Paper 1823-2018

Cool SQL Tricks

Russ Lavery, Contractor

ABSTRACT

This paper is a collection of eleven tips and tricks using SAS PROC SQL. It is intended for an intermediate SAS person who would like to take a peek under the hood. Code for the H.O.W. will be put on the web at www.LexJansen.com. That code has the same section organization as this paper and an interested person can get that code and read this paper while executing the code. That process will provide examples of the concepts developed below and, largely, duplicate an H.O.W.attendance.

Topics covered are: 1) the SAS data engine, 2) indexing, 3) the optimizer, 4) correlated and uncorrelated sub-queries, 5) placement of sub-queries in SQL syntax, 6) views, 7) fuzzy merging, 8) coalescing, 9) finding duplicates using SQL, 10) reflexive joins, 11) Using SQL dictionary tables and a macro to document SAS tables . The eleventh section adds some small tweaks to Vince DelGobbo's excellent work on how to create a hyperlinked Excel workbook.

The appendix contains miscellaneous examples of SQL from which a reader can "steal"

INTRODUCTION

The original presentation is like a cartoon with many images that differ from each other by small characteristics. Because this paper is a static representation of a cartoon of how SAS works, it will be longer than the typical paper. The method will be to take screen prints at "important" parts of the process and to link those screen prints with text.

Topics are:

A: 1 OF 11) THE SAS DATA ENGINE **B: 2 OF 11)** INDEXING
C: 3 OF 11) THE SQL OPTIMIZER AND IMPROVING PERFORMANCE
D: 4 OF 11) SUB-QUERIES: CORRELATED AND UNCORRELATED
E: 5 OF 11) PLACEMENT OF SUB-QUERIES IN SQL SYNTAX
F: 6 OF 11) VIEWS **G: 7 OF 11)** FUZZY MERGING **H: 8 OF 11)** COALESCING
I: 9 OF 11) FINDING DUPLICATES **J:10 of 11)** REFLEXIVE JOINS
K: 11 of 11) USE SQL DICTOINARY TABLES TO DOCUMENT DATA SETS IN HYPERLINKED EXCEL WORKBOOK

A: 1 OF 11) THE SAS DATA ENGINE:

No matter what machine you are using, the SAS interface look the same.

When SAS was introduced, many different manufacturers wanted SAS ported to their hardware. SAS wanted this as well.

SAS came up with techniques to efficiently move SAS to new machines and an important technique was the data engine.

90% of the code that runs SAS on a Windows machine is the same as the code on a UNIX machine. A major difference between different SAS installs is the data engine – the part of SAS that talks to the hardware.

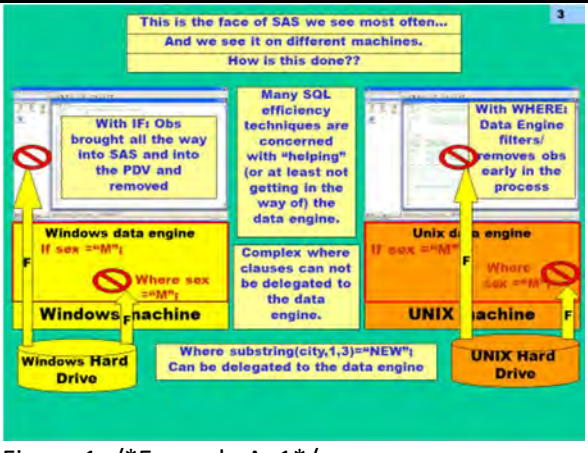


Figure 1: /*Example A_1*/ Section A in SAS code

When SAS started implementing this technique they called it “multi-vendor architecture”. SAS is a layered software package, with a top layer of SAS - talking to lower layers of SAS - talking to lower layers of SAS - talking to the hardware. The program data vector (PDV) is in the highest layer of SAS and is farthest away from the hardware. The data engine is a SAS software layer that is close to the hardware.

We all want fast run times and, while electrons move fast, they do take time to move through the layers of the system. If we can eliminate electron movement, jobs run faster. If SAS can filter observations in a ”lower level” jobs run faster and SAS uses the data engine to filter, data close to the hardware.

You can see the effect of this if you execute the code below. I expect everyone is familiar with SASHELP.class and we will use that data set, to illustrate this point. “If statements” are processed in the program data vector, at the very top of the SAS system. “Where clause processing”, whenever possible, will be delegated to the data engine.

As shown in the figure below, WHERE conditions are applied, by the data engine, before the data enters the input buffer while IF conditions are applied after the data enters the program data vector. If a programmer uses an IF statement, SAS reads 19 observations but only reads 10 observations if the programmer uses a Where clause. Note that clock time measurements are erratic for jobs this small.

```
Data A01_IF_example ;
   /*Reads 19 obs*/
   set sashelp.class ;
   if sex NE "F";
   run;
NOTE: There were 19 observations read from the data set SASHELP.CLASS.
NOTE: The data set WORK._A01_ has 10 observations and 5 variables.

Data A02_Where_Example ;
   set sashelp.class ;
   /*Reads 10 obs*/
   Where sex NE "F";
   run;
NOTE: There were 10 observations read from the data set SASHELP.CLASS.
    WHERE sex not = 'F';
NOTE: The data set WORK._A02_ has 10 observations and 5 variables.
```

Figure 2

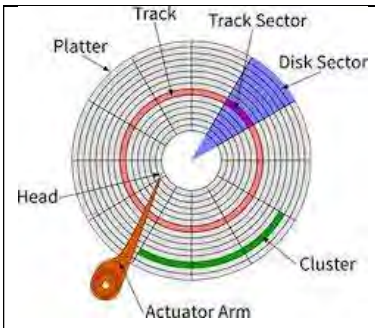
You can, in your code, get in the way of SAS and the data engine. Generally, calculations in a where get in the way of the data engine. Much of the benefits that one can get from using PROC SQL comes from building indexes and then not getting in the way of how the SQL optimizer interacts with the data engine.

At this point, you might want to get the code for this H.O.W. and run the code in Section A- 1 of 11.

B: 2 OF 11) INDEXING:

The where clause part of the SQL optimizer is always looking for the presence of an index and, if one exists, will *consider* using the index to reduce run time. SQL does this evaluation without asking your permission and without even notifying you. If index use reduces run time, the index will be automatically used. Therefore; indexing is important in SQL performance. Surprisingly few programmers build indexes on their data sets or request that IT build indexes on data sets that IT supplies to SAS users. Some review of indexing is in order.

I would recommend that any SQL user read Mike Raithel’s excellent article “Creating and Exploiting SAS® Indexes”. Mike, in a very literal way, wrote the book on SAS indexes. His discussion of how not to get in the way of the where clause optimizer is excellent.



Above, and to the right, are some pictures of disk drives. Depending on your SAS install, an index is either part of a SAS dataset or a secondary file. The index points file to where a group of observations are stored on a hard drive.

The index does not point to the location of an individual observation, but points to a section of the hard drive (a page of data- think of a page being part of the little red sector in Figure 3) that contains the observation of interest and many others as well.

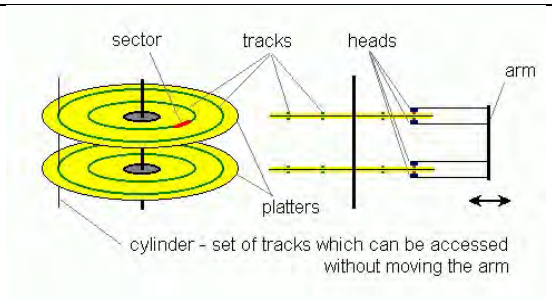


Figure 3 (in three images)

Disk drives, even disk drives in a laptop, spin very fast (5400 RPM or 7200 RPM are common) and the actuator arm will move so fast that it cannot be seen. With all that in mind, accessing data on a disk drive is much slower than accessing data in RAM. A commonly quoted figure is 20,000 times slower.

If you do not build an index, and ask for an observation, SAS will read a file from top to bottom. SAS is very fast at reading files from top to bottom. It has implemented all sorts of tricks, like anticipatory reading of data and complicated caching schemes, to minimize the need to move the actuator arm and get data to the user quickly.

An index can greatly speed up data access and index use is encouraged (with two caveats). Indexes take time to create and space after you create them. Most users create them, with a batch file, in the middle of the night to minimize time spend waiting.

In Figure 4, we see an index file that was built on subject ID. Depending on the version of SAS you are using, the index could be part of the data file or a separate file. If you use the free SAS SPDE engine to store your files, you can ask SAS to create more complicated indexes than are shown here.

The index file contains the subject ID and a pointer to where that data is on the hard drive (It points to a page of data). To understand how the index file is used requires a bit more review.

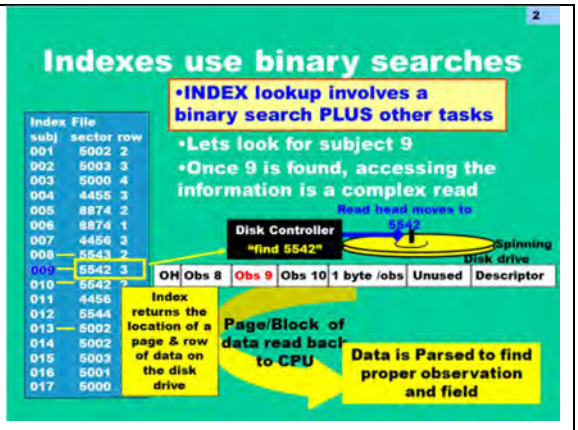


Figure 4

.

Imagine we are looking for subject nine in the index shown in Figure 4.

Index files are searched using a binary search. This seems pretty silly but, in fact, is a very fast search algorithm in a wide variety of circumstances. Let's search for subject nine using a binary search.

SAS picks the row in the exact middle of the file (008) and checks if that's the desired ID. It is not. SAS then decides if the desired ID is larger than, or smaller than, the ID it "has". In this case 9 is larger than 8.

SAS divides the file in half and only searches between the row it has and the upper end of the file. It has divided the file in half and will continue to divide the file in half until it finds the subject ID it wants.

Picking the middle of the "range to be considered" returns a 013. This is not the number SAS wants and SAS asks if the number it wants is larger or smaller than 013. It is smaller and SAS now knows that subject nine is between 008 and 013. Repeatedly picking the middle of the "range to be considered" gets us quickly to subject 009. Number of searches, in this technique, is insensitive to the size of the index.

The information on the 009 row tells us where that observation is located on the hard drive. It points to a spot on the hard drive (Did you ever wonder why you could copy files from one drive to another but you couldn't copy the index? The index points to a location on a particular hard drive so copying an index to a new hard drive makes no sense.) To get information on subject 9, sector and row are passed to the disk drive controller which waits for the disc to spin so that the head is above the proper sector and then reads a page of data – the page contains multiple observations.

The page of data (multiple observations) is read into the CPU. The CPU takes that page apart and returns the information for subject 009. Moving the hard drive actuator is slow. It is much faster to bring many observations into the CPU and have the CPU separate out the desired information than it is to position the hard drive actuator over the exact observation desired.

The important points are:
SAS is very fast on reading a file from top to bottom. SAS "watches" how your program reads data and will, often, guess the observations you want next. SAS will "read ahead" and cache data in anticipation of your next data request. With anticipatory reading and caching, the *per-row* time in a top-to-bottom read is low. However; the file can be so large that reading the whole file from top to bottom can still take an inconveniently long time.

An individual read of a page of data, using an index, is slow because it involves positioning the actuator arm to a specific place on the hard drive. You would not want to read the whole file using an index access. However; if you only had to read 1% of the file it would likely be faster to use an index lookup rather than a top-to-bottom read.

The SQL Optimizer (more about that later) always checks for the existence of an index and applies some logic to decide on the best way to recover data (index vs top-to-bottom read vs…) from a hard drive.

You can build more than one index on a file. You can have an index that is the combination of more than one variable (e.g.first name and last name).

If you're going to build an index, consider sorting the data file using the variable on which you will build the index. Figure 5 shows a sorted dataset and illustrates why it is useful to sort and index on the same variable.

If someone codes a where clause asking for people from Delaware. The Delaware people will be stored "together" on the hard drive. One move of the hard drive actuator (and read) will return many people from Delaware. Combining sorting and Indexing a file reduces the number of "slow" movements of the hard disk actuator.

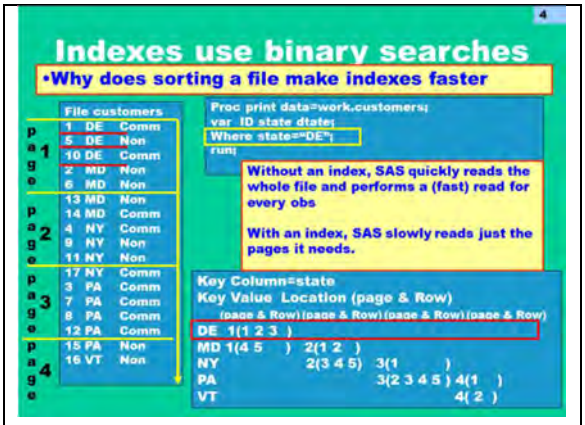


Figure 5

To provide experience with index use, I encourage a reader to paste the code below into SAS and run it. The evidence of index use can be seen in the log. options msglevel=i; is a system option that requests information about index use be written to the log. As you run code, look for the presence, or absence, of this note: "`INFO: Index Name selected for WHERE clause optimization`" as an indication of whether SAS use the index or not.

```sas
/*Example B_1 */
options msglevel=i;
ODS Listing;
ODS HTML close;
PROC SQL;
Create table B01_MyClass
 as select *
 from SASHelp.class;

Create index name on
B01_MyClass(name);

PROC Contents data=B01_MyClass
centiles; run;

PROC SQL;
Select *
 /*No note in log*/
 from B01_MyClass;

Select *
 from B01_MyClass
/*There is a note in the log*/
 where name="Jane";

Select *
 from B01_MyClass
/*There is a note in the log */
 where substr(name,1,1)="J";

 Select *
 from MyClass
/*There is a note in the log */
 where substr(name,2,1)="o";

 Select *
 from B01_MyClass
 /*No Note (violates the 10% rule?)*/
 where substr(name,2,1)in
("a","e","i","o","u");

 Select *
 from B01_MyClass
 /*No note in log - we did not have
an index on age*/
 where age=13;
```

```sas
/*Example B_2 */
options msglevel=i;
ODS Listing;
ODS HTML close;
Data B02_My_Class(index=(name));
 set SASHelp.class;
 run;

PROC Contents data=B02_My_Class
centiles;run;

Data _null_;
 set B02_My_Class;
 /*No note in log*/
 run;

Data _null_;
 set B02_My_Class;
/*There is a note in the log*/
 where name="Jane";  run;

 Data _null_;
 set B02_My_Class;
/*There is a note in the log */
 where substr(name,1,1)="J"; run;

 Data _null_;
 set B02_My_Class;
/*There is a note in the log */
  where substr(name,2,1)="o";  run;

Data _null_;
 set B02_My_Class;
 /*No Note violates the 10% rule?)*/
    where substr(name,2,1)in
("a","e","i","o","u");  run;

Data _null_;
 set B02_My_Class;
 /*NO INFO in log, but reads 3 obs*/
    where age=13;
 run;
```

For learning only. While SAS does access the index, it is not worth building an index on such a small file.

Figure 6 Section B in SAS code

It is especially interesting to see the note below in the log after you run the last data null. You see:

```
NOTE: There were 3 observations read from the data set WORK.CLASS.  WHERE age=13;
```

We did not build an index on age but only three observations are read. I suggest this is evidence of the existence of the data engine filtering rows of data using a process that is independent of index use.

At this point, you might want to get the code for this H.O.W. and run the code in Section B- 2 of 11.

C: 3 OF 11) THE SQL OPTIMZER AND IMPROVING PERFORMANCE:

I've mentioned the optimizer a few times and now we will explain what it is and how it works. I must warn the reader that information below is only approximate. SAS has a commitment to getting us quick run times and will improve the optimizer without making much fanfare. It is very possible that the optimizer is improved with no notice to the general programming population. We just notice that our jobs run faster.

When you type SQL code, you are typing characters that the computer hardware does not understand. Your "English like" SQL statements must be translated into machine language. That translation is done by a powerful subroutine called "the optimizer". Every version of SQL has an optimizer. Some SQLs have multiple optimizers and some SQL optimizers must be "tuned" to the hardware on which they run. One should note the fact that all open source SQL packages are sharing, because they are open source, the code for their optimizers. SAS developers, therefore, are privy to all of the techniques in any of the open-source packages. Any new technique from the open source community would be quickly brought into SAS. The SQL optimizer gets upgraded and improved on a continual basis with almost no fanfare. SAS does not brag about it – SAS just improves the product.

The easy to read, English – like, sentences you type after the PROC SQL statement are turned into a program by the optimizer.

```
PROC SQL _method _tree;
select name     from sashelp.class
   where sex="M"
    order by Age;
```

The code above, requires the program shown in Figure 7. What is shown there is the "explain plan" for the SQL query. It is a graphical representation of the low level steps the optimizer feels are required to produce the desired result.

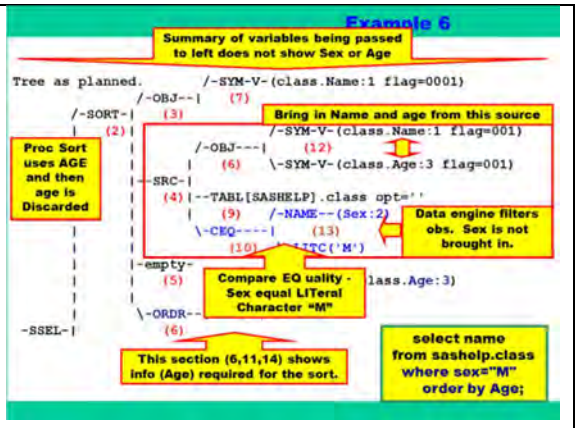


Figure 7 /*Example C_1*/ Section C in SAS code

For more about the optimizer you can Google "The SQL Optimizer Project: _Method and _Tree in SAS®9.1". (The author spent most of the summer trying to understand how the optimizer worked so that he could improve on it, and found that he could not improve on what the optimizer does).

I must, again, warn the reader that the representation of the optimizer in this section is only approximate – for two reasons. The SAS Optimizer is a trade secret of SAS Institute and it appears to be "fluid". When improvements are made, they are often made with no notice to the user community (except we notice we get to go home earlier).

It is useful to see some optimizer details. Figure 8 shows the optimizer applying a where clause. The optimizer interprets SQL code and decides if the base engine can/should handle the sub- setting (here: removing rows where the state is not Pennsylvania).

The next step is important to note. If the base engine cannot handle sub-setting the optimizer will automatically search for an index and see if that index has not been "disabled" by some odd coding in the where clause.

If there is a usable index, the optimizer checks to see whether it's faster to do a top to bottom read of the data or to use the index to find just the rows you want. All this searching for the fastest program to write happens without your knowledge.

Some sources say the cutoff for deciding whether to use an index is if the where clause returns 10% of the data set - other sources say 15%.

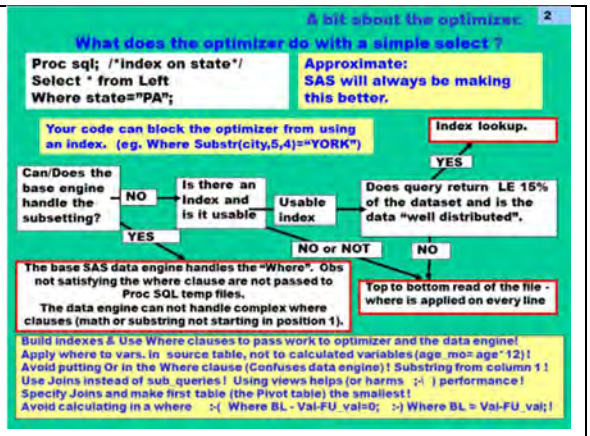


Figure 8

The meaning of that cutoff is this. If a where clause requests to find customers from California (a big state) it will likely cause the optimizer to "code" a top to bottom read of the data set. A search for customers in Delaware (a small state) will likely use an index. Very similar SQL code can generate greatly different machine –level programs.

There is another rather, to me, astounding implication of this. Imagine you're doing some sort of nationwide clinical trial with several sites in California. Due to IT issues, only one California site reported in the first data delivery so the percentage of subjects from California was small. A request for information on California subjects, at week 1, would likely use an index.

Now imagine, a month later, that all of the sites from California report current data and back fill in the data that they had failed to report previously. Imagine California subjects now make up 25% of the data file. If you run the exact same (find California people) SQL code you had run the previous month, the optimizer will likely create a different program. The optimizer does that because a different program will run faster and let you deliver your results quicker.

The optimizer's logic for joining (Figure 9) is even more complex than the where clause (Figure 7). The critical thing to do to produce a fast running join is to have an equality in the join condition. You can have several join criteria and, to reduce the run time, I often will put ***any*** valid equality condition I can in the where clause, if the original where clause is not very selective.

If I want to match people, and the business logic does not have any equality constraints, I sometimes add any equality constraint I can think of. I might code where left.state equals right.state just to keep SQL from doing the Cartesian product join or "Step Loop Join" or "Brute Force Join" over the whole nation.

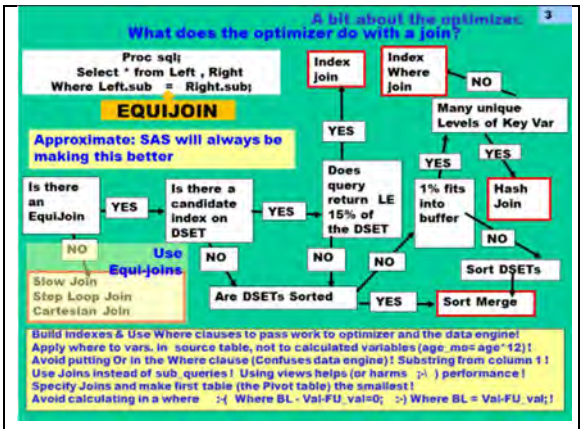


Figure 9

Another important take away from Figure 9 is that SQL does not have many tools that good programmers do not have. Your data step code ***could*** be just as fast as SQL – if you followed good programming practice. It is rare, unless someone is writing a frequently used production program, that a programmer has the time to evaluate all of the conditions (see Figure 9) that affect run time and code the "very best method". However; SQL always evaluates all the tools and always applies "good programming practice".

A second important take away from, Figure 9, is that SQL will change the program it writes depending on: the size of the file, the amount of available memory (buffer), the presence of an index and other environmental characteristics. Even if some programmer were to take the time to evaluate all of these factors, when s/he wrote an initial program, s/he would never be given the time to re-evaluate the

environment, and modify the program, when the program had to be next run. SQL never gets lazy and never gets tired. It always applies good programming practice and SQL will automatically modify a program if conditions change. This makes SQL extremely attractive as a programming tool.

There are a few points to keep in mind. SAS automatically decides whether to use an index, or not, whenever a programmer uses a where clause. Keep the number of indexes to a minimum to reduce disk space and update time. The decision to use an index is most accurate if the key variable values are uniformly distributed and the minimum and max values are not outliers. Do not create indexes on small files because small files can be quickly read top to bottom. Do not create an index on a variable with a few number of values. For good reason, people generally do not build indexes on a variable like sex. With values of M and F, sex is a poor choice for an index variable; since any data pull will likely involve more than 15% of the table. Index use is most effective if the index "returns" a small subset of the file and less than 10% is a common rule of thumb. For more details read Mike Raithal's paper.

Figure 10 shows how the order of tables, in a multi table join, can have a great impact on run time and on the amount of resources the query uses.

Figure 10 shows, in the yellow box, an SQL query that is joining three tables.

The customer file has one line per customer and 10,000 customers. The customer and order file has, on average five rows per customer – the average customer has ordered five times. The "order and item" file has 250,000 rows. Each order has, on average, five items.

SQL joins tables two at a time. If we ask SQL to join three tables, it will first join two and produce a result. It then joins the third table to the result of the first join.

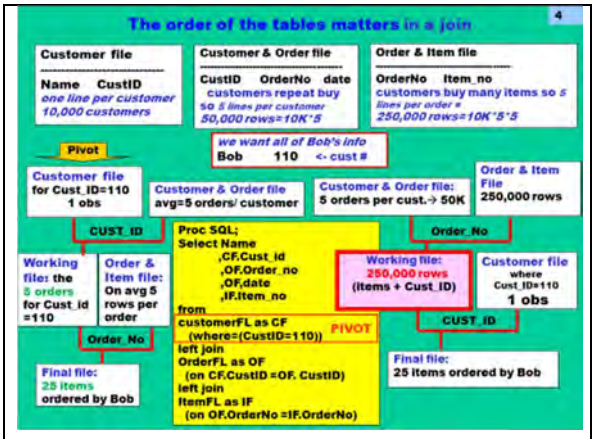


Figure 10 /*Example D1*/

Imagine we are looking for information on customer 110. On the left, and the right, side of the yellow box we can see "trees" showing results of join table ordering.

Code in the yellow box shows the customer file being selected as the first file (this is shown in the tree to the left of the yellow box). The first file is called the pivot table. We put a where clause on the customer file so that it only returns one row of data. We then join it to the order table and we get a table in memory, a table internal to the SQL process, that has five rows of data. We then join that five row table to the order_item file and get a final result that has 25 rows. This is the proper way to code these tables.

On the right-hand side of this slide we see the effect of improper ordering of tables (and maybe a little foolishness). The SQL code for this is not shown on the slide. It is similar to the code in the yellow box except that the ordering of the tables has been changed.

This programmer joins the customer order file to the order item file (with no where clause) and the internal file has 250,000 rows. This file could then be joined to the customer file, with a where clause, with the resulting table having 25 rows. This ordering has a large internal file requires much more resource use.

In a related factoid, here is an example of how smart SAS is when reading data. If you ask SAS to read a non-indexed dataset and supply a where clause involving a variable (e.g. where age =14), SAS checks to see if the data is sorted by that variable. If the dataset is sorted by that variable (that characteristic shows in a Proc Contents) SAS does the following to get you quick results. SAS does not know the location of the *first* row that satisfies the where, so it starts reading from the beginning of the file. Using the example above (where age =14), SAS reads all rows where age is less than 14 and discards those rows. SAS reads the rows where age equals 14 and keeps them. When SAS reads the first row of data where age is greater than 14, it knows that no further rows can be of interest and stops the read.

At this point, you might want to get the code for this H.O.W. and run the code in Section C- 3 of 11.

D: 4 OF 11) SUB-QUERIES: CORRELATED AND UNCORRELATED

The next section discusses subqueries: both correlated and uncorrelated. It is hoped that the graphic will make clear some of the issues that people have found confusing. As a warning to the reader; it seems that almost everyone has to see examples of ***both*** a correlated query ***and*** an uncorrelated query before they understand them. It seems that the "compare – and – contrast" makes everything clear.

A sub query is a select statement written inside another SQL statement. A sub query is often called an inner query – because it seems to be located inside an outer query. A sub query must be enclosed in parentheses.

A sub query, since it is a query can return an answer. a table, with one of the following four "shapes":

One value (a 1 row by 1 column table- a scalar subquery)
A column of values (a N rows by 1 column table - a column subquery)
A row of values (a 1 row by N column table - a row subquery)
A table of values (a N1 row by N2 column table - a table subquery)

A subquery can be coded in the from clause, the where clause, the select clause or the having clause.

The placement of a subquery, in one of the four clauses limits the shape of the answer it can return without generating an error.

Main purposes of this section are: 1) to demonstrate the difference between a correlated query and an uncorrelated query and why correlated queries should be avoided. 2) this section also demonstrates how the shape of the result of a sub-query can cause SAS to throw an error.

Figure 11 will let us talk about some characteristics of an uncorrelated query. This graphic was created to be animated and, in the presentation, shows records for individual students moving through the system. It can be a bit confusing to see this in a static presentation.

There is a query inside another query. Inside the red box we see a complete query inside a pair of matching parentheses. The sub-query is replaced with the results of the sub-query (orange box).

Uncorrelated queries are "good" because <u>they only run one time</u>. This sub-query can run on its own – and, in fact, does. Before the outer query starts to run the inner query executes and returns the results to the outer query which uses it as a data source (see arrows).

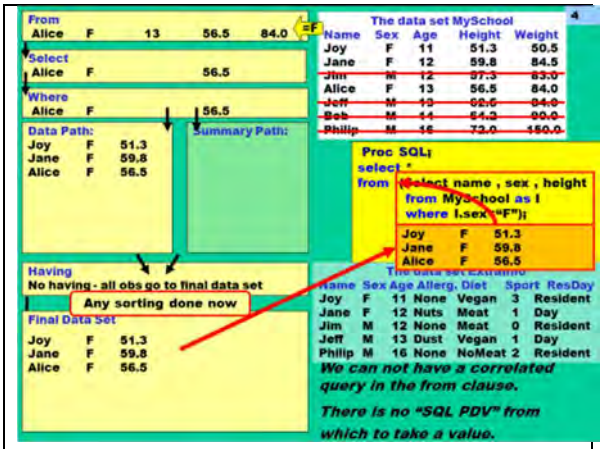


Figure 11 /*Example D_1*/

It was silly to replace this where clause with a sub query, but it illustrated characteristics of an uncorrelated query.

The animation uses a graphical representation of SQL processing that I developed. I'd like to try and explain how the animation would look if you were in the audience for this seminar. The outer query is being processed through the boxes with the light yellow coloring. The idea is that the where clause (where sex="F") has been passed to the data engine so only females get into what is, in my imagination, the SQL equivalent of the program data vector. Observations will be read, one at a time, and processed through a series of steps that are, unfortunately, all shown at the same time on this non-moving image.

Each of the females – individually:

has been read into SQL (the from box)
has had variables needed for the query "retained" (in the red select box)
has had the where filter applied two times (in the where box)
and been stored in an internal file (in the data path box)

After all the females have been processed and are “in” the data path:

observations are merged with summary statistics (that had been calculated in the summary path)
the having clause is applied to the results of the merge
observations are, individually, sent to the final data set

The reason for making this an animated cartoon is that showing the time sequence is much clearer than describing the time sequence.

Figure 12 shows another example of an uncorrelated query. While this query involves two tables, it is still an uncorrelated query because the sub query can “run on its own”.

SQL will run the sub query and replace the sub query code with the results of the sub query (see the F in the semi-transparent box). As an aside, this code is a bit dangerous. It does not bomb because only one sex plays the maximum number of sports.

Rows from the outer query are then processed and compared to the results (the F) of the inner query.

Observations from the data set MySchool that get through the where clause are stored in the box labeled data path and get into the output.

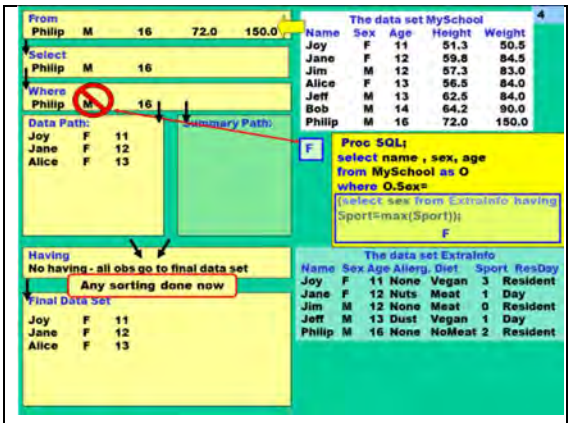


Figure 12 /*Example D_2*/

Replacing uncorrelated queries with joins can be done but might not provide the programmer with a performance improvement. However; it is often suggested that correlated queries be replaced with joins. This can often reduce run times. We will expand on that idea below.

Figure 13 illustrates a correlated query. The sub query is in the where clause and uses the data set ExtraInfo as a data source. This sub query cannot run until it’s provided a value for O.sex and that value comes into existence when the outer query is processing the data set MySchool.

What makes this a correlated query is that the sub-query cannot run until the outer query runs and provides information on O.sex.

The where clause will “pass” students who are of the same age as the student of their gender who plays the maximum number of sports. I admit this example makes no business sense – but it does not require the use of new data sets, it illustrates the principle involved and fits on a slide.

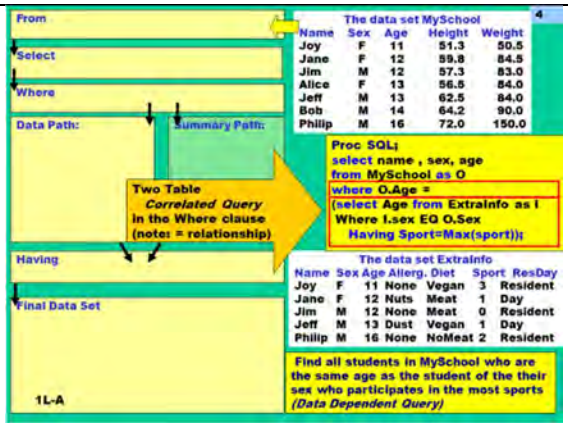


Figure 13 /*Example D_4*/

In the next graphic we will see more of the process

In Figure 14 we have started to process the data from MySchool. Joy was read into the where clause and the optimizer tried to find out if Joy's age was the same as the age of the female student who played the most sports.

The optimizer doesn't know the answer to the question in the where clause. It passes Joy's gender into the sub query and the sub query executes. This causes a full table scan of the data set ExtraInfo and the result is stored in memory. For females, the age is 11. Joy's data is compared to the results in memory and she passes through the where clause.

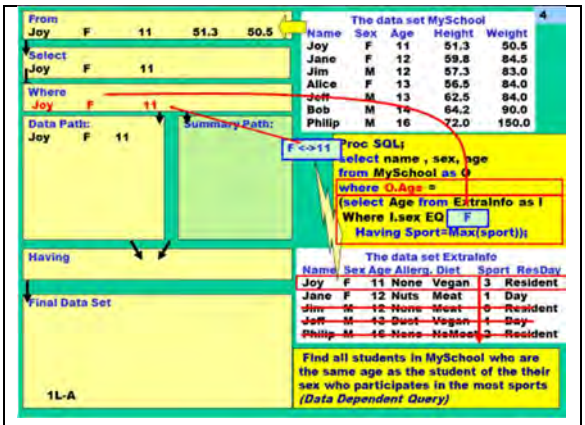


Figure 14 /*Example D_4*/

We will skip many slides in this paper. The seminar shows Jane and Alice being evaluated in the where clause and I lack the space to do that in this paper. Hopefully; you realize that, at the end of processing, Joy will be the only female in the final data set.

See Figure 15. If you are a female, SQL already knows the answer and will not run the sub-query again. Storing results of the sub query in an internal file allows SQL to get your answer quicker.

The repeated running of the sub query is why correlated queries are to be avoided. Repeated processing takes time and SAS avoids that as much as possible.

Rather than running the sub query for every row of data in the outer query, SQL only runs the sub query when it does not know the answer for the row of data in the outer query. In this case, the sub query must run every time the outer query is processing a new gender.

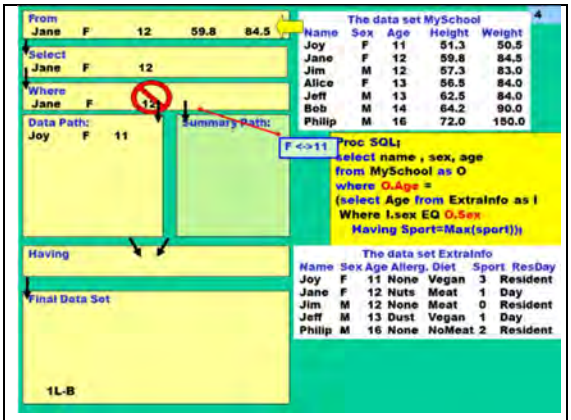


Figure 15 /*Example D_4*/

To save space, Figure 16 skips ahead to Jim.

To be fast, SQL wants to compare Jim's data to some information it has stored in memory. It knows Jim is a male.

SAS SQL first checks if Jim's answer is in the "storage area holding previously run results". SQL does not find an answer for male and must rerun the sub query.

It passes Jim's sex into the sub query and the sub query does **another** full table scan of the data set ExtraInfo.

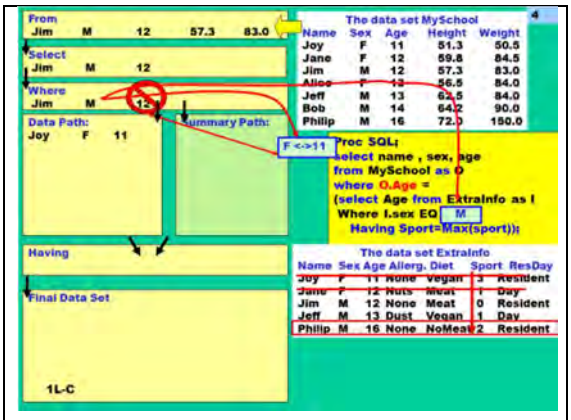


Figure 16 /*Example D_4*/

Figure 17 shows the “storage area holding previously run results” has been updated to hold the answer for males. With a typical data set (two genders) the sub query would not run again.

Philip is the male who plays the most sports and his age is 16. Phillip’s data, from ExtraInfo, is loaded into the “storage area holding previously run results”.

Jim’s age is compared to the age in the “storage area holding previously run results” and he does not pass through the where clause.

Interestingly, when Phillip’s data from MYSchool is being processed by the outer query we already know that we will compare his age to 16.

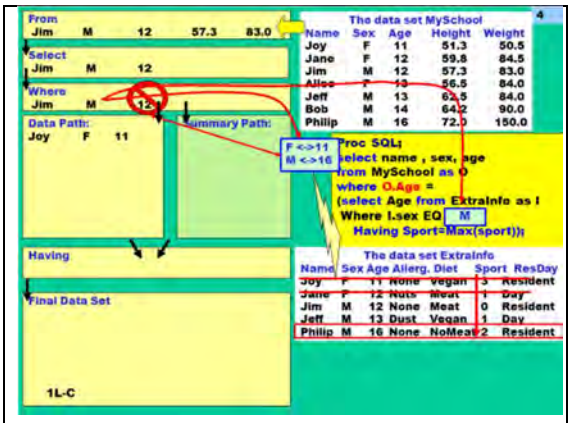


Figure 17 /*Example D_4*/

In Figure 18 we show the start of a more complicated correlated query. We are trying to find all the students in MySchool who have a person in the data set extra info that are of the same age but the opposite sex.

This may seem rather forced but I suggest it is a query that runs millions of times a day. This is the “matchmaking-web-site.com” sales – pitch query.

Because the where relationship is “exists” we will not tell our prospective customer the names of the people that meet their romantic criteria – we are just telling them if we have any suitable people in our database.

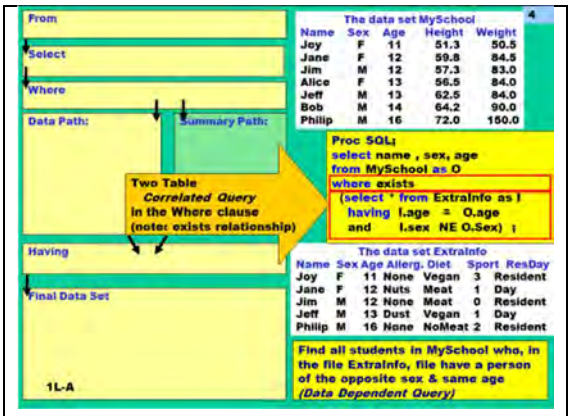


Figure 18 /*Example D_4*/

For free, we will use ‘exists” to search our database to see if we have people meeting a set of criteria. If a person wants to know who they are, they would have to give us money and we will run a different query.

In this case the sub query needs two pieces of information. It needs a customer’s age and sex.

Figure 19 jumps forward many slides. The sub query has run many times and several rows are in the “storage area holding previously run results”.

When the outer query runs, for Philip, SQL checks in the “storage area holding previously run results” to see if it knows the answer for a 16-year-old male. It does not and must run the subquery again.

It passes Phillip’s age and gender into the sub query which scans the data set extra info. It will (as indicated by the lightning bolt pointing upward and to the left) return a new value to the “storage area holding previously run results”.

Philip does not find a match and is not in the final data set.

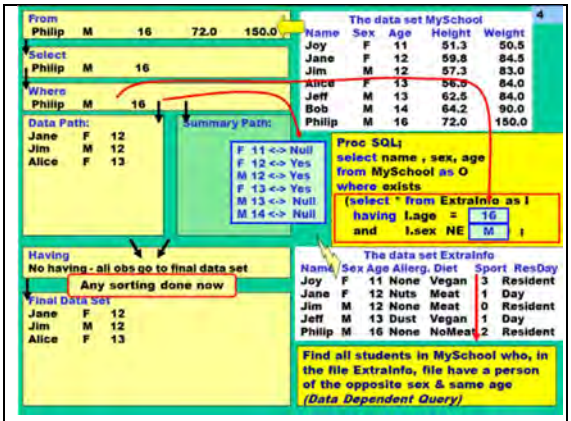


Figure 19/*Example D_4*/

As code in Figure 20 shows, the results of the correlated query in the above figures can be reproduced using a join. The join will only require one pass through the data set extra info.

The results of the PROC Compare are:

```
Number of Observations with
   Some Compared Variables Unequal: 0.
Number of Observations with
   All Compared Variables Equal: 4.

NOTE: No unequal values were found.
  All values compared are exactly equal
```

Programming manuals often mention that correlated queries can be replaced with joins, but few explain why a programmer should do this..

```
PROC SQL;
/*This is a correlated query in the WHERE
clause
 ** note I.sex  NE O.Sex */
  Create table D01_Corr_Sub_Q as
  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
     ) ;

PROC SQL;
  /*This is a join query" note inner join */
  Create table D03_Equivolent_Join as
  Select O.name , O.sex, O.age
  From
     MySchool as O
  Inner join
     (select name, sex, age
       from ExtraInfo
      ) as WasSub
  On O.Age=WasSub.Age
     and O.sex NE WasSub.sex;

 PROC compare base=D01_Corr_Sub_Q
          compare =D03_Equivolent_Join;
 title "This compares the results
   from the correlated query and the join";
 run;
title "";
```

Figure 20/*Example D_4*/

At this point, you might want to get the code for this H.O.W. and run the code in Section D- 4 of 11.

E: 5 OF 11) PLACEMENT OF SUBQUERIES IN SQL SYNTAX

I warn the reader that Figure 21 never seems to make sense *until people have seen several examples*. I hope that, after a reader goes through this section, Figure 21 will be very understandable.

Sub-queries can be in different parts of the main query. The subquery can be in: the from clause, the select statement, the where clause or the having clause.. Whether SQL throws an error depends on three things and how those things relate.

Not erroring depends on:

1) the shape of the data set returned by the query
2) the position of the subquery in the outer query
3) the relationship specified (see blue columns in figure 21)

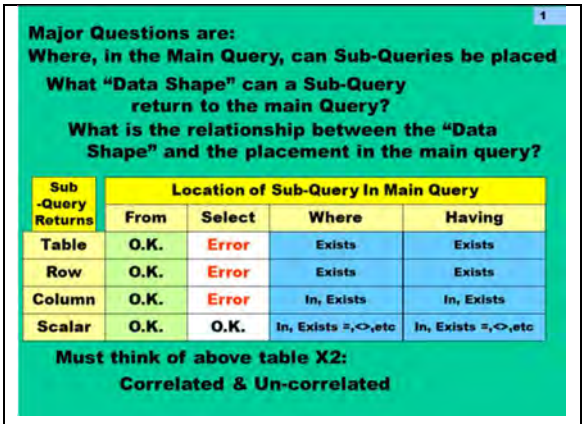


Figure 21

Figure 22 illustrates queries in the from clause and shows the “shapes of the data” returned by four different subqueries.

The shapes can be:
one column by one row (a 1 by 1)
a column
a row
a table

The from clause does not care at all about the “shape” of the data table “fed” to it. No matter what shape data your subquery returns, it will be processed by the from clause without error.

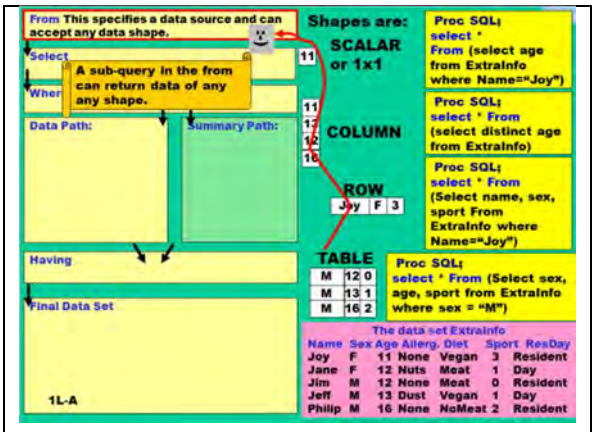


Figure 22

Figure 23 shows subqueries in the select statement returning different data shapes.

I think of the select statement as being a one row Excel table. It has 1 row and lots of columns, and (I imagine) calculations are done in the select statement.

If the subquery returns a 1 by 1 shape, I could fit that into one cell in an Excel spreadsheet. The same situation happens with SQL. If the query returns a 1 by 1 into the select statement, SQL will not error.

But, look at the other shapes. It would be impossible to fit a column of data, or a row of data, or a table of data into just one cell in an Excel spreadsheet. If your subquery returns these data shapes, SQL will error.

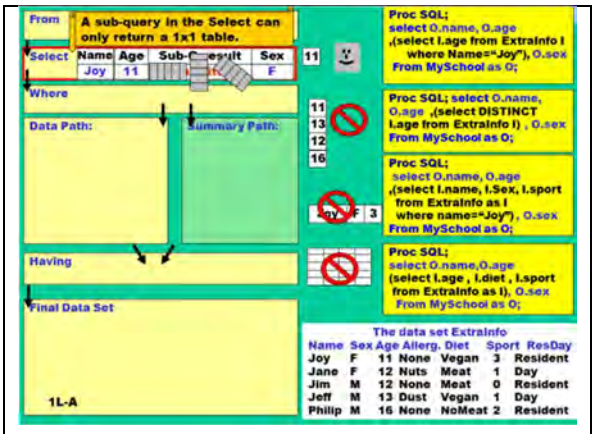


Figure 23 /*Example E_1 to /*Example E_4*/

Figure 24 introduces subqueries in the where and the having clauses. Where and having clauses have the same function – to remove rows under certain conditions.

The logical task here is to relate one row of data, (or more accurately one variable-value or one column-value) to different shapes of data. We will explore the relationships we can build between Joy’s age and sub query results having different shapes.

In Figure 24 we see the final slide in a series of slides that develops these issues. Interpreting Figure 24 is little complicated but I hope to make it understandable with a little effort.

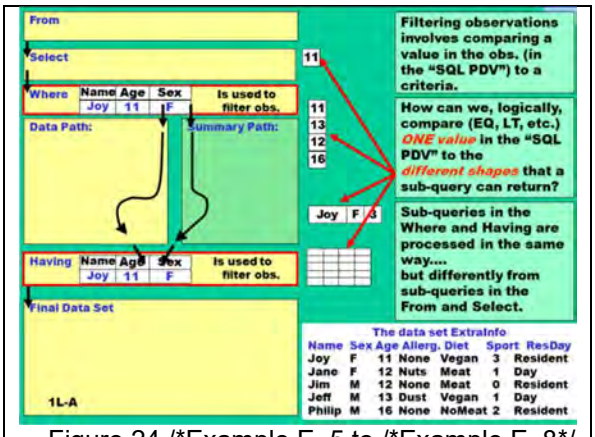


Figure 24 /*Example E_5 to /*Example E_8*/

Figure 25 summarizes the issues and we will work through its logic in steps.

Step one: How would we compare Joy's age or sex to a one by one value? This is a rather simple task and we do it often. A relationship could be age = something. The common relationship operators are: +, <>, GT , GE and LT. All of these operators will establish a relationship between one value and one other value.

If the subquery returns a column of values then the relationship between Joy's age and the values in that column could be an "in relationship". It is very logical to ask if Joy's age is **in** a column of "interesting ages". In a different situation we might check if a customer zip code is in a list of valid zips.

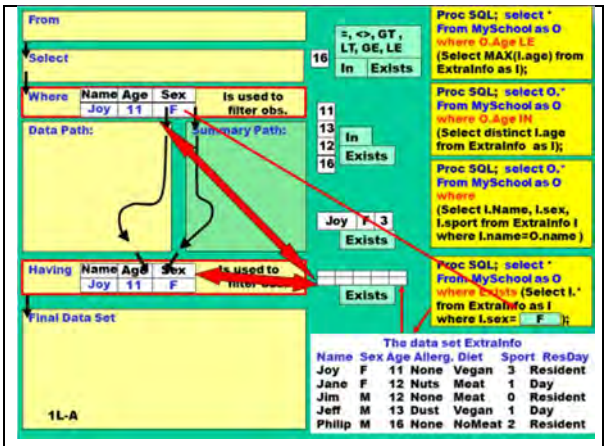


Figure 25 /*Example E_5 to Example E_8*/

If the subquery returns a row of values, it is returning elements of different data types. You can't compare Joy's age to a row of data, returned by the sub query, that contains name, sex and age.

You can, logically, ask if the query returned *anything* and allow Joy's observation to pass on to the data path if the **subquery returns anything**. In this case you would use an "exists" relationship. If the subquery returns a table of data, you can logically ask if the query returned anything at all and use an "exists" relationship.

That is how the shape of the query result affects query syntax and how subqueries can be placed in an outer query. At this time I suggest you review Figure 21 and I hope it makes sense now.

At this point, you might want to get the code for this H.O.W. and run the code in Section E- 5 of 11

F: 6 OF 11) VIEWS

Views are used in almost every company that runs a relational database. Views are programs. They look like data sets, but are really instructions to run a query and run when accessed. There are several reasons why views are so common. Views are used to help users, by giving them the data they want at its "freshest" without having to code any complicated joins. IT also departments often give users access to views, and not the underlying tables, as a way to protect the tables from programmers with "fat fingers". IT departments often create custom views for users, to give the users just the columns they want and, also, to not show users columns, or rows, they have no need to/rights to see.

A view is stored SQL syntax (stored code) that looks, to users like a data set. It contains no rows of data and retrieves the data when the view is "accessed" (clicking on it in the SAS Explorer or reading from it via a program). Typically accessing a view is slower than accessing the same information from a table. Do not sort within views because users of the view might not need the data sorted and sorting takes time. If a user needs sorted data s/he can call the view and pass it to a PROC Sort or an SQL with an order by. Finally, having your view access the proper library requires a bit of thinking and special coding.

I often use SQL views in production jobs because they can be "chained". Chaining in SQL view is passing the output from one view into the from clause of another view. Please see the example in Figure 26.

When you "chain" a view, the result of the first view, observation by observation, is passed to the second view – without being written to disk. Disk I/O is one of the things that makes jobs run slowly and should be avoided if possible.

Chained views /*Example F_3 /

```
PROC SQL;
  create VIEW F3_boys as
  select * from sashelp.class
  where sex='M';

PROC SQL;
  create VIEW F4_Old_boys as
  select * from F3_boys
  where age GE 14;

PROC SQL;
  CREATE TABLE
F05_Chained_Views as
select
  'number of old boys' as tag
  ,count(*) as Nmbr_old_boys
from F4_Old_boys
;
QUIT;

PROC PRINT
data=F05_Chained_Views; run;
```

Note that the log only contains ONE mention of rows and columns.
The SQL View-queries were processed without any I/O activity to disk

Log for running code to left

```
199  PROC SQL;
200    create VIEW F3_boys as
201    select * from sashelp.class
202    where sex='M';
NOTE: SQL view WORK.F3_BOYS has been defined.

204  PROC SQL;
205    create VIEW F4_Old_boys as
206    select * from F3_boys
207    where age GE 14;
NOTE: SQL view WORK.F4_OLD_BOYS has been defined.
208
209  PROC SQL;
210    CREATE TABLE F05_Chained_Views as
213  select
214    'number of old boys' as tag
215    ,count(*) as Nmbr_old_boys
216  from F4_Old_boys217  ;
NOTE: Table WORK.F05_CHAINED_VIEWS created, with 1
rows and 2 columns.

220  PROC PRINT data=F05_Chained_Views;
223  run;
NOTE: There were 1 observations read from the data
set WORK.F05_CHAINED_VIEWS.
```

Figure 26 /*Example F_3 /

The problem with developing a series of chained views is that you don't see the output of the views. If you have a mistake, you have to figure out whether it came from the first SQL view, or the second SQL view, or the third SQL view and so on.

If I'm doing ad-hoc queries, I rarely use views. It's too convenient to have the ability go back and look at the results of the previous query.

However, if I am writing a production job, I want to minimize disk use. When I'm developing a production program, I use create table statements and create tables in work - so that I can check things as I go along. In Figure 26, during development, the "create view" code would've been "create table". Before the job went into production, I would change the code from create table to create view to chain program steps and reduce I/O.

The idea that a view is stored SAS code will be illustrated in Figure 25. Code below makes a copy of SASHelp.class in the work directory and also a view of SASHelp.class in the work directory (this is a temporary view).

When you ask SQL to describe a table it gives you information about the variables in the table. When you ask SQL to describe a view it gives you the syntax that it uses to produce the desired result. Please see Figure 26.

```
 /*Show the diifference between a
table of data and a view of data*/
PROC SQL;
  Create table F06_Class_table as
  select *
  from SASHelp.class;

  Describe table F06_Class_table;
```

Note: log has the description of columns

```
191    Describe table F06_Class_table;
NOTE: SQL table WORK.F06_CLASS_TABLE was
created like:

create table WORK.F06_CLASS_TABLE(
bufsize=65536 )
  (
   Name char(8),
   Sex char(1),
   Age num,
   Height num,
   Weight num
  );
```

```
PROC SQL;
  Create view F06_Class_View as
  select *
  from SASHelp.class;

  DESCRIBE VIEW F06_Class_View;
```

Note: Log has SQL code

```
198    DESCRIBE VIEW F06_Class_View;
NOTE: SQL view WORK.F06_CLASS_VIEW is defined
as:

        select *
          from SASHELP.CLASS;
```

Figure 27 /*Example F_4 /

Figure 28 illustrates two things you should know if you're planning to use views often. The first is; how to make a view permanent. The second is; how to specify a library that points to the data set you want the view to access.

There is no guarantee that a view user will have established the same libraries as the view creator. Therefore; the view should contain its own libref.

In Figure 28 we see that the view is going to be stored in the library called ViewLoc (C:\PermViews). You would access this view using the two-part file name ViewLoc.Metric.

The source data is created in a data step and sent to a real table named OneSpot.Permclass.

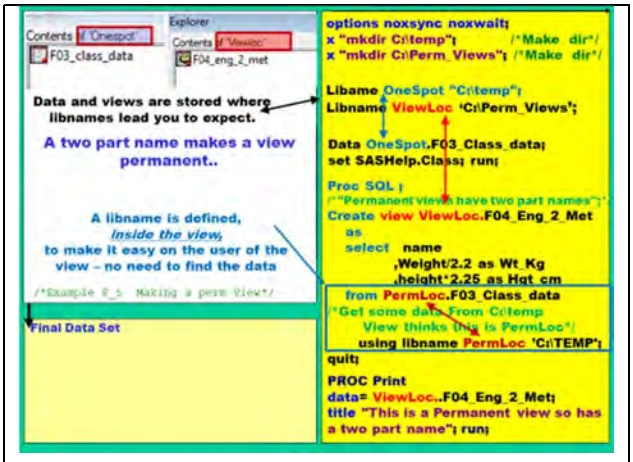


Figure 28 /*Example F_5*/

OneSpot points to c:\temp. A library reference, PermLoc. is defined inside the view and PermLock also points to c:\temp. If the view user does not have a library reference to c:\temp, the view will still run.

At this point, you might want to get the code for this H.O.W. and run the code in Section F- 6 of 11

G: 7 OF 11) FUZZY MERGING

If you live in the world of dirty data you might have to do fuzzy merging/matching. Fuzzy merging /matching is very common when you have to merge two files by person name and address. If people misspell names of subjects, or towns, you often cannot do an inner join by simply codeing `where left.name equals right.name` and expect to get many matches.

Fuzzy merging does not do all of the work involved in merging. It scores merged observations by merge quality (Perfect to no merge at all). A human must review the results and do two things. The human accepts some matches as good and also fixes "problems" (often spelling) for merges that seem close.

Figure 29 shows some dirty data to be merged. We want to merge two physician lists on name, city and whether they are board-certified (yes or no).

You can see (look for the little red underlines) that O'Banyan has one spelling issue - on his name. Jaramillo has one matching issue and Mafume has three matching issues.

Note that we are doing a Cartesian product and this will take a long time. In situations like this I often will add an equality join that says O.country = N.country. I am often willing to tolerate a few lost observations, or failures to match, in order to reduce the run time significantly.

Figure 29/*Example G_1*/

Please look at the code below: Note that source files are named Old (O) or New (N).

```
select      ((O.name=N.name)          *9
             +(O.City=N.City)         *5
             +(O.BrdCert=N.BrdCert)   *3) as score
```

O.name=N.name is a Boolean expression and is not an assignment (an assignment would be making O.name have the value of N.name). This code is asking *if* O.name is the same as N.name and will return either a zero or a one. That same process happens for City and BrdCert. The expressions return either a zero or a one.

The red numbers are the business importances of that match. I, without much justification, assigned these "values" or "relative worths" to having these variables be exact matches. I'm trying to match doctors from two lists and if the names are the same that's a strong indication that these people might be the same person.

If, as is in the case of Dr. Lu, there is a perfect match on name, city and board certification, that row will get a score of 9+5+3 or 17. Notice that the fuzziness is *not in the quality of the variable matching*. To get a nine the names must match *exactly*. The fuzziness is coming from how many variables match and from the *importance* I assign to the different variables on which we are matching.

We want to be able to find individual people, after we've done this query, so we bring in identifying information from both source files and send it to the output of the query. Since name is a variable in both tables, I prefix the variable with either an N or an O. It is easy to remember that ON means Old Name and NC means New City. I do the same for the other variables. Please look at the code below. Printing data from both tables helps us find things like spelling errors.

```
 /* O stands for Old=2010 and N stands for New=2011*/
, O.name as ON , O.city as OC ,O.BrdCert as OBC
, N.name as NN , N.city as NC ,N.BrdCert as NBC
```

Please look at the code below. This is used to remove terrible matches

```
from Old as O , new as N
     having score >= 2
     order by score desc;
```

We have to do something like a Cartesian product to do fuzzy merging but we don't want to see a huge Cartesian product in the output. If the two rows match on anything they will have a minimum score of three. To reduce run time in this example, we restrict data going into the output file to those rows that have a score greater than two. Actually, a score of three is a pretty bad match. In real life one might only consider scores above 12 to shrink the size of the output file.

Remember, you must "eyeball" the results of your fuzzy merge.

When we order by score descending I am putting the best matches at the top. Sometimes this makes business sense. If I'm trying to match some file to another file you might order by O.name (Old name) and score descending. That puts, for each OLD name, the best new match at the top of the "name group".

Figure 30 shows the results of the query.

After you do a fuzzy match, you have to 'eyeball' the matches and decide on which matches are good enough to use. The fuzzy matching greatly reduces, but does not eliminate, tedious manual checking.

We had perfect matches for two physicians and they were assigned a score of 17.

Two other physicians had high enough matching scores 12 and 8) so that they might be worth considering.

We had many matches with a score of three. They are poor matches and really should have been removed programmatically. I left them in to provide more detail to the reader as we discused this technique.

Figure 30 /*Example G_1*/

Figure 31 shows a format trick that can be used if you're trying to show your fuzzy merge output to a client. It's often useful to show the client *how things matched* and not just a score. Figure 29 illustrates how a PROC format could be used to make the score human readable.

Apply the format below to produce the results to right (red box).

Proc Format ;
value Mtching
17="17 Name & City & Board"
12="12 Name & Board"
8 =" 8 City & Board"
9 =" 9 Name"
5 =" 5 City"
3 =" 3 Board only"
0 =" 0 no Match";

Figure 31 /*Example G_2*/

This example showed how to do fuzzy matching and that the fuzziness comes from the number of variables that match exactly. In order to get a nine added to a score, the names had to match exactly.

People often want to say that Smith is very close to Smyth and names that close together should not have a zero "component score". It might be that an exact match and a Boolean (0, 1) is too restrictive. If you feel this way, SAS SQL has functions that will compare how close spellings, or phonetic spellings, are between two words. You might consider using soundex and sounds like functions in scoring how close the spellings of the two a variable are and then weighting the closeness score by a business importance score. Links below point to excellent treatments of this issue.

https://blogs.sas.com/content/sgf/2015/01/27/how-to-perform-a-fuzzy-match-using-sas-functions/

http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.421.3170&rep=rep1&type=pdf

At this point, you might want to get the code for this H.O.W. and run the code in Section G- 7 of 11

H: 8 OF 11) COALESCING

Coalesce takes a list of values as arguments and returns the first non-missing value - but that doesn't tell you how much fun you can have with coalesce.

Figure 32 shows a simple coalesce. I have a missing value for balance. If that is put into an interest formula, it produces another missing value. Imagine, the business owner of the process would prefer to see zeros instead of missing values. Coalesce checks to see if balance is valued and, if it is, uses that value to create balance2. If balance is missing it uses zero.

To do the interest calculations I must remember that balance2 is not in the source data set. In order to use it in the calculations, I must tell SAS that balance2 is not in the original data set but was *calculated* in the select statement.

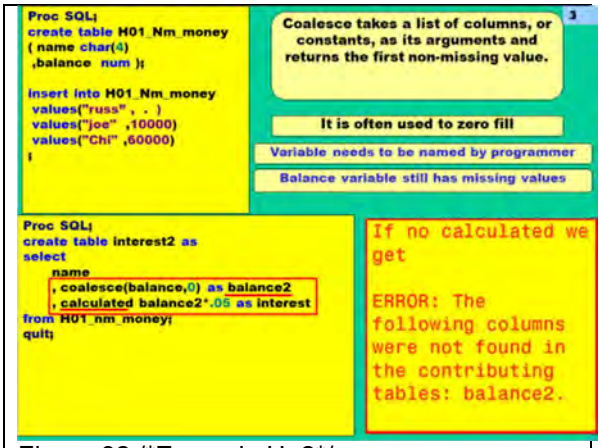


Figure 32 /*Example H_2*/

Figure 33 shows a situation that can occur when using coalesce while merging.

This is a left join and Joe is not in the output data set.

Russ was spelled two ways, we lose Russ's information on "hours on the job" but we do not lose Russ as a row in the output data set.

This is a particularly dangerous situation because it looks like the code worked and the answer is incorrect.

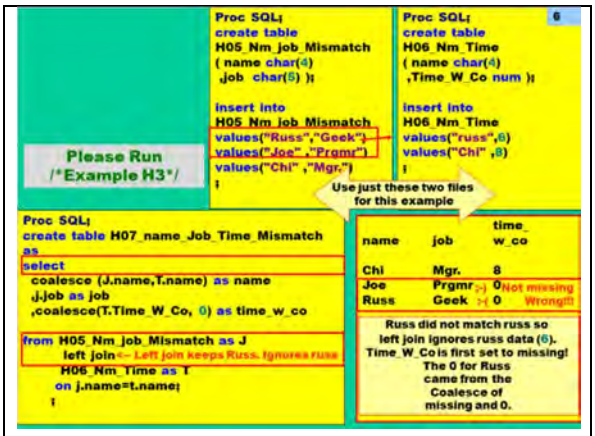


Figure 33 /*Example H_2*/

As a hint, you might upcase all of the names as part of the on statement.

Imagine we have three years of records for charitable donations. Figure 34 shows a three table join where the business goal is to tag a customer ID with the most recent bits of information. People change the name by which they like to be called. Using the most recent form of address increases the chance of getting another donation.

In order to make this a simple example, the information is just name and home state. The year, as part of state, just lets us track observations.

This is a pretty rich example of how coalesce and joins work. This technique makes it very easy to update mailing lists with new information.

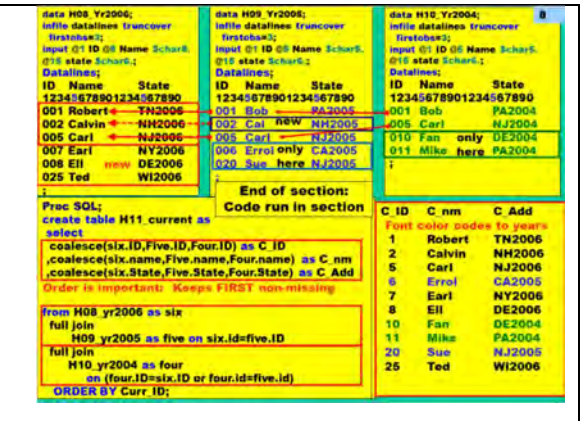


Figure 34 /*Example H_3*/

Please note, you can also do this very simply with the data step merge.

At this point, you might want to get the code for this H.O.W. and run the code in Section H- 8 of 11

I: 9 OF 11) FINDING DUPLICATES

Figure 35 shows one of my favorite uses of SQL. SQL is very good at finding duplicate values in data sets. I often deal with messy data and need to find different "levels" of duplication.

Sometimes I'm just concerned that the key values are duplicated. Sometimes I'm concerned that a whole row is a duplicate of some other row. ***Sometimes I'm concerned that the key values, and some of weird other columns, are duplicated.*** SQL allows a flexible check for different "levels" of duplication.

In this silly example we expect that name and sex should uniquely identify a row. If such information is duplicated, we'd like to see the rows themselves and a measurement that tells us how many times the row is duplicated. We see that to the right.

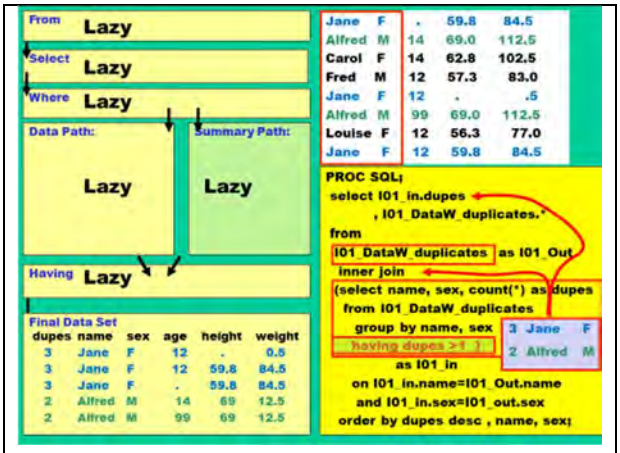


Figure 35 /*Example I_1*/

The business story for this example is that data for this research project was being entered through an online mechanism. Both Jane and Alfred were distracted during the typing process and hit submit more than once. They have duplicate data and we need to find it. Once we find the duplicate rows we want to select the one row that contains best data. We will need to "eyeball" rows of duplicate data to select the best row and will then go back and remove bad data with some additional code.

The sub query counts the number of times we have the same values for name and sex and drops all the rows where we only have one occurrence of the same name and sex. That result used, with an inner join, to do two things. Firstly; the merge adds a variable, called dupes, which shows the number of duplications for that particular combination of name and sex. Secondly; it removes all of the rows from MyClass that don't have any duplicates. The resulting report is often quite useful and can be seen in the "Final Data set" box in Figure 35.

At this point, you might want to get the code for this H.O.W. and run the code in Section I - 9 of 11

J: 10 OF 11) REFLEXIVE JOINS

Figure 36 shows a reflexive join. Most people see reflexive joins as being complicated.

The tricky part of them is that the employee number and the supervisor number have the same range of values. Jim Goodnight, in this data set, is employee number one. He reports to no one and his supervisor variable has a value of missing.

The people at job level two (Kurt, Church and Lee) all have a supervisor variable with a value of one. These people report to the president.

The trick in this reflective join is to join a person's supervisor number to someone else's employee number. Figure 37 shows a more complicated join.

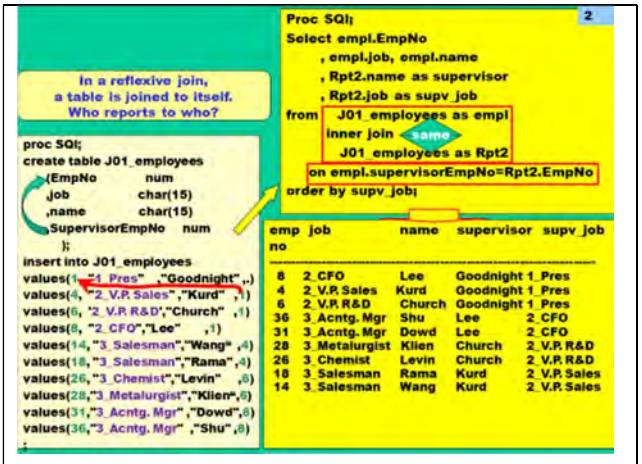


Figure 36 /*Example J_1*/

In Figure 37 we are trying to find our way from the West Coast to the East Coast. We want to fly from Los Angeles to Philadelphia and would like to use the shortest path (by time).

This is a simple network and a bit artificial. All flights have one intermediate stop and that makes the coding simple.

However; the structure that allows you to do a reflective join is present- though in a slightly different form from that in the example above.

Every flight has a starting point and an ending point and we want to use SQL to link the starting point of a flight to the ending point of another flight.

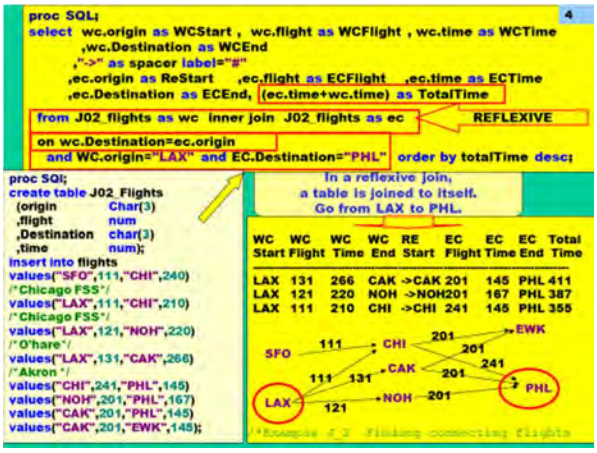


Figure 37 /*Example J_2

I've never tried to make this example more complicated, but the technique seems to be interesting and powerful.

At this point, you might want to get the code for this H.O.W. and run the code in Section j - 10 of 11

K: 11 OF 11) USE SQL DICTIONARY TABLES TO DOCUMENT DATA SETS IN HYPERLINKED EXCEL WORKBOOK

The code below uses SQL, and a macro, to document ALL your tables into a hyperlinked Excel Spread sheet. This combination of SQL and macros will, in a bit of time (it depends on how many files you have), create an XLS workbook that documents all the SAS tables in a library. I know that 9.4 TS Level 1M3 will run, I have that service pack installed and have tested it.

Vince did great coding, and the heavy lifting, to create XLS workbooks with hyperlinks. I saw him present and tweaked the code, a bit, to produce documentation on all SAS tables in a library.

```
/*************** SAS 9.4 tm3***********************************/
%MACRO Check_by_VDG_V94_TM3(LibName=SASHelp /*<-- only use upper case letters*/
                    );
/*excel limit 1,048,576 rows by 16,384 columns*/
%local lp Libname FileList DateList NObsList SizeList Lenlist NOfVarslist;
%local            ThisFile ThisDate ThisNObs ThisSize ThisLen ThisNVar;
%let Libname=%UpCase(&LibName);
%put Libname=&Libname;

ods _all_ close;
%let path2lib = %sysfunc(pathname(&LibName));
ODS Excel File="E:\_2018_SAS_Global_Forum\Macros\Contents_of_&Libname..xlsx "
          nogtitle nogfootnote style=HTMLBlue ;
ODS Excel options(embedded_titles='yes' embedded_footnotes='yes');

Proc SQL noprint /*inobs=10*/;
select memname , modate, nobs, filesize, obslen , NVar
  into :filelist separated by " "
               ,:DateList separated by " "
               ,:NObsList separated by " "
               ,:SizeList separated by " "
               ,:Lenlist separated by " "
               ,:NOfVarslist separated by " "
 from Dictionary.tables
 /*Below will eliminate views and graphic data types*/
 where libname="&Libname" & MEMTYPE="DATA" and typememem ="DATA" and nobs GE 0;
%put filelist  = &filelist   ;
%put DateList  = &DateList   ;
%put NObsList  = &NObsList   ;
%put SizeList  = &SizeList   ;
%put Lenlist   = &Lenlist    ;
%put NOfVarslist=&NOfVarslist;

 /*this is the list of all the tables and goes on the first tab*/
```

```
ods Excel options(sheet_name="List_of_tables_in_lib" );
Proc report data=sashelp.vtable /*(obs=10)*/ nowd;
title "Tables in this library (&Libname) and workbook";
title2 "S=sorted ** SK= sorted with no duplicate key values ** SR - sorted with no duplicate records";
title3 "YES | CHAR= compresses (variable-length records) by SAS using RLE (Run Length Encoding). Compresses
repeated comsiecutive characters.";
title4 "Binary=obs. compresses (variable-length records) by SAS using RLE (Run Length Encoding).";
 Column  ('Click MemName to see desired data' libname memname MEMTYPE modate typemem nobs filesize obslen NVar
                                              indxtype  sorttype  sortchar compress pcompress);
 compute memname;
     urlstring="#'"||strip(memname)||"'!A1";
        call define(_col_,'url',urlstring);
    call define(_col_,'style', 'style=[textdecoration=underline]');
 endcomp;
 where libname="&Libname" & MEMTYPE="DATA" and typemem ="DATA" and nobs GE 0;
run;quit;

/***/
ods Excel options(sheet_name="List_of_indexes_lib" );
Proc Report data=sashelp.vindex(obs=10) ;
title "Indeces in this workbook";
where libname="&Libname" & MEMTYPE="DATA" ;
run;quit;


%let lp=1;
 %do %while(%scan(&filelist,&Lp) NE);
    %let ThisFile = %scan(&filelist,&Lp);
       %let ThisDate = %scan(&DateList,&Lp);
       %let ThisNObs = %scan(&NObsList,&Lp);
       %let ThisSize = %scan(&SizeList,&Lp);
       %let ThisLen  = %scan(&Lenlist,&Lp);
       %let ThisNVar = %scan(&NOfVarslist,&Lp);

    ods excel options(sheet_interval='table');
    ods exclude all;
       data _null_;
       declare odsout obj();
       run;

   ods select all;
   ods excel options(sheet_interval='none' sheet_name="&ThisFile" );


     title "&Libname &ThisFile: rows=&ThisNObs NVars=&ThisNVar ModDate=&ThisDate Size=&ThisSize
Obslen=&ThisLen";
        Title    link="#'List_of_tables_in_lib'!A1" '(Click to return to list of tables (first tab))';
     Footnote link="#'List_of_tables_in_lib'!A1" '(Click to return to list of tables (first tab))';
     *footnote2 "&libname is: &path2lib and workbook" ;
    Proc Report data=sashelp.VColumn nowd;
       Column
         libname memname memtype name type length npos varnum label format informat
         idxusage sortedby xtype notnull precision scale transcode diagnostic ;
       where libname="&Libname" & MemName="&ThisFile";
       run;quit;

       title "&Libname &ThisFile: rows=&ThisNObs NVars=&ThisNVar :: SHOW ten obs" ;
       Proc print data=&Libname..&ThisFile(obs=10) ;
       run;quit;
       title "";

     %let Lp = %eval(&Lp+1);
%end;
ods Excel Close;
%MEND Check_by_VDG_V934_TM3;
%Check_by_VDG_V94_TM3(Libname=sashelp /*<-- only use upper case letters*/
```

CONCLUSION

This paper was written for a H.O.W. at the 2018 SAS Global Forum and has a particular style. That style was used to make this a companion to a SAS program used in that H.O.W. with the intention of creating a learn-on-your-own experience for people who do not attend. Reading this paper, while running code from the program, will largely duplicate the experience of the H.O.W. If a reader is interested in studying this topic, I would suggest they get the SAS program, either from Lex Janssen's webpage or from the author and step through the program as they read the paper.

Lex Jansen will also host copies of the macro that creates documentation for your SAS files on his web site. He will host version for different service packs as they are created.

REFERENCES

Dilorio, Frank C.(2005) "Using Dictionary Tables: An Introduction to SAS Metadata" Proceeding of the 2005 South East SAS User Group http://analytics.ncsu.edu/sesug/2005/HW07_05.PDF

Dilorio, Frank C., Nancy J. Michal, (2005) "Data About Data: An Introduction to Dictionary Tables" Proceeding of the 2005 SAS global Forum http://www.sascommunity.org/sugi/SUGI95/Sugi-95-33%20Dilorio%20Michal.pdf

Dilorio, Frank C., and Abolafia , Jeff (2003) Dictionary Tables and Views: Essential Tools for Serious Applications Proceeding of the 2003 (SUGI 29) SAS Global Forum http://www2.sas.com/proceedings/sugi29/237-29.pdf

Lavery, Russ, (2008) "An Animated Guide: Knowing SQL Internal Processes makes SQL Easy, makes SQL Easy"
Proceeding of the 2008 PhUSE Conference http://www.lexjansen.com/phuse/2008/tu/tu07.pdf

Lavery, Russ, (2005) "The SQL Optimizer Project: _Method and _Tree in SAS®9.1" Proceeding of the 2005 SAS (SUGI 30) Global Forum http://www2.sas.com/proceedings/sugi30/101-30.pdf

Raithel, Michael A., "Creating and Exploiting SAS® Indexes" Proceeding of the 2004 SAS (SUGI 29) Global Forum http://www2.sas.com/proceedings/sugi29/123-29.pdf

ACKNOWLEDGMENTS

I would like to thank everyone who has taken the time to create a presentation for a SAS users group. I owe a debt to all of those who shared their skills with me. I have learned from hundreds of fellow programmers.

CONTACT INFORMATION <HEADING 1>

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

Name: Russ Lavery, Contractor, Bryn Mawr PA
E-mail: Russ.lavery@verizon.net

Appendix A Misc. Below is the SQL code I carry on a thumb drive. Feel free to study and steal

```
/********************** Proc SQL ***************************/
/*SQL is very powerful - read the SQL manual*****************/
/*buy the ten dollar V6 SQL manual for an quick way to learn*/
/* it is called "Getting started with SQL procedure" and is SAS #55042*/
/*

Get papers via http://www.lexjansen.com/Phuse 2008;

Read An Animated Guide: Knowing SQL internal processes makes SQL easy
```

```
http://www2.sas.com/proceedings/sugi30/101-30.pdf
  or via http://www.lexjansen.com/Phuse 2008

Very good explanation of the where clause and indexing
Where clause optimization is CRITICAL to making your query run quickly
Power Indexing: A Guide to Using Indexes Effectively in Nashville Releases  Diane Olson,
http://www2.sas.com/proceedings/sugi25/25/dw/25p124.pdf

Using the Magical Keyword "INTO:" in PROC SQL
by Thiru Satchi, SUGI 27-paper71

Storing and Using a List of Values in a Macro Variable
by Arthur L. Carpenter SUGI30- paper 028

SQL SET OPERATORS: SO HANDY VENN YOU NEED THEM
by Howard Schreier, SUGI 31-paper242

Fuzzy Key Linkage: Robust Data Mining Methods for Real Databases
by Sigurd Hermansen

Existential Moments in Database Programming: SASÂ® PROC SQL EXISTS and NOT EXISTS Quantifiers,
and More Sigurd W. Hermansen and Stanley E. Legum, SAS Global Forum 2008 paper 084

How Do I Look it Up If I Cannot Spell It:An Introduction to SASÂ® Dictionary Tables
Peter Eberhardt & Ilene Brill SUGI 31 paper 259

SQL Step by Step: An advanced tutorial for business users
by Lauren and Nelson

WHAT WOULD I DO WITHOUT PROC SQL AND THE MACRO LANGUAGE
By Jeff Abolafia SUGI31 - paper 30

Using Data Set Options in PROC SQL by
Kenneth W. Borowiak SIGI 31 paper 131

/*Example 1: Housekeeping - Use SQL to create an index and delete files */
data class
     class2;
set sashelp.class;
run;

options nocenter;
Proc SQL;
create index age on class(age);              /*simple index*/
create index ageSex on class(age,sex);       /*compound index*/
drop table work.class2;                                 /*delete a table*/
quit;


/*** Example 2:  general SQL use***/
Proc SQL  _method _tree       /*shows optimizer execute plan*/
              /*noprint*/    /*suppresses printing*/
              number      /*adds row number ot output*/
              BUFFERSIZE=64M /* 65536=page size of 65536 bytes. 64k= 65536 bytes page size */
/*large values can makes SQL more likley to use hash object join*/
              feedback   /*Expands query in log*/
              double     /*double spaces output in list*/
              inobs=15  /*Number of obs to read -4 quick runs - useful in QC on large tables */
              outobs=10 /*Number of obs to print or send to outfile - useful in QC*/
              FLOW=5  /*character columns longer than n are flowed to multiple lines*/
                        /*- can be very useful*/
;     /*First semicolon is a long way from the Proc SQL*/
  /*create table SQLExample as */ /*uncomment if you want to make a table*/
  select  Name as Subject_Name
              , *
              ,COUNT(*) AS SUBJ_PASSING_WHERE_FILTER
              , avg(Height) as AVERAGE_AGE
                     from sashelp.class
                     where sex="M"
                     Group by height
                     Having substr(name,1,1) NE "P"
```

```
                                        Order by Subject_Name DESCENDING;
;                       quit;

**** Example 3 Create a macro variables** ;
data bigclass;
Set sashelp.class;
output;
output;
run;

proc SQL noprint;/*Works OK*/
select distinct name, sex, age
                into :Namelist separated by " "
                        ,:sexlist  separated by " "
                        ,:AgeList separated by " "
        from bigclass;
        quit;
%put &namelist;
%put &Sexlist;
%put &Agelist;

proc SQL noprint;/*Works OK*/
select name, sex, age
                into :AllNamelist separated by " "
                        ,:Allsexlist  separated by " "
                        ,:AllAgeList separated by " "
        from bigclass;
        quit;
%put &Allnamelist;
%put &AllSexlist;
%put &AllAgelist;

proc sql; /*this works but takes three passes through the data*/
/*No way to get this on one pass through the data*/
select Distinct name into :DistinctNamelist separated by " " from Bigclass;
select Distinct Sex into :DistinctSexlist separated by " " from Bigclass  ;
select Distinct age into :Distinctagelist separated by " " from Bigclass  ;
run;
%put &DistinctNamelist;
%put &DistinctSexlist;
%put &DistinctAgelist;


**** Example 4 Create a macro variable and use it in a scan loop** ;
proc SQL noprint; /*Has an error*/
select distinct name, sex, age
                into :Namelist /* without separated, will only store one value. See example 3*/
                        ,:sexlist  /* without separated, will only store one value. See example
3*/
                        ,:AgeList separated by " "
        from SAShelp.class;
        quit;
%put &NameList;
%put &SexList;
%put &AgeList;

proc SQL noprint;
select distinct age
                into :AgeList separated by " "
        from SAShelp.class;
        quit;
%put &NameList;
%put &SexList;
%put &AgeList;


%macro scanloop;
%let counter=1;
%do     %while(%scan(&AgeList,&counter,%str( )) NE );
  %let ThisAge=%scan(&AgeList,&counter,%str( ));
  %put &ThisAge;
```

```
        proc print data=sashelp.class;
                where age=&thisage;
                run;
    %let counter=%eval(&counter+1);
%end;
%mend scanloop;
%scanloop;

**** Example 4 Create a bunch of macro variables and use them in a macro loop** ;
PROC SQL ;
        SELECT DISTINCT name , sex
        INTO :name1-:name99, :Gndr1-:Gndr99 /*Lazy - just make bigger than you need*/
        FROM sashelp.class;quit;
%put  _user_ ;
%put &SqlObs;

        %macro WhatIS;
                Options nosymbolgen nomlogic;
                %do i=1 %to &SqlObs;
                        %put On loop number &i we see:;
                        %put Name&i is &&Name&i and Gender &i is  &&Gndr&i;
                        %put;
                %end;
        %Mend WhatIs;
        %WhatIs;


/*SQL makes six automatic macro vars(SQLOBS, SQLRC, SQLOOPS, SQLEXITCODE, SQLXRC,& SQLXMSG)*/
/*SQLOBS is # of rows that were processed by an SQL procedure statement. */
/*E.G. the # of rows that were formatted & displayed in output by a SELECT statement*/
/*  the number of rows that were deleted by a DELETE statement. */
/*    If an SQL view is created, then SQLOBS contains the value 0.*/

%macro showMe;
options nomprint nomlogic nosymbolgen;
%do I=1 %to &sqlobs; /*,- I use &sqlobs here to control the loop*/
   %put   &&name&i has gender = &&Gndr&i  ;
%end;
%mend showMe;
%showme;

*******Example 5 Applying Quotes ***************;
PROC SQL NOPRINT;
SELECT DISTINCT
        name
        ,QUOTE(name)
        ,quote(strip(name))
        ," ' " || (name) || " ' "
        ,age
        ,age FORMAT Z6.2
INTO :E1 SEPARATED BY " , "
                ,:E2 SEPARATED BY " , "
                ,:E3 SEPARATED BY " , "
                ,:E4 SEPARATED BY " , "
                ,:M1 SEPARATED BY " "
                ,:M2 SEPARATED BY " , "
     FROM sashelp.class(keep=name age); /*Thanks to Ken Borowiak*/
     QUIT;

%PUT &E1;
%put;
%PUT &E2;
%put;
%PUT &E3 ; /*Best??*/
%put;
%PUT &E4 ;
%put;
%PUT &M1;
%put;
%PUT &M2;
%put;
```

```
*******Example 6 the CASE statement (SQL's verson of an IF statement) ***************;
*Create some datasets with merge problems;
data LeftFile(keep= name Sex);
set sashelp.class;
if mod(_n_,3)=0;
run;

data RightFile(keep= name height);
set sashelp.class;
if mod(_n_,4)=0;
run;

options nocenter;
proc sql;
select
       left(coalesce(l.name, r.name)) as CoalescedName
 , case /*Could be replaced with a coalesce but this shows use of 'IS NOT NULL' */
       when sex IS NOT NULL then sex
       else                 "?"
       end
       as CaseSex
 ,case /*Could be replaced with a coalesce but this shows use of 'IS NULL' */
       when height IS NULL then .
       else                     height
       end
       as CaseHeight
 ,case
       when substr(coalesce(l.name, r.name),1,1)="J" then "This is one of out many J people"
       When substr(coalesce(l.name, r.name),1,1)="B" then "This name starts with a B       "
       else                                               "Who cares about this name       "
       end as comment
from
(select * from LeftFile) as L
full join
(select * from RightFile) as R
on l.name=r.name;
quit;


****Example 7*****************************************;
Proc sql;
create table SevenRows
(Seven a char(5) ,Seven_B num);
proc SQL;
insert into SevenRows
values("Bob", 1)
values("Sue", 2)
values("Lee", 3)
values("Sam", 4)
values("Chi", 5)
values("Ed ", 6)
values("AJ ", 7)
;
run;

Proc sql;
create table FiveRows
(Five_A char(3) ,Five_B num);
insert into FiveRows
Set Five a="Bob",Five B=11
Set Five_a="Sue",Five_B=12
Set Five_a="Lee",Five_B=13
Set Five_a="Sam",Five_B=14
Set Five_a="Chi",Five_B=15
;
quit;


****Example 8 Some Joins ******************************************;
data Left_File (keep=name sex ObsNo)
        Right_file(keep=name age ObsNo) ;
```

```
set sashelp.class;
ObsNo= n ;
if mod(_n_,3)=0 then output Left_File;
if mod(_n_,4) in (0,1) then output Right_File;
run;

options nocenter;
proc sql;
title "Comma form of inner join - using Where syntax";
title2 "Most of the time we would use a coalesce on common varaibles";
select L.*, "*" as separator, R.*
       from
       Left_file as L
       ,
       Right_file as R
       where L.name=R.name
       ;

options nocenter;
proc sql;
title "Inner join form of inner join - using on syntax";
title2 "Most of the time we would use a coalesce on common variables";
title3 "SQL Joins are different from Data step joins";
title4 "The differences show up in many-to many merges";
select L.*, "*" as separator, R.*
       from
       Left_file as L
       Inner join
       Right_file as R
       on L.name=R.name
       ;


options nocenter;
proc sql;
title "Left join";
title2 "Most of the time we would use a coalesce on variables in both data sets";
title3 "SQL Joins are different from Data step joins";
title4 "The differences show up in many-to many merges";
select L.*, "*" as separator, R.*
       from
       Left_file as L
       Left join
       Right_file as R
       on L.name=R.name
       ;

options nocenter;
proc sql;
title "Right join";
title2 "Most of the time we would use a coalesce";
title3 "SQL Joins are different from Data step joins";
title4 "The differences show up in many-tomany merges";
select L.*, "*" as separator, R.*
       from
       Left_file as L
       Right join
       Right_file as R
       on L.name=R.name
       ;

options nocenter;
proc sql;
title "Full join";
title2 "Most of the time we would use a coalesce";
title3 "SQL Joins are different from Data step joins";
title4 "The differences show up in many-to many merges";
select L.*, "*" as separator, R.*
       from
       Left_file as L
       Full join
```

```
        Right_file as R
        on L.name=R.name
        ;

******EXTRA full join example****COOL VARIABLES TO MONTOR MERGE - LIKE IN IN DATA STEP********;
             /*http://www.ats.ucla.edu/stat/sas/modules/sqlmerge.htm*/
                /*The two datasets may have records that do not match.
                Below we illustrate this by including an extra dad (Karl in famid 4)
                that does not have a corresponding family, and there are two extra families
                     (5 and 6) in the family file that do not have a corresponding dad. */

                        data dads;
                         infile datalines truncover;
                         input famid name $ inc;
                        datalines;
                        2 Art  22000
                        1 Bill 30000
                        3 Paul 25000
                        4 Karl 95000
                        ;
                        run;

                        data fam_income;
                         infile datalines truncover;
                         input famid faminc96 faminc97 faminc98;
                        datalines;
                        3 75000 76000 77000
                        1 40000 40500 41000
                        2 45000 45400 45800
                        5 55000 65000 70000
                        6 22000 24000 28000
                        ;
                        run;

/*Let's apply the previous example to these two datasets.
We see that the unmatched records have been dropped out in the merged data set,
since the  where statement eliminated them. */
               proc sql;
                 create table dadkid3 as
                 select  *
                 from dads, fam_income
                 where dads.famid=fam_income.famid
                 order by dads.famid;
               quit;

               proc print data=dadkid3;
               run;

/*What if we want to keep all the records from both datasets even they do not match?
        The following proc sql does it in a more complex way.
        Here we create two new variables. One is indic,
          an indicator variable that indicates whether an observation is from both datasets,
            1 being from both datasets and 0 otherwise.
        Another variable is fid, a coalesce of famid from both datasets.
               This gives us more control over our datasets.
               We can decide if we have a mismatch and where the mismatch happens. */

        proc sql;
          create table dadkid4  as
          select  *, (d.famid=i.famid) as InBothindicator,
                     (d.famid ~=.) as InDadIndicator,
                     (i.famid ~=.) as InFamIndicator,
                     coalesce(d.famid, i.famid) as family_id
            from dads as d full join fam_income  as I
             on d.famid=I.famid;
               quit;

        proc print data=dadkid4;
        run;
```

```
options nocenter;
proc sql;
title "Natural join";
title2 "Most of the time we would use a coalesce";
title3 "SQL Joins are different from Data step joins";
title4 "The differences show up in many-tomany merges";
select L.*, "*" as separator, R.*
       from
       Left_file as L
       Natural join
       Right_file as R
/*     on L.name=R.name */
       ;

*** Example 9 Reflexive join *********;
*data Quality is important - in a real project check/CLEAN the mapping files *;

data HospChain;
infile datalines truncover firstobs=2;
input @1 OrgID $char5. @10 ParID $char5.  @20 Rx;
/*Chn stands for Hospital Chain*/
datalines ;
12345678901234567890123456  /*Naming logic and QC Logic*/
Chn01    Chn01     100     /*Chn Stands for chain*/
Chn02    Chn02     200
Chn03    Chn03     000
Chn04    Chn04      99  /*Problem linking down -This chain has hospital but no outpatient sites*/
Chn05    Chn05      99  /*Problem linking down -This has no hospitals*/
;
run;

data Hospitals;
infile datalines truncover firstobs=2;
input @1 OrgID $char5. @10 ParID $char5.  @20 Rx;
/*Hsp stands for hospital*/
datalines ;
12345678901234567890123456 /*Naming logic and QC Logic*/
Hsp1A    Chn01     0      /*Hsp 1 rolls up to chain 1*/
Hsp2A    Chn02     0
Hsp2B    Chn02     30
Hsp3A    Chn03     10
Hsp3B    Chn03     20
Hsp3C    Chn03     30
Hsp4E    Chn04     99     /*Problem lilnking down - this Hospital has no outpatient sites*/
Hsp6E    Chn06     99     /*Problem lilnking up -there is no hospital chain six*/
;
run;

data OutpatientSites;
infile datalines truncover firstobs=2;
input @1 OrgID $char5. @10 ParID $char5.  @20 Rx;
/*OpS stands for Outpatient Sites*/
datalines ;
12345678901234567890123456  /*Naming logic and QC Logic*/
OpS1A    Hsp1A     0       /*OpS 1 rolls up to Hsp 1 to chain 1 */
OpS1B    Hsp1A     10
OpS1C    Hsp1B     99      /*Deliberate Error - there is no HSP1B to roll up into*/
OpS2A    Hsp2A     0
OpS2B    Hsp2A     10
OpS2C    Hsp2B     10
OpS3A    Hsp3A     0
OpS3B    Hsp3A     10
OpS3C    Hsp3B     10
OpS3D    Hsp3B     10
OpS3E    Hsp3E     10       /*Deliberate Error - there is no HSP3E into which we can roll up*/
;
run;

/*Assemble all the files into one */
Data AllRx;
```

```
set  HospChain
     Hospitals
     OutpatientSites;
run;


/*Create a mapping file - map any level to the hospital Chain - IN STAGES*/
/*we will use the mapping file to assign rx later */
Proc Sql;
Create table OpS2Hosp  as
select  Ops.OrgId    as OrgID
       ,Ops.ParID    as HospOrChainId
from AllRx as Ops
     Left Join  /**/
     AllRx as Hsp
  on OPS.ParId = Hsp.OrgId
  ;


Proc Sql;
Create table OpS2Hosp2Chain  as
select  O2H.OrgID
       ,O2H.HospOrChainId
          ,All.ParID    as ChainID
from OpS2Hosp as O2H /*Outpatient 2 hospital*/
     Inner Join
     AllRx as All
  on O2H.HospOrChainId = All.OrgId
  order by OrgID
  ;

/*Now bring in the Rx*/
proc SQL;
Create table RxRolledUp as
select Rx.*
       ,Ru.HospOrChainId
          ,Ru.ChainID
   from AllRx rx
     left join
     OpS2Hosp2Chain as RU
        on Rx.OrgID= Ru.OrgID
     order by Ru.ChainID, Ru.HospOrChainId, Rx.OrgID;
;

Proc SQl;
select "the Rx file has this total for rx" , sum(rx) from allRx
union
select "the Rolled up file has this total for rx" , sum(rx) from RxRolledUp
;

/*Checked by eye as well*/
OPTIONS NOCENTER;
proc print data=RxRolledUp;

title  "Chn04    Chn04      99     /*Problem lilnking down -This chain has a hospital but no
outpatient sites */";
title2 "Chn05    Chn05      99     /*Problem lilnking down -This has no hospitals*/";
title4 "Hsp4E    Chn04     99     /*Problem lilnking down - Hospital has no outpatient sites*/";
title5 "Hsp6E    Chn06     99     /*Problem lilnking up -there is no hospital chain six*/";
title7 "OpS1C    Hsp1B     99      /*Deliberate Error - there is no HSP1B to roll up into*/";
title8 "OpS3E    Hsp3E     10       /*Deliberate Error - there is no HSP3E into which we can roll
up*/";
title10 "DATA QUALITY IS IMPORTANT - bad mapping files are easy to create and hard to bebug";

VAR ChainID Rx  HospOrChainId ParID oRGid;


run;

/*Save time by asking SAS to check syntax -
  sometimes jobs with bad syntax can run for minutes before they return an error message */
```

```
proc sql; /*Validae is best*/
validate /*Validate does not work with create table*/
 /*create table girls as */
 select * from sashelp.class
 where sex="M";
 quit;

proc sql noexec; /*I thnink Noexec is less useful than validate*/
/*NOEXEC does not work on create table queries
  and does not give any message if the code is good*/
 /*create table girls as */
 select * from sashelp.class
 where sex="M";
 quit;

proc sql noexec;
/*NOEXEC does not work on create table queries
  and does not give any message if the code is good*/
/*  create table girls as */
 select * from sashelp.class
 where Ssex="M"; /*Problem on htis line*/
 quit;



***** Example ??? Unions - Venn joins*****;
/*      Examples are coming but for now
        *google these strings "Howard Scirier"  "so handy Venn "  and read his paper */

data Myclass;
infile datalines firstobs=3;
input @1 name $ @9 sex $1. @12 age @19 height @28 weight;
datalines;
Name    Sex Age    Height       Weight
12345678901234567890123456789012345678901234567890
Alfred  M   14      69.0      112.5
Alice   F   13      56.5       84.0
Carol   F   14      62.8      102.5
James   M   12      57.3       83.0
Jane    F   12      59.8       84.5
Janet   F   15      62.5      112.5
Jeffrey M   13      62.5       84.0
Joyce   F   11      51.3       50.5
Louise  F   12      56.3       77.0
Mary    F   15      66.5      112.0
Philip  M   16      72.0      150.0
Robert  M   12      64.8      128.0
Thomas  M   11      57.5       85.0
;
run;

options nocenter;
proc print data=Myclass;
run;

/*UNION : produces all unique rows from both queries.*/
proc sql;
create table simple as
  select "Boys " as Group, name, age, sex
  from MyClass
  where sex="M"
union
   select "GIRLS" as Group, name, age, sex
   from MyClass
   where sex="F";
quit;

proc print data=simple;
run;

/*UNION : produces all unique rows from both queries.
```

```
and wants columns in the same order*/
proc sql;
create table Boys  as select Name , sex from myclass where sex="M";
create table girls as select sex, name from myclass  where sex="F";

proc print data=boys;
run;
proc print data=girls;
run;


proc sql;
create table WRONG as
select * from Boys
union
Select * from girls;
;
ods listing ;
proc print data =WRONG;
run;

/*UNION :
CORRESPONDING (CORR) overlays columns that have the same name in both tables.
When used with EXCEPT, INTERSECT, and UNION, CORR suppresses columns that are not in both
tables.*/
proc sql;
create table Good as
select * from Boys
union Corr
Select * from girls;
;
ods listing ;
proc print data =Good;
run;

proc sql;
create table Boys  as select Name , Sex from myclass where sex="M";
create table girls as select Sex, Name from myclass  where sex="F";

create table Union_Corr as
select * from Boys
union Corr
Select * from girls;
proc print data = Union_Corr;   run;

/**/
proc sql;
create table Union_Corr2 as
select * from Boys
union Corr
Select * from boys;
proc print data = Union_Corr2;   run;

/*UNION :
CORRESPONDING (CORR) overlays columns that have the same name in both tables.
  When used with EXCEPT, INTERSECT, and UNION, CORR suppresses columns that are not in both
tables.
All: does not suppress duplicate rows.
  When the keyword ALL is specified, PROC SQL does not make a second pass through the data to
eliminate duplicate rows.
  Thus, using ALL is more efficient than not using it.
  ALL is not allowed with the OUTER UNION operator. */
proc sql;
create table Union_Corr_all as
select * from Boys
union Corr All
Select * from Boys;
proc print data = Union_Corr_all;   run;

/*EXCEPT : produces rows that are part of the first query only. */
proc sql;
```

```
create table Union_Except as
select * from Boys
Except
Select * from Boys;
proc print data = Union_Except;   run;

/*
EXCEPT
CORRESPONDING (CORR) overlays columns that have the same name in both tables.
When used with EXCEPT, INTERSECT, and UNION, CORR suppresses columns that are not in both tables.
*/
proc sql;
create table Except_2 as
select * from Boys
Except corr
Select * from girls;
proc print data =Except_2;   run;

/*Intersect produces rows that are common to both query results
CORRESPONDING (CORR) overlays columns that have the same name in both tables.
When used with EXCEPT, INTERSECT, and UNION, CORR suppresses columns that are not in both tables.
*/
proc sql;
create table Intersect_1 as
select * from Boys
Intersect corr
Select * from boys;
proc print data = Intersect_1;   run;
```

```
/* Intersect
CORRESPONDING (CORR) overlays columns that have the same name in both tables.
When used with EXCEPT, INTERSECT, and UNION, CORR suppresses columns that are not in both tables.
*/
proc sql;
create table Intersect_2 as
select * from Boys
Intersect corr
Select * from girls;
proc print data = Intersect_2;   run;

/*outer union : concatenates the query results.
CORRESPONDING (CORR) overlays columns that have the same name in both tables.
  When used with EXCEPT, INTERSECT, and UNION, CORR suppresses columns that are not in both
tables.
ALL is not allowed with the OUTER UNION operator. \
*/
proc sql;
create table Outer_U as
select * from Boys
Outer Union
Select * from girls;
proc print data = Outer_U;   run;

/*outer union : concatenates the query results.
CORRESPONDING (CORR) overlays columns that have the same name in both tables.
  When used with EXCEPT, INTERSECT, and UNION, CORR suppresses columns that are not in both
tables.
ALL is not allowed with the OUTER UNION operator. \
*/
proc sql;
create table Outer_U_corr as
select * from Boys
Outer Union Corr
Select * from girls;
```