An Introduction to SQL in SAS®

Pete Lund, Looking Glass Analytics, Olympia WA

ABSTRACT

SQL is one of the many languages built into the SAS® System. Using PROC SQL, the SAS user has access to a powerful data manipulation and query tool. Topics covered will include selecting, subsetting, sorting and grouping data—all without use of DATA step code or any procedures other than PROC SQL.

THE STRUCTURE OF A SQL QUERY

SQL is a language build on a very small number of keywords:

- **SELECT** columns (variables) that you want
- **FROM** tables (datasets) that you want
- **ON** join conditions that must be met
- **WHERE** row (observation) conditions that must be met
- **GROUP BY** summarize by these columns
- **HAVING** summary conditions that must be met
- **ORDER BY** sort by these columns

For the vast majority of queries that you run, the seven keywords listed above are all you’ll need to know. There are also a few functions and operators specific to SQL that can be used in conjunction with the keywords above.

*SELECT* is a statement and is required for a query. All the other keywords are clauses of the *SELECT* statement. The *FROM* clause is the only one that is required. The clauses are always ordered as in the list above and each clause can appear, at most, once in a query.

The nice thing about SQL is that, because there are so few keywords to learn, you can cover a great deal in a short paper. So, let’s get on with the learning!

As we go through the paper you will see how much sense the ordering of the SQL clauses makes.

You’re telling the program what columns you want to *SELECT* and *FROM* which table(s) they come. If there is more than one table involved in the query you need to say *ON* what columns the rows should be merged together. Also, you might only want rows *WHERE* certain conditions are met.

Once you’ve specified all the columns and rows that should be selected, there may be a reason to *GROUP BY* one or more of your columns to get a summary of information. There may be a need to only keep rows in the result set *HAVING* certain values of the summarized columns.

Finally, there is often a need to *ORDER BY* one or more of your columns to get the result sorted the way you want.

Simply thinking this through as we go through the paper will help you remember the sequence of clauses.

CHOOSING YOUR COLUMNS - THE SELECT STATEMENT

The first step in getting the data that you want is selecting the columns (variables). To do this, use the *SELECT* statement:

```
select BookingDate,
     ReleaseDate,
     ReleaseCode
```

List as many columns as needed, separated by commas. There are a number of options that can go along with columns listed in a *SELECT* statement. We’ll look at those in detail shortly.

We’ll also see that new columns can be created, arithmetic and logical operations can be performed and summary functions can be applied. In short, *SELECT* is a very versatile and powerful statement.
CHOOSING YOUR TABLE – THE FROM CLAUSE

The next step in getting that data that you want is specifying the table (dataset). To do this, use the FROM clause:

```sql
select BookingDate,
    ReleaseDate,
    ReleaseCode
from SASclass.Bookings
```

The table naming conventions in the FROM clause are the normal SAS dataset naming conventions, since we are referencing a SAS dataset. You can use single-level names for temporary datasets or two-level names for permanent datasets.

These two components (SELECT and FROM) are all that is required for a valid query. In other words, the query above is a complete and valid SQL query. There are just a couple additions we need to make for SAS to be able to execute it.

USING SQL IN SAS

Now that we know enough SQL to actually do something “legal” (SELECT and FROM), let’s see how we would use SQL in SAS: PROC SQL.

```sql
proc sql;
    select BookingDate,
        ReleaseDate,
        ReleaseCode
    from SASclass.Bookings;
quit;
```

There are a number of things to notice about the syntax of PROC SQL.

First, note that the entire query (SELECT…FROM…) is treated a single statement. There is only one semicolon, placed at the end of the query. This is true no matter how complex the query or how many clauses it contains.

Second, the procedure is terminated with a QUIT statement rather than a RUN statement. Queries are executed immediately, as soon as they are complete – when it hits the semicolon on the SELECT statement. This has a couple of implications: 1) a single instance of PROC SQL may contain more than one query and 2) the QUIT statement is not required for queries to run.

Finally, SQL statements can only run inside PROC SQL. They cannot be embedded in other procedures or in data step code.

By default, the output of the query above would produce these results (see right) in the SAS output window.

The columns are laid out in the order in which they were specified in the SELECT and, by default, the column label is used (if it exists).

WHAT TO DO WITH THE RESULTS

By default, the results of a query are displayed in the SAS output window. This statement is actually a bit narrow.

PROC SQL works just like all other SAS procedures: the results of a SQL SELECT are displayed in all open ODS destinations. The following code:

```sql
ods html body='c:\temp\Bookings.html';
ods pdf file='c:\temp\Bookings.pdf';

proc sql;
    select BookingDate,
        ReleaseDate,
        ReleaseCode
    from SASclass.Bookings;
quit;

ods html close;
ods pdf close;
```
will produce output in all three open ODS destinations – the output window (the LISTING destination is open by default), an HTML file and a PDF file.

You can create a SAS dataset (a table) from the results of the query by preceding the SELECT statement with a CREATE TABLE statement.

```
proc sql;
create table ReleaseInfo as
  select BookingDate,
    ReleaseDate,
    ReleaseCode
  from SASclass.Bookings;
quit;
```

The CREATE TABLE statement does two things:
2. Suppresses the printed output of the query. No matter how many ODS destinations are open, no output is generated.

Notice that the order of columns in the SELECT statement not only determines the order in the query output, it also determines the order in a table if CREATE TABLE is used.

The naming conventions for tables in the CREATE TABLE statement are the same as elsewhere in the SAS System because we're really just creating a SAS dataset.

**SELECT STATEMENT OPTIONS**

The columns on a SELECT statement can be renamed, relabeled or reformatted.

- Rename: use the *AS* keyword
- Label: use the *LABEL* keyword
- Format: use the *FORMAT* keyword
- Length: use the *LENGTH* keyword

The following query would create a temporary SAS dataset called ReleaseCodes having three variables:

```
proc sql;
create table ReleaseCodes as
  select BookingDate
    as BD
  , ReleaseDate
    as RD
  , ReleaseCode
    label='Rel Code'
  from SASclass.Bookings;
quit;
```

By default, all formats and labels are carried over from the original table when a new table is created. In the above query we associated a new format with ReleaseDate, which we renamed RD. We also attached a new label to ReleaseCode. Any other formats or labels would remain as they were.
Notice that the attributes of the columns in the new table are the same as those in the original, unless they were changed. Even though we gave BookingDate a new name (BD) the other attributes remain – the same format and label.

There is no restriction on the number of attributes that can be set on a column. A column can be renamed, reformatted and relabeled all at the same time. Simply separate the attribute assignments with spaces – note that, in the query above, ReleaseDate is both renamed and reformatted.

### Selecting All Columns – A Shortcut

All the columns in a table can be specified using an asterisk (*) rather than a column list. For example,

```sql
proc sql;
  create table BookingCopy as
  select *
  from SASclass.Bookings;
quit;
```

If you have a number of columns in a table, and you want them all, this shortcut is obviously a time saver.

The downside is that you must know your data! If you specify the columns, you know what you’re getting. This shortcut can also be a little more problematic when you start working with multiple tables in joins and other set operations.

### Creating New Columns

Just as we did to rename a column, the `AS` keyword is used to name a new column. In the above example two existing columns were used to create a new column. Notice that the syntax of the assignment is opposite of a normal arithmetic expression – the expression is on the left side of “AS” (the “equal sign”).

The output from the above query would look like this:

```sql
select BookingDate,
       ReleaseDate,
       ReleaseDate - BookingDate as LOS
from SASclass.Bookings;
```

You can see that the new column is labeled “LOS” which would have been the variable name if we’d had a `CREATE TABLE` statement. Without the `AS`, the column in the output would have had no label, just blank space, and the name of the new column in the new table would be _TEMA001. If you had additional new columns created without AS they would be named _TEMA002, _TEMA003 and so on. You can see the importance of naming your new columns!

Any SAS formats, including picture formats, and informats can be used in a `SELECT` statement. The following `SELECT` statements are all valid:

```sql
select put(RaceCd,$Race.) as Race,
       input(Bail,BailGroup.) as GroupedBail,…
       put(Infractions,InfractionGrp.) as NumInfractions,…
```

The syntax is exactly the same as using a `PUT` or `INPUT` function in data step code.

### Conditional Logic in the Select Statement – The CASE Operator

There is yet another way to create new columns in the `SELECT` statement. While there is no “if…then…else” statement, the `CASE` operator gets close. The syntax of the `CASE` operator is quite simple – a series of `WHEN...THEN` conditions wrapped in `CASE...END AS`:

```sql
CASE
  WHEN <condition> THEN <value>
  WHEN <condition> THEN <value>
  ELSE <value>
```

```
The following query selects all the columns from the Infractions table, notice the ",", and then creates a new column, CheckThese, that is set to "X" for inmate on staff assaults ("IS") or serious infractions ("S").

```sql
select *
, case
  when InfractionType eq 'IS' then 'X'
  when Severity eq 'S' then 'X'
  else '
end as CheckThese
from SASclass.Infractions;
```

There can be as many WHEN conditions as you like and they can contain any valid SAS expression, including logical operators. So, the CASE logic above could also have been written as:

```sql
case
  when InfractionType eq 'IS' or Severity eq 'S' then 'X'
  else '
end as CheckThese
```

As the WHEN conditions are processed the value is set based on the first condition that is true. For example, let's change the query above to prioritize the infractions that we want to check on. We want to look at severe inmate on staff assaults first, then other inmate on staff assaults, then other severe infractions. The new CASE logic could be:

```sql
case
  when InfractionType eq 'IS' and Severity eq 'S' then 1
  when InfractionType eq 'IS' then 2
  when Severity eq 'S' then 3
  else 0
end as CheckThese
```

The order of the conditions is important, since the severe inmate on staff assaults meet all three of our criteria for assigning a value to CheckThese. Just like IF...THEN...ELSE, a value is assigned when the first WHEN condition is true.

The ELSE and AS keywords are optional. If AS is omitted the new column will be auto-named, as we saw earlier, starting with _TEMA001.

If ELSE is omitted the value of the new column will be set to missing if none of the WHEN conditions are met. A note will be included in the SAS log reminding you of this. In our first query there would be no difference in the outcome if the ELSE was omitted or not, since we set the value to missing. However, it is good practice to be specific in the assignment of the "default" value. You avoid the note in the log and you make it easier to debug and maintain later on.

If all of the WHEN conditions use the same column the column name can be added to the CASE operator and omitted from the WHEN condition. For example, the following two examples are equivalent:

```sql
case
  when Age lt 13 then 'PreTeen'
  when Age lt 20 then 'Teenager'
  else 'Old Person'
end as AgeGroup
```

The column name (Age) is simply moved out of the WHEN condition and into the CASE operator. This is only valid when using a single column.

**CHOOSING YOUR ROWS – THE WHERE CLAUSE**

Now that we've selected and created the columns that we want, we might not want all the rows in the table. The WHERE clause gives us a way to select rows.

The WHERE clause contains conditional logic that determines whether a row will be included in the output of the query. The WHERE clause can contain any valid SAS expressions plus a number that are specific to the WHERE clause.
Remember, like everything except SELECT and FROM, that the WHERE clause is optional. If it is included in a query it always comes immediately after the FROM clause. If we wanted to select rows from the Infractions table for only serious infractions, we could use the following query.

```
select *
from SASclass.Infractions
where Severity eq 'S';
```

The syntax of the WHERE clause is one or more conditions that, if true, causes the record to be included in the output of the query. If there is more than one condition to be met, use AND and/or OR to put multiple conditions together. For example, if we wanted to further restrict the rows selected from the Infractions table to select only serious inmate-on-inmate assaults we just need to add the second condition the WHERE clause:

```
select *
from SASclass.Infractions
where Severity eq 'S' and InfractionType eq 'II';
```

There is no limit to the number of conditions you can have in your WHERE clause.

There is a difference between how SAS handles mismatched data types in WHERE clauses and other parts of the language. (This is true whether using WHERE in PROC SQL or in a data step or other procedure.) In most cases SAS will perform an automatic type conversion, from character to numeric or vice versa, to make the comparison valid. For example, if NumCharges is a numeric field, the following datastep will execute but the SQL query will not:

```
data DataStepResult;
  set SASclass.Bookings;
  if NumCharges eq '2';
run;
```

```
proc sql;
  create table QueryResult as
  select *
  from SASclass.Bookings
  where NumCharges eq '2';
quit;
```

The IF statement in the datastep does an automatic conversion of the ‘2’ to numeric and evaluates the expression. A note is written to the log alerting you to this:

```
NOTE: Character values have been converted to numeric values at the places given by:
  (Line):(Column).
```

The WHERE clause requires compatible data type and the query above generates an error:

```
ERROR: Expression using equals (=) has components that are of different data types.
```

The WHERE clause data-compatibility requirement is true whether it is used in a SQL query, datastep or other procedure. The error messages written to the log may be different. For example, running the above datastep with a WHERE statement rather than an IF statement generates a slightly different error message (though the step still fails to execute):

```
data DataStepResult;
  set SASclass.Bookings;
  where NumCharges eq '2';
run;
```

```
ERROR: Where clause operator requires compatible variables.
```

### Special WHERE Clause Operators

There are a number of operators that can be used only in a WHERE clause or statement. Some of them can add great efficiency and simplicity to your programming as they often reduce the code needed to perform the operation.

- **THE IS NULL AND IS MISSING OPERATORS**
You can use the * IS NULL or IS MISSING operators to return rows with missing values. The advantage of * IS NULL (or IS MISSING) is that the syntax is the same whether the column is a character or numeric field.

```sql
select *
from SASClass.Charges
where SentenceDate is null;
```

**Note**: IS MISSING is a SAS-specific extension to SQL.

In most database implementations there is a distinction between empty (missing) values and null values. Null values are a unique animal and compared successfully to anything but other null values. You need to be aware of how nulls values are handled if you are using SQL in a non-SAS environment.

- **The BETWEEN Operator**

The BETWEEN operator allows you to search for a value that is between two other values.

```sql
select *
from SASClass.Bookings
where BookingDate between '1jul2001'd and '30jun2002'd;
```

When using BETWEEN, be aware that the end points are included in the results of the query. In the example above, 7/1/2001 and 6/30/2002 are both included.

The column used with BETWEEN can be numeric or character. There are dangers in using character fields in any non-equality comparison. You need to know the collating sequence that your operating system uses (i.e., is "a" greater than or less than "A") and how any unfilled values are justified (i.e., "a" is not the same as "a ").

There is an interesting behavior of the BETWEEN operator. The values are treated as the boundaries of a range and are automatically placed in the correct order. This means that the following two conditions produce the same result:

```sql
where ADPmonth between '1jul2001'd and '30jun2002'd
where ADPmonth between '30jun2002'd and '1jul2001'd
```

The order of the BETWEEN values is not important, though it is recommended that they be specified in the correct sequence for ease of understanding and maintainability.

- **The CONTAINS Operator**

You can search for a string inside another string with the CONTAINS operator. The following query would return all rows where ChargeDesc contains "THEFT" – for example, "AUTO THEFT", "THEFT 2", "PROPERTY THEFT".

```sql
select *
from SASClass.Charges
where ChargeDesc contains 'THEFT';
```

Like other SAS sting comparisons it is case sensitive. To make the comparison case insensitive you can use the UPCASE function. The new WHERE clause:  

```sql
where upcase(ChargeDesc) contains 'THEFT'
```

would also return "Car Theft" and "Theft of Property".

The CONTAINS condition does not need to be a separate “word” in the column value. So, the above query would also return "THEFT2" and "AUTOTHEFT".

- **The LIKE Operator**

You can do some rudimentary pattern matching with the LIKE operator.

- _ (underscore) matches any single character
- % (percent sign) matches any number of characters (even zero)
- other characters match that character
The WHERE clause below says, "Give me all rows where the charge description has the characters 'THEFT' with any number of characters before or after."

$$\text{where ChargeDesc like 'THEFT%';}$$

Does this sound familiar? It will return the exact same rows as CONTAINS "THEFT" would. But, LIKE is much more powerful! With LIKE you can specify, to a degree, where you want your search string to be found. Let’s take the example above and look for "THEFT" in a number of places in the charge description.

$$\text{where ChargeDesc like 'THEFT%';}$$

This would return any charge descriptions that starts with the work "THEFT" and is followed by anything else. Values like "THEFT", "THEFT 2" and "THEFT-AUTO" would all be found.

$$\text{where ChargeDesc like '%THEFT';}$$

This WHERE would return all rows that ends with the word "THEFT" and starts with anything else. Again, "THEFT" would be a valid value, as would "AUTO THEFT" and "3RD DEGREE THEFT". It would not return "AUTO THEFT 3" as we made no provision for anything to come after "THEFT".

$$\text{where ChargeDesc like '%_THEFT';}$$

Now, this WHERE is similar to the one above, except that now there must be at least one character before "THEFT". This would exclude rows with the value "THEFT" from the result set. What would be the difference if we replaced the underscore with a space? Remember, the underscore means any character and the space would mean a space. So, the underscore would return "AUTO THEFT" and "AUTO-THEFT" whereas the space would only return "AUTO THEFT".

Let’s look at one more example. Suppose we were interested in finding those people who had a previous theft charge and never showed up for court. Now they are being booked into jail for failing to appear for their theft charge. We can find that by looking for "THEFT" and "FTA" (failure to appear). We can use the following WHERE clause to find them:

$$\text{where ChargeDesc like 'THEFT_FTA';}$$

Here are the results of that query:

<table>
<thead>
<tr>
<th>ChargeDesc</th>
<th>Frequency</th>
<th>Percent</th>
<th>Cumulative Frequency</th>
<th>Cumulative Percent</th>
</tr>
</thead>
<tbody>
<tr>
<td>THEFT FTA</td>
<td>3</td>
<td>11.11</td>
<td>9</td>
<td>14.11</td>
</tr>
<tr>
<td>THEFT+Thn</td>
<td>1</td>
<td>3.70</td>
<td>4</td>
<td>14.11</td>
</tr>
<tr>
<td>THEFT/TThn</td>
<td>23</td>
<td>85.19</td>
<td>27</td>
<td>100.00</td>
</tr>
</tbody>
</table>

This doesn’t look too bad. We found all the rows that had a “THEFT” and “FTA” separated by any single character. We found some with a space, some with a dash and some with a slash. However, what if not all the booking officers use that convention (single character between the two words) when they enter the charge description? Look at the results of the next WHERE:

$$\text{where ChargeDesc like 'THEFT_FTA';}$$

Notice that the original values are still there, but we pick up a lot more possibilities. It pays to know your data!

### Sounds Like

The "sounds like" operator (=*) uses the Soundex algorithm to match like character values. A little background on the Soundex algorithm will help in determining when and where you might use it.

The Soundex algorithm encodes a character string, usually a name, according to rules originally developed by Margaret K. Odell and Robert C. Russel in 1918. The method was patented in 1918 and 1922. It was used extensively in the 1930’s by WPA crews working to organize Federal Census data from 1880 to 1920. It is widely used in genealogical software and other applications where name searching and matching is paramount.

The algorithm returns a character string composed of a single letter followed by a series of digits. The rules for calculating the value are as follows:
- Retain the first letter in the argument and discard the following letters:
  - A E H I O U W Y
  - Assign the following numbers to these classes of letters:
    1: B F P V
    2: C G J K Q S X Z
    3: D T
    4: L
    5: M N
    6: R
- If two or more adjacent letters have the same classification from Step 2, then discard all but the first. (Adjacent refers to the position in the word prior to discarding letters.)

For example,

```
select LastName, FirstName
from SASClass.Inmates
where LastName = 'Smith';
```

Using the above rules, what is the value of “SMITH” that we used in the query above?

Step 1 – retain the first letter (“S”) and eliminate the “I” and “H”: SMT
Step 2 – “M” gets a value of 5 and “T” gets a value of 3
So, “SMITH” has a Soundex value of “S53”.

Let’s look at the results of our example query and check some of the values that were returned:

<table>
<thead>
<tr>
<th>Last Name</th>
<th>Frequency</th>
<th>Percent</th>
<th>Cumulative Frequency</th>
<th>Cumulative Percent</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMITH</td>
<td>27</td>
<td>27.10</td>
<td>28</td>
<td>98.55</td>
</tr>
<tr>
<td>SMITHEE</td>
<td>1</td>
<td>3.23</td>
<td>30</td>
<td>98.77</td>
</tr>
<tr>
<td>SNEED</td>
<td>1</td>
<td>3.23</td>
<td>31</td>
<td>100.00</td>
</tr>
</tbody>
</table>

“SNEED” and “SCHMIDT” seem a bit far removed from “SMITH”, in spelling anyway — let’s follow the same steps as above to give them a value.

SNEED:
  Step 1 – retain the first letter (“S”) and eliminate the “E”: SND
  Step 2 – “N” gets a value of 5 and “D” gets a value of 3
  SNEED = “S53”

SCHMIDT
  Step 1 – retain the first letter (“S”) and eliminate the “H” and “I”: SCMDT
  Step 2 – eliminate the duplicate value letters (S/C, both 2, and D/T, both 3): SMD
  Step 3 – “M” gets a value of 5 and “D” gets a value of 3
  SCHMIDT = “S53”

Soundex is most useful for matching English-based names. It is less useful for other strings and there are other algorithms for phonetic matching — they are just not built into SAS’ SQL!

**ONE MORE WORD ABOUT WHERE**

All that we’ve learned about WHERE clauses so far can be applied to columns that are not in the SELECT list of columns. This can be a handy way of subsetting a table, but can make debugging a bit more challenging since you cannot confirm that your selection logic worked properly. The examples above are quite simple, but a complicated WHERE with a number of columns referenced and multiple ANDs and ORs could pose a problem. It is generally good practice to initially include the WHERE columns in the SELECT statement so that the logic can be verified. Then those columns can be removed later.

**DATASET OPTIONS IN THE FROM CLAUSE**

Any dataset option is valid on the tables in the FROM clause. As the example below shows, the WHERE clause in the query and the WHERE option on the table in the FROM clause return are both valid and return identical results.
select *
from SASClass.Inmates
where Sex eq 'M';

select *
from SASClass.Inmates
where Sex eq 'M';

There are a couple situations where the use of dataset options may be preferred to the equivalent syntax in the query itself. Both have to do with columns that are named with SQL reserved words. Suppose that our charge data had a court case number stored in a column named Case and we wanted to get a set of rows that had a value in that field. The following query fails.

create table WithCaseNumber as
select *
from ChargeData
where year(BookingDate) eq 2004 and
  Case ne '';

We can’t use the column name Case in the SELECT statement or WHERE clause because it is a reserved word. But, we a couple ways we can get around this using dataset options.

We can rename the column with a RENAME option or use, like in the example at the top of the page, use a WHERE option. Either of the following queries would produce the desired results.

select *
from ChargeData(rename=(Case=CaseNumber)
where year(BookingDate) eq 2004 and
  CaseNumber ne '';
select *
from ChargeData(where=(Case ne ''))
where year(BookingDate) eq 2004;

SORTING YOUR DATA

Up to now, everything we’ve seen of SQL could be done in a single datastep. So far, we’ve looked at SELECT, FROM and WHERE. These all play a role in determining what information is going to be included in the results of the query. The ORDER BY clause takes the rows selected by those three and sorts them. The ORDER BY clause is optional and if it is used it must follow the WHERE clause, if used, or the FROM clause, if WHERE is not used.

select *
from SASClass.Charges
where ChargeType eq 'A'
order by FIM;

This query would select all the assault charges and sort them by the type of warrant (FIM: felony, investigation, misdemeanor).

As you can probably guess, the syntax of the ORDER BY clause is similar to the BY statement of PROC SORT. The column, or columns, you want the rows sorted by are listed with multiple columns separated by commas.

Let’s say that we wanted to be a bit more selective in our sorting above and we wanted to look at a list of the agencies that caused the inmate to come to jail for each warrant type. We can just add another column, OrgAgency, to the order list.

select OrgAgency format=$Agency.,
  BookNum,
  FIM
from SASClass.Charges
where ChargeType eq 'A'
order by FIM,OrgAgency;

Now our results are sorted by warrant type and then by agency: first all the felonies by agency, then all the investigations by agency, then all the misdemeanors by agency. Here are some partial results:
Look closely at the results and you’ll see that we probably didn’t quite get what we wanted. We forgot that there are multiple types of each warrant type: “FB”, “FC”, etc. for felonies and so on. There is a sorted list of agencies for each of the subtypes.

What we probably wanted was a list of all the agency values together regardless of the subtype of warrant. How would we do this with PROC SORT? You’d have to make a new column that had only the first character of the warrant type and sort by that. Or, you could create a format that mapped all the warrant types to that first character. The point is that there would be another step involved.

Why mention this unless there’s a way to do it with SQL? Well, here’s a powerful feature of ORDER BY in SQL: you can use functions in the ORDER BY clause. This allows us to get the results below by tweaking our query just slightly.

```
select OrgAgency format=$Agency., BookNum, FIM
from SASclass.Charges
where ChargeType eq 'A'
order by substr(FIM,1,1), OrgAgency;
```

Now we are just sorting by the first character of the warrant type and then by agency. We didn’t need to create another column or do any additional steps to get what we wanted.

### Alphabetic Sorting

We’ve just looked at the ability to use functions in the ORDER BY clause. Now, let’s see how to take advantage of that to perform a task that is commonly desired.

Let’s say our booking officers were willy-nilly in the way that they entered the charge description: sometimes all upper-case and sometimes all lower-case. If we wanted a sorted list of charge, we could use ORDER BY to get that list.

```
select ChargeDesc
from MixedCase
order by ChargeDesc;
```

Sorted by charge description, but the results are far from what we want.

We have “AGGRESSIVE BEGGING” at the top of the list, but it’s also more than 6,000 rows down as “aggressive begging” — a conundrum with PROC SORT for years! Again, we could create a new variable and sort by that or use a function in the ORDER BY clause to get the desired order.

```
select ChargeDesc
from MixedCase
order by upper(ChargeDesc);
```

There are some things to notice about this new result:
- the results are now in the desired order
- the original case has not been changed – that is determined by the SELECT statement
- there can be a “randomness” to the groups of values – look at “aggressive begging”
- officers don’t always spell that well – (achol, alcohl)

Note: if you want to get rid of the “randomness” and group all the upper case together (“AGGRESSIVE BEGGING”) and all the lower case together (“aggressive begging”) you can sort by both the upper-cased column and the original column.
select ChargeDesc
from MixedCase
order by upper(ChargeDesc),
ChargeDesc;

Now, within each charge description the upper case values will be first, followed by the lower case values. No more “randomness”!

You’ll notice that all the queries above give a strange note in the log:

NOTE: The query as specified involves ordering by an item that doesn’t appear in its SELECT clause.

This is because the upper-cased value of ChargeDesc was never named in the SELECT statement, only in the ORDER BY clause.

Note: the UPPER and LOWER functions are SQL standard functions and cannot be used in elsewhere in SAS. They operate the same as the SAS UPCASE and LOWCASE functions. UPCASE and LOWCASE can also be used in SQL queries.

The DESC Option of ORDER BY

The default sort order is ascending. To get a descending sort, use the DESC option following the column name.

select OrgAgency,
BookNum,
FIM
from SASclass.Charges
where ChargeType eq 'A'
order by substr(FIM,1,1) desc,
OrgAgency;

This is the same query we used to get a list of agencies in warrant type order. By default, the felonies (“F…”) came first. Now, the misdemeanors (“M…”) will come first, followed by investigations (“I…”) and felonies.

The DESC option can be added to as many columns as needed, always following the column name and before the comma or semicolon. For example, to list the agencies in reverse order and keep misdemeanors listed first we simply add another DESC:

select OrgAgency format=$Agency.,
BookNum,
FIM
from SASclass.Charges
where ChargeType eq 'A'
order by substr(FIM,1,1) desc,
OrgAgency desc;

The result table would be something like this (see right). The misdemeanors come first and within the misdemeanors, the agencies are in descending order.

Note: the placement and spelling of the descending option is different than PROC SORT. In that procedure, the DESCENDING option precedes the variable name.

Aggregate Functions

You can use aggregate (or summary) functions to summarize the data in your tables. These functions can act on multiple columns in a row or on a single column across rows. Here is the list of aggregate functions.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Avg</td>
<td>average of values</td>
</tr>
<tr>
<td>Count</td>
<td>number of non-missing values</td>
</tr>
<tr>
<td>CSS</td>
<td>corrected sum of squares</td>
</tr>
<tr>
<td>CV</td>
<td>coefficient of variation</td>
</tr>
<tr>
<td>NMiss</td>
<td>number of missing values</td>
</tr>
<tr>
<td>Prt</td>
<td>probability of a greater absolute value of Student’s t</td>
</tr>
<tr>
<td>Range</td>
<td>range of values</td>
</tr>
<tr>
<td>Std</td>
<td>standard deviation</td>
</tr>
</tbody>
</table>
Freq  (same as Count)
Max  maximum value
Mean  (same as Avg)
Min  minimum value
N  (same as Count)

- StdErr  standard error of the mean
- Sum  sum of values
- T  Student’s t value for testing that the population mean is 0
- USS  uncorrected sum of squares
- Var  Variance

Note: the SUMWGT function is not listed here as there is no provision for a WEIGHT statement in SQL and the results of SUMWGT are the same as COUNT.

We’ve already looked at using SAS functions in a SELECT statement to create new columns based on the value of an existing column. Aggregate functions create new columns as well, either by summarizing columns across a single row or by summarizing a column down multiple rows.

If more than one column is included in the function argument list then that operation is performed for each row in the table. In the table below, each row in the Charges table gets a new column, OffDays, that is the sum of GoodTime and CreditDays.

```
select *,
    sum(GoodTime, CreditDays) as OffDays
from SASClass.Charges;
```

If the argument list contains a single column the operation is performed down all the rows in the table. The following query would summarize Bail, in a number of different ways, for the entire table. Notice that the results of this query is a single row containing the sum, mean and maximum values of Bail, as well as the number of rows with a missing value for the Bail column.

```
select sum(Bail) as TotalBail,
    mean(Bail) as MeanBail,
    max(Bail) as MaxBail,
    nmiss(Bail) as NoBailSet
from SASclass.Charges;
```

Note: you can reference as many aggregate functions as you need in a single query and they don’t all have to act on the same column.
You’ll notice in the preceding query that no columns were listed other than those using the aggregate functions. Remember, we’re telling SQL to summarize down all the columns in the table. How would it be interpreted if we asked for some other columns as well? For instance,

```sql
select BookNum, 
  ChargeNum, 
  sum(Bail) as TotalBail 
from SASClass.Charges;
```

The query is saying two things, “Give me the booking number and bail amount for all the rows in the table” and “Give me the sum total of bail for the whole table.” One of those questions would return back 11,000+ rows while the other would return one. The actual result is that both requests are granted. First, the summary function will be computed and the total bail will be calculated. Then, that result will be attached, as a new column, to every row in the output.

<table>
<thead>
<tr>
<th>BookNum</th>
<th>ChargeNum</th>
<th>Bail</th>
<th>TotalBail</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>$1,050</td>
<td>$31,157.597</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>.</td>
<td>$31,157.597</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>.</td>
<td>$31,157.597</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>$500</td>
<td>$31,157.597</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>$500</td>
<td>$31,157.597</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>$500</td>
<td>$31,157.597</td>
</tr>
<tr>
<td>7</td>
<td>3</td>
<td>.</td>
<td>$31,157.597</td>
</tr>
</tbody>
</table>

You’ll also get a note in the log telling you that the query had to run through things more than once:

NOTE: The query requires remerging summary statistics back with the original data.

**THE GROUP BY CLAUSE**

By default, summary functions work across all the rows in a table. You can summarize groups of data with the GROUP BY clause. For example,

```sql
select BookNum, 
  sum(Bail) as TotalBail 
  from SASClass.Charges 
  group by BookNum;
```

Partial results of the query above are shown here:

Notice that there is now one row per booking number and that the bail amounts have been summarized. Remember from the previous section that without the GROUP BY clause the table total bail would have been added to each row of the table.

The GROUP BY clause also orders the rows by the values of the grouped columns. We’ll see later that there is also an ORDER BY clause if you want a different sort order.

There is no requirement that the order of GROUP BY columns matches the order of the SELECT columns.

You will almost always want to include all non-summary columns from your SELECT statement in the GROUP BY clause. Let’s look at the following query to see what happens if you do not.

```sql
select FIM, 
  ChargeType, 
  mean(Bail) as TotalBail 
  from SASClass.Charges 
  group by FIM;
```

This query will pass through the table twice and do the following:
- calculate the mean bail for each value of FIM (pass 1)
- return all rows of the original table (pass 2)
ordered by FIM
- with the appropriate value (FIM-specific) of TotalBail attached to each row

Notice that the FIM total bail has been added to each row. Chances are, however, that this is not what you wanted from the query and you need to add the charge type to the GROUP BY clause.

We mentioned a few pages back that there are instances where eliminating or having an incomplete GROUP BY can be used to your advantage. Let’s assume that we have a table with one row per charge type containing the total bail. We can make use of the summary behavior to get the percent of total bail that each charge type represents. This query

```sql
select ChargeType format=$Charge.,
       Bail format=dollar15.,
       Bail / sum(Bail) as PctOfTotal format=percent9.2
from BailGroups;
```

adds the table total bail to each row of the table. Instead of using this as a separate field, as we’ve done in the past, we can use this value in an expression to calculate the percentage of total bail.

As we’ve seen before this query will make two passes through the table – first to calculate the total bail and then to write out each row of the table.

### The Special Case of COUNT()

The `COUNT` function is a little different than other summary functions in a couple different ways. First, it can accept either a character or numeric column name as the argument – the other functions require a numeric column. The `COUNT` function returns the number of non-missing values of the specified column. Secondly, you can pass an asterisk (*) to the function and get a count of the total number of rows in the table (or group).

The following query counts non-missing value rows for two numeric columns (GoodTime and SentenceDate), a character column (ChargeDesc) and counts all the rows in the table (*).

```sql
select count(GoodTime) as GoodTime_Count format=comma7.,
       count(ChargeDesc) as ChargeDesc_Count format=comma7.,
       count(SentenceDate) as SentenceDate_Count format=comma7.,
       count(*) as TotalRow_Count format=comma7.
from SASclass.Charges;
```

We can see from the output that there are no missing values for good time days or charge description but, as would be expected, there are a number of missing values for the sentence date.

We can mix and match the scope of `COUNT` (column-specific counts vs total counts) in the same query. This query gets the column-specific non-missing counts and divides them by the total row count to get the percentage of non-missing values. We’re also using a GROUP BY clause, so the counts will be specific to each value of charge type.

```sql
select ChargeType format=$Charge.,
       count(ChargeDesc) as ChargeDesc_Count format=comma7.,
       count(SentenceDate) as SentenceDate_Count format=comma7.,
       calculated SentenceDate_Count / count(*) as SentenceDate_Pct format=percent9.2
from SASclass.Charges
group by ChargeType;
```
Note: There is one other difference between COUNT and other aggregate functions. In other aggregate functions passing multiple column names to the function causes it to calculate the desired statistic across those columns for each row in the table. The COUNT function is not designed to operate across columns and will not produce meaningful results if multiple columns are passed to it.

THE DISTINCT OPERATOR

While DISTINCT isn’t really a summary function, we’ll discuss it here because it has some of the properties of a summary function.

```
select distinct Facility
from SASClass.Bookings;
```

The first query above would return a list of all the unique values of facility:

No counts, just a list. You can have more than one column listed after the DISTINCT operator and you would get a list of all combinations of the columns. If a column had a permanent format associated with it, or you used the FORMAT option to assign a format, the distinct value list would have formatted values. For instance, this query returns formatted values of the unique combinations of race and sex.

```
select distinct Race format=$Race.,
          Sex format=$Gender.
from SASclass.Inmates;
```

Note: When multiple columns are listed with DISTINCT, only the actual value combinations are returned – not all possible combinations. If, in the Inmates table used above, there were no Native American females that row would not be in the result set.

```
select distinct Facility as Facility format=$Secure.
from SASclass.Bookings;
```

If you look at the distinct facilities list from the first example you can see that we end up with the same number of “distinct” values in the list. If you matched up the rows in the two outputs you can conclude that the values “4”, “5”, “M”, “P”, “R” are formatted “Secure” and “H” and “W” are formatted “Alternative”.

The following query demonstrates the use of DISTINCT within the COUNT function. This query will return the number of unique values of Facility – in this case, 7.

```
select count(distinct Facility)
from SASClass.Bookings;
```

Note: you cannot list more than one column in the COUNT function. If you wanted the number of distinct combinations of race and gender in your data the following query would generate an error:

```
select count(distinct Race, Sex)
```
from SASclass.Inmates;

You could get the count by concatenating the two columns into one and getting a count of that value. For instance,

```sql
select count(distinct Race||Sex)
from SASclass.Inmates;
```

The query above returns a value of 10, which we can see from the example above is the number of values of race and gender.

**The Calculated Keyword**

There is an important timing feature of SQL to keep in mind before we discuss the `CALCULATED` option. The `SELECT` statement and `WHERE` and `GROUP BY` clauses are acting on columns as they "come into" the query. The `ORDER BY`, and soon to be discussed `HAVING`, clauses which act on columns as they "leave" the query. This means that `SELECT`, `WHERE` and `GROUP BY` all need to reference columns that are in the tables referenced in the query.

In an earlier query we used the `SUM` function to add good time and credit days to get a new column, OffDays. We might try to write a query to select records that had some "off days" as follows:

```sql
create table OffDays as
select *,
    sum(GoodTime,CreditDays) as OffDays
from SASClass.Charges
where OffDays gt 0;
```

We would have seen an error because the `WHERE` clause doesn’t know what OffDays is – it’s not in the Charges table.

The `CALCULATED` option is simply a shortcut to SQL to say, "replace this with the expression that created this new column." The following two `WHERE` clauses are equivalent:

```sql
create table OffDays as
select *,
    sum(GoodTime,CreditDays) as OffDays
from SASClass.Charges
where sum(GoodTime,CreditDays) gt 0;
```

```sql
where calculated OffDays gt 0;
```

You must also use `CALCULATED` in the `SELECT` statement if you want to reference another column that was created in the `SELECT` statement. Let’s say we wanted to create another column that was simply a flag ("*") turned on if the new OffDays column was greater than zero. The following query creates the OffDays column and then references it to create the flag, OffDayFlag.

```sql
select *
    sum(GoodTime,CreditDays) as OffDays,
    case
        when calculated OffDays gt 0 then '*'
        else ' '
    end as OffDayFlag
from SASClass.Charges;
```

One other thing to note about using `CALCULATED` column references. The column must have been created in the `SELECT` prior to its being referenced with a `CALCULATED` option. Switching the order of the columns in the query above would cause an error. Remember, that `CALCULATED` tells SQL to "replace the name with the expression" – if the expression hasn’t been seen yet, the substitution cannot happen.

**Relative Column Referencing – a GROUP BY Shortcut**

Instead of using column names in the `GROUP BY` clause you can use the column position. The following two `GROUP BY` clauses are equivalent:
Relative column referencing is often a nice little time saver. All you have to do is use the position of the column you want rather than the column name. It is most useful when you're grouping on calculated columns and you don't want to type "calculated" over and over again.

There are a couple warnings about using relative referencing:

- First, it can be a bit of a troubleshooting issue as you have to look back and forth between the `SELECT` and `GROUP BY` to see what columns you're really grouping on.
- Second, if you decide to switch the order of the columns in your `SELECT` statement, the row order in the result set may also change. Take the example above:

```
select FIM, ChargeType, mean(Bail) as TotalBail
from SASClass.Charges
group by 1,2;

select FIM, ChargeType, mean(Bail) as TotalBail
from SASClass.Charges
group by FIM, ChargeType;
```

In both of the queries above the result rows are sorted by ChargeType and FIM.

Just some things to keep in mind when you use relative referencing rather than column name referencing in the `GROUP BY` clause.

### GROUP BY and ORDER BY Together

The `GROUP BY` clause has an implied sort and the results are displayed in that order. If you use both `GROUP BY` and `ORDER BY` in the same query, the `ORDER BY` determines the sorted order of the result.

You can see the results of the two methods in the query snippets below. The first results are grouped by the type of warrant (FIM) and are displayed in the alphabetical order of the values of that column ("F", "I", "M"). In the second query the `ORDER BY` clause which will sort the result in descending order of mean bail.

### The HAVING Clause

To select rows based on the results of a summary function, use the `HAVING` clause. The `HAVING` clause is similar to the `WHERE` clause in that it allows you to select rows to keep in the result set of your query. It is an optional clause and is placed after the `GROUP BY` clause.

```
select BookingMonth,
       sum(NumCharges) as TotalCharges
from SASClass.Charges
group by BookingMonth
having TotalCharges gt 275;
```
In the query above the **HAVING** condition (TotalCharges > 275) is evaluated after the query has run and the rows have been grouped and then only rows that meet the condition are written to the result set.

It’s important to stress the difference between **HAVING** and **WHERE**. The **WHERE** clause selects rows as they come into the query. It has to reference columns that exist in the query table or are calculated using columns in the query tables. It cannot reference summary columns. **HAVING**, on the other hand, references summary columns as the rows go out of the query.

What happens if you use **HAVING** on non-summary columns? A lot depends on the rest of the **SELECT**. If you do not have any summary columns, the **WHERE** and **HAVING** clauses will produce identical results. For example, these two queries produce exactly the same result.

```
select Race
from SASclass.Inmates
where Sex eq 'M';
```

```
select Race
from SASclass.Inmates
having Sex eq 'M';
```

There is no summarization of rows, so the values going in and the values coming out of the query are the same.

There are some big differences, however, when there is a summary function in the **SELECT** statement and the **HAVING** clause references a non-summary column. Let’s look at the queries below.

```
select Race,
    count(*) as Total
from SASclass.Inmates
where Sex eq 'M'
group by Race;
```

```
select Race,
    count(*) as Total
from SASclass.Inmates
having Sex eq 'M';
```

<table>
<thead>
<tr>
<th>Race</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>91</td>
</tr>
<tr>
<td>B</td>
<td>297</td>
</tr>
<tr>
<td>I</td>
<td>31</td>
</tr>
<tr>
<td>U</td>
<td>6</td>
</tr>
<tr>
<td>W</td>
<td>1358</td>
</tr>
<tr>
<td>x</td>
<td>2</td>
</tr>
<tr>
<td>y</td>
<td>3</td>
</tr>
</tbody>
</table>

The query above will process only rows where the gender is male. The **GROUP BY** clause applies only to males.

The results of the query to the right are very different and lead to two very interesting questions.

Let’s think about those questions. If **HAVING** references a non-summary column that is not in the **SELECT** it “adds” it to the list of columns. So, the query on the right above is equivalent to this one, except that the sex column is not kept.

```
select Race,
    sex,
    count(*) as Total
from SASclass.Inmates
group by Race
having sex eq 'M';
```

Remember that when a **GROUP BY** clause is incomplete? The entire table is returned to the result set and the summarized value is added to every row. So, we get the total number of rows for each race and that value is added back to each row. Then, the **HAVING** is applied and the male rows are kept. The result set contains all the male rows with the count for all the rows.

Can you see why we want to reserve **HAVING** for summary column filtering and **WHERE** for non-summary column filtering?

**SUMMARY FUNCTIONS IN HAVING**

We just saw that there is a big danger in using non-summary columns in the **HAVING** clause. However, using summary columns in the **HAVING** that are not in the **SELECT** can be a very handy thing.
Let's look at a way to get all the charges that have a bail amount greater than the mean amount.

```sql
select BookNum, ChargeNum, Bail,
       mean(Bail) as MeanBail
from sasclass.charges;
```

Without a GROUP BY the summary value is added to all the rows in the result table.

<table>
<thead>
<tr>
<th>BookNum</th>
<th>ChargeNum</th>
<th>Bail</th>
<th>MeanBail</th>
</tr>
</thead>
<tbody>
<tr>
<td>19200002</td>
<td>1</td>
<td>10500</td>
<td>4491.375938</td>
</tr>
<tr>
<td>19200006</td>
<td>1</td>
<td>4491.375938</td>
<td></td>
</tr>
<tr>
<td>19200006</td>
<td>2</td>
<td>4491.375938</td>
<td></td>
</tr>
<tr>
<td>19200046</td>
<td>1</td>
<td>550</td>
<td>4491.375938</td>
</tr>
<tr>
<td>19200016</td>
<td>1</td>
<td>500</td>
<td>4491.375938</td>
</tr>
<tr>
<td>19200016</td>
<td>2</td>
<td>500</td>
<td>4491.375938</td>
</tr>
</tbody>
</table>

```sql
select BookNum, ChargeNum, Bail,
       mean(Bail) as MeanBail
from sasclass.charges
having Bail gt MeanBail;
```

We can add a HAVING clause to pick up the rows that we want.

<table>
<thead>
<tr>
<th>BookNum</th>
<th>ChargeNum</th>
<th>Bail</th>
<th>MeanBail</th>
</tr>
</thead>
<tbody>
<tr>
<td>19200238</td>
<td>1</td>
<td>50000</td>
<td>4491.375938</td>
</tr>
<tr>
<td>19200306</td>
<td>1</td>
<td>5000</td>
<td>4491.375938</td>
</tr>
<tr>
<td>19200306</td>
<td>2</td>
<td>5000</td>
<td>4491.375938</td>
</tr>
<tr>
<td>19200306</td>
<td>3</td>
<td>10000</td>
<td>4491.375938</td>
</tr>
<tr>
<td>19200605</td>
<td>1</td>
<td>5000</td>
<td>4491.375938</td>
</tr>
</tbody>
</table>

But, we don’t really need that mean bail column repeated on every row in the table. Well, we can put the summary function in the HAVING clause rather than in the SELECT and get the same result.

```sql
select BookNum, ChargeNum, Bail
from sasclass.charges
having Bail gt mean(bail);
```

We get the same rows as we did with the query above, but we've eliminated that repetitive column.

We can go a step further here. By adding an incomplete GROUP BY clause, one which does not reference all the columns in the SELECT statement, and keep a summary function in a HAVING clause we can filter our result set based on the grouped values of the summary statistic. If, in the example above, we wanted to get charges that had a bail higher than the mean bail for the warrant type of the charge we could add FIM to the SELECT and a GROUP BY. Just as above, the mean bail would be implicitly added to each row, but this time it would be the mean of the GROUP BY column.

```sql
select BookNum, ChargeNum, FIM, Bail
from sasclass.charges
group by FIM
having Bail gt mean(Bail);
```

The mean bail for FIM value is added to each row and the result set is filtered based on the group mean.

<table>
<thead>
<tr>
<th>BookNum</th>
<th>ChargeNum</th>
<th>FIM</th>
<th>Bail</th>
</tr>
</thead>
<tbody>
<tr>
<td>19000351</td>
<td>1</td>
<td>FB</td>
<td>40000</td>
</tr>
<tr>
<td>19040101</td>
<td>2</td>
<td>FB</td>
<td>25000</td>
</tr>
<tr>
<td>19000308</td>
<td>1</td>
<td>FB</td>
<td>20000</td>
</tr>
<tr>
<td>19000310</td>
<td>1</td>
<td>MB</td>
<td>2000</td>
</tr>
<tr>
<td>19000316</td>
<td>2</td>
<td>MB</td>
<td>1500</td>
</tr>
<tr>
<td>19401325</td>
<td>1</td>
<td>MB</td>
<td>1250</td>
</tr>
<tr>
<td>20010210</td>
<td>1</td>
<td>MB</td>
<td>2500</td>
</tr>
</tbody>
</table>

CONCLUSION

It’s amazing what you can do with those seven keywords and a few options. The power of SQL to organize, summarize and order your data all in one query can save you considerable time in developing your applications.
You can learn a lot about SQL and its implementation in SAS in just a few pages, but there is so much more: multiple table joins, embedded queries, passing queries to other databases, etc. I hope that this paper has whetted your appetite for learning more about SQL.

**AUTHOR CONTACT INFORMATION**

Pete Lund  
Looking Glass Analytics  
215 Legion Way SW  
Olympia, WA 98501  
(360) 528-8970  
(360) 570-7533  
pete.lund@lgan.com  
www.lgan.com

**ACKNOWLEDGEMENTS**

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.