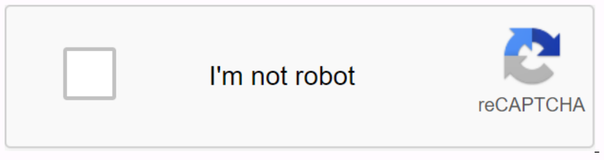

Continue

Sas 9.4 proc sql pdf

Wondering how you can use SQL code inside the SAS? Want to become a more efficient, dynamic programmer? While most of the tasks you can do with Proc SQL can also be accomplished using base SAS, often there are ways to do a task in Proc SQL with fewer lines of code and in a more efficient way. Depending on your data sets and environment, proc SQL code is sometimes faster to boot than base SAS code as well! In this article, we will show you 9 different ways to manipulate and analyze your data using the PROC SQL process. Comparisons about how to fill the same task with the base SAS code are also made throughout the article to show all the efficiencies that can be obtained using the Proc SQL method shown. Software Before proceeding, check whether you have SAS Studio or SAS 9.4 installed. Don't you have the software? Download SAS Studio now. It's free! Data sets The examples used in this article are based on CLASS and CLASSFIT data sets from the SASHelp library. For some examples in this article, you'll also need to use a modified version of the CLASSFIT dataset. Here we will create a data set called CLASSFIT_MALES that contains all the variables from CLASSFIT, but only those records where Sex is 'M' (i.e. only those records for men). To create this temporary data set and save it to a working directory, use the following code: data classfit_males; set sashelp.classfit; where the sex = 'M';run; After running the above code, classfit_males data set will remain in your WORK directory for the duration of the SAS session. If, for any reason, you need to close and reopen an SAS meeting, simply restart the above code to recreate the classfit_males data set that will be used with a few examples later in this article. The classfit_males to display as follows after running the above code: [You do not already have the software? Download SAS Studio for free.] 1. Display the dataset The most basic application of Proc SQL is to display (or print) all variables (columns) and observations (rows) from a specific data set in the SAS Results window. You use sashelp. CLASS dataset with Base SAS code, you can see here how to print the entire data set in the results window with the PRINT process: proc print data=sashelp.class;run; With proc SQL, you can obtain equivalent results by using the SELECT statement. To display all columns in the Results window, the asterisk(*) is used after the SELECT command to indicate that you want to keep all variables (columns) in the output. Proc SQL call is concluded with semi-work, followed by a QUIT statement and another half-work as shown here: proc sql; select * from sashelp.class ;quit; Both Proc Print and Proc SQL with SELECT shown above produce the following results: For data sets with a large number of variables, it may be best to view only a subset of these variables. Proc Print would achieve this with the VAR statement. Here, the VAR statement is used to print only Name and age from proc print data=sashelp.class; the age of the var;name; In Proc SQL, the same output is created by replacing the asterisk(*) from the previous example with a list of the desired variables with a distinct comma as shown here: proc sql; select name,age from sashelp.class;leave; Both Proc Print and Proc SQL statements shown above, produce the following output Names and Ages in the Results window: 2. Creating a data set from existing data Similar to step data in basic SAS programming, Proc SQL can also be used to create new datasses from existing data. To create a new data set in a WORK library called class_new that contains all variables and observations from SASHELP. CLASS, the basic SAS data step is used together with the SET statement as follows: Data class_new; Set sashelp.class; Run; In proc SQL, the equivalent output data set is produced using the CREATE TABLE and AS outputs before the SELECT: proc sql; Create a class_new select * from sashelp.class ;quit; To limit which variables are stored in a data set, you can replace the asterisk(*) in proc SQL code with a list of variables delimited by branches. To keep only the name and height in the class_new, the following code can be used: proc sql; Create a class_new as select a name,height from sashelp.class;quit; This would be equivalent to using a food statement with a step code in the SAS database: Data class_new; Set sashelp.class; Keep the height of the name; Run; Both Proc SQL and Base SAS Code shown above produce the following data set: 3. Dynamically Create variables So far displayed cases require approximately the same amount of code with an SAS or Proc SQL database. However, by using Proc SQL variables, you can easily create a dynamic within a SELECT statement, something that requires separate lines of code in the SAS database. By using the CLASS dataset as an example, you want to know what height is for each person in both inches and centimeters. With a single extract, the variable names are retained, you can rename the current height variable to height_inches, and you can create a new elevation variable called height_cm dynamically by multiplying the height in centimeters by 2.54 to convert to centimeters. proc sql; Create a class_heights name, height as height_inches,(height*2.54) as height_cm from sashelp.class;quit; The same result can also be produced with the following basic SAS code, but requires the use of renaming and sticking statement, in addition to a separate statement to create a new height_cm variable. data class_heights_base; set sashelp.class; Renaming height = height_inches; height_cm = height*2,54; Keep the name height height_cm;run; Both Proc SQL and the basic SAS code shown above produce the following table: Are you brand new to the SAS? Take our PRACTICAL SAS TRAINING COURSE for Beginners and Learn your first SAS program! 4. Producing proc SQL frequencies/counting is also a useful tool for the production of frequencies/frequency counting/counting within groups. Recall that using Proc Freq you can easily obtain frequencies (or counting) of different values that are found within the variable using the tables statement: Proc Freq data=sashelp.class; Sex tables; Run; Which produces the following table (frequencies are highlighted below): A similar result, excluding percentages, can be created with the COUNT function in Proc SQL. The variable that you want to count values for is closed in parentheses after the COUNT function. as counting is included after counting(gender) so that the number of names is also added to the newly created variable (otherwise, blank will appear in the Result window). proc sql; select sex,count as Count from sashelp.classrun; However, you will notice that the results produced by this code are not usually the desired results: At this point, the SAS does not know that you would like a number for males and females separately, so it simply counts how much value is found with variable sex throughout the dataset. To get the desired results in this case, (i.e. the number of males and females separately) it is necessary to add an additional GROUP BY statement to the Proc SQL code: proc sql; select gender,count(gender) as counted from sashelp.class group by gender;run; With the added group GROUP BY, SAS now knows that it gains a number for each group, Males and Females and outputs the following result: 5. Frequencies /Counting with multiple variables Compared to PROC FREQ, Proc SQL also has some advantages when it comes to reporting more by variable or variable. Continue to use SASHELP. As an example, note that you want to specify the distribution of males and females within each age group for this set of people. Using PROC FREQ in the SAS database requires two steps. First, you need to sort the data set by age and gender (in this order) as shown here: proc sort data=sashelp.class out=class; by age gender;appetite; Please note that we also need to create a new temporary data set in the WORK library named CLASS, since data sets stored in the SASHELP library cannot be modified. With the correctly sorted data set, you can now use PROC FREQ as before, except with two by variable, age and gender: proc freq data=class; table sex; by age gender;appetite; Unfortunately, the PROC FREQ results, which are output, are divided into multiple tables, with one table for each unique combination of variable BY values. For data sets where there are many potential values for either BY variable, this will result in a very long output as shown here: Here, there are 2 different advantages to using PROC SQL over PROC FREQ when you search for a set of counts within your data set using multiple aggregation variables (BY). The first advantage is that you do not need to sort the data set before running the PROC SQL code. It not only requires less code (and thus less typing), but can also be more efficient, depending on your environment and the size of your data. The second advantage is that counts for all groups are included in a single table that is easy to read. To retrieve a number for multiple aggregation variables in a data set with PROC SQL, the code is almost the same as the code for creating frequencies with one variable. Simply specify the variables that you want to report in the SELECT statement (age, gender), include the COUNT function with the variable for which you want frequencies, and then end with the GROUP BY statement that lists both variables in the order you want to combine: proc sql; select age,gender,count(gender) as count from sashelp.class by age,gender;stop; Which produces the following easy-to-read output: 6. Grouping/grouping data sets One of the best uses of PROC SQL is to combine data sets. When it comes to data aggregation, PROC SQL not only requires less code, but can often be more computationally efficient compared to Base SAS. Depending on the datas available and the task at hand, there are many different scenarios for combining data. The four simplest and most likely most common scenarios are as follows: There are other combinations of mergers that you can encounter, but once you understand the foundations behind these 4 joins, it is easy to use your knowledge for other scenarios that you may encounter. In addition to the output you want, it's important to consider the content of the data sets you plan to combine. When combining data, there are essentially 4 different types of mergers: one-to-one, one to more, more to one and more-to-more. One-to-one is when you have a unique ID (or Key) variable in each data set that you group. For example, if you combine file A with file B shown below, you notice that for each ID in file A and File B Proc freq data = sashelp.cars order=freq is only 1 record; Table type /out=cars_freq outcum; Run; With a one-on-multiple merge, there is only 1 record for each ID in file A and multiple records for each ID in file B: The multi-on-one merge is simply the other way around from the above example, with repeating ID values in file A and unique ID values in file B. In multi-on-multiple merge, files A and File B have recurring ID values: These different dataset scenarios are important to consider whether you want to combine data using basic SAS data merge or PROC SQL association. The key difference is that when you combine one-to-one or one-to-one, data step joins and proc sql equivalents produce the same results. However, by merging more-to-more, the PROC SQL merger produces a different result than you would expect an equivalent data merge step to produce. With PROC SQL JOIN, the multi-on-multiple merger produces what is known as a Cartesian product, which is explained later in the article. In most practical cases of data aggregation, the most common merges are one-to-one or one-to-several. As such, the following X in full association, association, Join, left and right, use data sets that have a unique ID variable so that the results generated by Data Step Join and PROC SQL Join are the same. Keep in mind that if you use similar code on your data that has a relationship more toward more, the data step and SQL results may not always be the same. This difference is later described in the Cartesian section. Full Join First, let's look at simple two merging data sets (merge) that will keep all records that match data sets A and B, non-matching A records, and B records that do not match. This is the first diagram where all circles are rotated above, and will be performed with FULL JOIN in PROC SQL. In this case, we will merge SASHELP. class data set for WORK. CLASSFIT_MALES data we've created before. Note that when referring to a data set stored in your temporary work directory, you do not need to specify a WORK library name before the data set name. Let's see how this could be done first at the SAS base. Using basic SAS, combining data sets is a two-step process. The first step is to determine which variable in each data set you will merge, and then sort the two data sets by this variable. Note that these variables must also have the same variable names. Here we will sort and merge the two SASHELP. Class and work. CLASSIT_MALES on the IME variable as shown here: proc sort data=sashelp.class out=class; by name;run; proc sorts data=classfit_males; by name;run; Remember that we also need to create a temporary copy of the CLASS dataset in the WORK library, since we cannot change the datass available in the SASHELP library. When datas are sorted, you will use the SAS data step in combination with the MERGE statement. The BY post merge statement should also be included here to ensure that records with matching names from the two data sets are aggregated as shown here: data classfit_combined; the merger of classfit_males; by name;Run Result is a data set that contains all matching records (based on NAME) from CLASS and CLASSFIT, as well as those records that did not match the NAME variable. An important note is that all variables with common names between two data sets will be overwritten by the second data set specified in the MERGE statement. For example, if the record for Alfred is in the class and in CLASSFIT_MALES, the values for SEX, AGE, HEIGHT, and WEIGHT will be overwritten with values for those same variables found in CLASSFIT_MALES because CLASSFIT_MALES also contains the variables SEX, AGE, HEIGHT, and WEIGHT. In this case, we know that the women found in the class will not be found in the CLASSFIT_MALES data set that we created, and so all those records with missing values for the PREDICT variable are actually records that do not match, because the PREDICT values only come from data CLASSFIT_MALES as shown here: Performing an equivalent task with PROC SQL has two advantages :P arva advantage is that you don't need to sort data sets before you complete the merge (I know sql as merge). Another advantage is that variables that you are merging from both data sets do not need to have the same variable names (as opposed to merging data steps in an SAS database). Please note that in this case, the variables already have the same names, so we are not taking advantage of this here. To complete JOIN using PROC SQL, you must first start with the CREATE TABLE statement (assuming you want to create a data set) followed by a SELECT statement that points to the first set of data you plan to join. In this example, we plan to create a data set called CLASSFIT_COMBINED_SQL, start by selecting all records from the CLASS dataset as shown here: proc sql; Create a table classfit_combined_sql select * from the class;leave; On the basis of the above, we now add 1 row indicating the type of merger you would like to do, which in this case is FULL JOIN, followed by the variable(s) on which we join (1 from each table in this example: proc sql; Create a classfit_combined_sql select * from the full join classfit_males class class.name = classfit_males.name;quit; The output result must now be the same as the data steps shown above. Although the output data set should be the same as the above data step, you will notice that this PROC SQL code in the SAS log will show you the following warnings that we haven't seen by merging the data steps: These appear in our selected statement because in this case, the variables Name, Age, Gender, Height, and Weight will be included in both data sets, so the warning as two variables cannot exist in the same data set of the same name. Similar to the Data Step, when you complete SQL Join and use SELECT*, SAS will automatically select one of the duplicate variables you want to keep. In the case of PROC SQL, the first instance of the selected variable will be retained. In our case, the value for Sex found in the CLASS dataset will be the value for Sex found in the CLASSFIT_MALES not taken into account. To bypass this, you must specify which variables you want to keep from which table. For example, say that you want to keep all variables from CLASSFIT, but only the variables PREDICT, LOWERMEAN and UPPERMEAN from CLASSFIT_MALES. To do this, you must include the table name, followed by the period, and then the variable name you want. You can also use the table name with an asterisk(*) to include all variables from a given table shown here: proc sql; Create a classfit_combined_sql table as the selected class.*, classfit_males.predict, classfit_males.lowermean, classfit_males.uppermean from full join classfit_males to class.name = classfit_males.name;quit; Which produces the following data set: You may think that if you have long data set names, this way, the list of variables in PROC SQL code will become long Boring. Fortunately, you can also The SQL alias advantage in PROC SQL to create short forms for table names. To assign aliases, simply <ALIAS name=> after either OD <TABLE name=> or JOIN <TABLE name=> within proc SQL code as shown here: proc sql; Create a classfit_combined_sql as select cl.*, clm.predict, clm.lowermean, clm.uppermean from class as cl full join classfit_males as clm on cl.name = clm.name ;quit; When you assign an alias, you can use that alias anywhere in the PROC SQL code that you need to refer to a table, such as a SELECT statement or JOIN statements as shown above. Ideally you want your aliases to be as short as possible without them being unrecognizable. Normally two or three letters of reference with some meaning letters will work for most locates (here cl is used short for the class and is added m at the end to the clm used for classfit_males). Internal Join Internal Join is the second scenario shown in the Vein diagram that was previously shown in this article, retaining only those records that are found in both A+B data sets. In this case, we will also bring sashelp together. class data set for WORK. CLASSFIT_MALES data we created before, but keep only those people found in class and CLASSFIT_MALES data sets, and exclude any records that don't match based on the name. As before, we will sort and merge both SASHELP. Class and work. CLASSIT_MALES on the IME variable as shown here: proc sort data=sashelp.class out=class; by name;run; proc sorts data=classfit_males; by name;run; When datas are sorted, you will re-use the SAS data step in combination with the MERGE statement. In this scenario, we will also need to use the IN dataset option, which creates a tracking variable, which observation originates from which data set. Note that a variable created with THE OPTION is not available in the SAS data set itself, but can be used in the entire SAS data step in which it was created. The BY statement is reactued to after the merge, which must be here to ensure that records with matching names from both datases are merged. Finally, we include a simple IF statement telling the SAS that if the record is from dataset a and data set b, then we would like to include it in our final data set: data classfit_combined_inner; class merge (and=a) classfit_males(in=b); by name; if a and b;running; Using PROC SQL, the code is almost identical to the first case of full merging. As before we combine data sets, we don't have to pre-sort, and everything we need to do here is simply replace FULL JOIN with INTERNAL JOIN. proc sql; Create a classfit_combined_sql as select * from the internal classfit_males group class.name = classfit_males.name;quit; With the above BASE SAS and proc SQL internal association, the result data set should contain all variables in the class and CLASSFIT_MALES, but </TABLE></TABLE></ALIAS> </TABLE></TABLE></ALIAS> have 10 rows, as only matching names are stored from both data sets (i.e. males in this case): *Note the remaining 4 variables LOWERMEAN, UPPER, LOWER i UPPER is not displayed on the screen's backs, or should be used to feed your left join A Left Join data set, the third scenario is shown in the Vein diagram u article, where the A+B recordings are preserved , a non-compliant records from database A. As before, let's see how you would do this in the SAS database. In this case, we will reunite SASHELP. class data set for WORK. CLASSFIT_MALES data we created before, but keep those people found in both CLASS and CLASSFIT_MALES data sets in addition to those found in the CLASS dataset, but not in the CLASSFIT_MALES. As before, we will sort and merge both SASHELP. Class and work. CLASSIT_MALES on the IME variable as shown here: proc sort data=sashelp.class out=class; by name;run; proc sorts data=classfit_males; by name;run; Once the dataset are sorted, you will re-use the SAS data step in combination with the MERGE statement and the previously described IN operator. The BY statement is reactued to after the merge, which must be here to ensure that records with matching names from both datases are merged. Finally, we include a simple IF statement telling the SAS that if the record is from a data set, then we would like to include it in our final data set: data classfit_combined_inner; class merge (and=a) classfit_males(in=b); by name; if a; run; USING PROC SQL, the code is again almost equal to FULL JOIN and INTERNAL JOIN. As before, we do not need to pre-sort data sets before combining data sets, and all we need to do here is simply replace internal JOIN from the previous example with LEFT JOIN. proc sql; Create a classfit_combined_sql as select * from the class left to classfit_males to class.name = classfit_males.name;quit; With the above base SAS Data Step Merge and PROC SQL LEFT JOIN, the result of the data set must contain all variables in the class and CLASSFIT_MALES. The resulting dataset should also have 19 records, The same set of original class dataset from how we hold we underheded the record plus we have original records from THE CLASS: *Have on the remaining 4 varijable LOWERMEAN, UPPERMEAN, LOWER and UPPER is not displayed on the screen sniping, or should be nazis u your datasetu Right Join A RIGHT JOIN is the fourth scenario shown on Venovo shemu u article i just suprotane lejci. Join. With the right JOIN, matching A+B records are in addition to misained records from data set B (as opposed to left join, where records that do not match from data set A are preserved in addition to the matching A+B record). Similar to the LEFT JOIN case, we will reunite SASHELP. class data set for WORK. CLASSFIT_MALES the data we have created before, but this time keep those people who data sets CLASSFIT_MALES in addition to those that are CLASSFIT_MALES data sets, but not in the classroom. As before, we will sort and merge both SASHELP. Class and work. CLASSIT_MALES on the IME variable as shown here: proc sort data=sashelp.class out=class; by name;run; proc sorts data=classfit_males; by name;run; The basic SAS data step code is basically the same as the code used for the LEFT JOIN case, but this time we will change the IF statement to tell the SAS that if the record is from data set b, then we would like to include it in our final data set: data classfit_combined_inner; class merge (and=a) classfit_males(in=b); by name; if b; run; Using PROC SQL, the code is again almost identical to the LEFT JOIN case. As before, we do not need to pre-sort data sets before combining data sets, and here we just need to replace LEFT JOIN from the previous example with RIGHT JOIN. proc sql; Create a classfit_combined_sql * from the class right join classfit_males to class.name = classfit_males.name;quit; With the above base SAS Data Step Merge and PROC SQL RIGHT JOIN, the result of the data set must contain all variables in the class and CLASSFIT_MALES. The data set resulting from this example should also have 10 records, the same set of records from the original CLASSFIT_MALES the press from the CLASSFIT_MALES. (i uncongruous records from THE CLASS): *Have in sight the remaining 4 lowermean, UPPERMEAN, LOWER AND UPPER, not displayed on the screen's back, or it should be nazis u your dataset Cartesian Product How was the sign i pre care hundred is displayed, the unique feature of PROC SQL is that when you try to combine more-to-more, the results from the PROC SQL association are different from comparable code from the basic SAS step step step. With a multi-on-more join in PROC SQL, SAS produces what is

known as a Cartesian product. With the basic Cartead association product, the number of rows in the result table is the product of the number of rows found in each of the entry tables. In this basic case, PROC SQL is used to select all records from both SASHELP. CLASS AND SASHELP. CLASSFIT: proc sql; Create a table cart_product as you select * from sashelp.class,sashelp.classfit ,quit Under this scenario PROC SQL combines each row from the first table (CLASS, n=19) with each row from the second table (CLASSFIT, n=19) rather than the ID factor or matching variable. The result is a data set of 19x19=361 lines long. This may be useful, or regardless of the desired outcome or task, but it's important to understand how the basic step of SAS MERGE and PROC SQL JOIN data behaves differently when merging. To show how the BASE SAS merge and PROC SQL JOIN behave differently, we will create two new data sets that have duplicate ID values and then try to combine them in the following example. In order to the duplicate SASHELP ID values, we will be able to post a copy of SASHELP. the class dataset, in addition to combining a copy of SASHELP. CLASSFIT data set with itself. To combine two datasps vertically (e.g. a leaflet one at the top of the other), we will use the SET statement: data classfit_duplicates; set sashelp.classfit sashelp.classfit;run; data class_duplicates; set sashelp.class sashelp.class;run; After sorting these dataset by name, you can see that we now have two records for each person in each of the two datases: proc sort data=classfit_duplicates; by name;run; proc sorts data=class_duplicates; by name;run; Here's what WORK. CLASS_DUPLICATES should look now: It's similar to this job. CLASSFIT_DUPLICATES should look like this: *Note the remaining 4 lower, UPPER, LOWER, AND UPPER screen variables are not displayed in the screenshot, but should also be found in the Forward data set to combine these data sets and keep only matching records, use the following data merge code as before: data classfit_combined_cart; merge class_duplicates(and=a) classfit_duplicates(and=b); by name; if a and b;running; The data set must contain 38 records, the same number of records as CLASS_DUPLICATES and CLASS_DUPLICATES. Here, the first appearance of the name from A corresponds to the first appearance of the same name from B, I the second appearance of the name from A is combined with the second phenomenon of classfit name <merged with=>1, 2. occurrence of CLASS Name 2, Classfit 1 Phenomenon Name <merged with=>2, 1st occurrenceCLASS Name 2, 2nd occurrence CLASSFIT Name 2, 2nd Which occurrence produces the following dataset: *Note the <merged with=>4 variable LOWERMEAN, UPPERMEAN, LOWER and UPPER not shown in the screenshot, but should be found in your dataset, however, while doing a comparable JOIN INNER WITH SQLC, the results are quite different : proc sql; Create a classfit_combined_sql_cart as select * class_duplicates internal join classfit_duplicates on class_duplicates.name = classfit_duplicates.name order by name;quit; Here in the table, which is the result, the Cartesian product is shown as described earlier, but stores only those records that meet the INTERNAL JOIN condition. In other words, the first appearance of the name 1 from CLASS_DUPLICATES matches the first and also the second appearance of the name 1 in the CLASSFIT_DUPLICATES. In addition, the second appearance of name 1 also corresponds to the first and second appearances of name 1 in CLASSFIT_DUPLICATE as shown here: Classfit name name <merged with=>1, 1. occurrence OF CLASS Name 1, 1. occurrence <merged with=>CLASSFIT Name 1, 2. OccurrenceCLASS Name 1, 2. occurrence <merged with=>CLASSFIT Name 1, 1. occurrenceCLASS Name 1, 2. occurrence CLASSFIT Name <merged with=>1, 2.</merged> </merged> </merged> </merged> </merged> occurrence Table result now contains 4 records for each name for a total of 76 rows as shown here: *Note the remaining 4 lowermean, UPPERmean, LOWER, and UPPER variables are not displayed in the screenshot, but should also be found in data set 7. Summary Statistics In addition to having the ability to create frequencies in PROC SQL, you can also use PROC SQL to calculate other summary statistics on one variable, such as sum, minimum, maximum, or average (average). While these calculations can also be done with the SAS database in PROC MEANS, proc SQL allows you to add them directly to your data set or use them in other dynamic calculations on the fly. First, let's take a simple example of how to calculate the sum, min, max and height of the HEIGHT variable in SASHELP. CLASS using basic SAS and PROC SQL. By using base SAS you can do this using PROC MEANS as mentioned above. In the lower SAS code sum, mean, min and max options with proc means statement tells SAS which statistics you would like to calculate. The VAR statement is used to determine which variable(s) you would like to calculate these statistics. proc means data=sashelp.classfit sum mean min max; var height;run; Using the above code, the following output is created: Proc SQL can calculate the same summary statistics in a SELECT statement using the sum, min, max, and mean functions, followed by an interest variable for parentheses. Because each calculation creates a new variable at the same time, these variables will have an empty name in the output. To add a column name to the output, you can use the AS followed by the corresponding name: proc sql; select sum(height) AS Sum, min(height) AS Minimum, max(height) AS Maximum, summary(height) AS Mean from sashelp.class ;quit; Using the above code, the following output is created: While PROC MEANS and PROC SQL can produce a similar output for summary statistics, PROC SQL provides you with a unique ability to add these values directly to the dataset and also use them to calculate new values. You use sashelp. FOR EXAMPLE, we can calculate the difference between each student's height and the average height of all students, and then add the result directly to the data set with a single PROC SQL call. As before, we use CREATE TABLE to create a new data set called height_diff. We then use the SELECT statement to determine which variables we would like to keep in height_diff data. Here we will keep the name, age, height from the original dataset and add two new variables. The first new variable to avg_height avg function with the height in parentheses. The height_diff is then created by subtracting the average height from the original height variable as shown here: proc sql; Create a height_diff as select name, age, height, avg(height) as avg_height, height-AVG(height) as height_diff from sashelp.class ;quit; Using the above code, the following dataset is created: Yes further increase these showcases of Use additional code in the same PROC SQL call to add variable tags and format calculated values to 1 decimal place. To add a variable label, simply add a= label after each variable that you want to mark in the statement<variable label=>SELECT. Similarly, if you want to add a format, simply include format= for each variable for which you want to<format> to apply. In this case, use numerical format 4.1 to ensure that we have a numeric length of 4 and a round to 1 decimal place for this variable. This is shown here: proc sql; create a table height_diff as the selected name, age, height label='Original Height', avg(height) as avg_height label='Average Height of all Students' format=4.1, height-AVG(height) as height_diff label='Difference in Height from Average' format=4.1 from sashelp.class ;quit; With the added tag and format statements, the data set must be similar to the following: *Note that you need to select View: Column labels as highlighted above in the SAS studio to view column labels instead of column names in the data set. 8. Creating macro variables Another useful feature of PROC SQL is the ability to easily create macro variables based on existing data. The simplest example that becomes a useful utility in more complex SAS macros is to create a macro variable that contains the number of records found in the data set. First, let's see how to get the number of records in the dataset. Reuse SASHELP. CLASS dataset, we simply use the COUNT function with an asterisk in parentheses to show that you would like to count all records as shown here: proc sql; select number(*) from sashelp.class ;quit; This produces the following output, which indicates that we have 19 records in SASHELP. class data set. To inaute this value into a macro variable, the INTO statement is used, followed by the column and the name of the macro variable you want to create. In this example, we create a macro variable called numrecs as shown here: proc sql; select count(*) in :numrecs from sashelp.class ;quit; To verify that a macro variable was created correctly, the %put statement can be used to exit the macro variable value in the log. To refer to a newly created macro variable, you must add ampersand (&) directly before the macro variable name as shown here: %put The number of records is &numrecs.; After running the above proc SQL statement in conjunction with the %put statement, you will see the following in the SAS LOG: 9. SQL Dictionary tables Dictionary tables are a valuable resource available to SAS PROC SQL programmers. The simplest terms in dictionary tables contain what is known as Metadata or, in other words, data data. The dictionary tables provide real-time information about datasets, options, macros, and various other features used in an SAS session. There are different uses for dictionary tables that are limited only by your Here we will outline some examples of using dictionary tables, but of course there are many other potential uses of these tables. First, before you specify how you can take advantage of dictionary views, it's important to know the content and structure of the tables. A TABLE DESCRIPTION STATEMENT CAN BE USED TO SPECIFY THE CONTENTS OF A DICTIONARY TABLE WITH PROC SQL. For example, to see the contents of a dictionary table, simply use the TABLE DESCRIPTION, followed by the predefined dictionary libname and the name of the dictionary table about which you want more information. In this case, you would like to know the contents of the table dictionary table. proc sql; describe the table.tables dictionary ;quit; After running the code above, the table description contains a list of columns, their type, length, as well as all available tags in the SAS log as shown here: DICTIONARY. The TABLES table is useful for getting a high-level overview and attributes of all data sets available in an SAS session. In the first example, dictionary tables will be used to determine the number of observations found in each SASHELP library dataset. In order to determine which datasps have the most observations under the SASHELP, the results will then be ordered at least from most observations. To create a list of datasiting and the number of observations, we start with a SELECT statement indicating that we want the name of the mem (data set name) and nobs (number of observations) to be preserved from the DICTIONARY. TABLE TABLE. Then add a WHERE statement indicating that you would like to keep these records only with memtype = DATA (e.g. list sas data sets in the SASHELP library and ignoring all SAS views). Then we add the ORDER BY statement to indicate that we want to order the results from the variable nobs. Finally, the descending option is included in ORDER BY to sort the output from the records with most observations to the least observations. proc sql; select memname, nobs from dictionary.tables where memtype = DATA order after nobs dropping ;quit; After running the above code, the following output is created in the results window: *Note that this is only a partial screenshot of the output, there will be additional rows in your output. It is also worth noting that SQL dictionaries tables have equivalent SASHELP views that are accessible by the Data Step code in the SAS database through a predefined SASHELP library. The full list of SQL dictionaries tables and relevant SASHELP views are shown below: below: