in" logical comparison can be made between a value in the outer query PDV and a list of values. An "Exists" comparison also makes logical sense.

The query in the top golden box returns a 1 x 1 shaped data element and this data shape is of obviously of one data type. Consider, for a moment, how one can logically compare one data element in the "outer query PDV" to a 1 x 1 data element returned by the sub-query. Logically, one could use many of the SAS relationships (EQ, GE, LE, GT, LT, NE Like) in this situation. One could also use "In" and "Exists".

CONCLUSION

The SQL Optimizer is a very smart sub-routine and implements good programming practice. Observations and variables are dropped as soon as possible.

Summary functions are restricted to the Select and Having Clauses.

Mentioning a data variable in the Select clause triggers the creation of the detail File. Mentioning a summary statistic (count, sum, mean, min, max stc.) in the Select or Having clause triggers the creation of the summary File. SQL ALWAYS groups the data when it calculates a summary statistic. If there is no "Group by" clause, SQL considers the whole file to be one group. There is a row in the summary file for each level of the group variable(s). Detail and summary files are merged by the group variable(s).

It is helps to think of Having and the Where as both filtering observations. Where filtering can happen at more than one place, but always happens before summary statistics are calculated. **Where filtering removes observations and affects the values of summary statistics**. Having filters observations after the detail and summary result sets are merged. **A Having removes observations but does not change values of summary statistics.**

Sub-queries, can be coded in the From, Select, Where and Having clauses but correlated sub-queries can not be coded in the From clause.

Since sub-queries simply *deliver* data to the From and Select clauses (you might imagine that SQL has something like the program data vector), we do not need to establish relationships between results of a sub-query in the From or Select to a variable in the outer query.

Sub-queries in the Where and Having are comparing a 1x1 data element in the outer query PDV with results from the sub-query. The allowed form of the relationship (=, GT, LT, In, Exists, etc.) varies with the shape of the data returned by the sub-query. Specifically:
If the sub-query, in a Where or Having, returns a:

-Scalar (a 1x1), the relationship must be: =, GT, LE, etc. *as well as* "In" or "Exists"
-Column (a R X 1), the relationship must be one of two kinds: "In" or "exists"
-Table (a R x C), the relationship must be: "Exists"
Note that the syntax and logic for an Exists comparison is very counterintuitive.
Example below:

```
Where exists (select * from InnerDSN as Inner where Inner.age=Outer.age)
```

The "Select *" doesn't mean select all variables.
The exist comparison does not care what variables are returned.
It's a shorthand for "Does the query return anything?"
-Row (a 1 x C) the relationship must be: "Exists".

Correlated sub-queries have the potential to execute many times. Consider replacing correlated sub-queries with joins.

ACKNOWLEDGMENTS Thanks to: Lewis Church, Ian Whitlock, Paul Sherman

REFERENCES

Hermansen, Sigurd, 1997, "Ten good Reasons to Learn SAS® Software's SQL Procedure", Proceedings of the Twenty Second Annual SAS Users Group International, paper 35

Hermansen, Sigurd, 2001 "Fuzzy Key Linkage", Proceedings of the Annual South East SAS Users Group

Hermansen , Sigurd , 2002 "Structured Query Language: Logic, Structure, and Syntax "Proceedings of the Annual South East SAS Users Group , paper TU24

Lavery, Russ, 2005 'The SQL Optimizer Project: _Method and _Tree in SAS®9.1"Proceedings of the Thirtieth Annual SAS Users Group International, paper 101

Schrier, Howard, 1991, "Picking Up Where the SQL Optimizer Leaves Off", Proceedings of the Sixteenth Annual SAS Users Group International, paper AT008

Schrier, Howard, 2006, "SQL Set Operators: So handy Venn You Need Them", Proceedings of the Thirty-First Annual SAS Users Group International, paper 31

CONTACT INFORMATION

Russ Lavery Independent Contractor for Numeric Resourcess Russ.Lavery@verizon.net