

The SQL Procedure Introduction and Review

Winnipeg SAS User Group

November 12th, 2008

Presented by: Craig Kasper

Manitoba Health and Healthy Living
Health Information Management Branch

Introduction

- In May of 2008, I made a presentation to the Winnipeg SAS user group on the SQL procedure. The presentation in May covered the following subjects:
 - Retrieving data using SQL SELECT queries
 - Analyzing data using summary functions and grouping.
- This presentation is a follow-up to that May presentation, and covers some more advanced analysis techniques
 - These techniques build on what was presented in May.
 - The next three slides contain a very brief review of what was covered at that time, as a framework to build on for the rest of the presentation.

Review: Parts of a basic SELECT query

- **PROC SQL;**
CREATE TABLE {tablename} AS
SELECT {columns}
FROM {table}
WHERE {row inclusion criteria}
ORDER BY {sort order};
- The parts of a select query must be assembled in this order. (If the order is different, SAS will not be able to understand the query and will report an error.)
- Only the SELECT and FROM parts of the query are mandatory.

Review: Summary Functions

- SAS supports the following summary functions (also known as “aggregate functions”) in the SQL procedure:
 - MAX(variable) selects the highest value present in that variable.
 - MIN(variable) selects the lowest value present in that variable.
 - AVG(variable) calculates the average of all non-missing values in that variable.
 - COUNT(variable) counts the number of rows present in which that variable has a non-missing value
 - SUM(variable) calculates the total of all values present in that variable
- Other summary functions also exist, although these are among the most generally useful.¹

Review: Grouping and GROUP BY

- SQL's summary functions return values for specific *groupings* of rows.
- If no groupings have been specified, the entire data set is considered to be a single group.
- Grouping is specified by adding a GROUP BY clause to the query.
- **PROC SQL;**
 SELECT Neighb, SaleYear,
 MAX(SellingPrice) as MaxPrice,
 MIN(SellingPrice) as MinPrice,
 AVERAGE(SellingPrice) as AvgPrice
 FROM RealEstateData
 GROUP BY Neighb, SaleYear
 ORDER BY Neighb, SaleYear;
- There will be one row in the output data set for each combination of the grouped variables present in the data set.

The SQL Procedure – Part 3

Advanced Analysis Techniques

Winnipeg SAS User Group

November 12th, 2008

Presented by: Craig Kasper

Manitoba Health and Healthy Living

Health Information Management Branch

The DISTINCT key word

- Ordinarily, an ungrouped SQL query returns one row of data for each row of source data. For certain kinds of queries, this may result in a lot of rows being selected which are the exactly the same.
- To only include each distinct combination of field values once in the output, add the DISTINCT keyword just after the SELECT keyword.
 - `SELECT DISTINCT Supplier, ProductType FROM Inventory;`

The DISTINCT key word

- The DISTINCT keyword can also be used to consolidate duplicate values inside of summary functions. This is usually the most useful with the COUNT function.
- For example, to count the number of unique surnames in a demographics table, the following code could be used:
 - **SELECT COUNT(DISTINCT Surname)
FROM Demographics;**

WHERE ... IN

- In addition to supporting all of the mathematical comparisons (EQ, NE, GT, and so on), SAS also supports another very useful comparison: IN.
- The IN comparison is a test for membership in a defined group.
- The IN comparison has actually been implemented slightly differently for DATA step use and SQL use. For the sake of clarity, this presentation deals with the SQL version only.
 - For more information on using the IN comparison in a DATA step, see the online help entry “SAS Operators in Expressions”
 - In the table of contents, this entry is under SAS Products → Base SAS → SAS Language Concepts → SAS System Concepts → Expressions.

WHERE ... IN

- With PROC SQL, there are two ways the IN comparison can be used.
- One way is to use it with a list of values:
 - PROC SQL;
SELECT Name, Address
FROM TeachingStaff
WHERE Faculty IN ('Arts', 'Music', 'Law');
- The other way is to use it with an SQL query providing the list of values:
 - PROC SQL;
SELECT Name, Address
FROM TeachingStaff
WHERE Faculty IN (Select Distinct Faculty FROM
Deans WHERE Name="Vacant");

Conditional Analysis

- We have already used WHERE clauses to conditionally exclude data from an SQL query.
- Sometimes, however, we may wish to analyze data differently if it meets particular conditions, without performing multiple queries to do so.

Conditional Analysis

- SAS's PROC SQL statement allows different analyses to be performed conditionally by using a CASE WHEN... THEN ... expression in a query.
- CASE WHEN ... THEN... works similarly to IF ... THEN ... statements in a DATA step.
 - Like the condition following an IF, there is a condition following the WHEN that is checked by SAS, with different results depending on whether the condition is true or false.
 - As with IF ... THEN ... ELSE ..., multiple conditions to be checked can be chained one after the other, with any condition in the chain being checked only if all of the previous conditions were false.

CASE WHEN ... THEN...

- A fully fleshed-out CASE WHEN block contains the following, in order:
 - The keyword CASE
 - For every condition to be considered:
 - The keyword WHEN
 - The condition which is to be determined to be true or false
 - The keyword THEN
 - The value (either specified or calculated) which is to be used in the query if the WHEN condition is true.
 - Optionally, the keyword ELSE, followed by the value (either specified or calculated) which is to be used in the query if none of the WHEN conditions are true.
 - The keyword END
- There is also an abbreviated form that can be used if all comparisons involve the same variable; for more information on this, see SAS's online help.
- The WHEN... THEN ... clauses, and the ELSE clause, match the values to be assigned to the conditions that they should be assigned under.
- Tip: for faster execution of your queries, order your WHEN conditions from most likely to least likely.

CASE WHEN... THEN...

- How does this work in practice?
- Consider the case where a company charges interest of 1 percent per month on accounts due for at least 30 days and 1.5 percent per month on accounts due for at least 90 days, but does not charge interest otherwise.
- **PROC SQL;**
CREATE TABLE interestCalc as
SELECT AccountID, Customer,
CASE WHEN DaysDue >= 90 THEN Balance*.015
WHEN DaysDue >= 30 THEN Balance*.01
ELSE 0 END as interestAmount
FROM AccountBalances;
- Note that the whole CASE... WHEN... THEN... clause is treated like any other SELECTed value in the query, as it returns only a single value.

More On Summary Functions

- So far summary functions have been discussed only in the context of analyzing a single specific variable.
 - Summary functions can also be used to analyze on multiple variables from the same row of the table. When this is done:
 - Grouping is considered to be at the row level.
 - The CALCULATED keyword must be used when using the calculated value in a WHERE clause.
- Summary functions can also summarize the values of calculated expressions.

```
SELECT AVG(ClassesAttended/30) as AvgAttendancePct...
```

- Summary functions can also be used within calculated expressions in a query.

```
SELECT (unitsSold/SUM(unitsSold)) as MarketShare...
```

CASE WHEN + Summary Functions

- This is not obvious, but it is very useful: CASE-WHEN expressions can be included *inside* a summary function. An example:
- ```
PROC SQL;
CREATE TABLE StatementTotals as
SELECT AccountID, Customer,
 SUM(BillingAmtUnpaid) as invoiceAmountOwing
 SUM(CASE WHEN DaysDue GE 90 THEN BillingAmtUnpaid*.015
 WHEN DaysDue GE 30 THEN BillingAmtUnpaid*.01
 ELSE 0 END) as interestAssessed
FROM Invoices
GROUP BY AccountID, Customer
ORDER BY AccountID;
```
- This query calculates the total invoiced amount due in the normal way. However, it calculates the total interest on the past-due invoices differently based upon how long a particular invoice has been due.

# CASE WHEN and Tabulation

- Using CASE WHEN... with summary functions can be very useful when analyzing data over multiple variables.
- One of the most useful ways to use CASE WHEN with summary functions is to use it for creating tabulated summarized tables, instead of PROC TABULATE. Here's an example:
  - ```
PROC SQL;  
CREATE TABLE PerformanceByGrade AS  
SELECT GradeLevel,  
AVG(CASE WHEN Subject="Math" THEN finalGrade  
ELSE . END) as MathAvg ,  
AVG(CASE WHEN Subject="English" THEN finalGrade  
ELSE . END) as EnglishAvg,  
AVG(CASE WHEN Subject NOT IN ("English" "Math")  
THEN finalGrade ELSE . END) as OtherAvg  
FROM GradesTable  
ORDER BY GradeLevel;
```
- This query analyzes the final grades over the grade level and subject variables. Unlike using PROC TABULATE to do this, however, no pre-analysis step is required to handle non-English, non-Math subjects correctly.
 - The resulting table is clearer and easier to read, too!

Thank You for Listening

- Any questions?

The SQL Procedure Bonus Material

Winnipeg SAS User Group
November 12th, 2008

Presented by: Craig Kasper
Manitoba Health and Healthy Living
Health Information Management Branch

A Word about Macro Variables

- Macro programming is programming at a level above SAS's procs and data steps.
 - Not surprisingly, a macro variable is a variable that is created, stored, and maintained at a level above SAS's procs and data steps.
 - Macro variables can be used to create SAS programs that modify themselves based on the values of those variables.
 - This can be done with or without actual macro programming.

Working with macro variables

- To set a macro variable, use a %LET statement outside of a PROC or a DATA step:
 - `%LET ReportCriteria=ReportYear eq 2007 and Province eq "MB" ;`
 - When this is run, everything between the first equals sign and the semicolon is assigned to the variable ReportCriteria.
- To refer to the value of a macro variable, use an ampersand (&) followed by the name of the variable.
- `PROC SQL;`
`SELECT * FROM MyTable WHERE &ReportCriteria;`
- When this is processed by SAS, SAS substitutes in the value of the macro variable and interprets this as:
- `PROC SQL;`
`SELECT * FROM MyTable WHERE ReportYear eq 2007 and Province eq "MB" ;`
- In practice, there are more wrinkles to it than what I've just explained, but at a conceptual level, this is how macro variables work.

SQL and SAS Macro Variables

- SAS has been designed to allow macro processing and the SQL procedure to work well together.
- It's relatively simple to take values from SAS macro variables and use them in a PROC SQL statement – you just add a reference to the variable and let SAS do the rest.
- ```
PROC SQL;
SELECT varname1, &myChosenVar
FROM MyTable
WHERE varname2="&MyChosenValue." ;
```

# SQL and SAS Macro Variables

- If you're just learning about SAS macros and macro variables, it may seem impossible to use values selected using SQL to define macro variables. It's not, however.
- The secret is to tell PROC SQL to put the select values directly into your macro variables. Not surprisingly, this is done by adding the INTO keyword to the query.

# SQL and SAS Macro Variables

- Here's an example:
  - PROC SQL NOPRINT;  
SELECT value1, value2 INTO :Result1, Result2  
FROM exampleTable  
WHERE MyRowIndex=27;
    - Note that using the NOPRINT option prevents PROC SQL from printing the query results to the output window, as it normally would for a query not being used to create a table.
- Once the query is complete, the variables Variable1 and Variable2 will contain the values of fields Value1 and Value2 from the first row returned by the query.
- Accordingly, a query of this sort is the most useful if you're looking up a single row from a table, or if you're using summary functions.
- It is possible, however, to get values from rows beyond the first row, too.

# SQL and SAS Macro Variables

- There are two ways to read values from multiple rows into macro variables
- The first way is to specify multiple macro variables in the INTO part of the SQL.
- PROC SQL NOPRINT;  
SELECT value1, value2 INTO :FirstResult1 THROUGH :FirstResult4, SecondResult1 THROUGH SecondResult4  
FROM exampleTable  
WHERE MyRowIndex in (27, 28, 29, 30)  
ORDER BY MyRowIndex;
  - The values in the first row will go into FirstResult1 and SecondResult1, the values in the second row will go into FirstResult2 and SecondResult2, and so on.

# SQL and SAS Macro Variables

- The second way to read values from multiple rows into macro variables is to read all of the selected values from a specific field into a particular macro variable, separated by characters you specify.
- ```
PROC SQL NOPRINT;  
SELECT value1, value2 INTO :FirstResult,  
SecondResult1 SEPARATED BY ", " FROM  
exampleTable;
```
- The value of the resulting macro variable can be used as a single value, or separated into smaller pieces using SAS's SCAN function, or its macro function counterpart, %SCAN.

End of Bonus Material

- Any questions?

Footnotes

1. SAS's PROC SQL understands over a dozen different types of summary functions. A master list of these functions is given below. Where multiple functions yield the same result, they are listed on the same line
 - AVG, MEAN Average or mean of values
 - COUNT, FREQ, N Aggregate number of non-missing values
 - CSS Corrected sum of squares
 - CV Coefficient of variation
 - MAX Largest value
 - MIN Smallest value
 - NMISS Number of missing values
 - PRT Probability of a greater absolute value of Student's t
 - RANGE Difference between the largest and smallest values
 - STD Standard deviation
 - STDERR Standard error of the mean
 - SUM Sum of values
 - SUMWGT Sum of the weight variable values which is 1
 - T Testing the hypothesis that the population mean is zero
 - USS Uncorrected sum of squares
 - VAR Variance
- It is also worth mentioning that while SAS's PROC SQL understands a MEDIAN function, it does not do so in a normal or even useful way. PROC MEANS or PROC UNIVARIATE should be used to calculate medians instead.