PROC SQL

According to the SAS documentation, Structured Query Language (SQL) is a "standardized, widely used language that retrieves and updates data" in tables and databases. It was developed nearly fifty years ago by Raymond Boyce and Donald Chamberlin[1] following the publishing of relational theory by E. F. Codd at IBM[2]. PROC SQL is the SAS implementation of SQL and can be used with any SAS data set. Nearly all of the data management steps performed in the SAS datastep and some of the calculations performed in other SAS procedures can be accomplished through using PROC SQL. PROC SQL allows the user to create reports, calculate summary statistics, access, combine, modify, or create tables, and more[3]. The primary benefit of PROC SQL is that it allows the user to perform simple calculations and data management steps in a single call of the procedure. It is particularly useful for merging and querying large data sets. However, there are some goals that are better accomplished with a datastep or other SAS procedures.

The purpose of this section is to examine PROC SQL and use it to access and manage data sets. All programs and data sets used for these notes are available from the course website. New files that we have not used before are gpa_names.xlsx and cpt.xlsx.

Basics of PROC SQL

Because this procedure implements SQL, there are some differences in coding conventions from other Base SAS procedures. Some of these are detailed below:

[1] https://www.businessnewsdaily.com/5804-what-is-sql.html
[2] https://www.seas.upenn.edu/~zives/03f/cis550/codd.pdf
[3] http://support.sas.com/documentation/cdl/en/sqlproc/63043/HTML/default/viewer.htm#p1typbj1zqaum2n13o7mph0tdqsc.htm

- When a set of PROC SQL code is run, the code is executed immediately (without a RUN statement) and continues to run until a QUIT statement.
- PROC SQL statements are divided into clauses. Only the final clause is followed by a semicolon and items within clauses are separated by commas.
- The SELECT statement retrieves data and automatically displays output unless the NOPRINT option is specified. There are no observation/row numbers displayed by PROC SQL.
- The order of clauses within statements matters!

Statements within PROC SQL include the following:

- CONNECT and DISCONNECT establish and end a connect with a DBMS, respectively.
- CREATE TABLE creates a new table for use outside PROC SQL.
- DROP deletes tables (views and indexes).
- DELETE removes rows and INSERT adds rows to a table (or view).
- SELECT selects columns and rows of data from tables (and views).
- Other statements include ALTER TABLE, CREATE INDEX, CREATE VIEW, DESCRIBE, EXECUTE, RESET, UPDATE, and VALIDATE.

This section will primarily discuss CONNECT, DISCONNECT, CREATE TABLE, INSERT and SELECT. Additional statements can be investigated in the SAS documentation.

Importing data sets

There are a few different ways to get data into a SAS data set using PROC SQL. Below are some examples:

- Create a table using the CREATE TABLE statement. Include data using the INSERT statement with the VALUES clause.

```
libname procsql "C:\Users\bhitt\Desktop\STAT 850\PROC SQL";

proc sql;
   create table procsql.set1
      (HS num,
       College num);
   title 'PROCSQL.SET1 Table';
   insert into procsql.set1
      values(3.04, 3.1)
      values(2.35, 2.3)
      values(2.7, 3.0)
      values(2.55, 2.45)
      values(2.83, 2.5)
      values(4.32, 3.7)
      values(3.39, 3.4)
      values(2.32, 2.6)
      values(2.69, 2.8)
      values(2.83, 3.6)
      values(2.39, 2.0)
      values(3.65, 2.9)
      values(2.85, 3.3)
      values(3.83, 3.2)
      values(2.22, 2.8)
      values(1.98, 2.4)
      values(2.88, 2.6)
      values(4.0, 3.8)
      values(2.28, 2.2)
      values(2.88, 2.6);
   select * from procsql.set1;
quit;
```

Notes:

- The CREATE TABLE statement defines the names and attributes of the table columns inside parentheses. You can specify a column's name, type, length, informat, format, and label. A numeric column is usually specified using `numeric` (or `num`), with some more specific options (such as `integer`, `decimal`, or `float)` available as well.
- The TITLE statement can be used inside PROC SQL just as it has been used before.
- The INSERT statement inserts rows into the specified table using the VALUES clause. Without the INSERT statement, the table created in the CREATE TABLE statement has no rows.
- A separate VALUES clause is used for each row of the data set and they are not separated by commas. Values being added to the data set are contained inside a set of parentheses and separated by commas.
- Each column in the data set must have a value specified. A missing character value can be specified by a space in single quotation marks. A missing numeric value can be specified by a period.
- The final VALUES clause is followed by a semicolon. The semicolon does not need to be on a separate line as it does when using a datastep.
- The SELECT statement prints the data set that was created. A basic SELECT statement includes SELECT and FROM clauses. The asterisk notation tells PROC SQL to print all fields/columns from the data set specified. One can also specify multiple column names separated by commas. This eliminates the need for PROC PRINT.

- Insert data using the INSERT statement with the SET clause.

```
proc sql;
   create table procsql.set2
      (HS num,
       College num);
   insert into procsql.set2
      set HS=3.04, College=3.1
      set HS=2.35, College=2.3
      set HS=2.7, College=3.0
      set HS=2.55, College=2.45
      set HS=2.83, College=2.5
      set HS=4.32, College=3.7
      set HS=3.39, College=3.4
      set HS=2.32, College=2.6
      set HS=2.69, College=2.8
      set HS=2.83, College=3.6
      set HS=2.39, College=2.0
      set HS=3.65, College=2.9
      set HS=2.85, College=3.3
      set HS=3.83, College=3.2
      set HS=2.22, College=2.8
      set HS=1.98, College=2.4
      set HS=2.88, College=2.6
      set HS=4.0, College=3.8
      set HS=2.28, College=2.2
      set HS=2.88, College=2.6;
   title 'PROCSQL.SET2 Table';
   select * from procsql.set2;
quit;
```

Notes:

- A separate SET clause is used for each row of the data set and they are not separated by commas. Values being added to the data set are specified by `variable=value` and are separated by commas.
- This notation does not require a value to specified for each column. A missing value can be specified by simply omitting a variable from the SET clause.

  - The final SET clause is followed by a semicolon. The semicolon does not need to be on a separate line as it does when using a datastep.
  - The VALUES and SET clauses will usually not be used to manually create an entire data set from scratch. Instead, you might use these clauses to insert a few new observations into an already existing data set.

- Non-numeric variables: There is no $ notation used in PROC SQL. Character values are specified in the CREATE TABLE statement using `character(width)` or `char(width)`. Date values are specified in the CREATE TABLE statement using `date` and formatting options. For example, one could use the following code to manually enter names and dates in the GPA data set.

```
proc sql;
   create table procsql.set3
      (HS num,
       College num,
       First char(10),
       Last char(15),
       date num informat=date9. format=date9.);
   insert into procsql.set3
      set HS=3.04, College=3.1, First="first1",
          Last="last1", date="01dec2011"d
      set HS=2.35, College=2.3, First="first2",
          Last="last2", date="05feb2012"d
      set HS=2.7, College=3.0, First="first3",
          Last="last3", date="17jun2012"d;
   title 'PROCSQL.SET3 Table';
   select * from procsql.set3;
quit;
```

  Notes:

  - The number in parentheses that follows `char` tells SAS the permitted length (number of characters) of the character variable.

  - Values for character variables are specified within quotation marks (single or double).
  - The date must be specified as numeric in the CREATE TABLE statement, but the specifics of the date value are given by the `format` and `informat` options. The informat specifies how SAS should read in the date. The format specifies how SAS should output the date.
  - The date9. format/informat uses two digits for the day, a three letter abbreviation for the month and four digits for the year. Values for the date are given within quotation marks and must be followed by a "d" to denote a date. For other date formats, see the SAS documentation.[4]
  - Dates are most often used in PROC SQL to create queries. For example, imagine that you have a data set that includes the dates of patient blood draws and you want to run a query to find blood draws within a week of the patient being diagnosed with a disease of interest.

- Connecting to a database management system (DBMS).

```
proc sql outobs=5;
   connect to excel (path="C:\data\gpa_names.xlsx");
   title 'GPA_NAMES Table';
   create table procsql.set4 as
      select * from connection to excel
         (select * from [gpa_names$A1:D21]);
   disconnect from excel;
   select * from procsql.set4(obs=10);
quit;
```

  Notes:

  - This method uses a pass-through facility to send commands directly to the DBMS for execution. According to SAS, the preferred method of accessing a DBMS (such as Oracle) is using the LIBNAME statement. However, this method

[4]http://support.sas.com/documentation/cdl/en/lrcon/65287/HTML/default/viewer.htm#p1wj0wt2ebe2a0n1lv4lem9hdc0v.htm

does not seem to work with Excel files, delimited files, or text files.

  - The CONNECT statement opens a connection to a DBMS (Excel) by specifying the name of the DBMS and the path of the data file. The DISCONNECT statement closes the connection.
  - The CREATE TABLE statement creates a new data set called `set4` by selecting all variables from the connection to Excel. Without this statement, the table will be displayed but not accessible as a SAS data set.
  - The range of the data set must be specified within single brackets as `dataset$begin:end`.You can use `dataset$` to denote all cells in the data set.
  - The SELECT statement displays the first 10 rows of the table. You can use `outobs=10` in the PROC SQL statement to limit the contents of `set4` to only 10 rows.

- Using a datastep or PROC IMPORT, as shown previously in class.

```
proc import out=procsql.gpa datafile="C:\data\gpa.csv"
            dbms=csv replace;
   getnames=yes;
   datarow=2;
run;

proc sql;
   title 'PROCSQL.GPA Table';
   select * from procsql.gpa;
quit;
```

We can make the data set more descriptive using labels, as was done previously.

```
proc sql;
   create table procsql.set4label as
      select HSGPA label="High School GPA",
             CollegeGPA label="College GPA",
             First label="First Name",
             Last label="Last Name"
      from procsql.set4;
   title 'PROCSQL.SET4LABEL Table';
   select * from procsql.set4label;
quit;
```

The CREATE TABLE statement creates a new data set (`set4label`) from an already existing data set (`set4`) by using the SELECT/FROM clauses. The FROM clause is similar to the SET statement in a datastep, telling PROC SQL to select the specified variables from the dataset. One can relabel variables by simply specifying the name of the variable followed by `label=` and the new variable name in quotation marks.

PROC SQL also allows us to create an empty copy of an existing table by using the CREATE TABLE statement and the LIKE clause.

```
proc sql;
   create table procsql.set4new
      like procsql.set4label;
   describe table procsql.set4new;
quit;
```

The SAS log shows us that the new table (`set4new`) has the same column names and attributes as the old table (`set4label`), but it is empty.

```
121  proc sql;
122      create table procsql.set4new
123          like procsql.set4label;
NOTE: Table PROCSQL.SET4NEW created, with 0 rows and 4 columns.
124      describe table procsql.set4new;
NOTE: SQL table PROCSQL.SET4NEW was created like:
```

```
create table PROCSQL.SET4NEW( bufsize=65536 )
  (
  HSGPA num label='High School GPA',
  CollegeGPA num label='College GPA',
  First char(255) format=$255. informat=$255. label='First Name',
  Last char(255) format=$255. informat=$255. label='Last Name'
  );
125  quit;
```

This method of creating tables can be useful if you are dealing with a large data set. You can create a copy of the table, keeping the same column names and attributes with any existing formatting, and then add your own observations or make other changes. The INSERT statement can now be used to add data to the new table as was done previously. As always, make sure to examine the data after you read it in to make sure it is correct. Also, make sure to check the log for important notices.

Exporting data sets

The CREATE TABLE statement creates a table by the specified name. If the library name is not specified, the new data set is created in the WORK library. If the two-name method of referring to data sets is used, the CREATE TABLE statement will export a SAS data set to the specified library. For example, the code below creates a SAS data set (`set4export`) in the `procsql` library.

```
proc sql;
   create table procsql.set4export as
      select * from procsql.set4;
quit;
```

PROC SQL cannot create other file formats such as Excel, CSV, or text files. PROC EXPORT can be used to export SAS data sets created in PROC SQL to other file formats as shown previously.

Cereal data

PROC SQL can be used to perform the steps/calculations on the cereal data set that were shown in the Introduction notes. First, the data set is read into SAS using PROC IMPORT. Instead of using separate chunks of code to print the first five observations, adjust the nutritional content variables, sort the data, and print a subset of the data, one can use a single call to PROC SQL to accomplish all these tasks simultaneously.

```
proc import out=cereal datafile="C:\data\cereal.csv"
            dbms=csv replace;
run;

proc  sql;
   title 'Brianna Hitt, STAT 850';
   title2 'Cereal data adjusted for serving size';
   create table cereal1 as
      select ID, Shelf, Cereal,
             sugar_g/size_g as sugar,
             fat_g/size_g as fat,
             sodium_mg/size_g as sodium
      from cereal
      order by shelf, sugar;
   select shelf, cereal, sugar
      from cereal1(obs=5)
      where shelf=1;
quit;
```

Notes:

- The CREATE TABLE statement is used to create a new data set called `cereal1` that includes the ID, shelf, name, and nutritional content for each cereal.
- The names of the variables included in the new data set are specified in the SELECT clause and separated by commas. This allows variables to be specified without using the DROP or KEEP statements that are used in a datastep. All the

variables from the cereal data set can be included by using the asterisk notation.

- Variables are shown in the order given in the SELECT statement.
- The new `sugar`, `fat`, and `sodium` variables are created by providing the mathematical formula for the adjusted variable followed by AS and the name of the new variable.
- Labels and formats can also be provided for the new variables (see program for examples).
- The ORDER BY clause sorts the data set by shelf number and sugar content. PROC SORT is no longer needed to sort the data set.
- The final SELECT statement tells PROC SQL to print only the shelf, cereal name, and sugar content from `cereal1`, and only the first five observations from the first shelf. The full data set can be seen in the work library in the Explorer window.

Data summary and analysis

PROC MEANS was previously used to calculate means for the sugar, fat, and sodium variables and output the means to a new data set called `out_cereal1`. Below is the code to perform the same steps using PROC SQL.

```
proc sql;
   title2 'Means for cereal data';
   create table out_cereal1 as
      select mean(sugar) as mean_sugar,
             mean(fat) as mean_fat,
             mean(sodium) as mean_sodium
      from cereal1;
   select * from out_cereal1;
quit;
```

We can also calculate the means by shelf.

```
proc sql;
   title2 'Means for cereal data by shelf';
   create table out_cereal1 as
      select shelf,
             mean(sugar) as mean_sugar,
             mean(fat) as mean_fat,
             mean(sodium) as mean_sodium
      from cereal1
      group by shelf;
   select * from out_cereal1;
quit;
```

If the GROUP BY statement is omitted, the overall mean is displayed for every observation. We can also use PROC SQL to calculate other summary measures.

```
proc sql;
   title2 'Means for cereal data by shelf';
   create table out_cereal1 as
      select shelf,
             mean(sugar) as mean_sugar,
             min(sugar) as min_sugar,
             median(sugar) as med_sugar,
             max(sugar) as max_sugar,
             std(sugar) as sd_sugar,
             mean(fat) as mean_fat,
             min(fat) as min_fat,
             median(fat) as med_fat,
             max(fat) as max_fat,
             std(fat) as sd_fat,
             mean(sodium) as mean_sodium,
             min(sodium) as min_sodium,
             median(sodium) as med_sodium,
             max(sodium) as max_sodium,
             std(sodium) as sd_sodium
      from cereal1
      group by shelf;
   select * from out_cereal1;
quit;
```

Notes:

- Unlike PROC MEANS and other SAS procedures, the order of the statements does matter here. The order of the clauses within each statement also matters. For example, the SELECT statement must use the following order for clauses: SELECT, FROM, WHERE, GROUP BY, HAVING, ORDER BY.
- The data does not need to be sorted beforehand, like it does when using PROC MEANS.
- One can easily calculate summary statistics such as those above. Other statistics such as confidence limits are much more complicated to calculate in PROC SQL. Rather than specifying the formula for a confidence limit, it makes more sense to use PROC MEANS or PROC UNIVARIATE for such statistics. A list of available summary functions can be found in the SAS documentation[5].

Counts

PROC SQL cannot create contingency tables, but can calculate counts as was done with other summary statistics. Below is the code to create the data set of counts for the `placekick` data set.

```
proc import out=placekick datafile="C:\data\placekick.csv"
            dbms=csv replace;
run;

proc sql;
   title2 'Cross-classifications of good vs. change';
   select change, good,
          count(*) as Total
      from placekick
      group by change, good;
quit;
```

[5]http://support.sas.com/kb/25/279.html

If the `change` and `good` variables are not specified in the SELECT statement, the counts will be calculated and shown without labels, making it difficult to determine which totals correspond to which values of `change` and `good`.

Conditional execution

While conditional execution can be achieved using if-then-else statements within a datastep, the same can be done using CASE/WHEN clauses in PROC SQL. We again want to create indicator variables for the field variable (1 = "G", 0 = "T"), the location/kicking team variables (1=home team, 0=away team), and the wind variable (1=windy conditions, 0=non-windy conditions).

```
proc import out=placekick1
            datafile="C:\data\placekick_datastep.csv"
            dbms=csv replace;
run;

proc sql;
   title2 'Portion of placekick data';
   create table placekick2 as
      select *,
      case when field="G" then 1
           when field="T" then 0 else .
           end as field2,
      case when loc=team then 1 else 0
           end as home,
      case when type="O" and speed > 15 then 1 else 0
           end as wind
      from placekick1;
   select field, field2, loc, team,
          home, type, speed, wind
      from placekick2(obs=10);
quit;
```

Comments:

- Inside the SELECT clause, one use the CASE clause to provide if-then-else conditions. The clause begins with CASE and ends with END. The condition follows WHEN (similar to "if") and the value follows THEN. Each case must have an ELSE value specified or the case will not be created.
- There can be multiple WHEN clauses for each case (one can specify multiple values for each case).
- Missing values for numeric variables are specified by a period and missing values for categorical variables are specified by empty quotation marks ("").
- Because the CASE clause exists inside the SELECT clause, one must separate all variables and cases with commas. The final variable or case is not followed by a comma, but is followed by the FROM clause.
- In order to create a new variable from an if-then-else condition, one can use END AS and specify a new variable name.
- As with the datastep, one can use `and`, `or`, and parentheses to create if/then conditions.
- Variables in the output data set will appear in the order they are specified in the final SELECT statement.
- Similar to "nested" if-then-else statements, one can use "nested" CASE/WHEN clauses. This may be needed to take care of a large number of conditions.
- The binary variable `change` can be created as before. In the code below, the `diff_score` is calculated and its value is conditioned on to create the `change` variable. One does not need to use separate calls to PROC SQL, but does need to specify that `diff_score` is calculated when referring to it (such as in the CASE clause).

```
proc sql;
   title2 'Portion of placekick data';
   create table placekick3 as
      select *,
         sc_team - sc_opp as diff_score,
         case when (PAT="Y" and -1 le calculated
                     diff_score le 0) or
                    (PAT="N" and -3 le calculated
                     diff_score le 0) then 1
              else 0
              end as change,
         case when qrtr=2 or qrtr=4 then 15 + elap2
              else elap2
              end as elap3
         from placekick2;
   select sc_team, sc_opp, diff_score, PAT, change,
          field, field2, loc, team,
          home, type, speed, wind,
          qrtr, elap2, elap3
      from placekick3(obs=20);
quit;
```

- There may be instances when a set of commands need to be completed when a condition is satisfied. PROC SQL requires a separate CASE clause for each new variable. PROC SQL cannot create multiple variables from a single condition as a datastep can. Please see an example in the program.
- One can also use the `ifc` (categorical variable) or `ifn` (numeric variable) functions to create binary variables. Both functions use the form `ifn`(condition, value if true, value if false). For example, the following new variables can be listed inside the SELECT clause.

```
ifn(field="G", 1, 0) as field1,
ifc(field="G", "Y", "") as SecondCommand
```

Re-organizing data

Concatenation

The SET statement was used previously to vertically concatenate data sets with variables of the same name. The same can be done in PROC SQL by using a single CREATE TABLE statement with two SELECT clauses separated by UNION ALL. Below is the code and output illustrating the process.

```
proc sql;
   title2 'New form of the HS/College data set';
   create table procsql.gpa_plot as
      select set1.College as GPA,
             "College" as school
      from procsql.set1
      union all
      select set1.HS as GPA,
             "HS" as school
      from procsql.set1;
   select * from procsql.gpa_plot;
quit;
```

Notes:

- When using the datastep previously, two separate data sets were created, one for college and one for high school, before concatenating them. With PROC SQL, only one CREATE TABLE statement is needed.
- The first SELECT clause selects the `College` column from the `set1` data set, and creates a new column `school` with a value of “College”. The second SELECT clause selects the `HS` column from the `set1` data set and creates a new column `school` with a value of “HS”.
- The UNION clause tells SAS to combine observations from the two data sets (which now have the same variables) and the ALL option tells SAS to include all observations from the

two data sets. Without the ALL option, SAS will remove duplicates from the combined data set.

Create multiple data sets and merge

One can create multiple data sets by using multiple CREATE TABLE statements. For example, suppose we would like separate cereal data sets based on the shelf. Four CREATE TABLE statements are used to create four separate data sets based on shelf. These four separate data sets are then merged together using a final CREATE TABLE statement.

```
proc sql;
   title2 'Reorganized cereal data set';
   create table shelf1 as
      select ID as ID_new,
             sugar as sugar1,
             fat as fat1
      from cereal1 where shelf=1;
   create table shelf2 as
      select ID-10 as ID_new,
             sugar as sugar2,
             fat as fat2
      from cereal1 where shelf=2;
   create table shelf3 as
      select ID-20 as ID_new,
             sugar as sugar3,
             fat as fat3
      from cereal1 where shelf=3;
   create table shelf4 as
      select ID-30 as ID_new,
             sugar as sugar4,
             fat as fat4
      from cereal1 where shelf=4;
   create table merged as
      select shelf1.sugar1, shelf1.fat1,
             shelf2.sugar2, shelf2.fat2,
             shelf3.sugar3, shelf3.fat3,
             shelf4.sugar4, shelf4.fat4
      from shelf1, shelf2, shelf3, shelf4
      where shelf1.ID_new=shelf2.ID_new and
            shelf2.ID_new=shelf3.ID_new and
            shelf3.ID_new=shelf4.ID_new;
   select * from merged(obs=3);
quit;
```

Notes:

- PROC SQL does not allow you to join (horizontally concatenate) tables that do not share a key (an identifying variable), so we have to create a key that will match for the four data sets. We can simply subtract 10, 20 or 30 from the ID number so that we have cereals 1 through 10 for each shelf.
- The new ID is used to match the shelf data sets using a WHERE statement. This is also considered an inner join (which can be accomplished using different syntax that will be shown later in these notes).
- Without the WHERE statement, a cartesian product will be created where every observation in `shelf1` is matched with every observation in `shelf2`, `shelf3`, and `shelf4`.
- Notice the new use of the period syntax. We have previously used `libname.dataset_name` to denote a permanent SAS data set, but here we use `dataset_name.variable_name` for a different reason. In this situation, we are selecting variables from multiple data sets and the period syntax helps SAS to identify which variables are being selected from which data sets.

Joining data sets

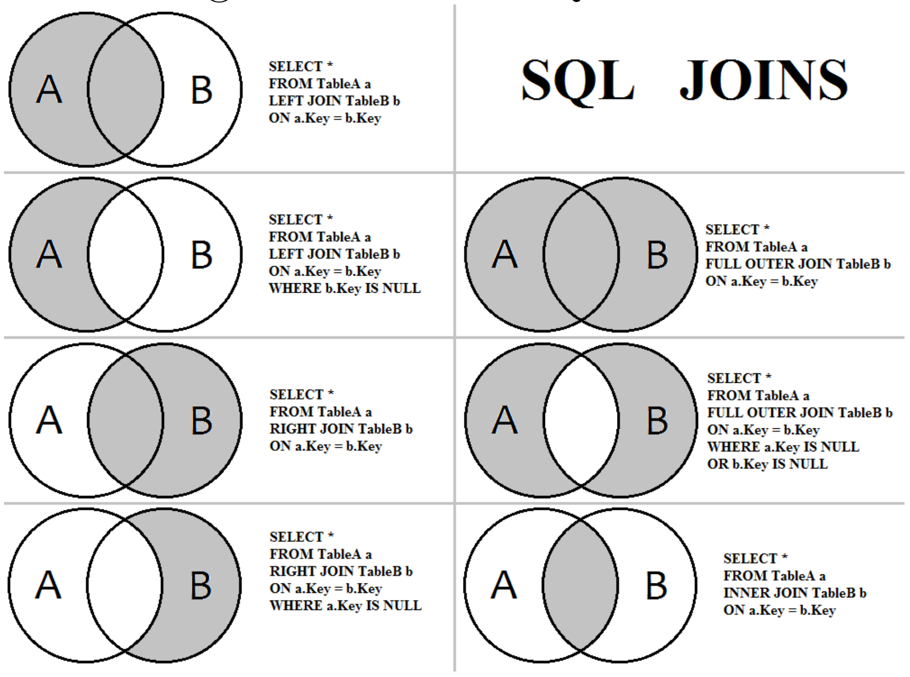
The merging of the data sets above represented a simple vertical concatenation, where we used a WHERE statement to match separate data sets. This was the same as an inner join, which is one way to join data sets. A join is a combination of rows from two or more tables, based on a shared variable (or key) between them. Below are two simple data sets which are merged together using a left join, which includes all observations from table A matched with selected information from table B:

```
proc sql;
   create table merge1
      (name char(1),
       response1 num);
   insert into work.merge1
      values("a", 1)
      values("b", 2)
      values("c", 3)
      values("d", 4)
      values("e", 5)
      values("f", 6);
   create table merge2
      (name char(1),
       response2 num);
   insert into work.merge2
      values("a", 10)
      values("a", 11)
      values("b", 20)
      values("c", 30)
      values("d", 40)
      values("e", 50);
   create table merged_set1 as
      select * from merge1 A
         left join merge2 B
         on A.name=B.name;
   select * from merged_set1;
quit;
```

Notes:

- We use two CREATE TABLE statements and two INSERT INTO statements to create the `merge1` and `merge2` data sets. The final CREATE TABLE statement creates the `merged_set1` data set by performing a left join.
- The basic syntax for any type of join is the same as above.

Figure 1: Basic SQL Joins


SQL JOINS
A
B
SELECT * FROM TableA a LEFT JOIN TableB b ON a.Key = b.Key
SELECT * FROM TableA a LEFT JOIN TableB b ON a.Key = b.Key WHERE b.Key IS NULL
SELECT * FROM TableA a FULL OUTER JOIN TableB b ON a.Key = b.Key
SELECT * FROM TableA a RIGHT JOIN TableB b ON a.Key = b.Key
SELECT * FROM TableA a FULL OUTER JOIN TableB b ON a.Key = b.Key WHERE a.Key IS NULL OR b.Key IS NULL
SELECT * FROM TableA a RIGHT JOIN TableB b ON a.Key = b.Key WHERE a.Key IS NULL
SELECT * FROM TableA a INNER JOIN TableB b ON a.Key = b.Key

First, we select variables from the first data set and provide a label (in this case, A). Then, we left join the first data set with the second data set and provide a label for the second data set (in this case, B). Finally, we tell PROC SQL that we want to join on a key, or a shared variable (in this case, name).

- The labels “A” for `merge1` and “B” for `merge2` are arbitrary. We simply need to provide some label for each data set in the join. These labels are then used in the ON clause to specify the key in the format `dataset_label.key_variable`.
- There are several other types of joins that can be performed in PROC SQL. Inner joins include only the observations that overlap between datasets A and B. Outer joins include all observations from both datasets A and B. A left join includes all observations from table A and a right join includes all observations from table B. For examples of how to perform other types of joins, see Figure 1[6] and the SAS documentation[7].

[6]https://www.solutionfactory.in/posts/Difference-between-Join-And-Union-in-SQL
[7]http://support.sas.com/documentation/cdl/en/sqlproc/63043/HTML/default/viewer.htm#p0o4a5ac71mcchn1kc1zhxdnm139.htm

Transpose a data set

The goal is create a new form of the data such that all `c_hits` values are put in a row for the same patient, and calculate means over time for each dose group. Below is the process:

```
proc sql;
   title2 'The transposed data';
   create table procsql.cpt_wide as
      select distinct patient, dose,
         sum(case when time=1 then c_hits else 0 end)
            as time1,
         sum(case when time=2 then c_hits else 0 end)
            as time2,
         sum(case when time=3 then c_hits else 0 end)
            as time3,
         sum(case when time=4 then c_hits else 0 end)
            as time4
      from procsql.cpt
      group by patient
      order by dose;
   select * from procsql.cpt_wide(obs=5);
quit;

proc sql;
   title2 'Means over time for each dose group';
   select dose,
         mean(time1) as mean1,
         mean(time2) as mean2,
         mean(time3) as mean3,
         mean(time4) as mean4
      from procsql.cpt_wide
      group by dose;
quit;
```

Notes:

- PROC SQL can perform a transpose, but it is not one of the primary benefits of the procedure. PROC TRANSPOSE provides a much simpler method for transposing a data set.

- We create dummy variables for each time point using CASE/WHEN statements. We then sum over the four dummy variables so that the value of `c_hits` is shown in all four rows for a patient.
- Because there are still four rows for each patient (which we are assuming are identical), we can use DISTINCT before the patient variable in the SELECT statement to tell PROC SQL to select only one row for each patient.
- Without the GROUP BY statement in the first call to PROC SQL, the same `c_hits` value will be displayed for every patient. Without the GROUP BY statement in the second call to PROC SQL, the same mean will be displayed for every dose.
- How can we check that our new data set is correct before reducing it to a single row for each patient?

We could have also calculated the means over time for each dose group without first transposing the data.

```
proc sql;
   title2 'Means over time for each dose group';
   select dose, time,
          mean(c_hits) as mean_chits
      from procsql.cpt
      group by dose, time;
quit;
```

General functions

Summary

We have already seen how the `max` function works, but we can do other simple mathematical operations in PROC SQL.

```
data set1;
   input x1 x2 x3;
   datalines;
   1 2 3
   4 5 6
   ;
run;

proc sql;
   title2 'Illustrate the sum, sqrt, and max functions';
   create table general_set2 as
      select x1, x2, x3,
             x1 + x2 + x3 as sum1,
             sum(x1, x2, x3) as sum2,
             sqrt(x1) as sqrt1,
             max(x1, x2, x3) as max1
      from general_set1;
   select * from general_set2;
quit;
```

Some of the shorthand notation used in other procedures is not compatible with PROC SQL. For example, we cannot use the `x1-x3` notation in PROC SQL.

Probability distributions - quantiles, probabilities, and random number generation

Below are examples of how to calculate quantiles and probabilities from a standard normal distribution.

```
proc sql;
   create table prob_set1
      (area_to_left num);
   insert into prob_set1
      values(0.975);
   title2 'Standard normal quantiles and probabilities';
   create table prob_set2 as
      select *, probit(area_to_left) as quant1,
         quantile("normal", area_to_left, 0, 1)
            as quant2,
         probnorm(calculated quant1) as prob1,
         CDF("normal", calculated quant1, 0, 1)
            as prob2
      from prob_set1;
   select * from prob_set2;
quit;
```

We cannot include functions when manually creating a new data set like we can in a datastep. Instead, we need to create a data set with a value, `area_to_left`, that can be referred to in the probability functions in a second CREATE TABLE statement.

In the Datastep notes, we simulated observations from a random normal distribution. It doesn't make sense to do this in PROC SQL because we would have to create a data set (containing a mean and standard deviation, for example) and then simulate based on the values in the data set. We can, however, use PROC SQL to perform simple random sampling.

```
proc sql outobs=10;
   select *
      from cereal1
      order by ranuni(8791);
quit;
```

Notes:

- The `ranuni` function is used to randomly order the data set.
- The `outobs` option in the PROC SQL statement allows us to specify the sample size.

It also doesn't make sense to use loops in PROC SQL. We could use a macro, but it makes the most sense to use a datastep for something like a do loop.

Additional items

- To create a variable in a data set that represents the observation number, one can use the `monotonic` function. This is similar to the `_n_` notation that can be used in a datastep.

  ```
  proc sql;
     select monotonic(),
            loc, type, field, field2,
            home, wind, diff_score, change
     from placekick3(obs=10);
  quit;
  ```

  One can also use the monotonic function to select rows/observations in a dataset.

  ```
  proc sql;
     select *
        from cereal1
        where monotonic() in (1,5,12,13,25,31,36);
  quit;
  ```

- One can use the `count` and `nmiss` function to find the number of observations with non-missing or missing values, respectively. The `unique` function and the `distinct` option can be used to find the unique values of a variable.

  ```
  proc sql;
     select count(*) as n 'Total observations',
            count(temp) as n_temp
               'Non-missing temperature',
            nmiss(temp) as nmiss_temp
               'Missing temperature',
            count(speed) as n_speed
               'Non-missing speed',
            nmiss(speed) as nmiss_speed
  ```

```
            'Missing speed',
         count(unique(loc)) as n_locations
            'Unique locations',
         count(distinct loc) as n_loc
            'Distinct locations'
   from placekick1;
quit;
```

Notes:

– Each function counts the number of observations for the variable specified in parentheses.

– The asterisk notation can be used to count the total number of rows/observations in the overall data set.

– Notice that we can provide labels for each new variable without specifying `label=` before the string in quotation marks.

- The put function returns a value using a specified format, and allows the user to condition on user-defined formats. For example, we can create groups in the cereal data set based on sugar content and display only those observations with "low" or "high" sugar content.

```
proc format;
   value sugar_group low - 0.2 = "low"
                     0.201 - 0.4 = "mid"
                     0.401 - high = "high";
run;

proc sql;
   select *, sugar format sugar_group.
      from cereal1
      where put(sugar, sugar_group.)
         in ("low", "high");
quit;
```

Notes:

- PROC FORMAT allows the user to define formats for variables. The procedure doesn't require a data set, but instead defines the format as its own entity.
- The VALUE statement provides the name of the format, `sugar_group`, followed by values or ranges and their corresponding labels or formats.
- We can use the keywords LOW and HIGH to refer to the lowest and highest value of a variable, respectively.
- In PROC SQL, we select all variables from `cereal1` and add a new variable, the sugar content formatted as `sugar_group`.
- In order to refer to the user-defined format, we need to use a period after the format name.
- Using the WHERE statement, we condition on the sugar variable, formatted by `sugar_group`, selecting only those cereals with a low or high sugar content.
- If we wanted to add the newly formatted sugar variable to the `cereal1` data set, we would need to use either a CREATE TABLE statement or an UPDATE statement (see below).

• We can also use PROC SQL to confirm user-defined formats (see program for an example).

• When performing data management steps, we usually create new data sets so that we don't permanently change the original data set. However, sometimes we might want to update or change a data set rather than creating a copy of it. The ALTER TABLE statement adds columns to, drops columns from, or changes column attributes in an existing table. The UPDATE statement modifies values in an existing table.

```
proc sql;
   title2 'Cereal data set';
   create table cereal2 as select * from cereal1;
   select * from cereal2(obs=5);
quit;

proc sql;
   update cereal2
      set fat=fat*
         case when (Cereal contains "Kellog" OR
                    Cereal contains "Post") then 0.95
            else 1.05
         end;
   alter table cereal2
      modify sugar format sugar_group.
         drop sodium;
   title2 'Updated Cereal data set';
   select * from cereal2;
quit;
```

Notes:

- Both the UPDATE and ALTER TABLE statements require the name of the table to be specified, in this case `cereal2`.
- The UPDATE statement tells SAS to set the new `fat` value to be the original value multipled by some percentage. For Kellogg's and Post cereals, the fat content is 95%. For all other cereals, the fat content is 105%.
- The CASE/WHEN clause does not have a variable name because we are updating `fat`, not creating a new variable.
- The MODIFY clause in the ALTER TABLE statement tells SAS to permanently change `sugar` based on the format `sugar_group`. This will remove the numeric values and leave only "low", "mid" or "high" values for sugar content.
- The DROP clause in the ALTER TABLE statement tells SAS to permanently remove the `sodium` variable from the data set.